E PROGRAMADORES, PARA PROGRAMADORES™



PROFESIONAL

Programación Java Server con J2EE Edición 1.3

Subrahmanyam Allamaraju, Cedric Beust, John Davies, Tyler Jewell, Rod Johnson, Andrew Longshaw, Ramesh Nagappan, Dr P. G. Sarang, Alex Toussaint, Sameer Tyagi, Gary Watson, Mark Wilcox, Alan Williamson, Daniel O'Connor

wrox

ANAYA MULTIMEDIA

Sobre los autores

Subrahmanyam Allamaraju

Subrahmanyam Allamaraju es Ingeniero y trabaja desde hace años para *BEA Systems Inc.* Trabaja en las áreas de tecnologías de empresa/distribuidas, modelos de objeto basados en XML y otras áreas relacionadas. Si desea más información sobre sus actividades, inquietudes y otros trabajos, consulte su página Web en www.Subrahmanyam.com.

Subrahmanyam desearía dar las gracias a Varaa por su colaboración en las pruebas de código (enfrentándose a estrictos plazos) y por compartir su frustración y su euforia.

Cedric Beust

Cedric Beust es ingeniero de software y trabaja para el equipo de EJB en *BEA Systems Inc.* Ha participado en la puesta en marcha de la versión 2.0 del contenedor WebLogic EJB. Doctorado en Informática por la Universidad de Niza, Francia. Antes de doctorarse, trabajó para *Sun Microsystems* donde se centró principalmente en CORBA. A lo largo de los años, Cedric ha participado en diversas comisiones dedicadas, por ejemplo, a EJB, CORBA y C++. Sus intereses abarcan desde todo lo relacionado con la informática distribuida y la ingeniería de software en general, hasta aficiones como el golf, el squash, el tenis y el voleibol.

John Davies

John Davies trabaja como Director Jefe de Tecnología (*CTO*) para *Century 24 Solutions Ltd. (C24)*, www.C24Solutions.com, una compañía de software con sede en Londres que ofrece soluciones basadas en Java y J2EE para el mercado financiero. El último producto de C24, Elektra, diseñado para proporcionar una única visión de las transacciones financieras complejas, hace extensivo el uso de todas las tecnologías J2EE. John está embarcado en la actualidad en el proyecto de BNP-Paribas a la cabeza del grupo *Technology Consulting Grup*, recomendando el uso de tecnologías para proyectos actuales y futuros de todo el mundo.

John comenzó en el mundo de las tecnologías de la información a finales de los setenta trabajando en el área de hardware, más tarde en ensambladores, en C, C++ en el año 87 y finalmente en Java desde principios del 96. Trabajó durante catorce años como asesor principalmente en el sector bancario. Diez de estos años los pasó fuera del Reino Unido en diversos países, incluido Estados Unidos, Asia y gran parte de Europa, donde aprendió a hablar algunos idiomas. En el tiempo que le queda libre entre su trabajo y escribir, John imparte clases de Java y XML para *Learning Tree* y es editor técnico de su curso de EJB 2.0. Sus aficiones incluyen viajar, la fotografía, la guitarra clásica, pilotar avionetas, el buen vino, la buena cerveza, la comida picante y socializar. John quisiera dar las gracias a su compañero licenciado en Astronomía Steve Miller por corregir sus trabajos y le gustaría saludar a su madre y pedirle perdón por lo mal que se comportó en el colegio.

Una vez más, nada de esto hubiera sido posible sin el cariño y el apoyo de mi fantástica esposa Rachel, Siento mucho las noches que te ha tocado pasar sola y gracias por Luc y James, os quiero

Puede ponerse en contacto con en John.Davies@C24Solutions.com.

Tyler Jewell

Tyler Jewell trabaja como "predicador" para *BEA Systems Inc.*, donde escribe y alaba sobre las tecnologías de empresa para programadores. Es un experto preparador y mentor, especializado en arquitecturas a gran escala de e-business.

Tyler es el autor de 19 cursos de preparación sobre tecnologías de e-business y ha impartido más de 200 seminarios sobre tecnología a clientes y público en general. Es co-autor del libro *Mastering Enterprise*

JavaBeans 2.0 (O'Reilly 2001) y en la actualidad trabaja en el proyecto *Java Web Services* (O'Reilly 2002). Tyler mantiene una columna habitual sobre J2EE en www.onjava.com, es miembro de la editorial *O'Reilly's Editorial Masthead* y es asesor de la página www.theserverside.com, Tu Comunidad J2EE.

Tyler dedica su tiempo libre a su familia y amigos. Su afición favorita es el póquer *Texas Hold 'Em* en el que dos cartas cualquiera pueden ganar, pero siempre que caigan en sus manos.

Rod Johnson

Rod Johnson es arquitecto de Java de empresa especializado en aplicaciones Web reajustables. Ha pasado los dos últimos años diseñando una solución J2EE para `FT.com`, el portal de empresa más grande Europa y en la actualidad está escribiendo un libro para *Wrox Press* sobre diseño y desarrollo de J2EE.

Después de sus estudios, en los que se especializó en música e informática, Rod realizó un doctorado en musicología antes de volver a dedicarse al desarrollo de software. Rod ha trabajado con Java tanto en áreas de cliente como en áreas de servidor desde su lanzamiento y se ha centrado en el desarrollo de Internet desde 1996. Su principal interés es la arquitectura J2EE, EJB y el desarrollo Web OO.

Rod reparte su tiempo entre Londres y Sydney, y le gusta el tenis, el esquí, la historia y tocar el piano. Puede ponerse en contacto con él en rod.johnson@interface21.com.

Gracias a Kerry por su amor y su apoyo, ahora que comenzamos nuestra nueva aventura.

Andy Longshaw

Andy Longshaw es asesor independiente, escritor y educador especializado en J2EE, XML, tecnologías y componentes de base Web, en particular en las decisiones de diseño y arquitectura necesarias para utilizar estas tecnologías satisfactoriamente. Andy ha estado explicando tecnología durante la mayor parte de la última década en sus funciones principales de Director de Tecnología de *Content Master Ltd.* y *QA Training*. También ofrece conferencias sobre J2EE, XML y arquitecturas de componentes de grada media. Circula un rumor sin confirmar que dice que algunas personas consiguen no dormirse durante estas sesiones. Está dispuesto a responder a cualquier comentario, pregunta o crítica sobre el capítulo de diseño que ha escrito para este libro en la página www.blueskyline.com.

A Sarah, Adam y Joshua que son la inspiración de todo lo que hago, y a mis padres que se aseguraron de que yo recibiera una educación lo suficientemente buena como poder escribir este libro.

Ramesh Nagappan

Ramesh Nagappan es Arquitecto de Tecnología especializado en Java y en arquitecturas de aplicación de distribución de base CORBA. Es un predicador de Java y también un activo contribuyente a las especificaciones e implementaciones de fuente abierta. Antes de engancharse a Java y a CORBA, trabajó como ingeniero de Investigación para desarrollar soluciones CAD/CAM y de dinámica de fluidos computacional para aplicaciones aeroespaciales. En su tiempo libre le gusta disfrutar de los deportes de agua y jugar con hijo Roger. Puede encontrarle en su dirección de correo nramesh@mediaone.net.

Dedicado a mi esposa Joyce y a nuestro hijo Roger, por su cariño, apoyo e inspiración, y también a mis queridos padres por alegrar mi vida.

Dr. P. G. Sarang

Como contratista de *Sun Microsystems*, el Dr. Sarang forma a los clientes de la compañía con diversos cursos del programa oficial de *Sun*. También dirige el programa "Formar a formadores" y las "Pruebas de autorización de instructores" en representación de *Sun*.

Como Director Jefe Ejecutivo (*CEO*) de *ABCOM Information Systems Pvt. Ltd*, el Dr. Sarang está especializado en formación y en el desarrollo de proyectos sobre la plataforma Java/CORBA. Con casi 20 años de experiencia en el sector, el Dr. Sarang ha desarrollado diversos productos y ha completado con éxito varios proyectos industriales. Es orador habitual en muchas conferencias de ámbito nacional e internacional y contribuye regularmente con artículos técnicos a la edición de revistas y diarios de tirada internacional. Sus intereses actuales incluyen la plataforma .NET y C#.

Me gustaría dedicar este libro a mi madre por su constante apoyo y paciencia.

Alex Toussaint

Alex Toussaint es Director de Ingeniería para *Vignette Corporation* en Austin, Texas. Cuenta con más de diez años de experiencia en desarrollo de software y tiene una amplia experiencia en el manejo de Java desde 1996 y de las tecnologías J2EE desde 1998. Alex ha colaborado en artículos sobre aplicaciones Web y comercio electrónico para revistas on-line, tales como *Microsoft MSDN*, y es co-autor del libro *Professional Site Server* publicado por *Wrox*. Alex también ha sido invitado a impartir clases en la Universidad de Texas en *Austin Red McCombs School of Business* sobre temas como Comercio Electrónico y Desarrollo de Software de Empresa. Alex reside en Austin, Texas, en la actualidad con su esposa Danielle y su perro Sasha. Alex está a su disposición en la dirección alex_toussaint@yahoo.com.

Sameer Tyagi

Sameer escribe regularmente para publicaciones on-line e impresas. Cuenta con más de cuatro años de experiencia en diseño y desarrollo de software y está especializado en aplicaciones distribuidas de lado del servidor de base Java (arquitecturas n-Grada, JDBC, JNDI, EJB, JMS, RMI, JSP, Servlets, *et al.*). Es licenciado en Ingeniería Electrónica y tiene una amplia formación sobre el tema.

Es un adicto a Java y consigue su dosis saltando de cabeza a casi cualquier cosa que se compila en códigos de bytes y se le conoce como el culpable de esa nueva área del cerebro llamada *Javaesfera* por dichos estímulos. Cuando no esté disfrutando de un café, puede encontrarle volando a unos 15000 pies de altura en una pequeña *Cessna*.

Gary Watson

Gary Watson ha estado desarrollando el uso de Java durante los últimos 4 años. Es asesor independiente de Tecnologías de la Información y, en la actualidad, trabaja como Arquitecto Técnico en el *Financial Times*, `FT.com`, en el desarrollo de una importante solución J2EE. Graduado por la Universidad de Teeside en 1993, Gary es Licenciado en Ciencias de la Informática.

Siempre que le es posible, Gary disfruta del windsurf, del esquí y de los aviones en miniatura. No participa en estas actividades tan a menudo como le gustaría. Puede encontrarle en Gary@com-contracts.demon.co.uk.

A mi esposa Ángela por su amor, apoyo y aliento mientras luchamos juntos en la vida diaria.

Marc Wilcox

Marc trabaja en el grupo de servicios profesionales de *WebCT Inc.*, la principal compañía del mundo de desarrollo de Sistemas de Gestión/Entornos de Aprendizaje Virtual. La próxima versión de su producto, cuyo nombre en clave es Cobalt, estará basada en un entorno de aplicación J2EE.

Me gustaría dedicar este libro a Doug y a Sammy, los mejores peluqueros del mundo.

Alan Williamson

Alan Williamson es todo un veterano del mundo Java, con un lenguaje que todavía intenta encontrar su lugar en el mundo. Alan cuenta con más de 15 años de experiencia en el mundo del desarrollo de software. Se graduó con Honores en Ciencias de la Informática por la Universidad de Paisley. Alan trabajó principalmente en tareas de investigación y desarrollo hasta que estableció la primera compañía del Reino Unido dedicada únicamente al asesoramiento en Java hace 4 años, especializándose en Java en lado del servidor (http://www.n-ary.com). Alan también ha llegado hasta lo más alto convirtiéndose en Editor Jefe de la revista más importante del mundo sobre Java, *Java Developers Journal* y puede encontrarle dando conferencias por todas partes.

Me gustaría dar las gracias a mi querida Ceri y a mi nuevo hijo Cormac por guardar el fuerte mientras yo trabajaba e investigaba para escribir este capítulo. Buen trabajo. Además, me gustaría dar las gracias a Keith y Marion por realizar sus tareas de profesor y leer mi capítulo corrigiendo mis errores de gramática. Finalmente, me gustaría dar las gracias a Wrox por hacer de este proceso de escritura algo alegre y no algo doloroso.

Índice de contenidos

Introducción 37

J2EE Edición 1.3 ... 37
¿Qué ha cambiado en esta edición del libro? ... 38

¿A quién va dirigido este libro? ... 38

¿Cuáles son los contenidos de este libro? ... 39

Lo necesario para utilizar este libro ... 39
Contenedor Web ... 39
Contenedor EJB ... 39
Bases de datos ... 39
Software adicional ... 40

Convenciones ... 40

Atención al cliente ... 41

Cómo cargar el código de muestra para el libro ... 41

Erratas ... 41

Apoyo vía e-mail ... 41

p2p.wrox.com ... 42
¿Por qué este sistema ofrece el mejor apoyo? ... 42

Capítulo 1. Plataforma J2EE 45

Programación para empresas ... 46

La empresa actual ... 46
¿Es Java la respuesta? ... 49
Independiente de la plataforma ... 49
Objetos gestionados ... 49
Reusabilidad ... 49
Modularidad ... 50

Estilos de arquitectura de empresa 50
Arquitectura de dos niveles 50
Arquitectura de tres niveles 51
Arquitectura de n niveles 52
Arquitectura de empresa 53

La plataforma J2EE 55
Periodo de ejecución de J2EE 55
Los API de J2EE 56

Arquitectura J2EE - Contenedores 58

Arquitectura de contenedor 60
Contratos de componentes 61
API de servicio de contenedor 62
Servicios declarativos 63
Otros servicios de contenedor 65

Tecnologías J2EE 65
Tecnologías de componentes 66
Componentes Web 66
Componentes Enterprise JavaBean 67
XML 68
Tecnologías de servicio 69
JDBC 69
Java Transaction API y Servicio 69
JNDI 69
JMS 70
JavaMail 70
Java Conector Architecture 70
JAAS 70
Tecnologías de comunicación 70
Protocolos de Internet 70
TCP/IP 71
SSL 71
Protocolos de objeto remoto 71

Desarrollo de aplicaciones J2EE 71
Funciones de desarrollo e implementación de aplicaciones 72
Desarrollo de componentes de aplicación 73
Composición de componentes de aplicación en módulos 73
Composición de módulos en aplicaciones 73
Implementación de la aplicación 74

Resumen 75

Capítulo 2. Servicios de directorio y JNDI **77**

Servicios de designación y de directorio 77
Servicios de designación 78
Servicios de directorio 79
LDAP 80
Datos LDAP 81

Introducción de JNDI 83
Las concesiones 85
¿Por qué utilizar JNDI cuando tenemos LDAP? 85
LDAP sin JNDI 86
JNDI sin LDAP 86
¿Y XML? 86

Uso de JNDI 86
Instalación de JNDI 87

Proveedores de servicios JNDI 87
Cómo obtener proveedores de servicios JNDI 88
Cómo desarrollar su propio proveedor de servicios 88

Java y LDAP 89
Control de acceso 89
Autentificación 89
Autorización 90
Servicios de páginas blancas 90
Directorio de procesamiento distribuido 91
Configuración de la aplicación 91

Operaciones LDAP 92
Operaciones LDAP estándar 92
Conexión al servidos LDAP con JNDI 92
Asociación 93
Seguridad simple, SSL/TLS y SASL 94
Simple 94
SSL/TLS 94
SASL 95
Autentificación en LDAP v2 y LDAP v3 95

Búsqueda en un servidor LDAP 95
Filtros LDAP de ejemplo 96
Determinar el alcance de LDAP 96
LDAP_SCOPE_SUBTREE 96
LDAP_SCOPE_ONELEVEL 97
LDAP_SCOPE_BASE 97
Ejecución de una búsqueda JNDI 98
Cómo funciona el programa de búsqueda 100
Búsqueda autentificada 101
Restringir los atributos presentados 103
Añadir entradas 105
Modificar una entrada 111
Eliminar una entrada 113

Almacenar y recuperar objetos Java en LDAP 114
LDAP tradicional 115
Java serializado 116
Referencias Java 116

Volver a JNDI sin LDAP 116
Ejemplo de aplicación DNS 117

Resumen 119

Capítulo 3. Procesamiento distribuido con RMI 121

La arquitectura RMI 122
Capa de Stub y Skeleton 123
Stubs 124
Skeletons 124
Capa de referencia remota 125
Capa de transporte 125

Localización de objetos remotos 126
Archivos de política 128

Excepciones RMI 129

Desarrollo de aplicaciones con RMI 130
Definir la interfaz remota 131
Implementar la interfaz remota 131
Grabar el cliente que utiliza los objetos remotos 133
Generar stubs y skeletons 134
Registrar el objeto 134
Ejecutar el cliente y el servidor 135

RMISecurityManager 135

Pasar parámetros en RMI 136
Parámetros de primitivas 136
Parámetros de objeto 136
Parámetros remotos 137

El recolector de residuos distribuidos 138

Clases de carga dinámica 141

Retrollamadas remotas 146

Activación de objeto 149
El grupo de activación 150
ActivationID 151
Descriptor de activación 152
Convertir objetos en activables 153
Paso 1: Crear la interfaz remota 154
Paso 2: Crear la implementación del objeto 154
Paso 3: Registrar el objeto con el sistema 154
Alternativa a ampliar la clase Activatable 156
Iniciar múltiples JVMs sin recurrir a rmid 157
Desactivación 159

Sockets de adaptación y SSL 160

RMI, cortafuegos y HTTP 172
Tunelado HTTP 172
HTTP a puerto 172
HTTP a CGI 173

El protocolo SOCKS ... 173
Factorías de sockets descargados ... 174

RMI sobre IIOP ... 174
Interoperatividad con CORBA ... 175
Escribir programas con RMI-IIOP ... 176
En el servidor ... 176
HelloServer.java ... 176
En el cliente ... 177
HelloClient.java ... 177

RMI-IIOP y Java IDL ... 180
El archivo IDL ... 180
La implementación de servidor ... 180
La implementación cliente ... 182

RMI-IIOP y J2EE ... 184

Ajustar aplicaciones RMI ... 185

Resumen ... 189

Capítulo 4. Programación de bases de datos con JDBC 191

Drivers de bases de datos ... 193
Tipo 1 – Puente JDBC-ODBC ... 193
Tipo 2 – Parte Java, parte driver nativo ... 194
Tipo 3 – Servidor intermediario de acceso a bases de datos ... 195
Tipo 4 – Drivers Java puro ... 196
Cómo comenzar ... 196

El paquete java.sql ... 197
Gestión de conexión ... 197
Acceso a bases de datos ... 198
Tipos de datos ... 198
Metadatos de base de datos ... 199
Excepciones y advertencias ... 200
Cargar un driver de base de datos y abrir conexiones ... 201
Los URL de JDBC ... 201
DriverManager ... 202
Métodos para gestionar drivers ... 203
Métodos para obtener conexiones ... 203
Métodos de registro ... 204
Driver ... 205
Establecer una conexión ... 205
Crear y ejecutar instrucciones SQL ... 208
Un ejemplo: catálogo de películas ... 209
Crear la tabla Movie ... 210
Insertar datos ... 210
Métodos para el manejo de excepciones ... 211
Consultar la base de datos ... 214
Métodos para recuperar datos ... 214
Interfaz ResultSetMetaData ... 216

Instrucciones preparadas 217
Representar tipos SQL en Java 219
Apoyo a transacciones 221
Puntos de salvaguardia 223
Listas de resultados desplazables y actualizables 225
Listas de resultados desplazables 225
Métodos relacionados con la posición del cursor 227
Métodos para desplazamiento 227
Dirección y tamaño de toma 227
Listas de resultados actualizables 230
Actualizar una fila 230
Eliminar una fila 231
Insertar una fila 231
Actualizaciones de lotes 231

El paquete javax.sql 232

Fuentes de datos JDBC 234
La interfaz javax.sql.DataSource 234
El método getConnection() 234
El método getLoginTimeout() 235
El método setLoginTimeout() 235
El método getLogWriter() 235
El método setLogWriter() 235
JNDI y fuentes de datos 236
Crear una fuente de datos 237
Recuperar un objeto DataSource 238
Características clave 238
Revisión del catálogo de películas 239

Reserva de conexiones 240
Reserva de conexiones tradicional 241
Reserva de conexiones con el paquete javax.sql 243
La interfaz javax.sql.ConnectionPoolDataSource 244
La interfaz javax.sql.PooledConnection 244
La interfaz javax.sql.ConnectionEventListener 245
La clase javax.sql.ConnectionEvent 245
Implementación de reservas de conexión 245

Transacciones distribuidas 246
¿Qué es una transacción? 246
Antecedentes 247
Procesamiento de transacciones – Conceptos 248
Demarcación de transacciones 248
Contexto de transacciones y propagación 249
Alistamiento de recursos 249
Aceptar en dos fases 249
Construcción de bloques de sistemas de procesamiento de transacciones 250
Componentes de aplicación 250
Gestores de recursos 250
Gestor de transacciones 251
Transacciones distribuidas JDBC 251

La interfaz javax.sql.XADataSource ... 252
La interfaz javax.sql.XAConnection ... 252
La interfaz javax.Transaction.UserTransaction ... 252
Pasos para la implementación de transacciones distribuidas ... 253
Configuración ... 254
Iniciar una transacción ... 254
Operaciones de base de datos ... 254
Finalizar una transacción ... 255
Precauciones especiales ... 255

Objetos RowSet ... 256
La interfaz javax.sql.RowSet ... 257
Propiedades ... 257
Eventos ... 258
Ejecución de comandos y resultados ... 258
Tipos de objetos RowSet ... 258
La implementación CachedRowSet ... 259
La implementación RowSet JDBC ... 261
La implementación RowSet Web ... 261

Resumen ... 262

Capítulo 5. Introducción a los contenedores Web 265

El protocolo HTTP ... 266
Métodos de solicitud HTTP ... 266
El método de solicitud GET ... 266
El método de solicitud POST ... 267
El método de solicitud HEAD ... 267
La respuesta HTTP ... 267

Contenedores Web y aplicaciones Web ... 268
Servlets Java ... 269
Páginas JavaServer ... 272
Descriptores de despliegue ... 273
Estructura de aplicaciones Web ... 274
Tipos de contenedores Web ... 274

Una sencilla aplicación Web ... 275
Prepara el contenedor Web ... 275
Crear el archivo HTML ... 275
Crear un servlet ... 276
Construir la aplicación Web ... 277
Desplegar la aplicación Web ... 281
Ejecutar la aplicación Web ... 283
Cómo funciona la aplicación ... 283
El servlet Greeting ... 284
Importar los paquetes de servlet ... 284
Declaración de clase ... 285
Revisar la solicitud POST HTTP ... 285
Extraer parámetros de HttpServletRequest ... 285
Generar respuesta ... 285

El descriptor de despliegue 287

Resumen 288

Capítulo 6. Programación de servlets 291

Análisis del API Java Servlet 292

Implementación de servlet 296
La interfaz servlet 296
El método init() 297
El método service() 297
El método destroy() 297
El método getServletConfig() 298
El método getServletInfo() 298
La clase GenericServlet 298
La interfaz SingleThreadModel 299
La clase HttpServlet 300
Los métodos service() 300
Los métodos doXXX() 301
El método getLastModified() 301

Configuración de servlets 302
La interfaz ServletConfig 302
El método getInitParameter() 303
El método getInitParameterNames() 303
El método getServletContext() 303
El método getServletName() 303
Obtener una referencia a ServletConfig 303
Durante la inicialización del servlet 303
Utilizar el método getServletConfig() 304

Excepciones servlet 304
La clase ServletException 304
La clase UnavailableException 305

El ciclo de vida de los servlets 305

El ciclo de vida de un servlet - FreakServlet 308
Instanciación 314
Inicialización 314
Servicio 315
Destruir 317

Solicitudes y respuestas 318
La interfaz ServletRequest 319
Métodos para solicitar parámetros 319
El método getParameter() 319
El método getParameterValues() 319
El método getParameterNames() 320
El método getParameterMap() 320
Métodos para atributos de solicitud 320
El método getAttribute() 320

El método getAttributesNames() 320
El método setAttribute() 320
El método removeAttribute() 320
Métodos de entrada 321
El método getInputStream() 321
El método getReader() 321
El método getCharacterEncoding() 321
El método setCharacterEncoding() 321
La clase ServletRequestWrapper 321
La interfaz HttpServletRequest 321
Métodos para solicitar ruta y URL 323
El método getPathInfo() 323
El método getPathTranslated() 323
El método getQueryString() 324
El método getRequestURI() 324
El método getRequestURL() 324
El método getServletPath() 324
Métodos para cabeceras HTTP 324
El métodogetHeader() 324
El método getHeaders() 324
El método getHeaderNames() 324
El método getMethod() 324
La clase HttpServletRequestWrapper 325
La interfaz ServletResponse 325
Métodos para tipo de contenido y longitud 325
El método setContentType() 325
El método setContentLength() 325
Métodos de salida 326
El método getOutputStream() 326
El método getWriter() 326
Métodos para salida en búfer 326
El método setBufferSize() 326
El método getBufferSize() 326
El método resetBuffer() 326
El método flushBuffer() 327
El método isCommitted() 327
El método reset() 327
La clase ServletResponseWrapper 327
La interfaz HttpServletResponse 327
Métodos para el manejo de errores 328
El método sendError() 328
El método sendError() 328
El método setStatus() 328
El método sendRedirect() 328
La clase HttpServletResponseWrapper 328
Función de las clases envoltorio 328

Programación de servlets – Aplicación de apoyo técnico 329
Configurar la página HTML 330
Preparar la base de datos 332

Escribir el servlet ... 333
Extraer datos del formulario ... 334
Insertar la solicitud de apoyo técnico ... 334
Generar la respuesta ... 335
Compilar la fuente ... 336
Construir la aplicación Web ... 336
El descriptor de despliegue ... 340
Configurar la fuente de datos ... 340
Desplegar la aplicación Web ... 341
Apoyo técnico en marcha ... 344

Resumen ... 346

Capítulo 7. Sesiones de servlets, contexto y colaboración 349

Protocolo sin estado y sesiones ... 350

Enfoques del registro de sesión ... 352
Reescritura de URL ... 353
Campos de formulario ocultos ... 354
Cookies ... 354

Registro de sesión con el API Java Servlet ... 356
Creación y registro de sesiones ... 357
La interfaz HttpSession ... 357
Métodos para vida de sesión ... 358
El método getCreationTime() ... 359
El método getId() ... 359
El método getLastAccessedTime() ... 359
El método getMaxInactiveInterval() ... 359
El método setMaxInactiveInterval() ... 359
El método isNew() ... 359
El método invalidate() ... 360
Demostrar el ciclo de vida de la sesión con cookies ... 360
Ciclo de vida de sesión sin cookies ... 364
Métodos para gestionar el estado ... 365
El método getAttribute() ... 365
El método getAttributeNames() ... 365
El método setAttribute() ... 365
El método removeAttribute() ... 366
Demostrar la gestión de estado ... 366
Manejo de eventos de ciclo de vida de sesión ... 369
La interfaz HttpSessionListener ... 369
La interfaz HttpSessionActivationListener ... 370
El método sessionDidActivate() ... 370
El método sessionWillPassivate() ... 370
La clase HttpSessionEvent ... 370
El método getSession() ... 370
Manejo de eventos de atributos de sesión ... 371
La interfaz HttpSessionBindingListener ... 371
El método valueBound() ... 371
El método valueUnbound() ... 371

La interfaz HttpSessionAttributeListener 371
El método attributeAdded() 372
El método attributeRemoved() 372
El método attributeReplaced() 372
La clase HttpSessionBindingEvent 372
El método getName() 372
El método getValue() 372
El método getSession() 372
Un sencillo carro de la compra utilizando sesiones 372
El servlet del catálogo 373
El servlet ShoppingCart 374

Contexto de servlet 377
La interfaz ServletContext 378
El método getMimeType() 378
El método getResource() 378
El método getResourceAsStream() 378
El método getRequestDispatcher() 379
El método getNamedDispatcher() 379
El método getRealPath() 379
El método getContext() 379
El método getServerInfo() 379
El método getServletContextName() 379
El método getResourcePaths() 379
Los métodos getInitParameter() y getInitParameterNames() 380
Los métodos getAttribute(), getAttributeNames(), setAttributes() y removeAttribute() 380
Los métodos log() 380
El manejo de eventos de ciclo de vida de ServletContext 380
La interfaz ServletContextListener 380
ServletContextAttributeListener 381
Una aplicación de chat que utiliza contexto y sesiones 381
La clase ChatRoom 384
La clase ChatEntry 384
El servlet de administración 385
Servlets para chat 388
La clase ListRoomsServlet 388
La clase ChatRoomServlet 391
Configuración del chat 396
Una página de bienvenida 396
Descriptor de despliegue 396
Desplegar y probar 397

Colaboración de servlet 399
Encadenado de servlet 399
Lanzamiento de solicitudes 400
La interfaz RequestDispatcher 400
El método forward() 401
El método include() 401
Obtener un objeto RequestDispatcher 401
Revisión de Apoyo Técnico 401
La página techsupp.html 402

TechSupportServlet ... 404
La página register.html ... 406
RegisterCustomerServlet ... 407
ResponseServlet ... 408
BannerServlet ... 409
Configuración de Apoyo Técnico y despliegue ... 411
Utilizar RequestDispatcher para colaboración ... 412
Reenviar una solicitud a ResponseServlet ... 412
Reenviar solicitud a regiter.html ... 412
Incluir insignia en ResponseServlet ... 413

Resumen ... 413

Capítulo 8. Filtros para aplicaciones Web 415

¿Qué es un filtro? ... 416

Filtro de muestra ... 418
Despliegue ... 421

El API filtro ... 422
La interfaz de filtro ... 422
El método init() ... 423
El método doFilter() ... 423
El método destroy() ... 423
La interfaz FilterConfig ... 424
El método getFilterName() ... 424
El método getInitParameter() ... 424
El método getInitParameterNames() ... 424
El método getServletContext() ... 424
La interfaz FilterChain ... 424
El método doFilter() ... 425

Descriptor de despliegue para filtros ... 425

La aplicación de chat con filtros ... 427
Registro de mensajes ... 428
Moderación de mensajes ... 430
Descriptor de despliegue ... 434
Despliegue ... 436
Chat con registro y moderación ... 437

Resumen ... 438

Capítulo 9. Despliegue Web, autentificación y empaquetado 441

Estructura de aplicación Web ... 441
Estructura de directorio ... 442

Ficheros en archivo Web ... 444
Desplegar un archivo WAR ... 445
¿Cuándo deben utilizarse los archivos WAR? ... 446

Asociar de solicitudes a aplicaciones y servlets ... 446

Asegurar aplicaciones Web 451
Seguridad programada 454
El método getAuthtype() 454
El método getRemoteUser() 454
El método getUserPrincipal() 454
El método isUserRole() 454
Autentificación basada en un formulario 454
Configuración de despliegue 457
Parámetros de inicialización de contexto 458
Parámetros de inicialización de servlet 459
Cargar servlets en el arranque 459
Tiempo límite de sesión 460
Asociaciones MIME 460
Archivos de bienvenida 461
Páginas de error 462
Enviar errores 463
Lanzar ServletException 463
Manejar errores y excepciones HTTP 464
Aplicaciones distribuibles 465
Resumen 468

Capítulo 10. Fundamentos y arquitectura de JSP 471

La especificación JSP 1.2 471
Presentación de JSP 472
Los aspectos prácticos 476
Directrices JSP 477
La directriz page 477
Ejemplo 479
La directriz include 479
Ejemplo 480
La directriz taglib 482
Elementos de directivas 482
Declaraciones 483
Ejemplo 483
Scriptlets 484
Ejemplo 484
Expresiones 486
Ejemplo 487
Comentarios 487
Acciones estándar 488
<jsp:useBean> 488
<jsp:setProperty> 490
<jsp:getProperty> 492
<jsp:param> 495
<jsp:include> 495
<jsp:forward> 500
<jsp:plugin> 502
Objetos implícitos 505

El objeto Request ... 505
El objeto Response ... 505
El objeto PageContext ... 505
El objeto Session ... 505
El objeto Application ... 506
El objeto Out ... 506
El objeto Config ... 506
El objeto Page ... 506
Alcance ... 506
Alcance de página ... 507
Alcance de solicitud ... 507
Alcance de sesión ... 507
Alcance de aplicación ... 507
Páginas JSP como documentos XML ... 507
Directrices ... 508
Elementos de directivas ... 508
Acciones ... 509
Una página de ejemplo ... 509
Sintaxis JSP ... 509
Sintáxis XML ... 509

Apoyo técnico JSP ... 509
Diseño de aplicación ... 511
La página de bienvenida ... 511
La página JSP de procesamiento de solicitudes ... 512
La clase JDBCHelper ... 513
TechSupportBean ... 514
Formulario de registro ... 516
JSP de registro ... 517
Las páginas JSP de respuesta y de anuncio ... 517
La página de error ... 518
Desplegar la aplicación ... 519

Estrategias de diseño JSP ... 520
Diseños centrados en la página o cliente-servidor ... 521
Vista de página ... 521
Vista de página con bean ... 522
El patrón de controlador frontal ... 523
Implementación de una arquitectura de controlador frontal ... 524
Servlet controlador ... 524
Manejadores de solicitud ... 525
Beans de página ... 525
Vistas JSP ... 525
Implementación ... 526
Ventajas ... 534
Utilizar un sistema de controlador genérico ... 535

Resumen ... 535

Capítulo 11. Extensiones de etiqueta de JSP 539

Extensiones de etiqueta ... 539

Mejoras de la extensión de etiqueta JSP 1.2 ... 542

Una sencilla etiqueta ... 542

Anatomía de una extensión de etiquetas ... 546
Manejadores de etiqueta ... 547
La interfaz javax.servlet.jsp.tagext.Tag ... 547
La interfaz javax.servlet.jsp.tagext.IterationTag ... 550
La interfaz javax.servlet.jsp.tagextBodyTag ... 550
La clase javax.servlet.jsp.tagextBodyContent ... 552
Clases de conveniencia ... 553
Objetos de disposición de los manejadores de etiqueta ... 554
Revisión del sencillo ejemplo ... 555
Descriptores de biblioteca de etiqueta ... 556
Utilizar extensiones de etiqueta en páginas JSP ... 558
Despliegue y empaquetado de bibliotecas de etiquetas ... 559
Asociaciones en Web.xml ... 560
Paquete JAR de biblioteca de etiquetas ... 561
Asociación por defecto ... 561
Combinaciones de asociación ... 561

Escribir extensiones de etiqueta ... 561
Atributos de procesamiento ... 561
Contenido de cuerpo ... 565
Variables de directivas introducidas por etiquetas ... 567
Cambios en el manejador de etiqueta ... 568
Ejemplo de variables de directiva ... 568
Definición programática de variables de directiva ... 571
Iteración y manipulación de contenido de cuerpo ... 573
Evaluación repetida con la interfaz IterationTag ... 573
Etiquetas de cuerpo que filtran su contenido ... 576
Anidamiento de etiqueta ... 578
Validar el uso de extensiones de etiqueta en páginas JSP ... 583
Manejo de errores ... 584
TryCatchFinally: ¿un peligro para el diseño? ... 590

Eventos de ciclo de vida de aplicación ... 590

Locuciones de extensiones de etiqueta ... 595

Resumen ... 597

Capítulo 12. Escribir aplicaciones JSP con bibliotecas de etiquetas 599

Ventajas de utilizar bibliotecas de etiquetas personalizadas ... 600

Ejemplos de bibliotecas de etiquetas existentes ... 601

Introducción de la Biblioteca de Etiquetas Estándar JSP (JSPTL) ... 602
Obtener JSPTL ... 602
¿Qué abarca JSPTL? ... 602

Iniciarse en JSPTL ... 603

Integrar JSPTL en sus páginas JSP 604
Un ejemplo de iteración 604
Ejecución 605

Etiquetas JSPTL 608

Algunas consideraciones de diseño de JSPTL 609
Algunas etiquetas básicas 609
Etiquetas de flujo de control 610
Etiquetas de iteración 612
Estado de iteración 618
Capacidad de ampliación de la etiqueta de iteración 618
Etiquetas condicionales 619

Una aplicación de registro y autentificación 622
La página de bienvenida 623
El servlet controlador de registro 624
La página de formulario de registro 628
Etiquetas para el manejo de formularios HTML 632
Gracias por registrarse 636
Internacionalización de nuestro código JSP 637
¿Qué es un lote de recursos? 637
Cómo se localizan los lotes de recursos 638
Aplicar lotes de recursos a aplicaciones JSP 639
La página de error de registro 643
La página de formulario de entrada 644
Ver su perfil de usuario 645
Algunas ideas fundamentales sobre etiquetas en JSP 648
Página de error de entrada 651
Ver el sitio Web favorito del usuario 652
Server-Side Includes (SSI) 652
Algunos temas relacionados con Server-Side Includes 653
Una etiqueta Server-Side Include 654
Administración: la página Mostrar-todos-los-usuarios 657
Administración: un servicio Web de base XML 659

Desplegar la aplicación de registro 662

Lo que está por llegar: la biblioteca de etiquetas estándar 665

Resumen 665

Capítulo 13. JavaMail **669**

Protocolos de correo 670
SMTP 670
POP3 671
IMAP 671
MIME 672

Análisis de JavaMail 672

Instalación y configuración 673

¡Rápido, envíame un e-mail! 674

API de JavaMail 675
javax.mail.Session 675
javax.mail.Authenticator 677
javax.mail.Message 678
javax.mail.internet.MimeMessage 679
De 680
Para, CC y CCO 680
Responder a 681
Asunto 681
Fecha 682
ID del mensaje 682
Otros... 682
javax.mail.Part 682
Sistema de activación de JavaBeans 683
javax.mail.Multipart 685
javax.mail.BodyPart 686
Contenido del mensaje 686
Indicadores de mensaje 688
javax.mail.Address 689
javax.mail.internet.InternetAddress 690
javax.mail.internet.NewsAddress 691
javax.mail.Store 691
Acceso a las carpetas 692
javax.mail.URLName 694
javax.mail.Folder 695
Abrir y cerrar carpetas 695
Mensajes de listado 696
Recuperación avanzada de mensajes 697
Copiar y mover mensajes 698
Buscar mensajes 699
Javax.mail.Transport 700

Trabajar con correo 701
Enviar correo 701
Enviar documentos adjuntos 703
Leer correo 706
Eliminar correo 710
Recibir archivos adjuntos 711
Guardar y cargar correo 712

Recursos JavaMail 716

Resumen 717

Capítulo 14. Arquitectura y diseño EJB **719**

¿Qué son los EJB? 720
Enterprise JavaBeans versus JavaBeans 721
Variedades de beans 722
¿Por qué utilizar EJB? 723

El contenedor EJB y sus servicios 724
Persistencia 725
Transacciones declarativas 726
Seguridad declarativa 726
Manejo de errores 726
Sistema de componentes para lógica de empresa 726
Reajustabilidad 727
Portabilidad 727
Gestionabilidad 727
Cómo proporciona servicios el contenedor 727
Contratos 728
Servicios 729
Interposición 730

Trabajar con EJB 731
La vista del desarrollador cliente 732
La vista del proveedor de bean 734
¿Qué no puede hacer un EJB? 745
API responsable de los hilos 746
AWT 747
Funcionar como un servidor de red 747
Escribir en campos estáticos 747
El paquete java.io 747
Cargar una biblioteca nativa 747
Utilizar "this" como argumento o valor de retorno 748
Retrollamadas 748

Componentes EJB en la Web 748

Acceso de nivel cliente a EJB 751

Diseño del nivel EJB 752
Ejemplos de uso de UML 752
Objetos de análisis 753
Objetos interfaz 753
Objetos control 753
Objetos entidad 753
Análisis versus implementación 754
La función de estado en el objeto interfaz 756
Un ejemplo de diseño EJB 757
Crear un producto 759
Realizar un pedido 759
Cancelar un pedido 759
Seleccionar un pedido para su manufactura 759
Elaborar un producto 759
Enviar un pedido 759
Registrar pedidos atrasados 760

Resumen 766

Capítulo 15. Beans de sesión y lógica de empresa **769**

Beans de sesión y estado 770

Representación de lógica de empresa ... 770
Beans de sesión como fachada ... 771
El problema del estado conversacional ... 773
Beans de sesión y almacenamiento persistente ... 775

Bean calculadora de asistencia financiera ... 776
Bean calculadora de necesidad financiera sin estado ... 777
La interfaz remota de calculadora de necesidad financiera ... 777
El cliente de necesidad financiera ... 777
La interfaz inicial calculadora de necesidad financiera ... 779
La clase de implementación de la calculadora de necesidad financiera ... 779
public void ejbCreate() ... 780
public void ejbActivate() ... 781
public void ejbPassivate() ... 781
public void ejbRemove() ... 781
public void setSessionContext(SessionContext ctx) ... 781
El descriptor de implementación ... 783
Bean calculadores de necesidad financiera con estado ... 784
La interfaz remota de necesidad financiera con estado ... 784
La interfaz inicial calculadora de necesidad financiera con estado ... 785
La clase de implementación de la siguiente calculadora financiera con estado ... 785
El cliente para la calculadora de la necesidad financiera con estado ... 789
Descriptores de implementación ... 789
Combinar beans con estado y sin estado ... 792
Las nuevas interfaces y cliente de la calculadora de necesidad con estado ... 792
La nueva clase de implementación de la calculadora de necesidad con estado ... 792
Los descriptores de implementación ... 795

Implementación de nuestra aplicación de fabricación ... 796
Clientes e interfaces de lógica de empresa ... 798
El EJB ManageOrders ... 799
El EJB de fabricación ... 802
Los clientes de aplicación ... 803
El cliente BeginManufacture ... 805
El cliente CompleteManufactured ... 806
El cliente PlaceSampleOrders ... 808
El cliente ManagerSampleOrders ... 809
El cliente CreateProducts ... 810
Implementación del bean de sesión sin estado ... 811
Métodos de empresa ... 811
Métodos de ayuda de implementación ... 818
Métodos de ciclo de vida y de sistema ... 820
Implementación del bean de sesión con estado ... 820

Resumen ... 826

Capítulo 16. Beans de entidad y persistencia 829

¿Por qué no utilizar beans de sesión? ... 830
Utilizar un bean de sesión con estado ... 830
Utilizar un bean de sesión sin estado ... 833
Ventajas de los beans de entidad ... 834

Persistencia gestionada por contenedor versus persistencia gestionada por bean **835**
Nuevas funciones para CMP introducidas en EJB 2.0 837
Accesores abstractos 838
Interfaces locales 839
Relaciones 840
Lenguaje de consulta EJB 840
La cláusula SELECT 841
La cláusula FROM 841
La cláusula WHERE 842
El laboratorio SportBean 843
Claves primarias 843
Las retrollamadas C.R.U.D. 846
Crear 846
Los métodos create() de Sports Team, interfaz inicial 846
La clase de implementación de Sports Team CMP 848
La clase de implementación BMP de SportsTeam 849
Leer 851
Crear la tabla 851
Beans de entidad CMP y ejbLoad() 851
Beans de entidad BMP y ejbLoad() 852
Actualizar 853
CMP y el método ejbStore() 853
BMP y el método ejbStore() 854
Eliminar 854
CMP y el método remove() 854
BMP y el método remove() 855
Retrollamadas BMP versus retrollamadas CMP 855
El descriptor de implementación 856
Persistencia en los descriptores de implementación 857
Almacenamiento en memoria caché 858
Métodos buscadores 861
Implementación de métodos buscadores 861
CMP y métodos buscadores 862
BMP y métodos buscador 863
Activación y pasivación 864
El ciclo de vida completo 865
Reentrada 866
Completar el ejemplo del equipo de deporte 867
El cliente CMP 868
El cliente BMP 869
Configurar WebLogic 6.1 para nuestros EJB 872

Relaciones **873**
Crear interfaces locales 873
Definir las relaciones 874

Completar nuestra aplicación de fabricación **875**
El bean Pedido 876
La interfaz inicial del bean Pedido 876
La interfaz remota del bean Pedido 877
La clase de implementación del bean Pedido 877

La clase de clave primaria del bean Pedido ... 880
La excepción OrderNotCancelableException ... 881
El bean Producto ... 881
La interfaz inicial del bean Producto ... 882
La interfaz inicial local del bean Producto ... 882
La interfaz remota del bean Producto ... 883
La clase de implementación del bean Producto ... 883
La clase de excepción NoSuchRoutingInstruction ... 885
El descriptor de despliegue completo ... 885
Ejecutar la aplicación de fabricación ... 892

Resumen ... 894

Capítulo 17. Servicios de contenedor EJB 897

Transacciones ... 898
Transacciones sin contenedor ... 900
Semántica declarativa para transacciones ... 907
El atributo transaccional NotSupported ... 907
El atributo transaccional RequiresNew ... 908
El atributo transaccional Required ... 908
El atributo transaccional Supports ... 908
El atributo transaccional Mandatory ... 908
El atributo transaccional Never ... 908
Especificar atributos transaccionales ... 908
Elegir atributos de transacción ... 909
Transacciones controladas por el usuario ... 911
Niveles de aislamiento ... 913
Transacciones largas ... 915
Bloqueo optimista ... 916
Bloqueo pesimista ... 917
Realización en dos fases ... 917

Seguridad ... 917
Especificar los requisitos de seguridad ... 919
Roles de seguridad ... 919
Permisos de método ... 921
Control programado de acceso ... 922
Seguridad y diseño de aplicación ... 924

Excepciones ... 925
Excepciones de aplicación ... 926
Excepciones de aplicación predefinidas ... 929
Excepciones de sistema ... 929

Comunicación ... 932
Comunicación entre servidores heterogéneos ... 932
Llamadas de método en VM ... 934

Resumen ... 934

Capítulo 18. Roles de desarrollo y de implementación 937

El proveedor de bean de empresa ... 938

<resource-ref> ... 943
<ejb-ref> ... 943
<ejb-local-ref> ... 944

El ensamblador de la aplicación ... 952

El implementador ... 963

El administrador del sistema ... 973

Vendedor de contenedor/servidor de aplicación ... 974

Una interfaz Web para la aplicación de fabricación ... 975

Consejos para resolver problemas ... 999

Resumen ... 1000

Capítulo 19. JMS y beans controlados por mensaje 1003

Breve historia del mensaje ... 1004

Servicio de Mensajes Java (JMS) ... 1005
Punto-a-punto ... 1006
Publicar/suscribir ... 1007
La arquitectura JMS ... 1008
Ejemplo de cola punto-a-punto ... 1010
Producir un mensaje ... 1010
Consumir un mensaje sincrónicamente ... 1013
Consumir un mensaje asincrónicamente ... 1015
Ejemplo de apartado publicar/suscribir ... 1018

El API JMS ... 1025
Utilizar transacciones con JMS ... 1028
Tres implementaciones JMS de peso industrial ... 1029
SwiftMQ ... 1029
SpiritWave ... 1030
SonicMQ ... 1030

Beans controlados por mensaje ... 1031
Ciclo de vida del bean controlado por mensaje ... 1032
Transacciones en beans controlados por mensaje ... 1032
Transacciones gestionadas por bean ... 1033
Transacciones gestionadas por contenedor ... 1033
Un ejemplo de bean controlado por mensaje ... 1033
Implementación del bean controlado por mensaje ... 1034

Resumen ... 1041

Capítulo 20. La arquitectura de conectores J2EE 1045

Integración de EIS y el rol de JCA ... 1046

La Arquitectura de Conectores J2EE y sus elementos ... 1048
Comparación de JCA con JDBC ... 1049
El adaptador de recursos y sus contratos ... 1050

Contratos de aplicación ... 1050
Contratos de nivel de sistema ... 1051
Gestión de conexiones ... 1051
Gestión de transacciones ... 1051
Gestión de seguridad ... 1052
Acceso a EIS ... 1052
Empaquetado e implementación de un adaptador de recursos ... 1053
Empaquetar un adaptador de recursos ... 1054
El descriptor de implementación del adaptador de recursos (ra.xml) ... 1055
Roles y responsabilidades de la implementación ... 1058
Proveedor del adaptador de recursos ... 1058
Vendedor del servidor de aplicación ... 1058
Proveedor del componente de aplicación ... 1058
Implementador ... 1059
Opciones de implementación ... 1059
Implementar un adaptador de recursos como una aplicación J2EE ... 1059

Adaptadores de recursos de caja negra ... 1060
Utilizar un adaptador de caja negra. El ejemplo DemoConnector ... 1060
Seleccionar un adaptador de caja negra ... 1060
Transacciones XA y no-XA que utilizan un adaptador de caja negra ... 1061
Implementación del adaptador de recursos de caja negra ... 1061
Configurar la base de datos y la tabla Demoaccount ... 1061
Implementación del adaptador de recursos ... 1062
Poner a prueba el adaptador de recursos de caja negra ... 1067
La interfaz inicial ... 1067
La interfaz remota ... 1068
La clase de implementación del bean ... 1068
La clase ProcessingExceptions ... 1073
El descriptor de implementación EJB ... 1073
Implementación del bean ... 1073
El cliente ... 1074
La Interfaz Común de Cliente (CCI) ... 1076
Interfaces y clases CCI ... 1076
Conexión con una aplicación gestionada (J2EE) ... 1079
Conexión a una aplicación no gestionada (de dos niveles) ... 1080
Utilizar una CCI con adaptador de caja negra ... 1082
Implementación de un bean de sesión con CCI ... 1082
La interfaz inicial ... 1083
La interfaz remota ... 1083
La clase de implementación del bean ... 1083
El descriptor de implementación EJB ... 1086
El cliente ... 1087
Implementación del adaptador de recursos de caja negra de CCI ... 1087
Configurar la base de datos ... 1088
Implementación del adaptador ... 1089
Implementación y prueba de la aplicación CCI ... 1091

Ventajas de la arquitectura de conectores J2EE ... 1092
Integración de aplicación de empresa (EAI) con JCA ... 1092
Portales de empresa capacitados con JCA ... 1092

Integración empresa-a-empresa con JCA 1093
Elementos que faltan en JCA 1.0 1093
Resumen 1094

Capítulo 21. Consideraciones de diseño para aplicaciones J2EE **1097**

El mundo sigue cambiando 1098
Arquitectura y diseño 1099
Estilos de arquitectura 1099
Contexto de diseño 1101
Los requisitos de empresa 1102
El sistema existente 1102
El sistema deseado 1103
El contexto de empresa 1104
Elaborar los requisitos 1105
Construir el modelo 1105
Exploración el modelo 1106
Elaborar el contexto 1106
Adaptarse al terreno 1107
Diseño distribuido 1107
Elegir y perfeccionar una arquitectura 1109
Arquitectura para el sistema de pedidos de compra 1109
Iteración y retroalimentación 1110
Aplicar patrones 1111
¿Qué son los patrones? 1111
Patrones J2EE 1112
Patrón de controlador frontal 1114
Patrón de vista de composición 1115
Patrón de fachada de sesión 1117
Patrón de localizador de servicios 1118
Patrón de Objeto valor 1119
Patrón de Delegado de empresa 1121
Patrón de Objeto de acceso a datos 1122
Patrón de Ayudante de vista 1123
Patrón de Vista de lanzador 1124
Patrón de Servicio al trabajador 1125
Patrón de Manejador de lista de valores (Iterador página-a-página) 1125
Patrón de Lector de vía rápida 1127
Comenzar por el principio 1128
Presentación de datos del producto al usuario 1128
Abstraer el acceso a datos 1129
Distinguir entre funcionalidad y presentación 1130
Separar interacción de usuario, presentación y datos 1131
Evolución del sistema 1133

Añadir el nivel medio (Middle-Tier) 1135

De compras 1137
Encapsular el flujo de trabajo de pedidos 1138
Modelos con estado versus modelos sin estado 1139
Presentar el pedido 1141
Elegir un mecanismo asíncrono 1141
Reajustabilidad y disponibilidad a través de la asincronicidad 1143
Cuestiones relacionadas con sistemas asíncronos 1143
Aprobar el pedido 1144

Más allá de sistema de pedidos de compra 1146
Diseño de interfaz EJB 1146
Eventos distribuidos 1148
Diseño para bases de datos 1148

Lecciones aprendidas 1149
Separar intereses siempre que sea posible 1149
Minimizar el tráfico de la red 1149
Utilizar abstracción para contribuir a la flexibilidad 1150
Usar patrones comunes 1150
Reducir el acoplamiento de mecanismos asíncronos 1151
Transacciones de plan 1151

Resumen 1151

Capítulo 22. J2EE y servicios Web **1153**

¿Qué son los servicios Web? 1154
Servicios inteligentes 1155

Tecnologías de servicios Web 1155
SOAP 1155
Interoperabilidad 1157
Implementaciones 1157
Mensajes SOAP con anexos (SwA) 1157
WSDL 1158
Documento WSDL 1158
El elemento types 1159
El elemento message 1160
El elemento portType 1160
El elemento binding 1160
El elemento port 1161
El elemento service 1161
UDDI 1162
Implementación 1162
ebXML 1162

Tecnologías J2EE para servicios Web 1163
JAXP 1164
JAXB 1164
Esquema XML 1164
Asociación de datos XML 1165

JAXM 1166
JAX-RPC 1168
JAXR 1169
Arquitectura 1169
Objetivos de diseño 1170

Desarrollo de servicios Web 1170
Arquitectura de servicio Web 1171
Localizar servicios Web 1171
La interfaz del servicio Web 1172
Implementar la lógica de empresa 1172
Integrar otros recursos 1172
Devolver resultados a clientes 1172
Un sencillo servicio Web 1173
Desarrollar el archivo Java StockQuote 1173
Crear el JAR 1174
Generar archivos WSDL 1174
El archivo StockQuote_Service.wsdl 1177
El archivo StockQuote_Service-interface.wsdl 1177
Generar la clase proxy 1179
Registrar el servicio en el servidor Web 1179
Desarrollar un cliente 1182

Crear servicios más inteligentes 1183
Contexto compartido 1183
Servicios Web inteligentes 1184
La iniciativa Sun ONE 1184
Análisis de la arquitectura funcional 1185
Arquitectura de servicios Web inteligentes 1186
Provisión inteligente 1187
Gestión inteligente 1187
Proceso inteligente 1187
Política inteligente 1187

Apoyo del vendedor a servicios Web 1187
Servicios Web sencillos 1187
Servicios Web complejos 1188

Resumen 1188

Capítulo 23. Elegir una implementación J2EE 1191

Servidores de aplicación 1191

Implementación de las especificaciones J2EE 1194

Competencia en el mercado de servidores de aplicación 1196
Implementaciones J2EE 1197
Funciones de valor añadido 1198
Implementación actual de normas futuras 1200
Posibles caminos potenciales para funciones de valor añadido 1201

Parámetros de evaluación 1202

Comunidad de desarrollo 1205
Resumen 1206

Capítulo 24. Empaquetado e implementación J2EE 1209

Análisis del empaquetado J2EE 1210
¿Qué elementos pueden ser empaquetados? 1210
Roles de empaquetado 1213
Las limitaciones del empaquetado 1214
Comprender los esquemas de carga de clases 1214
La opción anterior a EJB 2.0 1215
La opción posterior a EJB 2.0 1217
Una ambigüedad en la especificación J2EE 1218

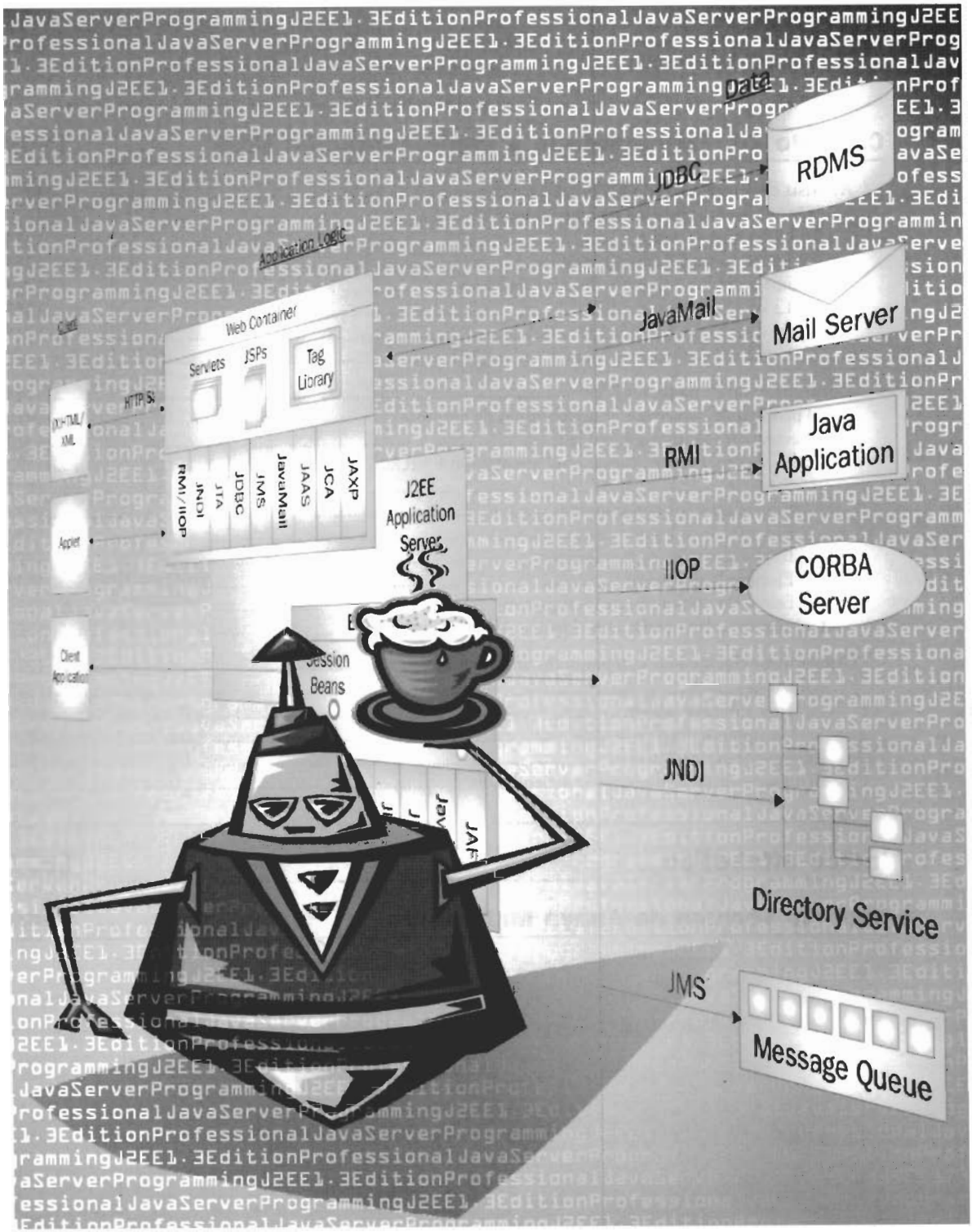
Configuración de paquetes J2EE 1218
El proceso de desarrollo de aplicaciones de empresa 1218
La estructura de un paquete J2EE 1220
Trabajar con el descriptor de implementación EAR 1222
Ejemplo 1223
Empaquetar los componentes 1223
Ensamblaje de la aplicación 1224
Implementación de la aplicación 1225
Ejecutar la aplicación 1225
Etiquetas opcionales de la etiqueta de implementación 1226
Cuestiones relacionadas con el orden de los módulos 1227

Cuestiones relacionadas con paquetes de dependencia 1227
Soluciones 1228
Una solución mejor 1229
Comprender la ruta de clase manifiesta 1229
Ejemplo de dependencia 1230
El impacto de las bibliotecas de dependencia 1233

Resumen 1234

Índice alfabético 1237

Soporte Técnico de Anaya Multimedia 1246


Data
RDMS
JDBC
Application Logic
Web Container
Servlets
JSPs
Tag Library
Client
HTTP/S
(X)HTML/ XML
Applet
Client Application
RMI/IIOP
JNDI
JTA
JDBC
JMS
JavaMail
JAAS
JCA
JAXP
J2EE Application Server
Session Beans
JavaMail
Mail Server
RMI
Java Application
IIOP
CORBA Server
JNDI
Directory Service
JMS
Message Queue

Introducción

Bienvenido a la tercera edición de *Programación para Java Server Profesional*. A diferencia del cambio entre la primera y la segunda edición, las diferencias de la versión 1.3 de J2EE son relativamente menores. Las diferencias fundamentales radican en que los capítulos sobre servlets, JSP y EJB han sido actualizados para reflejar los cambios realizados en las especificaciones relevantes, por ejemplo en EJB 2.0. Además, se han incluido nuevos capítulo para las nuevas características de J2EE tales como la Arquitectura de Conectores y algunos capítulos menos relevantes para el desarrollo básico de J2EE han sido eliminados.

J2EE Edición 1.3

La última entrega de JRS-58, más conocida como **Plataforma Java 2, Enterprise Edition (J2EE)**, representa la evolución de la plataforma de desarrollo del lado del servidor de Sun Microsystems hacia una especificación más madura y sofisticada. Además de la inclusión de nuevas sub-especificaciones como JAAS (Java Authentication and Authorization Service) y la Arquitectura de Conectores, la verdadera naturaleza centrada en el contenedor de la especificación de J2EE no ha sufrido cambios significativos.

Los cambios más notables de esta entrega están relacionados con las modificaciones realizadas a las sub-especificaciones, en especial a Servlet, JavaServer Pages (JSP) y Enterprise JavaBeans (EJB). Los servlets obtienen eventos y filtrado; JSP obtiene una nueva sintaxis XML y mejoras en los mecanismos de ajuste de designador; y EJB cuenta con algunos cambios significativos en su modelo de persistencia gestionado por su contenedor.

Las versiones actuales de API en J2EE 1.3 se encuentran en:

- Servlets 2.3

- JavaServer Pages 1.2
- Enterprise JavaBeans 2.0
- JDBC 2.0 Extension
- Java Message Service 1.0
- Java Transaction API 1.0
- JavaMail 1.2
- Java Activation Framework 1.0
- Java API para XML Processing 1.1
- Java Conector Architecture 1.0
- Java Authentication and Authorization Service 1.0

A lo largo de este libro examinaremos todas estas versiones.

¿Qué ha cambiado en esta edición del libro?

Observará que una parte del material de esta edición J2EE 1.3 es muy similar al contenido de su predecesora, la edición J2EE. Esto se debe a que antes de remendar notablemente el material esencialmente bueno y sólido del libro, hemos decidido simplemente actualizar material relevante para seguir siendo consecuentes con las últimas especificaciones identificadas por J2EE. En algunos casos, esto ha supuesto la inclusión de un capítulo adicional, pero en otros sólo encontrará pequeños cambios.

Con todo ello, hay algunas diferencias significativas en lo que se refiere a capítulos entre esta edición y la anterior. La edición J2EE 1.3 contiene capítulos sobre Arquitectura de Conectores Java, Servicios Web, cómo elegir una implementación J2EE y Paquete de aplicaciones J2EE.

Lo que ha sido eliminado son los capítulos más ajenos que no eran esenciales para el aprendizaje sobre el desarrollo de aplicaciones J2EE, tales como la Internacionalización.

¿A quién va dirigido este libro?

Este libro está destinado a programadores profesionales de Java que, aunque no tengan mucha experiencia práctica, estén como mínimo familiarizados con los conceptos fundamentales de programación Web y de redes. También presupone estar familiarizado con el lenguaje Java y los API, a través de la lectura de *Beginning Java 2*, o de algún otro libro práctico que abarque un área similar. Sin embargo, todos los conceptos relacionados con la programación Java de lado servidor serán abordados sin presuponer conocimientos previos.

No obstante, se recomienda cierta familiaridad con las tecnologías básicas Java de lado servidor, ya que este libro abarca una amplia área de conocimientos rápidamente y no pretende ser exhaustivo en todos los campos.

> **Además, los que ya posean la edición J2EE anterior del libro puede que encuentren que esta edición no les aporta muchos conocimientos ya que gran parte del material esencial es muy similar a la edición anterior.**

¿Cuáles son los contenidos de este libro?

En este libro, examinamos tres temas principales:

- Las normas de las especificaciones de tecnología que deben seguir los creadores de programas para crear componentes de empresa.
- Los límites y beneficios de las implementaciones típicas reales de las compañías distribuidoras de las especificaciones J2EE.
- Los aspectos prácticos resultantes de los diseños reales que utilizan las tecnologías J2EE.

El libro tiene la siguiente estructura básica:

- Comenzaremos revisando las últimas demandas planteadas a un agente de desarrollo de empresa de Java y analizando cómo Java (y en particular J2EE) avanza para alcanzar estos retos. También tendrá su primer contacto con la arquitectura de contenedor J2EE.
- Después de que nos hayamos introducido a toda velocidad en la arquitectura J2EE, comenzaremos a analizar algunas de las tecnologías fundamentales en el desarrollo de una empresa: RMI, JDBC y JNDI.
- Más tarde, volveremos a J2EE de un modo más explícito considerando cómo desarrollar componentes Web utilizando los servlets de Java.
- Una vez hayamos comprendido la tecnología de servlets, estudiaremos cómo se incorpora a JavaServer Pages y se amplia para proporcionar un medio más flexible para crear contenido Web dinámico.
- Finalmente, analizaremos algunas temas extensos de J2EE tales como consideraciones de diseño y cómo crear paquetes con sus aplicaciones J2EE.

Lo necesario para utilizar este libro

La mayor parte del código de este libro ha sido probado con la Plataforma Java 2, Standard Edition SDK (JDK 1.3) y la Plataforma Java 2, Enterprise Edition SDK 1.3 Implementación de Referencia. Sin embargo, para algunos capítulos, bien la implementación de referencia no es suficiente o necesita software adicional:

Contenedor Web

Para ejecutar los componentes Web utilizados en este libro, necesitará un contenedor Web que soporte las especificaciones Servlet 2.3 y JSP 1.2. Nosotros utilizamos la Implementación de Referencia, que utiliza el motor Jakarta Tomcat bajo la cobertura. Puede que necesite la última versión de Tomcat (disponible en http://jakarta.apache.org/tomcat) para poder ejecutar algunas los ejemplos de las librerías de designador JSP.

Contenedor EJB

Para los capítulos EJB, también necesitará un contenedor EJB que mantenga la versión 2.0 de la especificación EJB. Nosotros utilizamos WebLogic Server 6.1 de BEA (http://www.bea.com).

Bases de datos

Varios capítulos requieren también acceso a una base de datos. Para estos capítulos, nosotros utilizamos:

- Cloudscape (una versión interna viene con el RI de J2EE), http://cloudscape.com/

Software adicional

Finalmente, un par de capítulos también requieren algunas piezas de software adicional:

- JNDI SDK de Sun, que está incluido en JDK 1.3
- Java Secure Sockets Extensión (JSSE) 1.0.1, http://java.sun.com/products/jsse/
- Servidor LDPA (iPlanet Directory Server versión 4.11 de Netscape), http://www.iplanet.com/
- Servicio SMTP y/o POP3
- JSP Standar Tag Library, http://jakarta.apache.org/taglibs
- Web Services Toolkit de IBM, http://www.alphaworks.ibm.com/tech/webservicestoolkit

El código del libro funcionará en un ordenador sencillo, siempre que esté en red (es decir, que pueda ver http://localhost a través del navegador local).

El código fuente completo del libro está disponible para su desacarga en:

http://www.wrox.com/

http://www.AnayaMultimedia.es (sección "*atención al cliente*", opción "*complementos*")

Convenciones

Para ayudarle a sacar el máximo partido al texto y seguir el hilo de lo que está ocurriendo, hemos utilizado un número de convenciones a lo largo del libro:

Por ejemplo:

> **Estos recuadros contienen información importante que debe recodar y que es relevante con relación al texto en el que se introduce.**

Mientras que el estilo en cursiva se utiliza para comentarios al margen del tema que está siendo tratado.

En cuanto a los estilos en el texto:

- Al introducirlas, **destacamos** en negrita las palabras importantes.
- Mostramos las combinaciones de teclado así: *Control-A*
- Mostramos los nombres de los archivos y códigos en el texto así: `doGet()`
- El texto sobre interfaces de usuario y URL se muestran como: `Menu`

Presentamos el código de tres formas diferentes. Las definiciones de métodos y propiedades se muestran del siguiente modo:

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                        throws ServletException, IOException
```

Ejemplo de un código:

```
En nuestros ejemplos de códigos, el estilo de primer plano de código
   muestra el código nuevo, importante y pertinente
```

```
Mientras que el fondo de código muestra el código menos importante en su
   contexto o indica que ya ha sido mostrado con anterioridad.
```

Atención al cliente

Siempre valoramos recibir noticias de nuestros lectores y queremos saber qué piensa sobre este libro: lo que le ha gustado, lo que no le ha gustado y lo que considera que podemos mejorar en la próxima edición. Puede enviarnos sus comentarios, bien mediante la tarjeta de respuesta incluida al final del libro o por e-mail a `feedback@wrox.com` o `a_multimedia@anaya.es`. Asegúrese de que incluye el título del libro en su mensaje.

Cómo cargar el código de muestra para el libro

Cuando visite la página de Wrox, http://wrox.com/, simplemente localice el título en nuestra sección `Search` o utilice uno de los títulos de la lista. Haga clic en `Download` en la columna `Code` o en `Download Code` en la página de detalles del libro. También podrá descargar el codigo en la página de Anaya Multimedia, http://www.AnayaMultimedia.es (sección "*atención al cliente*", opción "*complementos*")

Los archivos disponibles para su descarga en nuestra página han sido archivados utilizando el programa WinZip. Cuando haya guardado los archivos adjuntos en una carpeta de su disco duro, necesitará extraer los archivos utilizando un programa de descompresión como WinZip o PKUnzip. Cuando extraiga los archivos, el código se extraerá normalmente en carpetas de capítulos. Cuando inicie el proceso de extracción, asegúrese de que su programa (WinZip, PKUnzip, etc.) está configurado para utilizar los nombres de las carpetas.

Erratas

Nos hemos esforzado al máximo para que no haya errores en el texto o en el código. Sin embargo, nadie es perfecto y puede haber errores. Si encuentra algún error en alguno de nuestros libros, como un error de ortografía o un segmento de código erróneo, le agradeceríamos que nos lo hiciera saber. Informándonos de las erratas, ahorrará a otro lector horas de frustración y, por supuesto, nos estará ayudando a ofrecer información de mayor calidad. Simplemente, envíenos un e-mail con la información a `support@wrox.com` o `a_multimedia@anaya.es`; su información será comprobada y, si es correcta, será enviada a la página de erratas del libro en cuestión o utilizada en siguientes ediciones del libro.

Para encontrar erratas en la página Web, vaya a http://www.wrox.com y simplemente localice el título en nuestro apartado del `Advanced Search` o en la lista de títulos. Haga clic en el vínculo `Book Errata`, que se encuentra bajo la cubierta gráfica de la página de detalle del libro.

También podrá encontrar las erratas en la página Web http://www.AnayaMultimedia.es (sección "*atención al cliente*", opción "*complementeos*").

Apoyo vía e-mail

Si desea plantear directamente un problema del libro a un experto que conozca el libro detalladamente, envíe un e-mail a `a_anayamultimedia@anaya.es` o a `support@wrox.com` (incluyendo el título del libro y los cuatro últimos números del ISBN en inglés en el campo de asunto del e-mail). Un e-mail convencional incluiría los siguientes datos:

- El **título del libro**, los **cuatro últimos dígitos del ISBN en inglés**, un **número de página** donde se encuentra el problema en el campo `Asunto`.

- Su **nombre**, los **datos de contacto** y el **problema** en el cuerpo del mensaje.

No le enviaremos ningún correo basura. Necesitamos los detalles para ahorrarle tiempo a usted y a nosotros. Cuando envíe un e-mail, éste pasará por la siguiente cadena de atención:

- Apoyo el cliente. Su mensaje es enviado a nuestro personal de atención al cliente, que serán los primeros en leerlo. Ellos cuentan con archivos sobre las preguntas más frecuentes y responderán a cualquier pregunta general sobre el libro o sobre la página Web inmediatamente.
- Editorial. Las preguntas de mayor profundidad son enviadas al equipo editor técnico responsable del libro en cuestión. Ellos tienen experiencia con el lenguaje de programación o con el producto en cuestión y están capacitados para responder a preguntas técnicas sobre el tema.
- Autores. Finalmente, en el improbable caso de que el editor no pueda resolver su problema, enviará la pregunta al autor. Intentamos proteger al autor de cualquier distracción en su trabajo; sin embargo, nos satisface poder plantearles problemas específicos. Ellos enviarán un e-mail al cliente y al editor con su respuesta y una vez más todos los lectores se beneficiarán.

El proceso de Apoyo de Wrox sólo puede ofrecer apoyo sobre temas que están directamente relacionados con nuestros títulos publicados. La respuesta a las preguntas que están fuera del ámbito normal del apoyo a nuestros libros puede encontrarla en las listas comunitarias de nuestro foro en http://p2p.wrox.com/.

p2p.wrox.com

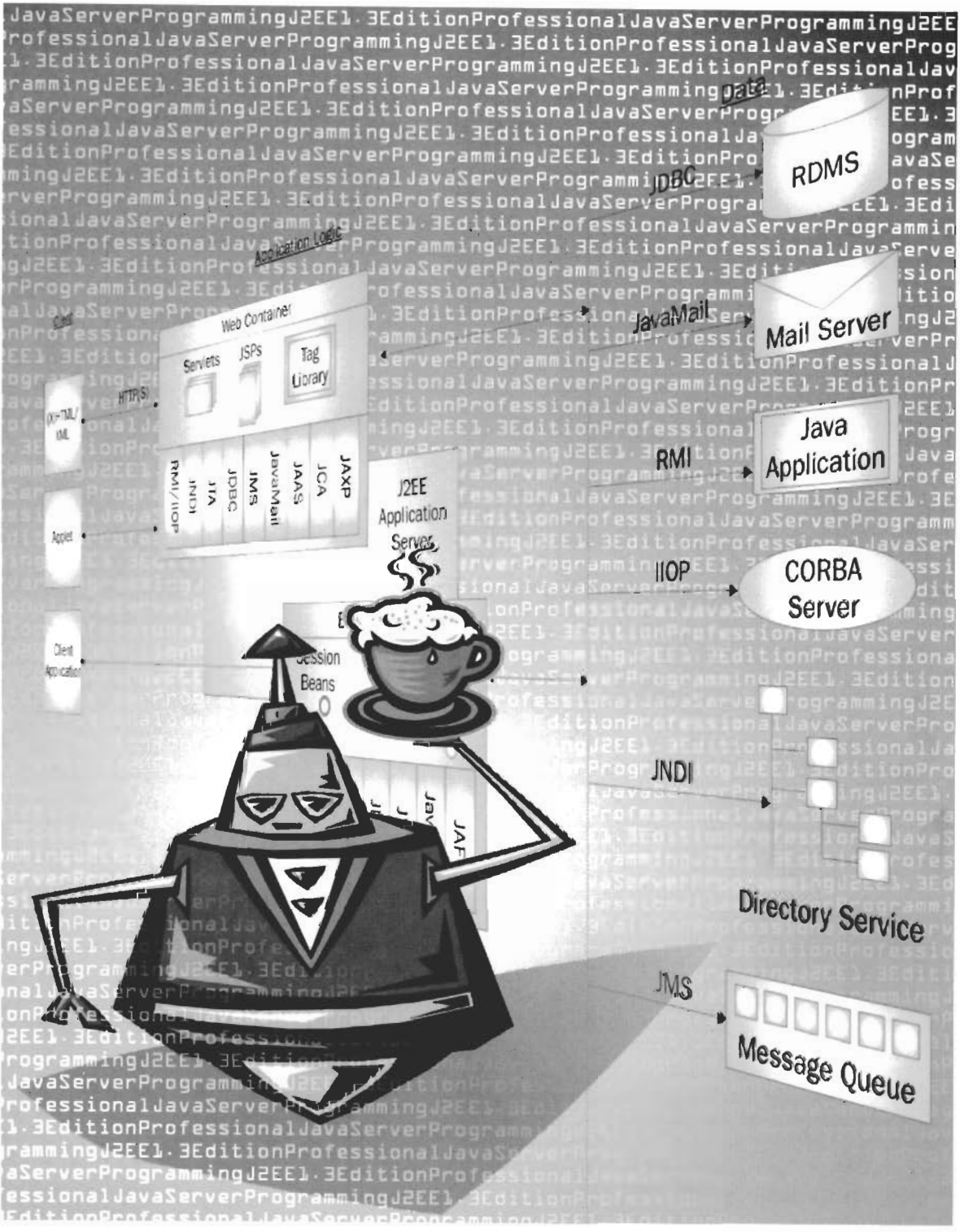
Para participar en debates con el autor y otras personas, regístrese en las listas de correo de P2P. Nuestro sistema único proporciona contacto **programmer-to-programmer™** en las listas de correo, en los foros y en los grupos de noticias, todo ello sumado al sistema individual de apoyo vía e-mail. Si envía sus dudas a P2P, puede estar seguro de que serán examinados por muchos de los autores de Wrox y otros expertos de la industria que están incluidos en nuestras listas de correo. En `p2p.wrox.com` encontrará diversas listas diferentes que le ayudarán no sólo en la lectura de este libro, sino también a desarrollar sus propias aplicaciones. Para este libro, son especialmente apropiadas las listas **j2ee** y **pro_java_server**.

Para suscribirse a una lista de correo, debe seguir los siguientes pasos:

- Vaya a http://p2p.wrox.com/
- Elija la categoría apropiada de la barra de menú situada a la izquierda
- Haga clic en la lista de correo a la que desea suscribirse
- Siga las instrucciones para suscribirse y rellene los campos de su dirección de e-mail y su contraseña
- Conteste al e-mail de confirmación que reciba
- Utilice el gestor de suscripciones para suscribirse a más listas y fije sus preferencias de correo

¿Por qué este sistema ofrece el mejor apoyo?

Puede elegir entre suscribirse a las listas de correo o puede recibirlas semanalmente. Si no tiene tiempo, o medios, para recibir las listas de correo, puede buscar en nuestros archivos on-line. Los correos basura y molestos son borrados automáticamente, y su dirección de correo está protegida por el sistema único Lyris. Las preguntas sobre cómo suscribirse o darse de baja en las listas, y para cualquier otra duda general sobre las listas, envíe un mensaje a `listsupport@p2p.wrox.com`.


Application Logic
Web Container
Servlets
JSPs
Tag Library
HTTP(S)
Applet
Client Application
RMI/IIOP
JNDI
JTA
JDBC
JMS
JavaMail
JAAS
JCA
JAXP
J2EE Application Server
Session Beans
RDMS
JDBC
JavaMail
Mail Server
Java Application
RMI
IIOP
CORBA Server
JNDI
Directory Service
JMS
Message Queue

1
Plataforma J2EE

En la actualidad, Java es uno de los lenguajes de programación más elaborados y más utilizados para la creación de software de empresa. La evolución de Java, que ha pasado de ser un medio de desarrollo de applets para ser ejecutados en navegadores a un modelo de programación capaz de manejar las aplicaciones de una empresa de hoy en día, ha sido extraordinaria. Con el paso de los años, Java ha desarrollado tres ediciones de plataformas diferentes, cada una de ellas destinada a cubrir un conjunto diferente de necesidades de programación:

- **La plataforma Java 2 Standar Edition (J2SE):**
  Es la plataforma más utilizada dentro de Java, consistente en un entorno de tiempo de ejecución y un conjunto de varios API para crear una amplia variedad de aplicaciones que abarca desde applets, pasando por aplicaciones independientes ejecutables en distintas plataformas, hasta aplicaciones de cliente para diversas aplicaciones de empresa.

- **La plataforma Java 2, Enterprise Edition (J2EE):**
  J2EE es una plataforma para crear aplicaciones de servidor. Como verá a lo largo de este libro, las aplicaciones de servidor tienen requisitos adicionales durante la fase de desarrollo. J2EE proporciona la infraestructura necesaria para satisfacer estas necesidades.

- **La plataforma Java 2, Micro Edition (J2ME):**
  Esta edición, la última en agregarse a la familia Java, permite la creación de aplicaciones Java para "micro-dispositivos" (dispositivos con un apoyo de pantalla y memoria limitado, como teléfonos móviles, PDAs, etc.).

Este libro es un recorrido por la plataforma J2EE y, en particular, por la versión 1.3 de la especificación. Ya que J2EE 1.3 requiere J2SE, también examinaremos algunas de las características de J2SE cuando así se requiera.

En este libro, comenzaremos presentando las necesidades de programación para el desarrollo de aplicaciones de empresa, las diversas tecnologías disponibles para la creación de tales aplicaciones y el

modo de crear tales aplicaciones utilizando esas tecnologías. A medida que vayamos avanzando capítulo a capítulo, nos encontraremos con una amplia variedad de tecnologías, algunas de ellas de mayor complejidad y cada una de ellas abarcando un grupo específico de necesidades técnicas.

Este libro le presentará prácticamente toda la gama de tecnologías J2EE. A pesar de la complejidad de algunas de estas áreas, este libro pretende presentar cada uno de los temas desde los principios más básicos. De este modo, el lector puede estudiar estos temas en el orden que prefiera.

Programación para empresas

J2EE lleva cuatro años de funcionamiento. Durante estos años, ha reemplazado a diversas tecnologías de propiedad y no estándar como opción idónea para e-comercio y otras aplicaciones de empresa de base Web. Considerando la multitud y la complejidad de los requisitos de infraestructura técnica para las aplicaciones de e-comercio, en este capítulo conocerá cómo J2EE se adapta a estos requisitos proporcionando una infraestructura exhaustiva. Hoy en día, J2EE es una de las dos opciones disponibles para la creación de aplicaciones de e-comercio. La otra opción la encontramos en Windows de Microsoft y en las tecnologías . NET.

Desde sus comienzos, Java ha dado lugar a la creación de nuevos modelos y tecnologías de programación en diferentes campos, desde los dispositivos más sencillos hasta aplicaciones de empresa, pasando por telefonía. Al mismo tiempo, Java ha funcionado como catalizador a la hora de conseguir que ciertos campos de la tecnología adopten formas más seguras y sólidas. La plataforma de informática de empresa de Java, la **Plataforma Java 2, Enterprise Edition (J2EE)**, es uno de estos campos.

En el pasado, los debates en los medios de comunicación, así como en los círculos técnicos, han cuestionado si Java es un lenguaje de programación o una plataforma sobre la que construir y presentar aplicaciones. J2EE es, de hecho, uno de los intentos más efectivos por parte de Sun y sus asociados en hacer que Java resulte creíble como *plataforma* para informática de empresa distribuida.

¿Por qué J2EE? ¿Por qué es la plataforma adecuada? ¿Por qué debería elegir esta tecnología para crear aplicaciones en el ámbito empresarial, desde cliente-servidor a Internet hasta móviles? Este capítulo le ofrece una visión equilibrada de J2EE y le ayuda a encontrar respuestas a estas preguntas. El resto de este libro es un análisis más profundo de las tecnologías individuales y demuestra cómo construir y gestionar con éxito tales aplicaciones de empresa.

En este capítulo introductorio centraremos nuestra atención en los siguientes puntos:

- La arquitectura técnica de J2EE.
- Las características que hacen de J2EE una plataforma creíble.
- Los retos a los que se enfrenta J2EE.
- Las tecnologías que comprende la plataforma J2EE.

En primer lugar, no obstante, empezaremos con el reto que supone hoy en día para las empresas el desarrollo de aplicaciones.

La empresa actual

Este libro aborda la creación de aplicaciones de empresa, pero ¿a qué nos referimos exactamente cuando hablamos de una "empresa"?

Una empresa es una organización económica y las aplicaciones de empresa son aquellas aplicaciones de software que facilitan diversas actividades dentro de una empresa.

Las aplicaciones de empresa pueden ser aquellas que hacen concesiones a los usuarios finales vía Internet, a colaboradores vía Internet o redes privadas, a diversas unidades económicas dentro de la empresa vía distintos tipos de interfaces de usuario, etc. En esencia, las aplicaciones de empresa son aquéllas que permiten a una empresa gestionar sus actividades económicas. Ejemplos de tales actividades incluyen planificación de recursos, inventarios y catálogos de productos, preparación de facturas, satisfacción de bienes y servicios prestados, etc. Crear aplicaciones para una empresa siempre ha sido un reto. Algunos de los factores que contribuyen a este reto y su complejidad son los siguientes:

- **Diversidad de necesidades de información**
  En una empresa, la información es creada y consumida por varios usuarios en una serie de formas diferentes, dependiendo de necesidades específicas. Es muy común encontrar que cada actividad económica procesa la información de un modo diferente.

- **Complejidad de procesos económicos**
  La mayoría de procesos económicos de empresa conllevan recabar información compleja, procesarla y distribuirla. Con mucha frecuencia, se enfrentará a una lógica compleja para recabar y procesar información. Esto tiene como resultado unos complejos requisitos técnicos y de arquitectura para crear aplicaciones de empresa.

- **Diversidad de aplicaciones**
  Debido a la compleja naturaleza de los procesos económicos de empresa, es frecuente encontrar una empresa consistente en un gran número de aplicaciones, cada una de ellas creada en distintos momentos para satisfacer las necesidades de distintos procesos económicos. Esto desemboca con frecuencia en la presencia de aplicaciones creadas utilizando diferentes arquitecturas y tecnologías. Uno de los retos a los que se enfrentan las empresas hoy en día es la necesidad de conseguir que dichas aplicaciones se comuniquen entre sí de modo que los procesos económicos puedan llevarse a cabo sin interrupciones.

Estos factores son muy comunes y las empresas incurren en enormes costes para crear y gestionar aplicaciones que hagan frente a estos retos.

Durante los últimos años, estos retos han alcanzado dimensiones monstruosas. Gracias a Internet y al reciente crecimiento del e-comercio, los dispositivos de información de una empresa son ahora más valiosos. Este cambio hacia una economía de la información está forzando a muchas compañías a reformular incluso sus prácticas económicas más básicas. Con el objetivo de mantener un margen competitivo, la adopción de nuevas tecnologías para cubrir las necesidades del momento rápidamente se ha convertido en un factor clave en la habilidad de una empresa para explotar al máximo sus dispositivos de información. Fundamentalmente, la adaptación de estas nuevas tecnologías para que funcionen en combinación con los sistemas legados ya existentes es ahora uno de los requisitos principales de la empresa.

Uno de los lugares donde estos cambios se han percibido de un modo más intenso ha sido en el ámbito del desarrollo de aplicaciones. A lo largo de los últimos años, los fondos y el tiempo invertidos en el desarrollo de aplicaciones se han visto reducidos, mientras que las demandas para crear complejos procesos económicos han aumentado. Sin embargo, ambos son obstáculos que los promotores pueden superar, pero también se encuentran con los siguientes requisitos que cumplir:

- **Productividad de programación**
  La adopción directa de nuevas tecnologías es insuficiente si éstas no son utilizadas adecuadamente aprovechando su potencial al máximo y si no son integradas apropiadamente con otras tecnologías

relevantes. De este modo, la habilidad para desarrollar y después desplegar las aplicaciones tan rápida y efectivamente como sea posible es muy importante. Conseguir este objetivo puede resultar complicado debido a la gran variedad de tecnologías y normas que han sido desarrolladas a lo largo de los años, que requieren capacidades altamente desarrolladas; tan sólo adquirirlas y mantenerse a la altura es un problema en sí mismo. Además, el rápido ritmo de cambio en las mismas normas dificulta la tarea de asegurar un eficiente engranaje de las tecnologías.

- **Fiabilidad y disponibilidad**
  En la economía actual de Internet, el tiempo de inactividad puede resultar fatal para una empresa. La habilidad para organizar y poner en marcha sus operaciones de base Web, y mantenerlas activas, es crucial para lograr el objetivo con éxito. Y por si no fuera suficiente, también debe ser capaz de garantizar la fiabilidad de sus transacciones económicas de modo que sean procesadas por completo y con exactitud.

- **Seguridad**
  Internet no sólo ha aumentado exponencialmente el número de usuarios potenciales sino también el valor de la información de una compañía, de modo que la seguridad de esta información se ha convertido en una preocupación de primer orden. Es más, a medida que las tecnologías son más avanzadas, las aplicaciones más sofisticadas y las empresas más complejas, poner en marcha un modelo de seguridad efectivo es cada vez más difícil.

- **Reajustabilidad**
  La capacidad de una aplicación para crecer y ajustarse a una nueva demanda, tanto en su operación como en su base de usuario, es esencial. Esta condición se cumple especialmente cuando consideramos que la base potencial de usuarios de una aplicación puede ser de millones de usuarios individuales, vía Internet. La capacidad efectiva de reajuste depende no sólo de la habilidad para manejar un amplio aumento en el número de clientes sino también del uso efectivo de los recursos de un sistema.

- **Integración**
  Aunque la información ha crecido hasta convertirse en un dispositivo económico clave, mucha de esta información existe como datos en sistemas de información viejos y obsoletos. Para maximizar la utilidad de esta información, las aplicaciones deben poder ser integradas en los sistemas de información ya existentes, lo cual no es necesariamente una tarea fácil ya que las tecnologías actuales con frecuencia han avanzado hasta superar algunos de estos sistemas legados. La habilidad para combinar las nuevas y las viejas tecnologías es fundamental para el éxito del desarrollo de las empresas de hoy en día.

¿Estamos buscando fórmulas mágicas para enfrentarnos a estos desafíos? A pesar de las promesas de evangelistas y vendedores, no es aconsejable esperar (o confiar en) una solución que evite o supere por completo estos retos. Lo que las empresas están buscando es una tecnología y una infraestructura permisiva que simplifique algunos de los temas técnicos complejos. A medida que vaya avanzando en este libro, encontrará el análisis de estos temas.

Ninguno de estos ámbitos problemáticos es especialmente nuevo para los creadores de programas de una empresa. Pero resolverlos de un modo exhaustivo y rentable sigue siendo crucial. Puede que sea consciente de que han existido diversas tecnologías para resolver una o más de las demandas anteriores. Sin embargo, lo que siempre ha faltado ha sido una plataforma con una rica infraestructura y numerosas posibilidades de arquitectura que promueva al mismo tiempo un rápido entorno de desarrollo.

El objetivo de J2EE es simplificar algunas de las complejidades técnicas y especificar los siguientes puntos para crear aplicaciones de empresa:

- Un **modelo de programación**, consistente en un conjunto de interfaces de programación de aplicaciones (API) y en los enfoques para la creación de dichas aplicaciones.

- Una **infraestructura de aplicación**, para respaldar las aplicaciones de empresa creadas utilizando los API.

¿Es Java la respuesta?

Hasta ahora hemos examinado nuestra arquitectura de sistema desde una perspectiva agnóstica en cuanto a su ejecución. De hecho, existen muchos caminos potenciales que pueden tomarse para poner en marcha su empresa. Existen esencialmente dos caminos principales: uno emprendido por Microsoft, con su nueva iniciativa. NET, y el segundo, emprendido por Sun y otras empresas tales como BEA, IBM, Oracle, etc. Teniendo en cuenta las opciones, ¿por qué debería ser Java una excelente elección? Examinemos algunas de sus ventajas.

Independiente de la plataforma

Con la información de una empresa desplegada en formatos dispares, en distintas plataformas y aplicaciones, es importante adoptar un lenguaje de programación que pueda funcionar perfectamente en la empresa sin tener que recurrir a mecanismos extraños e ineficientes de traducción. Un modelo de programación unificador también reduce las dificultades surgidas de la integración de muchas de las diversas tecnologías que se convierten en específicas para ciertas plataformas y aplicaciones.

Objetos gestionados

J2EE proporciona un entorno gestionado para componentes y las aplicaciones de J2EE son **céntricas respecto del contenedor**. Ambas nociones son críticas para construir aplicaciones de cliente. Al ser gestionados, los componentes de J2EE utilizan la infraestructura proporcionada por los servidores de J2EE sin que el programador sea consciente de ello. Las aplicaciones de J2EE son también **declarativas**, un mecanismo con el que puede modificar y controlar el funcionamiento de las aplicaciones sin cambiar de código. A primera vista, estas características pueden parecer incomodas de ejecutar. Sin embargo, cuando considera las aplicaciones a gran escala con varios cientos de componentes interactuando para ejecutar complejos procesos empresariales, estas características hacen que crear y mantener tales aplicaciones resulte menos complejo. Junto con su independencia respecto de la plataforma, éste es uno de los aspectos más importantes de J2EE.

Reusabilidad

La reutilización del código es el Santo Grial de la programación. Separar los requisitos económicos de una aplicación en sus partes integrantes es un modo de conseguir la reutilización; utilizar la orientación al objeto para encapsular la funcionalidad compartida es otro. Sin embargo, a diferencia de los objetos, los componentes distribuidos requieren una infraestructura más compleja para su construcción y gestión. Los conceptos básicos orientados a objetos no proporcionan este tipo de marco, pero la Enterprise Edition de Java ofrece una arquitectura notablemente rigurosa para la reutilización de componentes. Los componentes de empresa (llamados Enterprise JavaBeans) desarrollados en J2EE son de básica granulometría (aunque la especificación EJB 2.0 en J2EE 1.3 permite un enfoque de granulometría más fina) y reproducibles. Manteniendo estos componentes básicos, es posible crear funcionalidades de empresa complejas de un modo libremente coordinado. Además, ya que los componentes son reproducibles, lo cual quiere decir que es posible identificar ciertos metadatos sobre los componentes, las aplicaciones pueden ser creadas componiendo tales componentes. Ambas características fomentan la reutilización del código a alta granulometría.

Modularidad

A la hora de desarrollar una aplicación de servidor completa, los programas pueden ampliarse y complicarse rápidamente. Siempre es mejor romper una aplicación en módulos discretos, cada uno de ellos responsable de una tarea específica. Esto hace que nuestras aplicaciones sean mucho más fáciles de mantener y comprender. Por ejemplo, los servlets de Java, las Páginas de JavaServer y Enterprise JavaBeans proporcionan un modo de modularizar nuestra aplicación, rompiendo nuestras aplicaciones en diferentes niveles y tareas individuales.

A pesar de las características a su favor mencionadas, Java no presentó un modelo de programación unificador a escala de empresa hasta principios del año 2000. No es que no tuviera la capacidad, sino que en el pasado estaba algo desorganizada. Sun reconoció este fallo y lanzó la **Plataforma Java 2, Enterprise Edition (J2EE).**

> **La idea subyacente a la plataforma J2EE es proporcionar un estándar sencillo y unificado para aplicaciones distribuidas en un modelo de aplicación basado en componentes.**

En el resto del capítulo, y por supuesto en el resto del libro, analizaremos lo que aporta J2EE al desarrollo del lado servidor con Java.

Estilos de arquitectura de empresa

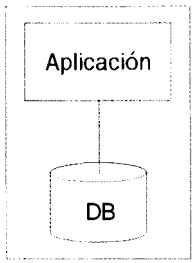
Antes de que entremos de lleno en la arquitectura de J2EE, es necesario que consideremos los estilos de arquitectura de las aplicaciones distribuidas contemporáneas: las arquitecturas de nivel 2, nivel 3 y nivel n. El propósito de esta sección es principalmente capacitar al lector para reconocer el alcance y la localización de estos patrones. Sin embargo, debemos tener presente que en realidad posiblemente nuestras aplicaciones requieran patrones más complejos o en ocasiones una combinación de éstos.

Aunque estos estilos de arquitectura se encuentran con bastante frecuencia en las empresas de la actualidad, merece la pena señalar que estos estilos surgieron debido a la llegada de nuevas plataformas de hardware para clientes y servidores más baratas y de tecnologías de redes. Anteriormente, los motores de las empresas eran sistemas principales con todos los servicios informáticos (desde la generación usuario-interfaz hasta procesamiento de transacciones de gran volumen) centralizados. Consecuentemente, la tecnología cliente-servidor fue lo que provocó una automatización masiva de la empresa.

Los sistemas cliente-servidor típicos están basados en la arquitectura de dos niveles, donde existe una clara separación entre los datos y la lógica presentación/empresa. Estos sistemas están generalmente controlados por datos, siendo el servidor, en gran cantidad de ocasiones, un servidor de base de datos. Aunque este enfoque nos permite compartir datos en la empresa, tiene muchos inconvenientes.

Arquitectura de dos niveles

En una aplicación tradicional de 2 niveles, la carga de procesamiento es facilitada el PC cliente mientras que el servidor actúa simplemente como controlador del tráfico entre la aplicación y los datos. Como resultado, el rendimiento de la aplicación no sólo sufre debido a los recursos limitados del PC sino que el tráfico de la red también tiende a aumentar. Cuando la aplicación completa es procesada en un PC, la aplicación es forzada a realizar múltiples peticiones de datos antes incluso de presentar algo al usuario. Estas múltiples peticiones de bases de datos pueden sobrecargar la red:



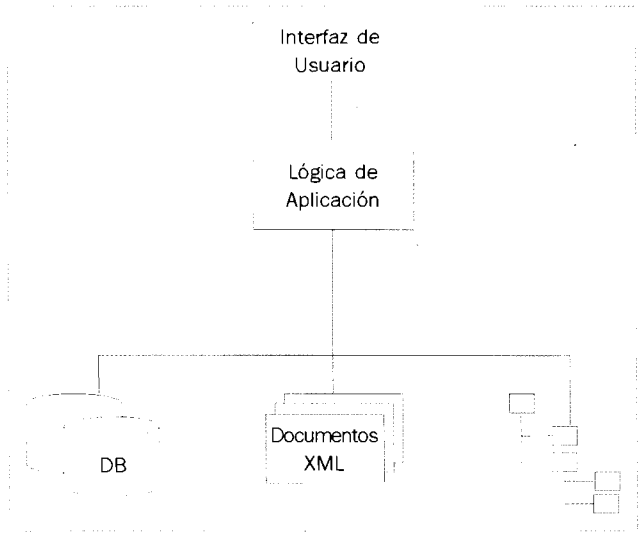
Otro problema típico relacionado con el enfoque de 2 niveles es el del mantenimiento. Incluso el menor cambio realizado a una aplicación puede conllevar una completa alteración en la base de usuario. Aunque sea posible automatizar el proceso, todavía debe enfrentarse a la actualización de cada instalación de cliente. Es más, algunos usuarios puede que no estén preparados para una alteración total y posiblemente ignoren los cambios mientras que otro grupo puede que insista en realizar los cambios de inmediato. Esto puede provocar que diferentes instalaciones de cliente utilicen diferentes versiones de la aplicación.

Arquitectura de tres niveles

Para enfrentarse a estos temas, la comunidad de software desarrolló la noción de una arquitectura de 3 niveles. Una aplicación se divide en tres capas lógicas distintas, cada una de ellas con un grupo de interfaces perfectamente definido. La primera capa se denomina **capa de presentación** y normalmente consiste en una interfaz gráfica de usuario de algún tipo. La capa intermedia, o **capa de empresa**, consiste en la aplicación o lógica de empresa, y la tercera capa, la capa de datos, contiene los datos necesarios para la aplicación.

La capa intermedia (lógica de aplicación) es básicamente el código al que recurre el usuario (a través de la capa de presentación) para recuperar los datos deseados. La capa de presentación recibe entonces los datos y los formatea para su presentación. Esta separación entre la lógica de aplicación de la interfaz de usuario añade una enorme flexibilidad al diseño de la aplicación. Puede construirse y desplegarse múltiples interfaces de usuario sin cambiar en absoluto la lógica de aplicación, siempre que la lógica de aplicación presente una interfaz claramente definida a la capa de presentación. Como verá más adelante en este capítulo, y en el resto del libro, J2EE ofrece diversas abstracciones para satisfacer las necesidades de cada una de estos niveles. Por ejemplo, los EJB ofrecen mecanismos para abstraer tanto el acceso a datos como la lógica de empresa. De un modo similar, los servlets y las Páginas JavaServer le permiten abstraer la capa de presentación y su interacción con la capa de empresa.

La tercera capa contiene los datos necesarios para la aplicación. Estos datos consisten en cualquier fuente de información, incluido una base de datos de empresa como Oracle o Sybase, un conjunto de documentos XML o incluso un servicio de directorio como el servidor LDAP. Además del tradicional mecanismo de almacenamiento relacional de bases de datos, existen muchas fuentes diferentes de datos de empresa a las que pueden acceder sus aplicaciones:



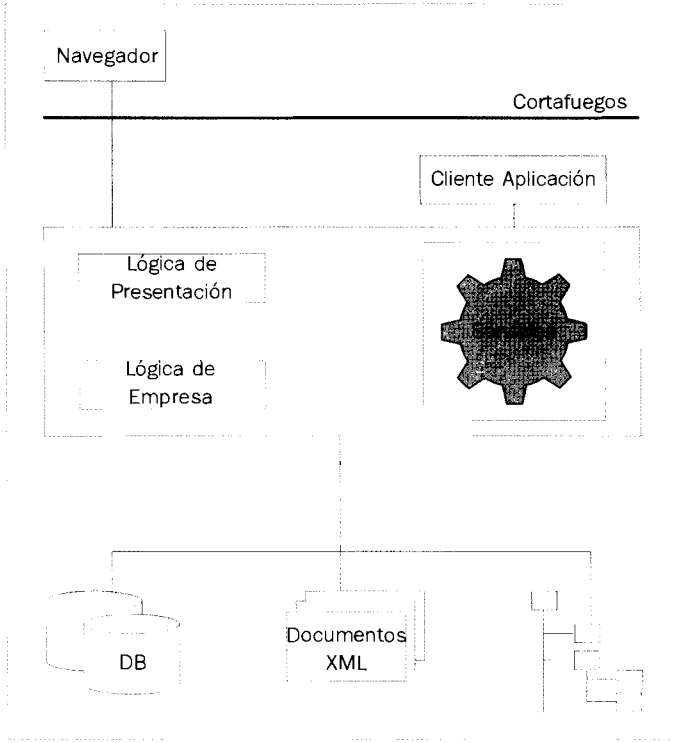
Sin embargo, todavía no hemos finalizado nuestra subdivisión de la aplicación. Podemos dar un paso más en la segregación para crear una arquitectura de n niveles.

Arquitectura de n niveles

Como sugiere el título, no hay un modo más preciso de definir las capas de aplicación para un sistema de n niveles. De hecho, este tipo de sistema puede dar cabida a varias configuraciones diferentes. En una arquitectura de n niveles, la lógica de aplicación está lógicamente dividida por funciones y no físicamente.

Una arquitectura de n niveles se descompone en las siguientes partes:

- Una **interfaz de usuario** que maneja la interacción del usuario con la aplicación. Ésta puede ser un navegador Web ejecutado mediante un cortafuegos, una aplicación de escritorio más laboriosa o incluso un dispositivo inalámbrico.
- Una **lógica de presentación** que define lo que muestra la interfaz de usuario y cómo son gestionadas las demandas del usuario. Dependiendo de las interfaces de usuario que se mantengan, puede que sea necesario contar con versiones de la lógica de presentación ligeramente diferentes para satisfacer al cliente adecuadamente.
- Una **lógica de empresa** que modele las reglas de empresa de la aplicación, a menudo a través de la interacción con los datos de la aplicación.
- **Servicios de infraestructura** que proporcionen la funcionalidad adicional requerida por los componentes de la aplicación, tales como mensajería y apoyo transaccional.
- La **capa de datos** donde residen los datos de la empresa.



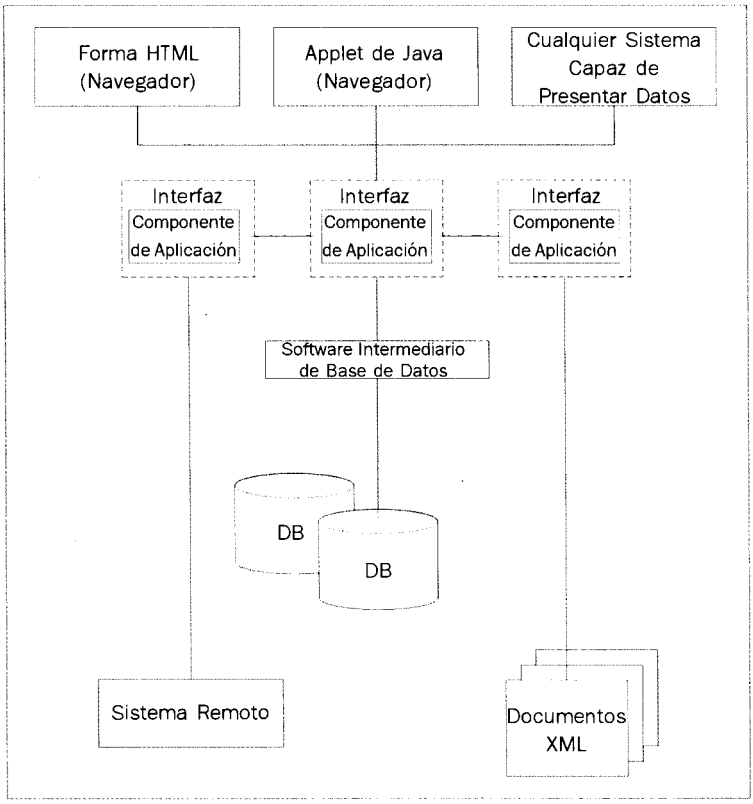
Las aplicaciones basadas en esta arquitectura emplean esencialmente el patrón ***Model-View-Controller*** *(**MVC**). Lo que esto significa fundamentalmente es que los datos (el modelo) son independientes respecto de la presentación de la información (la vista). En una posición intermedia se encuentra la lógica de aplicación/empresa (el controlador) que controla el flujo de información. De este modo, una aplicación es diseñada a partir de estos tres componentes funcionales (modelo, vista y controlador) interactuando entre sí.*

Arquitectura de empresa

Hasta el momento nos hemos centrado realmente en una sola arquitectura de aplicación; nos arriesgamos a considerar estas aplicaciones "tuberías". En otras palabras, puede que terminemos con diferentes aplicaciones (posiblemente incluso con diferentes arquitecturas) que no se comunican entre sí. En una empresa, intentamos crear una arquitectura cohesiva que incorpore diferentes aplicaciones.

Antes que un cambio en la arquitectura (la arquitectura de empresa es básicamente n niveles) necesitamos un cambio en la percepción. Para convertir un sistema n niveles en un sistema de empresa, simplemente necesitamos ampliar el nivel intermedio permitiendo la existencia de múltiples **objetos de aplicación** en lugar de una única aplicación. Estos objetos de aplicación deben tener una interfaz individual que les permita funcionar entre sí.

Una interfaz puede concebirse como un contrato. Cada objeto afirma a través de su interfaz que aceptará ciertos parámetros, que realizará ciertas operaciones y devolverá un conjunto específico de resultados. Los objetos de la aplicación se comunican entre sí utilizando sus interfaces, como muestra el siguiente diagrama:



Con la arquitectura de empresa, podemos tener múltiples aplicaciones utilizando un conjunto común de componentes a través de una organización. Esto promueve la normalización de la práctica empresarial gracias a la creación de un único conjunto de funciones de empresa al que puede acceder toda la organización. Si la norma de una empresa cambia, dichos cambios deben trasladarse también al "objeto de empresa" y, si es necesario, a la interfaz asociada y consecuentemente a cualquier objeto que acceda a la interfaz. Observe, sin embargo, que ciertas funcionalidades de los sistemas legados existentes no pueden eliminarse y que debe desarrollarse un "envoltorio" (o interfaz) específico a esta funcionalidad en el nivel intermedia.

Es importante señalar que a la hora de diseñar un objeto y su interfaz, es conveniente hacer que la interfaz sea lo más genérica posible para evitar cambios en el futuro. Debido que a otros objetos se comunican con la interfaz y no con el objeto en sí mismo, los cambios realizados al objeto, y no a la interfaz, son relativamente sencillos y rápidos.

Observe que, en todos estos patrones, las interfaces de usuario son la parte que aparece en la parte superior de cada diagrama. A partir de estos diagramas, puede obtener la impresión de que las interfaces de usuario son las que siempre dirigen los sistemas. Aunque ésta es la forma más comúnmente visible de interacción de sistema, al considerar la integración de aplicaciones intra e inter-empresa, y en los procesos entre empresas, comprobará que las aplicaciones interactúan con otras aplicaciones. No obstante, la mayoría de las tecnologías que examinaremos y los diseños que encontrará en este libro también se aplican a estos sistemas.

La plataforma J2EE

Como podrá comprobar en el resto del libro, J2EE es una de las mejores soluciones encontradas hasta el momento para satisfacer las demandas de la empresa de hoy en día. J2EE especifica tanto la infraestructura para gestionar sus aplicaciones, como los servicios API para construir sus aplicaciones.

La plataforma J2EE es esencialmente un entorno distribuido aplicación-servidor, un entorno Java que ofrece:

- Un conjunto de varios API de extensión Java para construir aplicaciones. Estos API definen un modelo de programación para aplicaciones J2EE.
- Una infraestructura de periodo de ejecución para albergar y gestionar aplicaciones. Éste es el periodo de ejecución en el que residen sus aplicaciones.

Las aplicaciones que puede desarrollar con estos dos elementos pueden ser programas para controlar páginas Web o componentes para implementar transacciones complejas de bases de datos, o incluso applets de Java, todos ellos distribuidos por la red.

Periodo de ejecución de J2EE

Si bien J2EE asocia los API que han existido de una forma u otra durante algún tiempo, quizás su aspecto más significativo sea la abstracción de la infraestructura del **periodo de ejecución.** Observe que la especificación J2EE no especifica cómo debería o podría construirse un periodo de ejecución. Esto provoca una clara demarcación entre las aplicaciones y la infraestructura del periodo de ejecución. Esta demarcación permite al periodo de ejecución abstraer la mayor parte de los servicios de infraestructura que los desarrolladores de programas de empresa han intentado construir tradicionalmente por su cuenta. Como resultado, los desarrolladores de programas de la aplicación J2EE sólo podían centrar sus esfuerzos en la lógica de aplicación y servicios relacionados, mientras relegaban el periodo de ejecución para todos los servicios relacionados con la infraestructura. Sin embargo, ¿cuáles son los requisitos de infraestructura para aplicaciones de empresa?

Es interesante señalar que el desarrollo de aplicaciones en un entorno de ritmo crecimiento rápido, como Internet, tiene como resultado normalmente diseños efímeros.

Por otro lado, a la hora de considerar los aspectos de diseño a largo plazo, podemos tener dificultades para encontrar un término medio entre las demandas a corto plazo y a largo plazo. Existen tecnologías que se prestan a un rápido desarrollo y otras tecnologías que permiten construir aplicaciones teniendo en cuenta este tipo de intereses a largo plazo como la reusabilidad, coste de mantenimiento, etc. A menudo, estos dos tipos no se combinan, pero J2EE es lo suficientemente flexible como para construir aplicaciones que incluyan ambos. La razón es que J2EE le permite construir cada capa de su aplicación en combinación libre con todas las otras capas. Cada capa puede consecuentemente evolucionar para cubrir sus respectivas necesidades de evolución. De este modo, ajustándose a un sistema diseñado sobre interfaces, la implementación es muy ampliable.

Además de especificar una serie de API estándar, la arquitectura de J2EE también proporciona un medio uniforme de acceder a los servicios a escala de la plataforma a través de su entorno de periodo de ejecución. Estos servicios incluyen transacciones distribuidas, seguridad, mensajes, etc. Antes de pasar a los detalles del enfoque de J2EE, examinemos el procesamiento informático distribuido tradicional.

Hasta la llegada de J2EE, el procesamiento informático distribuido era considerado, en general, programación cliente-servidor. Grabaríamos una aplicación de servidor implementando una interfaz, una aplicación de cliente para conectar con el servidor y entonces iniciar el servidor y el cliente. Aunque este

proceso parece bastante sencillo, en la práctica existen varios obstáculos críticos, dependiendo de la tecnología utilizada.

Por ejemplo, pensemos en un objeto CORBA para construir aplicaciones distribuidas. El procedimiento típico para construir objetos CORBA de lado servidor empezaría especificando una interfaz, utilizando el Lenguaje de Definición de Interfaces (IDL), para cada objeto. Los pasos siguientes incluyen compilar este IDL para generar "stubs" y "esqueletos" para el lenguaje que elija, implementando el objeto basado en el esqueleto, grabando después la aplicación de cliente, preparando el entorno, etc. En este caso, el stub es la clase que representa el objeto CORBA en el lado cliente. El esqueleto proporciona el método con el que es implementada la lógica del lado servidor. Este procedimiento en sí mismo no es complicado y puede automatizarse fácilmente. Como irá aprendiendo en este libro, los EJB requieren también algunos de estos pasos y son similares en concepto a los objetos CORBA.

Considere ahora que nuestros servidores, al igual que los clientes, necesitan acceder a servicios tales como transacciones distribuidas, mensajería, etc. Para hacer uso de estos servicios, sería necesario que añadiéramos una importante proporción de código de instalación a nuestras aplicaciones. En muchas ocasiones, será necesario que instalemos y configuremos diferentes soluciones de software intermedio, y que realicemos llamadas API a los API específicos de vendedor para acceder a los servicios. Además de servicios tales como el acceso a bases de datos relacionales, la mayoría de estos servicios son de propiedad o no estándar. El resultado es que nuestras aplicaciones serán más complejas, invertirán más tiempo y serán más caras de desarrollar, gestionar y mantener.

Además de tener que gestionar todos estos API diferentes, existe otra demanda crítica en las aplicaciones del lado servidor. En el lado servidor, los recursos son escasos. Por ejemplo, no podemos permitirnos crear el mismo número de objetos podríamos crear normalmente en aplicaciones del lado cliente. Otros recursos del lado servidor que requieren especial atención incluyen hilos de ejecución (threads), conexiones de bases de datos, seguridad, transacciones, etc. Construir a medida una infraestructura que maneje este tipo de recursos ha sido siempre un desafío. Esta tarea resulta casi imposible en la economía de Internet. ¿Se molestaría en construir una reserva de conexión, o un objeto caché, o "una elegante" capa de objeto para acceso a bases de datos, cuando su ciclo de vida de desarrollo es sólo de tres meses?

Debido a que estos requisitos de lado servidor son bastante comunes a una amplia variedad de aplicaciones, es más apropiado considerar una plataforma que cuente con soluciones integradas. Esto nos permite separar estos aspectos de infraestructura de los aspectos más directos relacionados con traducir sus requisitos de aplicación en un software que funcione. El periodo de ejecución de J2EE se enfrenta a dichos aspectos. Fundamentalmente, dejamos que sea el vendedor de servidor J2EE quien implemente los rasgos específicos de acuerdo con las normas establecidas.

Como ya hemos mencionado anteriormente, J2EE no especifica la naturaleza y estructura del periodo de ejecución. En su lugar, introduce lo que se denomina **contenedor** y mediante los API de J2EE, especifica un contrato entre contenedores y aplicaciones. Tendrá más detalles sobre este contrato más adelante.

Antes de examinar con más detalle los contenedores de J2EE, revisemos brevemente los API de J2EE.

Los API de J2EE

Las aplicaciones distribuidas requieren acceso a una serie de servicios de empresa. Los servicios típicos incluyen procesamiento de transacciones, acceso a bases de datos, mensajería, multihilos (multithreading), etc. La arquitectura de J2EE unifica el acceso a estos servicios en un API de servicios de empresa. Sin embargo, en lugar de tener que acceder a estos servicios a través de interfaces de propietario o no estándar, los programas de aplicación en J2EE pueden acceder a estos API mediante el contenedor.

Una típica plataforma J2EE comercial (o servidor de aplicación J2EE) incluye uno o más contenedores y el acceso a los API de empresa viene especificado por la J2EE.

Fíjese en que los servidores de aplicación J2EE no necesitan implementar estos servicios ellos mismos; los contenedores sólo deben proporcionar acceso a cada implementación de servicio a través de un API de J2EE. Por ejemplo, una implementación J2EE puede delegar las llamadas API de Java Message Service a una solución de software intermedio orientada a mensajes. Otra implementación podría incluir una solución de software intermedio orientada a mensajes dentro de un contenedor.

La especificación de la plataforma J2EE 1.3 incluye un grupo de extensiones estándar Java que cada plataforma J2EE debe mantener:

- **JDBC 2.0**
  Más específicamente, JDBC 2.0 Optional Package. Este API mejora el API estándar de JDBC 2.0 añadiendo medios más eficientes de obtener conexiones, reserva de conexión, transacciones distribuidas, etc. Observe que la siguiente versión de JDBC, el API de JDBC 3.0, combina la extensión API con el API JDBC habitual. Quizás futuras versiones de J2EE incluyan JDBC 3.0.
- **Enterprise JavaBeans (EJB) 2.0**
  Especifica una estructura de componentes para aplicaciones multidistribuidas multinivel. Proporciona un medio estándar para la definición de componentes de lado servidor y especifica una rica infraestructura de periodo de ejecución para albergar componentes en el lado servidor.
- **Java Servlets 2.3**
  El API Java Servlet proporciona abstracciones orientadas al objeto para construir aplicaciones Web dinámicas.
- **Java Server Pages (JSP) 1.2**
  Esta extensión refuerza un poco más las aplicaciones Web de J2EE proporcionando desarrollo de aplicaciones Web dirigidas por un modelo.
- **Java Message Service (JMS) 1.0**
  JMS ofrece un API Java para colas de mensaje y publica y suscribe tipos de servicios de software intermediario orientados al mensaje.
- **Java Transaction API (JTA) 1.0**
  Este API está destinado a implementar aplicaciones transaccionales distribuidas.
- **JavaMail 1.2**
  Este API proporciona una estructura independiente de la plataforma y del protocolo para construir aplicaciones de e-mail basadas en Java.
- **JavaBeans Activation Framework (JAF) 1.0**
  Este API es necesario para el API JavaMail. El API JavaMail utiliza JAF para determinar los contenidos de un mensaje MIME (Multipurpose Internet Mail Extension) y determina qué operaciones apropiadas pueden realizarse en diferentes partes de un mensaje.
- **API Java para XML Parsing (JAXP) 1.1**
  Este API proporciona abstracciones para analizadores XML y API de transformación. JAXP ayuda a aislar analizadores específicos XML, o implementaciones DOM (Document Object Model), o API de transformación XSLT de códigos de aplicación J2EE.
- **Java Conector Architecture (JCA) 1.0**
  Este API ha sido incluido recientemente en J2EE y proporciona un medio para integrar componentes de aplicación J2EE para sistemas de información legados.
- **Java Authenticaction and Authorization Service (JAAS) 1.0**
  Este API proporciona mecanismos de autentificación y autorización para las aplicaciones J2EE.

La mayoría de estos API son especificaciones, independientes de implementación. Es decir, se debería poder acceder a los servicios proporcionados por estos API de un modo estándar, con independencia de cómo son implementados.

Los API específicos de J2EE mencionados anteriormente se suman a los siguientes API de J2SE:

- **API Java Interface Definition Language (IDL)**
  Este API permite a los componentes de aplicación J2EE invocar objetos CORBA vía IIOP (véase más adelante).
- **JDBC Core API**
  Este API proporciona las instalaciones de programación de bases de datos básicas.
- **API RMI-IIOP**
  Proporciona una implementación del habitual API **Java Remote Meted Invocation (RMI)** sobre **Internet Inter.-ORB Protocol (IIOP)**. Salva la distancia entre las aplicaciones RMI y CORBA. Éste es el protocolo de comunicación asignado a utilizar entre contenedores J2EE.
- **API JNDI**
  El API **Java Naming and Directory Interface (JNDI)** normaliza en acceso a diferentes tipos de servicios de designación y directorio disponibles en la actualidad. Este API está diseñado para ser independiente de cualquier implementación específica de servicio de designación o directorio. J2SE también especifica una interfaz proveedora de servicio JNDI (SPI), para ser implementada por proveedores de servicios de designación y directorio.

Examinaremos más detalladamente estos API en posteriores apartados de este capítulo.

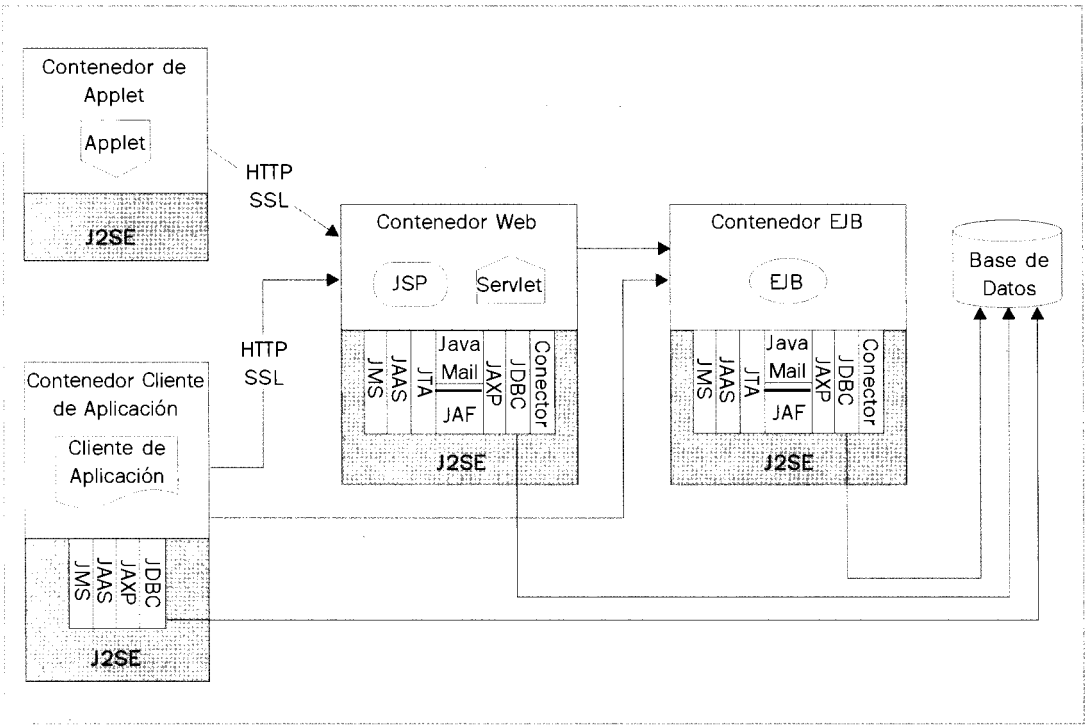
Arquitectura J2EE – Contenedores

Como ya hemos visto en la sección anterior, una típica plataforma comercial J2EE incluye uno o más **contenedores**. Pero ¿qué exactamente un contenedor?

> **Un contenedor J2EE es un periodo de ejecución para gestionar los componentes de la aplicación desarrollados según las especificaciones del API y destinados a proporcionar acceso a los API de J2EE.**

Más allá de la identidad asociada al periodo de ejecución, J2EE no especifica ninguna identidad para contenedores. Esto se traduce en una gran flexibilidad para conseguir una serie de características dentro del periodo de ejecución del contenedor.

La siguiente figura muestra la arquitectura de J2EE en términos de sus contenedores y sus API.



Esta arquitectura muestra cuatro contenedores:

- Un **contenedor Web** para albergar servlets Java y páginas JSP.
- Un **contenedor EJB** para albergar componentes Enterprise JavaBean.
- Un **contenedor applet** para albergar applets Java.
- Un contenedor de **aplicación cliente** para albergar aplicaciones Java estándar.

En este libro, nuestro centro de atención se limita únicamente a los contenedores Web y EJB.

Cada uno de estos contenedores proporciona un entorno de periodo de ejecución para los respectivos componentes. Los componentes J2EE también son denominados objetos gestionados, ya que estos objetos con creados y gestionados en el interior del periodo de ejecución del contenedor.

En la figura anterior, los bloques verticales situados en la parte inferior de cada contenedor representan los API de J2EE. Además del acceso a estos API a nivel infraestructura, cada contenedor también implementa el correspondiente API específico a cada contenedor (API Java Servlet para el contenedor Web y el API EJB para el contenedor EJB).

Los rectángulos (servlets, páginas JSP y los EJB) de esta figura son los programas que se desarrollan y se albergan en estos contenedores. En el lenguaje J2EE, estos programas se conocen como **componentes de aplicaciones**.

Observe que la figura anterior no incluye los API J2SE enumerados anteriormente, es decir, el Java IDL, memoria JDBC, RMI-IIOP y JNDI.

En esta arquitectura, existen principalmente dos tipos de clientes:

- **Clientes Web**, que se ejecutan normalmente en navegadores Web.
  Para estos clientes, la interfaz de usuario es generada en el lado servidor como HTML o XML y es descargada y después convertida por los navegadores. Estos clientes utilizan http para comunicarse con contenedores Web. Los componentes de aplicación en los contenedores Web incluyen servlets Java y páginas JSP. Estos componentes implementan la funcionalidad requerida por los clientes Web. Los contenedores Web son los responsables de aceptar solicitudes por parte de los clientes Web y de generar respuestas con la ayuda de los componentes de la aplicación.

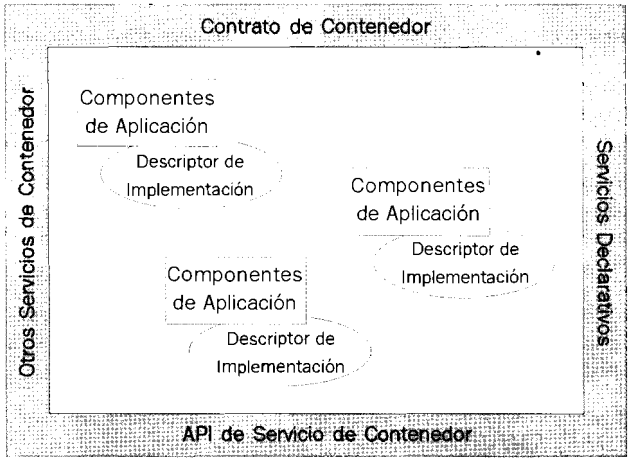
  Otros tipos de clientes relacionados incluyen dispositivos móviles habilitados por la Web capaces de generar WML (Wireless Markup Language) o cHTML (Compact HTML). Para estos clientes, el contenido puede ser generado directamente, o bien el contenido generado como XML puede ser transformado en estos lenguajes de marcado utilizando hojas de estilo XML.

- Los **clientes EJB** son aplicaciones que acceden a componentes EJB en contenedores EJB.
  Existen tres posibles tipos de clientes EJB. La primera categoría está constituida por los clientes de aplicación. Los clientes de aplicación son aplicaciones autosuficientes que acceden a los componentes EJB utilizando el protocolo RMI-IIOP. La segunda categoría de clientes de aplicación está formada por componentes del contenedor Web. Es decir, los servlets de Java y las páginas JSP también pueden acceder a los componentes EJB a través del protocolo RMI-IIOP del mismo modo que los clientes de aplicación. La última categoría es otro EJB ejecutado dentro del contenedor EJB. Estos se comunican a través de llamadas estándar de método Java por medio de de una interfaz local (esta característica es una novedad de J2EE 1.3).

En los dos casos, los clientes acceden a los componentes de la aplicación a través de sus respectivos contenedores. Los clientes Web acceden a las páginas JSP y a los servlets Java a través del contenedor Web, y los clientes EJB acceden a los componentes EJB a través del contenedor EJB.

Arquitectura de contenedor

Después de haber analizado las partes integrantes de J2EE, volvamos a nuestro análisis de la arquitectura y estudiemos la arquitectura de un contenedor J2EE:



En esta arquitectura, como desarrolladores de programas estamos obligados a proporcionar los siguientes elementos:

- **Componentes de aplicación**
  Como ya hemos visto en la sección anterior, los componentes de aplicación incluyen servlets, páginas JSP, EJB, etc. En J2EE, los componentes de aplicación pueden ser incluidos en ficheros en archivo.
- **Descriptores de implementación**
  Un descriptor de implementación es un archivo XML que describe los componentes de aplicación. También incluye información adicional requerida por contenedores para gestionar de un modo efectivo los componentes de aplicación.

El resto de esta figura forma el contenedor. La arquitectura de un contenedor puede dividirse en cuatro partes:

- **Contrato de contenedores**
  Un conjunto de API especificados por el contenedor, que sus componentes de aplicación están obligados a ampliar o implementar.
- **API de servicio de contenedor**
  Los servicios adicionales proporcionados por el contenedor, que comúnmente son requeridos por todas las aplicaciones en el contenedor.
- **Servicios declarativos**
  Servicios que el contenedor introduce en sus aplicaciones, basados en la descripción de implementación proporcionada por cada componente de aplicación, como la seguridad, las transacciones, etc.
- **Otros servicios de contenedores**
  Otros servicios de periodo de ejecución, relacionados con ciclos de vida de los componentes, reserva de recursos, recolección de basura, etc.

Examinemos a continuación cada uno de los elementos anteriores más detenidamente.

Contratos de componentes

Como ya hemos mencionado anteriormente, el propósito básico del contenedor en la arquitectura J2EE es proporcionar un periodo de ejecución para los componentes de aplicación. Es decir, se crean ejemplares de los componentes de aplicación y se invocan en el JVM del contenedor. Esto convierte al contenedor en responsable de la gestión del ciclo de vida de los componentes de la aplicación. Sin embargo, si están llamados a ser gestionables en el periodo de ejecución del contenedor, están obligados a guiarse por ciertos **contratos** especificados por el contenedor. En otras palabras, las aplicaciones deben ser desarrolladas de acuerdo con un marco de operación definido.

Para comprender mejor este aspecto, consideremos un applet Java. Normalmente, un applet es descargado por el navegador, instanciado e inicializado en el JVM del navegador. Es decir, el applet vive en el periodo de ejecución proporcionado por el JVM del navegador.

Sin embargo, para que el contenedor tenga la capacidad de crear, inicializar, e invocar métodos sobre componentes de aplicación, es necesario que los componentes de aplicación implementen o amplíen ciertas interfaces o clases Java. Por ejemplo, si tomamos de nuevo el ejemplo del applet, un applet Java está obligado a ampliar la clase `java.applet.Applet` especificada por el JDK. El JVM del navegador espera que los applets amplíen esta clase. Esto permite al JVM del navegador llamar a los métodos `init()`, `start()`, `stop()` y `destroy()` de sus applets. Estos métodos controlan el ciclo de vida de un applet; si su applet no amplía la clase `java.applet.Applet`, el JVM del navegador no cuenta con los medios para llamar a estos métodos y controlar su ciclo de vida.

En J2EE, todos los componentes de aplicación son instanciados e inicializados en el JVM del contenedor. Además, debido a que los componentes de aplicación J2EE son siempre remotos al cliente, los clientes no pueden invocar métodos directamente en estos componentes; de hecho, los clientes realizan solicitudes al servidor de las aplicaciones y es el contenedor el que realmente invoca los métodos. Puesto que el proceso del contenedor es el único punto de entrada a los componentes de la aplicación, es necesario que todos los componentes de la aplicación sigan el contrato especificado por el contenedor. En J2EE, este contrato está concretado en interfaces y clases que las clases del usuario implementan o amplían, con normas adicionales que la definición del componente debe seguir.

Existe una consecuencia importante de este contrato de componente. Todos los componentes de aplicación J2EE son **gestionados**. La gestión incluye la ubicación, instanciación, inicialización, invocación de servicio y eliminación de componentes del servicio. Estos aspectos del ciclo de vida del componente están bajo la responsabilidad del contenedor y la aplicación no tiene control directo.

Analicemos a continuación los diversos contratos especificados por el J2EE.

En el caso de los contenedores Web, los componentes de aplicación Web deben seguir los API de Java Servlet y de JSP. En este modelo, todos los servlets Java deben ampliar la clase `javax.Servlet.http.HttpServlet`, e implementar ciertos métodos de esta clase tales como `doGet()`, `doPost()`, etc. Del mismo modo, al ser compilado, las clases correspondientes a las páginas JSP amplían la clase `javax.Servlet.jsp.HttpJspPage`.

En el caso de los contenedores EJB, los beans de sesión y de entidad de empresa deben tener interfaces `javax.ejb.EJBHome` y `javax.ejb.EJBObject` especificados, a la vez que implementan una interfaz `javax.ejb.SessionBean` o `javax.ejb.EntityBean`. Del mismo modo, los beans orientados a mensajes deben implementar las interfaces `javax.ejb.MessageDrivenBean` y `javax.jmx.MessageListener`. Como verá en el capítulo 14, las especificaciones e implementaciones de componente también deben seguir ciertas reglas. Aparte de las interfaces, todas estas reglas forman parte del contrato de componente para componentes EJB.

API de servicio de contenedor

Como hemos visto anteriormente, la plataforma J2EE define un conjunto de extensiones estándar de Java que cada plataforma debe mantener. Los contenedores J2EE proporcionan una abstracción a nivel de servicio de los API. Como resultado, el usuario puede acceder a los API de servicio tales como JDBC, JTS, JNDI, JMS, etc. dentro del contenedor, como si el contenedor subyacente los implementara.

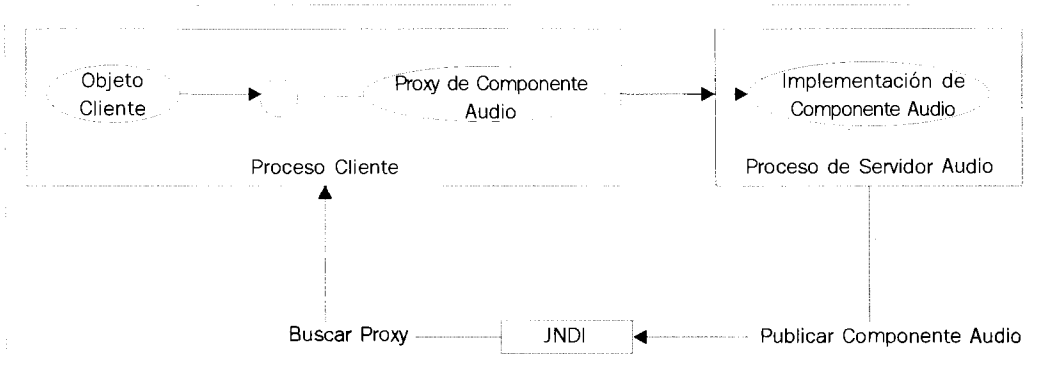
> **Un contenedor en la arquitectura de J2EE proporciona una visión consolidada de varios API de empresa especificados en la plataforma J2EE.**

En J2EE, los componentes de aplicación pueden acceder a estos API a través de objetos apropiados creados y publicados en el servicio o implementación JNDI. Por ejemplo, cuando queremos utilizar JMS en nuestros componentes de aplicación, configuraremos nuestra plataforma J2EE para crear factorías de conexión JMS, colas de mensajes y apartados, y publicar estos objetos en el servicio JNDI. Entonces nuestras aplicaciones pueden consultar el JNDI, obtener referencias para estos objetos, e invocar métodos. En este proceso, no importa si la plataforma J2EE tiene una implementación JMS incorporada o si está utilizando una solución de software intermediario de mensajes de tercero.

Consideremos de nuevo los applet de Java. Por ejemplo, si queremos reproducir un archivo de audio desde un applet, llamaríamos al método `play()` definido en el superclase (`java.applet.Applet`), con un URL apuntando al archivo de audio. En este caso, la funcionalidad para reproducir archivosde audio es implementada por la superclase. Éste es uno de los enfoques para la reutilización de código. Una mejor alternativa a este enfoque es delegar esta funcionalidad en un componente común. Es necesario que un

componente de este tipo no sea una parte de la misma jerarquía de herencia. Lo único que se requiere es una referencia a un objeto que implemente esta funcionalidad. Una vez que nuestra aplicación pueda obtener una referencia de este objeto, podemos delegar la funcionalidad "play" a ese objeto. Una importante ventaja de este enfoque es que permite que otros objetos procedentes de más de una jerarquía de herencia accedan la funcionalidad "play".

En el caso de aplicaciones distribuidas es más complicado, ya que los servicios son remotos. Sólo como analogía, consideremos el mismo componente de audio ejecutado como un servidor distinto. ¿Cómo obtenemos una referencia a este componente? La solución es permitir a la plataforma J2EE crear el componente de audio y publicar su nombre en un servicio de designación (es decir, en JNDI) disponible para su aplicación. Esto proporciona un método simplificado de acceso a los API de servicio. También nos permite acoplar diferentes implementaciones de estos servicios sin alterar las aplicaciones utilizando los servicios. El siguiente esquema ilustra esta posibilidad:



Observe la libre conexión entre la implementación y el cliente. Siempre que la interfaz expuesta por el proxy se mantenga sin alterar, la implementación puede ser modificada sin afectar el código cliente. Ésta es la forma más adecuada de abstracción en el lado servidor, ya que no sólo utiliza delegación en lugar de herencia, sino que también reduce el acoplamiento.

Todos los API de servicio J2EE utilizan el enfoque anterior para proporcionar servicios a las aplicaciones.

Las siguientes son las implicaciones clave de este enfoque:

- Como único estándar que puede establecerse en una variedad de sistemas de bases de datos existentes, sistemas de procesamiento de transacciones, servicios de designación y directorio, etc., el API de servicio elimina la heterogeneidad inherente relacionada con la agrupación de estas tecnologías en nuestras aplicaciones.
- En J2EE, estos servicios también están integrados estrechamente en el modelo de programación. En próximos capítulos, veremos cómo acceder a estos API sin interrupciones a partir de nuestros componentes de aplicación y clientes.
- La plataforma J2EE también especifica un mecanismo uniforme para acceder a estos servicios.

Servicios declarativos

Uno de los aspectos importantes de la arquitectura de J2EE es su capacidad para introducir servicios para componentes de aplicación. Está basado en declaraciones especificadas fuera de nuestros componentes de aplicación. La arquitectura de J2EE proporciona medios sencillos de especificar tales declaraciones. Se denominan **descriptores de implementación**.

Un descriptor de implementación define el contrato entre el contenedor y el componente. Como desarrolladores aplicaciones, debemos especificar un descriptor de implementación para cada grupo de componentes de aplicación.

Por ejemplo, un conjunto de componentes EJB puede ser descrito conjuntamente en un único archivo de descriptor de implementación. De modo similar, en el caso de los contenedores Web, cada aplicación Web debe tener un descriptor de implementación especificado.

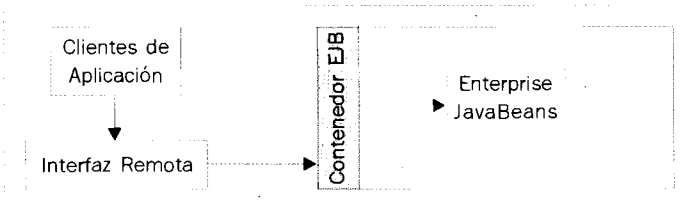
Dependiendo del tipo de componente, ciertos tipos de servicios (tales como transacciones, seguridad, etc) pueden ser especificados en el descriptor de implementación. El propósito de este enfoque es minimizar la programación de la aplicación requerida con el fin de hacer uso de tales servicios.

El método estándar de invocar servicios es mediante **invocación explícita**. Por ejemplo, para implementar transacciones para el acceso a bases de datos, podemos empezar por programar una transacción antes de acceder a la base datos y realizar o deshacer la transacción una vez completados los métodos de empresa. En el caso de invocación declarativa, nuestros componentes de aplicación no necesitan explícitamente empezar y detener transacciones. En su lugar, podemos especificar en el descriptor de implementación que nuestros métodos de empresa deben ser invocados en una nueva transacción. Basándose en esta información, el contenedor puede iniciar automáticamente una transacción siempre que los métodos de empresa de las aplicaciones del usuario sean invocados.

En términos sencillos, un servicio declarativo es un servicio o acción realizado por el contenedor en nuestro nombre.

¿Cómo funciona este enfoque? Los contenedores J2EE son, por naturaleza, distribuidos y los componentes de la aplicación son remotos a los clientes. Por consiguiente, las solicitudes a los componentes de aplicación y las respuestas procedentes de los componentes de aplicación tienen lugar en los límites del proceso, plataforma y red.

Además, puesto que los componentes de aplicación son mantenidos en el periodo de ejecución del contenedor, el proceso del contenedor es responsable de recibir las solicitudes y de delegarlas a los componentes de aplicación adecuados. Por ejemplo, en el caso del contenedor EJB, el contenedor recibe todas las solicitudes del cliente y las delega a los objetos EJB pertinentes implemtados en el contenedor. De un modo similar, en el caso de los contenedores Web, el contenedor Web recibe las solicitudes http y las delega en los servlets y en las páginas Web. El siguiente diagrama representa una visión simplificada de esta invocación. En esta figura, la interfaz remota es lo que utilizan los clientes para comunicarse con el EJB en el contenedor. Observe que el proceso del contenedor maneja todas las solicitudes y respuestas intercambiadas por los componentes de la aplicación:



Este enfoque proporciona al contenedor la capacidad para **interponer** un nuevo servicio antes de transferir una solicitud al componente de aplicación. En el caso de transacciones declarativas, el contenedor puede comenzar la transacción antes de delegar la solicitud entrante en la implicación de método de empresa y terminar la transacción tan pronto como vuelta al método.

¿Cuál es la ventaja de este enfoque? Podemos interponer nuevos servicios sin cambiar el componente de aplicación. Más específicamente, esta función nos permite posponer decisiones sobre tales servicios hasta el periodo de ejecución, en lugar del tiempo de diseño. En otras palabras, el contenedor puede reforzar selectivamente nuestros componentes basados en el descriptor de desarrollo.

Para contenedores EJB, los servicios declarativos incluyen transacciones y seguridad. Ambos pueden ser especificados en descriptores de implementación. En el caso de los contenedores Web, podemos especificar las funciones de seguridad requeridas para acceder a componentes en las aplicaciones Web.

Otros servicios de contenedor

Los siguientes son algunos de los servicios de periodo de ejecución que proporcionan los contenedores:

- **Gestión de ciclo de vida de los componentes de aplicación**
  Esto conlleva la creación de nuevo ejemplares de componentes de aplicación y su reserva o destrucción cuando los ejemplares ya no resulten necesarios.
- **Reserva de recursos**
  Los contenedores pueden implementar opcionalmente reserva de recursos, como reserva de objetos o reserva de conexiones.
- **Poblar el espacio de nombre de JNDI basado en el despliegue de nombres asociados a los componentes EJB**
  Esta información es proporcionada normalmente en el tiempo de despliegue. Trataremos la implementación en la siguiente sección.
- **Poblar el espacio de nombre JNDI con objetos necesarios para utilizar los API de servicio del contenedor**
  Algunos de los objetos incluyen objetos fuentes de datos para el acceso a bases de datos, colas y factorías de conexión de apartados para obtener conexiones con JMS y objetos de transacción de usuario para controlar transacciones programáticamente.
- **Agrupamiento**
  En J2EE, los contenedores pueden ser distribuíbles. Un contenedor distribuíble consiste en un número de JVM ejecutados en una o más máquinas anfitrionas. En esta configuración, los componentes de aplicación pueden ser desplegados en una serie de JVM. Dependiendo del tipo de estrategia de equilibrio de carga, el contenedor puede distribuir la carga de solicitudes entrantes a una de estas JVM. La agrupación es esencial para reforzar la reajustabilidad y disponibilidad de las aplicaciones.

Ahora que ya hemos cubierto la arquitectura de las aplicaciones J2EE, examinemos más de cerca algunos de las diversas tecnologías incluidas en la plataforma J2EE.

Tecnologías J2EE

Después de haber examinado toda la arquitectura de la plataforma J2EE, queremos ahora abarcar la colección de tecnologías que proporcionan los mecanismos que necesitamos para construir amplias y distribuidas aplicaciones de empresa. Esta gran colección, formada por tecnologías bastante dispares, puede ser clasificada según su uso:

- **Las tecnologías de componentes**

Estas tecnologías son utilizadas para albergar la mayor parte de la aplicación: la lógica de empresa. Existen tres tipos de componentes: páginas JSP, servlets y Enterprise JavaBeans; examinaremos cada uno de ellos a continuación.

- **Las tecnologías de servicios**
  Estas tecnologías proporcionan los componentes de la aplicación con servicios reforzados para que funcionen de un modo eficiente.
- **Las tecnologías de comunicación**
  Estas tecnologías, que son en su mayoría transparentes para el programador de la aplicación, proporcionan mecanismo para la comunicación entre diferentes partes de la aplicación, ya sean locales o remotas.

Examinemos a continuación la clasificación de API J2EE y sus tecnologías asociadas.

Tecnologías de componentes

Con cualquier aplicación, el elemento más importante es el modelado de la lógica de empresa necesaria mediante el uso de componentes o de unidades reusables en el nivel de aplicación. Anteriormente en este capítulo, hemos descrito un contenedor como un elemento que alberga el periodo de ejecución para componentes de aplicación, por lo que, aunque el contenedor pueda proveer muchos de los servicios y gran parte de la infraestructura de comunicación, es en última instancia responsabilidad del agente de desarrollo crear los componentes de aplicación. Sin embargo, estos componentes dependerán de su contenedor para muchos servicios, tales como la gestión del ciclo de vida, el enhebrado (threading) y seguridad. Esto nos permite concentrarnos en proporcionar la funcionalidad de empresa indispensable sin entrar en detalles de semántica de bajo nivel (nivel contenedor).

La plataforma J2EE ofrece tres tecnologías para el desarrollo de componentes.

> **Un aspecto que debe quedar claro es que la plataforma J2EE no especifica que una aplicación necesite hacer uso de los tres tipos de tecnologías de componentes; en muchos casos, utilizar EJB puede producir una sobrecarga.**

Componentes Web

Estos componentes pueden clasificarse como cualquier otro componente que responda a una solicitud HHTP. Se puede realizar otra distinción más profunda basada en el contenedor anfitrión para los componentes de la aplicación. Como ya hemos visto en este capítulo, los dos contenedores básicos de lado del servidor son el contenedor Web y el contenedor EJB.

Servlets

Los **servlets** son programas de lado del servidor que permiten a la lógica de aplicación ser incorporada al proceso solicitud-respuesta HTTP. Los servlets proporcionan un medio de ampliar la funcionalidad del servidor Web para admitir contenido dinámico en HTML, XML, u otros lenguajes Web. Con el lanzamiento de J2EE 1.3, la especificación de los Servlets ha alcanzado la versión 2.3.

Trabajaremos con servlets en los Capítulos del 5 al 9.

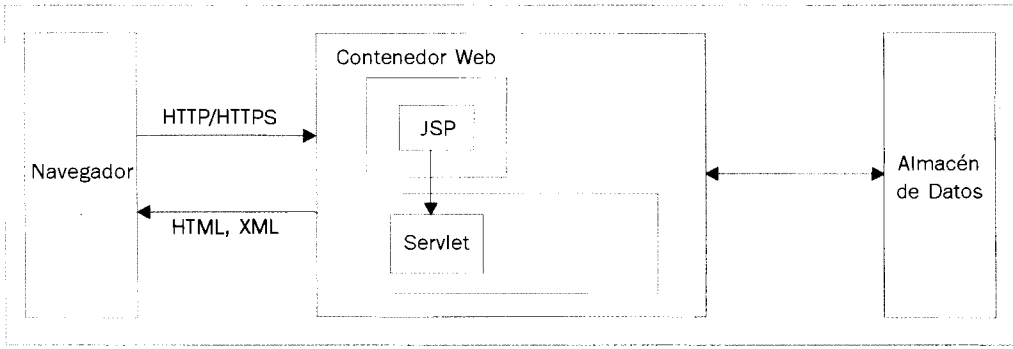
Páginas JavaServer

Las **JavaServer Pages (JSP)** proporcionan un modo de incorporar componentes en una página y de hacer que realicen su función para generar la página que será enviada el cliente eventualmente. Una página JSP

contiene componentes HTML, código Java y JavaBean. Las páginas JSP son de hecho una extensión del modelo de programación del Servlet. Cuando un usuario solicita una página JSP, el contenedor Web compila la página JSP en un Servlet. El contenedor Web invoca al Servlet y devuelve el contenido resultante al navegador Web. Una vez que el Servlet ha sido compilado a partir de la página JSP, el contenedor Web puede simplemente devolver el servlet sin tener que recopilar cada vez. De este modo, las páginas JSP proporcionan un mecanismo de ensamblaje de páginas dinámico que se beneficia de las diversas ventajas de la plataforma Java.

En comparación con los servlets, que son puro código Java, las páginas JSP son mero documentos basados en texto hasta que el contenedor Web los compila en los correspondientes servlets. Esto permite una separación más precisa entre la lógica de aplicación y la lógica de presentación. A su vez, permite a los desarrolladores de programas de aplicación centrarse en materias de empresa y a los diseñadores Web concentrarse en la presentación. Con la versión actual de J2EE, la especificación JSP llega a la versión 1.2. Los Capítulos 10 a 12 examinan en detalle las páginas JSP.

Una arquitectura típica para una aplicación Web que incluya páginas JSP tendría la siguiente estructura:



Componentes Enterprise JavaBean

Enterprise JavaBean (EJB) 2.0 es un modelo de componente distribuido para el desarrollo de componentes seguros, reajustables, transaccionales y multi-usuario. Para expresarlo de un modo sencillo, los EJB son (idealmente) unidades de software reusables que contienen lógica de empresa. Del mismo modo que las páginas JSP permiten la separación entre la lógica de aplicación y la lógica de presentación, los EJB permiten la separación entre la lógica de aplicación y los servicios a escala del sistema, permitiendo así al agente de desarrollo concentrarse en los temas de dominio de empresa y no programar el sistema. Estos objetos de empresa Enterprise Bean adoptan tres formas básicas (una vez más, no es necesario implementarlas todas): **beans de sesión, beans de entidad y beans dirigidos por mensaje.** Examinaremos los EJB en acción en los Capítulos 14 al 19.

Beans de sesión

Los beans de sesión se presentan en dos tipos. Un **bean de sesión de estado** es un objeto transitorio utilizado para representar la interacción de cliente con el sistema; ejecuta la solicitud del cliente en la aplicación, accediendo a la base de datos, etc. y cuando las operaciones del cliente están completas es destruido (es decir, existe mientras dura la sesión del cliente). Un ejemplo de este tipo de bean es un cliente de aplicación enviando una serie de solicitudes a una aplicación para llevar a cabo un proceso empresarial. En tales casos, un bean de sesión de estado puede registrar en estado de la interacción entre el cliente y la aplicación. La alternativa, un **bean de sesión sin estado**, no mantiene estado alguno entre las solicitudes del cliente. Generalmente, este tipo de bean de sesión es utilizado para implementar un servicio específico que no requiere el estado del cliente, por ejemplo, una simple actualización de base de datos.

Beans de entidad

Un **bean de entidad**, por otro lado, es un objeto persistente que modela los datos contenidos en el almacén de datos, es decir, es un objeto envoltorio para los datos. Por ejemplo, datos sobre una pedido de compra pueden ser modelados utilizando un bean de entidad PurchaseOrder que presente una visión agregada de todos los datos relacionados con dicho pedido. En comparación con los beans de sesión, que pueden ser utilizados por cualquier cliente, muchos clientes pueden acceder a los beans de entidad pero deben mantener una única identidad a través de una clave primaria. De hecho, en la arquitectura del contenedor J2EE podemos elegir entre tener el estado persistente del bean de entidad gestionado por el contenedor o implementarlo "a mano" en el mismo bean.

Beans dirigidos por mensaje

Los beans dirigidos por mensaje son una clase especial de EJB que no están destinados a la invocación directa de cliente. El objetivo de los beans dirigidos por mensaje es procesar los mensajes recibidos vía JMS. Los beans dirigidos por mensaje complementan la naturaleza asíncrona de JMS proporcionando un medio de procesamiento de mensajes dentro del contenedor EJB. Cuando un cliente de aplicación o una aplicación envía un mensaje vía JMS, el contenedor invoca al bean dirigido por mensaje adecuado para procesar el mensaje.

XML

Como ya hemos mencionado en nuestro análisis anterior, J2EE 1.3 incluye el API JAXP. Éste sirve para abstraer el parseado XML y los API de traducción XML. Además, hay otros API de Java relacionados con XML que están siendo desarrollados en la actualidad. Puede encontrar un resumen sobre el tema y más detalles en la dirección http://java.sun.com/xml.

XML (**Extensive Markup Language**) influye en cómo percibimos, procesamos, transportamos y gestionamos datos. XML es un lenguaje autodescriptivo con el que puede enviar datos así como metadatos. En los sistemas de empresa de hoy día, XML es utilizado para representar datos de empresa tanto en el interior como a través de aplicaciones.

Uno de los puntos fuertes de XML es la común utilización de estos "vocabularios", todos ellos con la misma sintaxis básica, los mismos analizadores y otras herramientas. Los vocabularios XML compartidos proporcionan más fácilmente documentos de búsqueda y bases de datos, y un modo de intercambiar información entre diferentes organizaciones y aplicaciones informáticas.

Sin embargo, XML puede ser mucho más que un mecanismo de descripción:

- **Transformación de XML**
  La transformación permite al programador copiar un documento XML en una determinada forma basándose en una serie de reglas. Las transformaciones son utilizadas para traducir vocabularios XML similares así como traducir documentos XML en otros formatos de archivos de base textual como valores delimitados por comas. Éste es el tipo de representación estándar de datos en una empresa.
- **XML y bases de datos**
  Aunque el modelo de datos XML es inherentemente jerárquico mientras que las bases de datos son esencialmente relacionales (creando algunas dificultades de copia), proporciona un mecanismo de integración de datos existentes en nuevos sistemas. Muchos distribuidores de bases de datos están añadiendo en la actualidad funione XML en sus sistemas al reconocer que los programadores necesitan recursos para interfacear XML y bases de datos.
- **Comunicación entre servidores**

Las aplicaciones complejas de empresa utilizan a menudo software de servidor distintos ejecutados y distribuidos en diversas tecnologías de procesamiento. XML proporciona una capa de abstracción para integrar estos sistemas similares. CML puede obtenerse de un servidor, ser manipulado y después enviado a otro servidor de tal modo que pueda comprender la solicitud.

Aunque no abarcaremos explícitamente XML en este libro, aparecerá cuando creemos varios descriptores de implementación J2EE. Para más información sobre el uso de XML con Java, puede consultar *Profesional Java y XML* y *Referencia para Programadores Java y XML*.

Tecnologías de servicio

Como ya hemos visto, algunos de los servicios de J2EE para componentes de aplicación son gestionados por los mismo contenedores, permitiendo así al agente de desarrollo concentrarse en la lógica de empresa. Sin embargo, en ocasiones los desarrolladores de programas considerarán necesario invocar ellos mismos programáticamente algunos servicios utilizando algunas de las diversas tecnologías de servicio.

JDBC

Aunque todos los accesos a los datos deberían ser accesibles a través de un único API estándar de la arquitectura de Conector en el futuro, la conectividad de bases de datos es probablemente uno de los servicios clave que los desarrolladores de programas implementan en su componente de aplicación.

El API **JDBC** (siglas no oficiales para Java Database Connectivity) proporciona al agente de desarrollo la capacidad de conectar con sistemas de bases de datos relacionales. Como veremos en el capítulo 4, permite las consultas transaccionales, la recuperación y la manipulación de datos desde una base de datos que se ajuste a JDBC. Por el momento, merece la pena destacar que J2EE añade una extensión al núcleo API JDBC (que está incorporado en J2SE) para ofrecer características avanzadas, como reserva de conexiones y transacciones distribuidas.

Java Transaction API y Servicio

El API **Java Transaction (JTA)** es un medio para trabajar con transacciones y con transacciones especialmente distribuidas independientes de la implementación del gestor de transacciones (el **Java Transaction Service (JTS)**). En la plataforma J2EE, las transacciones distribuidas están consideradas generalmente como transacciones controladas por contenedor; por ello, nosotros, como desarrolladores de programas, no deberíamos estar preocupados por las transacciones entre componentes. Sin embargo, el modelo de transacción de J2EE es todavía algo limitado y en ocasiones puede que tengamos que realizar el trabajo duro.

JNDI

La función del API **Java Naming and Directory Interface (JNDI)** en la plataforma J2EE es doble:

- En primer lugar, proporciona los medios para ejecutar operaciones estándar en un recurso de servicio directorio como *LDAP*, *Novell Directory Services* o *Netscape Directory Services*.
- En segundo lugar, una aplicación J2EE utiliza JNDI para buscar interfaces utilizadas para crear, entre otras cosas, EJB y conexiones JDBC.

En el próximo capítulo, veremos cómo utilizar JNDI para acceder a un recurso de servicio directorio.

JMS

En el entorno de empresa, puede que los diversos componentes distribuidos no estén siempre en constante contacto entre sí. Por lo tanto, es necesario que haya algún mecanismo de envío de datos de modo asíncrono. El **Java Message Service (JMS)** proporciona exactamente esta funcionalidad para enviar y recibir mensajes mediante el uso de software intermediario orientado al mensaje.

JavaMail

JavaMail es un API que puede ser utilizado para abstraer instalaciones para enviar y recibir e-mail. JavaMail se ajusta a los protocolos de correo más utilizados en Internet como IMAP4, POP3 y SMTP, pero en comparación con JMS es más lento y menos fiable.

Java Conector Architecture

Java Connector Architecture (JCA), presentado en la versión 1.3, es un medio estandarizado por el que las aplicaciones J2EE pueden acceder a una variedad de aplicaciones legadas, normalmente sistemas Enterprise Resource Planning (ERP) como SAP R/3 y PeopleSoft. La especificación del Conector define una sencilla arquitectura en la que los distribuidores de servidores de aplicación J2EE y sistemas legados colaboran para producir componentes "plug-and-play" que nos permitan acceder al sistema legado sin tener que tener conocimientos muy específicos sobre cómo trabajar con él.

JAAS

El servicio **Java Authentication and Authorization Service** proporciona un medio para conceder permisos basado en quién ejecuta el código. JAAS utiliza una arquitectura conectable de módulos de autentificación de modo que el usuario pueda introducir módulos basados en diferentes implementaciones de autentificación, como Kerberos o PKI (Public Key Infrastructure).

Tecnologías de comunicación

El último grupo de tecnologías está formado por las tecnologías que proporcionan los medios para los diversos componentes y servicios de la aplicación J2EE para su comunicación entre sí; una aplicación distribuida sería bastante ineficaz si estas tecnologías no proporcionaran el "pegamento" para unir todos los elementos.

Protocolos de Internet

Mientras hablamos de aplicaciones de n-niveles en este libro, nuestro cliente será muy a menudo un navegador potencialmente situado en cualquier lugar del mundo. Las solicitudes de un cliente y las respuestas de un servidor se comunican a través de tres protocolos principales.

HTTP

HTTP o **Hypertext Transfer Protocol** es un protocolo genérico, sin estado, de nivel aplicación, que tiene muchos usos más allá de sus sencillas capacidades como hipertexto. Funciona sobre una base solicitud/respuesta. Un cliente envía una solicitud al servidor en forma de método de solicitud, URI (Uniform Resource Identifier) y versión de protocolo, seguido por un mensaje tipo MIME que contiene modificadores de solicitud, información del cliente y posible contenido de cuerpo en una conexión con un

servidor. El servidor a su vez responde con una línea de estado seguida por un mensaje de tipo MIME que contiene información del servidor, información meta de entidad y posible contenido entidad-cuerpo.

TCP/IP

TCP (Transmisión Control Protocol) sobre **IP (Internet Protocol)** son realmente dos protocolos distintos, que se combinan normalmente en una única entidad. IP es el protocolo responsable de asegurar que los datos son recibidos en ambos extremos de la comunicación a través de Internet. Cuando tecleamos la dirección de un sitio Web en nuestro navegador, el protocolo IP certifica nuestras solicitudes y qu el cumplimiento de esas solicitudes llegan hasta su destino. Para mayor eficiencia, los datos que son enviados y recibidos entre un cliente y un servidor Web son descompuestos en varias piezas o paquetes. No todos estos paquetes tienen que tomar la misma ruta al ser enviados entre el cliente y el servidor Web. TCP es el protocolo que registra los paquetes y certifica que son reunidos en el mismo orden en el que fueron divididos y sin errores. Por lo tanto, TCP y IP trabajan juntos para mover datos en Internet. Por este motivo, casi siempre veremos estos dos protocolos combinados en TCP/IP.

SSL

Secure Sockets Layer (SSL) utiliza la criptografía para cifrar el flujo de información entre el cliente y el servidor. También proporciona un medio para que ambas partes puedan autentificarse entre sí. Secure http (HTTPS) se distingue normalmente del http corriente sin cifrar al ser servido en diferente número de puerto (443, por defecto).

Protocolos de objeto remoto

En aplicaciones donde los componentes se distribuyen normalmente en diferentes niveles y servidores, se requieren algunos mecanismos para utilizar los componentes de forma remota, preferiblemente dejando que el cliente ignore que el componente no es local respecto a sí mismo.

RMI y RMI-IIOP

Remote Meted Invocation (RMI) es uno de los mecanismos primarios en las aplicaciones de objetos distribuidos. Nos permite utilizar interfaces para definir objetos remotos. Podemos entonces llamar a métodos en estos objetos remotos como si fueran locales. Los mecanismos de transporte utilizados dependen de cada implementación. Por ejemplo, Sun utiliza Java Remote Meted Protocol (JRMP) sobre TCP/IP, pero otras implementaciones, como BEA WebLogic, por ejemplo, tienen su propio protocolo.

RMI-IIOP es una extensión de RMI pero en IIOP (Inter.-ORB Protocol), que nos permite definir una interfaz remota a cualquier objeto remoto que pueda ser implementado en cualquier lenguaje que se ajuste a la asociación OMG y ORB. En el capítulo 3, examinaremos con detenimiento cómo escribir aplicaciones distribuidas con RMI.

JavaIDL

Mediante el uso de **JavaIDL**, un cliente Java puede invocar llamadas de método en objetos CORBA. Estos objetos CORBA no necesitan ser escritos en Java sino implementar meramente una interfaz IDL definida.

Desarrollo de aplicaciones J2EE

La especificación J2EE concreta los siguientes pasos en el proceso de desarrollo y despliegue de aplicaciones.

1. **Desarrollo de componentes de aplicación**

Durante este paso modelamos las reglas de la empresa en forma de componentes de aplicación.

2. **Composición de componentes de aplicación en módulos**
   En este paso, los componentes de aplicación son empaquetados en módulos. Esta fase también conlleva proporcionar descriptores de implementación para cada módulo.

3. **Composición de módulos en aplicaciones**
   Este paso integra múltiples módulos en aplicaciones J2EE. Requiere ensamblar uno o más módulos en una aplicación J2EE y facilitarle los archivos de descriptor.

4. **Despliegue de aplicación**
   En el paso final, la aplicación empaquetada es realmente desplegada e instalada en el servidor(es) de aplicación de la plataforma J2EE.

La plataforma J2EE especifica múltiples niveles para empaquetado, adaptación, e instalación. El empaquetado es el proceso de componer aplicaciones a partir de componentes de aplicaciones. J2EE especifica un plan de empaquetado de tres niveles para componer componentes en aplicaciones. Estos son: componentes de aplicación, módulos y aplicaciones.

¿Cuál es el propósito de estas fases para el desarrollo e implementación de aplicaciones? En J2EE, la arquitectura de aplicaciones empieza con la descomposición de la aplicación en módulos y de los módulos en componentes de aplicación. Éste es un proceso descendente hacia la construcción de los bloques de construcción más pequeños. La noción de empaquetado y despliegue de aplicación en niveles múltiples intenta conseguir este propósito. En este proceso, componemos componentes de aplicación de grano fino en módulos y después los módulos en aplicaciones.

Funciones de desarrollo e implementación de aplicaciones

Antes de pasar a examinar detenidamente el proceso de desarrollo e implementación, desviémonos momentáneamente para analizar quién debería hacer qué. La especificación J2EE, además de definir un proceso, también define una serie de funciones en el desarrollo de aplicaciones J2EE:

- **Proveedor del producto J2EE**
  El Proveedor del producto J2EE proporciona la plataforma base J2EE sobre la cual desarrollamos nuestras aplicaciones. Éste será nuestro distribuidor de servidor que implementa la arquitectura del contenedor y los API son definidos por la especificación J2EE.

- **Proveedor de componente de aplicación**
  Es esencialmente el agente de desarrollo de la aplicación quien crea la funcionalidad de la aplicación, aunque es posible subdividir la función en áreas específicas de conocimientos técnicos, por ejemplo, agente de desarrollo de Web, agente de desarrollo de EJB, etc.

- **Ensamblador de aplicaciones**
  Como veremos en breve, el ensamblador de aplicaciones toma los componentes de aplicación y los empaqueta a través de una serie de módulos y archivos de descriptores de modo que puedan ser implementados a los servidores de producción.

- **Desplegador**
  El desplegador toma la aplicación empaquetada, la instala y la configura para el entorno particular de operación en el que será ejecutada la aplicación.

- **Administrador del sistema**
  El Administrador del sistema es responsable de mantener y administrar la aplicación una vez que ya ha sido implementada.
- **Proveedor de herramientas**

  El Proveedor de Herramientas proporciona las herramientas que pueden ser de uso en el desarrollo e implementación de los componentes de aplicación.

Observe que esta clasificación de funciones puede ajustarse o no a su aplicación específica y esta clasificación sólo debería considerarse una configuración típica que puede servir como guía.

Desarrollo de componentes de aplicación

En J2EE, el proceso de desarrollo comienza con el diseño y el desarrollo de componentes de aplicación. La mayor parte de este libro versará sobre este tema, por lo que no es necesario que profundicemos en ello aquí.

Composición de componentes de aplicación en módulos

Un módulo es utilizado para empaquetar uno o más componentes de aplicación relacionados del mismo tipo. Además de los componentes de aplicación, cada módulo incluye un descriptor de implementación que describe la estructura del módulo. Hay tres tipos de módulos en J2EE:

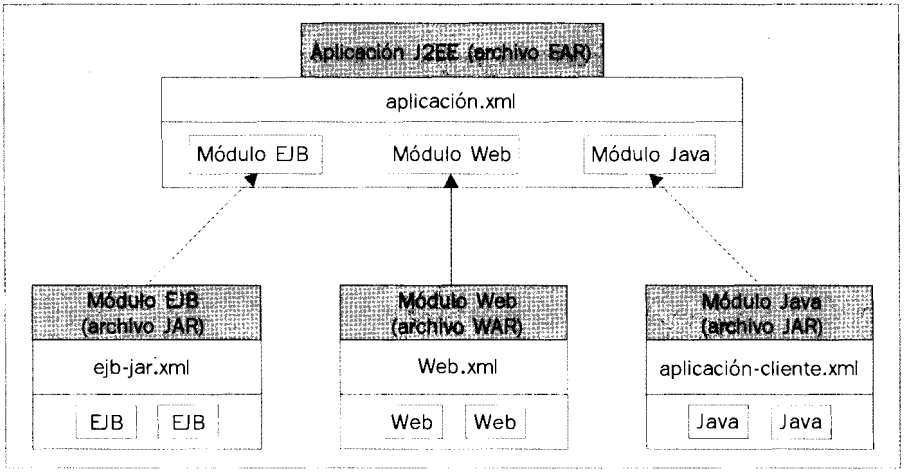
- **Módulos Web**
  Un módulo Web es una unidad desplegable compuesta de Servlets de Java, páginas JSP, bibiliotecas de etiquetas, archivos JAR de biblioteca, documentos HTML/XML y otros recursos públicos tales como imágenes, archivos de clase applet, etc. Un módulo Web es empaquetado en un **archivo Web Archive**, también conocido como archivo **WAR**. Un archivo WAR es similar a un archivo JAR; la diferencia radica en que el archivo WAR contiene un directorio `WEB-INF` con la descripción de implementación contenida en un archivo `Web.xml`. Conoceremos más detalles de los módulos Web y su empaquetado en próximos capítulos.
- **Módulos EJB**
  Un módulo EJB es una unidad desplegable comupuesta de un EJB, asociado a archivos JAR de biblioteca y recursos. Los módulos EJB están empaquetados en archivos JAR, con un descriptor de implementación (`ejb-jar.xml`) en el directorio `META-INF` del archivo JAR.
- **Módulos Java**
  Un módulo Java es un grupo de clases de cliente Java empaquetados en archivos JAR. El descriptor de implementación para un módulo Java es un archivo `application-client.xml`.

Composición de módulos en aplicaciones

El nivel más alto de empaquetado está en forma de aplicaciones. Cada aplicación J2EE es una unidad independiente de código dentro de un cajón virtual propio. Normalmente, los contenedores J2EE cargan cada aplicación utilizando un cargador de clase diferente de modo que cada aplicación queda aislada del

resto. El objetivo de este aislamiento es permitir a las múltiples aplicaciones J2EE coexistir en un contenedor J2EE.

Una aplicación J2EE consiste en uno o más módulos compuestos en un **archivo Enterprise Archive (EAR)**. Un archivo EAR es similar a un archivo JAR; la diferencia es que el primero contiene un archivo `application.xml` (situado en el directorio`META-INF`) que describe la aplicación:



¿Cuál es la función de los múltiples archivos de descriptor en la figura anterior? Los componentes de aplicación de menor tamaño pueden ser adaptados mientras son integrados en módulos. Esto requiere el uso de un descriptor de implementación en un módulo. Sin embargo, no toda la información relacionada con la implementación final sobre una plataforma J2EE estará disponible en el momento del ensamblado de módulos. Además, necesitamos medios para especificar qué módulos componen la aplicación.

La ventaja de esta estructura es que permite la reutilización a diferentes niveles. Los componentes de aplicación pueden ser reusados en múltiples módulos Web. Por ejemplo, los componentes Web relacionados con el registro de usuario pueden ser empaquetados en diferentes módulos Web responsables de los servicios de pedido y de compra on-line. De un modo similar, los módulos pueden ser reusados en múltiples aplicaciones de modo que un módulo EJB para la gestión del carro de la compra no necesita ser restringido a una única aplicación, sino que puede ser empaquetado en múltiples aplicaciones de comercio on-line.

Sin estos medios de empaquetado reusables, el objetivo de la reutilización de componentes sólo se alcanzaría parcialmente. Al definir la estructura anterior, J2EE permite una reutilización más refinada de los componentes de aplicación.

Implementación de la aplicación

Finalmente, la implementación de aplicación es el proceso de instalación y adaptación de los módulos empaquetados sobre la plataforma J2EE. Este proceso conlleva dos pasos:

- Preparar la aplicación para su instalación sobre un servidor de aplicación J2EE. Esto conlleva copiar los archivos EAR sobre el servidor de aplicación, generando clases de implementación adicionales con la ayuda del contenedor, e instalar finalmente la aplicación sobre el servidor.

- Configurar la aplicación con la información de la aplicación específica del servidor. Un ejemplo sería la creación de fuentes de datos y factorías de conexiones, ya que la creación real de estos objetos es específica del servidor de aplicación.

Resumen

En estos momentos debería tener una idea más completa de la tipología del paisaje J2EE. En particular, hemos visto cómo J2EE amplía la arquitectura multi-nivel tradicional de procesamiento distribuido para incluir el concepto de contenedores. Hemos examinado las partes constituyentes de esta arquitectura de contenedor, con relación a:

- El contrato de componentes
- Servicios de contenedor
- Servicios declarativos
- Otras instalaciones de periodo de ejecución

Los siguientes son los puntos clave de este examen:

- J2EE es una arquitectura centrada en el contenedor. Un contenedor en más que un simple periodo de ejecución: proporciona varios niveles de abstracción. En próximos capítulos, examinaremos cómo estas abstracciones simplifican la impelementación de aplicaciones.
- J2EE reconoce la necesidad de componer los componentes en módulos y módulos en aplicaciones. Es un intento de normalizar la reutilización de componentes de aplicación y módulos.
- J2EE representa un enfoque muy intuitivo para construir aplicaciones. Mientras que el proceso de diseño es descendente, el proceso de implementación es ascendente y es un proceso de composición, que compone módulos a partir de componentes y aplicaciones a partir de módulos.

Después de concluir nuestro análisis introductorio de la perspectiva de J2EE de la programación Java de lado servidor, comenzaremos un recorrido detallado examinando de cerca la búsqueda de servicios utilizando JNDI.

2
Servicios de directorio y JNDI

Las interfaces **JNDI (Java Naming and Directory Services Interfaces)** están diseñadas para simplificar el acceso a la infraestructura de directorio utilizada en el desarrollo de aplicaciones de redes avanzadas. Los **directorios** son un tipo especial de bases de datos que proporcionan rápido acceso a sus almacenes de datos. Las bases de datos son consideradas normalmente el modelo relacional de almacén de datos por excelencia, como Oracle, Microsoft SQL Server, etc. En contraste con esto, una base de datos de directorio almacena información de forma jerárquica y optimizada para la lectura.

Normalmente, hemos tenido que utilizar diferentes API para acceder a distintos servicios de directorio, como el servicio de información de Sun Network Information Service (NIS) o **Lightweight Directory Access Protocol (LDPA)**. Sin embargo, JNDI proporciona un API estándar para acceder a cualquier tipo de directorio. JNDI también nos permite almacenar y recuperar objetos Java en la red, por ejemplo, del servidor de aplicación adaptado a J2EE, como BEA WebLogic.

En este capítulo, responderemos a las siguientes preguntas:

- ¿Qué es un servicio de directorio?
- ¿Qué distingue a JNDI de un API de directorio tradicional?
- ¿Qué es LDPA y cómo lo utilizamos para trabajar con un servicio de directorio?

Además, examinando algunos programas de ejemplo, demostraremos de primera mano las aplicaciones prácticas del uso de JNDI para gestionar la información del directorio.

Servicios de designación y de directorio

"La red es el ordenador" es una frase que Sun Microsystems ha utilizado para vender sus productos de software y hardware. Es también el dogma de la exitosa programación Java. Una de las razones por las que Java es tan popular es porque ha conseguido que la programación de redes sea más fácil que con otros

lenguajes, como con C. Esto se debe a que Java es mucho más que una especificación de lenguaje. Java incorporada los sistemas de redes en su API central, mientras que lenguajes tales como C requieren una bibilioteca externa para reproducir la misma funcionalidad. Las funciones de red que incorpora Java garantiza el éxito de este leguaje debido a que el número de ordenadores conectados que utilizan los protocolos estándar de Internet ha crecido a un ritmo excepcional.

Los dispositivos se sitúan normalmente en una red para facilitar su gestión, ahorrando tiempo y, a su vez, dinero. Por ejemplo, en un complejo de edificios un sistema medioambiental en red puede informar a un servidor central. Este servidor puede informar sobre el estado de todos los sistemas de aire acondicionado y termostatos de los diferentes edificios a los ingenieros medioambientales. Desde esta fuente central, los ingenieros pueden saber cuándo una unidad se está sobrecalentando, o cuándo deben planificar los servicios de mantenimiento. Si el sistema medioambiental no estuviera conectado en red, los miembros del equipo tendrían que comprobar cada dispositivo de forma individual, lo que supondría una tarea laboriosa y larga cuando se tratara de docenas de edificios.

Por supuesto, aunque tenemos dispositivos en red y un gran lenguaje de desarrollo como Java, puede resultar difícil combinarlos de un modo efectivo. Siempre que se añade un dispositivo a la red, la información de configuración debe ser adaptada. Esta información de configuración incluye propiedades tales como una dirección IP, un nombre de dominio, las prestaciones de cada dispositivo de red y los privilegios de usuario para cada dispositivo. Esta información está disponible normalmente para el dispositivo mismo, pero no siempre lo está para el mundo exterior.

Por ejemplo, cuando establecemos una nueva estación de trabajo en la red, con frecuencia necesitamos configurarla para utilizar una impresora de red específica. A menudo no es posible descubrir las funciones de las impresoras que están disponibles, porque las impresoras no ofrecen esta información. Utilizando el servicio de directorio y Java, las prestaciones y la configuración de la impresora pueden ser incorporadas a la red. Los dispositivos pueden consultar a la impresora para descubrir, por ejemplo, si puede imprimir en color.

Los diseñadores del entorno J2EE y sus API comprenden estos problemas. Por este motivo, el API Java Naming and Directory Interfaces (JNDI) se convirtió en una parte fundamental de J2EE. Descubrirá que prácticamente todas las aplicaciones J2EE utilizan JNDI en algún momento.

Para entender JNDI, necesitamos una comprensión básica de lo que es un servicio de designación y directorio. También necesitamos comprender por qué es importante que JNDI proporcione una interfaz normalizada para estos servicios.

Servicios de designación

> **Un servicio de designación es un servicio que permite la creación de un nombre estándar para un conjunto determinado de datos.**

En Internet, cada host tiene un nombre de dominio completo y cualificado, **Fully Qualified Domain Name (FQDN)**, como www.wrox.com o www.coe.unt.edu. Un FQDN se construye a partir de un hostname, cero o más nombres de subdominio, y un nombre de dominio. Por ejemplo:

FQDN	Hostname	Subdominio	Dominio
www.coe.unt.edu	www	coe	unt.edu
www.wrox.com	www	–	wrox.com
www.p2p.wrox.com	www	p2p	wrox.com

Cada FQDN es único, lo que significa que sólo se permite un www.wrox.com y un www.coe.unt.edu. Utilizando nombres de subdominios y de dominios, los sistemas que comparten el mismo nombre de anfitrión siguen considerándose entidades separadas. Por ejemplo, www.p2p.wrox.com es diferente de www.wrox.com. En este caso, el subdominio p2p.wrox.com está realmente por debajo del dominio raíz de wrox.com obteniendo su nombre de espacio único.

Servicios de directorio

> **Un servicio de directorio es un tipo especial de base de datos optimizado para acceso de lectura utilizando varias técnicas de indización, almacenamiento en memoria y acceso a disco. La información contenida en un servicio de directorio se describe utilizando un modelo de información jerárquica.**

El modelo de información jerárquica que emplean los servicios de directorio contrasta con el modelo de información relacional empleado por las bases de datos, como Oracle y Microsoft SQL Server, que están creadas para manejar transacciones. Una operación sólo se realiza con éxito si un conjunto de operaciones más pequeñas son realizadas con éxito. Un servicio de directorio también es diseñado normalmente de modo que pueda ser distribuido y copiado.

Un servicio de directorio siempre tendrá un servicio de designación, pero un servicio de designación no siempre tiene un servicio de directorio. Un ejemplo sencillo de un servicio de directorio es un listín telefónico. Un listín telefónico permite al usuario encontrar rápidamente el número de teléfono de una persona, siempre que el usuario conozca el nombre de la persona cuyo número está buscando.

Si nos quedamos un momento con este ejemplo, podemos subrayar los beneficios de los servicios de directorio en red. Es decir, un servicio de directorio sólo es tan útil como la actualidad de su información y la eficacia de realizar una búsqueda. Los recursos estáticos, tales como copias físicas de listines telefónicos, guías de televisión, etc., se tornan obsoletas poco después de ser publicadas y normalmente sólo ofrecen un medio de acceso (por ejemplo, mediante el "apellido" en el caso de un listín telefónico). Un servicio de directorio electrónico, por otro lado, es claramente más dinámico y permite mucha más flexibilidad de búsqueda. La información que se recibe probablemente esté actualizada en el momento de la consulta.

Existen muchos servicios de directorio (y seudo-directorio) en uso en las redes en la actualidad. Un servicio de directorio muy utilizado es el **Domain Naming Service (DNS)** utilizado en Internet. DNS acepta un FQDN y devuelve una dirección IP. Este servicio es esencial para Internet ya que una comunicación eficaz entre dos ordenadores confía en que cada sistema conoce la dirección IP del otro.

Las direcciones IP consisten en la actualidad en 32 bit, por ejemplo `198.137.240.92`, mientras que un hostname toma la forma www.wrox.com. Los ordenadores están acostumbrados a tratar con números, pero la mayoría de los humanos son más diestros a la hora de recordar nombres (es mucho más fácil recordar www.wrox.com que recordar `198.137.240.92`). El sistema DNS es un ejemplo de un servicio de directorio altamente especializado. Cuando introducimos el hostname en un navegador, el ordenador obtiene la dirección IP del servidor vía DNS. Analizaremos más detenidamente el lado práctico del sistema DNS con un ejemplo en partes posteriores del capítulo.

DNS nos proporciona en realidad algo más que direcciones IP. Por ejemplo, los nombres de las máquinas responsables de encaminamiento de correo, pero sigue siendo muy especializado en sus funciones.

Las organizaciones a menudo utilizan uno o más de los siguientes servicios de directorio de propósito general:

- Novell Directory Services (NDS)
- Network Information Services (NIS/NIS+)
- Active Directory Services (ADS)
- Windows NT Domains

Aunque el Servicio NT Domains no es un verdadero servicio de directorio, son muchos los que intentan utilizarlo como tal.

Cada uno de estos servicios de directorio proporciona más información que la simple copia del nombre de IP que obtenemos de DNS. Cada servicio de directorio también permite que información sobre usuarios sea almacenada (nombres de usuario y contraseñas), grupos de usuario (para control de acceso) y ordenadores (ethernet y direcciones IP). NDS y ADS albergan más funciones (como la localización de impresoras de red y software) que NIS o NT Domains.

Como hay tantos servicios de directorio (cada uno con su protocolo) y como tenemos tantos sistemas en nuestras redes, han surgido algunos problemas importantes. En esencia, se reducen a dos temas clave:

- Mantener un registro de usuarios.
- Mantener un registro de recursos de red como ordenadores e impresoras.

Un marco particularmente difícil sería aquel en el que los usuarios necesitarían acceder a:

- Un sistema Novell (que utilice NDS) para uso común de archivos e impresión.
- Una cuenta en una unidad Windows NT para ejecutar Microsoft Office.
- Una cuenta en una unidad UNÍX (utilizando NIS) para hacer uso del e-mail y publicar páginas Web.

Por desgracia para los gestores de red, ninguno de estos servicios de directorio puede interactuar fácilmente entre sí, lo que dificulta bastante al desarrollador de aplicaciones tradicionales interactuar con varios servicios de directorio diferentes. Como consecuencia, los usuarios suelen tener al final diferentes nombres de usuario y contraseñas para cada sistema. Si un usuario se da de baja, puede resultar difícil asegurar que todas sus cuentas son eliminadas. Los vacíos de seguridad tienen lugar en forma de cuentas "huérfanas", que son cuentas que pertenecen a usuarios que ya no están en una organización, aunque todavía tienen libre acceso a los sistemas.

Una solución a estos problemas es utilizar JNDI, que proporciona un API estándar para una variedad de servicios de directorio. Sin embargo, debido a que no todo el mundo utiliza Java, es necesario que exista otro modo de facilitar la comunicación de información de directorio entre sistemas.

LDAP

El protocolo **Lightweight Directory Access Protocol (LDAP)** fue desarrollado a principios de los años noventa como protocolo de directorio estándar. LDAP es ahora probablemente el protocolo de directorio más popular y, debido a que JNDI puede acceder a LDAP, invertiremos algún tiempo en aprender cómo aprovechar LDAP para mejorar las aplicaciones Java con JNDI.

LDAP puede remontar su origen al protocolo X.500 (también conocido como "Heavy" Directory Access Protocol), que estaba originalmente basado en los protocolos de red OSI (un primer "competidor" de los protocolos de Internet).

El usuario interactuará a menudo con un servidor que ha sido específicamente construido para LDAP, como es el caso de iPlanet Directory Server. Sin embargo, LDAP puede ser enfrentado a cualquier tipo de almacén de datos. Por ello, los servicios de directorio de mayor aceptación cuentan en la actualidad con un frontal LDAP de algún tipo que incluye NIS, NDS, Active Directory e incluso Windows NT Domains.

> **LDAP define cómo los clientes deben acceder a los datos en el servidor. Sin embargo, no especifica cómo deben ser almacenados los datos en el servidor.**

Datos LDAP

Los datos en LDAP están organizados en forma de árbol jerárquico, llamado **Directory Information Tree (DIT)**, o Árbol de Información de Directorio. Cada "hoja" del DIT es conocida como una entrada y la primera entrada en un DIT se denomina entrada raíz.

Una entrada comprende un **Distinguished Name (DN)**, o Nombre Diferenciado, y cualquier número de pares de valor atributo. El DN es el nombre de una entrada y debe ser único, igual que la clave única de una tabla de base de datos relacional. Un DN muestra cómo la entrada está relacionada con el resto del DIT, igual que el nombre de la ruta de un archivo muestra cómo éste se relaciona con el resto de los archivos de un sistema.

La ruta hacia un archivo en un sistema se lee de izquierda a derecha a la hora de leer desde la raíz al archivo. Un DN se lee de derecha a izquierda a la hora de leer desde la raíz a la entrada, por ejemplo:

```
uid=scarter, ou=People, o=airius.com
```

La parte situada a la izquierda de un DN se denomina **Relative Distinguished Name (RDN)**, o Nombre Diferenciado Relativo, y está formado por un par de valor atributo que se encuentra en la entrada. El RDN en este ejemplo sería `uid=scarter`.

Los atributos LDAP a menudo utilizan mnemotécnicas como nombres. Éstos son algunos de los atributos LDAP más comunes:

Atributo LDAP	Definición
`Cn`	Nombre común
`Sn`	Apellido
`Givenname`	Primer nombre
`Uid`	ID de usuario
`Dn`	Nombre Diferenciado
`mail`	Dirección de e-mail

Los atributos pueden tener uno o más valores. Por ejemplo, un usuario puede tener más de una dirección de e-mail, por lo que necesitarán más de un valor para su atributo `mail`. Los valores de atributo pueden ser texto o datos binarios y se hace referencia a ellos en paresnombre/valor.

Hay también un atributo especial llamado `objectclass`. El atributo `objectclass` de una entrada especifica qué atributos son necesarios y qué atributos están permitidos en una entrada en particular. Al igual que los objetos en Java, las clases de objeto en LDAP pueden ser ampliadas. Cuando una clase de objeto es ampliada para una entrada en particular, la clase de objeto mantiene los atributos existentes, pero los nuevos atributos pueden ser especificados.

Éste es un ejemplo de entrada LDAP (sobre las que hablaremos más adelante en este capítulo) representado en el formato **LDAP Data Interchange Format (LDIF)**, que es el modo más común de mostrar los datos LDAP en un formato legible:

```
dn: uid=scarter, ou=People, o=fedup.com
cn: Sam Carter
sn: Carter
givenname: Sam
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
ou: Accounting
ou: People
l: Sunnyvale
uid: scarter
mail: scarter@fedup.com
telephonenumber: +1 408 555 4798
facsimiletelephonenumber: +1 408 555 9751
roomnumber: 4612
```

Los atributos también tienen reglas de concordancia. Estas reglas indican al servidor cómo decidir si una entrada concreta concuerda o no con una determinada consulta. Las posibles reglas de concordancia son estas:

Regla de concordancia	Significado
DN	El atributo está en forma de un Nombre Diferenciado
Case-Insensitive String (CIS)	El atributo puede concordar si el valor de la consulta es igual al valor del atributo, con independencia del caso
Case-Sensitive String (CSS)	El atributo puede concordar si el valor de la consulta es igual al valor del atributo incluido el caso
Telephone	Es lo mismo que CIS excepto que caracteres como "-" y "()" son ignorados a la hora de determinar la concordancia
Integer	La concordancia de atributo es determinada utilizando sólo números
Binary	El atributo concuerda si el valor de la consulta y el valor del atributo son los mismos valores binarios (por ejemplo, consultar una base de datos LDAP para buscar una foto concreta)

La definición de atributos, las reglas de concordancia de atributos y la relación entre clases de objetos y atributos están definidas en el esquema del servidor. Un servidor contiene un esquema predefinido pero, siempre que el servidor se ajuste al protocolo LDAP v3 tal y como se define en RFC 2251 (http://www.ietf.org/rfc/rfc2251.txt), el esquema puede ser ampliado para incluir nuevos atributos y clases de objeto.

Los servidores LDAP tienen otras ventajas. Admiten referentes (apuntadores hacia otros directorios LDAP en los que residen datos), de modo que un único servidor LDAP puede buscar millones de entradas a partir de la consulta de un solo cliente y los datos LDAP pueden ser replicados para mejorar la fiabilidad y aumentar la velocidad. LDAP cuenta también con un modelo de seguridad muy fuerte que utiliza ACL para proteger los datos que se encuentran en el interior del servidor y ajustándose a los protocolos de **Secure Socket Layers (SSL)**, **Transport Layer Security (TLS)** y **Simple Authentication and Security Layer (SASL)**.

LDAP está teniendo una creciente aceptación como servicio de directorio central para sistemas de red. Para más información acerca de LDAP debería consultar Implementando LDAP *por Mark Wilcox, de Wrox Press, ISBN 1-861002-21-1.*

Introducción de JNDI

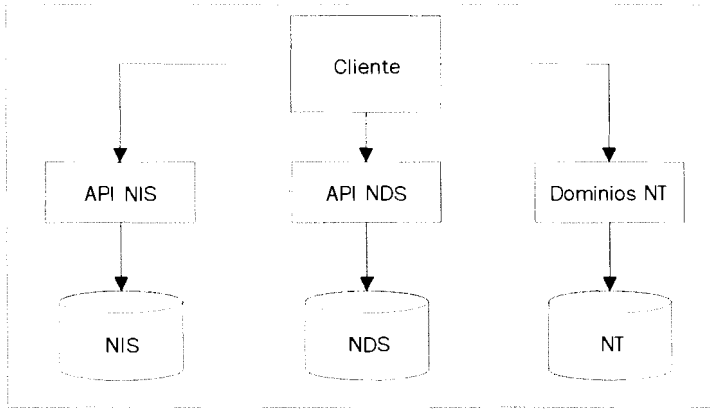
Aunque LDAP está creciendo en popularidad y en uso, le queda todavía mucho camino que recorrer hasta llegar a ser omnipresente. Otros servicios de directorio tales como NIS son todavía muy utilizados y es posible que sigan siéndolo durante mucho tiempo todavía. Otro aspecto es que las aplicaciones de empresa necesitan con frecuencia ajustarse a las normas de procesamiento distribuido existentes, tales como **Common Object Request Broker Architecture (CORBA)**, que se utiliza mucho en muchas grandes organizaciones para permitir que diferentes tipos de aplicaciones interactúen entre sí.

CORBA es una arquitectura independiente del lenguaje y de la plataforma utilizada que permite la programación de aplicaciones distribuidas, en la que una aplicación en una máquina puede acceder a una función de una aplicación diferente situada en una máquina diferente como si estuviera llamando una función interna. CORBA utiliza el servicio `COSNaming` (CORBA Object Service) para definir la posición de los objetos disponibles.

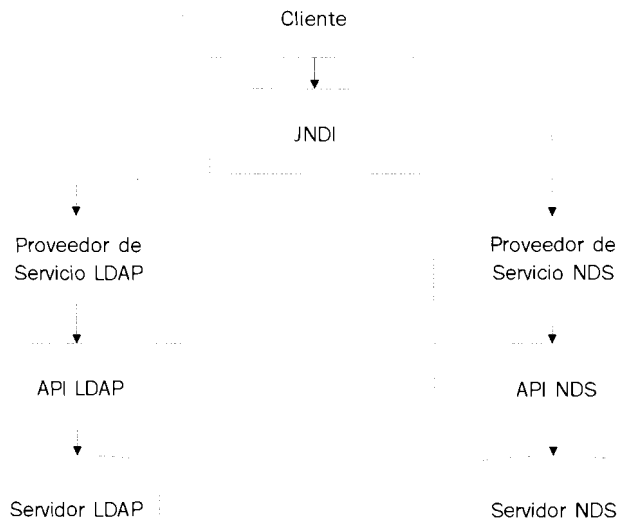
Para superar estas dificultades, se ha creado un API estándar de Java para que interactúe con los servicios de designación y directorio. Este API es análogo al modo en el que los desarrolladores de programas utilizan JDBC para interactuar con todo tipo de bases de datos.

JNDI es fundamental para el desarrollo a largo plazo de Java, en particular de la iniciativa Enterprise JavaBeans (EJB). El significado de EJB para J2EE queda reflejado en el hecho de que una larga proporción de este libro está dedicado a EJB. Empezando por el capítulo 16, veremos de primera mano la importancia de JNDI para la funcionalidad de EJB. Un componente clave de la tecnología EJB es la capacidad para almacenar y recuperar objetos en la red. Un servicio de directorio va a ser el almacén primario de datos para objetos estables Java, que son recuperados de la red en más ocasiones de las que son almacenados. Esto ocurre porque, cuando los objetos son cargados de la red, queremos poder localizarlos rápidamente y un servicio de directorio nos permite una rápida búsqueda y recuperación de datos. Estos datos pueden ser cualquier cosa, incluido un registro de un usuario particular de datos binarios, como un objeto Java serializado.

Los siguientes diagramas le ayudarán a ilustrar la relación entre los servicios de directorio, JNDI y LDAP. El primer diagrama muestra la relación entre un cliente y una variedad de servicios de directorio. Cada servicio de directorio requiere su propio API, que añade complejidad y sobrecarga de código para nuestra aplicación de cliente:



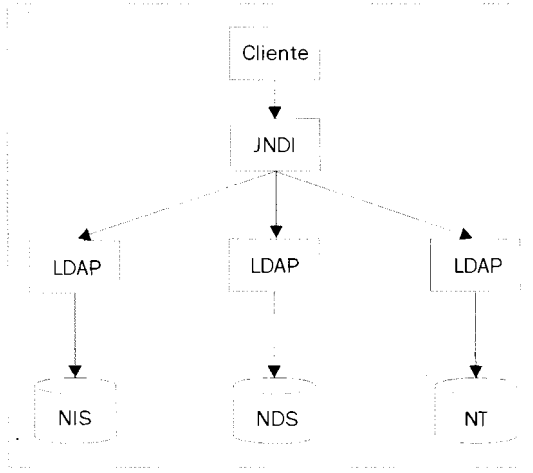
Este diagrama puede simplificarse utilizando JNDI. Hay todavía múltiples servidores y múltiples API por debajo, pero para el creador de la aplicación aparece como un único API:



Así, como desarrolladores de programas, nos resulta más fácil crear aplicaciones de red porque podemos concentrarnos en un único API. Sin embargo, todavía tenemos que enfrentarnos a problemas potenciales porque no todos los proveedores de servicios JNDI están creados del mismo modo. Los **proveedores de servicios** en terminología JNDI son los dispositivos que permiten la interacción con diferentes servicios de directorio. Cada servicio de directorio es claramente diferente (nombran sus entradas de forma diferente y tienen diferentes capacidades), por lo que los desarrolladores de programas todavía construyen aplicaciones que son mayores y más proclives a errores.

Y un aspecto más importante si cabe es que puede ser difícil integrar sistemas que puedan comprender LDAP pero sean incapaces de utilizar Java.

Finalmente, mostramos cómo JNDI y LDAP pueden trabajar juntos, ofreciendo la solución más elegante:



Aquí utilizamos JNDI para comunicar con varios servidores LDAP. El creador de programas sólo tiene que preocuparse por un protocolo en particular (LDAP) y un API (JNDI). Por supuesto, estamos confiando en que los distribuidores proporcionan interfaces a sus protocolos de propiedad. Sin embargo, esto no es un problema ya que, para estos servicios de directorio de gran aceptación, hay productos adicionales que nos permiten comunicarnos con ellos vía LDAP.

Las concesiones

Aunque JNDI proporciona un API común para programación a diferentes servicios de directorio, hay algunas concesiones que deben realizarse:

- Una concesión indica que hay mucho sobregasto. Por ejemplo, cada objeto `DirContext` (los objetos que utiliza JNDI para interactuar con un servicio de directorio) debe ajustarse a un número de métodos, muchos de los cuales nunca utilizará. Cuando implemente un objeto `DirContext`, es sencillo dejarlo vacío de modo que no cause problemas con su aplicación, pero sí añade peso extra en términos de memoria y de tiempo de compilación durante la ejecución de programa. Veremos un ejemplo de esto más adelante, en este mismo capítulo.
- Además, para que JNDI pueda mantener una interfaz común, no siempre se ajusta a todas las funciones del servicio de directorio. Un buen ejemplo de éste es el comando LDAP `compare`. Nos permite comparar un valor particular de un atributo en una única entrada en el servidor LDAP. El servidor LDAP debe devolver un código resultado en lugar de una serie de resultados de búsqueda completa, convirtiendo así la operación de comparación en un trámite ligero y rápido. Sin embargo, JNDI no se ajusta a una verdadera comparación LDAP, porque LDAP es el único servicio de directorio que apoya esta operación. En su lugar, JNDI realiza una sencilla búsqueda que recupera físicamente la entrada en la memoria del servidor y realiza la operación de comparación en la aplicación. El resultado es el mismo para la aplicación pero ya no contamos con los beneficios de la comparación real LDAP.
- Finalmente, no es posible obtener acceso directo a la conexión del servidor de directorio, lo que significa que es muy difícil implementar la reserva de conexión o comprobar la expiración de una aplicación.

Para la mayoría de las aplicaciones, estos inconvenientes no suponen un serio problema. JNDI proporciona un API común a una variedad de servicios de directorio, que está incluido como parte estándar de Java, del mismo modo que JDBC proporciona un API estándar a las bases de datos relacionales. En algunos caso, JNDI puede ser el único API disponible para un servicio de directorio en particular como es el caso de DNS o de un servidor de aplicación Java como BEA WebLogic.

Sin embargo, en algunos casos, las aplicaciones sí necesitan acceso de bajo nivel al servicio de directorio y, si la aplicación no necesita ninguno de los servicios que ofrece JNDI (como almacenamiento y recuperación de objetos Java), puede que queramos optar por un API Java de bajo nivel. Por ejemplo, en LDAP está el Netscape Directory SDK para Java, que es en realidad más elaborado que el API JNDILDAP. Es un SDKde código abierto, disponible en http://www.mozilla.org/directory/.

JNDI puede que tampoco sea la mejor solución cuando una aplicación necesita manipular archivos en el sistema de archivos local. Si los objetos Java como los objetos DataSource para programación JDBC no deben ser cargados, es mejor en este caso utilizar las clases estándar disponibles en el paquete `java.io`.

¿Por qué utilizar JNDI cuando tenemos LDAP?

Entonces, ¿por qué necesitamos JNDI si podemos comunicar con los sistemas vía LDAP? Asimismo, ¿por qué necesitamos LDAP cuando tenemos JNDI? Para responder a estas preguntas, estudiemos cómo podemos utilizar JNDI o LDAP de forma independiente.

LDAP sin JNDI

LDAP es un buen modo de convergir el acceso a datos de directorio mediante un único protocolo. Resulta útil no sólo en la programación Java sino también para otros tipos de aplicaciones.

Por ejemplo, LDAP es un buen modo de proporcionar un libro de direcciones de red basado en un estándar (la mayoría de los paquetes de e-mail cuentan con la capacidad de consultar un servidor LDAP para búsquedas de direcciones de e-mail) o simplemente de mantener un registro de los dispositivos de una red.

JNDI ofrece la capacidad de comunicar directamente con diferentes servicios de directorio de red cuando no necesitamos LDAP. Por ejemplo, puede que necesite determinar si alguien ha accedido a una aplicación determinada basada en su pertenencia a un grupo NIS concreto, o puede que necesite recuperar un registro MX en un servidor DNS. JNDI facilita la tarea al creador de programas en estos casos proporcionando un API consistente en vez de tener que aprender cómo hacer lo mismo utilizando diferentes grupos de API.

JNDI sin LDAP

Aunque podría proporcionar un servicio similar a LDAP para sus aplicaciones Java a través de JNDI, esto sólo sería útil para sus aplicaciones Java y probablemente tendría que repetirlo en cada aplicación Java que grabe que necesite este tipo de servicio.

LDAP es un estándar abierto mantenido por Internet Engineering Task Force (IETF). Esto significa que es accesible desde diversos clientes y distribuidores. Ésta es una de las extraordinarias características de Internet y la razón por la que los protocolos basados en Internet han reemplazado a los protocolos de propiedad como medios primarios de comunicaciones en red.

¿Y XML?

Cuando XML fue lanzado por primera vez, hubo cierto debate sobre la posibilidad de que reemplazara a todas las tecnologías anteriores incluido Java y LDAP. Por supuesto, esto no ocurrió. En su lugar, XML se ha convertido en el estándar *de facto* para datos persistentes, en especial para situaciones en las que necesite intercambiar datos persistentes entre aplicaciones grabadas en diferentes lenguajes de aplicaciones (por ejemplo, entre Visual Basic y Java).

XML también tiene su función en LDAP y en JNDI. Hay una especificación XML concreta para LDAP llamada **Directory Services Markup Language (DSML)** que está siendo desarrollada principalmente para distribuir información de directorio a través de un protocolo de base XML-RCP como SOAP. Consiste en un lenguaje de marcado para representar servicios de directorio en XML. Si quiere más detalles, vaya a http://www.dsml.org/.

DSML está diseñado para reemplazar el modo estándar de intercambio de datos LDAP externos al protocolo LDAP. El estándar actual es el formato LDAP Data Interchange Format (LDIF) y ése es el que utilizaremos para mostrar nuestros ejemplos en este capítulo.

Uso de JNDI

Ahora que ya tenemos un conocimiento básico de los servicios de directorio, JNDI y LDAP, es el momento de ensuciarnos las manos trabajando.

Para utilizar JNDI tal y como se describe en este capítulo necesitaremos lo siguiente:

- JDK 1.3 de Sun, que incluye JNDI estándar, o JNDI Software Development Kit (SDK) 1.2.2 (disponible en http://java.sun.com/jndi/).
- Un servidor de directorio que cumpla las normas LDAP v3. El servidor Directorio de Netscape, disponible en http://www.iplanet.com/download/download/index.html, se utiliza en este capítulo.

Todos los ejemplos que encontrará en este capítulo utilizan los datos de prueba `arius.com` que viene en este servidor de directorio.

Aunque JNDI está incorporado en JDK 1.3 y superior, puede que todavía necesite obtener el archivo ldap.jar de JNDI 1.2.2 para conseguir toda la funcionalidad, incluida la capacidad de utilizar los controles LDAP y de almacenar/recuperar objetos RMI en un servidor LDAP.

Instalación de JNDI

Éstos son los pasos generales a seguir para empezar a funcionar en un sistema estándar:

- Obtener información de su administrador de servidor LDAP o instalar un servidor que cumpla con los requisitos LDAP v3. Yo he utilizado varios servidores LDAP para diversos desarrollos, incluido openLDAP (http://www.openldap.org) en Red Hat Linux 6.2, el servidor LDAP gratis de Eudora para Windows (http://www.eudora.com) y los servidores Iplanet 4 & 5 Directory (http://www.iplanet.com) tanto en Red Hat Linux 6.2 como en Windows 2000.
- Instalar JDK 1.3, que contiene JNDI y el proveedor LDAP básico. Si no está utilizando funciones avanzadas de LDAP como controles, no necesitará instalar nada más excepto JDK 1.3. Si está explorando funciones avanzadas LDAP, lea la página JNDI en http://java.sun.com/JNDI/ para más información.

Ahora ya estamos preparados para utilizar JNDI y LDAP.

Proveedores de servicios JNDI

El usuario no puede utilizar JNDI sin utilizar un proveedor de servicios, por lo que primero debemos entender el concepto de proveedor de servicios.

> **Un proveedor de servicios es un grupo de clases Java que le permite comunicar con un servicio de directorio de un modo similar al que los dispositivos JDBC le permiten comunicar con una base de datos.**

Para que un proveedor de servicios esté disponible para su uso en JNDI debe implementar la interfaz Context. La mayoría de los proveedores de servicio que utilizará es probable que implementen la interfaz DirectoryContext, que extiende la interfaz Context para capacitar servicios de directorio.

Esto significa que sólo tenemos que aprender a manejar JNDI para saber que las llamadas API conectan a un servicio de designación o de directorio, mientras que el proveedor de servicios se preocupa de los detalles aburridos como el protocolo de red y los valores codificado/decodificado.

Por desgracia, los proveedores de servicios no son la panacea para el uso de servicios de directorio. Debemos saber algo sobre el servicio de directorio subyacente de modo que podamos nombrar nuestras entradas correctamente y construir correctas consultas de búsqueda. A diferencia de las bases de datos relacionales, los servicios de directorio no comparten un lenguaje de consulta común como SQL.

En JNDI 1.2, los proveedores de servicios pueden ajustarse a un concepto de **federación** donde un proveedor de servicios puede pasar una operación a otro proveedor de servicios si no entiende el designación o el esquema de la operación. Por ejemplo, si quisiéramos encontrar el registro XM (este

contiene el servidor de e-mail de destino preferido para un dominio) para un dominio concreto de DNS, pero nuestro proveedor inicial de servicio era LDAP, no podría responder a la consulta. Si hubiera federación, podría pasar la solicitud al siguiente proveedor de servicios de la lista de federación y, si fuera un proveedor de servicios DNS, podría tramitar la solicitud por nosotros.

En una situación ideal, esto debería resultar transparente para el programador de la aplicación. Sin embargo, como esta es una característica nueva, todavía no ha sido probada ni implementada por completo.

En conclusión, un proveedor de servicios permite a nuestras aplicaciones JNDI comunicar con un servicio de designación/directorio. El resto de las interfaces, clases y excepciones giran todas alrededor de nuestra interacción con un proveedor de servicios.

Cómo obtener proveedores de servicios JNDI

Si descargamos el SDK JNDI, veremos que, junto con el API y la documentación, se incluye una serie de proveedores de servicios existentes. Éstos incluyen proveedores para LDAP, NIS, COS y el registro RMI y sistema de archivos. Muchos distribuidores diferentes también facilitan proveedores de servicios para otros servicios de directorio, o como sustitutos para los proveedores por defecto de Sun. Por ejemplo, Novell tiene un proveedor de servicios para NDS, mientras que tanto IBM como Netscape cuentan con proveedores de servicios alternativos para LDAP.

Resulta fácil cambiar entre proveedores de servicios. Por ejemplo, para utilizar el proveedor de servicios LDAP de Sun por defecto realizaríamos una llamada como ésta:

```
// Especificar qué clase utilizar para nuestro proveedor JNDI
env.put(Context.INITIAL CONTEXT FACTORY,
          "com.sun.jndi.ldap.LdapCtxFactory");
```

Ahora para utilizar el proveedor de servicios LDAP de IBM simplemente sustituiría com.sun.jndi.ldap.LdapCtxFactory por el nombre completo del paquete del proveedor de servicios LDAP de IBM del siguiente modo:

```
// Especificar qué clase utilizar para nuestro proveedor JNDI
env.put(Context.INITIAL CONTEXT FACTORY,
          "com.ibm.jndi.LDAPCtxFactory");
```

Puede encontrar una lista de proveedores de servicios actuales en http://java.sun.com/products/jndi/serviceproviders.html.

Cómo desarrollar su propio proveedor de servicios

Puede que también necesite implementar su propio proveedor de servicios si necesita utilizar un servicio de directorio (como los dominios de Windows NT o Bayan Vines) que todavía no tiene un proveedor de servicios.

En el SDK JNDI, Sun facilita un ejemplo de cómo grabar un proveedor de servicios. Esta tutoría JNDI está disponible en http://java.sun.com/products/jndi/tutorial/index.html. Sun también proporciona un documento PDF que describe el proceso de grabar un proveedor de servicios en ftp://ftp.javasoft.com/docs/jndi/jndispi.pdf. El proveedor de servicios LDAP de Netscape está disponible como proyecto de abierto en http://www.mozilla.org/directory/.

Java y LDAP

Antes de que comencemos a examinar activamente el uso de LDAP en Java, debemos analizar brevemente el uso común de LDAP en las aplicaciones de Java.

En la actualidad, hay tres aplicaciones básicas de LDAP en Java:

- Control de acceso
- Servicio de páginas blancas
- Directorio de procesamiento distribuido

Examinemos estas aplicaciones más detenidamente.

Control de acceso

Todas las aplicaciones dictan qué usuarios pueden acceder a ellas. Este acceso puede desde permitir el inicio a cualquier usuario que haga clic en el icono de la aplicación hasta permitir el acceso únicamente a un usuario que se ajuste a un escáner de retina. La mayoría de las aplicaciones se encuentran entre estos dos extremos.

El control de acceso puede descomponerse en autentificación y autorización.

Autentificación

La autentificación consiste en determinar quién es la persona que está utilizando un elemento de software. Debido a la naturaleza de la informática, nunca podemos estar cien por cien seguros de quién es el usuario, pero varios mecanismos de autentificación aumentan las probabilidades de que el usuario sea quien dice ser.

La autentificación requiere el uso de un secreto compartido, cuya forma más común es una contraseña. El usuario facilita un nombre de usuario y una contraseña. El nombre de usuario se utiliza para buscar el registro de la persona en cuestión en una base de datos (esta base de datos puede ser un simple archivo plano como el archivo `passwd` UNIX o una sofisticada base de datos como LDAP) y si la contraseña dada es la misma que la almacenada en la base de datos, entonces asumimos que el usuario es quien dice ser.

La autentificación basada en una contraseña es la más sencilla pero uno de los métodos menos seguros; puesto que las contraseñas son introducidas con el teclado, pueden ser detectadas en el tránsito por las redes o pueden ser fácilmente descifradas por alguien sin escrúpulos que acceda sin autorización a la base de datos de contraseñas. En el actual mundo de Internet, la mayor amenaza para las contraseñas (más allá de preguntar al usuario personalmente por su contraseña) es viajar por la red libremente. Por lo tanto, es mejor tomar medidas para proteger las contraseñas a la hora de viajar por la red. Dos mecanismos muy comunes son **MD5** y **Secure Socket Layers** (**SSL**).

En MD5 el cliente crea en primer lugar una refundición MD5 de la contraseña antes de introducirla en la red. El servidor compara entonces esta refundición con la que ha almacenado en su base de datos. Aunque este método evita que alguien adivine la contraseña de texto y que se registre en otros sistemas, no evita que alguien robe la contraseña refundida. Esta contraseña refundida robada puede ser utilizada igual que una contraseña de texto robada para acceder al sistema (conocido como ataque "*man in the middle*", algo así como "hombre en el medio"). Otro inconveniente es que la mayoría de los sistemas no almacenan sus contraseñas como MD5 (aunque ciertas implementaciones BSD sí lo hacen) porque se puede ver forzado a cambiar las contraseñas de todos antes de poder implementar este sistema.

Con SSL toda la conexión de red está cifrada. Esto no sólo protege la contraseña sino que también protege cualquier dato transferido entre el cliente y el servidor. Ésta es la razón por la que SSL se utiliza

con frecuencia para proteger aplicaciones de e-comercio. También resulta útil en LDAP porque los servidores LDAP a menudo contienen datos sensibles como números de identificación personal, números de licencia de dispositivos, etc. No queremos que estos datos viajen de forma insegura por la red igual que no queremos que lo hagan las contraseñas.

SSL también añade otra forma de autentificación en conjunto: el **certificado digital**. Los certificados digitales utilizan el cifrado público de claves para reforzar el proceso de autentificación. Bajo el cifrado público de claves, el usuario cuenta con una clave privada a la que sólo él tiene acceso. Todos los datos cifrados por la clave privada sólo pueden ser descifrados por la clave pública y viceversa. El servidor tiene acceso a su clave pública. Con SSL (que cifra toda la transferencia de datos), el servidor envía al usuario algunos datos para cifrar con su clave privada. El usuario cifra los datos y los devuelve al servidor. Si el servidor puede descifrar los datos con la clave pública y los datos coinciden, entonces se asume que se trata del usuario. El estándar de certificado digital (X.509) es una extensión del estándar de servicios de directorio original. De este modo, LDAP es uno de los modos más eficientes de gestión de certificados digitales. Sin embargo, hay que señalar que esta autentificación de usuario ocurre en realidad *después* de que el servidor SSL haya sido autentificado.

LDAP tiene la capacidad de ajustarse a un amplio espectro de servicios de autentificación incluido contraseñas, certificados digitales y el protocolo Simple Authentication and Security Layer (SASL).

Autorización

Después de haber autentificado a alguien, necesitamos determinar qué les está permitido hacer. Por ejemplo, podríamos decidir que sólo los miembros de los `Enanitos` tienen derecho a acceder a los archivos de `BlancaNieves`. También podríamos añadir granularidad más fina indicando simplemente si alguien tiene acceso o no. Por ejemplo, podríamos establecer que los miembros del `Consejo Jedi` tienen plenos derechos sobre elementos de la base de datos `Fuerza`, mientras que los miembros de los `Caballeros Jedi` tienen capacidad para editar ciertos elementos en la base de datos `Fuerza`, mientras que `padawan` sólo tiene capacidad de leer la base de datos `Fuerza`.

Podemos utilizar LDAP para desarrollar sofisticadas políticas de autorización.

Servicios de páginas blancas

Los servicios de páginas blancas permiten a alguien buscar usuarios basándose en atributos contenidos en sus entradas. Por ejemplo, puede buscar la dirección de e-mail de Mark Wilcox y obtener el número de teléfono de la oficina de Ingeniería, el número de edificio de Recursos Humanos, etc. Se llaman **páginas blancas** porque este tipo de información es similar al tipo de información que puede encontrar en las páginas blancas de teléfonos.

Este tipo de servicios son los más públicos de todas las operaciones LDAP y son para lo que muchos usuarios, si no todos, utilizan LDAP. En Java, lo estamos situando en segundo lugar porque la mayoría de los servicios de páginas blancas se facilitan mediante un cliente LDAP que se encuentra en un paquete de e-mail como Netscape Messenger o Microsoft Outlook en vez de a través de una aplicación Java.

Las aplicaciones Java realizan normalmente esta función bien como segundo plano de una interfaz LDAP de página Web o para proporcionar servicios de flujo de trabajo en una aplicación Java de empresa. Como ejemplo, estudiemos una sencilla situación en la que Juan introduce una solicitud de orden de compra (PO) electrónica de 100 artilugios de la marca Acme. La aplicación de compra tiene una regla comercial que establece que después de que alguien introduzca una solicitud (PO) debe ser enviada al encargado de autorizaciones de PO de su grupo. Por lo tanto, la aplicación consulta LDAP para saber quién es el autorizador de PO para el grupo de Juan en LDAP y les envía una notificación de e-mail utilizando el atributo de e-mail LDAP del autorizador.

Directorio de procesamiento distribuido

Uno de los segmentos de crecimiento más rápido de programación de servidor es la programación de red distribuida, donde una aplicación utiliza código que realmente reside independientemente de la aplicación ejecutada. El código puede estar en una JVM independiente (o máquina similar tipo servidor CORBA) o en una máquina física diferente situada al otro extremo del mundo.

A menudo utilizamos este tipo de programación para facilitar la reutilización del código de legado existente o para mejorar el rendimiento descargando un gran volumen de procesamiento en una máquina distinta.

En Java, hay tres arquitecturas distribuidas diferentes disponibles:

- Remote Meted Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)
- Enterprise JavaBeans (EJB)

Las tres arquitecturas proporcionan un servicio de registro que una aplicación de cliente utiliza para localizar el código distribuido. Debido a que su especificación fue desarrollada después de JNDI, EJB utiliza JNDI para sus servicios de registro.

RMI y CORBA utilizan sus propios servicios independientes de registro. Uno de los problemas de estos servicios es que no proporcionan un mecanismo que permite buscar en su registro para saber qué objetos están disponibles para su uso.

Con JNDI y LDAP el usuario puede proporcionar a estos servicios referencias indirectas con el objeto Reference JNDI. Por ejemplo, puede tener una entrada con un nombre de "`Real Time Stock Quote Service`" que tiene un atributo que contiene la localización de red real para el objeto RMI o CORBA.

Este tipo de servicio facilita la comprensión de su código y le otorga más flexibilidad en su aplicación. Por ejemplo, si necesita desplazar un determinado objeto distribuido a un nuevo servidor, en lugar de tener que cambiar todas las aplicaciones que referencian ese objeto, sólo tiene que cambiar su posición en LDAP y todas las aplicaciones de cliente actualizarán sus posiciones automáticamente la próxima vez que referencien el objeto.

LDAP también puede almacenar otros atributos descriptivos en una entrada, de modo que el usuario podría almacenar mejores descripciones de su objeto en el directorio para construir páginas blancas para objetos. Quienes han estado implicados en proyectos de sus compañías para el desarrollo de librería de códigos centralizadas encuentran a menudo que el desarrollo del código era la parte fácil; conseguir que el código sea fácil de encontrar en la parte difícil. Utilizar LDAP como un directorio de clase es una solución a estos problemas.

Podríamos añadir otros elementos a la descripción de nuestro ejemplo de objeto almacenado. Por ejemplo, podríamos añadir elementos como el lenguaje de desarrollo utilizado, una anotación sobre su propósito y el nombre del creador del programa o del equipo de desarrollo.

Configuración de la aplicación

LDAP también puede utilizarse para almacenar información de configuración sobre una aplicación. Esto es especialmente útil cuando la aplicación la utiliza el mismo usuario pero en diferentes máquinas y sería útil proporcionar la misma información de configuración sin tener en cuenta la máquina utilizada por el usuario para acceder a ésta.

A continuación dejaremos de hablar en términos hipotéticos y de explicar los conceptos básicos para introducirnos en la práctica. En el resto de este capítulo, veremos algo de código real y cómo realizar operaciones LDAP básicas con JNDI.

Operaciones LDAP

Antes de que podamos realizar cualquier tipo de operación en un servidor LDAP, debemos en primer lugar obtener una referencia y una conexión de red para el servidor LDAP. También debemos especificar cómo queremos estar seguros con respecto al servidor, bien de forma anónima o como un usuario autentificado. Muchos servidores LDAP accesibles a Internet permiten algún tipo de acceso anónimo (generalmente funciones de sólo lectura para atributos como direcciones de e-mail y números de teléfono), pero LDAP también se ajusta a características de seguridad muy avanzadas a través de ACL que dependen de quien autentifica la conexión.

Por ejemplo, un servidor LDAP puede tener diversos niveles de derechos para cualquier entrada determinada:

- Usuarios anónimos pueden acceder a la dirección de e-mail y al número de teléfono de una empleado.
- El empleado puede ver la entrada completa de dichos usuarios pero sólo puede modificar ciertos atributos como el número de teléfono, la contraseña y el número de despacho de la oficina.
- El gestor de un usuario puede actualizar el número de teléfono de un empleado y el número de despacho de la oficina pero ningún elemento más. También puede acceder al registro completo del empleado.
- Un grupo reducido, los Administradores del Directorio, tiene plenos derechos sobre el servidor completo incluida la posibilidad de añadir y eliminar cualquier entrada.

Operaciones LDAP estándar

Existen algunas operaciones estándar en LDAP. Son las siguientes:

- Conexión al servidor LDAP
- Asociación a un servidor LDAP (puede considerar este paso como autentificación)
- Realización de una serie de operaciones LDAP:
  - Búsqueda del servidor
  - Adición de una nueva entrada
  - Modificación de una entrada
  - Eliminación de una entrada
- Desconexión de un servidor LDAP

Analizaremos cada una de las operaciones anteriores, realizando demostraciones con ejemplos siempre que sea pertinente.

Conexión al servidor LDAP con JNDI

Primero debemos obtener una referencia para un objeto que implemente la interfaz `DirContext`. En la mayoría de las aplicaciones, utilizaremos el objeto `InitialDirContext` que tome una tabla de

Para asociar específicamente al servidor, debemos facilitar el medio con el método para nuestra autentificación (por ejemplo, "simple", SSL, o SASL). Después debemos especificar el DN de la entrada que deseamos asociar y la contraseña de la entrada:

```
Hashtable env = new Hashtable();

// Así envía en id y la contraseña como texto en la conexión
env.put(Context.PROVIDER URL, "myhost");
env.put(Context.SECURITY AUTHENTICATION,"simple");
env.put(Context.SECURITY PRINCIPAL, MGR DN);
env.put(Context.SECURITY CREDENTIALS, MGR PW);

// Obtener una referencia a un contexto de directorio
DirContext ctx = new InitialDirContext(env);
```

Seguridad simple, SSL/TLS y SASL

La última especificación LDAP, LDAP v3, permite tres tipos de seguridad:

- Simple
- SSL/TLS
- SASL

Estudiemos cada tipo sucesivamente.

Simple

Seguridad simple significa que el usuario sólo necesitará autentificar al servidor utilizando ID y contraseñas de texto estándar sin ningún cifrado en la red. Éste es con diferencia el más común y el menos seguro de los distintos métodos de autentificación. Es inseguro por dos motivos: el primero consiste en que los ID y contraseñas de los usuarios son transmitidas al servidor a través de una red pública, donde cualquiera puede robar los ID y las contraseñas de la red; en segundo lugar, nada garantiza que la persona que introduce el ID y la contraseña de usuario es el verdadero titular de ese ID y de esa contraseña.

SSL/TLS

El protocolo **Secure Socket Layer** fue desarrollado por Netscape Communications para mejorar la seguridad de las transacciones de base Web. Se ha convertido en una norma oficial llamada **Transport Layer Security** (**TLS**), pero todavía se denomina a menudo SSL.

SSL le permite cifrar su transacción completa en la red, dificultando a cualquiera la tarea de robar la información (como su ID o sus contraseñas). SLL estándar no verifica la identidad de la persona que introduce el ID y la contraseña de usuario pero sí certifica que la máquina desde la que el usuario está introduciendo el ID y la contraseña es "el verdadero servidor". La mayoría de los servidores que implementan SSL también se ajustan a los certificados cliente (los certificados son archivos de texto que son utilizados para confirmar la identidad del servidor y del cliente) para la autentificación del usuario. En vez de presentar un ID y una contraseña de usuario al sistema, puede presentar un certificado al servidor. Si el certificado que presenta coincide con un certificado permitido, el acceso está garantizado.

> *Los certificados se consideran más seguros porque son difíciles de falsificar. Sin embargo, los certificados se almacenan normalmente como un archivo en una máquina de usuario local, lo que significa que la máquina cliente está comprometida, entonces puede utilizarse un certificado del mismo modo que puede utilizarse un ID y una contraseña robada. Cuando los certificados son*

almacenados localmente, debe desarrollarse un mecanismo para recuperarlos si la máquina en la que están almacenados falla. Los certificados pueden ser almacenados en tarjetas inteligentes en lugar de archivos en una máquina local para mayor seguridad y fiabilidad. La emisión y gestión de certificados de cliente se encuentra todavía en fases tempranas de desarrollo.

SASL

Simple Authentication and Security Layer (SASL) es un estándar de Internet para implementar mecanismos de autentificación además de los métodos Simple o SSL.

Existen dos mecanismos SASL conocidos. Uno es el MD5 que ya hemos visto con anterioridad, basado en la comparación de contraseñas refundidas. MD5 no cifra la transacción y no resuelve el problema de "quién introduce la contraseña". Si la contraseña MD5 refundida es robada en su camino hacia el servidor, un hacker podría utilizarla para acceder al sistema, del mismo modo que si se tratara de texto. Sin embargo, sí hace que resulte más difícil adivinar la contraseña original por lo que, si un hacker roba la contraseña refundida, no podrá adivinar la contraseña original para intentar utilizarla para acceder a otros sistemas.

El segundo mecanismo SASL de uso extendido se denomina **Kerberos**. Kerberos cifra la transacción y, en una red con conocimiento de Kerberos, es muy fácil implementar un entorno de único registro debido al modo de funcionamiento de Kerberos. Sin embargo, gestionar una red Kerberos supone mucho tiempo y muchos recursos, por lo que muchas organizaciones no han implementado Kerberos todavía.

Puede grabar fácilmente sus propios mecanismos SASL, si su servidor LDAP se ajusta a ellos. Así, sería posible capacitar la autentificación biométrica, una vez esté disponible (o si su organización ya tiene acceso a la tecnología). La autentificación biométrica hace uso de parte del cuerpo del usuario, como su huella dactilar o su iris, para el ID y la contraseña de usuario. Estos sistemas son muy seguros.

Autentificación en LDAP v2 y LDAP v3

En LDAP v2, todos los clientes tenían que autentificarse a sí mismos antes de ejecutar cualquier operación. En LDAP v3, sin embargo, si un cliente no se autentificaba a sí mismo antes de ejecutar una operación, se asumía que la conexión era autentificada anónimamente.

Búsqueda en un servidor LDAP

La operación más común en cualquier servidor LDAP es la operación de búsqueda. Cualquier aplicación LDAP avanzada utiliza la búsqueda como funcionalidad central. En esencia, todas las funciones de búsqueda toman un controlador de conexión LDAP, la base desde la que iniciar la búsqueda, el espectro de la búsqueda y el filtro de la búsqueda. Un filtro de búsqueda es como una consulta SQL en la que indica al servidor los criterios a utilizar para encontrar entradas que cumplan los criterios. Las búsquedas siempre utilizan un nombre de atributo y un valor que buscar. Los filtros utilizan lógica y comodines booleanos. Algunos servidores, como el servidor de Directorio Netscape, se ajustan a capacidades de consulta incluso más avanzadas, del tipo "suena como", por ejemplo.

Para los ejemplos de búsqueda de esta sección, accederemos a los datos de muestra arius.com que están incorporados a Netscape Directory Server.

Filtros LDAP de ejemplo

Algunos filtros LDAP típicos aparecen en la siguiente tabla:

Descripción	Filtro LDAP
Buscar todos los usuarios con apellido Carter	`sn = Carter`
Buscar todos los usuarios con apellido que empiece por "Ca"	`sn = Ca*`
Buscar todas las clases de objeto del tipo `GroupofUniqueNames` que tengan "Managers" en su Nombre Común	`(&(cn = * Managers * ) (objectclass=groupofuniquenames))`

En JNDI, utilizamos el método de búsqueda de la interfaz `DirContext`. Esto devolverá un objeto `NamingEnumeration` si la búsqueda es realizada con éxito.

Más adelante, en esta sección, mostraremos los distintos modos de manipular los valores de este objeto para obtener atributos y valores de cada entrada recibida.

Determinar el alcance de LDAP

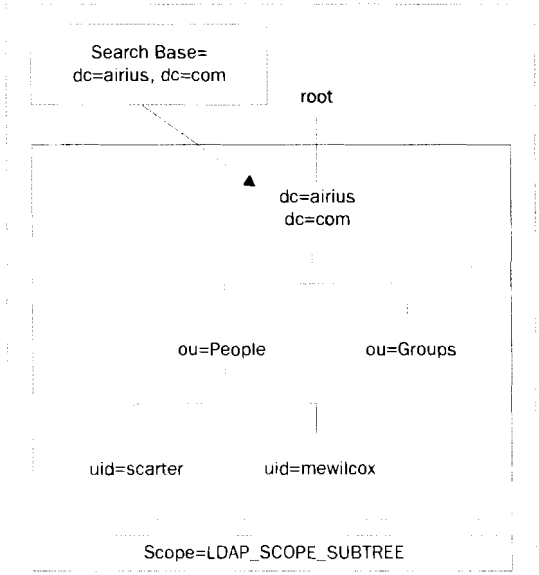
Cuando realiza una búsqueda, debe especificar el nodo (al que nosotros hacemos referencia como la **base**) del árbol que desea iniciar, así como el alcance la búsqueda. El alcance define exactamente qué proporción del árbol desea buscar. Hay tres niveles de alcance.

Alcance	Descripción
`LDAP_SCOPE_SUBTREE`	Este alcance comienza en la entrada base y busca todo lo que se encuentra por debajo, incluida la entrada base.
`LDAP_SCOPE_ONELEVEL`	Este alcance busca el árbol completo empezando un nivel por debajo de la entrada base. No incluye la entrada base.
`LDAP_SCOPE_BASE`	Este alcance busca únicamente la entrada base, útil si desea obtener sólo los atributos/valores de una entrada.

A continuación examinaremos estos tres niveles de alcance con la ayuda de algunos diagramas.

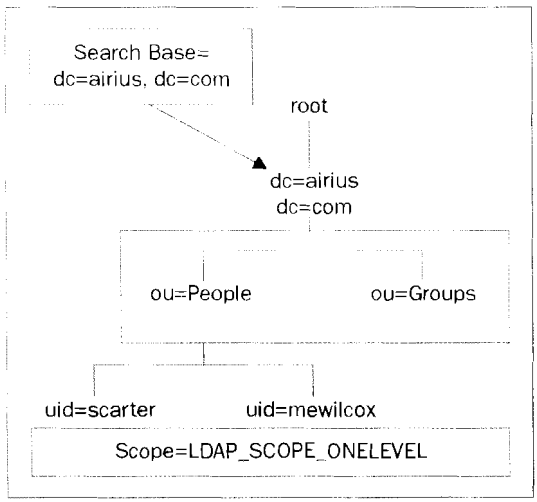
LDAP_SCOPE_SUBTREE

La primera figura, a continuación, ilustra una búsqueda utilizando `LDAP_SCOPE_SUBTREE`:



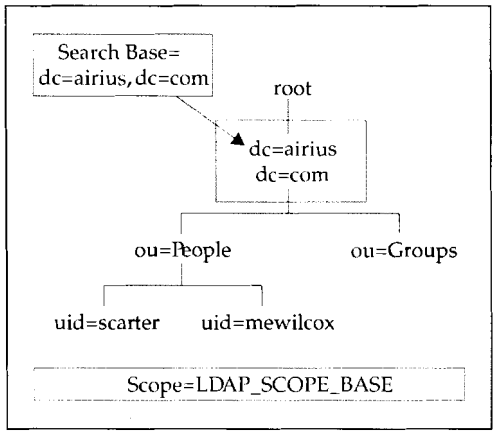
LDAP_SCOPE_ONELEVEL

Aquí ofrecemos un ejemplo de búsqueda utilizando `LDAP_SCOPE_ONELEVEL`:



LDAP_SCOPE_BASE

Finalmente, presentamos una búsqueda utilizando `LDAP_SCOPE_BASE`:



Ejecución de una búsqueda JNDI

Realizamos una búsqueda utilizando el método `search()` de un objeto que implementa la interfaz DirContext (como la clase `InitialDirContext`). El requisito mínimo para esto es la base de búsqueda y el filtro, aunque hay otros parámetros que podemos utilizar para ayudar a gestionar los resultados.

Deberíamos destacar que en la mayoría de los API de LDAP debemos especificar el alcance como un parámetro en el método de búsqueda LDAP. En JNDI, sin embargo, el alcance seconfigura en el objeto `SearchControls`, que es un parámetro opcional al de la función `search()` de la interfaz DirContext (la clase `InitialDirContext` proporciona una implementación de la función `search()`). Por defecto, está establecido en subtree. La búsqueda es en realidad ejecutada con cualquier objeto que tenga que implemente la interfaz `DirContext`, como la clase `InitialDirContext`.

Para nuestro primer ejemplo mostraremos un ejemplo de búsqueda muy sencillo, en el que el filtro de búsqueda es "`sn=Carter`". Éste devolverá todas las entradas que tengan un apellido (el atributo especificado por el mnemónico "`sn`") Carter. El primer ejemplo es una búsqueda anónima.

Los ejemplos no están basados en GUI para facilitar la comprensión de lo que está ocurriendo, pero no hay ningún motivo por el que no puedan ser incluidos en una GUI. Recuerde que todos los códigos fuente para los ejemplos descritos en este capítulo pueden ser descargados en http://www.wrox.com. El código queda como sigue:

```
// Búsqueda anónima estándar
import java.util.Hashtable;
import java.util.Enumeration;

import javax.naming.*;
import javax.naming.directory.*;

public class JNDISearch {

   // Implementación de contexto inicial
   public static String INITCTX = "com.sun.jndi.ldap.LdapCtxFactory";
   public static String MY HOST = "ldap://localhost:389";
   public static String MY SEARCHBASE = "o=Airius.com";
   public static String MY FILTER = "(sn=Carter)";
   public static void main(String args[]) {

     try {
```

```
            // Hashtable para información del medio
            Hashtable env = new Hashtable();

            // Especificar qué clase utilizar para nuestro proveedor JNDI
            env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);

            // Especificar host y puerto a utilizar para servicio de directorio
            env.put(Context.PROVIDER_URL, MY_HOST);

            // Obtener una referencia para un contexto de directorio
            DirContext ctx = new InitialDirContext(env);

            // Especificar el alcance de la búsqueda
            SearchControls constraints = new SearchControls();
            constraints.setSearchScope(SearchControls.SUBTREE_SCOPE);

           // Realizar la búsqueda real. Le otorgamos un base de búsqueda, un filtro
            // y las restricciones contenidas en el alcance de la búsqueda
            NamingEnumeration results = ctx.search(MY_SEARCHBASE,
                                                   MY_FILTER, constraints);

            // Ahora examinamos los resultados de la búsqueda
            while (results != null && results.hasMore()) {
               SearchResult sr = (SearchResult) results.next();

               String dn = sr.getName();
               System.out.println("Distinguished Name is " + dn);

               Attributes attrs = sr.getAttributes();

             for (NamingEnumeration ne = attrs.getAll(); ne.hasMoreElements();) {
                 Attribute attr = (Attribute) ne.next();
                 String attrID = attr.getID();

                 System.out.println(attrID + ":");
                 for (Enumeration vals = attr.getAll();vals.hasMoreElements();){
                    System.out.println("\t" + vals.nextElement());
                 }
              }
               System.out.println("\n");
            }
         } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
         }
      }
   }
```

Después de compilar y ejecutar este código, el resultado debe aparecer de modo similar a los resultados que aparecen en la siguiente captura de pantalla:



Cómo funciona el programa de búsqueda

Realizamos una búsqueda utilizando el método `search()` de un objeto que implementa la interfaz `DirContext` (como la clase `InitialDirContext`). El requisito mínimo para esta búsqueda es la base de búsqueda y un filtro. Existen otros parámetros que podemos utilizar para ayudar a gestionar los resultados. Si la búsqueda se realiza con éxito, se obtendrá un objeto `NamingEnumeration`.

Después de obtener el contexto inicial (al que fijamos la variable `ctx`), especificamos el alcance de la búsqueda. Si no especificamos un alcance, JNDI asumirá un alcance `subtree`, por lo que esta línea siguiente es en realidad redundante pero resulta útil para mostrar cómo especificar el alcance.

Analicemos ahora el proceso de funcionamiento de este código. En primer lugar, establecemos en alcance de `subtree`:

```
// Especificar el alcance de la búsqueda
SearchControls constraints = new SearchControls();
constraints.setSearchScope(SearchControls.SUBTREE_SCOPE);
```

Después de especificar el alcance, podemos realizar la verdadera búsqueda del siguiente modo:

```
// Ejecutar la verdadera búsqueda
NamingEnumeration results = ctx.search(MY_SEARCHBASE, MY_FILTER,
cMonstraints);
```

La clase `NamingEnumeration` es equivalente a la clase SearchResults de SDK Netscape Directory para Java.

Cada elemento del objeto `NamingEnumeration` contendrá un objeto `SearchResult` que podemos recuperar:

```
SearchResult sr = (SearchResult) results.next();
```

Después podemos obtener el DN de una entrada:

```
String dn = sr.getName();
```

Para obtener los atributos de una entrada utilice el método `getAttributes()` de la clase `SearchResults`:

```
Attributes attrs = sr.getAttributes();
```

Este método devolverá un objeto concreto que implementa la interfaz `Attributes` (la clase `InitialDirContext` devuelve un objeto`BasicAttributes`).

Después de conseguir un objeto `Attributes` (recuerde que es una clase de colección), podemos entonces repasar los atributos utilizando un objeto `NamingEnumeration` como éste:

```
for (NamingEnumeration ne = attrs.getAll(); ne.hasMoreElements();) {
    Attribute attr = (Attribute)ne.next();
    String attrID = attr.getID();

    System.out.println(attrID + ":");
    for (Enumeration vals = attr.getAll();vals.hasMoreElements();) {
        System.out.println("\t" + vals.nextElement());
    }
}
```

La clase `NamingEnumeration` nos ofrece métodos que podemos utilizar para examinar cada atributo obtenido en nuestra búsqueda. Cada elemento del objeto `NamingEnumeration` contendrá un objeto `Attribute` que representa un atributo y sus valores.

Cada elemento del objeto `Attribute` es un objeto que ha implementado la interfaz `Attribute` (la clase `InitialDirContext` utiliza objetos `BasicAttribute`). El método `getID()` de la interfaz `Attribute` devuelve el nombre del atributo. El método `getAll()` de la interfaz `Attribute` devolverá un objeto `Enumeration` Java estándar, al que podemos acceder para obtener los valores del atributo individual.

En todos los servidores LDAP hay ciertos atributos que no estarán disponibles para usuarios anónimos debido a los controles de acceso sobre el servidor. También hay ciertos atributos que puede que sólo estén disponibles para ciertos usuarios privilegiados (la escala salarial, por ejemplo, puede que sólo sea visible para el departamento de Recursos Humanos).

Búsqueda autentificada

El siguiente apartado muestra cómo podemos modificar el ejemplo para una búsqueda autentificada:

```
import java.util.Hashtable;
import java.util.Enumeration;

import javax.naming.*;
import javax.naming.directory.*;
public class JNDISearchAuth {

   public static String INITCTX = "com.sun.jndi.ldap.LdapCtxFactory";
   public static String MY_HOST = "ldap://localhost:389";
   public static String MGR_DN = "uid=kvaughan, ou=People, o=airius.com";
   public static String MGR_PW = "bribery";
   public static String MY_SEARCHBASE = "o=Airius.com";

   public static String MY_FILTER = "(sn=Carter)";
```

```
    public static void main(String args[]) {
      try {
          // Hashtable para información del medio
          Hashtable env = new Hashtable();

          // Especificar la clase a utilizar para nuestro proveedor JNDI
          env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);

          // Información de seguridad
          // Nos autentifica al servidor
          env.put(Context.PROVIDER_URL, MY_HOST);
          env.put(Context.SECURITY_AUTHENTICATION, "simple");
          env.put(Context.SECURITY_PRINCIPAL, MGR_DN);
          env.put(Context.SECURITY_CREDENTIALS, MGR_PW);

          // Resto de la clase sin alterar
          DirContext ctx = new InitialDirContext(env);

          SearchControls constraints = new SearchControls();
          constraints.setSearchScope(SearchControls.SUBTREE_SCOPE);

          NamingEnumeration results = ctx.search(MY_SEARCHBASE,
                                                 MY_FILTER, constraints);

          // Ahora repasar los resultados de la búsqueda
          while (results != null && results.hasMore()) {
             SearchResult sr = (SearchResult) results.next();

             String dn = sr.getName();
             System.out.println("Distinguished Name is " + dn);

           Attributes attrs = sr.getAttributes();

             for (NamingEnumeration ne = attrs.getAll(); ne.hasMoreElements();) {
                Attribute attr = (Attribute) ne.next();
                String attrID = attr.getID();

                System.out.println(attrID + ":");
                for(Enumeration vals = attr.getAll(); vals.hasMoreElements();){
                   System.out.println("\t" + vals.nextElement());
              }
            }
             System.out.println("\n");
          }
        } catch (Exception e) {
          e.printStackTrace();
          System.exit(1);
        }
      }
    }
```

Este segundo ejemplo de búsqueda es el mismo que el primero, excepto en que nos hemos autentificado al servidor.

Por ello, si intenta compilar y ejecutar este ejemplo, verá que produce exactamente el mismo resultado que antes. Observe que por defecto el servidor LDAP devuelve todos los atributos de la búsqueda. Sin embargo, puede que haya ocasiones en las que no deseemos estos resultados porque sólo nos interesen ciertos atributos.

Restringir los atributos presentados

En nuestro tercer ejemplo, solicitamos que sólo se muestren los atributos nombre común (`cn`) y dirección de e-mail (`mail`):

```
import java.util.Hashtable;
import java.util.Enumeration;

import javax.naming.*;
import javax.naming.directory.*;
public class JNDISearchAttrs  {

   // Implementación de contexto inicial
   public static String INITCTX = "com.sun.jndi.ldap.LdapCtxFactory";

   public static String MY_HOST = "ldap://localhost:389";
   public static String MY_SEARCHBASE = "o=Airius.com";

   public static String MY_FILTER = "(sn=Carter)";

   //Especificar qué atributos está buscando
   public static String MY_ATTRS[] = {"cn","mail"};

   public static void main(String args[]) {
     try {
        //Hashtable para información del medio
        Hashtable env = new Hashtable();

        //Especificar qué clase utilizar para nuestro proveedor JNDI
        env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);

        env.put(Context.PROVIDER_URL,MY_HOST);
        //Obtener una referencia a un contexto de directorio
        DirContext ctx = new InitialDirContext(env);

        SearchControls constraints = new SearchControls();
        constraints.setSearchScope(SearchControls.SUBTREE_SCOPE);

        NamingEnumeration results =
        ctx.search(MY_SEARCHBASE,MY_FILTER,constraints);

        while (results != null && results.hasMore()) {
           SearchResult sr = (SearchResult) results.next();
           String dn = sr.getName() + ", " + MY_SEARCHBASE;

           System.out.println("Distinguished Name is " + dn);

           Attributes ar = ctx.getAttributes(dn, MY_ATTRS);

         if (ar == null) {
            System.out.println("Entry " + dn +
                               " has none of the specified attributes\n");
         } else {
         for (int i =0;i<MY_ATTRS.length;i++) {
              Attribute attr = ar.get(MY_ATTRS[i]);
              if (attr != null) {
                 System.out.println(MY_ATTRS[i] + ":");
                 for (Enumeration vals = attr.getAll();vals.hasMoreElements();){
                     System.out.println("\t" + vals.nextElement());
               }
             }
```

```
                System.out.println("\n");
            }
        }
    } catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
  }
}
```

Como acabamos de especificar los atributos nombre común y mail, esta vez el resultado restringido será el siguiente:



La diferencia entre este código y nuestros ejemplos de búsqueda anteriores es que ahora limitamos el número de atributos que deseamos recuperar.

Hemos creado una matriz `String` que enumeraba los atributos que deseábamos:

```
public static String MY_ATTRS[] = {"cn","mail"};
```

Para recuperar este conjunto de atributos utilizamos en método `getAttributes()` de la interfaz `DirContext`, donde facilitamos el DN de una entrada específica y la matriz de atributos.

```
Attributes ar = ctx.getAttributes(dn, MY_ATTRS);
```

Esto devolverá un objeto Attribute.

Podemos entonces recuperar un objeto `Attribute` en particular de un objeto `Attributes`:

```
Attribute attr = ar.get("cn");
```

Llegados a este punto, deberíamos enfatizar que, aunque recuperar un conjunto concreto de atributos de una entrada particular es muy rápido, no resulta muy práctico para búsquedas generales. En una búsqueda LDAP general, el usuario final no va a conocer los Nombres Diferenciados existentes de las entradas que está buscando. Por ello, tendremos que buscar en un servidor LDAP y recuperar un conjunto de entradas.

En JNDI (en oposición a Netscape Directory SDK para Java), esta búsqueda devolverá todos los atributos asociados con cada entrada individual. Si realizamos entonces llamadas posteriores a `getAttributes()` para recuperar un subgrupo como en el ejemplo anterior, será necesaria otra llamada al servidor LDAP y recuperar el subgrupo de atributos. Este procedimiento es ineficiente porque requiere que utilicemos memoria extra para todos los atributos y ancho de banda extra para la comunicación extra. La memoria extra es necesaria porque nuestra aplicación debe mantener los datos de los resultados de la búsqueda LDAP para nosotros los procesemos.

Para mejorar el rendimiento en nuestras aplicaciones Java, debemos reducir el volumen de memoria que utilizamos porque el recolector de residuos de JVM puede tardar en reaccionar, o disminuir notablemente el ritmo de la aplicación, ya que reclama más memoria. A medida que mejora la capacidad del recolector de residuos y se reduce el precio de la memoria, puede que lo utilicemos sin pensárnoslo dos veces. Sin embargo, existe otra área potencial de problemas y se trata del hecho de que requiere que utilicemos múltiples conexiones de red, allí donde sólo habríamos utilizado una bajo un nivel más bajo de API LDAP.

Esto no significa que no debamos utilizar JNDI, sino que debemos comprender las implicaciones de su uso.

También podemos utilizar JNDI para añadir nuevas entradas al servidor, eliminar entradas y modificar las entradas existentes. A continuación, examinaremos estas operaciones.

Añadir entradas

Utilizar JNDI para añadir entradas a un servidor LDAP resulta, de hecho, más difícil que con otros SDK de LDAP. Esto se debe a que el objetivo primordial de JNDI es leer y escribir objetos Java en la red. Una consecuencia de esto es que un programador debe saltar obstáculos adicionales, como crear una clase Java para cada tipo de entrada que quiera añadir al servidor LDAP. Aquí estudiaremos cómo añadir y modificar una simple entrada en el servidor LDAP pero más adelante, en este mismo capítulo, aprenderemos cómo utilizar el servidor LDAP como memoria de un objeto.

Para almacenar una entrada en un servidor LDAP utilizando JNDI, debemos asociar un objeto a un Nombre Diferenciado (DN). Esto significa que cada objeto que almacenamos en el servidor (ya sea una sencilla entrada de persona o una clase Java serializada) debe tener un DN asociado. Recuerde que un DN es el nombre exclusivo que debe poseer cada entrada en un servidor LDAP. Si cambiamos a un servicio de directorio diferente (como NDS), todavía será necesario contar con un nombre exclusivo para cada objeto. Si todo esto le parece excesivo, sólo recuerde que se convertirá en algo secundario con el tiempo y que hacemos esto (proporcionar un nombre exclusivo) cada vez que guardamos un archivo en el disco duro. Ningún archivo del sistema de archivos puede compartir nombre con otro archivo. Si queremos tener dos archivos con el nombre `miarchivo.txt`, debemos guardarlos en directorios diferentes de nuestro sistema de archivos; de lo contrario, una version borrará a la otra.

Para almacenar incluso una simple entrada en el servidor LDAP, debemos crear una clase que implemente la interfaz `DirContext`. Esta interfaz define cómo debe ser almacenado el objeto (ya sea una persona o una clase Java serializada) en el servidor del directorio y recuperada del servidor. Por ejemplo, si tiene un objeto `Person`, su clase especificará cómo construir su DN, cómo almacenar los atributos disponibles (por ejemplo, nombre completo, dirección de e-mail, número de teléfono, ID de usuario, contraseña, etc.) y proporcionar varios mecanismos para manejar los datos recuperados. La interfaz `DirContext` también proporciona muchas formas sofisticadas de manejo de datos y es la interfaz básica para construir un proveedor de servicios de directorio.

Del mismo modo que con cualquier otro DSK LDAP, una operación de adición solamente puede ser ejecutada por un usuario autentificado con derechos para añadir una nueva entrada en el servidor. Los ACL de LDAP pueden ser configurados de modo que los usuarios sólo puedan añadir entradas en determinadas partes del árbol de directorio.

Nuestra siguiente muestra de código presenta una clase `Person` muy sencilla que implementa la interfaz `DirContext`. La mayoría de los métodos de la interfaz no están verdaderamente implementados (excepto para capturar excepciones) porque no los necesitamos en este ejemplo tan sencillo. Observe que esta clase se deriva del ejemplo `Drink.java` encontrado en el tutorial de JNDI (http://java.sun.com/products/jndi/tutorial).

Los métodos que implementamos aquí, `getAtributes()` y el constructor, nos permiten almacenar/recuperar los datos en una clase Person como entradas LDAP tradicionales. El resto de los métodos que no implementamos por completo son utilizados primordialmente para construir proveedores de servicios completos. En su lugar, simplemente capturamos excepciones afirmando que no nos ajustamos al servicio concreto. Los nuevos objetos también deben conformarse al esquema del servidor LDAP; de otro modo, las entradas no serán añadidas.

Resultará más fácil explicar cómo añadir una entrada con JNDI si explicamos en el código a medida que vamos avanzando. Una vez más, observe que el código fuente completo de este ejemplo está disponible para su descarga en la página Web de Wrox.

En primer lugar, nuestra declaración de clase. Fíjese en que explicamos que implementaremos los métodos para la interfaz `DirContext`:

```
import java.util.*;

import javax.naming.*;
import javax.naming.directory.*;

public class Person implements DirContext {
   String type;
   Attributes myAttrs;
```

A continuación, tenemos nuestro constructor. Este constructor toma varias cadenas que utilizaremos para construir una clase de objeto `inetOrgPerson`:

```
public Person(String uid, String givenname, String sn,
                  String ou, String mail) {
     type = uid;
```

Utilizaremos la clase `BasicAttributes` para almacenar nuestros atributos y sus valores. Al especificar `true` en el constructor `BasicAttributes` estamos indicándole que ignore el caso de nombres de atributos a la hora de realizar búsquedas de nombres de atributo:

```
     myAttrs = new BasicAttributes(true);
```

Para añadir un atributo multivalor, necesitamos crear un nuevo objeto `BasicAttribute`, que requiere el nombre del atributo en su constructor. Añadimos entonces los valores del atributo con el método `add()`:

```
Attribute oc = new BasicAttribute("objectclass");
     oc.add("inetOrgPerson");
     oc.add("organizationalPerson");
     oc.add("person");
     oc.add("top");

     Attribute ouSet = new BasicAttribute("ou");
     ouSet.add("People");
     ouSet.add(ou);
```

```
        String cn = givenname + " " + sn;
```

Finalmente, añadimos todos nuestros atributos al objeto `BasicAttributes`:

```
        myAttrs.put(oc);
        myAttrs.put(ouSet);
        myAttrs.put("uid", uid);
        myAttrs.put("cn", cn);
        myAttrs.put("sn", sn);
        myAttrs.put("givenname", givenname);
        myAttrs.put("mail", mail);
    }
```

Cuando `getAttributes()` es llamado, devuelve nuestro objeto `BasicAttributes` cuando es solicitado por un nombre en forma de cadena. Está diseñado para devolver únicamente los atributos de una entrada específica pero, puesto que esta clase sólo albergará una entrada, no va a ser llamada. Presentamos una manera de implementar si fuera del siguiente modo:

```
public Attributes getAttributes(String name) throws NamingException {
     if (! name.equals("")) {
        throw new NameNotFoundException();
     }
     return myAttrs;
  }
```

Este método tiene la misma función que el primer `getAttributes()` pero sólo es invocado cuando el nombre ha pasado el objeto `Name`:

```
public Attributes getAttributes(Name name) throws NamingException  {
     return getAttributes(name.toString());
  }
```

El siguiente método devuelve únicamente los atributos enumerados en la matrizd String ids. El nombre String debería ser un DN:

```
public Attributes getAttributes(String name, String[] ids)
                        throws NamingException {
   if (! name.equals("")) {
      throw new NameNotFoundException();
   }

      Attributes answer = new BasicAttributes(true);
      Attribute target;
      for (int i = 0; i < ids.length; i++) {
         target = myAttrs.get(ids[i]);
         if (target != null) {
            answer.put(target);
        }
   }
   return answer;
   }
```

Esta implementación es similar a las de otros métodos `getAttributes()`, excepto en que toma un objeto`Name`:

```
public Attributes getAttributes(Name name, String[] ids)
                        throws NamingException {
```

```
        return getAttributes(name.toString(), ids);
    }
```

El método `toString()` es utilizado para la codificación:

```
public String toString() {
    return type;
  }
```

Finalmente, las siguientes líneas se utilizan para implementar métodos que un proveedor de servicios JNDI (como la clase `InitialDirContext`) utilizaría para proveer a una aplicación de servicios tales como lectura de entradas desde el directorio o la autentificación al servidor:

```
// No utilizado para este ejemplo
  public Object lookup(Name name) throws NamingException {
    throw new OperationNotSupportedException();
  }

  public Object lookup(String name) throws NamingException {
    throw new OperationNotSupportedException();
  }

  public void bind(Name name, Object obj) throws NamingException {
    throw new OperationNotSupportedException();
  }
  ...
```

El código fuente completo para este ejemplo y para todos los ejemplos de este libro está disponible para su descarga en http://www.wrox.com/ o en http://www.AnayaMultimedia.es.

A continuación, demostramos la clase `JNDIAdd.java`, que utiliza la clase `Person`, desarrollada anteriormente, para añadir una entrada de Mark Wilcox al servidor LDAP (ya hemos visto gran parte de este código):

```
import java.util.Hashtable;
import java.util.Enumeration;

import javax.naming.*;
import javax.naming.directory.*;

public class JNDIAdd {

  // Implementación de contexto inicial
  public static String INITCTX = "com.sun.jndi.ldap.LdapCtxFactory";

  public static String MY_HOST = "ldap://localhost:389";
  public static String MGR_DN = "uid=kvaughan, ou=People, o=airius.com";
  public static String MGR_PW = "bribery";
  public static String MY_SEARCHBASE = "o=Airius.com";

  public static void main(String args[]) {
    try {
      // Hashtable para informaicón del medio
      Hashtable env = new Hashtable();
// Especificar qué clase utilizar para nuestro proveedor JNDI
      env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);

      env.put(Context.PROVIDER_URL,MY_HOST);
      env.put(Context.SECURITY_AUTHENTICATION,"simple");
      env.put(Context.SECURITY_PRINCIPAL,MGR_DN);
      env.put(Context.SECURITY_CREDENTIALS,MGR_PW);
```

```
            // Obtener una referencia para un contexto de directorio
            DirContext ctx = new InitialDirContext(env);

            Person p = new Person("mewilcox", "Mark",
                                  "Wilcox", "ou=Accounting",
                                  "mewilcox@airius.com");

            ctx.bind("uid=mewilcox,ou=People,o=airius.com", p);
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

En primer lugar, debemos crear un nuevo objeto Java que implemente la interfaz `DirContext` igual que nuestra clase `Person` del siguiente modo:

```
Person p = new Person("mewilcox", "Mark", "Wilcox",
                      "ou=Accounting", "mewilcox@airius.com");
```

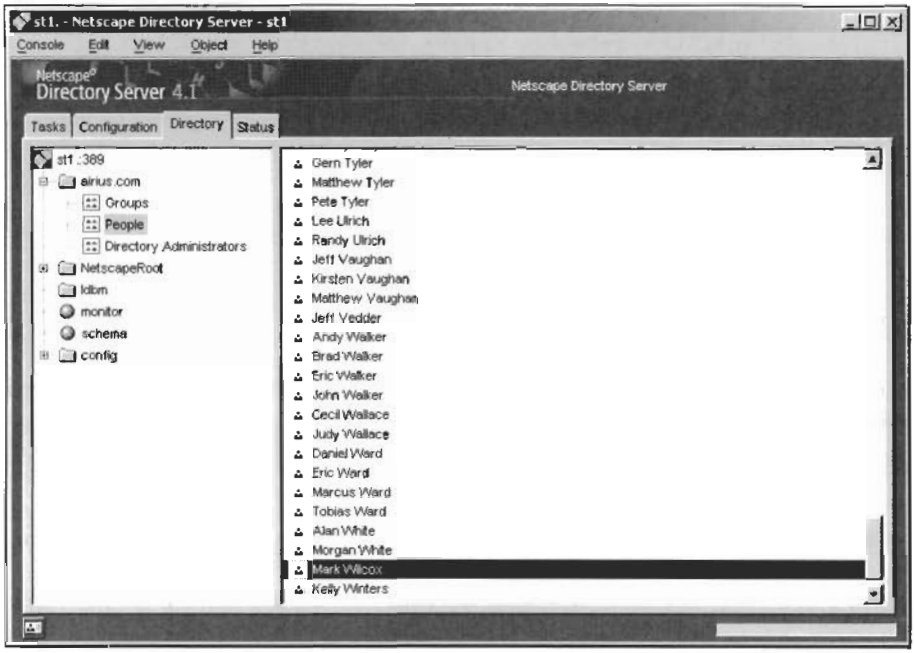
Entonces asociamos un nombre (específicamente el DN de la entrada) a este objeto en nuestro contexto actual con el método `bind()` de la interfaz `DirContext` del siguiente modo:

```
ctx.bind("uid=mewilcox,ou=People,o=airius.com", p);
```

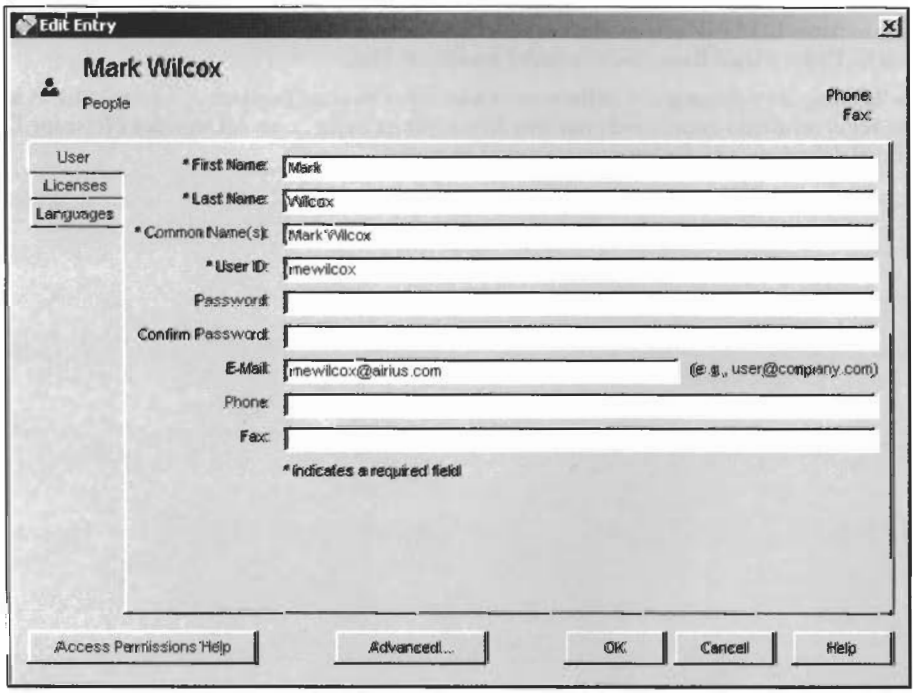
La interfaz `InitialDirContext` realizará una operación de adición LDAP. Tomará todos los atributos que hemos situado en nuestra clase Java y los codificará para su transferencia a un servidor LDAP.

Como hemos utilizado la clase `BasicAttributes` para construir nuestros atributos, éstos serán almacenados/recuperados como datos LDAP estándar y no como puros objetos Java. Esto significa que si almacena sus datos LDAP de este modo, cualquier otro cliente LDAP, independientemente de que esté escrito en C, Perl o Visual Basic, todavía podrá acceder a ellos.

Después de compilar y ejecutar con éxito nuestros archivos de clase `Person.java` y `JNDI.java`, podemos ver el resultado examinando nuestro directorio `airius.com` del servidor Netscape Directory Server. En él deberíamos poder ver nuestra nueva entrada:



Haciendo doble clic en el nombre destacado, podemos comprobar que todos nuestros detalles han sido añadidos de acuerdo con nuestro código:



Modificar una entrada

Tan pronto como añada una entrada a un servidor LDAP, necesitará modificarla. Los motivos pueden ser diversos, incluido un cambio en la contraseña de usuario, la actualización de la configuración de una aplicación, etc.

Las modificaciones realizadas a una entrada se hacen con las clases `ModificationItem` y `BasicAttributes`. Cuando realizamos una modificación, ésta puede ser de tipo ADD (añadir), REPLACE (reemplazar) o DELETE (borrar). La modificación REPLACE añadirá un atributo si éste no existe todavía.

> **También debemos ser conscientes de que si realizamos un reemplazamiento en un atributo que tiene múltiples valores sin enviar los valores extra junto con nuestro valor de reemplazamiento, todos los valores serán eliminados.**

Una vez más, las modificaciones deben ser realizadas por un usuario autentificado y tales modificaciones que pueden realizarse estarán determinadas por los derechos que la entrada asociada tiene sobre una entrada determinada. Por ejemplo, los usuarios pueden generalmente cambiar sus contraseñas pero no pueden cambiar nada más, mientras que los asistentes administrativos normalmente pueden modificar números de teléfono y direcciones de correo. Normalmente se necesita un administrador de bases de datos para realizar tareas tales como cambiar un ID de usuario.

El código que se muestra a continuación muestra cómo podemos modificar los atributos de la entrada de Mark Wilcox que añadimos en el ejemplo anterior:

```
import java.util.Hashtable;
import java.util.Enumeration;

import javax.naming.*;
import javax.naming.directory.*;

public class JNDIMod {

   // Implementación de contexto inicial
   public static String INITCTX = "com.sun.jndi.ldap.LdapCtxFactory";

   public static String MY_HOST = "ldap://localhost:389";
   public static String MGR_DN = "uid=kvaughan, ou=People, o=airius.com";
   public static String MGR_PW = "bribery";
   public static String MY_SEARCHBASE = "o=Airius.com";

   public static void main(String args[]) {

     try {
        // Hashtable para información del medio
        Hashtable env = new Hashtable();

        // Especificar qué clase utilizar para nuestro proveedor JNDI
        env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);

        env.put(Context.PROVIDER_URL, MY_HOST);
        env.put(Context.SECURITY_AUTHENTICATION, "simple");
        env.put(Context.SECURITY_PRINCIPAL, MGR_DN);
        env.put(Context.SECURITY_CREDENTIALS, MGR_PW);

        // Obtener una referencia para un contexto directorio
        DirContext ctx = new InitialDirContext(env);
```

```
        ModificationItem[] mods = new ModificationItem[2];

        Attribute mod0 = new BasicAttribute("telephonenumber", "940-555-2555");
        Attribute mod1 = new BasicAttribute("l", "Waco");

          mods[0] = new ModificationItem(DirContext.REPLACE_ATTRIBUTE, mod0);
          mods[1] = new ModificationItem(DirContext.ADD_ATTRIBUTE, mod1);

          // DirContext.DELETE_ATTRIBUTE not shown here
          ctx.modifyAttributes("uid=mewilcox,ou=People,o=airius.com", mods);
        } catch(Exception e) {
          e.printStackTrace();
          System.exit(1);
        }
      }
    }
```

Para modificar una entrada utilizamos la clase `ModificationItem`. La clase `ModificationItem` toma un tipo de modificación (por ejemplo ADD, REPLACE o DELETE) y un objeto `Attribute` como `BasicAttribute`. Éste es un ejemplo que nos permite añadir un nuevo atributo, localidad (el atributo `l`), con un nuevo valor de `"Waco"` en la entrada:

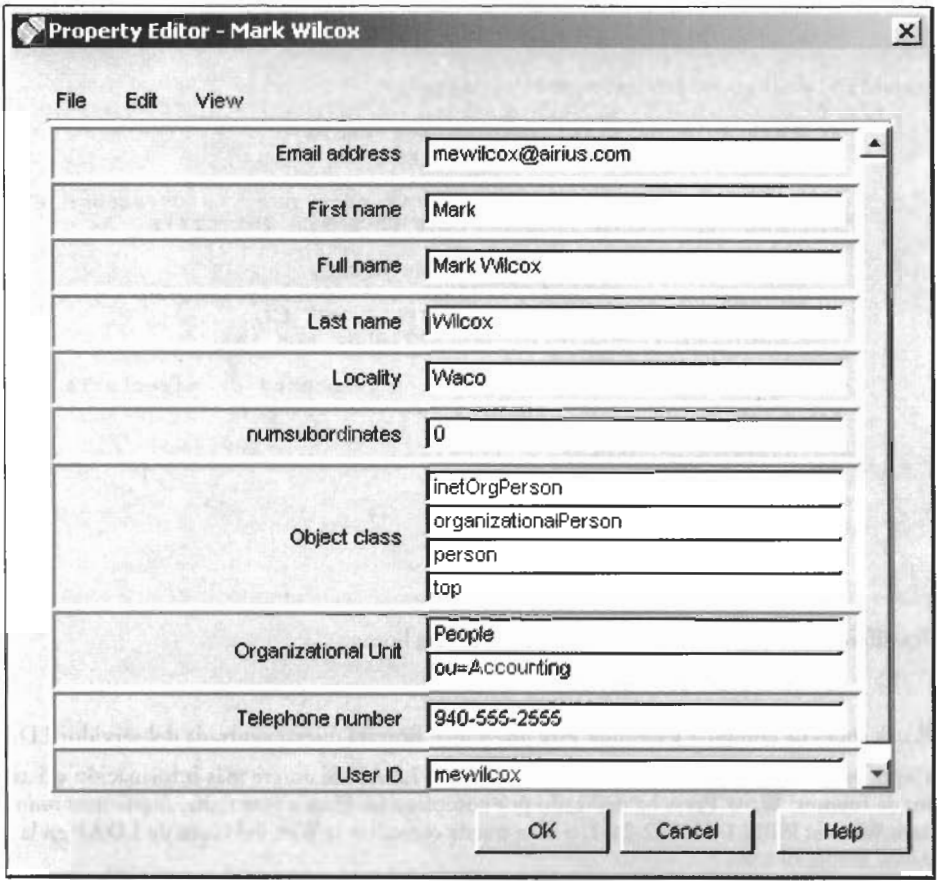
```
Attribute mod1 = new BasicAttribute("l", "Waco");
          mods[1] = new ModificationItem(DirContext.ADD_ATTRIBUTE, mod1);
```

La modificación real es realizada por el método `DirContext`, `modifyAttributes()`. A continuación, tenemos un ejemplo de este proceso:

```
ctx.modifyAttributes("uid=mewilcox,ou=People,o=airius.com", mods);
```

Una vez más, se modifica la entrada en un servidor LDAP utilizando LDAP tradicional y no como objeto de modo que cualquier otro cliente puede todavía acceder a estos datos.

Esta vez, cuando examinemos la entrada de Mark Wilcox en el Servidor Directorio y hagamos clic en `Advanced...`para obtener una visión detallada, veremos que el número de teléfono y la localidad han sido añadidos al directorio:



Eliminar una entrada

Finalmente, necesitará eliminar las entradas de su servidor LDAP. Esto se consigue fácilmente invocando el método `destroyContext()` de la interfaz `DirContext`, con el nombre diferenciado de la entrada que necesita ser eliminada. Normalmente, las operaciones de borrado están restringidas a los administradores de bases de datos LDAP.

Éste es nuestro ejemplo modificado para eliminar una entrada:

```
import java.util.Hashtable;
import java.util.Enumeration;
import javax.naming.*;
import javax.naming.directory.*;

public class JNDIDel {
   // Implementación de contexto inicial
   public static String INITCTX = "com.sun.jndi.ldap.LdapCtxFactory";

   public static String MY_HOST = "ldap://localhost:389";
   public static String MGR_DN = "cn=Directory Manager";
   public static String MGR_PW = "jessica98";
   public static String MY_SEARCHBASE = "o=Airius.com";
```

```
    public static String MY_ENTRY = "uid=mewilcox, ou=People,o=airius.com";

    public static void main(String args[]) {
      try {
          // Hashtable para información del medio
          Hashtable env = new Hashtable();

          // Especificar qué clase utilizar para nuestro proveedor JNDI
          env.put(Context.INITIAL_CONTEXT_FACTORY, INITCTX);

          env.put(Context.PROVIDER_URL, MY_HOST);
          env.put(Context.SECURITY_AUTHENTICATION, "simple");
          env.put(Context.SECURITY_PRINCIPAL, MGR_DN);
          env.put(Context.SECURITY_CREDENTIALS, MGR_PW);

          // Obtener una referencia para un contexto de directorio
          DirContext ctx = new InitialDirContext(env);

          ctx.destroySubcontext(MY_ENTRY);
      } catch(Exception e) {
          e.printStackTrace();
          System.exit(1);
      }
    }
}
```

La única diferencia real del código de este ejemplo es esta línea:

```
ctx.destroySubcontext(MY_ENTRY);
```

Por ello, después de compilar y ejecutar este programa, borrará nuestra entrada del servidor LDAP.

Hasta aquí, esto ha sido un vistazo a vuela pluma sobre LDAP. Si quiere más información existen un gran número de fuentes. Wrox Press ha dedicado por completo un libro a este tema, *Implementando LDAP* por Mark Wilcox, ISBN 1-261002-21-1, o bien puede consultar la Web del Gurú de LDAP en la dirección http://www.ldapguru.com/

Almacenar y recuperar objetos Java en LDAP

Uno de los atributos más importantes de JNDI es su capacidad para utilizar la red como almacén de objetos. Esto significa que podemos utilizar la red para compartir objetos Java con versiones distribuidas de una aplicación o con aplicaciones dispares. Por ejemplo, podríamos crear una clase que calcule el impuesto de venta sobre un artículo. Este tipo de clase podría ser utilizado potencialmente por nuestro departamento de ventas para crear órdenes de compra y por nuestra oficina de contabilidad para hacer sus cuentas.

Podemos utilizar casi cualquiera de los proveedores JNDI disponibles para almacenar nuestras clases, pero uno de los modos más útiles es utilizar un servidor para que realice este proceso.

Existen varias razones por las que preferiría utilizar LDAP como su almacén de datos:

- Ventaja de un recurso centralizado existente
- Ventaja de normas abiertas existentes
- LDAP está disponible en la red "fuera de la caja"
- LDAP está optimizado para un acceso de lectura extremadamente rápido

- LDAP tiene un fuerte sistema de seguridad incorporado

JNDI le permite almacenar diversos tipos de objetos relacionados con Java en el servidor LDAP:

- **Entradas del directorio LDAP estándar**
  Proporciona la capacidad de manipular los datos del directorio estándar (`inetOrgPerson`, `groupOfUniqueNames`, `objectclasses`, etc). Los datos del directorio estándar son de menor tamaño (y, por lo tanto, más fácil de acceder y de modificar) y pueden ser compartidos entre diferentes lenguajes. La capacidad de mantenerse neutral con respecto al lenguaje con datos de directorio es de vital importancia en una gran empresa, donde varios lenguajes diferentes (por ejemplo, Java, C/C++, Visual Basic, Cobol, etc.) pueden ser utilizados para el desarrollo de programas.

- **Objetos Java serializados**
  Nos proporciona la capacidad de almacenar y recuperar objetos Java que han sido serializados (los objetos actuales y todas las clases relacionadas y almacenadas en formato binario) en el servidor LDAP. Probablemente, el formato más fácil de utilizar pero también el que requiere mayor ancho de banda y espacio de almacenamiento. Este formato sólo es comprensible por Java en apoyo a la persistencia de objetos. Los dos usos más comunes de objetos Java serializados son los objetos EJB y JDBC DataSource.

- **Referencias**
  Como ya hemos debatido anteriormente, las plataformas de procesamiento distribuido como CORBA y RMI utilizan registros para registrar las clases de objetos que exponen. Sin embargo, estos registros no facilitan la capacidad de buscar en ellos y le obligan a codificar la posición del servidor en la aplicación. Un objeto `Reference` de Java permite almacenar datos en el servidor del directorio como referencia indirecta a la verdadera posición del objeto. Por ejemplo, podría tener un objeto almacenado en tiempo real en un servidor RMI. En lugar de codificar la posición del registro RMI en la aplicación, puede conectar a un servidor LDAP y localizar el objeto `"cn=Real Time Stock Quote"`, que contiene el URL RMI para el objeto stock. Esto reduce la necesidad de recompilar la clase cada vez que desplace la clase stock a un registro diferente y facilita la lectura del código.

La opción a utilizar para almacenar los objetos dependerá de la aplicación que está construyendo y de cómo necesita acceder a los datos. Estudiaremos estos tres objetos con más detalle en las siguientes secciones.

LDAP tradicional

Las organizaciones desearán a menudo acceder a los datos del servicio de directorios desde varios clientes utilizando lenguajes diferentes. Un uso muy extendido de un servicio directorio es el de autentificación de usuario. Resulta obvio que almacenar datos de autentificación de usuario en un formato que sólo Java puede utilizar reduce el número de aplicaciones que pueden hacer uso del directorio para autentificación. Esto, a su vez, aumenta el coste de las actividades empresariales porque en ese caso necesita otro servicio de directorio para facilitar servicios de autentificación a aplicaciones que no pueden acceder a objetos Java.

Algunas aplicaciones no necesitarán autentificación pero podrán beneficiarse del listín de direcciones del servicio de directorio. Por ejemplo, su programa de e-mail puede utilizarlo para encontrar la dirección de e-mail de compañeros de trabajo, su departamento de marketing puede utilizarlo para crear cartas de formato fusionado, mientras que los desarrolladores Web pueden utilizarlo para crear un portal adaptado a cada usuario. Cada una de estas aplicaciones puede construirse utilizando Java como lenguaje de desarrollo. Fundamentalmente, su compañía puede

reducir significativamente su sobregasto manteniendo datos constantes sobre sus empleados y sus clientes en una base de datos central.

Una de las ventajas de almacenar datos de este modo es que es posible tratar cada entrada como un objeto en Java pero otros lenguajes no tienen por qué están orientados a objetos para poder acceder a los datos.

Java serializado

Sin embargo, si tenemos un número creciente de aplicaciones Java que necesitan acceder al repositorio central de objetos Java preconstruidos, podemos utilizar la serialización Java para almacenar esos objetos en el servidor LDAP. Entonces, cuando una aplicación necesita un objeto determinado (por ejemplo, un dispositivo de representación 3D), puede recuperarlo del servidor LDAP, sólo cuando sea necesario, y luego recuperarlo. Otra característica positiva es que, a la hora de actualizar el dispositivo de representación, todas las aplicaciones que están utilizando el dispositivo tendrán acceso a la actualización sin tener que remendar las aplicaciones finales.

Referencias Java

Finalmente, si nos encontramos en un medio dominado por Java, puede que queramos aprovechar las ventajas de las referencias Java. Las referencias reducen los requisitos de almacenamiento y ancho de banda para almacenar y recuperar entradas porque no almacenan el objeto completo en el directorio; en su lugar, sólo almacenan componentes clave necesarios para reconstruir el objeto en una clase factoría. Con frecuencia, los componentes almacenados son un URL que apunta a la verdadera localización del objeto.

Por ejemplo, si creamos un objeto de impresora estándar, podemos construir un `PrinterFactory` que puede adoptar los siguientes parámetros:

- Localización de la red
- Opciones de color (monocromo, en color)
- Especialidades (postdatas, plotter gráfico)
- Estado actual (sin papel, on-line)

Estos parámetros pueden ser trasladados a `PrinterFactory` y siempre devolverán un objeto `Printer` que nuestra aplicación puede utilizar para imprimir. Reduce el ancho de banda y el almacenamiento porque sólo es necesario almacenar los parámetros anteriores. Resultan más fáciles de manejar para un programador de aplicaciones porque parte de la referencia es el nombre del paquete completamente cualificado, por lo que no tenemos que incluirlo con nuestra aplicación; sólo necesitamos asegurarnos de que JVM puede encontrar el paquete.

Si desea más información sobre el uso de JNDI y LDAP como almacén de objetos Java, puede dirigirse a la página oficial JNDI en http://java.sun.com/jndi.

Volver a JNDI sin LDAP

Aunque LDAP es una parte importante de una red de servicios de Internet y, a pesar de que mucha gente utiliza JNDI para poder utilizar LDAP en sus aplicaciones Java, LDAP no es la única opción disponible.

LDAP tiene sentido cuando es necesario gestionar información demográfica que no cambia pero que, con mucha frecuencia, es necesario acceder a ella muy rápidamente.

Sin embargo, hay ocasiones en las que se necesita acceder a datos que no pueden ser almacenados en LDAP, como es el caso de las entradas DNS o Enterprise JavaBeans.

Aunque el resto del capítulo se centra en el servicio de nombres de dominio, **Domain Naming Service (DNS)**, debemos señalar que fuera de LDAP, el lugar en el que utilizaremos JNDI con más frecuencia es EJB y otros servicios J2EE de un servidor de aplicación. Veremos algunos ejemplos más adelante.

La razón por la que nos centramos en DNS en este momento, en oposición a una muestra J2EE completa, es que los ejemplos de este capítulo esperan demostrar la flexibilidad de JNDI al utilizar ejemplos más sencillos de los que sería posible utilizar con EJB.

Ejemplo de aplicación DNS

Para demostrar cómo utilizaríamos las mismas clases JNDI básicas pero con diferentes proveedores de servicios, presentaremos una variación de la aplicación `JNDISearch` que ya utilizamos en nuestra sección LDAP.

> *Para esta sección se requiere el proveedor de servicios JDNI DNS que se encuentra disponible en Java Developer's Connection y que será incluido en JDK 1.4 (ya está disponible en la instalación de JDK 1.4 beta). Los archivos relevantes son* `dns.jary providerutils.jar`*, que deberían localizarse en classpath al ejecutar el programa.*

Tanto LDAP como DNS tienen entradas. A cada entrada se accede mediante un nombre exclusivo (en LDAP, es el Nombre Diferenciado, y en DNS, el nombre canónico). Ambos servicios son construidos en jerarquía; recuerde que ésta es una de las características de un servicio de directorio.

DNS tiene un modo perfectamente definido de construir el árbol (empezando por los dominios del nivel superior como `.com`, `.es`, `.net`, seguidos por dominios de segundo nivel como `yahoo.com`, seguidos por subdominios y anfitriones). LDAP, por otro lado, no tiene reglas establecidas para construir sus estructuras de árbol (por ello encontrará DNS tipo `o=Acme Inc`, `c=US` o `o=acme.com` o `dc=acme`, `dc=com`). Ambos servicios tienen entradas que contienen atributos con uno o más valores. LDAP tiene clases de objetos que pueden definir diferentes tipos de entradas. DNS sólo contiene entradas en referencia a anfitriones de Internet. LDAP tiene un potente modelo de seguidad basado en autentificación; DNS no lo tiene.

Ya hemos examinado los fundamentos de esta aplicación anteriormente, por lo que no invertiremos mucho tiempo explicando los detalles. Nos centraremos en las partes específicas de DNS.

```
import java.util.Hashtable;
import java.util.Enumeration;

import javax.naming.*;
import javax.naming.directory.*;

public class JNDISearchDNS {

   public static void main(String args[]) {
     try {
        // Hashtable para informaicón del medio
        Hashtable env = new Hashtable();
```

Observe que estamos cambiando nuestra factoría de proveedor de `com.sun.jndi.dns.LDAPCtxfactory` a `com.sun.jndi.dnsContextFactory`. Puesto que estamos cambiando nuestra factoría, necesitamos cambiar nuestro proveedor URL así como cambiar de `ldap://` a `dns://`. Los números `129.120.210.252` hacen referencia a la dirección IP del servidor DNS con el que nos comunicaremos:

```
// Especificar qué clase utilizar para nuestro proveedor JNDI
env.put("java.naming.factory.initial",
        "com.sun.jndi.dns.DnsContextFactory");
env.put("java.naming.provider.url", "dns://129.120.210.252/");
```

La variable instancia, `dns_attributes`, contiene la lista de atributos que desearíamos obtener de un determinado host, pero existe garantía alguna de que todos estos atributos estén a nuestro alcance. Por ejemplo, si el servidor no es un servidor de correo, no tendrá registro XM. O puede que el administrador DNS no tenga la información OS en el campo HINFO (abajo) por razones de seguridad (si un intruso puede determinar el OS host para una determinada máquina, entonces podría centrar su ataque utilizando herramientas para violar el OS en cuestión):

```
String dns_attributes[] = {"MX", "A", "HINFO"};
```

Aquí obtenemos el objeto inicial `DirContext` e intentamos recuperar los atributos para el host llamado `www.unt.edu`:

```
// Obtener una referencia para un contexto de directorio
DirContext ctx = new InitialDirContext(env);
Attributes attrs1 = ctx.getAttributes("www.unt.edu", dns_attributes);
```

A continuación, comprobamos que obtenemos algo de DNS. La razón por la que podemos no obtener resultados es que puede que no haya anfitriones que coincidan con nuestro hostname en el servidor DNS.

```
if (attrs1 == null) {
    System.out.println("host  has none of the specified attributes\n");
} else {
```

Ahora imprimimos nuestros resultados, utilizando el mismo algoritmo que hemos utilizado antes:

```
            for (int z = 0; z < dns_attributes.length; z++) {
               Attribute attr = attrs1.get(dns_attributes[z]);

               if (attr != null) {
                  System.out.print(dns_attributes[z]+": ");
                  for (Enumeration vals = attr.getAll();vals.hasMoreElements();) {
                   System.out.println(vals.nextElement());
                  }
                }

                System.out.println("\n");
               }
            }
         } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
         }
      }
   }
```

La captura de pantalla que aparece a continuación muestra los datos que probablemente reciba de DNS:



En este ejemplo, hay dos atributos para la entrada DNS que coinciden con www.unt.edu. Hay un registro A que indica la dirección IP del host y hay un registro HINFO, que afirma que el servidor ejecuta actualmente UNIX.

Como puede ver, el cambio de proveedores en JNDI resulta realmente fácil.

Resumen

Los servicios de directorio son una parte importante del entorno de servicios en red. Los directorios proporcionan la capacidad para asociar nombres comprensibles (y recordables) para el usuario con la información que los ordenadores necesitan procesar (por ejemplo, nombres de dominio para direcciones IP o nombres comunes para referencias Java). También nos permiten buscar direcciones de e-mail y asociar permisos para entornos seguros.

LDAP nos ofrece un protocolo de red estándar para comunicar con servicios de directorio, mientras que JNDI proporciona un API estándar para que Java se comunique con LDAP y con servicios de directorio especializados que no son accesibles vía LDAP (como DNS).

Después de esta breve introducción a los servicios de directorio, JNDI y LDAP, debe tener una visión general de los principios básicos y ya pueda aventurarse en su uso a partir de aquí.

Continuaremos nuestro viaje a través de algunos de los elementos fundamentales de la programación de empresa en el siguiente capítulo, examinando con detenimiento el procesamiento condicionado por RMI.

3
Procesamiento distribuido con RMI

El procesamiento informático distribuido se ha convertido en un término frecuente en el vocabulario de programación actual. Hace referencia al paradigma de diseño de aplicaciones en el que los programas, los datos que procesan y las computaciones reales se extienden por la red, bien para aprovechar el poder de procesamiento de múltiples computadoras o debido a la naturaleza inherente de una aplicación que comprenda diferentes módulos.

RMI (Remote Method Invocation) permite la comunicación objeto-objeto entre diferentes máquinas virtuales Java, Java Virtual Machines (JVM). Las JVM pueden ser entidades distintas situadas en el mismo o en diferentes ordenadores; incluso una JVM puede invocar métodos pertenecientes a un objeto almacenado en otra JVM. Esto permite a las aplicaciones invocar métodos de objetos localizados remotamente, compartir recursos y procesar cargas en sistemas. Los métodos pueden incluso aprobar objetos que una máquina virtual no ha encontrado nunca, permitiendo la carga dinámica de nuevas clases cuando sea necesario. Ésta es realmente una característica muy útil.

Si queremos que dos aplicaciones Java se ejecuten en diferentes máquinas virtuales para que se comuniquen entre sí, existen otros dos enfoques de base Java además de RMI:

- Sockets
- Java Message Service (JMS)

La programación básica de red con Sockets es flexible y suficiente para la comunicación entre programas. Sin embargo, requiere que todas las partes implicadas se comuniquen vía protocolos referentes a aplicaciones, cuyo diseño es complejo y puede ser susceptible de errores. Por ejemplo, consideremos una aplicación cooperativa, como una sencilla aplicación de chat, por ejemplo: para que participen múltiples clientes primero necesitaríamos diseñar algún tipo de protocolo y después utilizar Sockets para comunicarse en ese protocolo con el servidor. RMI, por otro lado, es un sistema distribuido que utiliza Sockets internamente sobre TCP/IP.

La diferencia entre RMI y JMS es que en RMI los objetos quedan residentes y están asociados a las máquinas virtuales (aunque los argumentos de método y los retornos, así como los stubs, viajan por la red), mientras que en JMS los mensajes (objetos) viajan asincrónicamente por la red desde una JVM a otra.

La arquitectura RMI

Antes de estudiar cómo funcionan los mecanismos RMI, definamos algunos términos utilizados con frecuencia.

> **Para citar la especificación, "En el modelo de objeto distribuido de Java, un objeto remoto es un objeto cuyos métodos pueden ser invocados desde otra máquina virtual de Java, potencialmente en un host diferente. Un objeto de este tipo es descrito por una o más interfaces remotas, que son interfaces Java que declaran los métodos de un objeto remoto. La invocación de métodos remotos, Remote Method Invocation (RMI), es la acción de invocar un método de una interfaz remota en un objeto remoto."**

El propósito de RMI es crear objetos en distintas JVM y actuar como objetos locales. La JVM que invoca al objeto remoto es conocida normalmente como un cliente y la JVM que contiene el objeto remoto es el servidor. Uno de los aspectos más importantes del diseño RMI está en su deseada transparencia. Las aplicaciones no saben si un objeto es remoto o local. Una invocación de método en un objeto remoto tiene la misma sintaxis que una invocación de método en un objeto local, aunque en el fondo sucede mucho de lo que alcanza la vista.

> **En RMI, el término "servidor" no hace referencia a un servidor físico o a una aplicación sino a un único objeto remoto que tiene métodos que pueden ser invocados remotamente. De un modo similar, el término "cliente" no hace referencia a una máquina cliente sino que en realidad se refiere al objeto invocando un método remoto en un objeto remoto. El mismo objeto puede ser tanto un cliente como un servidor.**

Aunque obtener una referencia a un objeto remoto es ligeramente diferente a obtenerla para objetos locales, una vez tenemos la referencia, utilizamos el objeto remoto como si fuera local. La infraestructura de RMI interceptará automáticamente la llamada del método, encontrará el objeto remoto y procesará la solicitud remotamente. Esta transparencia de posición incluye incluso recolección de residuos.

> **A un objeto remoto siempre se accede a través de su interfaz remota. En otras palabras, el cliente invoca métodos en el objeto sólo después de asignar la referencia a la interfaz remota.**

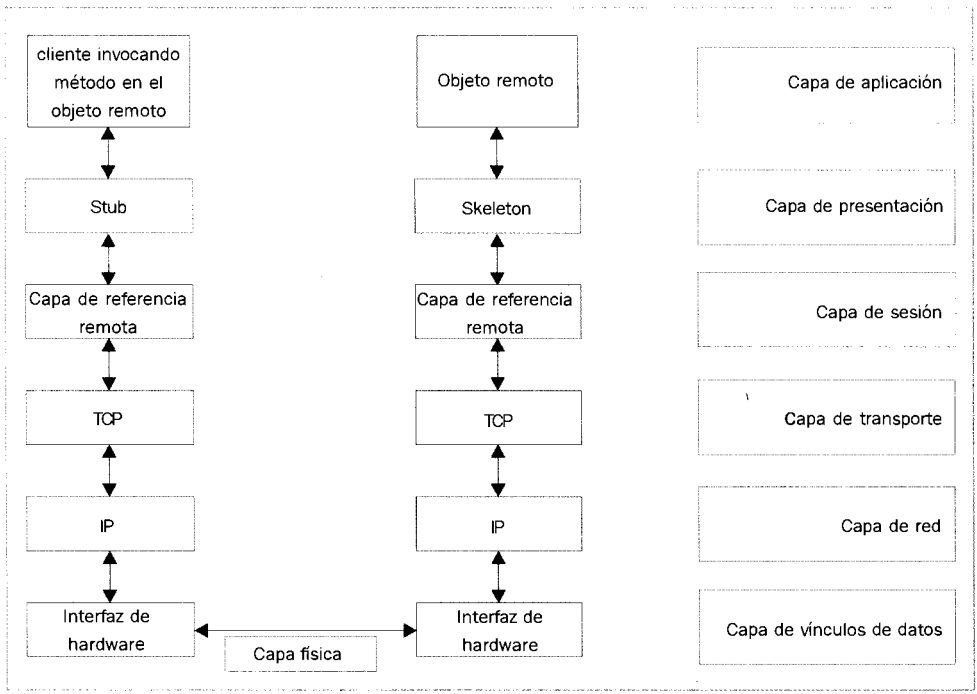
La implementación RMI es construida esencialmente a partir de tres capas de abstracción:

- **Capa de Stubs/Skeletons**
  Esta capa intercepta llamadas de método realizadas por el cliente a la referencia de la interfaz y redirige estas llamadas a un objeto remoto. Es conveniente recordar que los stubs son específicos del lado cliente, mientras que los skeletons se encuentran en el lado servidor.
- **Capa de referencia remota**

Esta capa maneja los detalles relacionados con la interpretación y la gestión de referencias realizadas por clientes a los objetos remotos. Conecta clientes con objetos remotos que están siendo ejecutados y exportados en un servidor por un vínculo de conexión uno-a-uno.

- **Capa de transporte**
  Esta capa está basada en conexiones TCP/IP entre máquinas de una red. Proporciona conectividad básica, así como algunas estrategias de penetración de cortafuego.

El siguiente diagrama muestra estas tres capas, junto con una interrupción de la capa de transporte. Las descripciones de la parte derecha representan las capas OSI (OSI está descrito de forma concisa en http://www.webopedia.com/TERMOS/OSI.html):



Esta arquitectura en capas proporciona una buena flexibilidad de implementación sin afectar a la arquitectura de aplicación. Cada una de las capas pueden ser reforzada o reemplazada sin afectar al resto del sistema. Por ejemplo, la implementación de la capa de transporte puede ser reemplazada por un distribuidor con el protocolo User Datagram Protocol (UDP) en lugar de TCP, sin afectar a las capas superiores.

A continuación, examinaremos algunas de estas capas más detenidamente.

Capa de Stub y Skeleton

Para conseguir la transparencia de posición, RMI introduce dos tipos especiales de objetos conocidos como stubs y skeletons que cumplen las funciones de una interfaz entre una aplicación y el resto del sistema RMI. El propósito de esta capa es transferir datos la Capa de Referencia Remota mediante marshaling (disposición) y unmarshaling. Marshaling hace referencia al proceso de convertir los datos u

objetos transferidos en flujo de bytes y desempaquetando, que es el proceso contrario, es decir, convertir el flujo en un objeto o en datos. Esta conversión se consigue mediante serialización de objetos.

La capa de stubs y skeletons de RMI se encuentra justo por debajo de la aplicación en sí y está basada en el modelo de diseño Proxy. En el uso RMI del modelo Proxy, la clase stub desempeña el papel del proxy para la implementación del servicio remoto. El skeleton es una clase asistente generada por RMI para ayudar al objeto a comunicarse con el stub en el vínculo RMI. El skeleton mantiene una conversación con el stub; lee los parámetros para la llamada de método desde el vínculo, realiza la llamada al objeto de implementación del servicio remoto, acepta el valor de vuelta y graba el valor de vuelta en el stub.

Resumiendo, el modelo Proxy fuerza las llamadas de método a través de un proxy que actúa como sustituto, delegando todas las llamadas en el objeto real de modo transparente para el llamador original.

> *Detalles específicos sobre el diseño de métodos proxy están más allá del objetivo de este libro, pero pueden encontrarse en Diseño de Métodos, Elementos reutilizables de software orientado a objetos Erich Gamma, et. al. (ISBN 0-201-63361-2) o métodos en Java: Un catálogo de diseños de métodos reutilizables (ISBN: 0-471333-15-8).*

Analicemos ahora los stubs y los skeletons con más detalle.

Stubs

El stub es un objeto del lado cliente que representa (o actúa como un proxy para) el objeto remoto. El stub tiene la misma interfaz, o lista de métodos, que el objeto remoto. Sin embargo, cuando el cliente invoca un método stub, el stub envía la solicitud mediante la infraestructura RMI al objeto remoto (vía skeleton), que es el que realmente la ejecuta. El siguiente apartado enumera la secuencia de eventos ejecutados por el stub en detalle:

- Inicia una conexión con la VM remota que contiene el objeto remoto
- Escribe y transmite (marshaling) los parámetros en la VM remota
- Espera el resultado de la invocación de método
- Lee (unmarshaling) el valor de vuelta o la excepción de retorno
- Devuelve el valor al llamador

El stub oculta la serialización de los parámetros del método (los parámetros deben ser serializables; el paso de parámetros será estudiado detenidamente posteriormente) y la comunicación en red con el objetivo de presentar al llamador un sencillo mecanismo de invocación.

En la VM remota, cada objeto remoto puede tener su skeleton correspondiente.

Skeletons

En el lado servidor, el objeto skeleton se encarga de los detalles de "posición remota" de modo que el objeto remoto no tenga que preocuparse por ellos. En otras palabras, podemos codificar un objeto remoto del mismo que si fuera local; el skeleton aísla al objeto remoto de la infraestructura RMI. Durante las solicitudes de método remoto, la infraestructura RMI invoca automáticamente el objeto skeleton de modo que éste pueda ejercer su poder. A continuación, una enumeración de los eventos en detalle:

- Lee (unmarshaling) los parámetros para el método remoto (recuerde que estos parámetros han sido escritos y transmitidos (marshaling) por el stub en el lado cliente)
- Invoca el método en la implementación del objeto remoto

- Escribe y transmite (marshaling) el resultado (valor devuelto o excepción) al llamador (que es entonces leído (unmarshaling) por el stub)

El JDK contiene la herramienta rmic que crea los archivos de clase para stubs y skeletons. Los detalles sobre rmic puede encontrarlos en el paquete de JDK u on-line en:
http://java.sun.com/j2se/1.3/docs/tooldocs/tools.html

Capa de referencia remota

La capa de referencia remota define y se ajusta a la semántica de invocación de la conexión RMI. Esta capa proporciona un objeto `java.rmi.server.RemoteRef` específico de JRMP (Java Remote Method Protocol: siguiente sección) que representa un descriptor para el objeto remoto. Un `RemoteStub` utiliza una referencia remota para realizar una invocación de método a un objeto remoto.

En JDK 1.2 y versiones posteriores, los stubs utilizan un único método, `invoke (Remote, Method, Object[], long)` en la referencia remota para llevar a cabo la escritura y transmisión (marshaling) de parámetros, la ejecución de método remoto y la lectura (unmarshaling) del valor devuelto.

La implementación Java 2 SDK de RMI añade una nueva semántica para la conexión cliente-servidor: objetos remotos activables (como veremos más adelante). Otros tipos de semántica de conexión también son posibles. Por ejemplo, con multidifusión, un único proxy podría enviar una solicitud de método a múltiples implementaciones simultáneamente y aceptar la primera respuesta (que mejoraría el tiempo de respuesta y, posiblemente, la disponibilidad). En el futuro, se espera que Sun añada semántica de invocación adicional a RMI.

Capa de transporte

La capa de transporte realiza las conexiones de red basadas en flujos de datos en TCP/IP entre JVM y es responsable de configurar y gestionar dichas conexiones. Aunque dos JVM estén funcionando en la misma computadora física, se conectan a través de su pila de protocolo de red TCP/IP de su ordenador host. RMI utiliza un protocolo de conexión llamado **Java Remote Method Protocol (JRMP)** sobre TCP/IP (una analogía es http sobre TCP/IP).

JRMP está especificado en http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmi-protocol.doc.html. *Es importante destacar que JRMP es específico de la "implementación" Sun. Implementaciones alternativas, como BEA WebLogic, NinjaRMI, ObjectSpace's Voyager, etc., no utilizan JRMP pero, en su lugar, utilizan su propio protocolo de conexión. Sun e IBM han desarrollado conjuntamente la siguiente versión de RMI, llamada RMI-IIOP, que está disponible con Java 2 SDK versión 1.3 (o por separado). En vez de utilizar JRMP, RMI-IIOP utiliza Object Management Group (OMG) Internet Inter-ORB Protocol (IIOP) para comunicar entre clientes y servidor. IIOP también permite la integración con objetos CORBA.*

A partir de JDK 1.2, el protocolo JRMP fue modificado para eliminar la necesidad de skeletons y, en su lugar, utilizar el proceso de reproducción para realizar la conexión al objeto de servicio remoto. De este modo, sólo necesitamos generar clases stub en implementaciones de sistema compatibles con JDK 1.2 y versiones anteriores. Para generar stubs, utilizamos la opción `-v1.2` con `rmic`.

Para una descripción detallada de cómo fue modificado JRMP para eliminar los skeletons, consulte las descripciones de CallData, Operation y Hash en la especificación JRMP en http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmi-protocol.doc3.html.

La reproducción es un API del paquete `java.lang.reflect` *en J2SE que permite al código Java descubrir información sobre los campos, métodos y constructores de clases cargadas en periodo de operación y operar sobre ellos.*

La capa de transporte RMI está diseñada para realizar una conexión entre clientes y servidor, enfrentándose incluso a obstáculos de la red. Aunque la capa de transporte prefiere utilizar múltiples conexiones TCP/IP, algunas configuraciones de red conciben una única conexión TCP/IP entre un cliente y servidor (por ejemplo, algunos navegadores restringen los applet a una sola conexión de red de vuelta a su servidor host). En este caso, la capa de transporte multiplexa múltiples conexiones virtuales en una única conexión TCP/IP.

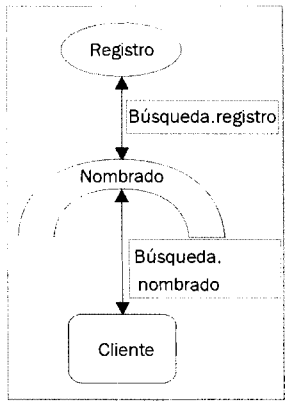
La capa de transporte en la implementación RMI actual está basada en TCP pero, una vez más, una capa de transporte basada en UDP puede ser sustituida en una implementación diferente.

Localización de objetos remotos

Hay todavía una pregunta crucial a la que todavía no hemos respondido. ¿Cómo encuentra un cliente el objeto? Los clientes encuentran servicios remotos utilizando un servicio de designación o de directorio. Esto puede parecer una lógica circular. ¿Cómo puede un cliente encontrar el servicio de designación? Es sencillo: un servicio de designación o de directorio es ejecutado en un número de host y de puerto que el cliente ya conoce (por ejemplo, un puerto conocido en un host público). RMI puede buscar claramente estos servicios diferentes de directorio con Java Naming and Directory Interface (JNDI), que ya ha sido descrito en el capítulo anterior.

El servicio de designación RMI, un registro, es un objeto remoto que sirve como servicio de directorio para clientes manteniendo una asociación de nombres tipo Hashtable con otros objetos remotos. No es necesario tener un único registro en un determinado host físico. Un objeto es libre para empezar su propio registro. El rendimiento del registro lo define la interfaz `java.rmi.registry.Registry`. RMI incluye una sencilla implementación de esta interfaz llamada Registro RMI (la herramienta `rmi registry` con JDK también puede ser iniciada programáticamente). El Registro RMI se ejecuta en cada máquina que "alberga" objetos remotos y acepta solicitudes de servicios, por defecto en el puerto 1009.

En términos sencillos, un objeto remoto es asociado a un nombre en este registro. Siempre que los clientes deseen invocar métodos en este objeto remoto, obtienen una referencia a éste buscando el nombre. La búsqueda devuelve una referencia remota, un stub, al objeto. RMI también proporciona otra clase, la clase java.rmi.Naming que sirve como punto de interacción del cliente con el objeto que sirve de registro en el host para esta búsqueda:



Es importante no confundir java.rmi.Naming con el contexto JNDI. Aunque proporcionan los mismos servicios, la clase Naming localiza específicamente objetos en el registro RMI. Sun facilita un proveedor JNDI para RMI que permite a los clientes buscar objetos RMI utilizando un contexto JDNI. Este registro RMI de base JNDI puede ser descargado desde http://java.sun.com/products/jndi/

Los métodos de la clase `Naming` toman, como uno de sus argumentos, un nombre que es un `java.lang.String` formateado como URL. Examinemos estos métodos y sus descripciones:

Método	Descripción
`Public static void bind (String name, Remote obj)`	Asocia el objeto remoto a un nombre string. El nombre mismo es el formato RMI URL descrito a continuación.
`Public static String[]list (String name)`	Devuelve un array de los nombres asociados en el registro.
`Public static Remote lookup (String name)`	Devuelve una referencia, un stub, para el objeto remoto asociado al nombre específico.
`Public static void rebind (String name, Remote obj)`	Reasocia (disocia el nombre si ya está asociado y lo vuelve a asociar) el nombre especificado si ya está en uso en un Nuevo objeto remoto. Esto puede ser peligroso si diferentes aplicaciones utilizan el mismo nombre en el registro pero resulta útil en el desarrollo.
`Public static void unbind (String name)`	Elimina la asociación con el nombre especificado.

Los métodos en la clase `Naming` y en la interfaz `Registry` tienen idénticas firmas y capturan varias excepciones (que estudiaremos en la siguiente sección). Hay una serie de acontecimientos que tienen lugar cuando un cliente invoca una búsqueda para un determinado URL en la clase `Naming`. Primero, se abre una conexión Socket al host en el puerto especificado (utilizando una factoría de Socket cliente si es necesario). A continuación, puesto que la implementación de registro en el host mismo es un objeto remoto, se devuelve un stub desde el host al registro remoto. Este stub actúa como el proxy cliente para el registro. Posteriormente, se realiza la búsqueda `Registry.lookup()` en este stub y devuelve otro stub para el objeto remoto que ha sido registrado con él en el servidor. Finalmente, una vez que el cliente tiene un stub para el objeto solicitado, interactúa directamente con el objeto en el puerto al que ha sido exportado.

El URL adopta la siguiente forma:

```
rmi://<host_name>[:<name_service_port>]/<service_name>
```

donde:

- `host_name` es un nombre reconocido en la red de área local (LAN) o un nombre DNS en Internet
- `name_service_host` necesita ser especificado sólo si el servicio de designación está siendo ejecutado en un puerto distinto del puerto por defecto 1099
- `service_name` es el nombre string al que se asocia el objeto remoto en el registro

Para facilitar este proceso en la máquina host, un programa servidor expone el servicio de objeto remoto ejecutando la siguiente secuencia de eventos:

1. Crea un objeto local
2. Exporta ese objeto para crear un servicio de escucha, que espera a que los clientes conecten y soliciten el servicio
3. Registra el objeto en el Registro RMI con un nombre público

El código expuesto a continuación resume estos pasos:

```
// Crear el objeto. La exportación tiene lugar en el constructor
HelloServer obj = new HelloServer();

// Asociar el objeto a un nombre en el registro que ha sido
// iniciado con anterioridad
Naming.rebind("/HelloServer", obj);
```

Existe una clase similar a la clase Naming, `java.rmi.registry.LocateRegistry`, que tiene varios métodos para obtener directamente una referencia al registro y para iniciar el registro. Como ya hemos indicado, iniciar el registro (ya sea mediante la herramienta o programáticamente) sólo consiste en exportar el objeto remoto que implementa la interfaz `Registry`.

> **El método `LocateRegistry.getRegistry` (String host) no contacta con el registro en el host sino que busca en el host para certificar que existe. Por ello, aunque este método dió resultado, esto no significa necesariamente que un registro sea ejecutado en el host especificado. Únicamente devuelve un stub que puede entonces acceder al registro.**

JDK tiene una herramienta llamada `rmiregistry` que inicia el registro en el host con el siguiente comando:

```
rmiregistry -J-Djava.security.policy=<policy file>
```

En cambio, para iniciar el registro con sus valores por defecto:

```
start rmiregistry
```

Recuerde que el registro es un objeto que se ejecuta en una JVM. El marcador `-J` se utiliza para pasar parámetros, como el **archivo de política**, a JVM.

> *También puede iniciar programáticamente el registro con el método `LocateRegistry.createRegistry (int port)` o el más detallado `LocateRegistry.createRegistry (int port, RMIClientSocketFactory csf, RMIServerSocketFactory ssf)`. Ambos métodos devuelven el stub, implementando la interfaz `java.rmi.registry.Registry`.*

Archivos de política

Al código se le conceden permisos en lo que se denomina archivos de política. Si buscamos en el directorio `%JAVA_HOME%\jre\lib\Security`, encontraremos el archivo de política por defecto para nuestra JVM llamado `java.policy`. Este archivo puede ser editado manualmente o utilizando el programa de herramientas de política que encontramos en `%JAVA_HOME%\bin_directory`.

Antes de continuar con un ejemplo, nos desviaremos para examinar las excepciones en RMI.

Excepciones RMI

Como ya hemos mencionado anteriormente, RMI es un sistema distribuido que utiliza Sockets sobre TCP/IP. En este entorno redificado, pueden fallar muchos elementos. Es importante que el cliente conozca de algún modo las excepciones de modo que puede manejarlas del modo adecuado y así solventar problemas a medida que vayan surgiendo. Por ejemplo, no queremos que un cliente espere indefinidamente una entrada en el Socket si la red falla o no es posible alcanzar el host.

Para que el cliente sea consciente de tales condiciones, todo método remoto debe lanzar la excepción `java.rmi.RemoteException` (o una de sus clases superiores como `java.io.IOException` o `java.lang.Exception`). `RemoteException` en una excepción genérica y hay subclases especializadas que son lanzadas (y capturadas) en condiciones específicas. Estas excepciones están descritas en la siguiente tabla:

Excepción	Descripción
`AccessException`	Lanzada por ciertos métodos de la clase `java.rmi.Naming` (concretamente `-bind()`, `rebind()` y `unbind()`) y métodos de la interfaz `ActivationSystem` (que estudiaremos en breve), para indicar que el llamador no tiene permiso para ejecutar la acción solicitada por la llamada del método.
`AlreadyBoundException`	Lanzada si existe un intento de asociar un objeto en el registro a un nombre que ya tiene una asociación. Observe que esta excepción no es lanzada (en las mismas condiciones) si se utiliza `rebind()`. Sin embargo, el uso de `rebind()` puede no ser correcto en todos los casos.
`ConnectException`	Lanzada si una conexión es rechazada el host remoto para una llamada de método remoto.
`ConnectIOException`	Lanzada si una `IOException` tiene lugar durante la realización de una conexión al host remoto para una llamada de método remoto.
`MarshalException`	Lanzada si ocurre una `java.io.IOException` durante la escritura y trasmisión (marshaling) de la cabecera, argumentos, o valor de retorno de llamada remota para una llamada de método remoto.
`NoSuchObjectException`	Lanzada si se realiza un intento de invocar un método sobre un objeto que ya no existe en la máquina virtual remota. Si se lanza esta excepción significa que el objeto ha sido recogido por el recolector de residuos o no exportado. Suele ocurrir que el cliente o clientes no consiguen repetidamente renovar sus alquileres y éstos expiran. Si esto ocurre, por ejemplo, la red se obstruye con tráfico o falla. Es importante también recodar que el stub con el cliente sólo es válido mientras esté vivo el servidor RMI. Si no desea que el objeto remoto sea recolectado como residuo, debe mantenerlo vivo en el JVM servidor almacenando una referencia al objeto de implementación en una variable estática. Como objeto disponible localmente, no será recolectado como residuo ni siquiera incluso en el caso de que los clientes sean ampliamente inalcanzables.
`NotBoundException`	Lanzada si realiza un intento de `lookup()` (buscar) o `unbind()` (desasociar) en el registro un nombre que no tiene ningún vínculo asociado.

La tabla continua en la página siguiente

Excepción	Descripción
`RemoteException`	La super clase común para una serie de excepciones relacionadas con la comunicación que puedan tener lugar durante la ejecución de una llamada de método remoto.
`ServerError`	Lanzada como resultado de una llamada de método remoto si la ejecución del método remoto en la máquina servidor lanza `java.lang.Error`.
`ServerException`	Lanzada como resultado de una llamada de método remoto si la ejecución del método remoto en la máquina servidor lanza `RemoteException`.
`StubNotFoundException`	Lanzada si una clase stub válida no pudiera ser encontrada para un objeto remoto al ser exportado.
`UnexpectedException`	Lanzada si el cliente de una llamada de método remoto recibe, como resultado de la llamada, una excepción comprobada que no se encuentra entre los tipos de excepciones comprobadas declaradas en la claúsula "throws" del método en la interfaz remota.
`UnknownHostException`	Lanzada si tiene lugar `java.net.UnknownException` durante la creación de una conexión al host remoto para una llamada de método remoto.
`UnmarshalException`	Puede ser lanzada durante el proceso de unmarshaling (lectura) de los parámetros o de los resultados de una llamada de método remoto si: ❑ Ocurre una excepción durante la lectura (unmarshaling) de la cabecera de llamada. ❑ Si el protocolo para el valor de retorno no es válido. ❑ Si ocurre una IOException durante la lectura (unmarshaling) de los parámetros (en el lado servidor) o en el valor de retorno (en el lado cliente).

Desarrollo de aplicaciones con RMI

Grabar aplicaciones cliente-servidor utilizando RMI conlleva seis pasos fundamentales:

1. Definir una interfaz remota
2. Implementar la interfaz remota
3. Grabar el cliente que utiliza los objetos remotos
4. Generar stubs (proxys de cliente) y skeletons (entidades de servidor)
5. Iniciar el registro y registrar el objeto
6. Ejecutar el servidor y el cliente

Estudiemos cada paso desarrollando una sencilla aplicación como ejemplo para obtener experiencia práctica y acelerar la comprensión de cada paso.

Definir la interfaz remota

Una interfaz manifiesta las operaciones expuestas y el programador cliente no necesita tener conocimiento de la implementación (la interfaz en este caso también sirve como marcador para la JVM). Una interfaz remota, por definición, es un conjunto de métodos que pueden ser invocados remotamente por un cliente:

- La interfaz remota debe ser declarada pública o el cliente obtendrá un error al intentar cargar un objeto remoto que implemente la interfaz remota, a no ser que el cliente esté en el mismo paquete que la interfaz remota.
- La interfaz remota debe ampliar la interfaz `java.rmi`.
- Cada método debe lanzar una excepción `java.rmi.RemoteException` (o una superclase de `RemoteException`).
- Si los métodos remotos tienen cualquier objeto remoto como parámetros o tipos devueltos, deben ser tipos de interfaces, no clases de implementación.

Observe que la interfaz `java.rmi` *no tiene métodos. Sólo se utiliza como un marcador por la JVM, de un modo similar a como la interfaz* `java.io.Serializable` *es utilizada para marcar objetos como serializables.*

Para nuestro ejemplo, definiremos nuestra interfaz remota de este modo:

```
public interface HelloInterface extends java.rmi.Remote {

   // Este método es llamado por clientes remotos y es implementado
   // por el objeto remoto.
   public String sayHello() throws java.rmi.RemoteException;
}
```

Si su objeto implementa otras interfaces, manténgalo separado de la interfaz remota. También puede reducir al mínimo el número de objetos registrados en el registro con un modelo de diseño Factoría o con un único objeto factoría.

Implementar la interfaz remota

La clase de implementación es la clase real que proporciona la implementación para métodos definidos en la interfaz remota. `Java.rmi.server.RemoteObject` amplía la funcionalidad proporcionada por la clase `java.lang`.Object en el dominio remoto ignorando los métodos `equals()`, `hashcode()` y `toString()`.

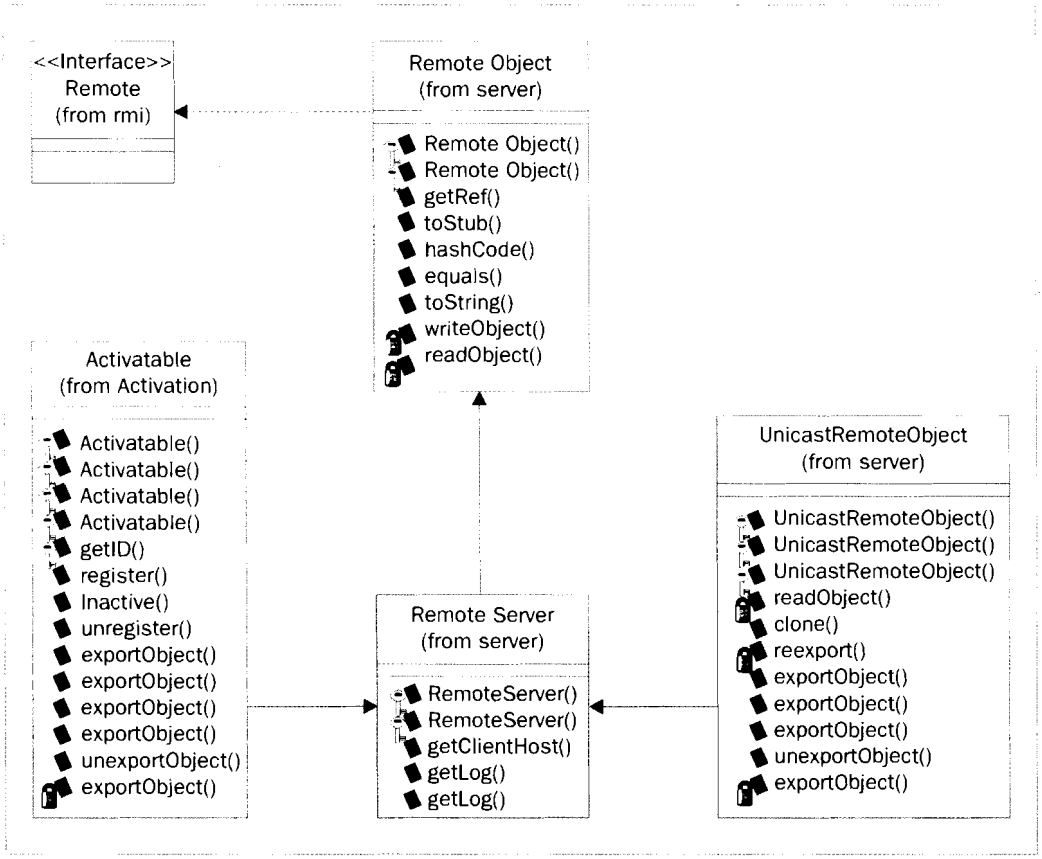
Recuerde que el genérico `java.rmi.server.RemoteObject` *es una clase abstracta y describe el funcionamiento de objetos remotos.*

La subclase abstracta `java.rmi.server.RemoteServer` describe el funcionamiento asociado con la implementación del servidor y proporciona la semántica básica para ajustarse a referencias remotas (por ejemplo, crear, exportar a un puerto determinado, etc.).

Java.`rmi.server.RemoteServer` tiene dos subclases:

- `java.rmi.server.UnicastRemoteObject`: define un objeto remoto no replicado cuyas referencias sólo son válidas mientras está el servidor esté activo.
- `java.rmi.Activation.Activatable`: clase concreta que define el funcionamiento para instanciado a solicitud de objetos remotos (véase sección posterior sobre activación).

El siguiente diagrama muestra las dos subclases y sus métodos contenidos en relación con la clase `java.rmi.server.RemoteServer`:



Un objeto puede presentar funcionamiento remoto como resultado de los siguientes factores:

- La clase amplía `java.rmi.RemoteServer` o una de sus subclases (incluidos otros objetos remotos). Sin embargo, la clase debe invocar uno de los constructores superclase de modo que pueda ser exportada.
- La clase se exporta a sí misma explícitamente pasándose a sí misma (`"this"`) a diferentes formas de métodos `UnicastRemoteObject.exportObject()`.

El término "exportar" encierra la semántica que implica la capacidad de un objeto remoto para aceptar solicitudes. Esto conlleva escuchar en un Socket (servidor) TCP. Observe que múltiples objetos pueden escuchar en el mismo puerto (véase la sección sobre Sockets para más información).

Además de éstas, la clase debe implementar una o más interfaces remotas que definen los métodos remotos.

> **Una clase remota puede definir cualquier método pero sólo los métodos en la interfaz remota pueden ser invocados remotamente.**

La clase de implementación de objeto remoto `HelloServer` para nuestro ejemplo es del siguiente modo:

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

public class HelloServer extends UnicastRemoteObject
                         implements HelloInterface {

   public HelloServer() throws RemoteException {
      super();          // Llamar al constructor superclase para exportar este objeto
   }

   public String sayHello() throws RemoteException {
      return "Hello World, the current system time is " + new Date();
   }
}
```

Observe que el código fuente para este ejemplo (y para todos los ejemplos de este libro) está disponible para su descarga en http://www.wrox.com *y en* http://www.AnayaMultimedia.es.

Grabar el cliente que utiliza los objetos remotos

El cliente realiza una búsqueda en el registro en el host y obtiene una referencia al objeto remoto. Recuerde que es de vital importancia difundir a la interfaz remota. En RMI, los clientes siempre interactúan con la interfaz, nunca con la implementación del objeto:

```
import java.rmi.*;

public class HelloClient {

   public static void main(String args[]) {
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new RMISecurityManager());
      }
      try {
         HelloInterface obj = (HelloInterface) Naming.lookup("/HelloServer");
         String message = obj.sayHello();
         System.out.println(message);
      } catch (Exception e) {
         System.out.println("HelloClient exception: " +e);
      }
   }
}
```

Generar stubs y skeletons

Ahora que tenemos la interfaz remota y las implementaciones, podemos generar stubs y skeletons (o sólo stubs en Java 1.2) con la herramienta rmic después de haber compilado las clases. No olvide configurar el directorio desde el que está trabajando en su ruta de clase; después puede utilizar la siguiente línea en una ventana de comando:

```
rmic -v1.2 HelloServer
```

Observe que el indicador `-v1.2` suprime la generación de skeletons.

Registrar el objeto

Ahora que ya tenemos la interfaz y la implementación, necesitamos poner este objeto a disposición de los clientes asociándolo a un registro. Esto permitirá a los clientes buscar el objeto en el host por un nombre String. Los stubs y los skeletons (si es que hay alguno) son necesarios para el registro. Después de todo, es el stub objeto el que va a ser pasado desde el registro a los clientes.

> **Con frecuencia se cree erróneamente que el objeto debe estar asociado a un registro para ser utilizado. En realidad, esto no es cierto. El objeto está disponible para su uso en el momento en que es exportado con éxito. Puede funcionar como cliente, invocar un método en otro objeto y pasarse a otro objeto.**

El siguiente código muestra cómo registramos nuestro objeto:

```
import java.rmi.*;

public class RegisterIt {

   public static void main(String args[]) {
      try {
         // Instanciar el objeto
          HelloServer obj = new HelloServer();
          System.out.println("Object instantiated: "  + obj);
          Naming.rebind ("/HelloServer", obj);
          System.out.println("HelloServer bound in registry");
      }  catch (Exception e) {
          System.out.println(e);
      }
   }
}
```

Existen dos métodos en la clase `java.rmi.Naming` que pueden asociar un objeto en el registro:

- El método `bind()` asocia un objeto a un nombre cadena (string) y lanzar una excepción `java.rmi.AlreadyBoundException` si la asociación ya existe
- El método `rebind()`, como ha sido utilizado anteriormente, sustituye cualquier asociación existente por una nueva

El registro debe estar siendo ejecutado para que el objeto se asocie. Puede ser iniciado por la herramienta programáticamente, como ya hemos explicado. Iniciamos el ejecutable con la siguiente línea de comando:

```
rmiregistry -J-Djava.security.policy=registerit.policy
```

La política de seguridad es necesaria debido al modelo de seguridad de Java 2. El registro necesita permisos para abrir Sockets (un Socket en un extremo de una conexión TCP) que están restringidos a extensiones estándar en el archivo de política por defecto.

Nuestro archivo de política para este ejemplo, `registerit.policy`, concede todos los permisos:

```
grant {
   // Permite todo por ahora
   permission java.security.AllPermission;
};
```

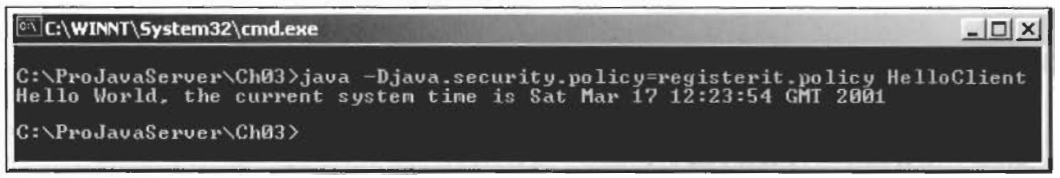
Naturalmente, no utilizaríamos esta política en un entorno de producción. Por defecto, hay un único archivo de política del sistema y un único archivo de política de usuario. El archivo de política de sistema está localizado por defecto en `%JAVA_HOME%\lib\Security\java.policy` (utilice barra oblicua con Solaris), mientras que el archivo de política de usuario se encuentra por defecto en `%USER_HOME%\java.policy`. Consulte la documentación de herramienta incluida en su JDK si desea más detalles sobre las políticas. Es conveniente recordar que al iniciar el registro todas las clases y stubs deben estar disponibles en la ruta de clase, o ésta no debería estar configurado en absoluto, para ajustarse a la carga dinámica.

Ejecutar el cliente y el servidor

Para ejecutar el cliente debemos primero compilar y ejecutar el archivo `RegisterIt`. Después, necesitamos abrir todavía otra ventana de comandos, de modo que tengamos una ventana ejecutando el registro RMI, otra ejecutando el programa `RegisterIt` y la tercera para ejecutar la clase `HelloClient` especificando nuestro archivo de política de seguridad:

```
java -Djava.security.policy=registerit.policy HelloClient
```

El resultado de este sencillo ejemplo debería ser el siguiente:



RMISecurityManager

El gestor de seguridad `java.rmi.RMISecurityManager` amplía la clase `java.lang.SecurityManager` y proporciona el contexto de seguridad en el que ejecutar las aplicaciones RMI. Si no se ha configurado ningún mecanismo de seguridad, los stubs y las clases sólo podrán ser cargados desde la ruta de clase local y no desde el host o desde la clase de código (véase la sección posterior sobre clases cargadas dinámicamente). Este gestor protege a las aplicaciones de la descarga de código no seguro vía invocaciones de método remoto.

En JDK 1.3, no hay realmente necesidad de configurar `RMISecurityManager` debido al control de acceso basado en la política. Además, recuerde que el gestor de seguridad en JDK 1.3 invoca `AccessController.checkPermission (Permission)` por defecto y se refiere a un archivo de política para certificación de permiso.

No hay en realidad ninguna razón para establecer el gestor de seguridad `RMISecurityManager` si un programa RMI desempeña un papel puramente de servidor en todos sus vínculos de comunicación. `RMISecurityManager` (y los gestores de seguridad definidos por el usuario y obtenidos ampliando `RMISecurityManager`) tiene como objetivo controlar las clases cargadas dinámicamente por una aplicación cliente para el control de seguridad. Si el cliente tiene acceso a las definiciones de objeto para el host, tampoco hay ninguna razón para utilizar `RMISecurityManager` en el lado cliente.

Pasar parámetros en RMI

La semántica habitual para métodos en una única JVM está gobernada por dos reglas:

> **Si el tipo pasado es una primitiva, el parámetro o resultado es pasado por valor. Sin embargo, si el tipo pasado es un objeto, el objeto es entonces pasado por referencia.**

Cuando un tipo de datos primitivos es pasado como parámetro a un método, la JVM simplemente copia el valor y pasa la copia al método (o la devuelve de éste). Por otro lado, un objeto reside en el bloque de memoria y una (o más) referencias acceden al él. Al ser pasado a un método, se realiza una copia de la variable referencia (aumentando en uno la cuenta de referencia al objeto) y se sitúa en la pila , y la copia va siendo pasada.

Dentro del método, el código utiliza la copia de la referencia para acceder al objeto. Al invocar cualquier método en la referencia, el estado actual del objeto original cambia. Si se altera la referencia en sí misma, no se produce ningún cambio en el objeto original. Sin embargo, alterar el objeto al que apunta la referencia sí altera el objeto inicialmente creado.

Por ello, ¿qué semántica se utiliza cuando la invocación de método remoto conlleva pasar parámetros o aceptar un valor de retorno? La respuesta a esta pregunta depende de si los parámetros son tipos de datos primitivos, objetos u objetos remotos. Examinemos estos tipos de parámetros con detenimiento.

Parámetros de primitivas

Cuando un tipo de datos primitivos es pasado como parámetro a un método remoto, o bien devuelto desde éste, el sistema RMI lo pasa por valor. Una copia del tipo de datos primitivos es enviada al método remoto y el método devuelve una copia de la primitiva desde su JVM. Estos valores son pasados entre diferentes JVM en un formato estándar independiente de la máquina permitiendo a las JVM ejecutadas en diferentes plataformas que se comuniquen entre ellas de forma fiable.

Parámetros de objeto

Una referencia a un objeto no tiene sentido a través de múltiples JVM puesto que los puntos de referencia a un valor en el bloque y diferentes JVM no comparten la memoria cúmulo. RMI envía el objeto, no su referencia, entre JVM. Es en realidad el objeto el que es pasado por valor, no la referencia al objeto. De un modo similar, una copia del objeto completo es devuelta al programa llamador.

Un objeto Java puede ser sencillo o puede referir a otro objeto Java en una estructura compleja tipo gráfico. Puesto que RMI debe enviar al objeto referenciado y a todos los objetos a los que referencia, utiliza la serialización de objeto RMI, RMI Object Serialization, para transformar el objeto en un formato lineal que puede ser entonces enviado por la red. La serialización de objetos esencialmente extiende un

objeto y cualquier objeto que éste referencie. Los objetos serializados pueden ser deserializados en la memoria de la JVM remota y preparados para su uso por un programa Java.

Pasar grandes gráficos de objetos puede suponer mucho tiempo de CPU y mucho ancho de banda de red. Intente que los argumentos de objeto para métodos remotos y los resultados de éstos sean sencillos para mayor optimización. Los objetos pasados deben implementar la interfaz `java.io.Serializable` o `java.io.Externalizable`.

Parámetros remotos

Pasar objetos remotos como argumentos de método o tipos de retorno no es lo mismo que pasar otros objetos. Un programa cliente puede obtener una referencia a un objeto remoto a través del programa RMI Registry o ésta puede ser devuelta al cliente desde una llamada método (véase el ejemplo `HelloWorld` en la siguiente sección). Pasar objetos remotos es muy importante, especialmente para retrollamadas.

Fíjese en el siguiente código:

```
HelloInterface obj = (HelloInterface) Naming.lookup("/HelloServer");
MsgInterface msg = obj.getMsg();
```

¿Qué ocurre cuando `MsgInterface` es, en sí misma, un objeto remoto que ha sido exportado al servidor?

RMI no devuelve una copia del objeto remoto. Sustituye el objeto remoto por el stub, lo serializa y lo envía al cliente. Como apunte, podemos decir que si el cliente envía este objeto remoto de vuelta al servidor como otro argumento, el objeto es tratado todavía como un objeto remoto en el servidor y no como local al servidor (aunque lo es). Aunque éste pueda parecer un encabezamiento de rendimiento, es crucial para preservar la integridad de la semántica.

Considere otro caso: ¿qué ocurre cuando el método remoto devuelve una referencia a `this`?

```
public class ThisServer extends UnicastRemoteObject
                        implements HelloInterface {

   public ThisServer()throws RemoteException{
      super();
   }
   public HelloInterface someMethod() throws RemoteException {
      return this;
   }
}
```

En el código servidor, this hace referencia a la implementación del servidor que reside en la JVM del servidor. Sin embargo, los clientes no tienen contacto directo con la otra JVM. Tratan con el proxy del objeto del servidor, el stub. Detrás de esto, RMI comprueba los parámetros de entrada y de salida desde un método remoto para ver si implementan la interfaz Remote. Si lo hacen, son claramente reemplazados por el stub correspondiente. Esto provoca en el cliente la ilusión de que están trabajando con los objetos locales porque incluso elementos como this pueden ser intercambiados entre diferentes JVM.

> **Recuerde que los archivos de clase en sí mismos nunca son serializados, sólo los nombres de las clases. Todas las clases deberían poder ser cargadas durante la deserialización utilizando los mecanismos habituales de carga de clases.**

El recolector de residuos distribuidos

Uno de los objetivos de diseño para las especificaciones de RMI era mantener la perspectiva del cliente de objetos remotos igual que la de otros objetos dentro de su propia JVM. Esto implica que los objetos remotos también deben estar sujetos a la recolección de residuos.

El sistema RMI proporciona un algoritmo de recolección de residuos distribuidos de contado de referencias basado en Modula-3's Network Objects. Internamente, el servidor mantiene un registro de qué clientes han solicitado acceso al objeto remoto. Cuando se realiza una referencia, el servidor marca el objeto como *sucio* y, cuando todos los clientes han suprimido la referencia, el objeto es marcado como *limpio*. Un objeto limpio es marcado para recolección de residuos y reclamado cuando se ejecuta el recolector de residuos.

Además del mecanismo de cuenta de referencias en el servidor, cuando un cliente obtiene una referencia, tiene en realidad un alquiler del objeto por un tiempo determinado. Si el cliente no renueva la conexión realizando llamadas *sucias* adicionales al objeto remoto antes de que expire el periodo de alquiler, el recolector de residuos distribuidos asume entonces que el objeto remoto ya no está referenciado por ese cliente (la referencia es considerada como muerta) y el objeto remoto puede ser recolectado como residuo.

Un objeto remoto puede implementar la interfaz `java.rmi.server.Unreferenced`. Éste tiene un método, `unreferenced()`, que es invocado por el periodo de ejecución RMI cuando ya no hay clientes que alberguen una referencia viva. Esto permite al recolector de residuos distribuidos (DGC) comprobar si alguna referencia remota sigue en uso.

> **El tiempo de alquiler es controlado por la propiedad de sistema `java.rmi.dgc.leaseValue`. (Su valor está en milisegundos y el valor por defecto es de 10 minutos.)**

Obviamente, toda esta suciedad, limpieza y estos alquileres nunca son visibles para usuarios o clientes. El mecanismo DGC es completamente transparente, se encuentra oculto en la capa de stubs-skeletons y está abstraído en el paquete `java.rmi.dgc`.

Es importante recordar que un objeto remoto puede ser recolectado como residuo, quedando indisponible para los clientes (lo que tiene como resultado habitualmente la excepción `java.rmi.ConnectException`). Debido a esta semántica de recolección de residuos, un cliente debe estar preparado para tratar con objetos remotos que hayan "desaparecido". En el servidor, si no quiere que su objeto "desaparezca", debe siempre retener una referencia explícita de modo que no sea tomada por el recolector de residuos.

No olvide que el registro (en sí mismo, un objeto remoto) actúa como un cliente para el objeto servidor y, por lo tanto, mantiene un alquiler con el objeto. Por ello, incluso si los clientes "reales" se desconectan de un servidor, el método `unreferenced()` puede no ser invocado en el objeto servidor mientras que el registro mantiene un alquiler.

Considere la siguiente versión modificada del mismo ejemplo `HelloWorld` que demuestra cómo funciona la recolección de residuos y cómo puede ser utilizado el método `unreferenced()`. Ya hemos estudiado cómo puede un cliente obtener una referencia a un objeto remoto como resultado de una invocación de método con anterioridad en la sección sobre pasado de parámetros. Este ejemplo utiliza este concepto.

Modificamos la interfaz para devolver un objeto remoto en lugar de una cadena (string):

```
import java.rmi.*;
```

```
public interface HelloInterface extends java.rmi.Remote {
   public MsgInterface getMsg() throws RemoteException, Exception;
}
```

La implementación del objeto remoto de esta interfaz es también bastante sencilla:

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

public class HelloServer extends UnicastRemoteObject
                         implements HelloInterface {

   public HelloServer() throws RemoteException {
      super();
   }

   public MsgInterface getMsg() throws RemoteException, Exception {
      return (MsgInterface)new MsgServer();
   }
}
```

`MsgInterfaz` no tiene métodos y sólo se utiliza para marcar el objeto que es pasado como remoto:

```
import java.io.Serializable;
import java.rmi.server.*;

public interface MsgInterface extends java.rmi.Remote {}
```

La sencilla implementación de esta interfaz está diseñada para interceptar los eventos del objeto imprimiendo información cuando el objeto es creado, cuando ya no es referenciado, cuando se finaliza y después se elimina. Este objeto también implementa la interfaz Unreferenced y el método `unreferenced()`. El método `unreferenced()` será invocado cuando no haya referencias de cliente al objeto; se invoca `finalize()` justo antes de que el objeto sea recolectado como residuo:

```
import java.io.Serializable;
import java.rmi.server.*;
import java.rmi.*;

public class MsgServer extends UnicastRemoteObject
                       implements MsgInterface, Serializable, Unreferenced {

   // Establecer un contador para el número de instancias creadas
   // de esta clase
   private static int counter;

   // Contener un id para la instancia del objeto
   private int id;

   public MsgServer() throws RemoteException {
      super();
      System.out.println("Created Msg:" + counter);
      counter++;
      setId(counter);
   }

   public void finalize() throws Throwable {
      super.finalize();
      System.out.println("Finalizer called for Msg: " + id);
   }
```

```
    public void unreferenced(){
       System.out.println("The unreferenced()method called for Msg: " + id);

      // si es necesario, invocamos aquí unexportObject ya que no hay nadie
      // que esté utilizando unexportObject(este, claro);
    }

    private void setId(int id){
       this.id=id;
    }
}
```

El programa de registro sigue siendo el mismo de los ejemplos anteriores:

```
import java.rmi.*;

public class RegisterIt {

   public static void main(String args[]) {
     try {
        HelloServer obj = new HelloServer();
        Naming.rebind("/HelloServer", obj);
        System.out.println("HelloServer bound in registry");
     }  catch (Exception e) {
        System.out.println(e);
     }
   }
}
```

Modificamos el cliente ligeramente para crear múltiples instancias del objeto remoto `MsgServer()` en el servidor (instanciamos explícitamente un objeto en el método `getMsg()`). Necesitamos realizar esta operación para nuestra demostración porque la JVM ejecuta el recolector de residuos distribuidos (igual que el habitual recolector de residuos), sólo cuando "cree" que es necesario reclamar memoria:

```
import java.rmi.*;

public class HelloClient {

   public static void main(String args[]) {
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new RMISecurityManager());
      }

      try {
         HelloInterface obj = (HelloInterface) Naming.lookup("/HelloServer");
         for(int i = 0; i < 100; i++) {
            MsgInterface msg = obj.getMsg();
         }
      } catch (Exception e) {
         System.out.println("HelloClient exception: " + e);
      }
   }
}
```

Inicie el servidor después de compilar las clases y de generar los stubs (y los skeletons si es necesario) para `HelloServer` y para `MsgServer` con los siguientes elementos (cambiando la ruta `codebase` cuando proceda; consulte la próxima sección para más detalles sobre `codebase`):

```
javac *.java
rmic -v1.2 HelloServer
rmic -v1.2 MsgServer
```

```
rmiregistry -J-Djava.security.policy=registerit.policy
java -Djava.rmi.dgc.leaseValue=1000 -
Djava.security.policy=registerit.policy RegisterIt
```

Una vez más, puede que desee experimentar con la propiedad del valor de alquiler así como con las opciones –ms y –mx para configurar el bloque de memoria para este ejemplo. Aquí tiene un fragmento de la salida al ejecutar HelloClient con los valores por defecto:

```
java -Djava.security.policy=registerit.policy HelloClient
```



LeaseCheckInterval en `sun.rmi.transport.DGCImpl` es leído desde la propiedad `sun.rmi.dgc.checkInterval` sin tener en cuenta el valor actual de la variable `leaseValue`. Esto provoca un retraso de cinco minutos al eliminar objetos no utilizados, incluso si `leaseValue` se fija en un valor menor. Para evitar este problema, fije la propiedad `sun.rmi.dgc.checkInterval` en la mitad de la configuración de `java.rmi.dgc.leaseValue`.

Clases de carga dinámica

Hemos hablado anteriormente sobre cómo las referencias, los stubs, los parámetros y las factorías de sockets son enviadas al cliente y sobre cómo las definiciones de clase no lo son. El cargado de clase dinámico hace referencia a este último, haciendo que las definiciones estén disponibles.

El cargado dinámico de clase lleva en Java bastante tiempo y esta capacidad para cargar e instanciar dinámicamente clases es un concepto muy eficaz. Los applet, por ejemplo, son descargados al navegador del cliente desde un servidor Web y son ejecutados en la JVM del cliente. Esto proporciona al periodo de ejecución del cliente la capacidad para acceder a aplicaciones que nunca han sido instaladas en su sistema.

Se da por sentado que el lector está familiarizado con el funcionamiento de los applet y cómo utilizan la propiedad `codebase`. Resumiendo, `codebase` es el lugar desde el que el cargador de clase carga clases en la JVM. Esto significa que las clases pueden ser desplegadas en un lugar central, como un servidor Web, para un sistema distribuido y todas las aplicaciones del sistema pueden descargar los archivos de clase para operar.

Hay dos importantes propiedades de sistema en RMI:

- `Java.rmi.server.codebase`
  Ésta especifica el URL (una localización `file://`, `ftp://` o `http://`) desde donde se puede acceder a las clases. Si un objeto es pasado como argumento de método o tipo de retorno, la JVM cliente necesita cargar los archivos de clase para ese objeto. Cuando RMI serializa el objeto, inserta el URL especificado por esta propiedad junto al objeto.

- `Java.rmi.server.useCodebaseOnly`
  Esta propiedad es utilizada para notificar al cliente que debe cargar clases únicamente desde la posición de base de código, codebase.

Repasemos el proceso de exportación y registro de un objeto remoto y observémoslo a través de una lupa:

- Al instanciar el objeto remoto y registrarlo con el registro (nuestro programa `RegisterIt` de los ejemplos anteriores), la base de código es especificada por la propiedad `java.rmi.server.codebase`.

- Cuando se realiza la llamada `bind()`, el registro utiliza esta base de código para ubicar el stub para el objeto. Recuerde que el registro mismo es un cliente para el objeto. Una vez se ha conseguido esto con éxito, el registro asocia el objeto a un nombre y la base de código es guardada junto con la referencia al objeto remoto en el registro.

- Cuando un cliente solicita una referencia al objeto remoto, el registro devuelve el stub al cliente. El cliente busca la definición de clase del stub en su ruta de clase local (siempre se busca antes la ruta de clae que la base de código) y entonces carga la clase local. Si la definición de clase de stub no se encuentra en la ruta de clase, el cliente intentará recuperar la definición de clase de la base de código del objeto remoto que ha sido almacenado en el registro.

- La definición de clase para el stub y cualquier otra clase que necesite, como factorías de Sockets, son descargadas al cliente JVM desde la base de código `http` o `ftp`.

- Cuando todas las definiciones de clase están disponibles, el método proxy del stub llama el objeto en el servidor.

Utilizando las dos propiedades y el mecanismo de descarga, pueden establecerse cinco configuraciones potenciales para distribuir clases:

- **Cerrada**
  No hay carga dinámica y todas las clases están ubicadas en las JVM respectivas y son cargadas desde la ruta de clase local (`java.rmi.server.codebase` no configurado).

- **Dinámica lado cliente**
  En el cliente, algunas clases son cargadas desde la ruta de clase local y otras desde la base de código especificado por el servidor.

- **Dinámica lado servidor**
  Es similar a la configuración dinámica lado cliente. Algunas clases en el servidor son cargadas localmente y otras desde la base de código especificado por el cliente (por ejemplo, en el caso de retrollamadas desde el servidor al cliente).

- **Cliente de autoarranque**
  Todo el código de cliente es cargado desde el codebase mediante un pequeño programa en el cliente (cargador de autoarranque).

- **Servidor de autoarranque**
  Igual que el anterior, sólo que en el lado servidor. El servidor utiliza un pequeño programa (cargador de autoarranque).

Hay dos clases, `java.rmi.RMISecurityManager` y `java.rmi.RMIClassLoader`, que comprueban el contexto de seguridad. El sistema RMI sólo descargará clases de ubicaciones remotas si ha sido enviado un gestor de seguridad. El `RMIClassLoader` tiene un método importante:

```
public static Class loadClass(String codebase, String name)
```

Este método carga la clase desde la base de código especificada. Otras formas de sobrecarga de este método toman un URL para una base de códiog o una lista de cademas de múltiples URL.

Tomemos el ejemplo HelloWorld y configurémoslo en la configuración cliente de arranque y servidor de arranque. Esta configuración es habitualmente la más popular debido a su flexibilidad. Descargaremos:

- Todas las clases del cliente, incluida la clase cliente, al cliente desde el servidor Web.
- Todas las clases del servidor, incluida la clase servidor, desde un servidor Web central.

Sólo necesitaremos una sencilla clase de arranque para el cliente y una para el servidor.

Primero, vamos a retocar un poco la clase `HelloWorld` de modo que se conecte al servidor al ser instanciada:

```
import java.rmi.*;

public class HelloClient {

   public HelloClient() {
     try {
        HelloInterface obj = (HelloInterface) Naming.lookup(
           "rmi://localhost/HelloServer");
        String message   = obj.sayHello();

      System.out.println(message);
    } catch (Exception e) {
      System.out.println("HelloClient excepciton: " + e);
     }
   }
}
```

Ya está. Mantenemos la implementación remota (`HelloServer.java`) y la interfaz (`HelloInterface.java`) igual que en nuestro ejemplo inicial.

Escribamos ahora una clase de arraque, *bootstrapped*, (en lugar del archivo `RegisterIt` que utilizamos anteriormente) que inicie el servidor. Accede a la propiedad `codebase` y carga la clase de servidor desde la base de código utilizando `RMIClassLoader`:

```
import java.rmi.Naming;
import java.rmi.Remote;
import java.rmi.RMISecurityManager;
import java.rmi.server.RMIClassLoader;
import java.util.Properties;

public class DynamicServer {

   public static void main(String args[]) {

      // Crear e instalar un gestor de seguridad
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new RMISecurityManager());
      }

      try {
```

```
            Properties p = System.getProperties();
            String url = p.getProperty("java.rmi.server.codebase");
            Class serverclass = RMIClassLoader.loadClass(url, "HelloServer");
            Naming.rebind("/HelloServer",(Remote)serverclass.newInstance());
            System.out.println("HelloServer bound in registry");
        }   catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Cuando la clase descargada es instanciada con `newInstance()`, el constructor para la clase es invocado (nuestro constructor `HelloServer()`) y el objeto es exportado en el constructor y registrado con el registro. Los cinco pasos que ya hemos descrito en este capítulo (*Desarrollo de aplicaciones con RMI*), suceden conjuntamente entre el servidor y el registro.

Existe un programa de arranque similar para el cliente que accede a la propiedad codebase y carga la clase cliente. Cuando la clase cliente descargada es instanciada, conecta al servidor y busca el objeto. Una vez más, los pasos de 1 a 5 suceden entre el cliente y el registro:

```
import java.rmi.RMISecurityManager;
import java.rmi.server.RMIClassLoader;
import java.util.Properties;
public class DynamicClient {

    public DynamicClient() throws Exception {
        Properties p = System.getProperties();
        String url = p.getProperty("java.rmi.server.codebase");
        Class clientClass = RMIClassLoader.loadClass(url, "HelloClient");

        // Start the client
        clientClass.newInstance();
    }

    public static void main (String args[]) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            DynamicClient dc = new DynamicClient();
        }   catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Compile todas las clases y genere los stubs. Ahora necesitamos situar los archivos de clase `HelloWorld` en un servidor Web de modo que `DynamicServer` pueda descargarlos en el periodo de ejecución. Para este ejemplo, utilizaremos un pequeño servidor HTTP de prueba de Sun llamado `ClassFileServer` (también incluido en el código fuente).

Sitúe los cuatro archivos de clase para el ejemplo `HelloWorld` en un directorio (`\public_html`) en el servidor HTTP.

Primero, inicie el registro RMI utilizando:

```
rmiregistry -J-Djava.security.policy=registerit.policy
```

después inicie el servidor HTTP. Para ClassFileServer sería algo así (asumiendo que ClassFileServer se encuentra en el classpath):

```
java ClassFileServer 8080 %BOOK_HOME%\Ch03\Dynamic\webserver\public_html
```

El primer parámetro de esta línea de comando especifica el puerto en el que se ejecuta el servidor y el segundo parámetro es la fuente para que los archivos sean servidos. Probablemente, resultará un poco más rápido configurar un archivo por lotes para ejecutar estos comandos.

Entonces, ejecutamos `DynamicServer` especificando `codebase` como servidor Web:

```
java -Djava.security.policy=registerit.policy
     -Djava.rmi.server.codebase=http://localhost:8080/ DynamicServer
```

Finalmente, ejecute `DynamicClient`, proporcionando de nuevo el `codebase`:

```
java -Djava.security.policy=registerit.policy
     -Djava.rmi.server.codebase=http://localhost:8080/ DynamicClient
```

En resumen:

- Cuando iniciamos el DynamicServer, éste accede a la base de código descarga la clase de implementación y exporta el objeto
- El registro utiliza el código de base para descargar el stub y asociar el objeto, registrando la base de código de utilizada
- Cuando inicia el `DynamicClient`, contacta de nuevo con la base de códigoy descarga la clase de cliente y el stub, e invoca un método en el objeto servidor

Resumiendo los pasos para ejecutar el ejemplo:

Nota: la fuente descargable contiene archivo por lotes de prueba para ejecutar el código.

- Inicie `rmiregistry`:

```
rmiregistry -J-Djava.security.policy=registerit.policy
```

- Inicie el servidor de archivo de clase. Asegúrese de que las clases compiladas del primer ejemplo `HelloWorld` así como los stubs, están disponibles en la carpeta `public_html`:

```
java ClassFileServer 8080 %BOOK_HOME%\Ch03\Dynamic\webserver\public_html
```

- Inicie el `DynamicServer`:

```
java -Djava.security.policy=registerit.policy
     -Djava.rmi.server.codebase=http://localhost:8080/ DynamicServer
```

- Inicie el `DynamicClient`:

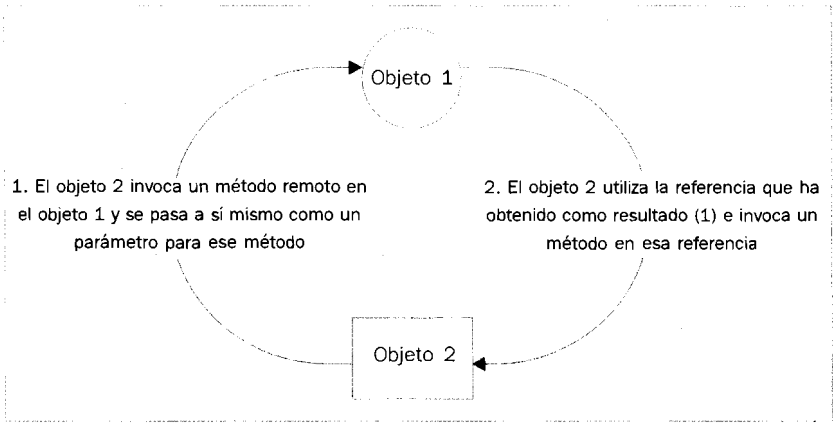
```
java -Djava.security.policy=registerit.policy
     -Djava.rmi.server.codebase=http://localhost:8080/ DynamicClient
```

Observe que todos los archivos de cliente HelloWorld, de servidor y stub se ubican en un solo lugar. La única distribución para el servidor es una clase y el único archivo necesario para la distribución de cliente es una clase:



Retrollamadas remotas

Hemos visto anteriormente cómo un cliente puede obtener una referencia a un objeto remoto como resultado de una invocación de método. De hecho, en realidad vimos en el ejemplo de DGC cómo un objeto remoto puede ser pasado dinámicamente. Recuerde que habíamos destacado inicialmente que cliente y servidor son términos utilizados para describir un rol, no ubicaciones físicas o arquitecturas. Un cliente también puede ser un objeto remoto. En muchas situaciones, un servidor puede necesitar realizar una llamada remota a un cliente, por ejemplo, datos de progreso o notificaciones administrativas. Un buen ejemplo sería una aplicación chat donde todos los clientes son objetos remotos.



No hay nada de particular en la comunicación entre pares o en las retrollamadas entre objetos remotos. Todo lo que ocurre es que una referencia remota es pasada y que los métodos son invocados en esa referencia.

Tomemos en mismo ejemplo `HelloWorld` que examinamos en un principio y modifiquémoslo para demostrar las retrollamadas.

La interfaz remota, `HelloInterface`, tiene un método que toma un `ClientInterface` como argumento:

```
public interface HelloInterface extends java.rmi.Remote {
   String sayHello(ClientInterface app) throws java.rmi.RemoteException;
}
```

Para el cliente, la ClientInterface es también muy sencilla, con un solo método:

```
public interface ClientInterface extends java.rmi.Remote {
   void popup(String msg) throws java.rmi.RemoteException;
}
```

`HelloServer` implementa `HelloInterface` e invoca el método `popup()` en `ClientInterface` como se muestra a continuación:

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;
```

```
public class HelloServer extends UnicastRemoteObject
                         implements HelloInterface {

   public HelloServer() throws RemoteException {
      super();
   }
   public String sayHello(ClientInterface ca) throws RemoteException {
      ca.popup("This is a message from the server!");
      return "Hello World, the current system time is " + new Date();
   }
}
```

El cliente que invoca métodos en el servidor necesita ser un objeto remoto en sí mismo. Escribamos un sencillo applet para demostrar esto. El applet se exporta a sí mismo y empieza a escuchar llamadas de entrada invocando explícitamente el método `UnicastRemoteObject.exportObject (this)` (una forma sobrecargada de este método le permite especificar también el puerto):

```
import java.applet.*;
import java.awt.*;
import java.io.Serializable;
import java.rmi.*;
import java.rmi.server.*;

public class CallbackApplet extends Applet
                            implements ClientInterface, Serializable {

   String message = "-n/a-";
   Frame f =new Frame();
   Label l1= new Label("                    ");

   public void init() {
      f.add(l1);
      try {
        // Exportar el objeto
        UnicastRemoteObject.exportObject (this);
        String host = "rmi://" + getCodeBase().getHost()+ "/HelloServer";
        HelloInterface obj = (HelloInterface)Naming.lookup(host);
        message = obj.sayHello((ClientInterface)this);
      }  catch (Exception e) {
        System.out.println("HelloApplet exception: " +e);
      }
   }

   // Mostrar el mensaje en la applet
   public void paint(Graphics g) {
      g.drawString(message, 25, 50);
   }

   // Implementar la interfaz
   public void popup(String txt) throws RemoteException{
      l1.setText(txt);
      f.setSize(100,100);
      f.show();
   }
}
```

Cuando el applet es cargado, localiza el objeto servidor e invoca el método remoto. Se pasa a sí mismo como argumento para ese método. Como resultado, el stub es transportado al servidor y el servidor puede ahora invertir los roles. Actúa como un cliente para este applet y puede invocar el método `popup()`.

Necesitamos una simple página HTML para este applet (`Applet.html`):

```
<html>
<applet codebase="http://localhost:8080/Callbacks" code="CallbackApplet.class"
        width=300 height=266>
</applet>
</html>
```

Es tan sencillo como esto. Generamos los stubs para el cliente y el servidor ejecutando `rmic` en `CallbackApplet` y `HelloServer`. Inicie el registro y registre `HelloServer` con él utilizando la misma clase `RegisterIt` que hemos utilizado anteriormente.

El applet debe ser cargado desde un servidor Web en el host (recuerde que los applet sin firmar pueden conectar de nuevo con el servidor) y puede utilizar su propio servidor Web o el de prueba `ClassFileServer` utilizado en el ejemplo anterior. Todas las clases que el cliente necesita deben estar accesibles en el servidor Web. Las definiciones de archivo de clase stub de `CallbackApplet` deberían estar disponibles en la ruta de clase de objetos del servidor o deberían ser cargadas dinámicamente como hemos visto anteriormente.

También puede probar las aplicaciones utilizando AppletViewer. Configure las preferencias en `Applet Viewer` como `"unrestricted"`.

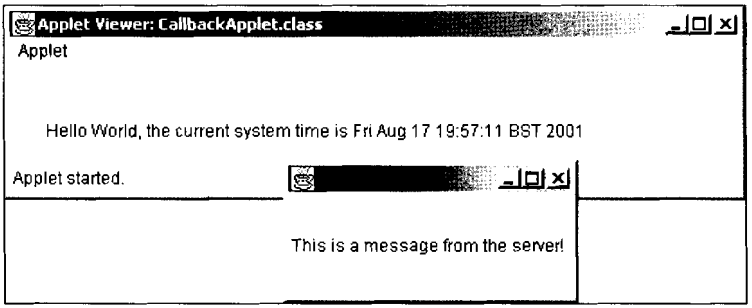
Escriba un archivo HTML con el siguiente código (`AppletViewer.html`):

```
<html>
   <applet codebase="." code="CallbackApplet.class" width=300 height=266>
   </applet>
</html>
```

Ejecute el ejemplo después de iniciar el registro y el servidor desde el mismo directorio como el archivo HTML y los archivos de clase.

```
appletviewer -J-Djava.security.policy=registerit.policy AppletViewer.html
```

Finalmente, al ejecutar este ejemplo, obtenemos algo parecido a esto:



Activación de objeto

Los objetos remotos que hemos analizado hasta el momento son instancias de la clase `java.rmi.UnicastRemoteObject` y tienen una característica principal. Son accesibles en todo momento, incluso cuando no hay clientes ejecutando. Consideremos un marco hipotético en el que el número de objetos remotos, o de recursos utilizados por ellos en un servidor sea alto. Esta situación fue identificada como un importante cuello de botella en Java 2, por lo que se introdujo el concepto de activación.

> **La activación de objetos permite ejecutar objetos remotos dependiendo de la necesidad. Es decir, cuando se accede a un objeto remoto "activable" (vía invocación de método), si ese objeto remoto no está siendo ejecutado en ese momento, el sistema inicia la ejecución del objeto en una JVM apropiada. RMI utiliza la activación lenta: es aquella en la que la activación de un objeto es aplazada hasta el primer uso de un cliente, la primera invocación de método.**

Entonces, ¿cuál es la diferencia entre un objeto activable y el habitual objeto remoto desde la perspectiva de un cliente? Ninguna. Para el cliente, todo el mecanismo de activación es transparente y el cliente nunca es consciente de lo que está ocurriendo detrás de la escena.

Las referencias al objeto remoto pueden ser consideradas referencias lentas o imperfectas. Las referencias a objetos activables contienen información persistente de descripción que permite la activación del subsistema para saber que el objeto debería estar iniciado si no está ya siendo ejecutado. Después de que una referencia activable sea utilizada por primera vez, la capa de referencia remota cambia a referencia remota regular de modo que no tenga que pasar por el sistema de activación posteriormente.

Puesto que el sistema de activación puede cambiar la referencia lenta a referencia remota activa, las referencias a objetos activables están siempre disponibles. Sin embargo, las referencias a un objeto remoto no sobreviven después de un fallo del sistema o de reiniciar.

Para comprender la verdadera semántica de utilizar el modelo de activación, debemos familiarizarnos con algunos términos útiles:

- **Activador**
  El activador es un componente principal en el servidor. Facilita la activación del objeto remoto registrando toda la información necesaria para activar un objeto y es responsable de iniciar instancias de las JVM en el servidor si es necesario.

- **Grupo de activación**
  Un grupo de activación crea instancias de objetos en su grupo e informa a su monitor sobre los diferentes estados activos y pasivos. La analogía más cercana a un grupo de activación es un grupo thread (hebra). Un grupo de activación es en esencia una instancia completa, independiente de la JVM que existe únicamente para albergar grupos de objetos activados. Esta nueva JVM es iniciada por el activador cuando así se requiere. Pueden existir múltiples grupos de activación.

- **Monitor de activación**
  Cada grupo de activación tiene un monitor de activación que registra el estado de un objeto en el grupo y el estado del grupo en su conjunto. El monitor de activación es creado cuando se activa el grupo.

- **Sistema de activación**
  El sistema de activación proporciona un medio para registrar grupos y objetos activables para ser activados dentro de esos grupos. Trabaja en estrecha relación con el activador, que activa objetos

registrados mediante el sistema de activación y con el monitor de activación, y obtiene información sobre objetos activos e inactivos, y grupos inactivos.

El modelo de activación y sus implementaciones asociadas aparece resumido en la siguiente tabla:

Entidad	Implementación	Implementado como
Activador	`java.rmi.activation.Activator`	Interfaz (observe que el activador es en sí mismo un objeto remoto).
Grupo de activación	`java.rmi.activation.ActivationGroup`	Clase abstracta.
Monitor de activación	`java.rmi.activation.ActivationMonitor`	Interfaz.
Sistema de activación	`java.rmi.activation.ActivationSystem`	Interfaz.

El mecanismo de activación utiliza identificadores y descriptores. Aunque pueda parecer excesivo, merece la pena recordar estos puntos:

- Cada objeto activable tiene un ID y un descriptor.
- Cada objeto activable pertenece a un grupo de activación. El grupo mismo tiene un ID y un descriptor.

El grupo de activación

Un grupo de activación, como ya hemos dicho anteriormente, se utiliza para mantener un grupo de objetos activables. El grupo de activación está asociado a un identificador de grupo (`java.rmi.Activation.ActivationGroupID`) y a un descriptor de grupo (`java.rmi.activation.ActivationGroupDesc`) que identifican y describen el grupo de activación respectivamente.

Un grupo de activación es creado explícitamente como resultado de invocar el método `ActivationGroup.createGroup()`:

```
public static ActivationGroup createGroup(ActivationGroupID id,
                                           ActivationGroupDesc desc,
                                           long incarnation)
```

donde:

- `id` es el identificador del grupo de activación
- `desc` es el descriptor del grupo de activación
- `incarnation` es el número de personificación del grupo de activación (cero en la creación inicial de un grupo)

`ActivationGroupID`, además de identificar el grupo de forma exclusiva dentro del sistema de activación, también contiene una referencia al sistema de activación del grupo. Esto permite al grupo interactuar con el sistema siempre y cada vez que sea necesario. Todos los objetos con el mismo `ActivationGroupID` son activados en la misma JVM.

Un `ActivationGroupDesc` contiene la información necesaria para crear o recrear el grupo en el que activar objetos. Contiene:

- El nombre de clase del grupo. Recuerde que `java.rmi.activation.ActivationGroup` es una clase abstracta y el activador (`rmid`) proporciona internamente su implementación concreta (por ejemplo, la clase `sun.rmi.server.ActivationGroupImpl`).
- La ubicación de la clase del grupo.
- Un objeto escrito y transmitido (marshaled) que puede contener datos de inicialización específicos del grupo.

`ActivationGroupDesc` contiene una clase interna `CommandEnvironment` que especifica las opciones de entorno de arranque para las clases de implementación de `ActivationGroup`. Esto permite un control exacto sobre las opciones de comando utilizadas para iniciar la JVM descendiente (un `CommandEnvironment` nulo hace referencia a los valores por defecto de `rmid`).

`ActivationGroupDesc` puede ser creado utilizando uno de los dos constructores especificados a continuación:

- Construir un descriptor de grupo que utilice la omisión del sistema para la implementación de grupo y la localización de código:

```
ActivationGroupDesc(Properties overrides,
                    ActivationGroupDesc.CommandEnvironment cmd)
```

- Especificar una implementación de grupo y un entorno de ejecución alternativo para ser utilizado por el grupo:

```
ActivationGroupDesc(String className, String location,
                    MarshalledObject data, Properties overrides,
                    ActivationGroupDesc.CommandEnvironment cmd)
```

Hemos dado un rápido repaso a los grupos de activación. Veamos ahora algo de código que resume la creación de grupos de activación:

```
// Crear el descriptor de grupo
Properties env = new Properties();
env.put("java.security.policy","file://%BOOK HOME%/Ch03/Activatable/
registerit.policy");
ActivationGroupDesc mygroupdes = new ActivationGroupDesc(props, null);

// Obtener una referencia al sistema de activación
ActivationSystem mysystem= ActivationGroup.getSystem();

// Registrar el descriptor de grupo con el sistema de activación y obtener el
id del grupo id ActivationGroupID groupid=y system.registerGroup(mygroupdes);

// Ahora que tenemos el id y el descriptor podemos crear el grupo explícitamente
ActivationGroup.createGroup(groupid, mygroupdes, 0);
```

ActivationID

Igual que `ActivationGroupID` es un ID para el grupo, `ActivationID` es un ID para el objeto. Una vez que el objeto es registrado con el sistema de activación, se le asigna un `ActivationID`. Contiene dos elementos esenciales de información:

- Una referencia remota al activador del objeto

- Invocando el método estático `Activatable.register (ActivationDesc desc)`.
- Instanciando el objeto mismo utilizando el primer o el segundo constructor de la clase `Activatable` (que toma `ActivationID` como parámetro). Éste registra y exporta el objeto.
- Exportando el objeto explícitamente mediante el primer o el segundo método `exportObject()` de la clase `Activatable` que toma un `ActivationID`, la implementación de un objeto remoto y un número de puerto como argumentos. Este registra y exporta el objeto.

Detrás de las escenas está ocurriendo lo siguiente:

- Cuando se genera un stub para un objeto activable, contiene información especial sobre el objeto. Esta información incluye el identificador de activación e información sobre el tipo de referencia remota del objeto.
- El activador localiza el descriptor de activación del objeto y el grupo de activación. Si el grupo de activación en el que debería estar este objeto no existe, el activador inicia una instancia de una JVM, crea un grupo de activación y después envía la solicitud de activación a ese grupo.
- El grupo de activación carga la clase para el objeto y lo instancia (utilizando constructores especiales que toman varios argumentos, como veremos pronto).

Cuando el objeto es activado, el grupo de activación devuelve una referencia de objeto al activador (ésta es una referencia serializada o marshaled). El activador graba el identificador de activación y el pareado de referencia y devuelve la referencia viva al stub. El stub envía entonces invocaciones de método mediante esta referencia viva directamente al objeto remoto (la referencia viva es como cualquier otra referencia).

Resumamos todo lo que hemos visto hasta ahora volviendo a ejecutar el ejemplo `HelloWorld` para convertirlo en activable.

Convertir objetos en activables

Como hemos visto, existen algunas entidades cooperantes en el marco de trabajo de la activación que hacen todo sea posible:

- El objeto.
- El programa envoltorio que registra el objeto. Es similar al programa RegisterIt que utilizamos anteriormente. Normalmente realiza unas llamadas de método al sistema de activación para proporcionar detalles sobre cómo debería ser activado el objeto.
- La tercera entidad es el demonio (daemon) de activación que graba información, igual que el registro, sobre cuándo y qué hacer con los objetos (este daemon es `rmid`).

Teniendo en cuenta estas entidades, los pasos para convertir un objeto en activable pueden resumirse del siguiente modo:

- El objeto debe ampliar la clase `java.rmi.Activation.Activatable` en lugar de la clase `UnicastRemoteObject` (existe una alternativa a esto que examinaremos posteriormente).
- El objeto debe incluir un constructor especial que tome dos argumentos, su identificador de activación de tipo `ActivationID`, y sus datos de activación opcionales, un `java.rmi.MarshalledObject`. Este es distinto de los objetos remotos no activables, que incluyen un constructor sin argumento. Este constructor especial es llamado por el sistema RMI cuando activa el objeto.

- Un descriptor de activación (`java.rmi.Activation.ActivationDesc`) debe ser creado y registrado con el activador (`rimd`).

Es importante destacar que la interfaz remota del objeto no necesita ser cambiada o modificada en ningún caso. Es lógico porque sólo estamos cambiando la implementación de la instancia del objeto, no la percepción exterior del objeto.

Paso 1: Crear la interfaz remota

No se diferencia de lo que teníamos anteriormente:

```
import java.rmi.*;

public interface HelloInterface extends Remote {
   public String sayHello() throws RemoteException;
}
```

Paso 2: Crear la implementación del objeto

Esta clase amplía `java.rmi.Activation.Activatable` e implementa la interfaz remota (puesto que la clase `Activatable` amplía `RemoteObject`). Además, debe contener un constructor de dos argumentos que toma `ActivationID` y `MarshaledObject` como argumentos. Este constructor debe llamar a los constructores de superclase adecuados para asegurar la inicialización:

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Date;

public class HelloServer extends Activatable implements HelloInterface {

   public HelloServer(ActivationID id, MarshaledObject data)
          throws RemoteException {
      // Registra el objeto con una activación del sistema
      // Registrarlo y exportarlo en un puerto anónimo
      super(id, 0);
   }

   public String sayHello() throws RemoteException{
      return "Hello World, the current system time is " + new Date();
   }
}
```

Paso 3: Registrar el objeto con el sistema

Esta clase contiene toda la información necesaria para registrar el objeto, sin crear verdaderamente una instancia del objeto:

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;

public class RegisterIt {

   public static void main(String args[]) throws Exception {
     try {
        //Instalar un gestor de seguridad
        System.setSecurityManager(new RMISecurityManager());
```

```
        // Crear el grupo
        Properties env = new Properties();
        env.put("java.security.policy",
                "file://%BOOK HOME%/Ch03/Activatable/
                activation/registerit.policy");
        ActivationGroupDesc mygroupdes = new ActivationGroupDesc(env,null);
        ActivationGroupID mygroupid = ActivationGroup.getSystem().
                                        registerGroup(mygroupdes);
        ActivationGroup.createGroup(mygroupid, mygroupdes, 0);

        // Crear los detalles sobre el objeto
        ActivationDesc objectdesc = new ActivationDesc("HelloServer",
                        "file://%BOOK HOME%/Ch03/Activatable", null);

        // Registrar el descriptor de activación con el activador
        HelloInterface myobject = (HelloInterface)Activatable
                                                .register(objectdesc);

        // Asociar el stub a un nombre del registro rmi
        Naming.rebind("helloObject", myobject);

        // Salir
        System.out.println("Done");

      } catch(Exception e) {
        System.out.println("Exception "+ e);
      }
    }
}
```

Casi hemos terminado. Ahora podemos utilizar el viejo cliente que grabamos para nuestro ejemplo inicial y ejecutarlo con los mismos argumentos de política y arranque:

```
import java.rmi.*;

public class HelloClient {

   public static void main(String args[]) {

      if (args.length < 1) {
         System.out.println ("Usage: java HelloClient <host>");
         System.exit(1);
      } else {

        try {
            HelloInterface server = (HelloInterface)Naming.lookup(
                                        "rmi://" + args[0] + "/helloObject");
            System.out.println(server.sayHello());
        } catch (Exception e) {
            e.printStackTrace();
        }
      }
   }
}
```

Compile los archivos y después genere los stubs y los skeletons para el objeto remoto. Inicie el sistema de Activación utilizando la utilidad `rmid` y luego inicie nuestro programa que registra nuestros objetos activables:

```
rmid -J-Djava.security.policy=registerit.policy
```

Ahora podemos ejecutar el archivo `RegisterIt`:

```
java -Djava.security.policy=registerit.policy
     -Djava.rmi.server.codebase=file:///%BOOK_HOME%/Ch03/Activatable/
      RegisterIt
```

Finalmente, podemos utilizar el cliente, como muestra la siguiente pantalla de salida:



Fíjese en que debería tener tres o cuatro ventanas de líneas de comandos abiertas para este ejemplo:

- `Rmiregistry`
- `Rmid`
- `RegisterIt`
- `HelloClient`

Las dos últimas podrían ejecutarse en una ventana.

Alternativa a ampliar la clase *Activatable*

En ocasiones puede que no sea posible ampliar la clase `Activatable` si el objeto remoto necesita ampliar otra clase. Por ejemplo, si estamos escribiendo un applet que queremos convertir en activable o en objeto remoto, nuestro applet necesitaría ampliar `java.applet.Applet` y en Java no existe la herencia múltiple.

Ya hemos visto anteriormente que era posible convertir un objeto en remoto exportando el objeto directamente utilizando uno de los métodos de exportación de la clase `java.rmi.UnicastRemoteObject`, en vez de ampliar la clase `UnicastRemoteObject`.

Existe una alternativa similar para los objetos activables. Al hablar sobre descriptores de activación hemos mencionado un método `exportObject()`. Es posible registrar y exportar un objeto invocando cualquiera de los métodos de exportación de la clase `java.Activation.Activatable`.

Reescribamos nuestra clase activable `HelloServer` para demostrar esto, manteniendo igual el resto:

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Date;

public class HelloServer implements HelloInterface {

   public HelloServer(ActivationID id, MarshaledObject data)
          throws RemoteException {

      // Registrarlo y exportarlo en un puerto anónimo
      Activatable.exportObject(this, id, 0);
      // OR
      /*
```

```
        Activatable.exportObject(this, id, 0,
                                 new SomeRMIClientSocketFactory csf(),
                                 new SomeRMIServerSocketFactory())
        //OR
        Activatable.exportObject(this, ".", data, false, 0);

        //OR
        Activatable.exportObject(this, ".", data, false, 0,
                                 new SomeRMIClientSocketFactory(),
                                 new SomeRMIServerSocketFactory());
        */
    }

    public String sayHello() throws RemoteException{
        return "Hello World, the current system time is" + new Date();
    }
}
```

En el ejemplo anterior, dos de los constructores presentados toman factorías de socket como argumentos. Trataremos las factorías de socket en breve. Por ahora, sólo recuerde que puede pasar factorías de socket como argumentos a sus objetos activables.

Iniciar múltiples JVMs sin recurrir a rmid

`rmid` o el activador mismo es un objeto remoto y se ejecuta en una JVM. Cada objeto que es activado es activado en su JVM del grupo de activación pero, en ocasiones, puede ser deseable producir una JVM independiente para cada objeto activable (por ejemplo, en una gran aplicación desearía eliminar el único punto de fallo aislando objetos en diferentes JVM). Es muy sencillo si tenemos en cuenta lo que mencionamos al referirnos a `ActivationGroupID`, es decir, "Todos los objetos con el mismo `ActivationGroupID` son activados en la misma JVM".

Para iniciar múltiples JVM, el objeto debe tener un `ActivateGroupID` diferente o, en otras palabras, debe estar en un grupo de activación independiente. El siguiente código muestra cómo el mismo objeto `HelloServer` es registrado de forma distinta de modo que cada objeto tenga un `ActivateGroupID` diferente (y, por consiguiente, un grupo de activación diferente).

Ésta es la clase que registra nuestros objetos activables:

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;

public class RegisterIt {
   public static void main(String[] args) {
      try {
         //Instalar un gestor de seguridad
         System.setSecurityManager(new RMISecurityManager());

         // crear otra para una nueva VM
         Properties env = new Properties();
         env.put("java.security.policy",
                 "file://%BOOK HOME%/Ch03/Activatable/MultiVM/
                  registerit.policy");

         ActivationGroupDesc mygroupdes = new ActivationGroupDesc(env,null);
         ActivationGroupID mygroupid = ActivationGroup.getSystem()
                                              .registerGroup(mygroupdes);
         ActivationGroup.createGroup(mygroupid, mygroupdes, 0);
         ActivationDesc objectdesc = new ActivationDesc("HelloServer",
```

```
                    "file://%BOOK HOME%/Ch03/Activatable/MultiVM/",
                    null);
            HelloInterface myobject = (HelloInterface)Activatable.register(
                                        objectdesc);
            Naming.rebind("helloObject", myobject);

            // Registrar otro id de grupo
            ActivationGroupID mygroupid 2 = ActivationGroup.getSystem()
                                            .registerGroup(mygroupdes);
            ActivationDesc objectdesc 2 = new ActivationDesc("HelloServer",
                    "file://%BOOK HOME%/Ch03/Activatable/MultiVM/",
                    null);
            HelloInterface myobject 2 = (HelloInterface)Activatable.register(
                                        objectdesc 2);
            Naming.rebind("helloObject_2", myobject_2);

            System.exit(0);

        } catch (Exception e) {
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

El cliente inicia la nueva JVM en el servidor debido a la activación lenta (activación la primera vez que el método es invocado):

```
import java.rmi.*;

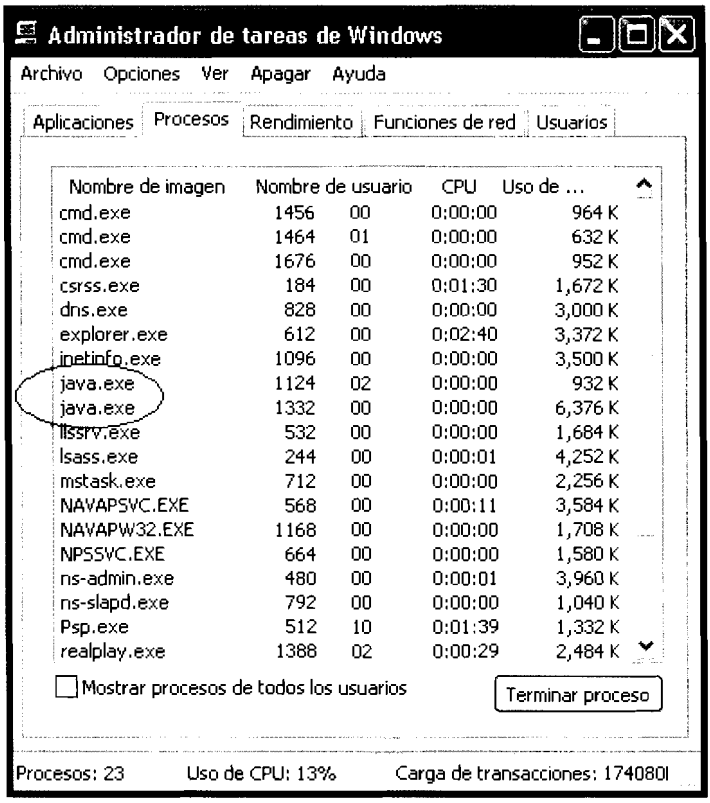
public class HelloClient {

   public static void main(String args[]) {
      if (args.length < 1) {
         System.out.println ("Usage: java HelloClient <host>");
         System.exit(1);
      }
      try {
         HelloInterface obj = (HelloInterface)Naming.lookup(
                                        "rmi://" + args[0] + "/helloObject");
         System.out.println(obj.sayHello());

         // Producir la segunda VM en el servidor
         HelloInterface obj 2 = (HelloInterface)Naming.lookup(
                                        "rmi://" + args[0] + "/helloObject 2");
         System.out.println(obj_2.sayHello());

      } catch (Exception e) {
         System.out.println("HelloClient exception: " + e.getMessage());
         e.printStackTrace();
      }
   }
}
```

Si ejecutamos este ejemplo y observamos los procesos al ejecutar el cliente, veremos que se utilizan dos VM de Java:



Desactivación

La activación permite a un objeto ser iniciado a petición; sin embargo, no desactiva automáticamente un objeto. La decisión de desactivar un objeto queda en manos del mismo objeto. Un objeto activable puede desactivarse a sí mismo invocando el método `Activatable.inactive(ActivationID id)`.

Es importante saber la diferencia entre eliminar un objeto del registro y desactivar un objeto. La desactivación es un estado temporal, permitiendo al sistema de activación reiniciar el objeto más tarde, mientras que al eliminar el objeto mediante`Activatable.unregister(ActivationID id)`, éste queda eliminado permanentemente del sistema de activación.

Otro método útil es el método`Activatable.unexportObject(Remote obj, bolean force)` que puede utilizarse para deshacer la exportación del objeto. El `booleano` es utilizado para forzar al objeto a ser no expoetarse incluso si hay llamadas pendientes o en progreso.

Aunque la activación puede ser un gran avance en rendimiento cuando el número de objetos de un sistema es alto, aliviando el drenaje de los recursos, debe tener cuidado al decidir cómo y cuándo utilizarla. El sobregasto necesario para crear y reactivar un objeto inactivo es importante. Puede suponer crear una nueva JVM, cargar, verificar clases y deserializar un estado de objeto persistente. Un objeto debería normalmente decidir en tiempo seguro mantenerse vivo. Una buena estrategia de desactivación podría basarse en el tiempo transcurrido desde la última llamada.

A continuación, dirigiremos nuestra atención a la capa Secure Sockets (SSL). Esta sección presupone que el lector está familiarizado con los Sockets TCP/IP y la redificación en Java.

Sockets de adaptación y SSL

Un socket es un extremo de comunicación. Dos sockets en colaboración, uno en la máquina local y el otro en la máquina remota, forman una conexión. Esta distinción entre sockets y conexiones es muy importante.

Hay dos tipos de sockets, **sockets de conexión** y **sockets de escucha**, también conocidos como sockets cliente y servidor, respectivamente. Un socket de conexión existe en cada extremo de una conexión TCP abierta y es abstraído por las clases `java.net.ServerSocket` y `java.net.socket`. Un socket de escucha no está asociado a ninguna conexión TCP pero sólo existe como una abstracción para permitir al núcleo TCP decidir qué próximas conexiones son aceptadas y quién obtiene el nuevo socket de conexión aceptado.

En cualquier momento, RMI tiene algunos sockets de escucha, uno para cada puerto escuchador (normalmente sólo uno porque RMI exporta todos los objetos en el puerto "por defecto" si no se especifica un puerto concreto en el método `export()` examinado anteriormente). RMI también crea sockets de conexión para conexiones de salida y para conexiones de entrada.

El número de conexiones de salida sólo depende del número de llamadas de salida concurrentes. La sencilla regla es la siguiente:

- Si un hilo quiere realizar una llamada remota y todas las conexiones al extremo están en uso, entonces el RMI abre una nueva conexión para transportar la llamada
- Si una conexión está libre (es decir, si no hay ninguna llamada en progreso utilizando la conexión), entonces el RMI la reutilizará para la siguiente llamada remota

RMI produce un hilo para escuchar cada socket de escucha (normalmente también uno). Cuando RMI acepta una nueva conexión, crea un nuevo hilo (y un hilo controla la nueva conexión y el otro vuelve para aceptar una nueva conexión). Cuando la conexión se cierra, su hilo asociado también se cierra.

Los hilos de control de conexiones producidos por RMI no están de ningún modo serializados. Si las llamadas llegan a la vez, son ejecutadas en hilos concurrentes. Las llamadas todavía pueden sincronizar objetos Java pero RMI no realiza esa sincronización de forma automática (el objeto remoto es responsable de su propia sincronización, ya sea por métodos sincronizados o por bloqueo sincronizado).

Un punto habitual de confusión es que si el stub remoto es devuelto por una llamada remota, el cliente puede en ocasiones realizar dos conexiones al servidor. Esto ocurre porque el subsistema distribuido de recolección de residuos necesita realizar una llamada `DGC.dirty()` para notificar al servidor que una nueva entidad alberga una referencia al objeto remoto.

Puede utilizar netstat para controlar los sockets de escucha. La siguiente captura de pantalla muestra los sockets justo después de que se inicie el registro en 1099. `HelloServer` *es exportado a 2000 y está asociado al registro. La columna de la izquierda enumera los sockets abiertos de la máquina. La primera línea es para el registro, la segunda para el objeto. La tercera muestra que un socket ha sido abierto por el objeto al registro y la última línea muestra el socket abierto por el registro al objeto.*



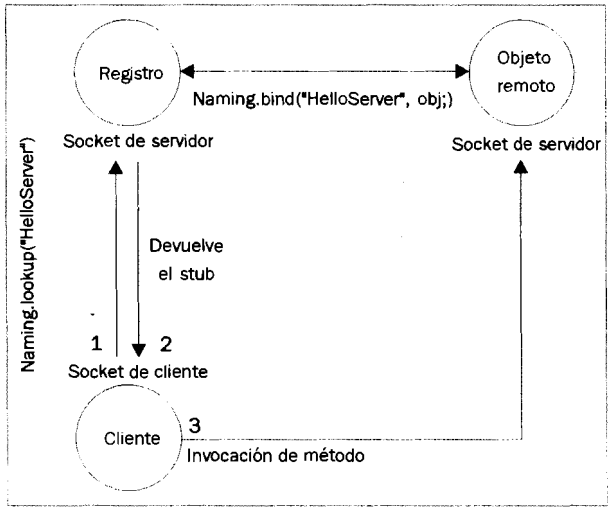
Una mejora notable que ha sido añadida a RMI desde la entrega 1.2 de Java ha sido la capacidad de utilizar factorías de sockets de adaptación basados en el modelo de diseño Factory. En lugar de utilizar los sockets convencionales en TCP/IP, cada objeto tiene la capacidad de utilizar su propio tipo de socket. Esto permite al objeto procesar datos (en vez de pasarlos simplemente), ya sea antes de ser enviado al socket o después de ser recibido por éste.

En JDK 1.1.x, fue posible crear una subclase de adaptación `java.rmi.RMISocketFactory` que produjo un socket de adaptación, diferente a `java.net.Socket`, para ser utilizado por la capa de transporte de RMI. Sin embargo, no fue posible para la factoría de socket instalada producir diferentes tipos de sockets para diferentes objetos. Por ejemplo, en JDK 1.1, una factoría de socket de RMI no podía producir sockets Secure Sockets Layer (SSL) para un objeto y utilizar el Java Remote Method Protocol (JRMP) directamente sobre TCP/IP para un objeto diferente en la misma JVM. Además, antes de la versión 1.2, era necesario que `rmiregistry` utilizara únicamente el protocolo de socket de adaptación del usuario.

La factoría de socket afectaba al sistema completo y no sólo a un objeto.

Para entender qué significa realmente esta afirmación, veamos primero dónde intervienen los sockets de arquitectura RMI, sus tipos y sus propósitos:

Posición	Socket de servidor	Socket de cliente
Registro RMI	Abre un socket de servidor (puerto por defecto 1099) y espera solicitudes de: ❑ Servidores para asociar, reasociar, y desasociar implementaciones de objeto. ❑ Clientes que desean localizar servidores.	Utiliza un socket de cliente para establecer una conexión a un objeto servidor justo después de haberse registrado.
Servidores RMI (implementación remota).	Abre un socket de servidor en un puerto especificado por el usuario vía método `exportObject()` o, por defecto, a un puerto local aleatorio; es utilizado para aceptar conexiones de las respuestas enviadas por el cliente (el stub).	Un socket de cliente es utilizado mientras se conecta al registro, en concreto para registrar la implementación del servidor.
Clientes RMI (objetos que invocan métodos remotos).	No se utiliza.	Los sockets de cliente son utilizados para la comunicación con el registro RMI, para localizar la implementación de servidor y para conseguir que el método RMI llame al servidor.



Cuando un cliente ejecuta una operación de "búsqueda", se realiza una conexión al socket de servidor en el `rmi registry`. En general, puede crearse o no una nueva conexión para una llamada remota. La capa de transporte RMI acumula conexiones para usos futuros. Si alguna conexión existente, y al menos una de ellas está libre, entonces se reutiliza; de no ser así, la implementación actual crea sockets adicionales a petición. Por ejemplo, si un socket existente está siendo utilizado por una llamada existente, entonces se crea un socket nuevo para la nueva llamada. Normalmente, hay como mínimo dos sockets abiertos puesto que el recolector de residuos distribuidos necesita realizar llamadas remotas cuando los objetos remotos son devueltos desde el servidor. Un cliente no tiene control explícito sobre las conexiones a un servidor, ya que las conexiones son gestionadas en el nivel de la capa de transporte RMI. Las conexiones expirarán si se mantienen sin utilizar durante un tiempo.

Como puede ver, existe un número de puntos de comunicación y sockets que son utilizados en este proceso de comunicación de tres vías entre el cliente, el registro y el objeto. Es también importante destacar que el registro sólo es necesario para localizar un objeto por su nombre. Una vez que el objeto está localizado, hay comunicación directa entre el objeto y el cliente.

Un objeto puede especificar qué clases de factoría son necesarias para que alguien se comunique con este tipo de socket a través del constructor `java.rmi.server.UnicastRemoteObject` (o `java.rmi.activation.Activatable` para ese asunto). Instancias de esas factorías son utilizadas para crear instancias de los tipos de sockets deseados:

```
protected UnicastRemoteObject(int port, RMIClientSocketFactory csf,
                              RMIServerSocketFactory ssf)
```

Como hemos explicado en la tabla anterior, una vez que el cliente ha localizado un objeto, utiliza un socket de cliente para conectar al socket de cliente que está escuchando el objeto. La factoría de socket de cliente debe implementar la interfaz `java.rmi.server.RMISocketFactory`. El socket de cliente es creado desde esta instancia de `RMISocketFactory` invocando el método `createsocket()` que devuelve una instancia del tipo de socket adaptado al servidor y al puerto especificados:

```
public Socket createSocket(String host, int port)
```

La factoría de socket de servidor debe implementar la interfaz `java.rmi.server.RMIServerSocketFactory` y proporcionar una implementación para el método `createServerSocket()`. El socket de servidor es creado (para empezar a escuchar llamadas entrantes) invocando la implementación de método en la clase de factoría de socket de servidor:

```
public ServerSocket createServerSocket(int port)
```

Esto es especialmente útil cuando se desea cifrar la comunicación entre los objetos. Por ejemplo, utilizando SSL o TLS. Examinemos un ejemplo de cómo se puede realizar.

Primero necesitaríamos un par de cosas, siendo la principal un proveedor de seguridad que proporciona una implementación del protocolo SSL en Java. Muchos productos terceros disponibles hacen esto.

> **SSL y TLS proporcionan ambos cifrado y certificación. En realidad, Transport Layer Security (TLS) es la nueva norma de Internet Engineering Task Force (IETF) y está basada en SSL.**

Las especificaciones SSL puede encontrarlas en http://home.netscape.com/eng/security/ *y las especificaciones TLS se encuentran en RFC 2246 en* http://www.ieff.org./rfc/rfc2246.txt. *Los detalles sobre las implementaciones SSL de fuente abierta (SSLeay/OpenSSL) puede encontrarlas en* http://www.opnessl.org.

Sun ofrece **Java Secure Sockets Extension (JSSE)** que implementa una versión Java de los protocolos SSL y TLS e incluye funcionalidad para el cifrado de datos, autentificación de servidor, integridad de mensaje y autentificación opcional del cliente.

El ejemplo examinado aquí utiliza JSSE. Necesitará descargarlo de forma independiente para J2SE (1.2 o 1.3) y serán incluido en J2SE (1.4), véase http://java.sun.com/products/jsse/ *para más información.*

Se va producir un rendimiento suplementario y la comunicación va a ser muy lenta debido a la complejidad del acuerdo inicial y el cifrado que supone. El modo en el que el proveedor implementa el algoritmo afectará también el rendimiento. Por ejemplo, una implementación nativa como GoNative Provider (http://www.rtfm.com/puretls/gonative.html) es más rápida que su equivalente Java.

Intentemos primero analizar qué parte de la comunicación queremos/necesitamos cifrar. El registro de designación es implementado como un servicio RMI estándar por lo que los intentos de registrar y buscar servicios supondrán el establecimiento de conexiones de red. Si necesita que la comunicación con el registro tenga lugar en sockets seguros, el registro de designación obviamente necesitará escuchar sobre sockets seguros de servidor.

Un modo muy directo de conseguirlo es que el servidor inicie su propio registro de designación. Este registro se beneficiará entonces del apoyo SSL del servidor. El servidor necesitará invocar y configurar explícitamente una factoría de socket e iniciar el registro:

```
RMISocketFactory.setSocketFactory (some vendor provided factory);
LocateRegistry.createRegistry(someport);
```

La factoría de socket SSL puede entonces ser instalada en el cliente antes de buscar el objeto y todo quedará asegurado por SSL.

> **Puesto que el mismo socket no puede escuchar en dos puertos, sería necesario ejecutar dos registros: uno seguro y uno normal, si necesitara buscar en diferentes protocolos. La mayoría de los vendedores proporciona un registro seguro con la implementación Java-SSL que puede ser utilizado si no está iniciando el registro programáticamente.**

En la mayoría de los casos, sólo es importante y necesita cifrado la comunicación entre los objetos, no la búsqueda; esto es lo que estudiaremos en el siguiente ejemplo. Para que la comunicación tenga lugar sobre

SSL o TLS, necesitamos dos cosas: claves y certificados. Para más información sobre estos dos elementos, consulte los vínculos a la documentación SSL y TLS.

Antes de seguir adelante, generamos las claves y certificados utilizando la utilidad JDK `keytool`. Ésta es una herramienta de gestión de certificados utilizada para gestionar pares de claves públicas/privadas y certificados asociados que deben ser utilizados en auto-autentificación y en servicios de integridad y autentificación de datos, utilizando firmas digitales. En un entorno comercial, probablemente utilizaría `keytool` para generar una solicitud de certificado y utilizaría esa solicitud para obtener un certificado de una Autoridad de Certificados (CA) como Verisign (http://www.verisign.com). Para nuestros propósitos, generaremos un certificado autofirmado (un certificado de la CA autentificando su propia clave pública). En otras palabras, utilizaremos nuestras propias claves y certificados utilizando la opción `-genkey` en `keytool`.

Ejecute la utilidad `keytool` con parámetros similares a éstos:

```
keytool -genkey -dname "cn=ProJavaServer, ou=Java, o=Wrox Press, c=US" -alias wrox
-keypass secret -storepass secret -validity 365 -keystore
%BOOK_HOME%\Ch03\SSL\.keystore
```

Podemos comprobar el certificado utilizando `keytool` de un modo distinto:

```
keytool -list -keystore %BOOK_HOME%\Ch03\SSL\.keystore -storepass secret
```



Esto genera un almacén de claves llamado "`wrox`" que contiene las claves y el certificado auto-firmado y contiene la contraseña "secreta".

> *Consulte la documentación adjunta a JDK para más detalles sobre cómo utilizar keytool, las claves y los certificados en la arquitectura de seguridad de Java 2.*

Finalmente, también exportamos el certificado generado a un archivo `clientimport.cer` para el uso de clientes que utilicen `keytool`:

```
keytool -export -keystore %BOOK_HOME%\Ch03\SSL\.keystore -storepass
secret -file clientimport.cer -alias wrox
```

> *Recuerde que no estamos utilizando SSL basado en navegador en ningún fragmento de estos ejemplos. Si desea utilizarlo, necesitará generar certificados con el algoritmo RSA (utilice la opción `-keyalg` en `keytool`). Para esto, necesitará instalar un proveedor que ofrezca la implementación del algoritmo RSA puesto que JDK 1.2 no se ajusta a esta implementación. Para nuestros propósitos, el proveedor JSSE contiene una implementación y el apoyo RSA está incorporado en JDK 1.3. También, la implementación JCE de libre distribución disponible en http://www.openjce.org es un buen proveedor de API de seguridad.*

Volvamos ahora al ejemplo `HelloWorld` que utilizamos anteriormente y rescribámoslo, para asegurar la comunicación de objeto a objeto.

De nuevo, la interfaz remota se mantiene igual:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloInterface extends Remote {
   public String sayHello() throws RemoteException;
}
```

Ahora rescribimos la implementación remota de esta interfaz:

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class HelloServer extends UnicastRemoteObject
                         implements HelloInterface {
   public HelloServer() throws RemoteException {
      super(0, new MyClientSocketFactory(), new MyServerSocketFactory());
   }
   public String sayHello() {
      return "Hello World, the current system time is " + new Date();
   }
}
```

Observe que la única diferencia es que pasamos nuestras propias factorías de adaptación que serán utilizadas para este objeto en lugar de las factorías por defecto.

Ahora utilizamos el mismo programa que utilizamos antes para registrar este objeto remoto:

```
import java.rmi.*;

public class RegisterIt {

   public static void main(String args[]) {
     try {
        // Instanciar el objeto
        HelloServer obj = new HelloServer();
        System.out.println("Object instantiated"   +obj);
        Naming.rebind ("/HelloServer", obj);
        System.out.println("HelloServer bound in registry");
     }  catch (Exception e) {
        System.out.println(e);
     }
   }
}
```

Estas sentencias que hemos pasado tan fácilmente son las factorías de sockets. Examinémoslas ahora en detalle. La clase `MyServerSocketFactory` es utilizada para crear instancias del socket de servidor en el puerto al que fue exportado el objeto. Esta clase debe implementar la interfaz `java.rmi.server.RMIServerSocketFactory` y debe ser serializable (para facilitar posibles transportes en la red). El intercambio seguro entre el cliente y el servidor conlleva un intercambio de claves y certificados y el socket creado debe primero inicializar la comunicación con el cliente antes de intercambiar ningún dato. La implementación JSEE nos permite realizar esto en unas cuantas líneas de código. Abrimos el almacén de claves (que hemos generado) del archivo utilizando su contraseña, inicializamos la implementación SSL/TLS y devolvemos una instancia de un socket seguro:

```
import java.io.*;
import java.net.*;
import java.rmi.server.*;
import java.security.*;
import javax.net.ssl.*;
```

```
import javax.security.cert.*;
import com.sun.net.ssl.*;

public class MyServerSocketFactory implements RMIServerSocketFactory,
                                               Serializable {
   // Implementar el método de interfaz
   public ServerSocket createServerSocket(int port)throws IOException {
      SSLServerSocketFactory ssf = null;
      try {
         // Configurar el gestor de seguridad para la autentificación de servidor
         char[] passphrase = "secret".toCharArray();

         // Obtener un contexto para el protocolo. Podemos utilizar SSL o
            TLS según sea necesario
         SSLContext ctx = SSLContext.getInstance("TLS");
         KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");

         // Abrir el almacén de claves con la contraseña
         // e inicializar el contexto SSL con este almacén de claves.
         KeyStore ks = KeyStore.getInstance("JKS");
         ks.load(new FileInputStream (".keystore"), passphrase);
         kmf.init(ks, passphrase);
         ctx.init(kmf.getKeyManagers(), null, null);
         ssf = ctx.getServerSocketFactory();
      } catch (Exception e) {
         e.printStackTrace();
      }
      return ssf.createServerSocket(port);
   }
}
```

La factoría de socket de cliente es utilizada para crear instancias de los sockets de cliente que conectan a las instancias de los sockets de servidor generados en la factoría anterior. Se supone que el proveedor JSSE también está instalado en la máquina cliente:

```
import java.io.*;
import java.net.*;
import java.rmi.server.*;
import javax.net.ssl.*;

public class MyClientSocketFactory implements RMIClientSocketFactory,
                                               Serializable {

   public Socket createSocket(String host, int port) throws IOException {

      // Obtenemos la factoría de socket SSL por defecto.
      SSLSocketFactory factory =
                           (SSLSocketFactory)SSLSocketFactory.getDefault();
      SSLSocket socket = (SSLSocket)factory.createSocket(host, port);
      return socket;
   }
}
```

El cliente se matiene igual:

```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloClient {

   public static void main(String args[]) {
      if (args.length < 1) {
```

```
            System.out.println("Usage: java HelloClient <host>");
            System.exit(1);
        }
    try{
            HelloInterface obj =(HelloInterface)Naming.lookup(
                                                "rmi://" + args[0] + "/HelloServer");
            System.out.println("Got a remote reference" + obj);
            System.out.println(obj.sayHello());
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

El archivo `ssl.policy` que utilizamos aquí es en realidad el mismo que utilizamos en los ejemplos anteriores:

```
grant {
   permission java.security.AllPermission;
};
```

Ahora podemos compilar las clases y generar los stubs para `HelloServer` utilizando el `rmic` empaquetado con el JDK.

Es preciso examinar dos pequeños detalles antes de ejecutar el ejemplo. Para que el cliente interactúe e intercambien contraseñas y certificados con el servidor, debe confiar en el proveedor del certificado digital. En otras palabras, como nuestro certificado es auto-generado, la máquina virtual cliente debe:

- Importar explícitamente el certificado en el almacén de claves que está utilizando y marcarlo como certificado confiable
- Utilizar el mismo almacén de claves que utilizamos en el servidor

Para la segunda opción, el proveedor JSEE debe ser instalado en la máquina cliente (si es diferente del servidor) en uno de dos modos posibles:

- **Estáticamente** en el archivo `java.Security`
- **Dinámicamente** utilizando `Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider ());`

Para detalles más específicos sobre estos métodos de instalación, el lector puede recurrir a la documentación JSSE.

El cliente puede importar el certificado desde el archivo que hemos generado anteriormente, `clientimport.cer`, al almacén de claves que está utilizando y marcarlo como confiable utilizando `keytool` con la sentencia `-import`:

```
keytool -import -file clientimport.cer -alias wrox
```



Una vez realizado esto, primero necesitamos iniciar el registro en el servidor (ejecutar `rmiregistry`):

```
rmiregistry -J-Djava.security.policy=ssl.policy
```

registrar entonces el objeto con el registro:

```
java -classpath %CLASSPATH% -Djava.security.policy=ssl.policy -
Djava.rmi.server.codebase=file://%BOOK_HOME%/Ch03/SSL/ RegisterIt
```

> **Recuerde instalar con cuidado JSEE. Ponga los archivos `jar` en su localización `<JAVA_HOME>\lib\ext` e instale el proveedor en su archivo `java.Security` (véase el archivo `INSTALL.txt` dentro del lote JSEE para más detalles). Si tiene múltiples entornos de periodo de ejecución, asegúrese de editar el archivo de seguridad correcto. La solución más sencilla es configurar el PATH para que sólo apunte el JDK que está utilizando.**



Ahora ejecute el cliente y especifique el almacén de claves a utilizar utilizando la propiedad JSSE `javax.net.ssl.trustStore`. Éste es el almacén de claves en el que ha sido importado el certificado o el mismo almacén de datos como servidor.

```
java -Djava.security.policy=ssl.policy -
Djavax.net.ssl.trustStore=.keystore HelloClient localhost
```



Si queremos que el cliente sea un applet, necesitamos primero resolver algunos puntos de configuración:

- El cliente debe tener una JVM J2SE 1.2 (o posterior). En el momento de publicación de este libro, Netscape 6 y 6.1 son los únicos navegadores con soporte J2SE 1.2. Por lo tanto, probablemente necesitemos instalar el módulo complementario de Java. (Este módulo es de descarga única y actualiza automáticamente la JVM del navegador a la última versión. Funciona tanto con IE como con Netscape.)
- La JVM cliente debe tener el almacén de claves y los certificados adecuados instalados.
- La JVM cliente debería tener la política correcta, permitiendo que los sockets sean abiertos de nuevo al servidor.
- La JVM cliente debería tener el proveedor de seguridad instalado o tener los permisos de seguridad adecuados de modo que pueda instalarse un proveedor de seguridad dinámicamente.
- Puede acceder a más información y descargas para el módulo complementario de Java en http://java.sun.com/products/plugin/.

Examinemos el siguiente applet. Es sencillo, similar al ejemplo que hemos visto antes, excepto que se trata de un applet. Es importante captar que el applet no tiene nada que ver con la configuración SSL del navegador. La conexión SSL tiene lugar a nivel del socket RMI. La clase de factoría de socket de cliente es descargada dinámicamente y se realiza la conexión segura al servidor:

```
import java.applet.*;
import java.awt.*;
import java.rmi.*;
import java.security.*;

public class SSLHelloApplet extends Applet{

   String message = "-n/a-";

   HelloInterface obj = null;

   // Obtener una referencia al objeto durante el inicio del applet
   public void init() {
     try{
        Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
        String host="rmi://" + getCodeBase().getHost() + "/HelloServer";
        HelloInterface obj = (HelloInterface)Naming.lookup(
                                                "rmi://localhost/HelloServer");
        message = obj.sayHello();
     }  catch (Exception e){
        System.out.println(e);
     }
   }

   // Mostrar un mensaje en el applet
   public void paint(Graphics g){
      g.drawString(message, 25, 50);
   }
}
```

A continuación, la página HTML (`index.html`) para el applet anterior. Esta página es convertida utilizando la herramienta conversora HTML empaquetada con el módulo complementario de Java. Cuando un cliente se encuentra con esta página, si no existe una versión compatible, el JRE 1.3 será descargado del URL especificado. (La página Web de Sun en este caso):

```
<html>
   <head>
   <title>Hello World</title>
   <head>

   <body>
      <center> <h1>Hello World</h1> </center>

The message from the HelloServer is:
<p>
<!-"CONVERTED APPLET"->
<!- CONVERTER VERSION 1.3 ->
<object classid = "clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        width = 500 height = 120 codebase =
"http://java.sun.com/products/plugin/1.3/jinstall-13-
                                        win32.cab#Version=1,3,0,0">
<param name = code value = "SSLHelloApplet" >
<param name = codebase value = "http://localhost:8080/hello/" >

<param name ="type" VALUE="application/x-java-applet;version=1.3">
```

```
<param name ="scriptable" VALUE="false">
<comment>
<embed type = "application/x-java-applet;version=1.3" code = SSLHelloApplet"
codebase = "http://localhost:8080/hello/" width = 500 height = 120
scriptable=false pluginspage = "http://java.sun.com/products/plugin/1.3/
plugin-install.html"> <noembed></comment>

    </noembed></embed>
  </object>

<!-
<applet code = "SSLHelloApplet" codebase = "http://localhost:8080/hello/"
width = 500 height = 120>
</applet>
->
<!-"end converted applet"->

  </body>
</html>
```

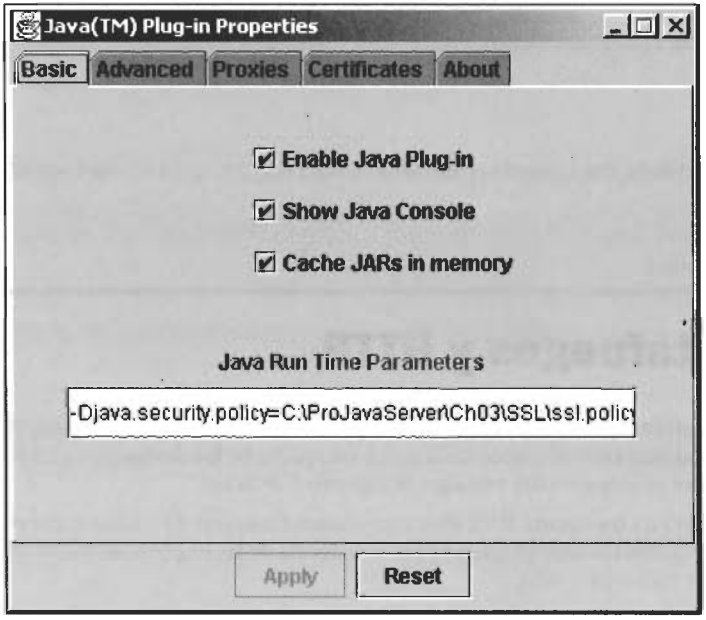
Cuando se accede a la página anterior desde el navegador, el navegador determina a partir de los etiquetas que un plug-in (JRE 1.3) es necesario para cargar este componente y le insta a descargarlo desde la ubicación especificada en el parámetro pluginspage (o codebase, dependiendo del navegador); esto es así, por supuesto, asumiendo que no está ya instalado:



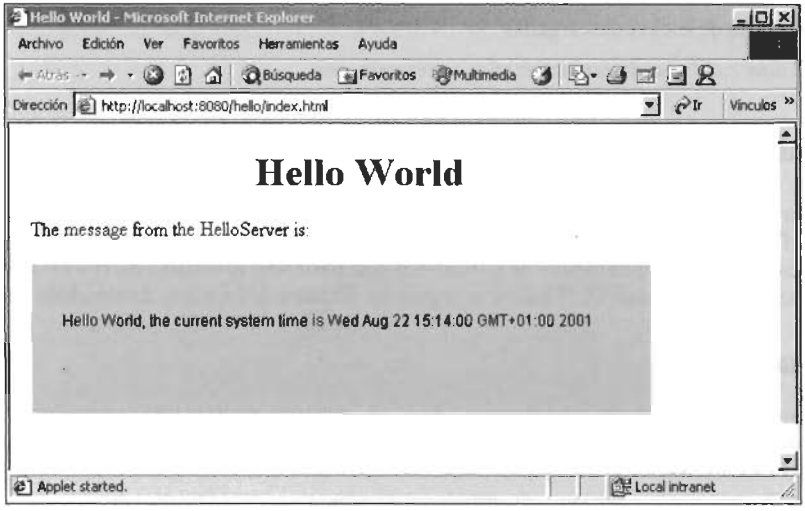
Una vez descargado e instalado el plug-in y una vez lo hayamos habilitado desde su consola en nuestra máquina, podemos acceder a la misma página. El navegador lo cargará entonces en la nueva JVM Java 1.3 (y no en la JVM por defecto del navegador).

Llegados a este punto, necesitamos asegurarnos de que añadimos los parámetros de periodo de ejecución necesarios para el plug-in utilizando el applet `Java Plug-in` en `Control Panel`. Necesitamos añadir los parámetros `-Djava.Security.policy` y `-Djava.net.ssl.trustStore`. En este ejemplo serán:

```
-Djava.security.policy=%BOOK_HOME%\Ch03\SSL\ssl.policy
-Djavax.net.ssl.trustStore=%BOOK_HOME%\Ch03\SSL\.keystore
```

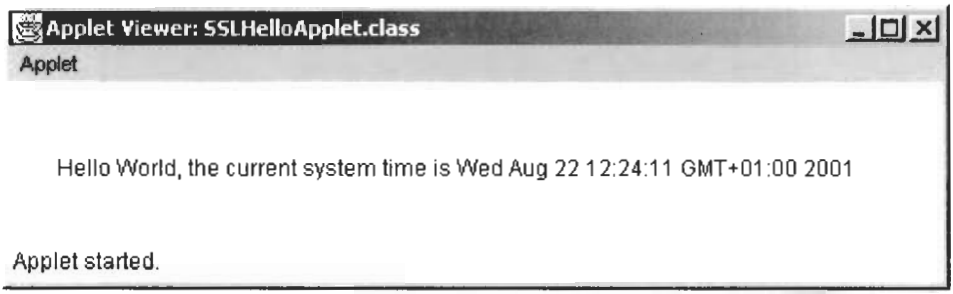


Finalmente, debemos obtener un resultado similar al siguiente:



Se puede acceder al mismo URL utilizando `appletviewer`:

```
appletviewer -J-Djava.security.policy=ssl.policy
             -J-Djavax.net.ssl.trustStore=.keystore
              http://localhost:8080/hello/index.html
```



RMI, cortafuegos y HTTP

Si ha trabajado en cualquier empresa redificada, probablemente esté familiarizado con cómo los cortafuegos bloquean todo el tráfico de la red, a excepción de los destinados a ciertos puertos "conocidos". Sin embargo, son necesarios para proteger la seguridad de la red.

Puesto que la capa de transporte RMI abre conexiones dinámicas de socket entre el cliente y el servidor, el tráfico JRMP es normalmente bloqueado por la mayoría de las implementaciones de cortafuego. RMI proporciona un rodeo para esto.

Hay tres métodos principales para evitar los cortafuegos:

- Tunelado HTTP
- SOCKS
- Factorías de socket descargadas

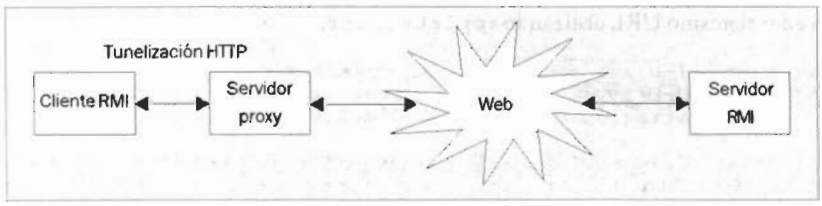
Examinaremos cada uno de estos métodos sucesivamente.

Tunelado HTTP

Para cruzar cortafuegos, RMI hace uso del tunelado HTTP encapsulando las llamadas RMI en una solicitud HTTP POST. Este método está muy extendido ya que no requiere apenas configuración y funciona bastante bien en entornos de cortafuego que permiten el manejo de HTTP a través de un proxy, pero rechaza las conexiones TCP salientes regulares. Existen dos formas de tunelado HTTP:

HTTP a puerto

Si un cliente está detrás de un cortafuegos y RMI no consigue realizar una conexión normal al servidor, la capa de transporte RMI lo reintenta automáticamente encapsulando los datos de llamada JRMP en una solicitud HTTP POST.



Puesto que casi todos los cortafuegos reconocen el protocolo HTTP, el servidor proxy especificado debería poder enviar la llamada directamente al puerto en el que el servidor remoto está escuchando. Es importante que la configuración del servidor proxy sea pasada a la JVM cliente. Una vez que los datos JRMP encapsulados en HTTP son recibidos en el servidor, RMI puede desenvolver automáticamente la solicitud tunelizada HTTP y decodificarla. La respuesta es enviada entonces de vuelta al cliente como datos encapsulados en HTTP.

La configuración del servidor proxy puede ser pasada utilizando propiedades tales como:

```
java -Dhttp.proxyHost=hostname -Dhttp.proxyPort=portnumber HelloClient
```

El tunelado HTTP puede ser inutilizada configurando la propiedad:

```
java.rmi.server.disableHttp=true
```

HTTP a CGI

Si no se puede contactar con el servidor desde el cliente, incluso después de el tunelado, porque también éste se encuentre detrás de un cortafuegos, la capa de Transporte RMI utiliza un mecanismo similar en el servidor. La capa de Transporte RMI sitúa las llamadas JRMP en las solicitudes http y envía esas solicitudes, del mismo modo que el método anterior. Sin embargo, en lugar de enviarlas al puerto servidor, las envía a http://hostname:80/cgi-bin/java-rmi?forward=<port>. Debe haber un servidor HTTP escuchando en el puerto 80 en el proxy, que tiene el `scriptjava-rmi.cgi`. El java-rmi.cgi a su vez invoca una JVM local, desenvuelve el paquete http y envía la llamada al proceso servidor en el puerto designado. Las respuestas basadas del servidor en JRMP RMI son enviadas de vuelta como paquetes HTTP al puerto cliente originario donde RMI desenvuelve de nuevo la información y la envía al stub RMI adecuado.

El script `java-rmi.cgi` está empaquetado con JDK y puede encontrarlo en el directorio `bin`. Para evitar cualquier problema de resolución DNS en el arranque, el nombre calificado completo del host debe ser especificado mediante una propiedad de sistema como:

```
java.rmi.server.hostname=www.host.domain
```

Una implementación de servlet del CGI llamada manejador de servlet está disponible en Sun en: http://java.sun.com/j2se/1.3/docs/guide/rmi/archives/rmiservlethandler.zip.

Otra alternativa a `java-rmi.cgi` es utilizar un redirector de puerto (por ejemplo, DeleGate proxy) en el puerto 80 que acepte conexiones y las redirija inmediatamente a otro puerto.

Aunque el tunelado HTTP es una alternativa, en general debería evitarse por las siguientes razones:

- Hay una significante degradación del rendimiento. Durante el tunelado, la aplicación RMI no podrá multiplexar las llamadas JRMP en una única conexión, debido al paradigma HTTP solicitud-respuesta.
- Utilizar el script java-rmi-cgi (o servlet) es una gran trampa de seguridad en el servidor. El script redirige cualquier solicitud entrante a cualquier puerto, circunvalando por completo el cortafuegos.
- Las aplicaciones RMI tunelizadas sobre HTTP no pueden utilizar retrollamadas.

El protocolo SOCKS

SOCKS (SOCKet Server) es un protocolo proxy de red que habilita a los hosts de un lado del servidor SOCKS para obtener acceso completo a los hosts del otro lado del servidor SOCKS sin requerir

accesibilidad IP directa. El servidor SOCKS redirige las solicitudes de conexión procedentes de hosts situados en lados opuestos de un servidor SOCKS (el servidor SOCKS autentifica y autoriza las solicitudes, establece un proxy de conexión y retransmite datos).

Por defecto, los sockets JDK utilizan un servidor SOCKS si está disponible y configurado. Sin embargo, los sockets de servidor no se ajustan a SOCKS por lo que este enfoque sólo es útil para llamadas salientes desde el cliente al servidor.

Factorías de sockets descargados

Ya hemos examinado con detenimiento cómo pueden utilizarse los sockets de adaptación. Las clases de factoría pueden codificarse para funcionar alrededor de un cortafuegos. Las factorías de sockets cargados dinámicamente ofrecen una buena alternativa a el tunelado HTTP, pero para que el código supere el cortafuegos debe ser codificado en las factorías. Esto es posible si tiene una configuración de red fija y sabe cómo funciona ese determinado cortafuegos. Sin embargo, diferentes clientes pueden tener diferentes cortafuegos y existen distintas cuestiones referentes a derechos de acceso para proporcionar este tunelado y, por supuesto, clases de factoría variables.

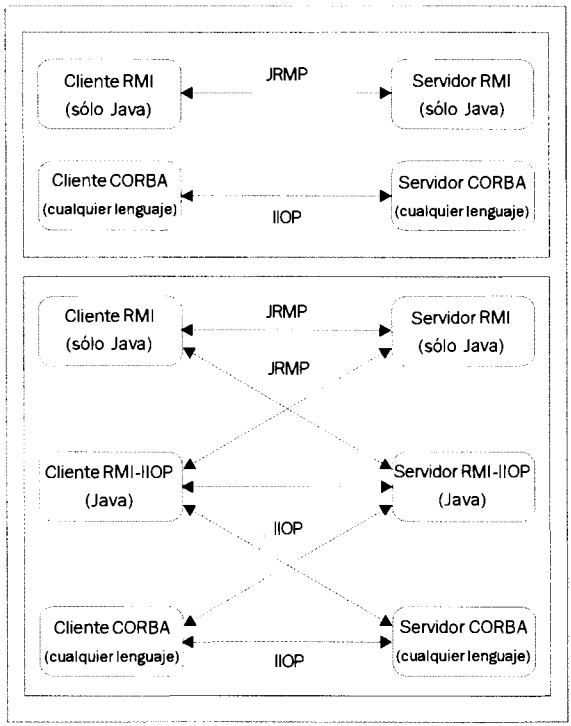
RMI sobre IIOP

RMI sobre Internet Inter-Orb Protocol (IIOP) integra procesamiento distribuido sometido a CORBA (Common Object Request Broker Architecture) directamente en Java. RMI sobre IIOP, desarrollado conjuntamente por IBM y Sun, es una nueva versión de RMI para IIOP que combina las sencillas características de programación de RMI con la interoperatividad de CORBA.

RMI y CORBA han evolucionado de forma independiente como modelos de programación de objetos distribuidos. RMI fue introducido para proporcionar un sencillo modelo de programación para desarrollar objetos distribuidos mientras que CORBA es un modelo de programación de objetos distribuidos muy conocido que se ajusta a una serie de lenguajes. El protocolo IIOP conecta productos CORBA de diferentes vendedores, asegurando la interoperatividad entre ellos.

> *El Grupo de Gestión de Objetos (Object Management Group, OMG), visite* http://www.omg.org/, *es el guardián oficial de información para CORBA y IIOP, incluido las especificaciones CORBA 2.0 disponibles en* http://www.omg.org/technology/documents/specifications.htm. *Adicionalmente, el IDL para representación Java se encuentra disponible en* http://www.omg.org/technology/documents/formal/corba_language_mapping_specs.htm.

RMI-IIOP es, hasta cierto punto, un matrimonio de RMI y CORBA, ya que las interfaces remotas pueden ser escritás en Java e implementadas utilizando los API RMI de Java. Estas interfaces, sin embargo, pueden ser implementadas en cualquier otro lenguaje que se ajuste a una representación OMG y a un ORB surtido por un vendedor para ese lenguaje. De un modo similar, los clientes pueden ser escritos en otros lenguajes, utilizando IDL derivados de las interfaces remotas Java.



La figura anterior resume el matrimonio RMI-IIOP. Puede parecer que las flechas que conectan los clientes /servidores JRMP a los clientes /servidores RMI-IIOP están mal situadas porque son protocolos diferentes. Estas flechas están realmente donde deben estar porque RMI-IIOP se ajusta a los dos protocolos, JRMP y IIOP.

Uno de los objetivos iniciales de diseño era conseguir que la migración a IIOP fuera factible, y de hecho fácil, y evitar la necesidad de un tercer modelo distribuido a aprender por los desarrolladores. El objeto servidor creado utilizando el API RMI-IIOP puede ser exportado como objeto que se ajusta a JRMP o como objeto que se ajusta a IIOP simplemente cambiando las propiedades de tiempo de implementación (sin cambiar o recompilar código).

RMI-IIOP también se ajusta a la exportación dual, lo que significa que un único objeto servidor puede ser exportado para ajustarse a JRMP y a IIOP simultáneamente.

Interoperatividad con CORBA

Un cliente RMI-IIOP no puede necesariamente acceder a todos los objetos CORBA existentes. La semántica de los objetos CORBA definida en IDL es un grupo de la semántica obedecida por los objetos RMI-IIOP, que es la razón por la que un IDL de objeto CORBA existente no puede siempre ser representado en una interfaz Java RMI-IIOP. Sólo cuando la semántica de un determinado objeto CORBA resulta corresponder con la de RMI-IIOP, un cliente RMI-IIOP puede llamar a un objeto CORBA. La conexión entre el cliente RMI-IIOP y el servidor CORBA es en ocasiones (pero no siempre) posible.

Sin embargo, la cuestión no debería exagerarse porque sólo se aplica al tratar con objetos CORBA ya existentes. Fijándonos en la parte inferior de la figura, si diseñamos un objeto nuevo con una interfaz Java RMI-IIOP, podemos realizar las siguientes observaciones:

- La implementación CORBA: puede automáticamente generar su correspondiente IDL con la herramienta `rmic`. Desde este archivo IDL, puede implementarlo como un objeto CORBA en cualquier lenguaje aceptable, como C++ por ejemplo. Este objeto C++ es un puro objeto CORBA que puede ser llamado por un cliente CORBA, así como por un cliente RMI-IIOP sin ninguna limitación. Para el cliente RMI-IIOP, este objeto CORBA C++ aparece como un puro objeto RMI-IIOP porque está definido por una interfaz Java RM-IIOP.
- La implementación RMI-IIOP: el objeto aparece como un objeto CORBA a un cliente CORBA (porque un cliente CORBA puede acceder a él a través de su IDL) y como un objeto RMI para clientes RMI (porque acceden a él a través de su interfaz Java RMI-IIOP).

Resumiendo, la diferencia entre un objeto CORBA y un objeto RMI-IIOP es sólo una cuestión de implementación.

Una de las razones de los problemas con los objetos existentes mencionados anteriormente es que dos mejoras significativas de las especificaciones CORBA 2.3 fueron realizadas en realidad para conseguir la interoperatividad de RMI-IIOP y CORBA. OMG aceptó estas especificaciones:

- **Especificación de objetos por valor:** este concepto ya está definido en Java en forma de Serialización de objetos y está destinado a conseguir que otros lenguajes implementen un protocolo similar.
- **Especificación de representación Java en IDL:** ésta es la representación utilizada para convertir las interfaces de Java RMI en definiciones CORBA IDL. No debería confundirse con la representación IDL en Java definida en CORBA 2.2.

Ambas especificaciones están disponibles en OMG y también puede acceder a ellas en http://java.sun.com/products/rmi-iiop/index.html. OMG ha aceptado oficialmente ambas especificaciones para CORBA 2.3 y JDK 1.3 para incluir RMI-IIOP y un compilador IDL en Java.

Escribir programas con RMI-IIOP

Existen algunas diferencias de desarrollo en sintaxis aunque el modelo en su totalidad sigue siendo el mismo; significativamente, RMI-IIOP utiliza el API JNDI para localizar y registrar objetos. Aunque el procedimiento de desarrollo para RMI-IIOP es prácticamente el mismo que el de RMI (JRMP), el entorno de periodo de ejecución es notablemente diferente en cuanto que la comunicación se realiza a través de un ORB sometido a CORBA 2.3 que utiliza IIOP para la comunicación entre servidores y clientes. Examinemos brevemente estas diferencias.

En el servidor

No hay ningún cambio significativo en el procedimiento de desarrollo para objetos RMI en RMI-IIOP. Los pasos básicos y su orden es el mismo que hemos descrito antes en nuestro ejemplo HelloWorld, con algunos cambios que también reflejan el uso de JNDI para buscar y asociar objetos.

HelloServer.java

```
import java.rmi.*;
import java.util.Date;
import javax.rmi.PortableRemoteObject;
```

La clase de implementación de un objeto remoto:

```
public class HelloServer extends PortableRemoteObject
                         implements HelloInterface   {
```

El objeto `PortableRemoteObject` es similar a `UnicastRemoteObject` pero proporciona la funcionalidad de base en el dominio IIOP.

```
    public HelloServer() throws RemoteException {
      super();          // Call the superclass constructor to export this
object
    }

    public String sayHello() throws RemoteException {
      return "Hello World, the current system time is " + new Date();
    }
}
```

- Genere un vínculo para IIOP con `rmic -iiop`: con la opción `-iiop` el compilador `rmic` genera las clases de stubs y vínculos que obedecen el protocolo IIOP. Sin esta opción `-iiop`, `rmic` genera un stub y un skeleton para el protocolo JRMP, o únicamente stubs si también utiliza la opción `-v1.2` (recuerde que los skeletons no son necesarios en 1.2).
- Ejecute `tnameserv.exe` como un servidor nombre: este servidor proporciona los servicios IIOP `CosNaming` para que los clientes busquen objetos.
- Genere IDL con `rmic -idl` para clientes CORBA (si a nuestro objeto también van a acceder clientes CORBA).
- Al iniciar el servidor, deben configurarse dos importantes variables de entorno para el contexto JNDI.
  a. `java.naming.factory.initial`. Éste es el nombre de la clase a utilizar como factoría para crear el contexto.
  b. `java.naming.provider.url`. Éste es el URL del servicio de designación, similar al formato URL rmi:// descrito anteriormente. El `rmi://` es reemplazado por `iiop://` y el puerto por defecto para el registro es el 900 en lugar del 1099.

Estas propiedades pueden ser especificadas bien al iniciar el servidor, como se muestra a continuación, o bien codificando los valores en `java.rmi.Properties` y pasándolo a `InitialContext` como hacemos en el siguiente ejemplo:

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
     -Djava.naming.provider.url=iiop://129.112.1.9:900 RegisterIt
```

En el cliente

El cliente tiene la misma sentencia de importación y utiliza el contexto JNDI para obtener una referencia al objeto, en lugar de utilizar el registro. El método JNDI `lookup()` devuelve un `java.lang.Object`, que debe ser entonces transmitido utilizando el método `narrow()` de `PortableRemoteObject`.

HelloClient.java

```
import java.rmi.*;
import javax.rmi.PortableRemoteObject;
import javax.naming.InitialContext;

public class HelloClient {

   public static void main(String args[]) {
     try {
```

Cree el contexto JNDI y asócielo a él:

```
        InitialContext ctx = new InitialContext (); // Create the JNDI
                                                                    context
        Object obj = ctx.lookup("/EgServer");
```

Así se asocia el objeto al contexto JNDI igual que hizo el método `Naming.bind()`.

```
HelloInterface myobj = (HelloInterface) PortableRemoteObject
                                    .narrow(obj, HelloInterface.class);

        String message = myobj.sayHello();
        System.out.println(message);
    }   catch (Exception e) {
        System.out.println("HelloClient exception: " + e);
    }
  }
}
```

Otra vez, las propiedades para el contexto JNDI (el nombre de factoría del contexto y URL proveedor) *deben* ser pasadas a las aplicaciones cliente en periodo de ejecución. Y, básicamente, eso es todo.

Sun proporciona una guía detallada empaquetada con JDK sobre la conversión de programas RMI y applets existentes. También se puede acceder a ella online en http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/rmi_iiop_pg.html.

Observe que también podemos escribir un servidor que se ajuste explícitamente a ambos clientes:

- Exportándolo a ambos protocolos utilizando el método `exportObject()`. Obviamente, en este caso no necesitamos ampliar ni `PortableRemoteObject` ni `UnicastRemoteObject`.
- Creando dos `InitialContext`, uno para permitir la asociación al registro RMI y uno para el servicio `COSNaming`, y asociando el servidor en ambos.
- No pasando el servicio de designación como un argumento de línea de comando utilizando la opción `-D`.

Por ejemplo, si quisiéramos convertir a nuestro objeto HelloServer inicial en accesible sobre RMI (JRMP) y sobre RMI-IIOP, entonces se podría modificar como sigue:

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;
import javax.rmi.PortableRemoteObject;
public class HelloServer implements HelloInterface {

   public void HelloServer() throws RemoteException {

      PortableRemoteObject.exportObject(this); // Export for IIOP
      UnicastRemoteObject.exportObject(this);  // Export for JRMP
   }

   public String sayHello() throws RemoteException {
      return "Hello World, the current system time is " + new Date();
   }
}
```

El objeto puede ser registrado con ambos contextos modificando nuestro archivo `RegisterIt`:

```
import java.rmi.RMISecurityManager;
```

```
import java.util.Properties;
import javax.naming.InitialContext;

public class RegisterIt {

   public static void main (String[] args){
     try{
        if(System.getSecurityManager() == null) {
           System.setSecurityManager (new RMISecurityManager());
        }
        HelloServer obj = new HelloServer ();  // Create the object

        // Crear un JDNIContext para JRMP y asociar el objeto al registro
        // Observe que estamos utilizando el registro RMI vía JNDI y no el
        // registro por defecto.
        Properties p 1 = new Properties();
        p 1.put("java.naming.factory.initial",
                  "com.sun.jndi.rmi.registry.RegistryContextFactory");
        InitialContext ctx 1 = new InitialContext ();
        ctx 1.rebind ("HelloServer", obj);
        System.out.println ("HelloServer bound in JRMP registry");

        // Repita el mismo paso para el registro IIOP
        Properties p 2 = new Properties();
        p 2.put("java.naming.factory.initial",
                  "com.sun.jndi.cosnaming.CNCtxFactory");
        InitialContext ctx 2 = new InitialContext (p 2);
        ctx 2.rebind ("RemoteHelloServer", obj);
        System.out.println ("HelloServer bound in IIOP registry");
     } catch(Exception e) {}
   }
}
```

Hemos abarcado muchos temas en esta sección, así que recapitulemos lo que acabamos de describir. Hemos hablado sobre los pasos a seguir en la escritura del servidor y del cliente RMI-IIOP, las diferencias entre ellos y cómo el mismo objeto servidor puede escribirse para ambos clientes RMI-JRMP y RMI-IIOP. Sin embargo, lo que no hemos mencionado es cómo un cliente CORBA puede solicitar servicios de este objeto o cómo el cliente RMI-IIOP puede acceder a un objeto CORBA. Examinemos este punto.

Una vez que tenemos una interfaz Java (estamos reutilizando el `HelloInterface` que escribimos antes), podemos utilizar el `rmic` con la opción `-idl` para generar su fuente IDL.

En nuestro caso, `rmic -idl HelloInterface` produce el siguiente archivo `HelloInterface.idl`:

```
/**
 * HelloInterface.idl
 * Generated by rmic -idl. Do not edit
 * 20 August 2001 18:27:51   BST
 */

#include "orb.idl"
#ifndef   HelloInterface
#define   HelloInterface

   interface HelloInterface {

      ::CORBA::WStringValue sayHello();
   };
#pragma ID HelloInterface "RMI:HelloInterface:0000000000000000"

#endif
```

Este archivo .idl puede ser utilizado por cualquier compilador IDL que se ajuste a CORBA 2.3 proporcionado por un vendedor, podemos generar un stub en nuestro lenguaje elegido (hablamos de C++ ya que es un lenguaje muy común para objetos CORBA) para generar las clases de stub para ese lenguaje.

El mismo compilador puede ser utilizado para generar clases skeleton/vínculo y los objetos de servidor C++ escritos para esas clases y se puede acceder a IDL con los clientes RMI-IIOP que hemos escrito.

RMI-IIOP y Java IDL

Java IDL es un Object Request Broker (ORB) proporcionado con JDK y que puede ser utilizado para definir, implementar y acceder a objetos CORBA desde Java. Java IDL cumple con la especificación CORBA/IIOP 2.0 y el Mapping de OMG IDL para Java.

La primera cuestión relacionada con este tema que se plantean la mayoría de los desarrolladores es: "¿se está eliminando RMI progresivamente a favor de RMI-IIOP?". La respuesta es un rotundo no. Otra pregunta que a veces se plantea a "¿es RMI-IIOP un sustituto para Java IDL?". Una vez más, la respuesta es no. Un cliente RMI-IIOP no puede necesariamente acceder a un objeto CORBA existente. Si queremos utilizar Java para acceder a objetos CORBA que ya hemos escrito, Java IDL es nuestra mejor elección. Con Java IDL, que es también una parte central de la plataforma Java 2, podemos acceder a cualquier objeto CORBA desde Java. La recomendación general de uso para RMI-IIOP y Java IDL es ésta: "Si desea utilizar Java para acceder a recursos CORBA existentes, utilice Java IDL. Si, por el contrario, quiere que los recursos Java RMI sean accesibles a usuarios CORBA, entonces debería utilizar RMI-IIOP. Si su aplicación va a ser toda Java, entonces utilice RMI-IIOP".

> **J2SE v1.3 incluye una nueva versión del idlj compilador de IDL a Java que toma un archivo IDL y genera el servidor Java y las asociaciones de cliente a partir de él.**

El archivo IDL

¿Recuerda el IDL que acabamos de generar? Tomemos ese archivo (`HelloInterface.idl`) como punto de partida para este ejemplo y veamos que podemos escribir objetos CORBA y utilizar Java IDL:

```
#include "orb.idl"

#ifndef    HelloInterface
#define    HelloInterface

   interface HelloInterface {
      ::CORBA::WStringValue sayHello();
   };
#pragma ID HelloInterface "RMI:HelloInterface:0000000000000000"
#endif
```

Dos cosas pueden ocurrir con un archivo IDL. Podemos escribir el servidor o podemos escribir un cliente (como ya hemos mencionado antes en el ejemplo RMI-IIOP). Experimentaremos ambas en este ejemplo utilizando el compilador `idlj`.

La implementación de servidor

Examinemos la secuencia de eventos asociados a la implementación de servidor:

- Genere las asociaciones de lado servidor para el archivo IDL. Esto se realiza con el compilador `idlj`:

```
idlj -i %JAVA_HOME%\lib -fserver HelloInterface.idl
```

Donde `%JAVA_HOME%` es el directorio de instalación de J2SE 1.3. Incluimos el archivo `orb.ifl` con la opción `-i` al compilador `idlj` porque está referenciado en el archivo IDL (recuerde que este IDL con el que empezamos fue autogenerado). La opción `-fserver` indica al compilador que genere las asociaciones de lado servidor.

- Escriba la implementación. En terminología CORBA, esto se denomina una clase sirviente.

El sirviente, `HelloSirviente`, es la implementación de la interfaz IDL y es una subclase de `HelloInterfaceImplBase`, que es generada por el compilador `idlj` (para cada archivo `xxx.idl` el compilador genera un archivo `_xxxImplBase.java`). El sirviente contiene un método para cada operación IDL. En este ejemplo, éste es sólo el método `sayHello()`. Observe que los métodos sirvientes son sólo métodos corrientes Java.

```
import java.util.Date;

public class HelloSirviente extends  HelloInterfaceImplBase {
   public String sayHello() {
      return "Hello World, the current system time is " + new Date();
   }
}
```

- Escriba la clase que asocia la implementaicón (el sirviente) al servicio de designación. Esta clase se denomina el *servidor* en terminología CORBA.

Entonces, la clase `HelloServer` quedará como sigue:

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class HelloServer {
   public static void main(String args[]) {
     try {

        // crear e inicializar el ORB
        ORB orb = ORB.init(args, null);

        // crear el sirvientey conectarlo con el ORB
        HelloSirviente helloRef = new HelloSirviente();
        orb.connect(helloRef);

        // obtener el contexto de designación raíz
        org.omg.CORBA.Object objRef =
           orb.resolve initial references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);

        // Asociar la referencia de sirviente al Contexto de Designación
        NameComponent nc = new NameComponent("Hello", "");
        NameComponent path[] = {nc};
        ncRef.rebind(path, helloRef);

        // Esperar las invocaciones de clientes
        java.lang.Object sync = new java.lang.Object();
        synchronized (sync) {
          sync.wait();
```

```
        }
      }  catch (Exception e) {
         System.out.println("Exception: " + e);
      }
    }
}
```

Así, queda claro en el código anterior que el servidor ejecuta las siguientes acciones:

- Instancia el ORB
- Instancia el sirviente y lo conecta al ORB
- Obtiene una referencia de objeto CORBA para un contexto de designación en el que registrar el Nuevo objeto CORBA
- Asocia el nuevo objeto en el contexto de designación con el nombre "Hello"
- Espera las invocaciones del nuevo objeto

La implementación cliente

Sigamos ahora el procedimiento asociado a la implementación cliente. Necesitaremos crear una subcarpeta cliente antes de ejecutar este paso:

- Genere las asociaciones de lado cliente para el archivo IDL:

```
idlj -i %JAVA_HOME%\lib -td client -fclient HelloInterface.idl
```

- Escriba la clase cliente. Ésta puede ser un applet o una simple clase Java.

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class HelloClient{
   public static void main(String args[]){
     try{

        // crear e inicializar el ORB
        ORB orb = ORB.init(args, null);

        // obtener el contexto de designación raíz
        org.omg.CORBA.Object objRef =
           orb.resolve initial references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);

        // resolver la Referencia de Objeto en Designación
        NameComponent nc = new NameComponent("Hello", "");
        NameComponent path[] = {nc};
        HelloInterface helloRef =
           HelloInterfaceHelper.narrow(ncRef.resolve(path));

        // llamar al objeto servidor Hello e imprimir resultados
         String hello = helloRef.sayHello();
         System.out.println(hello);
      }  catch (Exception e) {
         System.out.println("Exception : " + e) ;
      }
   }
}
```

Finalmente, podemos ejecutar este ejemplo utilizando la siguiente secuencia:

1. Compilar todos los archivos java:

```
javac *.java
```

2. Crear los stubs para el servidor:

```
rmic -iiop HelloServer
```

3. Ejecutar el servicio de Designación `tnameserv`:

```
tnameserv -ORBInitialPort 900
```

4. Iniciar el servidor:

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
     -Djava.naming.provider.url=iiop://localhost:900 HelloServer
```

5. Ejecutar el cliente`HelloServer`:

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
     -Djava.naming.provider.url=iiop://localhost:900 HelloClient
```

Con los dos ejemplos anteriores hemos visto cómo escribir un objeto RMI-IIOP y cómo hacerlo accesible a otros clientes CORBA. También hemos visto cómo tomar un archivo IDL y generar un objeto CORBA y acceder a él con un cliente Java-IDL.

`idlj` está sujeto tanto al modelo de herencia como al de delegación.

Esto significa que dada una interfaz XX en su archivo IDL, `idlj` genera `XXImplBase.java`. La implementación para XX que escribe debe ampliar la clase `_XXImplBase`. Esto se conoce como modelo de herencia.

En ocasiones esto puede no resultar útil si su implementación se amplía desde otra clase. (Recuerde que una clase Java puede implementar cualquier número de interfaces pero sólo puede ampliar una clase.)

En tal caso, puede indicar al compilador `idlj` que utilice en modelo Tie. Si así ocurre para la interfaz XX de su archivo IDL, `idlj` generará `XX_Tie_java` y el constructor para `XX_Tie` tomará un XX como argumento.

La implementación para XX que proporciona sólo es necesaria para implementar la interfaz XX. Para que esta implementación la utilice con el ORB, debe envolver su implementación en `XX_Tie`. Por ejemplo:

```
XXImpl obj = new XXImpl ();
XX_Tie tie = new XX_Tie (obj);
orb.connect (tie);
```

Como puede ver, el inconveniente de utilizar el modelo Tie es que añade un paso extra y una llamada de método extra.

El modelo de lado servidor por defecto del compilador `idlj` es el modelo de herencia. Para utilizar el modelo de delegación:

```
idlj -fserverTIE Hello.idl
```

o:

```
idlj -fallTIE Hello.idl
```

RMI-IIOP y J2EE

RMI-IIOP ofrece las siguientes ventajas para desarrolladores de programas con relación a RMI-JRMP:

- Interoperatividad con objetos escritos en otros lenguajes vía CORBA IDL independientes del lenguaje
- Los contextos de transacción y seguridad pueden ser propagados implícitamente debido al uso de IIOP, que puede propagar implícitamente información de contexto
- Mantenimiento de cortafuegos de base IIOP vía proxy IIOP que pueden pasar tráfico IIOP de un modo controlado y manejable

Con IIOP como protocolo de transporte, Java RMI encuentra el apoyo que necesita para fomentar el desarrollo de aplicaciones distribuidas de capacidad industrial, sólo en un entorno Java. No obstante, RMI-IIOP tiene algunos puntos débiles propios en comparación con RMI-JRMP:

- No tiene asistencia de recolección de residuos distribuidos. Las interfaces RMI DGC no representan los ID de objeto como lo hace CORBA, por lo que esas interfaces no son suficientes para CORBA/IIOP. No puede confiar en las prestaciones de Java RMI/JRMP mientras utiliza RMI-IIOP.
- El operador de conversión de clases no puede ser utilizado directamente en sus clientes después de obtener una referencia de objeto remoto. En su lugar, necesita utilizar un método especial para obtener el tipo adecuado.
- No le está permitido heredar en mismo nombre de método en su interfaz remota desde diferentes interfaces remotas base.
- Todas las definiciones constantes en interfaces remotas deben ser de tipos primitivos o Strings, y evaluadas en tiempo de compilación.

Entonces, ¿qué idea está detrás de la intención de convertir RMI-IIOP en el protocolo de facto en J2EE sustituyendo a RMI-JRMP?

> **Todos los contenedores J2EE son requeridos para mantener JRMP y RMI-IIOP y los contenedores también deberían poder exponer todos los beans de empresa utilizando RMI-IIOP mediante las interfaces remotas de los beans.**

Enterprise JavaBeans (EJB) es un constituyente clave de J2EE; los componentes de EJB residen dentro de los contenedores EJB, que proporcionan los entornos de periodo de ejecución para los componentes. Los contenedores EJB son desarrollados por diferentes vendedores tomando como base las especificaciones EJB (que utilizan la semántica RMI). Los vendedores son libres para elegir la implementación para sus contenedores. Para promover la interoperatividad para los entornos EJB que incluyen sistemas procedentes de diferentes vendedores, Sun ha definido una representación estándar de EJB a CORBA, basada en la especificación de la asociación Java en IDL desde OMG. Es decir, las interfaces EJB son intrínsecamente interfaces RMI-IIOP.

Puede acceder a la representación EJB en CORBA en http://java.sun.com/products/ejb/index.html.

Como veremos en este libro, EJB presenta un modelo de creación de aplicaciones de empresa distribuidas que pueden ser desplegadas en un entorno heterogéneo. RMI estándar con JRMP falla en algunos aspectos:

- Es la única solución de Java y EJB también debería ser accesible para otros clientes en una entorno de empresa
- No obedece a la propagación de contextos de transacción y seguridad en las JVM distribuidas (de hecho, no obedece a la propagación de contexto en absoluto)
- No es tan reajustable como IIOP

RMI-IIOP supera estas limitaciones de RMI-IIOP. La representación de CORBA en EJB no sólo permite la interoperatividad entre implementaciones de múltiples vendedores del contenedor EJB, sino que también capacita a clientes no Java para que accedan a aplicaciones de lado servidor como EJB a través de los API CORBA estándar. La especificación EJB 2.0 permite la operatividad entre contenedores utilizando RMI-IIOP.

Ajustar aplicaciones RMI

El API de RMI ofrece muchas propiedades de sistema que puede ser ejecutadas dinámicamente en el periodo de ejecución con la opción `-D` en la JVM. (Por ejemplo, `java -Djava.rmi.dgc.leaseValue=30000 MyApp`). Son muy útiles para ajustar el rendimiento y depurar las aplicaciones. Estas propiedades están catalogadas en las siguientes tablas, empezando con un resumen de todas las propiedades que pueden ser configuradas en las JVM cliente y servidor:

Propiedad	Introducida primera vez (versión)	Descripción
`java.rmi.dgc.leaseValue`	1.1	La duración del alquiler concedido por el DGC a los clientes que acceden a objetos remotos en esta JVM. Los clientes normalmente renuevan un alquiler cuando ha expirado en su 50%, por lo que un valor muy corto aumentará el tráfico en la red y arriesgará renovaciones tardías a cambio de demora en llamadas a `Unreferenced.unreferenced`. El valor por defecto es 600000 milisegundos (10 minutos).
`java.rmi.server.codebase`	1.1	Las posiciones desde las que las clases publicadas por esta JVM pueden ser descargadas. Puede ser un URL o una lista de URL separada por espacio (para 1.2) y puede ser configurada tanto en JVM cliente como en JVM servidor.
`java.rmi.server.hostname`	1.1	El nombre de host que debe estar asociado a stubs remotos para objetos remotos creados localmente, para permitir a los clientes invocar métodos en el objeto remoto. El valor por defecto se esta propiedad es la dirección IP del host local.
`java.rmi.server.logCalls`	1.1	Llamadas entrantes de método y excepciones serán registradas en `System.err` si este es true.

La tabla continua en la página siguiente

Propiedad	Introducida primera vez (versión)	Descripción
`java.rmi.server.randomIDs`	1.1.8	Si este valor es `true`, los identificadores de objeto para objetos remotos exportados por esta JVM serán generados utilizando un generador criptográfico seguro de números aleatorios. El valor por defecto es `false`.
`java.rmi.server.useCodebaseOnlyclases`	1.1	Si este valor es `true`, la JVM no puede cargar automáticamante desde otro lugar que no sea su classpath local y desde la propiedad `java.rmi.server.codebase` del servidor. Ésta puede configurarse en la JVM cliente y servidor.
`java.rmi.server.useLocalHostName`	1.1.7	RMI utiliza ahora una dirección IP para identificar el host local cuando la propiedad `java.rmi.server.hostname` no está especificada y no se puede obtener un nombre de dominio calificado para el host local. Si se configura en true, RMI está obligado a utilizar el nombre de dominio calificado por defecto.
`java.rmi.server.disableHttp`	1.1	Si este valor es `true`, el tunelado HTTP es desactivada, incluso cuando se configura `http.proxyHost`. El valor por efecto es false y esta propiedad es útil para JVM cliente.

La siguiente tabla describe las propiedades específicas de activación:

Propiedad	Introducida primera vez (versión)	Descripción
`javsa.rmi.activation.port`	1.2	El valor de esta propiedad representa el puerto TCP en el que el demonio de sistema de activación, rmid, escuchará las llamadas remotas entrantes. El valor por defecto es 1098.
`java.rmi.activation.activator.class`	1.2	La clase que implementa la interfaz java.rmi.Activation.Activator. Esta propiedad es utilizada internamente para localizar la implementación residente del activador desde el que puede encontrarse el nombre de clase stub.

Esta tabla final representa un resumen de propiedades que son específicas de la implementación JDK SUN de RMI:

Propiedad	Introducida primera vez (versión)	Descripción
`sun.rmi.activation .execTimeout`	1.2	El tiempo que rimd esperará para que un grupo de activación generado arranque. El valor por defecto es de 30,000 milisegundos.
`sun.rmi.activation .snapshotInterval`	1.2	El número de actualizaciones por las que esperaráen sistema de activación antes de que serialice una instantánea de su estado al archivo de registro rmid del disco. Una "actualización" hace referencia a un cambio persistente en el estado de sistema de activación (por ejemplo, el registro de un objeto activable) desde que fue tomada la última instantánea. Cambiar el valor de esta propiedad puede utilizarse para hacer que rmid se reinicie más rápidamente (tomando instantáneas del registro con más frecuencia) o conseguir que rimd sea eficiente (tomando instantáneas del registro con menor frecuencia). El valor por defecto es 200.
`sun.rmi.log.debug`	1.2	Si este valor es true, los detalles de la actividad de registro de rmid son enviados a System.err.
`sun.rmi.rmid .maxstartgroup`	1.2	El máximo número de JVM de grupo de activación que rmid permitirá simultáneamente en el estado de "generación pero todavía inactivo". Si necesitamos iniciar más JVM, éstas estarán en cola hasta que uno de los intentos actuales de generación sea efectivo o caduque. Esta propiedad no limita el número máximo de VMs activas; está destinada a suavizar picos imprevistos de actividad para evitar alcanzar límites del sistema operativo. Fijar el valor de esta propiedad en un número menor puede resultar en un mayor tiempo de arranque de rmid y fijar el valor a un número mayor puede reducir el tiempo de arranque; fijar este valor demasiado alto puede hacer fallar el rmid porque su sistema puede quedarse sin recursos. El valor por defecto es 3.
`sun.rmi.server .activation.debugExec`	1.2	Si este valor es true, el sistema de activación imprimirá información de depuración a la línea de comando que se utiliza para grupos de activación de generación. Por defecto, el valor es false, por lo que la información de depuración no se imprime.
`sun.rmi.dgc .checkInterval`	1.1	La frecuencia con la que el periodo de ejecución de RMI comprueba los alquileres DGC que ha expirado. El valor por defecto es 300,000 milisegundos.
`sun.rmi.dgc.logLevel`	1.1	Controla el registro de llamadas entrantes y salientes relacionadas con la concesión, renovación y expiración de alquileres.
`sun.rmi.dgc.server .gcInterval`	1.2	Cuando es necesario certificar que los objetos remotos inalcanzables no son exportados y son recogidos por el recolector de residuos oportunamente, el valor de esta propiedad representa el intervalo máximo que permitirá el periodo de ejecución entre recolecciones de residuos en el cúmulo local. El valor por defecto es 60,000 milisegundos.

La tabla continua en la página siguiente

Propiedad	Introducida primera vez (versión)	Descripción
`sun.rmi.loader .logLevel`	1.2	Controla el registro de cada nombre de clase y base de código, siempre que el periodo de ejecución RMI intenta cargar una clase como resultado de unmarshaling un argumento o un valor devuelto. La base de código impresa es la base de código anotada, pero puede que no sea necesariamente la verdadera base de código desde la que se carga la clase; el cargador de clase RMI difiere la carga de clase al cargador de clase de contexto de hilo, que puede que cargue la clase desde el classpath, y no desde la base de código anotada.
`sun.rmi.server .exceptionTrace`	1.2	Controla la salida de señales de pila de lado servidor procedentes de excepciones y errores lanzados por llamadas remotas entrantes lanzadas. Si este valor es `true`, las señales de pilas de excepción serán imprimidas. Por defecto (false), las señales de pilas de excepciones y errores no son imprimidas.
`sun.rmi.transport .logLevel`	1.1	Controla el registro detallado a través de la capa de transporte.
`sun.rmi.transport .tcp.localHost NameTimeOut`	1.1.7	Representa el tiempo que el periodo de ejecución RMI esperará para obtener un nombre de dominio calificado para el host local. El valor por defecto es 10,000 milisegundos.
`sun.rmi.transport .tcp.logLevel`	1.1	Controla el registro detallado para la subcapa de transporte específica de TCP.
`sun.rmi.transport .tcp.readTimeout`	1.2.2	Representa el tiempo invertido como tiempo límite de inactividad para conexiones RMI-TCP entrantes. El valor es pasado a Socket. setSoTimeout. Esta propiedad se utiliza sólo en casos en los que un cliente no ha suprimido una conexión sin usar como debería. El valor por defecto es 7,200,000 milisegundos (2 horas).
`sun.rmi.dgc .cleanInterval`	1.1	Representa el tiempo máximo que el periodo de ejecución RMI esperará antes de reintentar una llamada DGC "limpia" fallida. El valor por defecto es 180,000 milisegundos.
`sun.rmi.dgc.client .gcInterval`	1.2	Cuando es necesario asegurar que las llamadasDGC limpias para referencias remotas inalcanzables son efectuadas oportunamente, el valor de esta propiedad representa el intervalo máximo que el periodo de ejecución RMI permitirá entre recolecciones de residuos del cúmulo local. El valor por defecto es 60,000 milisegundos.
`sun.rmi.loader .logLevel`	1.2	Controla el registro de cada nombre de clase y base de código, siempre que el periodo de ejecución RMI intenta cargar una clase como resultados de lectura de un argumento o un valor devuelto. La base de código impresa es la base de código anotada pero puede que no sea necesariamente la base de código desde la que la clase se carga; el cargador de clase RMI difiere la carga de clase al actual cargador de clase de contexto del thread, que puede cargar la clase desde la ruta de clase, y no desde la base de código marcada.
`sun.rmi.server .logLevel`	1.1	Controla el registro de información relacionada con llamadas salientes, incluido alguna información de reutilización de conexiones.

Propiedad	Introducida primera vez (versión)	Descripción
`sun.rmi.transport.connectionTimeout`	1.1.6	Representa el periodo durante el que las conexiones de socket RMI pueden residir en estado "sin usar", antes de que el periodo de ejecución RMI permita que se liberen (cierren) esas conexiones. El valor por defecto es 15,000 milisegundos.
`sun.rmi.transport.proxy.connectTimeout`	1.1	Representa el máximo tiempo qie el periodo de ejecución RMI esperará para que se complete un intento de conexión (createSocket), antes de intentar contactar con el servidor a través de HTTP. Esta propiedad sólo se utiliza cuando la propiedad http.proxyHost está configurada y el valor de java.rmi.server.disableHttp es false. El valor por defecto es 15,000 milisegundos.
`sun.rmi.transport.proxy.logLevel`	1.1	Controla el registro de eventos (createSocket y createServerSocket) cuando la clase por defecto RMISocketFactory es utilizada. Este tipo de registro puede ser útil para aplicaciones que utilizan RMI sobre HTTP. Los eventos en factorías de socket de adaptación no son registrados en esta propiedad.

Las propiedades RMI de Sun descritas son útiles para depurar pero es importante recordar que no son sostenidas oficialmente y que están sujetas a cambios constantes. Las propiedades que terminan en "`.logLevel`" tiene salida redirigida a `System.err` con posibles valores de "`SILENT`", "`BRIEF`" y "`VERBOSE`".

Resumen

Hemos abarcado mucho terreno en este capítulo, desde la arquitectura básica RMI hasta el tunelado HTTP. Si se siente un poco abrumado por tanta información, tómeselo con calma. Repase los ejemplos y escriba algunas aplicaciones de prueba; eso le hará avanzar.

En el capítulo, hemos tratado:

- La arquitectura RMI.
- Desarrollo de aplicaciones RMI. Las etapas necesarias para la creación de aplicaciones efectivas.
- Pasada de parámetros en RMI, los tipos disponibles y su uso.
- Activación de objetos, los beneficios de ejecución basados en la necesidad.
- RMI, cortafuegos y HTTP. Los diferentes métodos diseñados para permitir el acceso del tráfico JRMP a través de cortafuegos.
- El desarrollo de RMI sobre IIOP y la integración de RMI sobre CORBA.

Recuerde que, aunque RMI-IIOP es el modelo para J2EE, JRMP está todavía muy activo. Si su aplicación requiere únicamente comunicación Java-Java, entonces RMI-JRMP es una solución viable.

La inclusión de RMI-IIOP en el núcleo de JDK (desde la versión 1.3 de J2SE) y su íntima asociación con EJB, lo establece como una tecnología de base para el software intermediario de empresa.

En perspectiva, RMI ha realizado un largo recorrido desde sus primeros días, cuando no existía ningún modelo distribuido en Java. En la actualidad, es el núcleo del modelo distribuido de Java en J2EE y constituye la base para futuras tecnologías, como JINI, que están transformando el modo en el que dispositivos y sistemas serán redificados conjuntamente. Aunque está fuera del alcance de este capítulo, Jini esencialmente revoluciona de forma espectacular el modelo y la arquitectura RMI.

El capítulo cuarto aborda el acceso a las bases de datos utilizando el API Java Database Connectivity (JDBC).

4

Programación de bases de datos con JDBC

Una base de datos relacional es normalmente el recurso primario de datos en una aplicación de empresa. El API **JDBC** ofrece a los desarrolladores de programas un modo de conectar con datos relacionales desde el interior del código Java. Utilizando el API JDBC, los desarrolladores pueden crear un cliente (que puede ser cualquier cosa, desde un applet a un EJB) que pueda conectar con una base de datos, ejecutar instrucciones de **Structured Query Language** (SQL) y que procese el resultado de esas instrucciones.

Si está familiarizado con SQL y las bases de datos relacionales, la estructura del API JDBC es fácil de entender y utilizar. El API proporciona conectividad y acceso a datos en toda la extensión de bases de datos relaciones. Puede conseguir este objetivo porque ofrece una serie de métodos genéricos de acceso a bases de datos para bases de datos relacionales que se ajusten a SQL. JDBC generaliza las funciones de acceso a bases de datos más comunes abstrayendo los detalles específicos de vendedor de una determinada base de datos. El resultado es un conjunto de clases e interfaces, localizadas en el paquete `java.sql`, que pueden ser utilizadas con cualquier base de datos que tenga un **driver JDBC** apropiado. Éste permite que la conectividad JDBC sea suministrada de forma constante para cualquier base de datos. También significa que, asegurándose de que la aplicación se ajusta a las características de las bases de datos disponibles más comunes, una aplicación puede utilizarse con una base de datos diferente cambiando simplemente a un driver JDBC diferente.

Sin embargo, hay mucha más en conectividad de bases de datos en un entorno de empresa que una simple conexión de bases de datos y ejecución de instrucciones. Hay aspectos adicionales que deben ser resueltos incluido el uso de reserva de conexiones para optimizar los recursos de la red y la implementación de transacciones distribuidas. Aunque algunos de los conceptos que se encuentran detrás de estos aspectos son avanzados, veremos que enfrentarse a ellos no es en sí demasiado complejo.

Estudiaremos el **API JDBC 3.0** que, en el momento de publicación de este libro, es la última versión de API JDBC. La versión 2.0 del API JDBC tiene dos partes: **JDBC 2.1 Core API** y **JDBC 2.0 Optional**

Package API, y aunque estos dos API han sido combinados en la versión 3.0, las clases e interfaces JDBC siguen encontrándose en dos paquetes Java: `java.sql`, y `javax.sql`.

- `java.sql`
  Este paquete contiene clases e interfaces diseñadas teniendo en mente la arquitectura tradicional cliente-servidor. Su funcionalidad se centra primordialmente en servicios de programación básicos de bases de datos, como creación de conexiones, ejecución de instrucciones e instrucciones preparadas, y consultas por lotes (batch). Funciones avanzadas tales como actualización de lotes, listas de resultados desplazables, aislamiento de transacciones y tipos de datos SQL también están disponibles.

- `javax.sql`
  Este paquete introduce algunos cambios importantes de arquitectura en la programación JDBC en comparación con java.sql y ofrece mejores transacciones para la gestión de conexiones, transacciones distribuidas y conectividad de legado. Este paquete también introduce reserva de conexiones dirigidas por contenedor, transacciones distribuidas y filas.

No intentaremos cubrir conceptos generales de programación de bases de datos en este capítulo; se requieren conocimientos básicos previos sobre SQL y sobre el desarrollo de aplicaciones cliente-servidor. Un único capítulo no es suficiente para abarcar la programación de bases de datos de forma exhaustiva por lo que adoptaremos un enfoque de alto nivel para el API JDBC. En lugar de intentar analizar, construir y analizar grandes aplicaciones, estudiaremos pequeños fragmentos de código que ilustren el uso de varias clases e interfaces de `java.sql` y `javax.sql`. En concreto, examinaremos los siguientes puntos:

- Conectividad de bases de datos utilizando drivers JDBC
- Cómo pueden realizarse varias operaciones SQL utilizando el API JDBC estándar incluido obtener conexiones de bases de datos, enviar instrucciones SQL a una base de datos y recuperar resultados de consultas de bases de datos
- Cómo representar tipos SQL en Java
- Características avanzadas del API JDBC, incluido listas de resultados desplazables y actualizaciones de lotes
- Nuevas características del API JDBC 3.0, incluidos puntos de salvaguardia

Entonces continuaremos para considerar temas de macro-nivel que incluyen:

- Utilizar el paquete javax.sql y JNDI para obtener conexiones de bases de datos
- Reserva de conexiones, incluido un análisis sobre reserva tradicional de conexiones *versus* reserva de conexiones basada en fuentes de datos
- Transacciones distribuidas (cómo el paquete `javax.sql`, junto con el API Java Transaction, habilita las transacciones distribuidas)
- Los conceptos básicos de una fila y cómo proporcionan un alto nivel de abstracción para el acceso a bases de datos y la fácil serialización de datos

Nos centraremos principalmente en el paquete java.sql, además de en la creación de conexiones de bases de datos y el control programático de transacciones.

Empezaremos comprendiendo cómo las aplicaciones Java pueden conectar y comunicarse con bases de datos.

Drivers de bases de datos

Es importante comprender que JDBC abstrae la conectividad de bases de datos de las aplicaciones Java. Un vendedor de bases de datos proporciona normalmente un conjunto de APIs para acceder a los datos gestionados por el servidor de base de datos. Los vendedores de bases de datos populares como Oracle, Sybase, e InFormix ofrecen APIs de propiedad para el acceso de cliente. Las aplicaciones de cliente escritas en lenguajes nativos como C/C++ puede utilizar estos APIs para obtener acceso directo a los datos. El API JDBC ofrece una alternativa al uso de estos APIs específicos del vendedor. Aunque este elimina la necesidad de que el desarrollador de Java acceda a APIs nativos específicos de vendedor, la capa JDBC puede todavía necesitar en última instancia realizar estas llamadas nativas. Un **driver JDBC** es una capa de software intermediario que traduce las llamadas JDBC a los APIs específicos del vendedor.

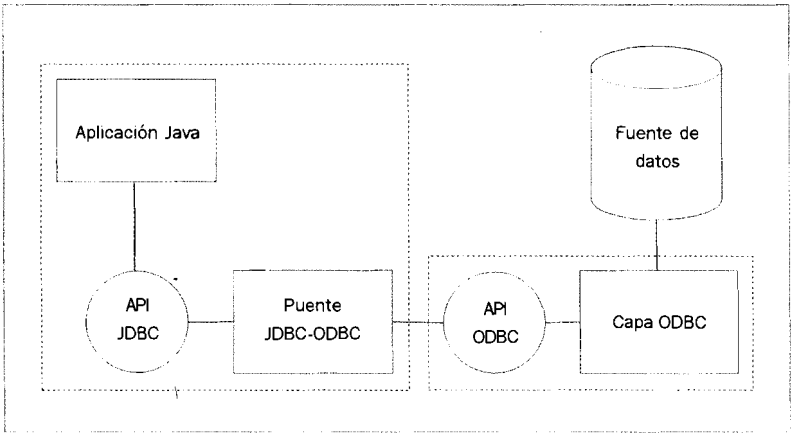
Dependiendo de si se está utilizando el paquete `java.sql` o el paquete `javax.sql`, existen diferentes enfoques para conectarse a una base de datos a través del driver. Con independencia de esto, para acceder a datos necesitará un driver de base de datos, probablemente suministrado por el vendedor de base de datos o por un proveedor del servidor J2EE. Un driver no es más que una implementación de varias interfaces especificadas en los paquetes `java.sql` y `javax.sql`.

Hay cuatro enfoques diferentes para conectar una aplicación a un servidor de base de datos a través de un driver de base de datos. La siguiente clasificación es una norma de industria y los productos comerciales de driver están clasificados según esta norma.

Tipo 1 – Puente JDBC-ODBC

Open Database Connectivity (ODBC) fue creado originalmente para proporcionar un API estándar para SQL en las plataformas Microsoft Windows, y fue más tarde perfeccionado para proporcionar SDK a otras plataformas. ODBC está basado, en parte, en la especificación X/Open Call-Level Interface (CLI), que es un API estándar para el acceso a bases de datos. Este API proporciona asociaciones en los lenguajes C y Cobol para el acceso a bases de datos. CLI está pretende ser neutral en cuanto a plataformas y bases de datos, ahí donde diverge ODBC de la especificación. SQL integrado fue uno de los enfoques barajados para el acceso a bases de datos por SQL Access Group. Esto suponía integrar instrucciones SQL en aplicaciones programadas en lenguajes de alto nivel y preprocesarlas para generar llamadas de función nativa. ODBC define un conjunto de funciones para acceso directo a datos, sin la necesidad de integrar SQL en aplicaciones cliente.

La primera categoría de drivers JDBC proporciona un puente entre el API JDBC y el API ODBC. El puente traduce las llamadas JDBC estándar a llamadas ODBC correspondientes y las envía a la fuente de datos ODBC a través de librerías ODBC:



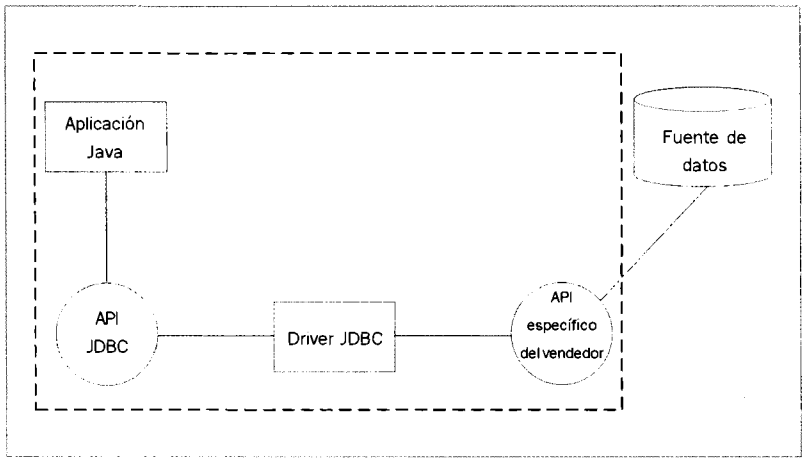
Los límites de proceso están marcados por una línea discontinua. El **Puente JDBC-ODBC** traduce las llamadas de API JDBC en llamadas equivalentes ODBC. El driver delega entonces estas llamadas en la fuente de datos. Las clases Java para el API JDBC y el Puente JDBC-ODBC son invocadas en el proceso de aplicación del cliente; la capa ODBC se ejecuta en otro proceso. Esta configuración requiere que cada cliente que ejecute la aplicación tenga el API Puente JDBC-ODBC, el driver ODBC y el API de lenguaje de nivel nativo, como la librería OCI para Oracle, instalado.

Esta solución para el acceso a datos no es eficiente para requisitos de acceso a bases de datos de alto rendimiento debido a las múltiples capas de desreferencia para cada llamada de acceso a bases de datos. Además esta solución limita la funcionalidad del API JDBC a favor de la funcionalidad del driver ODBC. El uso de JDBC-ODBC debe ser considerado únicamente para propósitos experimentales.

J2SE incluye clases para el Puente JDBC-ODBC y por ello no es necesario instalar ningún paquete adicional para utilizarlo. Sin embargo, sí tiene que configurar el gestor de ODBC creando nombres de fuente de datos, **data source names (DNS)**. Los DNS son simplemente configuraciones nombradas que vinculan una base de datos, un driver apropiado y algunas configuraciones adicionales. El puente JDBC-ODBC debe funcionar con la mayoría de los drivers ODBC 2.0.

Tipo 2 – Parte Java, parte driver nativo

Los drivers Tipo 2 utilizan una combinación de implementación Java y API nativos específicos de vendedor para proporcionar acceso a datos. Hay una capa menos que atravesar que en el driver Tipo 1 por lo que, en general, un driver Tipo 2 será más rápido que un driver Tipo 1:

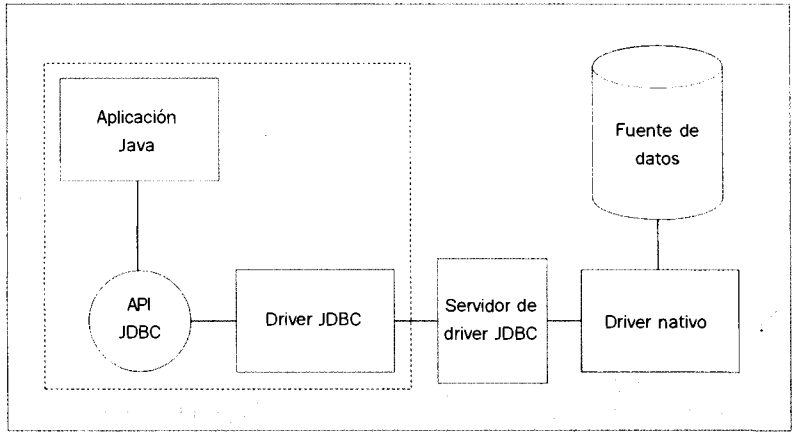


Las llamadas de bases de datos JDBC son traducidas en llamadas API específicas del vendedor. La base de datos procesará la solicitud y enviará de vuelta los resultados a través del API, que a su vez la enviará de vuelta al driver JDBC. El driver JDBC traducirá los resultados a la norma JDBC y los devolverá a la aplicación Java.

Igual que los drivers Tipo 1, la parte Java, el driver de código parte nativa y el API de lenguaje nativo específico del vendedor deben estar instalados en cada cliente que ejecute la aplicación Java. Como el código nativo utiliza protocolos específicos de vendedor para comunicar con la base de datos de forma eficiente, los drivers Tipo 2 son más eficientes que los drivers Tipo1. Además, ahora tenemos uso pleno del API del vendedor. Estos dos factores significan que los drivers Tipo 2 son, en general, preferidos a los drivers Tipo 1.

Tipo 3 – Servidor intermediario de acceso a bases de datos

Los drivers Tipo 3 utilizan un servidor intermediario (software intermediario) de base de datos que tiene la capacidad de conectar múltiples clientes Java a múltiples servidores de base de datos:



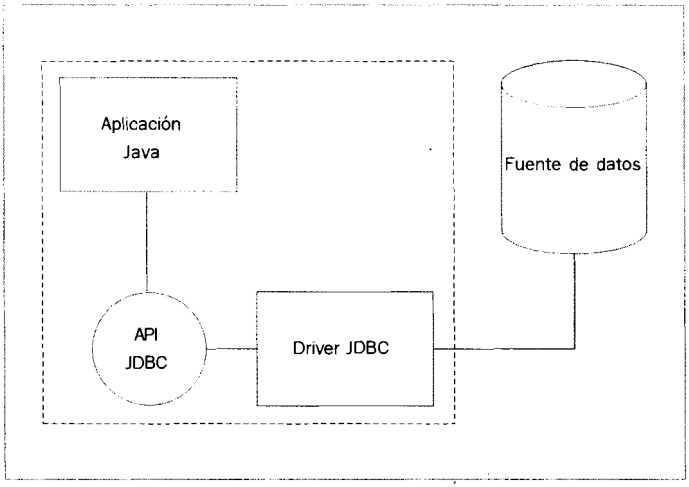
Los clientes conectan a los servidores de base de datos mediante un componente servidor intermediario (como un escuchador) que actúa como una puerta para múltiples servidores de bases de datos. La aplicación cliente Java envía una llamada JDBC a través de un driver JDBC al servidor intermediario de acceso a datos, que completa la solicitud a la Fuente de Datos utilizando otro driver (por ejemplo, un driver Tipo 2).

El protocolo utilizado para comunicar entre clientes y el servidor intermediario depende del vendedor del servidor de software intermediario pero el servidor intermediario puede utilizar diferentes protocolos nativos para conectar con diferentes bases de datos.

BEA WebLogic incluye un driver Tipo 3. Uno de las ventajas de utilizar un driver Tipo 3 es que concede flexibilidad a la arquitectura de la aplicación, ya que el servidor intermediario puede abstraer detalles de conexiones a servidores de bases de datos.

Tipo 4 – Drivers Java puro

Los drivers Tipo 4 son una alternativa Java puro a los drivers Tipo 2:



Los drivers Tipo 4 convierten las llamadas de API JDBC en llamadas de red directas utilizando protocolos de red específicos del vendedor. Lo realizan estableciendo conexiones de socket directas con la base de datos. Los driver Tipo 4 ofrecen generalmente mejor rendimiento que los drivers Tipo 2 y Tipo 2. Los drivers Tipo 4 son también los drivers más fáciles de desplegar ya que no hay librerías adicionales ni software intermediario que instalar. Todos los principales vendedores de bases de datos proporcionan drivers JDBC Tipo 4 para sus bases de datos y también están disponibles en vendedores de terceros.

Puede encontrar una lista de drivers JDBC, de los cuatro tipos, en http://industry.java.sun.com/products/jdbc/drivers/. En el momento de escribir este libro, hay más de 160 drivers disponibles.

Cómo comenzar

Para empezar a utilizar JDBC con nuestras aplicaciones, necesitamos instalar un driver. Obviamente, también necesitamos un servidor de base de datos. Para mayor simplicidad, en este capítulo, utilizaremos la

base de datos Cloudscape para demostrar que el API JDBC, como servidor de Implementación de Referencia J2EE de Sun, incluye una versión de Cloudscape. Cloudscape está escrito utilizando Java y puede encontrarlo en `%J2EE_HOME%\lib\Cloudscape\Cloudscape.jar`.

También puede obtener una copia de evaluación de Cloudscape en http://www.cloudscape.com. Para utilizar el API JDBC y la base de datos Cloudscape, añada el archivo `cloudscape.jar` a su `CLASSPATH`. Consulte la documentación de Cloudscape para instrucciones sobre cómo utilizar CloudView, una interfaz gráfica de usuario (GUI) para acceder a la base de datos Cloudscape.

Cloudscape es un sistema de gestión de base de datos de pequeño contorno construido en Java y que puede ser utilizado de modo integrado o cliente-servidor. Cuando es utilizado de modo integrado, Cloudscape se ejecuta dentro del proceso de aplicación. Éste es el modo más sencillo para la gestión de base de datos puesto que, aparte de incluir las clases Cloudscape en su `CLASSPATH`, no supone ninguna configuración. El modo cliente-servidor le permite acceder a Cloudscape en la forma tradicional cliente-servidor.

Como Cloudscape es implementado en Java, no se requieren drivers adicionales de bases de datos ya que las llamadas JDBC son representadas en llamadas API Java Cloudscape dentro del mismo proceso. En el modo cliente-servidor, estas llamadas son representadas en llamadas RMI en el proceso del servidor Cloudscape.

Si desea crear otra base de datos, asegúrese de que sigue las instrucciones para configurar la base de datos dadas por el vendedor de la base de datos e instale un driver de base de datos adecuado.

El paquete java.sql

Las clases del paquete `java.sql` pueden dividirse en los siguientes grupos basados en su funcionalidad:

- Gestión de conexión
- Acceso a bases de datos
- Tipos de datos
- Metadatos de base de datos
- Excepciones y advertencias

Examinemos las clases disponibles en cada grupo.

Gestión de conexión

Las siguientes clases/interfaces le permiten establecer una conexión a la base de datos. En la mayoría de casos, conlleva una conexión a la red:

Clase	Descripción
`java.sql.DriverManager`	Esta clase proporciona la funcionalidad necesaria para gestionar uno o más drivers de base de datos. Cada driver a su vez le permite conectar con una base de datos específica.
`java.sql.Driver`	Ésta es una interfaz que abstrae el protocolo de conexión específico del vendedor. Puede encontrar implementaciones de esta interfaz en vendedores de bases

La tabla continua en la página siguiente

Clase	Descripción
`java.sql.Array`	Esta interfaz proporciona una abstracción de ARRAY de lenguaje Java, una colección de tipos de datos SQL.
`java.sql.Blob`	Esta interfaz proporciona una abstracción de lenguaje Java de SQL tipo BLOB.
`java.sql.Clob`	Esta interfaz proporciona una abstracción de lenguaje Java de SQL tipo CLOB.
`java.sql.Date`	Esta clase proporciona una abstracción de lenguaje Java de SQL tipo DATE. Aunque `java.util.Date` proporciona una representación de fecha de propósito general, la clase `java.sql.Date` es preferible para representar datos en aplicaciones centradas en bases de datos, ya que este tipo representa directamente en tipo SQL DATE. Observe que la clase `java.sql.Date` amplía la clase `java.util.Date`.
`java.sql.Time`	Esta clase porpociona una abstracción de lenguaje Java de SQL tipo TIME y amplía la clase `java.util.Date`.
`java.sql.Timestamp`	Esta clase porpociona una abstracción de lenguaje Java de SQL tipo TIME, y amplía la clase `java.util.Date`.
`java.sql.Ref`	Esta clase proporciona una abstracción de lenguaje Java de SQL tipo REF.
`java.sql.Struct`	Esta interfaz proporciona una abstracción de lenguaje Java de tipos estructurados SQL.
`java.sql.Types`	Esta clase alberga un conjunto de número enteros, cada uno de ellos se corresponde con un tipo SQL.

Como veremos más adelante, además de las anteriores, el API JDBC también especifica representaciones estándar entre tipos primitivos en Java y en SQL.

El API JDBC también incluye recursos para tratar tipos de bases de datos adaptadas. Estos tipos pueden ser representados como objetos `java.sql.SQLData` y se puede acceder a los datos contenidos en estos tipos utilizando las interfaces `java.sql.SQLInput` y `java.sql.SQLOutput` que proporcionan una interfaz tipo corriente (stream) para acceder a los datos.

Metadatos de base de datos

El API JDBC también incluye recursos para obtener metadatos sobre la base de datos, parámetros para las instrucciones y resultados:

Clase	Descripción
`java.sql.DatabaseMetadata`	Puede encontrar información sobre las características de bases de datos utilizando esta interfaz. Puede obtener una instancia de esta interfaz utilizando `java.sql.Connection`.

La tabla continua en la página siguiente

Clase	Descripción
`java.sql.ResultSetMetaData`	Esta inetrfaz propociona métodos para acceder a metadatos del `ResultSet`, como los nombres de las columnas, sus tipos, el nombre de tabla correspondiente y otras propiedades.
`java.sql.ParameterMetadata`	Esta interfaz le permite acceder a los tipos de parámetros de bases de datos en instrucciones preparadas.

Excepciones y advertencias

Las siguientes clases encapsulan errores de acceso a bases de datos y advertencias:

Clase	Descripción
`java.sql.SQLException`	Esta excepción representa todas las condiciones de excepción relacionadas con JDBC. Esta excepción también incorpora todas las excepciones relacionadas con el driver/base de datos y códigos de error.
`java.sql.SQLWarning`	Esta excepción representa las advertencies de acceso a base de datos. En lugar de tener que capturar esta excepción, puede utilizar los métodos apropiados en `java.sql.Connection`, `java.sql.Statement`, y `java.sql.ResultSet` para acceder a las advertencias.
`java.sql.BatchUpdateException`	Éste es una caso especial de `java.sql.SQLException` destinada a actializaciones de lotes. Estudiaremos las actualizaciones de lotes más adelante en este capítulo.
`java.sql.DataTruncation`	Éste es un caso especial de `java.sql.SQLWarning` destinado a errores de truncamiento de datos. Observe que los tipos de datos no siempre coinciden entre Java y SQL, y la transferencia de datos puede suponer truncamiento de datos.

Además de estos rasgos, JDBC también proporciona recursos para registros y permisos de seguridad asociada para registros.

En las siguientes secciones, estudiaremos algunas de las clases e interfaces más utilizadas de este API. En particular, aprenderemos:

- Cómo cargar un driver de base de datos
- Cómo abrir una conexión de base de datos
- Cómo enviar instrucciones SQL a bases de datos para su ejecución
- Cómo extraer resultados devueltos de una consulta a una base de datos
- Qué son las instrucciones preparadas
- La función de los tipos JDBC
- A manejar excepciones y advertencias

Cargar un driver de base de datos y abrir conexiones

La interfaz `java.sql.Connection` representa una conexión con una base de datos. Es una interfaz porque la implementación de una conexión depende de la red, del protocolo y del vendedor. El API JDBC ofrece dos vías diferentes para obtener conexiones. La primera utiliza `java.sql.DriverManager` y es adecuada para aplicaciones no gestionadas como clientes de base de datos Java autónomos. El segundo enfoque está basado en el paquete `javax.sql` que introduce la noción de fuentes de datos y es adecuado para el acceso a bases de datos en aplicaciones J2EE.

Comencemos considerando cómo se obtienen las conexiones utilizando la clase `java.sql.DriverManager`. En una sola aplicación, podemos obtener una o más conexiones para una o más bases de datos utilizando diferentes drivers JDBC. Cada driver implementa la interfaz `java.sql.Driver`. Uno de los métodos que define esta interfaz es el método `connect()` que establece una conexión con la base de datos y devuelve un objeto `Connection` que encapsula la conexión de base de datos.

En lugar de acceder directamente a clases que implementan la interfaz `java.sql.Driver`, el enfoque estándar para obtener conexiones es registrar cada driver con `java.sql.DriverManager` y utilizar los métodos proporcionados en esta clase para obtener conexiones. `java.sql.DriverManager` puede gestionar múltiples drivers. Antes de entrar en los detalles de este enfoque, necesitamos entender cómo JDBC representa la posición de una base de datos.

Los URL de JDBC

La noción de un URL en JDBC es muy similar al modo en que los URL se utilizan en otras situaciones. Para poder entender la base lógica de los URL de JDBC, consideremos una aplicación que utiliza diversas bases de datos; a cada base de datos se accede mediante un diferentes driver de base de datos. En tal situación, ¿cómo identificamos de forma exclusiva un driver? Además, las bases de datos utilizan diferentes tipos de parámetros para obtener conexiones. ¿Cómo se especifican los parámetros al establecer las conexiones?

Los URL de JDBC proporcionan un modo de identificar un driver de base de datos. Un URL de JDBC representa un driver y la información adicional específica del driver para localizar una base de datos y conectarla a él. La sintaxis del URL de JDBC es como sigue:

```
jdbc:<subprotocol>:<subname>
```

Tiene tres partes separadas por dos puntos:

- Protocolo: En la sintaxis anterior, jdbc es el protocolo. Éste es el único protocolo permitido en JDBC.
- Sub-protocolo: El sub-protocolo es utilizado para identificar un driver de base de datos, o el nombre de un mecanismo de conectividad de una base de datos, elegido por los proveedores del driver de la base de datos.
- Subnombre: La sintaxis del subnombre es específica de driver. Un driver puede elegir cualquier sintaxis apropiada para su implementación.

Por ejemplo, para una base de datos Cloudscape llamada "Movies", el URL al que debe conectar es:

```
jdbc:cloudscape:Movies
```

Alternativamente, si estuviéramos utilizando Oracle mediante el puente JDBC-ODBC, nuestro URL sería:

```
jdbc:odbc:Movies
```

donde `Movies` es DNS establecido utilizando un administrador de driver ODBC.

Como puede ver, los URL de JDBC son lo suficientemente flexibles como para especificar información específica del driver en el subnombre.

DriverManager

El propósito de la clase `java.sql.DriverManager` (gestor de drivers) en JDBC es proporcionar una capa de acceso común encima de diferentes drivers de base de datos utilizados en una aplicación. En este enfoque, en lugar de utilizar clases de implementación `Driver` individuales directamente, las aplicaciones utilizan la clase `DriverManager` para obtener conexiones. Esta clase ofrece tres métodos estáticos para obtener conexiones.

Sin embargo, `DriverManager` requiere que cada driver que necesite la aplicación sea registrado antes de su uso, de modo que el `DriverManager` sepa que está ahí.

El enfoque JDBC para el registro de un driver de base de datos puede parecer oscuro al principio. En JDBC, intente cargar el driver de base de datos utilizando el actual `java.lang.ClassLoader`. Fíjese en el siguiente fragmento de código, que carga el driver de base de datos de Cloudscape:

```
try {
   Class.forName("COM.cloudscape.core.JDBCDriver");
} catch (ClassNotFoundException e) {
   // Driver no encontrado.
   ...
}
```

En el tiempo de ejecución, el `ClassLoader` localiza y carga la clase `COM.Cloudscape.core.JDBCDriver` desde al classpath utilizando el cargador de clase de autoarranque (bootstrap). Mientras carga una clase, el cargador de clase ejecuta cualquier código estático de inicialización para la clase. En JDBC, se requiere que cada proveedor de driver registre una instancia del driver con la clase `java.sql.DriverManager` durante esta inicialización estática. Este registro tiene lugar automáticamente cuando el usuario carga la clase del driver (utilizando la llamada `Class.forName()` como anteriormente). `java.sql.DriverManager` es una clase estática y proporciona el siguiente método para este propósito:

```
public static void registerDriver(java.sql.Driver)
```

Alternativamente, podemos especificar una lista de drivers utilizando el mecanismo de propiedades Java. Por ejemplo, el siguiente fragmento permitiría a la clase `java.sql.DriverManager` cargar la lista de drivers cuando se realizara un intento de establecer una conexión:

```
System.setProperty("jdbc.drivers", "COM.cloudscape.core.JDBCDriver");
```

Puede especificar múltiples drivers en forma de lista separada por dos puntos (:).

Una vez que el driver ha sido registrado con el `java.sql.DriverManager`, podemos utilizar sus métodos estáticos para obtener conexiones. Más adelante, en este mismo capítulo, examinaremos la interfaz `javax.sql.DataSource` que proporciona una mejor alternativa a las aplicaciones `java.sql.DriverManager` de J2EE. Sin embargo, puede que todavía sea necesario que utilice `java.sql.DriverManager` para el acceso directo a bases de datos fuera de las aplicaciones J2EE.

La clase `java.sql.DriverManager` especifica los siguientes tipos de métodos:

- Métodos para gestionar drivers
- Métodos para obtener conexiones
- Métodos para registros

Examinemos cada una de estas áreas sucesivamente.

Métodos para gestionar drivers

Los siguientes métodos gestionan drivers. Observe que estos métodos están dirigidos a la implementación de drivers y a las herramientas que manipulan drivers:

```
public static void registerDriver(Driver driver)
```

Este método es utilizado para registra un driver con el DriverManager. Una clase de driver recientemente cargada debe llamar este método para hacerse conocer por el DriverManager. Las clases que implementan la interfaz Driver llaman este método durante la inicialización estática para registrar una clase. Una vez registrada, el gestor de driver alberga una referencia al driver hasta que éste es desregistrado. Por razones de seguridad, el gestor de driver asocia el cargador de clases del llamante con cada driver, de modo que las clases cargadas desde un cargador de clase tengan acceso únicamente a aquellos drivers que son registrados por clases cargadas por el mismo cargador de clases.

```
public static void deregisterDriver(Driver driver)
```

Este método desregistra un driver del gestor de drivers.

```
public static Driver getDriver(String url)
```

Dado un URL de JDBC, este método devuelve un driver que puede entender el URL.

```
public static Enumeration getDrivers()
```

Este método devuelve una enumeración de todos los drivers registrados JDBC registrados por clases utilizando el mismo cargador de clase.

Métodos para obtener conexiones

El gestor de drivers tiene tres variantes de método estático `getConnection()` utilizado para establecer conexiones. El gestor de drivers delega estas llamadas en el método `connect()` de la interfaz `java.sql.Driver`.

Dependiendo del tipo de driver y del servidor de base de datos, una conexión puede conllevar una conexión de red física al servidor de base de datos o un proxy a una conexión. Las bases de datos integradas no requieren conexión física. Haya o no una conexión física, el objeto de conexión es el único objeto que utiliza una conexión para comunicar con la base de datos. Toda comunicación debe tener lugar dentro del contexto de una o más conexiones.

Consideremos ahora los diferentes métodos para obtener una conexión:

```
public static Connection getConnection(String url) throws SQLException
```

`java.sql.DriverManager` recupera un driver apropiado del conjunto del drivers JDBC registrados. El URL de la base de datos está especificado en forma de `jdbc:subprotocol:subname`. Que obtengamos o no una conexión con este método depende de que la base de datos acepte o no solicitudes de conexión sin autentificación.

```
public static Connection getConnection(String url,
                                       java.util.Properties info)
        throws SQLException
```

Este método requiere un URL y un objeto `java.util.Properties`. El objeto `Properties` contiene cada parámetro requerido para la base de datos especificada. La lista de propiedades difiere entre bases de datos. Dos propiedades comúnmente utilizadas para Cloudscape con `autocommit=true` y `create=false`. Podemos especificar estas propiedades junto con el URL como `jdbc:Cloudscape:Movies; autocommit=true;create=true` o podemos establecer estas propiedades utilizando el objeto `Properties` y pasar el objeto `Properties` en el método anterior `getConnection()`.

```
String url = "jdbc:cloudscape:Movies";
Properties p = new Properties();
p.put("autocommit", "true");
p.put("create","true");
Connection connection = DriverManager.getConnection(url, p);
```

El driver descuida las propiedades no existentes; si el valor de la propiedad no es válido, puede obtener una excepción.

Observe que estas propiedades son específicas del driver y debe recurrir a su documentación del driver para obtener la lista de propiedades requeridas. Para saber más sobre las propiedades disponibles para Cloudscape 4.0, diríjase a http://www.cloudscape.com/docs/doc_40/doc/html/coredocs/attrib.htm.

```
public static Connection getConnection(String url, String user,
                                       String password)
        throws SQLException
```

La tercera variante toma usuario y contraseña como argumentos además del URL. Éste es un ejemplo; el siguiente código utiliza un driver ODBC, donde Movies es un DNS fijado en la configuración de ODBC. Este DNS corresponde a una base de datos que requiere un nombre de usuario y una contraseña para obtener una conexión:

```
String url = "jdbc:odbc.Movies ";
String user = "catalog admin";
String password = "catalog admin";

Connection conn = DriverManager.getConnection(url, user, password);
```

Observe que todos estos métodos están `sincronizados`, lo que supone que más de un thread de aplicación no puede conseguir el mismo objeto `java.sql.Connection`. Estos métodos lanzan una excepción `SQLException` si el driver no consigue obtener una conexión.

En ocasiones es necesario especificar el tiempo máximo que debe esperar un driver mientras intenta conectar a una base de datos. Los siguientes dos métodos pueden ser utilizados para fijar/obtener el tiempo límite de registro:

```
public static void setLoginTimeout(int seconds)
public static int getLoginTimeout()
```

El valor por defecto para el tiempo límite de registro es específico del driver/base de datos.

Métodos de registro

Los siguientes métodos acceden o establecen un objeto `PrintWriter` con propósitos de registro:

```
public static void setLogWriter(PrintWriter out)
```

```
public static PrintWriter getLogWriter()
```

Además, las aplicaciones cliente también puden registra mensajes utilizando el siguiente método:

```
public static void println(String message)
```

Este método es utilizado junto con `PrintWriter` establecido por el método `setLogWriter()` para imprimir mensajes de registro.

Driver

En JDBC, cada driver es identificado utilizando un URL de JDBC, y cada driver debe implementar la interfaz `java.sql.Driver`. Por ejemplo, en Cloudscape, la clase `COM.Cloudscape.core.JDBCDriver` implementa la interfaz `java.sql.Driver`, que especifica los siguientes métodos:

```
public boolean acceptsURL(String url)
public Connection connect(String url, Properties info)
public int getMajorVersion()
public int getMinorVersion()
public DriverPropertyInfo getPropertyInfo(String url, Properties info)
public boolean jdbcCompliant()
```

La clase `DriverManager` utiliza estos métodos. En general, las aplicaciones cliente no necesitan acceder directamente a las clases de driver.

Establecer una conexión

Para comunicar con una base de datos utilizando JDBC, debemos en primer lugar establecer una conexión con la base de datos a través del driver JDBC apropiado. El API de JDBC especifica la conexión en la interfaz `java.sql.Connection`. Esta interfaz tiene los siguientes métodos públicos:

Función	Métodos
Crear instrucciones	`createStatement()` `prepareStatement()` `prepareCall()`
Obtener información de bases de datos	`getMetaData()`

Apoyo de transacciones	`setAutoCommit()`
	`getAutoCommit()`
	`commit()` `rollback()`
	`setTransactionIsolation()`
	`getTransactionIsolation()`

Estado de conexión y cierre
Configurar diversas propiedades, limpiar valores y recuperar cualquier advertencia generada
	`isClosed()` `close()`
	`setReadOnly()`
	`isReadOnly()`
	`clearWarnings()`
	`getWarnings()`

Convertir strings SQL en SQL específico de base de datos y configurar vistas y tipos definidos por el usuario	`nativeSQL()`
	`setCatalog()`
	`getCatalog()`
	`setTypeMap()`
	`getTypeMap()`

Las categorías de la tabla anterior representan un intento de descomponer los métodos en grupos lógicos según sus funcionalidad. En las siguientes secciones, estudiaremos la mayoría de estos métodos.

Éste es un ejemplo de establecimiento de una conexión JDBC a la base de datos Cloudscape `Movies`:

```
Connection connection;
String url = "jdbc:cloudscape:Movies;create=true"

try {

   connection = DriverManager.getConnection(url);

   // Acceso a datos utilizando el objeto de conexión
   . . .
} catch (SQLException e) {
     // Tratar el error aquí
} finally {
   try {
      connection.close();
   } catch(SQLException e) { // System error? }
}
```

En este ejemplo, el URL especifica el URL de JDBC; `create=true` indica que Cloudscape debería crear la base de datos `Movies` si ésta no existe. Siempre debe capturar la excepción `SQLException` e intentar cerrar la conexión después de utilizar la conexión para acceso a los datos.

En el API de JDBC, hay varios métodos que lanzan una excepción `SQLException`. En este ejemplo, la conexión se cierra al final del bloque `finally`, de modo que los recursos del sistema pueden ser liberados independientemente del éxito o fracaso de cualquier operación de base de datos.

Resumamos ahora lo que hemos analizado hasta el momento utilizando un ejemplo sencillo. Este ejemplo registra el driver Cloudscape y establece una conexión:

```
// Importar paquetes requeridos
import java.sql.Driver;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.SQLException;
import java.sql.DriverPropertyInfo;
import java.util.Enumeration;
import java.util.Properties;

public class DriverTest {
   public static void main(String arg[]) {
      String protocol = "jdbc:cloudscape:c:/CloudscapeDB";
      String driverClass = "COM.cloudscape.core.JDBCDriver";

      // Registrar el driver Cloudscape
      try {
         Class.forName(driverClass);
      } catch (ClassNotFoundException cne) {
         cne.printStackTrace();
      }

      // Comprobar los drivers registrados
      Enumeration drivers = DriverManager.getDrivers();
      while (drivers.hasMoreElements()) {
         Driver driver = (Driver) drivers.nextElement();
         System.out.println("Registered driver: "
                            + driver.getClass().getName());
```

```
        try {

            // Acepta este driver el URL conocido
            if (driver.acceptsURL(protocol)) {
               System.out.println("Accepts URL: " + protocol);
            }
          } catch (SQLException sqle) {
            sqle.printStackTrace();
          }
        }

       // Obtener una conexión del DriverManager.
        try {
            Connection connection = DriverManager.getConnection(protocol);

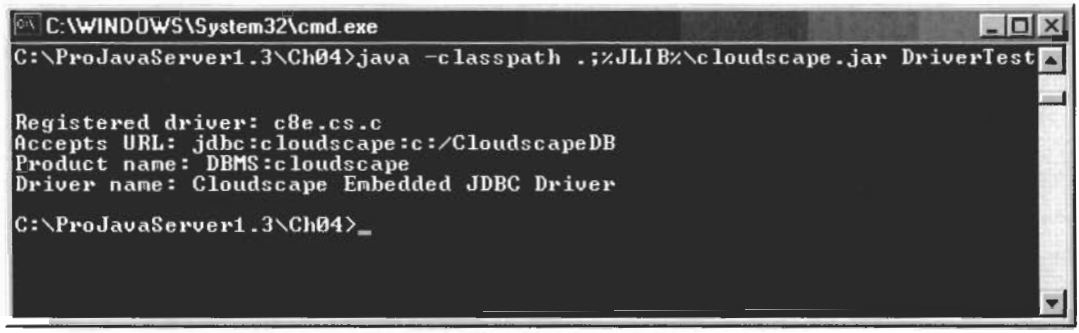
            // Obtener los metadatos
            DatabaseMetaData metaData = connection.getMetaData();
            System.out.println("Product name: "
                               + metaData.getDatabaseProductName());
             System.out.println("Driver name: " + metaData.getDriverName());
        }    catch (SQLException sqle) {
             sqle.printStackTrace();
        }
      }
  }
```

Este ejemplo ejecuta tres tareas cuyo propósito es acercarle los fundamentos de la programación JDBC:

- La primera tarea es registrar un driver. Puesto que los drivers realizan este registro vía inicialización estática, todo lo que tiene que hacer es cargar el nombre de clase para el driver específico que está utilizando. En este ejemplo, registramos el driver para Cloudscape. Observe que puede registrar más de un driver de una aplicación dada.
- La segunda tarea es encontrar cuáles son los drivers registrados. Utilizamos java.sql.DriverManager para encontrar una enumeración de los drivers registrados en ese momento. Como sólo hemos registrado un driver en el primer paso, el segundo paso encontrará sólo un driver. El segundo paso también determina si el driver encontrado acepta el URL conocido. En este caso, determinamos si el driver acepta el URL `"jdbc:Cloudscape:c:/CloudscapeDB"`, donde `"c:/CloudscapeDB"` es el nombre (y localización) de la base de datos para esta aplicación.
- El tercer paso es obtener una conexión a la base de datos utilizando el método `getConnection()` en `java.sql.DriverManager`. Después de obtener la conexión, también encontramos algunos de los metadatos de la base de datos utilizando `java.sql.DatabaseMetadata` obtenido invocando el método `getMetaData()` en `java.sql.Connection`.

Para poner a prueba este ejemplo, debe añadir las librerías Cloudscape a su classpath. La principal que requerimos para este ejemplo es `cloudscape.jar`, que encontrará en `%J2EE_HOME%\lib\cloudscape` (o `%J2EE_HOME%\lib\system`).

Después de configurar este classpath, compile y ejecute esta clase. Para Cloudscape 3.6.4, la salida de esta clase será como sigue:



Éste incluye el nombre de clase del driver, el URL aceptado, el nombre del producto y el nombre del driver. Fíjese en que el nombre del driver es algo críptico. Esto se debe a que las librerías Cloudscape son confundidas para prevenir la descompilación.

Crear y ejecutar instrucciones SQL

Podemos utilizar un objeto Connection para ejecutar instrucciones SQL creando un `Statement`, un `PreparedStatement` o un `CallableStatement`. Estos objetos abstraen instrucciones SQL normales, instrucciones preparadas y procedimientos almacenados respectivamente. Una vez hemos obtenido uno de esos objetos de instrucciones, podemos ejecutar la instrucción y leer los resultados gracias a un objeto `ResultSet`.

Como hemos mostrado en la tabla anterior, los siguientes métodos crean objetos de instrucciones:

```
Statement createStatement() throws SQLException
```

Este método crea un objeto `Statement`, que podemos utilizar para enviar instrucciones SQL a la base de datos. Las instrucciones SQL sin parámetros son ejecutadas normalmente utilizando objetos `Statement`.

```
Statement createStatement(int resultSetType, int resultSetConcurrency)
            throws SQLException
```

Esta variante de `createStatement()` requiere el Tipo `ResultSet` y argumentos `resultSetConcurrency`. Estos argumentos se aplican a objetos `ResultSet` creados ejecutando consultas.

De estos dos argumentos, el primero es utilizado para especificar el tipo `ResultSet` requerido. Como veremos más adelante, existen tres tipos de resultados dependiendo de su capacidad de desplazamiento: desplazables sólo hacia adelante (`ResultSet.TYPE_FORWARD_ONLY`), desplazables pero insensibles a los cambios realizados por otras transacciones (`ResultSet.TYPE_SCROLL_INSENSITIVE`) o desplazables y sensibles a los cambios realizados por otras transacciones (`ResultSet.TYPE_SCROLL_INSENSITIVE`). La interfaz `java.sql.ResultSet` especifica tres constantes como tipos de resultados.

El segundo argumento es utilizado para especificar si el `ResultSet` es de sólo lectura (`ResultSet.CONCUR_READ_ONLY`) o debería ser actualizable (`ResultSet.CONCUR_UPDATEABLE`).

El método de no argumento `createStatement()` devuelve un `ResultSet` que sólo es desplazable hacia adelante y de sólo lectura.

El método `createStatement()` no toma argumentos (excepto para tipo y concurrencia donde sean aplicables) y devuelve un objeto `Statement`. El objetivo final de un objeto `Statement` es ejecutar una instrucción SQL que puede o no devolver resultados. Veremos un ejemplo próximamente.

```
public PreparedStatement prepareStatement(String sql) throws SQLException
```

Podemos obtener un objeto `PreparedStatement` invocando este método en una `Connection`. Más adelante en este capítulo, haremos un recorrido por el uso de instrucciones preparadas para ejecutar instrucciones SQL.

```
public CallableStatement prepareCall (String sql) throws SQLException
```

Este método es utilizado para invocar un procedimiento almacenado.

La interfaz `Statement` tiene los siguientes métodos:

Función	Método

Ejecutar instrucciones	`execute()`
	`executeQuery()`
	`executeUpdate()`

Actualizaciones lotes	`addBatch()`
	`executeBatch()`
	`clearBatch()`

Tamaño de toma de resultset	`setFetchSize()`
	`getFetchSize()`
	`setFetchDirection()`
	`getFetchDirection()`

Obtener resultset actual	`getResultSet()`
Resultset concurrency and type	`getResultSetConcurrency()` `getResultSetType()`

Otros	`setQueryTimeout()`
	`getQueryTimeout()`
	`setMaxFieldSize()`
	`getMaxFieldSize()`
	`cancel()`
	`getConnection()`

`Statements` también se ajusta a los mismos métodos para transacciones que el objeto `Connection`, junto con el método `close()`.

Hay dos subinterfaces `PreparedStatement` y `CallableStatement`, que son utilizadas para invocar instrucciones SQL precompiladas y procedimientos almacenados de base de datos. Estas dos interfaces especifican métodos adicionales para preparar instrucciones e invocar procedimientos almacenados.

Un ejemplo: catálogo de películas

Para ilustrar el API JDBC, consideremos un simple catálogo de películas. La base de datos para este ejemplo consiste en una tabla llamada `CATALOG`.

Consideremos una clase Java `CreateMovieTable` que cree una tabla `CATALOG` e inserte los datos en la tabla. En este ejemplo, recuperamos los datos de un archivo de texto, `catalog.txt`, y los insertamos en la tabla `CATALOG`.

Crear la tabla Movie

El método `initialize()` carga el driver y obtiene una conexión. `createtable()` crea la tabla.

Puede utilizar el siguiente método en el objeto `Statement` para insertar datos:

```
int executeUpdate(String sql) throws SQLException
```

`executeUpdate()` es utilizado para ejecutar instrucciones SQL que no devuelven ningún resultado, por ejemplo instrucciones `INSERT`, `UPDATE` o `DELETE`. Este método devuelve un número entero que denota el número de las filas afectadas:

```
public class CreateMovieTables {
   static String driver = "COM.cloudscape.core.JDBCDriver";
   static String url = "jdbc:cloudscape:";
   Connection connection = null;
   Statement statement = null;
   // ...

   public void initialize() throws SQLException, ClassNotFoundException {
      Class.forName (driver);
      connection = DriverManager.getConnection(url + "Movies;create=true");
   }

   public void createTable() throws SQLException {
      statement = connection.createStatement();
      statement.executeUpdate("CREATE TABLE CATALOG" +
                              "(TITLE VARCHAR(256) PRIMARY KEY NOT NULL, " +
                              "LEAD ACTOR VARCHAR(256) NOT NULL, " +
                              "LEAD ACTRESS VARCHAR(256) NOT NULL, " +
                              "TYPE VARCHAR(20) NOT NULL, " +
                              "RELEASE DATE DATE NOT NULL)");
   }
```

En este código, las variables `connection` y `statement` son variables de instancias de la clase `CreateMovieTables`. El método `createTable()` obtiene una conexión del `DriverManager`. Para Cloudscape, especificamos el protocolo como `"jdbc:cloudscape:Movies;create=true"`.

Cuando creamos una tabla por primera vez utilizando este URL, Cloudscape DMBS crea un subdirectorio bajo el directorio actual llamado `"Movies"` para almacenar los datos. Consulte la documentación Cloudscape para especificar un directorio diferente. En caso de que desee limpiar su base de datos, puede eliminar este directorio y volver a crear todas las tablas. Puede acceder a la documentación de Cloudscape online en http://www.cloudscape.com/support/documentation.html.

En el código anterior, el `executeUpdate()` no actualiza ningún registro de la base de datos y por ello el método devuelve cero. En el caso de las instrucciones `INSERT`, `UPDATE` o `DELETE` puede que esté interesado en el número de filas insertadas, actualizadas o eliminadas, respectivamente.

Observe también que el esquema utilizado en este ejemplo es sólo con fines ilustrativos y no refleja de una forma realista los datos del catálogo.

Después de conseguir una conexión, el siguiente paso es crear una instrucción. Pasamos una instrucción SQL para crear la tabla utilizando el método `executeUpdate()` en la instrucción. Como puede ver en la instrucción anterior, la tabla `CATALOG` tiene cinco columnas.

Insertar datos

El método `insertData()` de la clase `CreateMovieTables` lee registros de película de una archivo de texto y los inserta en la base de datos:

```
public void insertData() throws SQLException, IOException {
    BufferedReader br = new BufferedReader(new FileReader("catalog.txt"));

     try {
       do {

           title = br.readLine();
           leadActor = br.readLine();
           leadActress = br.readLine();
           type = br.readLine();
           dateOfRelease = br.readLine();

           String sqlString = "INSERT INTO CATALOG " +
                              "(TITLE, LEAD ACTOR, LEAD ACTRESS, "+
                              "TYPE, RELEASE DATE) " +
                              "VALUES('" + title +
                              "','" + leadActor + "','" + leadActress +
                              "','" + type + "','" + dateOfRelease + "')";

         statement.executeUpdate(sqlString);
         statement.close();
       } while(br.readLine() != null); // This reads the termination line

     } catch (IOException e) {
        e.printStackTrace()
     } finally {
        br.close();
     }
  }
```

El formato del archivo de entrada es `titulo`, `actorPrincipal`, `actizPrincipal`, `tipo` y `fechadeEstreno` introducidos en líneas separadas, seguidos de una línea separatoria como se muestra a continuación:

```
Austin Powers
Mike Myers
Liz Hurley
Comedy
1999-04-01
-------
```

En el código anterior, la única instrucción que es relevante para nuestro análisis es el método `statement.executeUpdate()` invocado para insertar los datos en la tabla CATALOG. Observe que no capturamos aquí `SQLException`. En su lugar, estamos dejando al llamante de este método capturar la excepción SQL.

Métodos para el manejo de excepciones

Como hemos mencionado con anterioridad, la mayoría de los métodos en las interfaces y clases JDBC lanzan una instancia de `SQLException` para indicar que hay un fallo. Sin embargo, dependiendo de los drivers y las bases de datos que esté utilizando, podrá haber varias capas y fuentes de error. Para acomodar esto, la excepción SQL puede ser anidada, integrando diversas excepciones en una lista vinculada. Podemos utilizar el método `getNextException()` en la clase `SQLException` para recuperar todas esas excepciones. El siguiente fragmento de código recorre recursivamente las excepciones disponibles. Este enfoque puede resultar informativo para el manejo de excepciones en nuestras aplicaciones JDBC:

```
} catch(SQLException sqlException) {

   while(sqlException != null) {
```

```
            System.err.println(sqlException.toString());
            sqlException = sqlException.getNextException();
        }
    }
```

De un modo similar, puede recuperar advertencias (incluido las específicas del vendedor) recibidas o generadas por el driver utilizando el método `getWarnings()` en la conexión:

```
SQLWarning warnings = connection.getWarnings();

while(warnings != null) {
    System.err.println(connection.getWarnings());
    warnings = warnings.getNextWarning();
}
```

Ésta es la fuente completa:

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Date;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.EOFException;
import java.text.SimpleDateFormat;
import java.text.ParseException;

// Esta clase requiere un archivo de texto llamado catalog.txt que
   contenga
// los datos de entrada.

public class CreateMovieTables {

    static String driver = "COM.cloudscape.core.JDBCDriver";
    static String url = "jdbc:cloudscape:";

    String title, leadActor, leadActress, type, dateOfRelease;
    Connection connection;
    Statement statement;

    public void initialize() throws SQLException, ClassNotFoundException {
        Class.forName(driver);

        connection = DriverManager.getConnection(url + "Movies;create=true");
    }

    public void createTable() throws SQLException {
        statement = connection.createStatement();
        statement.executeUpdate("CREATE TABLE CATALOG" +
                                "(TITLE VARCHAR(256) PRIMARY KEY NOT NULL, " +
                                "LEAD ACTOR VARCHAR(256) NOT NULL, " +
                                "LEAD ACTRESS VARCHAR(256) NOT NULL, " +
                                "TYPE VARCHAR(20) NOT NULL, " +
                                "RELEASE DATE DATE NOT NULL)");
    }

    public void insertData() throws SQLException, IOException {
      BufferedReader br = new BufferedReader(new FileReader("catalog.txt"));
```

```
        try {
          do {

            title = br.readLine();
            leadActor = br.readLine();
            leadActress = br.readLine();
            type = br.readLine();
            dateOfRelease = br.readLine();

              String sqlString = "INSERT INTO CATALOG " +
                                 "(TITLE, LEAD ACTOR, LEAD ACTRESS, " +
                                 "TYPE, RELEASE DATE) " + "VALUES('" + title +
                                 "','" + leadActor + "','" + leadActress +
                                 "','" + type + "','" + dateOfRelease + "')";

              statement.executeUpdate(sqlString);
              statement.close();
            } while (br.readLine() != null); // This reads the termination line

          } catch (IOException e) {
            e.printStackTrace();
          }
          finally {
            br.close();
          }
        }

        public void close() throws SQLException {
          try {
            connection.close();
          } catch (SQLException e) {
            throw e;
          }
        }

        public static void main(String arg[]) {
          CreateMovieTables movies = new CreateMovieTables();

          try {
            movies.initialize();
            movies.createTable();
            movies.close();
          } catch (SQLException sqlException) {
            while (sqlException != null) {
              sqlException.printStackTrace();
              sqlException = sqlException.getNextException();
            }
          } catch (Exception e) {
            e.printStackTrace();
          }
        }
      }
```

Con objetivos de demostración, esta clase está estructurada en términos de pequeños métodos, cada uno de los cuales desempeña funciones JDBC específicas. En esta clase, el método `initialize()` carga el driver y crea una conexión. El método `createTable()` crea una tabla utilizando el método `executeUpdate()` en la instrucción `java.sql.Statement`. El método `insertData()` lee una serie de registros de un archivo de texto y los inserta en la tabla. El método `main()` de esta clase es el controlador que invoca los otros métodos en secuencia. Observe que el método `main()` invoca el método `close()` para cerrar la instrucción y la conexión.

Fíjese que de los campos insertados en el método anterior, aunque el campo RELEASE_DATE es de tipo DATE, utilizamos un string al crear la instrucción. Esto es válido siempre que el String utilizado pueda ser forzado de forma segura en un campo DATE. En este ejemplo, el string utilizado es de la forma YYYY-MM-DD, que es traspasable a un campo DATE. Sin embargo, para que sean seguros, debe considerar utilizar instrucciones preparadas para ejecutar SQL que conlleve datos introducidos como fechas, estampillas de tiempo, números, etc.

Para la mayor parte de los ejemplos restantes de este capítulo, seguiremos la clase anterior como plantilla. Si desea experimentar con el API JDBC, debe añadir nuevos métodos a esta clase.

Consultar la base de datos

El objeto `Statement` devuelve un objeto `java.sql.ResultSet` que encapsula los resultados de ejecución. Ésta es una interfaz que es implementada por vendedores de drivers. Puede desplazar los bloques de resultados utilizando un cursor para leer los resultados en `ResultSet`.

El siguiente método, `executeQuery()`, en la interfaz `java.sql.Statement` le permite ejecutar las instrucciones SQL `SELECT`:

```
public ResultSet executeQuery (String sql) throws SQLException
```

Existe también un método genérico `execute()` que puede devolver múltiples resultados:

```
public boolean execute(String sql) throws SQLException
```

Este método execute() puede utilizarse para ejecutar procedimientos almacenados conocidos para ofrecer múltiples resultados o strings SQL desconocidas (por ejemplo, instrucciones SQL leídas en otras fuente en el tiempo de ejecución). Este método devuelve un Boolean con valor true cuando la ejecución tiene como resultado uno o más bloques de resultados, o un valor false cuando la ejecución tiene como resultado una o más cuentas de actualización. Utilizando este valor de retorno, la aplicación puede invocar los métodos getResult() o getUpdate() para acceder al bloque de resultados o actualizar la cuenta, respectivamente.

El API JDBC 2.1 introduce dos tipos más de ResultSet que permiten desplazar los resultados hacia adelante y hacia atrás pero no todos los vendedores de bases de datos se ajustan en la actualidad a esta característica. Analizaremos el desplazamiento de resultados posteriormente en este capítulo.

Métodos para recuperar datos

La interfaz `java.sql.ResultSet` ofrece varios métodos para recuperar campos de diferentes tipos. Dependiendo de su esquema, deberá utilizar los métodos apropiados para recuperar los datos:

`getArray()`	`getAsciiStream()`	`getBigDecimal()`	`getBinaryStream()`
`getBlob()`	`getBoolean()`	`getByte()`	`getBytes()`
`getCharacter Stream()`	`getClob()`	`getDate()`	`getDouble()`
`GetFloat()`	`getInt()`	`getLong()`	`getObject()`
`getRef()`	`getShort()`	`getString()`	`getTime()`
`getTimestamp()`	`getURL()`		

Todos estos métodos requieren el nombre de la columna (como un string) o el índice de la columna como argumento. La sintaxis para las dos variantes de los métodos `getString()` son las siguientes:

```
public String getString(int columnIndex) throws SQLException
public String getString(String columnName) throws SQLException
```

En casos en los que el índice de la columna está sujeto al cambio debido a cambios en SQL o en el esquema de la base de datos es preferible utilizar nombres de columna en estos métodos.

Los tipos devueltos de estos métodos varían desde simples primitivas Java como `int`, `double`, `byte`, etc., hasta tipos SQL de propósito específico como `java.sql.Blob` y `java.sql.Clob`. Los métodos `getAsciiStream()`, `getBinaryStream()` y `getCharacterStream()` permiten el acceso a datos ASCII/caracteres arbitrariamente largos devolviendo objetos `java.io.InputStream` y `java.io.Reader`.

Volvamos a nuestro catálogo de películas y creemos otra clase, `QueryMovieTable`, para implementar diferentes tipos de consultas. El siguiente método, `queryAll()`, recupera todos los datos de la tabla CATALOG:

```
public void queryAll() throws SQLException {
   System.out.println("Query All");
   Statement statement = connection.createStatement();

   String sqlString = "SELECT CATALOG.TITLE, CATALOG.LEAD ACTOR, "+
                      "CATALOG.LEAD ACTRESS, CATALOG.TYPE, " +
                      "CATALOG.RELEASE DATE FROM CATALOG";

   ResultSet rs = statement.executeQuery(sqlString);

   while(rs.next()) {
      System.out.println(rs.getString("TITLE") + ", " +
                         rs.getString("LEAD ACTOR") + ", " +
                         rs.getString("LEAD ACTRESS") + ", " +
                         rs.getString("TYPE") + ", " +
                         rs.getDate("RELEASE DATE"));
   }
}
```

Este método crea primero un objeto `Statement`, que utiliza entonces para invocar `executeQuery()` con una instrucción SELECT SQL como argumento. El objeto `java.sql.ResultSet` contiene todas las filas de la tabla CATALOG que coinciden con la instrucción SELECT. Utilizando el método `next()` del objeto `ResultSet`, podemos repetir todas las filas contenidas en el bloque de resultados. En cualquier fila, podemos utilizar uno de los métodos `getXXX` de la tabla anterior para recuperar los campos de una fila.

¿Qué sucede si la consulta (o cualquier SQL enviado a la base de datos a través del driver para su ejecución) es muy cara y tarda mucho tiempo en completarse? Esto puede ocurrir, por ejemplo, si la consulta es compleja o si la base de datos está intentando devolver un gran número de resultados. Para controlar el tiempo que espera el driver para que la base de datos complete su ejecución, la interfaz `java.sql.Statement` tiene dos métodos para obtener o fijar un límite máximo de tiempo.

Por ejemplo, podemos modificar el método `queryAll()` para establecer un intervalo de tiempo máximo para la consulta (en segundos) utilizando el método `setQueryTimeout()`. Puede utilizar el método `getQueryTimeout()` para conocer el intervalo de tiempo de la consulta en segundos (o el valor por defecto, si no se ha fijado explícitamente). Una vez que la base de datos excede el intervalo máximo de tiempo, el driver aborta la ejecución y lanza una excepción `java.sqlSQLException`. Por ejemplo, aquí fijamos el valor del plazo límite de tiempo en 1 segundo:

```
Statement statement = connection.createStatement();
statement.setQueryTimeout(1);
// Establecer SQL para la instrucción
ResultSet rs = statement.executeQuery(sqlString);
```

Interfaz ResultSetMetaData

La interfaz ResultSet también nos permite conocer la estructura del bloque de resultados. El método `getMetaData()` nos ayuda a recuperar un objeto `java.sql.ResultSetMetaData` que tiene varios métodos para describir los cursores del bloque de resultados:

`GetCatalogName()`	`getScale()`
`getTableName()`	`getPrecision()`
`GetSchemaName()`	`isNullable()`
`GetColumnCount()`	`isCurrency()`
`GetColumnName())`	`isSearchable()`
`GetColumnLabel()`	`isCaseSensitive()`
`GetColumnType()`	`isSigned()`
`GetColumnTypeName()`	`isAutoIncrement()`
`GetColumnClassName()`	`isReadOnly()`
`GetColumnDisplaySize()`	`isDefinitelyWritable()`
`isWritable()`	

Tomando un bloque de resultados, podemos utilizar el método `getColumnCount()` para obtener el número de columnas del bloque de resultados. Utilizando este número de columnas, podemos obtener la meta-información de cada columna.

Por ejemplo, el siguiente método de nuestro ejemplo imprime la estructura del bloque de resultados:

```
public void getMetaData() throws SQLException {
     System.out.println("MetaData of ResultSet");
     Statement statement = connection.createStatement();

     String sqlString = "SELECT * FROM CATALOG";

     ResultSet rs = statement.executeQuery(sqlString);

     ResultSetMetaData metaData = rs.getMetaData();

     int noColumns = metaData.getColumnCount();

     // Número de columna empezando desde el 1
     for(int i = 1; i < noColumns + 1; i++) {
         System.out.println(metaData.getColumnName(i) + " " +
                            metaData.getColumnType(i));
     }
  }
```

El método anterior obtiene el número de columnas del bloque de resultados e imprime el nombre y el tipo de cada columna. En este caso, los nombres de columna son `TITLE`, `LEAD_ACTOR`, `LEAD_ACTRESS`, `TYPE` y `RELEASE_DATE`. Observe que los tipos de columna son devueltos como números enteros. Por ejemplo, todas las columnas tipo `VARCHAR` tendrán el tipo de columna 12, mientras que el tipo `DATE` es

tipo 91. Estos tipos son constantes definidas en la interfaz `java.sql.Types`. Fíjese también en que los números de las columnas empiezan desde `1` y no desde `0`.

Esta información puede utilizarse para decidir cómo tratar este tipo de datos. Por ejemplo, conociendo que la quinta columna es tipo `DATE` (fecha), la aplicación puede determinar que un `java.sql.Date` debería utilizarse para representar este dato. Aunque esta información puede derivarse del estudio del esquema de la base de datos y del SQL utilizado para extraer los datos, los metadatos de la base de datos pueden utilizarse para reducir la codificación de dicha información.

Instrucciones preparadas

Las instrucciones preparadas JDBC cumplen los siguientes requisitos:

- Crear instrucciones parametrizadas de modo que los datos para parámetros puedan ser sustituidos dinámicamente
- Crear instrucciones que impliquen valores de datos que no siempre pueden ser representados como strings de caracteres
- Precompilar instrucciones SQL para evitar la compilación repetida de las mismas instrucciones SQL

Analicemos brevemente las razones de la importancia de estos requisitos.

En la mayoría de los casos, puede que no tenga la información completa para construir una cláusula `DÓNDE` en SQL. Por ejemplo, para escribir una instrucción SQL `SELECT` para seleccionar los datos de un usuario, mientras escribe el código JDBC, necesita conocer el valor clave primario (como el `USER_ID`) para construir el SQL. En la mayor parte de los casos, esta información está disponible sólo en el periodo de ejecución (digamos, desde una interfaz de usuario o desde cualquier otra aplicación). Las instrucciones preparadas se enfrentan este problema proporcionando parámetros (expresados como signos de interrogación) en SQL.

En lugar de utilizar valore, puede utilizar calificadores `"?"` en SQL. Por ello, en vez de crear una instrucción con el string SQL:

```
SELECT <select_fields> FROM <table_name> WHERE USER_ID = <value>
```

puede utilizar una instrucción preparada con el string SQL:

```
SELECT <select_fields> FROM <table_name> WHERE USER_ID = ?
```

Puede sustituir valores actuales utilizando métodos en la interfaz `java.sql.PreparedStatement`.

Lo mismo se aplica a SQL cuando conlleva tipos de datos complejos como datos de texto largo, datos binarios o incluso datos de estampillas de tiempo. Tales tipos de datos (durante inserciones/actualizaciones o en cláusulas `DÓNDE` de SQL) no pueden ser expresados como sencillos strings. Por ejemplo, ¿cómo crearíamos una instrucción SQL para actualizar datos de imagen? Debido a que las instrucciones preparadas están parametrizadas, en lugar de expresar esos tipos de datos en la instrucción SQL directamente, puede configurar los datos utilizando diversos métodos en la interfaz `java.sql.PreparedStatement`.

Además, las mismas instrucciones SQL pueden ser ejecutadas varias veces con diferentes parámetros y la base de datos puede compilar estas instrucciones sólo una vez, mejorando así el rendimiento. Un objeto `PreparedStatement` puede albergar instrucciones precompiladas. Los siguientes métodos de la interfaz `java.sql.Connection` nos permiten crear objetos `PreparedStatement`:

```
PreparedStatement prepareStatement(String sql) throws SQLException
PreparedStatement prepareStatement(String sql, int resultSetType,
                                   int resultSetConcurrency)
                                   throws SQLException
```

Por ejemplo, en el método `insertData()` de la clase `CreateMovieTables`, la instrucción `INSERT` SQL es ejecutada varias veces. Esta instrucción debe ser compilada antes de ser ejecutada. La compilación es un proceso que requiere tiempo y que conlleva normalmente el análisis de la instrucción, asociando con las tablas y las columnas cualquier optimización y generación de código, por lo que puede que sea más conveniente utilizar una instrucción preparada como alternativa. Con instrucciones preparadas, esta compilación sólo se realiza una vez.

Para permitirnos especificar los datos, las instrucciones están parametrizadas, con cada parámetro representado como un signo de interrogación (`?`).

Podemos reemplazar el método `insertData()` de la clase `CreateMovieTables` con lo siguiente para hacer uso del objeto `PreparedStatement`:

```
public void insertPreparedData () throws SQLException, IOException {

   BufferedReader br = new BufferedReader(new FileReader("catalog.txt"));

   PreparedStatement preparedStatement =
        connection.prepareStatement("INSERT INTO" CATALOG" +
                                    "(title, lead actor, lead acctress, " +
                                    "type, release date)" +
                                    "VALUES(?, ?, ?, ?, ?)");

   SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

   try {
     do {
        preparedStatement.clearParameters();
        title = br.readLine();
        leadActor = br.readLine();
        leadActress = br.readLine();
        type = br.readLine();
        dateOfRelease = br.readLine();
        Date date = new Date(dateFormat.parse(dateOfRelease).getTime());

        preparedStatement.setString(1, title);
        preparedStatement.setString(2, leadActor);
        preparedStatement.setString(3, leadActress);
        preparedStatement.setString(4, type);
        preparedStatement.setDate(5, date);

        preparedStatement.executeUpdate();

     } while(br.readLine() != null);
   } catch (EOFException e) {
   } catch (java.text.ParseException pe) {
     pe.printStackTrace();
   } catch (IOException ioException) {
     ioException.printStackTrace();
   } finally {
       preparedStatement.close();
       br.close();
   }
}
```

Una vez se ha creado `PreparedStatement`, utilice los métodos `setXXX` para fijar los parámetros para `PreparedStatement`. Por ejemplo, podemos utilizar el método `setString()` para establecer los

parámetros VARCHAR. De un modo similar, para establecer el tipo DATE SQL, podemos utilizar el método `setDate()`. Estos métodos ocupan el lugar del número como primer argumento; entonces, `setString(1, tittle)` fija el valor del primer ? en `title`.

Observe también que el objeto fecha requiere especial tratamiento. Hay cuatro objetos diferentes que podemos utilizar para representar fecha y tiempo. La clase `java.util.Date` era el único objeto disponible anterior al API JDBC. Sin embargo, el paquete `java.sql` introduce tres clases más:

```
public class Date extends java.util.Date
```

Esta clase `Date` corresponde al tipo DATE SQL. Por lo tanto, debemos utilizar `java.sql.Date` en nuestra programación JDBC para ajustarnos a los tipos JDBC:

```
public class Time extends java.util.Date
```

Esta clase corresponde al tipo TIME DATE SQL:

```
public class Timestamp extends java.util.Date
```

Ninguna de las clases `java.sql.Date` y `java.sql.Time` captura la precisión del tipo TIMESTAMP SQL. La clase `java.sql.TimeStamp` cubre estas deficiencias. Los objetos de estampilla de tiempo (timestamp) pueden albergar nanosegundos de TIMESTAMP, además de los campos estándar de tiempo y fecha. En general, si está capturando instancias específicas de tiempo, considere utilizar `java.sql.TimeStamp`.

El API JDBC también ofrece facilidades para invocar procedimientos almacenados. La interfaz `java.sql.CallableStatement` puede utilizarse para crear instrucciones para ejecutar procedimientos almacenados. Los siguientes métodos de la interfaz `java.sql.Connection` crean objetos `CallableStatement`.

```
CallableStatement prepareCall(String sql) throws SQLException
CallableStatement prepareCall(String sql, int resultSetType,
                              int resultSetConcurrency)
                        throws SQLException
```

En la práctica, un procedimiento almacenado no es apropiado para un entorno J2EE, ya que asocia muy de cerca la aplicación con una base de datos específica.

Representar tipos SQL en Java

Los tipos de datos SQL y los tipos de datos Java no son equivalentes. Mientras que los tipos SQL incluyen tipos tales como VARCHAR, NUMBER y TIMESTAMP, y los tipos de datos en Java han evolucionado a partir de lenguajes de programación como C y C++. Esta disparidad entre tipos SQL y tipos de datos Java ha sido tratada con la especificación de tipos SQL genéricos y con su representación hacia y desde tipos de datos Java. Los tipos SQL JDBC están especificados en la clase `java.sql.Types`, como constantes.

Existen dos ocasiones en las que requerimos esta representación de información: al establecer los parámetros de entrada para instrucciones preparadas (utilizando métodos `setXXX()` en la interfaz `PreparedStatement`) y al obtener resultados de los objetos `ResultSet` (utilizando métodos `getXXX()` en la interfaz `ResultSet`). En ambos casos, los drivers representan los tipos en JDBC y los tipos Java.

La siguiente tabla resume los tipos JDBC y su representación en tipos Java, y los tipos de datos SQL:

TipoJDBC	Propósito	Tipo SQL	Tipo Java
ARRAY	Representa el ARRAY tipo SQL	ARRAY	java.sql .Array
BIGINT	Número entero signado de 64 bits	BIGINT	long
BINARY	Valor binario, pequeño de longitud fija	No hay correspondencia Consulte su documentación de driver	byte[]
BIT	Valor de un sólo bit (0 ó 1)	BIT	boolean
BLOB	Representa BLOB tipo SQL para almacenar grandes objetos binarios	BLOB	java.sql .Blob
BOOLEAN	RepresentaBOOLEAN tipo SQL	BOOLEAN	java.lang .Boolean
CHAR	String de caracteres pequeño, de longitud fija	CHAR	String
CLOB	Representa CLOB tipo SQL para almacenar objetos grandes de caracteres	CLOB	java.sql .Clob
DATALINK	Representa DATALINK tipo SQL	DATALINK	java.net.URL
DATE	Fecha consistente en día, mes y año	DATE	java.sql .Date
DECIMAL	Valores decimales de precisión fija	DECIMAL	java.math .BigDecimal
DISTINCT	Para representación adaptada de tipos definidos por usuario	DISTINCT	User defined
DOUBLE	Números de coma flotante de doble precisión con mantisa de 15 dígitos	DOUBLE PRECISION	double
FLOTA	Números de coma flotante de doble precisión con mantisa de 15 dígitos	FLOAT	double
INTEGER	Número entero signado de 32 bits	INTEGER	int
JAVA_OBJECT	Para almacenar objetos Java	No hay correspondencia	Object
LONGVARBINARY	Valor binario, grande de longitud variable	No hay correspondencia	byte[]
LONGVARCHAR	String grande de caracteres de longitud variable	No hay correspondencia	String

TipoJDBC	Propósito	Tipo SQL	Tipo Java
NULL	Para representar valores NULL	NULL	null for Java objects, 0 for numeric primitives, and false for Boolean
NUMERIC	Valores decimales de precisión fija	NUMERIC	java.math .BigDecimal
OTHER	Para almacenar/recuperar tipos específicos de base de datos	No hay correspondencia; destinados a tipos específicos de base de datos/driver	Object
REAL	Número de coma flotante de precisón única con mantisa de 7 dígitos	REAL	float
REF	NA	NA	NA
SMALLINT	Número entero signado de 16 bits	SMALLINT	short
STRUCT	NA	NA	NA
TIME	Tiempo consistente en horas, minutos y segundos	TIME	java.sql .Time
TIMESTAMP	Estampilla de tiempo consistente en FECHA, TIEMPO y un campo de nanosegundos	TIMESTAMP	java.sql .TimeStamp
TINYINT	Número entero no signado de 8 bit	TINYINT	short
VARBINARY	Valor binario pequeño de longitud variable	No hay correspondencia	byte[]
VARCHAR	String de caracteres pequeño de longitud variable	VARCHAR	String

Aunque esta tabla resume todas las posibles representaciones entre SQL y Java, en la práctica el apoyo a estas representaciones varía de driver a driver. Además, puesto que no todos los vendedores cumplen todos los tipos SQL y también hay varios tipos SQL específicos del vendedor de actualidad, debe comprobar su documentación de base de datos y driver para ajustarse a esta representación. Si la portabilidad de su aplicación es un problema, debe intentar que su SQL se adapte a los tipos de datos más utilizados.

Apoyo a transacciones

En algunas aplicaciones, puede que queramos agrupar una serie de instrucciones que necesiten, todas ellas, ser efectivas o fallar. En tales casos, un grupo de operaciones SQL constituye una unidad de trabajo. La noción de la transacción es importante para preservar la integridad de las transacciones empresariales que producen múltiples operaciones SQL. Cuando múltiples instrucciones son ejecutadas en una única

transacción, todas las operaciones pueden ser realizadas (convertidas en permanentes en la base datos) o deshechas (es decir, se deshacen los cambios aplicados a la base de datos).

El análisis de esta sección se limita a transacciones locales, es decir, transacciones realizadas con una única base de datos desde un único cliente. Estudiaremos con detenimiento las transacciones distribuidas más adelante, en este capítulo.

Cuando se crea un objeto `Connection`, éste está configurado para realizar automáticamente cada transacción. Esto significa que cada vez que se ejecuta una instrucción, se realiza en la base de datos y no puede ser deshecha. Los siguientes métodos en la interfaz `Connection` son utilizados para demarcar transacciones y bien deshacerlas o realizarlas en la base de datos:

```
void setAutoCommit(boolean autoCommit) throws SQLException
void commit() throws SQLException
void rollback() throws SQLException
```

Para iniciar una transacción, invocamos `setAutoCommit(false)`. Esto nos otorga el control sobre lo que se realiza y cuándo se realiza. Una llamada al método `commit()` realizará todas las instrucciones emitidas desde la última vez que se invocó el método `commit()`. Por el contrario, una llamada `rollback()` deshará todos los cambios desde la última `commit()`. Sin embargo, una vez se ha emitido una instrucción commit, esas transacciones no puede deshacerse con `rollback`.

Consideremos un caso práctico de empresa consistente en cuatro pasos, cada uno de los cuales conlleva ejecutar una instrucción SQL y lógica adicional implementada en la aplicación: crear un pedido, actualizar el inventario y crear un registro de envíos. La lógica de empresa puede requerir que todo sea efectivo o que todo falle.

El fracaso de la creación de un registro de envíos puede dictar que no cree un pedido. En tales casos, los efectos de las instrucciones SQL correspondientes a las dos primeras tareas (crear un pedido y actualizar el inventario) deben deshacerse.

El siguiente fragmento de código ilustra esta situación:

```
Connection connection = null;

// Obtener una conexión
...
try {
   // Iniciar una transacción
   connection.setAutoCommit(false);

   Statement statement = connection.createStatement();

   // Crear un pedido
   statement.executeUpdate("INSET INTO ORDERS(ORDER ID, PRODUCT ID,...) "+
                           "VALUES(...)");

   // Actualizar inventario
   statement.executeUpdate("UPDATE TABLE INVENTORY " +
                           "SET QUANTITY = QUANTITY - 1 " +
                           "WHERE PRODUCT ID = ...");

   // Crear un registro de envíos
   if(...) {
      // Business logic succeeded
      statement.execute("INSERT INTO SHIP RECORD(...) VALUES (...)");
      connection.commit();
    } else {
      // Fallo de lógica empresarial. No puede proceder con este pedido
      connection.rollback();
```

```
    }
}   catch (SQLException e)   {
    // Manejar excepciones aquí
}   finally {
    // Cerrar instrucción y conexión
}
```

En este fragmento, una vez invocado el método `rollback()`, la base de datos restaura las tablas `PEDIDOS` e `INVENTARIO` a su estado anterior. `commit()` convierte en permanentes los cambios efectuados por las instrucciones `INSERT` y `UPDATE`.

Analizaremos las transacciones con más detalle más adelante en este capítulo.

Puntos de salvaguardia

El API JDBC 3.0 introduce puntos de salvaguardia. Un punto de salvaguardia es similar a un marcador temporal fijado en una posición específica durante una serie de operaciones de bases de datos en una transacción. Un punto de salvaguardia le permite realizar o deshacer transacciones de forma parcial. Para comprender la noción de punto de salvaguardia, considere utilizar un procesador de textos para editar un archivo. Durante el proceso de edición, puede guardar el documento varias veces. Cada operación de guardar es como establecer un nuevo punto. Después de guardar puede volver a la versión guardada anteriormente negando simplemente todas las ediciones y reabriendo el mismo archivo. De un modo similar, el mecanismo `java.sql.Savepoint` le permite realizar/abortar operaciones en varios puntos de salvaguardia.

Considere la siguiente secuencia de operaciones de base de datos:

```
Connection connection = null;

// Obtener una conexión
...
try {
   // Iniciar una transacción
   connection.setAutoCommit(false);

   // Ejecutar el primer grupo de operaciones de base de datos
   ...

   // Ejecutar el segundo grupo de operaciones de base de datos
   ...
   // Ejecutar el tercer grupo de operaciones de base de datos
   ...
}   catch (SQLException e)   {
   // Manejar excepciones aquí
}   finally {
   // Cerrar instrucción y conexión
}
```

Supongamos que los tres grupos de operaciones están lógicamente relacionadas. Por ejemplo, si hubiera un fallo durante la ejecución del segundo grupo de operaciones, no habría modo alguno de deshacer sólo el segundo grupo y ejecutar otro grupo distinto de operaciones correctivas. Los métodos `commit()` y `rollback()` se aplican a los tres grupos de operaciones en su conjunto. Puede superar esta limitación con los puntos de salvaguardia.

Considere cómo puede implementarse la situación anterior utilizando puntos de salvaguardia:

```
Connection connection = null;
```

```
// Obtener una conexión
...
try {
   // Iniciar una transacción
   connection.setAutoCommit(false);

   // Ejecutar el primer grupo de operaciones de base de datos
   ...

   Savepoint sp1 = connection.setSavepoint("First Batch");
   // Ejecutar el segundo grupo de operaciones de base de datos
   ...

   // Si es necesario deshacer el segundo grupo de operaciones de base de
datos
   if(...) {
       connection.rollback(sp1);
       // Ejecutar operaciones correctivas si procede
   } else {
       connection.releaseSavepoint(sp1);
   }

   // Ejecutar el segundo grupo de operaciones de base de datos
   ...
} catch (SQLException e)  {
   // Handle exceptions here
} finally {
   // Cerrar instrucción y conexión
}
```

En este caso, después de ejecutar el primer lote de operaciones fijam.os un punto de salvaguardia con un determinado nombre (`sp1`). Observe que el nombre es opcional y es utilizado por los drivers para identificar un punto de salvaguardia. Cuando no se designa un nombre, los drivers utilizan un ID generado internamente en lugar del nombre. Después de fijar este punto de salvaguardia, posteriormente, podemos liberar el punto de salvaguardia, o deshacer todas las operaciones ejecutadas después de establecer el punto de salvaguardia. En el código anterior, la llamada `rollback (sp1)` deshace el segundo grupo de operaciones de base de datos sin afectar al primer grupo de operaciones.

Sin embargo, una vez ha liberado un punto de salvaguardia, cualquier llamada `rollback()` posterior deshace todas las operaciones incluido las ejecutadas antes de establecer el punto de salvaguardia.

Puede fijar más de un punto de salvaguardia durante una determinada transacción. Sin embargo, cualquier acción de deshacer ejecutada en un punto de salvaguardia invalida todos los puntos de salvaguardia fijados después. Por ejemplo, si establece tres puntos de salvaguardia `sp1`, `sp2` y `sp3` (fijados en el mismo orden) e invoca `rollback (sp2)`, el tercer punto de salvaguardia, `sp3`, queda invalidado.

Una vez establecido un punto de salvaguardia, puede liberarlo más tarde utilizando el método `releaseSavepoint()` en el objeto `java.sql.Connection` utilizado para fijar el punto de salvaguardia. Una vez liberado el punto de salvaguardia, puede utilizarlo para deshacer. Sin embargo, no puede cambiar la ubicación de un punto de salvaguardia una vez éste es establecido durante una transacción.

Fíjese en que la característica de punto de salvaguardia es relativamente nueva en el API JDBC y, por ello, en el momento de escribir este capítulo, no todos los drivers JDBC se ajustan a esta característica. Puede determinar si su base de datos/driver cumple este requisito, utilizando el siguiente método en `java.sql.DatabaseMetaData`:

```
boolean supportsSavepoints() throws SQLException
```

Aunque el punto de salvaguardia es nueva en el API JDBC, bases de datos como Microsoft SQL Server y Oracle se ajustan a esta característica.

Listas de resultados desplazables y actualizables

Antes de JDBC 2.1, un objeto `ResultSet` (lista de resultados) creado al ejecutar una instrucción era, por defecto, desplazable únicamente hacia adelante. Es decir, sólo podíamos recorrer la lista de resultados utilizando el método `next()`. `next()` devuelve false cuando se llega al último registro y no se pueden recuperar más detalles.

El API JDBC 2.1 (incluido en J2EE 1.2 y versiones posteriores) ofrece medios más flexibles para acceder a los resultados desde objetos `ResultSet`. Estos avances están clasificados del siguiente modo:

- **Listas de resultados desplazables**
  Los objetos ResultSet JDBC 2.1 son desplazables. Las listas de resultados desplazables tienen la capacidad de mover el cursor hacia detrás y también permiten una libertad absoluta de ubicación del cursor en una fila determinada de la lista de resultados.
- **Sensibilidad de desplazamiento**
  El API JDBC 2.1 también especifica listas de resultados sensibles al desplazamiento e insensibles al desplazamiento. Una lista de resultados insensible al desplazamiento representa una instantánea estática de los resultados al realizar la consulta. Por otro lado, una lista de resultados sensible al desplazamiento es sensible a los cambios efectuados a los datos después de que la consulta haya sido ejecutada, proporcionando así una visión dinámica de los datos a medida que éstos cambian.
- **Listas de resultados actualizables**
  Por defecto, las listas de resultados son de sólo lectura. Es decir, los contenidos de la lista de resultados sólo pueden ser leídos y no pueden ser alterados. El API JDBC 2.1 introduce las listas de resultados actualizables. Cuando una lista de resultados es actualizada, la operación de actualización también actualiza los datos originales correspondientes a la lista de resultados.

java.sql.ResultSet cuenta con métodos adicionales que cumplen estas características, que serán analizadas en los siguientes subapartados.

Listas de resultados desplazables

La interfaz `java.sql.ResultSet` especifica tres tipos de listas de resultados:

- `TYPE_FORWARD_ONLY`. Una lista de resultados de este tipo sólo se adapta al desplazamiento hacia delante.
- `TYPE_SCROLL_INSENSITIVE`. Una lista de resultados de este tipo se desplaza en ambas direcciones.
- `TYPE_SCROLL_SENSITIVE`. Una lista de resultados de este tipo es sensible a actualizaciones realizadas a los datos después de poblar la lista de resultados. Por ejemplo, si su consulta devuelve 10 filas y si otra aplicación elimina dos de las filas, su lista de resultados sólo tendrá 8 filas. Lo mismo ocurre para inserciones/actualizaciones.

Antes de utilizar los métodos de la interfaz `java.sql.ResultSet` para desplazarse por los resultados, debemos asegurarnos que el driver JDBC se ajusta a estas características. La interfaz `java.sql.DatabaseMetadata` proporciona varios métodos para descubrir la capacidad del driver de base de datos.

Para saber qué tipos de resultados se adaptan utilice `supportsResultSetType()`. Por ejemplo, el siguiente fragmento de código debe informarnos sobre el apoyo del driver a estos tipos:

```
public void testScrollable() throws SQLException {
   boolean supports;

    DatabaseMetaData md = connection.getMetaData();

    supports = md.supportsResultSetType(ResultSet.TYPE FORWARD ONLY);

    if(supports) {
        System.out.println("TYPE FORWARD ONLY - Supports");
    }   else {
        System.out.println("TYPE FORWARD ONLY - Does not support");
    }

    supports = md.supportsResultSetType(ResultSet.TYPE SCROLL INSENSITIVE);

    if(supports) {
        System.out.println("TYPE SCROLL INSENSITIVE - Supports");
    }   else {
        System.out.println( "TYPE SCROLL INSENSITIVE - Does not support");
    }

    supports = md.supportsResultSetType(ResultSet.TYPE SCROLL SENSITIVE);

    if(supports) {
        System.out.println("TYPE SCROLL SENSITIVE - Supports");
    }   else {
        System.out.println("TYPE SCROLL SENSITIVE - Does not support");
    }
}
```

Para listas de resultados sensibles al desplazamiento, los siguiente métodos de `java.sql.DatabaseMetadata` devuelven información sobre el nivel de sensibilidad:

```
public boolean othersUpdatesAreVisible(int type) throws SQLException
```

Este método devuelve `true` si los cambios efectuados por otras transacciones son visibles para una lista de resultados sensibles al desplazamiento.

```
public boolean othersDeletesAreVisible(int type) throws SQLException
```

`othersDeleteAreVisible()` devuelve `true` si las eliminaciones realizadas por otras transacciones son visibles en una lista de resultados sensible al desplazamiento.

```
public boolean othersInsertsAreVisible(int type) throws SQLException
```

`othersInsertAreVisible()` devuelve `true` si las inserciones efectuadas por otras transacciones son visibles en una lista de resultados sensible al desplazamiento.

De un modo similar, la interfaz `java.sql.DatabaseMetadata` especifica los métodos `ownUpdateAreVisible()`, `ownDeleteAreVisible()` y `ownInsertsAreVisible()` para saber si los cambios efectuados a los resultados son visibles en esta lista de resultados.

La interfaz `java.sql.ResultSet` se ajusta a los siguientes métodos para desplazar las listas de resultados. Estos métodos están agrupados según su uso.

Métodos relacionados con la posición del cursor

```
public boolean isBeforeFirst() throws SQLException
public boolean isAfterLast() throws SQLException
```

`isBeforeFirst()` devuelve `true` si la posición del cursor es anterior a la primera fila de la lista de resultados. `isAfterLast()` devuelve `true` si la posición del cursor es posterior a la última fila de la lista de resultados.

```
public boolean isFirst() throws SQLException
public boolean isLast() throws SQLException
```

`isFirst()` devuelve `true` si el cursor está situado en la primera fila de la lista de resultados. `isLast()` devuelve `true` si el cursor está en la última fila de la lista de resultados.

```
public void beforeFirst() throws SQLException
public void afterLast() throws SQLException
```

Estos dos métodos devuelven `true` si la posición del cursor es anterior a la primera fila o posterior a la última fila de la lista de resultados respectivamente.

Métodos para desplazamiento

```
public boolean first() throws SQLException
public boolean last() throws SQLException
```

Estos métodos desplazan el cursor hasta la primera y última fila de la lista de resultados respectivamente.

```
public boolean absolute(int row) throws java.sql.SQLException
```

`absolute()` desplaza el cursor hasta la fila especificada en la lista de resultados. El argumento de fila puede ser positivo o negativo. Un argumento negativo desplaza el cursor hacia atrás, mientras que un argumento positivo desplaza el cursor hacia delante.

```
public boolean relative(int rows) throws SQLException
```

Este método desplaza el cursor el número de filas especificado relativo a la posición actual del cursor. Si el argumento es positivo, este método desplaza el cursor hacia delante el número de filas especificado. Si es negativo, este método desplaza el cursor hacia atrás el número de filas especificado. Si el número especificado (positivo o negativo) es mayor que las filas disponibles (hacia delante o hacia atrás), este método sitúa el cursor en la primera o en la última fila de la lista de resultados. Este método no debería ser invocado cuando la posición del cursor no es válida. Por ejemplo, después de invocar `next()` recursivamente hasta que `next()` devuelva false (lo que significa el final de la lista de resultados), no debemos invocar `relative()`, puesto que la posición actual está fuera de la lista de resultados. Antes de invocar `relative()`, debemos levar el cursor hasta una posición válida utilizando los métodos `absolute()`, `first()` o `last()`.

```
public boolean previous() throws SQLException
```

`previous()` desplaza el cursor hasta la fila anterior, relativa a la posición actual. A diferencia del método `relative()`, este método puede ser invocado incluso cuando no existe una fila anterior. Por ejemplo, después de invocar `next()` recursivamente hasta que devuelva `false`, podemos invocar `previous()` para llevar el cursor hasta la última fila.

Dirección y tamaño de toma

```
public void setFetchDirection(int direction) throws SQLException
```

Este método nos permite establecer una dirección de toma. La interfaz `java.sql.ResultSet` especifica tres constantes de dirección de toma: `ResultSet.FETCH_UNKNOWN`, `ResultSet.FETCH_FORWARD`, y `ResultSet.FETCH_REVERSE`. Estos tipos indican la dirección de toma actual para los resultados.

Esta lista de resultados puede optimizar internamente las estructuras de resultados para el tipo de desplazamiento especificado.

```
public int getFetchDirection() throws SQLException
```

Este método devuelve la actual dirección de toma.

```
public void setFetchSize(int rows) throws SQLException
```

Este método nos da una indicación del driver JDBC en cuanto al número de filas que deberían ser tomadas de la base de datos. Observe que, cuando el tamaño de toma está fijado en 1, la lista de resultados puede tomar los datos de la base de datos siempre que invoquemos `next()`. Fijando un tamaño de toma mayor, podremos utilizar mejores tomas optimizadas.

```
public int getFetchSize() throws SQLException
```

Este método devuelve el tamaño de toma actual.

Observe que una configuración inapropiada de dirección y tamaño de toma puede afectar al rendimiento. Al establecer estos valores, debemos considerar el número de filas que esperamos obtener en la lista de resultados y su supuesto uso.

Los siguientes ejemplo ilustran estos métodos. Fíjese en que los resultados de este código dependen del apoyo del driver:

```
public void queryByScrollableResultSet() throws SQLException {
   connection.setAutoCommit(false);

   Statement statement =
               connection.createStatement(ResultSet.TYPE SCROLL INSENSITIVE,
                                          ResultSet.CONCUR READ ONLY);

   ResultSet scroller=
               statement.executeQuery("SELECT CATALOG.TITLE, " +
                                      "CATALOG.LEAD ACTOR, " +
                                      "CATALOG.LEAD ACTRESS, " +
                                      "CATALOG.TYPE, " +
                                      "CATALOG.RELEASE_DATE FROM CATALOG");
```

Para poner en práctica los métodos indicados para listas de resultados desplazables, cree primero una lista de resultados. El fragmento anterior puebla un objeto `scroller` de lista de resultados:

```
int type = scroller.getType();
   switch(type) {
      case ResultSet.TYPE FORWARD ONLY :
         System.out.println("Type = TYPE FORWARD ONLY");
         break;
      case ResultSet.TYPE SCROLL SENSITIVE :
         System.out.println("Type = TYPE SCROLL SENSITIVE");
         break;
      case ResultSet.TYPE SCROLL INSENSITIVE :
         System.out.println("Type = TYPE SCROLL INSENSITIVE");
         break;
   }
```

```
System.out.println("Current fetch size: " + scroller.getFetchSize());
scroller.setFetchSize(2);
System.out.println("Fetch size reset to: " + scroller.getFetchSize());
```

El código anterior indica el nivel de apoyo para listas de resultados desplazables y nos permite cambiar el tamaño de toma. Del mismo modo, podemos conocer el nivel de apoyo para concurrencia.

El siguiente código desplaza la lista de resultados tanto hacia delante como hacia atrás. Como ya hemos mencionado, los resultados de este código dependen del apoyo del driver. En el momento de escribir este libro, Cloudscape 3.6.4 se adapta a listas de resultados de sólo lectura pero a no proporciona listas de resultados actualizables:

```
int count = 0;
   System.out.println("Scrolling forward from the beginning ...");
   while(scroller.next()) {
      System.out.println(count++ + ": " +
                         scroller.getString("TITLE") + ", " +
                         scroller.getString("TYPE"));
   }

   scroller.first();
   System.out.println("Scroller moved to first.");

   scroller.absolute(count/2);
   System.out.println("Scroller moved to " + count/2);
   System.out.println("Scrolling forward ...");
   while(scroller.next())  {
      System.out.println("      " + scroller.getString("TITLE") + ", "
                          + scroller.getString("TYPE"));
   }

   scroller.first();
   System.out.println("Scroller moved to first.");

   int fetchDirection = scroller.getFetchDirection();
   switch(fetchDirection)  {
      case ResultSet.FETCH_UNKNOWN:
        System.out.println("Current fetch direction: FETCH_UNKNOWN");
        break;
      case ResultSet.FETCH_FORWARD:
        System.out.println("Current fetch direction: FETCH_FORWARD");
        break;
      case ResultSet.FETCH_REVERSE:
        System.out.println("Current fetch direction: FETCH_REVERSE");
        break;
   }
   scroller.setFetchDirection(ResultSet.FETCH_REVERSE);
   System.out.println("Fetch direction set to FETCH_REVERSE");
   scroller.absolute(-count/3);
   System.out.println("Scroller moved to " + (-count/3));

   System.out.println("Scrolling forward ...");
   while(scroller.next())  {
      System.out.println("      " + scroller.getString("TITLE") + ", " +
                         scroller.getString("TYPE"));
   }

   scroller.previous();
   scroller.relative(-2);
   System.out.println("Scroller moved by -2 rows relative to the current" +
                      "position");
```

```
        System.out.println("Scrolling forward ...");
        while(scroller.next())  {
           System.out.println("       " + scroller.getString("TITLE") + ", " +
                                          scroller.getString("TYPE"));
        }

        scroller.close();
    }
```

Listas de resultados actualizables

Por defecto, las listas de resultados son de sólo lectura. Es decir, los contenidos de la lista de resultados son de sólo lectura y no pueden ser alterados. El API JDBC 2.1 también introduce listas de resultados actualizables. Cuando una lista de resultados es actualizada, la operación de actualización también actualiza los datos correspondientes a la lista de resultados.

La interfaz `java.sql.ResultSet` especifica dos constantes para indicar si la lista de resultados es de sólo lectura o es actualizable:

- `CONCUR_READ_ONLY`: Utilizando este tipo de lista de resultado no podemos utilizar ninguno de los métodos analizados a continuación para insertar, actualizar o eliminar filas.
- `CONCUR_UPDATABLE`: Con esta constante, podemos insertar, actualizar o eliminar filas en la lista de resultados.

Estos tipos son denominados tipos de concurrencia. Podemos conocer el tipo de concurrencia de una lista de resultados invocando el método `getConcurrency()` en la lista de resultados.

La interfaz `java.sql.ResultSet` especifica el siguiente grupo de métodos para actualizar resultados.

Actualizar una fila

El siguiente grupo de métodos `updateXXX()` está destinado a actualizar los elementos de la fila actual en la lista de resultados:

`updateArray()`	`updateAsciiStream()`	`UpdateBigDecimal()`
`updateBinaryStream()`	`updateBlob()`	`UpdateBoolean()`
`updateByte()`	`updateBytes()`	`UpdateCharacterStream()`
`updateClob()`	`updateDate()`	`UpdateDouble()`
`updateFloat()`	`updateInt()`	`UpdateLong()`
`updateNull()`	`updateObject()`	`UpdateRef()`
`updateRow()`	`updateShort()`	`UpdateString()`
`updateTime()`	`updateTimestamp()`	

Estos métodos requieren el nombre de columna (como un string) o el índice de columna como primer argumento y un objeto de tipo XXX. Por ejemplo, hay dos variantes de métodos `updateTimestamp()`:

```
public void updateTimestamp(int columnIndex, Timestamp x)
public void updateTimestamp(String columnName, Timestamp x)
```

Después de invocar estos métodos, debemos invocar el método `updateRow()` para actualizar los cambios. Alternativamente, podemos invocar el método `cancelRowUpdates()` para cancelar todas las actualizaciones realizadas hasta el momento.

El método `rowsUpdate()` nos permite saber si la fila actual ha sido actualizada. El valor devuelto depende de si el resultado puede detectar o no actualizaciones.

Eliminar una fila

Podemos utilizar el método `deleteRow()` para eliminar la fila actual de la lista de resultados así como de la base de datos subyacente. El método `rowDeleted()` indica si una fila ha sido eliminada utilizando la lista de resultados. Observe que una fila eliminada puede dejar un vacío visible en una lista de resultados. el método `rowDeleted()` puede utilizarse para detectar estos vacíos. Al desplazar una lista de resultados que pueda tener filas eliminadas, puede invocar este método para detectar estas filas eliminadas.

Insertar una fila

Para insertar una fila en la lista de resultados y en la base de datos subyacente, debemos primero utilizar `moveToInsertRow()` y después uno o más métodos de `updateXXX()`.

Una llamada al método `moveToInsertRow()` desplaza el cursor hasta la fila insertar. La fila insertar es un búfer especial asociado a una lista de resultados actualizable. Después de mover el cursor hasta esta fila, podemos utilizar los métodos `updateXXX()` habituales para fijar los elementos de esta fila. Al final de estas llamadas, debe invocar `insertRow()` para insertar finalmente la fila en una base de datos y para limpiar la fila búfer.

Actualizaciones de lotes (batch)

Además de las listas de resultados desplazables, el API JDBC 2.1 especifica apoyo para actualizaciones de lotes. Esta característica permite múltiples instrucciones de actualización (`INSERT`, `UPDATE` o `DELETE`) en una única solicitud a la base de datos. Procesar en lotes gran número de resultados puede tener como resultado importantes avances de rendimiento.

Por ejemplo, podemos volver a modificar la tabla para la base de datos de Películas pero esta vez podemos ejecutar todas las instrucciones `INSERT` en un único lote. También podemos utilizar instrucciones preparadas en un lote. Al comparar con `insertData()` en el ejemplo, la única diferencia en esta implementación son las llamadas al método `addBatch()` y el método `executeBatch()` en la instrucción para la actualización de lotes:

```
public void insertBatchData() throws SQLException, IOException {
   BufferedReader br = new BufferedReader(new FileReader("catalog.txt"));
   statement = connection.createStatement();

   try {
     do {
        title = br.readLine();
        if(title == null) break;
        leadActor = br.readLine();
        leadActress = br.readLine();
        type = br.readLine();

        dateOfRelease = br.readLine();

        String sqlString = "INSERT INTO CATALOG VALUES('" + title + "','" +
                           leadActor + "','" + leadActress + "','" +
                           type + "','" + dateOfRelease + "')";
```

```
            statement.addBatch(sqlString);
        } while(br.readLine() != null);

        statement.executeBatch();
    } catch (EOFException e) {

    } finally {
        statement.close();
        br.close();
    }
}
```

¿Por qué esto produce un mejor rendimiento? En lugar de ejecutar una serie de instrucciones, estamos realizando todas las actualizaciones en una única instrucción. Esto reduce el sobregasto del servidor de base de datos, puesto que, de no ser así, hubiéramos tenido que destinar y mantener recursos /cursores, etc.) para cada instrucción.

El paquete javax.sql

El paquete `javax.sql` está basado desde el punto de vista de la arquitectura en el paradigma de programación cliente-servidor. Como ya hemos analizado en las secciones anteriores, la típica programación JDBC incluye los siguientes pasos:

- Cargar el driver de la base de datos (utilizando el nombre de clase del driver)
- Obtener una conexión (utilizando el URL JDBC para la base de datos)
- Crear y ejecutar instrucciones
- Utilizar las listas de resultados para navegar por los resultados
- Cerrar la conexión

Si está familiarizado con él, reconocerá este modelo por ser exactamente igual a ODBC. Este modelo de programación es apropiado para clientes portátiles con conexiones de larga duración y transacciones de bases de datos localizadas. En tales aplicaciones, uno o más clientes portátiles (construidos utilizando Visual Basic, PowerBuilder, etc.) conectan a una base de datos central y realizan varias visitas a la base de datos para implementar diversos casos prácticos de empresa. Sin embargo, este modelo no es apropiado para aplicaciones distribuidas de base Web, principalmente por las siguientes razones:

- **Gestión de conexiones**
  Existen dos limitaciones al utilizar `java.sql.DriverManager` para la gestión de conexiones. Cada vez que un cliente requiere una conexión, este API intenta obtener una nueva conexión. Como veremos más adelante en este capítulo, este estilo de gestión de conexión no resulta eficiente para aplicaciones distribuidas con centro en Internet. En segundo lugar, `java.sql.DriverManager` no aísla las aplicaciones cliente de clases específicas de driver.
- **Apoyo de transacciones**
  Además, el paquete `java.sql` no tiene apoyo de arquitectura para transacciones distribuidas. Como veremos más adelante, el apoyo a transacciones distribuidas es esencial para construir aplicaciones de empresa reajustables y resistentes a los fallos.

El paquete `javax.sql` ofrece las siguientes características para tratar los puntos anteriores:

- **Búsqueda de base JNDI para acceder a las bases de datos vía nombres lógicos**
  En lugar de que cada cliente intente cargar las clases de driver en sus máquinas virtuales respectivas, utilizar las búsquedas de base JNDI nos permite acceder a los recursos de base de datos utilizando nombres lógicos asignados a estos recursos.

- **Reserva de conexiones**
  El paquete `javax.sql` especifica una capa intermedia adicional para implementar la reserva de conexiones. Es decir, la responsabilidad para la reserva de conexiones es trasladada desde desarrolladores de aplicaciones a vendedores-servidores de drivers y aplicaciones.

- **Transacciones distribuidas**
  El paquete `javax.sql` especifica un marco para apoyar las transacciones distribuidas de forma transparente bajo el paquete `java.sql`. Con este marco de trabajo, el apoyo a transacciones distribuidas puede ser activado en un entorno J2EE con una mínima configuración.

- **Conjunto de filas (Rowset)**
  Un objeto `Rowset` es un objeto adaptado a JavaBeans que encapsula listas de resultados de base de datos e información de acceso. Un conjunto de filas (rowset) puede ser conectado o desconectado. Observe que una lista de resultados debe permanecer conectada a una base de datos. Es decir, la conexión está abierta mientras existe el objeto. Sin embargo, el conjunto de filas nos permite encapsular un conjunto de filas, sin mantener necesariamente una conexión. El conjunto de filas también nos permite actualizar los datos y propagar los cambios de vuelta a la base de datos subyacente.

Estas funciones simplifican la programación JDBC en un paradigma distribuido como es J2EE. Exceptuando el uso de un conjunto de filas en lugar de una lista de resultados, utilizando JNDI, los servicios de transacciones distribuidas y de reserva de conexiones sólo requieren cambios mínimos en las aplicaciones cliente. Estos servicios son ahora responsabilidad del driver JDBC y del servidor de aplicación J2EE. Los vendedores de driver JDBC y los vendedores de servidor de aplicaciones J2EE pueden hacer uso de las capas intermedias en este API para construir estos servicios.

La mayor parte de clases e interfaces de este paquete están pensada para que los vendedores de servidor de aplicaciones y de drivers de base de datos las implementen en sus productos. El número de clases e interfaces que necesitaremos para desarrollar aplicaciones de cliente es reducido. Por lo tanto, en el resto de este capítulo, nuestro punto de atención se centrará en el análisis de estos servicios y no en cómo utilizarlos para desarrollar aplicaciones de cliente. Siempre que sea aplicable, proporcionaremos fragmentos de código para ilustrar los aspectos de programación.

Para poner a prueba los conceptos analizados en este capítulo, debe tener la última Implementación de Referencia J2EE de Sun Microsystems (disponible para su descarga en http://java.sun.com/j2ee/) o un servidor comercial de aplicación J2EE. También necesitará los drivers que se ajustan a éstos. Debe tener en cuenta, sin embargo, que no todos los vendedores de servidor de drivers y aplicaciones se ajustan en la actualidad a todas las funciones de este API. No obstante, es importante comprender cómo pueden ser diseñados estos servicios. Si desea una lista actualizada de vendedores que cumplan estas características, diríjase a la base de datos del driver en http://industry.java.sun.com/products/jdbc/drivers/.

En esta sección, analizaremos los siguientes puntos:

- Fuentes de datos: cómo funciona este mecanismo y cómo simplifica el acceso a la base de datos.
- Reserva de conexiones: instalaciones para la reserva de conexiones en el paquete `javax.sql`.
- Transacciones distribuidas: antecedentes y conceptos asociados a las transacciones distribuidas. Veremos cómo el API JDBC, junto con el API de Transacción Java, gestiona transacciones distribuidas.
- El conjunto de filas (rowset).

Fuentes de datos JDBC

El paquete `javax.sql` proporciona un sustituto para la clase estática `java.sql.DriverManager`, aunque todavía está incluida para compatibilidad retroactiva. El repuesto es la interfaz `javax.sql.DataSource` como medio primario de crear conexiones de base de datos. Con respecto de la programación, los siguientes son los factores destacables:

- En lugar de cargar explícitamente las clases del gestor de drivers en el periodo de ejecución de la aplicación cliente, utilizamos una búsqueda centralizada de servicio JNDI para obtener un objeto `java.sql.DataSource`.
- En lugar de utilizar la clase `java.sql.DriverManager`, utilizamos la interfaz `javax.sql.DataSource`, que ofrece funciones similares para obtener conexiones de base de datos.

Este enfoque aísla las aplicaciones cliente de la responsabilidad inicialización del driver de base de datos. Esto significa que las aplicaciones cliente no necesitan conocer las clases de driver de base de datos y el URL de base de datos. Estas tareas pasan ahora a ser responsabilidad del administrador de servidor que configura las fuentes de datos. Como veremos más adelante en este capítulo, esta separación también permite a servicios adicionales como reserva de conexiones y apoyo a transacciones ser activados sin afectar a aplicaciones adicionales.

La interfaz javax.sql.DataSource

La interfaz `javax.sql.DataSource` es una factoría para crear conexiones de base de datos. Una factoría es un modelo utilizado para crear instancias de clases sin tener que instanciar directamente clases de implementación.

La interfaz `javax.sql.DataSource` proporciona los siguientes métodos:

- `getConnection()`
- `getLoginTimeout()`
- `setLoginTimeout()`
- `getLogWriter()`
- `setLogWriter()`

Comencemos por `getConnection()`.

El método getConnection()

```
public Connection getConnection() throws SQLException
public Connection getConnection(String username,
                                String password) throws SQLException
```

Estos métodos devuelven una conexión a una fuente de datos. Observe que el tipo devuelto de este método es un objeto `Connection`. Éste es el mismo que es devuelto de la interfaz `java.sql.DriverManager`. Observe que cuando recuperamos un objeto `Connection` de la clase `java.sql.DriverManager`, es probable que el objeto `Connection` encapsule directamente una conexión de red. Sin embargo, como veremos más adelante en este capítulo, el objeto `Connection` no

necesita encapsular directamente una conexión física de base de datos. En su lugar, el objeto `Connection` puede delegar la mayoría de sus operaciones en otra capa que pueda implementar la reserva de conexiones y las transacciones distribuidas.

El método getLoginTimeout()

```
public int getLoginTimeout() throws SQLException
```

Este método devuelve el intervalo de tiempo en segundos que representa el tiempo que esperará la fuente de datos mientras intenta obtener una conexión a la base de datos. El plazo límite por defecto de registro es específico del driver/base de datos.

El método setLoginTimeout()

```
public void setLoginTimeout(int seconds) throws SQLException
```

Este método especifica el intervalo de tiempo en segundos que la fuente de datos debe esperar mientras intenta obtener una conexión a la base de datos.

Observe que el driver JDBC tendrá un intervalo de tiempo por defecto.

El método getLogWriter()

```
public PrintWriter getLogWriter() throws SQLException
```

Este método devuelve el actual objeto `java.io.PrintWriter` al que el objeto `DataSource` escribe los mensajes de registro. Observe que por defecto, a menos que se establezca un objeto `PrintWriter` utilizando el método `setLogWriter`, el registro es desactivado y el método devolverá `null`.

El método setLogWriter()

```
public void setLogWriter(LogWriter out) throws SQLException
```

Este método establece un `java.io.LogWriter` con intenciones de registro. Por ejemplo, podemos crear un objeto `PrintWriter` con `System.out` como flujo de salida y configurarlo como registro, como se muestra a continuación:

```
PrintWriter logWriter = new PrintWriter(System.out);

// dataSource es el objeto DataSource
datasource.setLogWriter(logWriter);
```

Observe que todos estos métodos lanzan una excepción `java.sql.SLQException`.

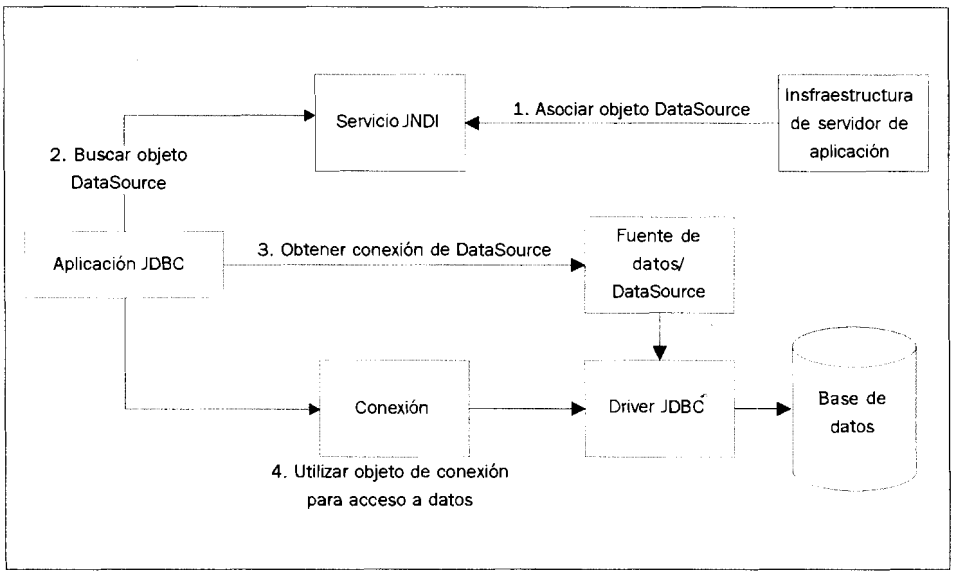
Como podemos ver en la lista de métodos anterior, esta interfaz es muy sencilla y su principal responsabilidad es crear conexiones.

Sin embargo, hay dos cuestiones referentes a este uso: ¿Quién crea un objeto implementando esta interfaz? ¿Cómo pueden los clientes de aplicación obtener esta instancia, de modo que las aplicaciones puedan obtener conexiones? El siguiente subapartado responde a estas preguntas.

JNDI y fuentes de datos

Una fuente de datos es considerada un recurso de red, recuperado de un servicio JNDI. En un servicio JNDI, las aplicaciones pueden asociar objetos a nombres. Otras aplicaciones pueden recuperar estos objetos utilizando estos nombres. Ambas aplicaciones que *asocian* objetos a nombres en el servicio JNDI y las aplicaciones que buscan estos nombres en el servicio JNDI pueden ser remotas. El API JDBC permite a los vendedores de servidor de aplicaciones y a los vendedores de drivers construir recursos de base de datos basados en este enfoque.

El siguiente diagrama ofrece un análisis de este enfoque:



Este diagrama muestra de forma esquemática cómo los objetos que implementan la interfaz `javax.sql.DataSource` están disponibles en un servicio JNDI y cómo los clientes de aplicación JDBC pueden buscar estos objetos, y crear objetos `Connection`.

Además de los clientes de aplicación JDBC y los recursos de base de datos, esta figura muestra dos componentes más: un servicio JNDI y una aplicación (normalmente, un contenedor J2EE) que asocia los objetos `javax.sql.DataSource` en el servicio JNDI. Un servicio JNDI es un proveedor del API JDBC. Normalmente, la aplicación que asocia los objetos `javax.sql.DataSource` en el servicio JNDI es un servidor de aplicación (que implementa posiblemente todos los servicios J2EE).

Los siguientes puntos son las responsabilidades de cada uno de los bloques de la figura anterior:

- El servidor de driver o de aplicación ("infraestructura" en la figura anterior) implementa la interfaz javax.sql.DataSource.
- El servidor de aplicación crea una instancia del objeto implementando la interfaz javax.sql.DataSource y la asocia a un nombre lógico (especificado por el administrador de servidor de aplicación) en el servicio JNDI indicado en el Paso 1.
- La aplicación JDBC (cliente) realiza una búsqueda en el servicio JNDI utilizando este nombre lógico, y recupera el objeto implementando la interfaz `javax.sql.DataSource`. Este es el segundo paso del diagrama.

- La aplicación JDBC (cliente) utiliza el objeto DataSource y obtiene conexiones de base de datos, como indica el Paso 3. La implementación de la fuente de datos puede que utilice el driver JDBC para recuperar una conexión.
- La aplicación JDBC (cliente) utiliza el objeto Connection para el acceso a todas sus bases de datos, utilizando el API JDBC estándar, el paso final en la secuencia.

Analicemos los detalles de los tres primeros pasos.

Crear una fuente de datos

Éste es un proceso sencillo y conlleva instanciar un objeto que implemente el objeto `javax.sql.DataSource` y asociarlo a un nombre. Esto está ilustrado a continuación.

```
XDataSource x = new XDataSource(...);
// Configurar propiedades para la fuente de datos

// Crear un contexto
try {

   Context context = new InitialContext();
   context.bind("jdbc/Orders", x);

} catch(NamingException ne) {
     // Error al crear el contexto o asociación.
}
```

En el fragmento de código anterior, `XdataSource` es una clase que implementa la interfaz `javax.sql.DaaSource`, implementada por los vendedores de driver y base de datos. Observe que el verdadero nombre de esta clase es irrelevante para el desarrollador de la aplicación. En su lugar, utilizamos el nombre lógico asignado a `DataSource`. En el ejemplo anterior, este es `jdbc/Orders`. Al compararlo con el enfoque basado en `java.sql.DriverManager`, el anterior sustituye la inicialización estática realizada por los drivers al registrar un driver.

La clase `InitialContext` es una clase que implementa la interfaz `javax.Naming.Context` del API JDBC. Dependiendo del entorno (lado cliente o lado servidor) en el que creamos el contexto inicial, puede que tengamos que utilizar el constructor `InitialContext` que toma un objeto `Hashtable` como argumento. Este `Hashtable` debe contener ciertos atributos de entorno que dirijan la creación del contexto inicial.

Algunos de los atributos más comúnmente utilizados incluyen `java.naming.provider.url`, `java.naming.security.principal` y `java.naming.security.credentials`. Estos atributos especifican un proveedor (un URL que apunta a un servicio JNDI) del servicio JNDI, la identidad (principal) de la aplicación o del usuario en cuyo nombre es creado este contexto, y las credenciales del principal. Mientras que el primer atributo especifica una posición del servicio JNDI, el segundo y tercer atributo especifican la identidad del llamante. Estos últimos argumentos pueden ser utilizados para implementar seguridad y control de acceso para el JNDI.

Si desea una lista completa de estos atributos, consulte la documentación de la interfaz `javax.naming.context` (disponible en http://java.sun.com/products/jndi/javadoc/javax/naming/Context.html).

Observe que tanto el constructor de la clase `javax.naming.InitialContext` como el método de asociación lanzan `javax.naming.NamingException`. Esta excepción es la clase raíz de todas las excepciones relacionadas con JNDI.

También es conveniente mencionar que el paso anterior es supervisado por el proveedor de servidor de aplicación J2EE. Es decir, cuando iniciamos el servidor de aplicación J2EE, éste instancia clases que

implementan la interfaz javax.sql.DataSource y asocia estos objetos a nombres lógicos del servicio JNDI. Fíjese en que la mayoría de los servidores de aplicación J2EE también incluyen un servicio JNDI, por lo que puede no ser necesario configura uno fuera del servidor de aplicación.

Aunque algunos de los vendedores de aplicación J2EE implementan la interfaz `javax.sql.DataSource` sobre drivers JDBC existentes, existen implementaciones `javax.sql.DataSource` procedentes de diferentes vendedores de bases de datos. En el momento de configurar el servidor de aplicación J2EE, será preciso que añadamos la configuración de la fuente de datos para los drivers de base de datos que pensamos utilizar. Hay dos situaciones diferentes:

- El servidor de aplicación que implementa el paquete `javax.sql` con driver JDBC de terceros. En este caso, será necesario que especifiquemos la clase `javax.sql.Driver` al configurar el servidor de aplicación.
- El vendedor de driver JDBC que implementa el paquete `javax.sql`: en este caso, sería necesario especificar la clase que implementa la interfaz `javax.sql.DataSource`. Podemos obtener esta información del vendedor del driver y en la documentación.

En ambos casos, cabe señalar que existe una capa adicional entre los drivers de base de datos tradicionales y el API que utilizamos para obtener conexiones.

Recuperar un objeto DataSource

Una vez que el servidor de la aplicación ha asociado un objeto `DataSource` en el servicio JNDI, cualquier aplicación cliente JDBC de la red puede recuperar el objeto `DataSource` utilizando el nombre lógico asociado a la fuente de datos. El siguiente fragmento de código ilustra este caso:

```
// Crear un contexto
try
{
    Context context = new InitialContext();
    DataSource dataSource = (DataSource) context.lookup("jdbc/x");
}
catch(NamingException ne)
{
    // Error al crear el conexto o la búsqueda.
}
```

El proceso es muy sencillo: creamos un objeto InitialContext y realizamos una búsqueda utilizando el nombre lógico asignado durante la configuración del servidor.

Puesto que un servicio JNDI existe habitualmente en una VM diferente y, normalmente, en una máquina diferente, la operación `lookup()` será, por lo general, una operación remota. De este hecho, podemos deducir que las clases de implementación `DataSource` también implementan la interfaz `java.io.Serializable`.

Características clave

Fíjese en las siguientes características clave de utilizar un método de fuente de datos:

- No es necesario que cada aplicación cliente inicialice los drivers JDBC. Como hemos visto anteriormente, los drivers JDBC requieren inicialización en cada aplicación cliente. Esto no será necesario con el paquete `javax.sql`. En cambio, es preciso que el servidor de aplicación convierta en disponibles los objetos de fuente de datos en el servicio JNDI.

- La aplicación cliente no tiene por qué conocer los detalles del driver. La única información requerida es un nombre lógico. Esto hace que el código de aplicación sea independiente de los drivers y de los URLs JDBC.
- Puesto que este enfoque utiliza un servicio JNDI para localizar los objetos de fuente de datos utilizando nombres lógicos, este enfoque proporciona una búsqueda independiente de la posición de las fuentes de datos. Los objetos `DataSource` pueden ser creados, desplegados y gestionados en un servicio JNDI, independiente de todas los clientes de aplicación.

Revisión del catálogo de películas

Modifiquemos ahora la aplicación del catálogo de películas para utilizar una fuente de datos. Sólo hay dos cambios que realizar a los archivos `CreateMovieTable.java` y `QueryMovieTable.java`. Éstos son analizados a continuación:

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.DatabaseMetaData;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

// Importar clases javax.sql
import javax.sql.DataSource;
// Importar clases JNDI
import javax.naming.Context;
import javax.naming.NamingException;
```

Fíjese en las dos instrucciones de importación adicionales para importar clases desde los paquetes `javax.sql` y `javax.naming`.

El otro cambio se encuentra en el método `initialize()` en el que cargamos orginalmente el driver y creamos una conexión. En lugar se esto, debemos ahora obtener un objeto `javax.sql.DataSource` y crear una conexión:

```
public void initialize() throws SQLException, NamingException {
   // En vez de registrar el driver como a continuación
   /*     Class.forName (driver);
   // buscar y obtener un objeto DataSource de JNDI
   Context initialContext = new InitialContext();
   dataSource = (DataSource) initialContext.lookup("jdbc/Cloudscape");

   // Reemplazar
   //     connection = DriverManager.getConnection(url);
   // utilizando el objeto DataSource para obtener una conexión.
   connection = dataSource.getConnection();
   }
```

La primera instrucción crea un objeto `javax.naming.Context`. La clase `InitialContext` implementa la interfaz `javax.naming.Context` y proporciona el punto de partida para la resolución de nombres en el servicio JNDI.

En el ejemplo anterior, utilizamos `initialContext` para buscar en `jdbc/Cloudscape`. Éste es nombre que especificamos al configurar el servidor aplicación, de modo que el servidor aplicación pueda crear una instancia de `javax.sql.DataSource` y asociarla con el nombre `jdbc/`

`Cloudscape` en el servicio JNDI. `jdbc` es el subcontexto de nombrado estándar para todos los objetos `DataSource`.

El método `lookup()` devuelve un objeto que implementa la interfaz `javax.sql.DataSource`. Puesto que el método `lookup()` devuelve un objeto `java.lang.Object`, necesitamos enviarlo a `javax.sql.DataSource`.

El resto de las operaciones de la base de datos se realizan de forma habitual. Para poner en práctica esta aplicación, necesitamos un servidor de aplicación J2EE que se ajuste a las fuentes de datos para aplicaciones cliente.

Reserva de conexiones

Una aplicación de servidor, por definición, realiza un servicio para uno o más clientes. A medida que aumenta el número de clientes, también aumenta la importancia de servir al cliente del modo más eficiente posible.

Una de las técnicas utilizadas para servir de forma eficiente las solicitudes de los clientes es la reutilización exhaustiva de objetos de recursos costosos. Siempre hay un sobregasto en la creación de un objeto (gasto de memoria, inicialización del objeto y la JVM debe mantener un registro del objeto de forma que pueda ser recogido por el recolector de residuos cuando ya no sea necesario). En general, es una buena idea minimizar el número de objetos que creamos en las aplicaciones; tanto más en las aplicaciones de lado servidor (como aplicaciones Web), donde los números de clientes y solicitudes de clientes no son fácilmente predecibles.

Además, los objetos que encapsulan las conexiones de red son más caros de crear que otros. Si podemos reutilizar un objeto, el rendimiento del servidor puede mejorar extraordinariamente.

La **reserva de conexiones** es una técnica destinada a gestionar y reutilizar objetos. Puesto que crear objetos `Connection` es una de las operaciones más caras en términos de recursos, el tipo más frecuente de reserva de objeto suele ser la reserva de conexiones. En JDBC, un objeto `Connection` representa una conexión nativa de base de datos (excepto en bases de datos integradas en la memoria, que no son muy comunes en sistemas de producción). El servidor de base de datos debe distribuir recursos de comunicación y de memoria así como autentificar el usuario y configurar un contexto de seguridad para cada conexión. Aunque existen varios parámetros que dictan el tiempo para obtener una conexión, no es usual ver tiempos de conexión de uno o dos segundos (dependiendo de la conexión, carga de base de datos, etc.). Compartiendo un conjunto de conexiones entre clientes, en lugar de intentar crearlas cada vez que sean necesarias, podemos mejorar las cargas en los recursos y, consecuentemente, la capacidad de respuesta de la aplicación.

Una reserva de conexiones de base de datos beneficia a la mayoría de aplicaciones de lado servidor que accede a una base de datos. Utilice una reserva de conexiones si las siguientes características describen su aplicación:

- Los usuarios acceden a la base de datos a través de un pequeño conjunto común de cuentas de usuario de base de datos. La alternativa es que cada usuario utiliza una cuenta específica. En aplicaciones centradas en Internet, las cuentas comunes de usuarios son más frecuentes.
- Una conexión de base de datos sólo se utiliza durante una única solicitud, en oposición a la duración combinada de múltiples solicitudes de la misma sesión.

El segundo criterio es más interesante. Una típica aplicación de e-comercio permite a los usuarios añadir elementos a un carro de la compra a su antojo mientras navegan por la página. Cuando los usuarios terminan sus compras, pasan por caja y pagan los contenidos del carro. Podemos diseñar esta aplicación al menos de dos formas:

- El carro de la compra es una tabla de base de datos. Cuando el usuario entra en la página, obtenemos una conexión de base datos y la mantenemos hasta que el usuario pasa por caja o sale de la página (o después del tiempo límite de la sesión). Cada artículo añadido al carro significa añadir una fila a la tabla. Si el usuario pasa por caja, realizamos la transacción de base de datos y si el usuario sale de la página sin comprar, deshacemos la transacción. Si la aplicación sigue este diseño, no es necesaria una reserva de conexión ya que cada conexión está asociada a las actualizaciones de datos en el carro y no puede ser compartida.
- El contenido del carro de la compra se mantiene en un objeto normal integrado en la memoria asociado al usuario. Por ejemplo, en una aplicación Web basada en un servlet/JSP, este objeto puede ser mantenido en el objeto `HttpSession`. Cada artículo añadido al carro significa añadir el artículo al objeto integrado en la memoria. Si el usuario pasa por caja, obtenemos una conexión a la base de datos, añadimos todos los artículos del objeto integrado en la memoria como filas en una tabla de base de datos, y realizamos la transacción. Si el usuario sale de la página sin comprar, simplemente suprimimos el objeto integrado en la memoria. Este modelo satisface los criterios anteriores por lo que podemos utilizar una reserva de conexiones.

El paquete `javax.sql` proporciona una reserva de conexiones basada en una fuente de datos que es transparente a las aplicaciones de cliente. Como veremos en breve, en este enfoque, la reserva de conexiones puede ser activada y configurada por un administrador de un servidor de aplicación J2EE que se ajuste a esta característica.

El enfoque tradicional ha ido uniéndose programáticamente durante tiempo al enfoque de reserva de conexiones. Observe que hay algunas variantes de este enfoque tradicional disponibles, como `DBConnectionPool` analizada en la primera edición de este libro (también disponible en un breve artículo en http://www.Webdevelopersjournal.com/columns/connection_pool.html) y `DBConnectionBroker` (disponible en http://javaexchange.com). Aunque estas reservas ofrecen una amplia gama de servicios, en principio son muy similares. Antes de analizar los conceptos subyacentes al apoyo JDBC para la reserva de conexiones, veamos cómo se realiza una reserva de conexiones tradicional.

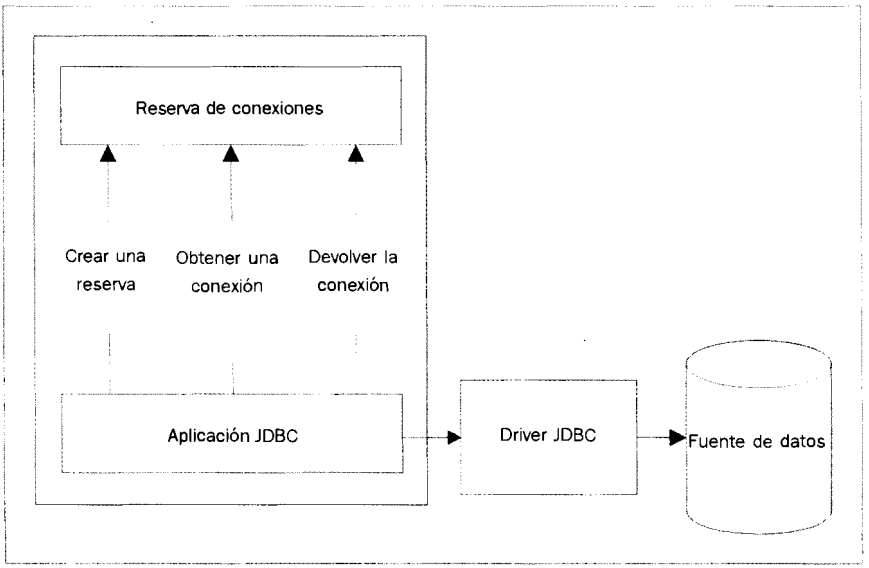
Reserva de conexiones tradicional

En este enfoque, una aplicación:

- Obtiene una referencia a la reserva o a un objeto que gestiona varias reservas
- Consigue una conexión de una reserva
- Utiliza la conexión
- Devuelve la conexión a la reserva

La aplicación es completamente consciente de que está utilizando una conexión de reserva, por lo que nunca debe cerrar la conexión. En su lugar, debe devolver la conexión a la reserva.

La implementación del modelo clásico consiste normalmente en una clase (por ejemplo, `ConnectionPool`) que gestiona un conjunto de objetos `Connection` JDBC. Esta clase proporciona métodos para inicializar la reserva (para fijar el número de conexiones que deben ser abiertas en el arranque, URL de JDBC, el número máximo de conexiones, etc), para obtener conexiones y para devolver conexiones. Los clientes de esta clase inicializan primero la reserva antes de obtener conexiones. El siguiente diagrama ilustra esta situación:



Como vemos en la figura anterior, la reserva de conexiones es tratada como una responsabilidad del cliente. La aplicación cliente crea la reserva, recupera las conexiones y libera las conexiones de nuevo. Además de esto, este enfoque requiere un contrato estricto por parte del desarrollador cliente de aplicación:

- La aplicación cliente no debe cerrar la conexión; sólo debe devolverla a la reserva. Un contrato como éste, sin embargo, no puede ser aplicado estrictamente en aplicaciones cliente. Si el cliente cierra accidentalmente la conexión antes de devolverla a la reserva, la conexión será inservible; la reserva eliminará la referncia a ella y la recreará si es necesario.
- Cuando el cliente utiliza el objeto Connection en una transacción y olvida terminar (aceptar o deshacer) la transacción, la reserva de conexiones no puede detectarlo. Esto provoca que las transacciones se mezclen y que se produzca una pérdida de consistencia de datos.

Aunque estos puntos pueden cuidarse codificando las prácticas concienzudamente, la implementación de reservas de conexiones consistentes es todavía responsabilidad de la aplicación cliente.

Esta clásica reserva de conexiones es válida únicamente para clientes que comparten el mismo periodo de ejecución. Ésta es una reserva de lado cliente. Dentro del alcance de este libro, hay dos posibilidades en las que múltiples clientes comparten el mismo entorno:

- Las aplicaciones Web J2EE desplegadas en un único contenedor. En este caso, múltiples servlets y objetos JSP serán invocados en diferentes threads pero en el mismo periodo de ejecución (siempre que el contenedor no sea distribuido).
- Una aplicación de servidor multi-thread que proporcione métodos de empresa para el acceso a bases de datos por parte de los clientes.

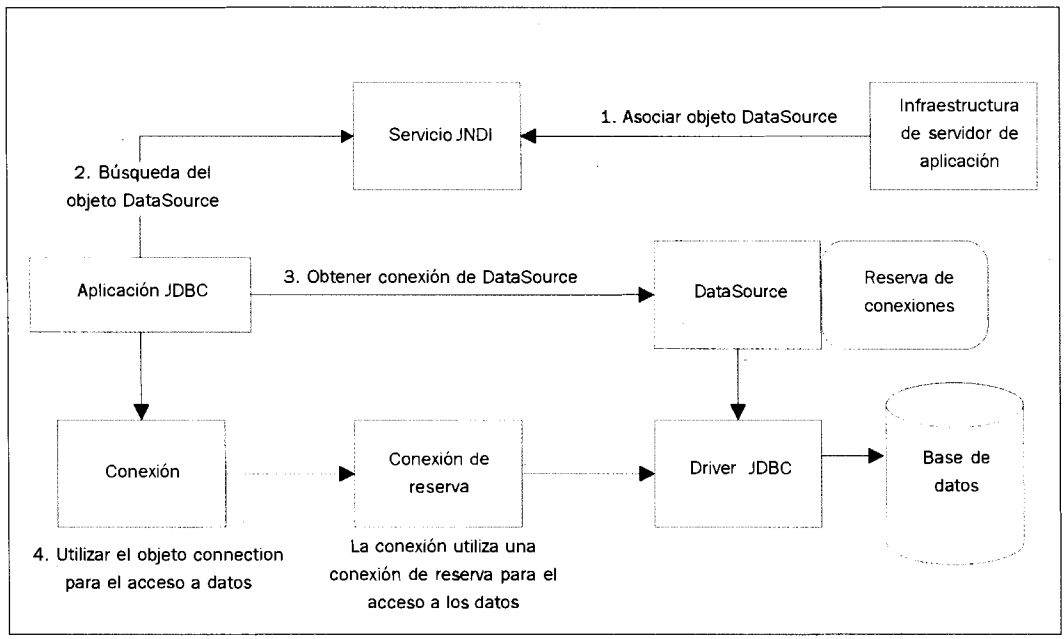
Cuando se requieren conexiones desde diferentes aplicaciones cliente, la clásica reserva no es la elección adecuada. Para esos casos, sería más conveniente una reserva de conexión de lado servidor. Cuando cada cliente mantiene su propia reserva de conexión, es probable que la reutilización de conexiones no sea óptima. Una reserva de conexiones de lado servidor evita esta situación. Como veremos en la siguiente sección, el API JDBC especifica reservas de conexiones que cumplen este requisito. Actualmente, la mayoría de los servidores de aplicación J2EE ofrecen esta función.

Dejando a un lado los detalles sobre qué clases son utilizadas para albergar conexiones, este enfoque no es sólido ni reajustable. El defecto fundamental de la reserva clásica de conexiones es que esta reserva no funciona con transacciones distribuidas. También hace estragos en los entornos controlados por contenedores como contenedores Web, ya que le obliga a mantener conexiones en variables estáticas, sobre las que el contenedor no puede ejercer su control.

Reserva de conexiones con el paquete javax.sql

El paquete javax.sql proporciona un medio transparente de reserva de conexión. Con este enfoque, el servidor de aplicación y/o el driver de base de datos maneja la reserva de conexiones de forma interna. Siempre que usemos los objetos DataSource para obtener conexiones, la reserva de conexiones se activará automáticamente una vez configuremos el servidor de aplicación J2EE.

Antes de entrar en los detalles de este enfoque, aquí tiene una visión esquemática:



En comparación con la primera figura, el único cambio es la reserva de conexiones adicional mantenida por el servidor de aplicación en coordinación con el driver JDBC. Esto significa que no es preciso programación adicional para las aplicaciones de cliente JDBC. En su lugar, el administrador del servidor J2EE tendrá que configurar una reserva de conexiones en el servidor de aplicación. La sintaxis exacta y los nombres de las clases dependen de la implementación. Sin embargo, con un servidor de aplicación y un driver de base de datos que se ajusten a JDBC 2.0 el administrador de servidor especifica normalmente lo siguiente:

- Una clase que implementa la interfaz `javax.sql.ConnectionPoolDataSource`
- Una clase que implementa la interfaz `java.sql.Driver`
- El tamaño de la reserva (tamaño máximo y mínimo)

- Tiempo límite de la conexión
- Parámetros de autentificación (registro, contraseña, etc.)

En el resto de esta sección, analizaremos las interfaces relacionadas con la reserva de conexiones y cómo se implementa la reserva de conexiones utilizando estas interfaces. El propósito de esta sección es ofrecerle una visión de la reserva de conexiones de JDBC 2.0. Además de configurar el servidor J2EE para activar la reserva de conexiones, las aplicaciones de cliente no necesitan implementar o acceder a estas interfaces directamente.

El paquete `javax.sql` especifica tres interfaces y una clase para la implementación de reservas de conexiones: `javax.sql.ConnectionPoolDataSource`, `javax.sql.PooledConnection`, `javax.sql.ConnectionEventListener` y `javax.sql.ConnectionEvent`.

La interfaz javax.sql.ConnectionPoolDataSource

Esta interfaz es similar a la interfaz `javax.sql.DataSource`. En vez de devolver objetos `java.sql.Connection`, esta interfaz devuelve objetos `javax.sql.PooledConnection`. Es decir, la fuente de datos de reserva de conexiones es una factoría para objetos de conexión de reserva. Los siguientes métodos devuelven objetos `javax.sql.PooledConnection`:

```
public javax.sql.PooledConnection getPooledConnection()
          throws java.sql.SQLException

public javax.sql.PooledConnection
                      getPooledConnection (String user, String password)
          throws java.sql.SQLException
```

Los métodos restantes de esta interfaz son los mismos que los de la interfaz `javax.sql.DataSource`.

Incluso después de que hayamos configurado la reserva de conexiones, las aplicaciones todavía utilizan objetos `javax.sql.DataSource` para obtener objetos de conexión, como hemos visto en el método `initialize()` de la aplicación del catálogo de películas. Es el objeto `javax.sql.DataSource` el que utiliza `javax.sql.ConnectionPoolDataSource` para crear y mantener una reserva de conexiones. Ésta es una parte de la implementación de `DataSource` (proporcionada por el vendedor de driver o de servidor) y la reserva de conexiones se mantiene transparente al código de aplicación. Una vez que el administrador establece la reserva de conexiones, empezaremos automáticamente a utilizar las conexiones de reserva.

La interfaz javax.sql.PooledConnection

Cuando se activa una reserva de conexiones, los objetos que implementan la interfaz `javax.sql.PooledConnection` albergan una conexión de base de datos física.

Esta interfaz es una factoría de objetos `javax.sql.Connection`. Esta interfaz especifica los siguientes métodos:

```
public java.sql.Connection getConnection() throws java.sql.SQLException
```

Este método devuelve un objeto `java.sql.Connection`. Sin embargo, el objeto Connection devuelto es realmente un asa (o proxy) para la conexión física que mantiene el objeto `javax.sql.PooledConnection`.

```
public void close() throws java.sql.SQLException
```

Este método cierra la conexión a la base de datos.

Pero, ¿cuál es la diferencia entre una conexión de reserva y una conexión?

- En primer lugar, la conexión es una conexión física en aplicaciones cliente JDBC. Con la reserva de conexiones JDBC, el objeto `Connection` no alberga una conexión de base de datos física. Lo que la aplicación obtiene es una abstracción de un objeto `Connection` corriente. El objeto `Connection` utiliza la conexión física que alberga el objeto `Connection` de reserva para el acceso a todos sus datos.
- En segundo lugar, cuando invocamos el método `close()`, la conexión física no se cierra. La implementación `DataSource` elimina meramente la asociación entre el objeto de conexión y el objeto `Connection` de reserva. Podemos, por lo tanto, cerrar la conexión después de utilizarla; es entonces cuando la fuente de datos reutiliza la conexión de reserva.
- En tercer lugar, no es necesario que devolvamos el objeto `Connection` a ninguna reserva. Simplemente lo cerramos, como haríamos sin reserva de conexiones.

En otras palabras, el objeto `Connection` es un envoltorio sobre `PooledConnection`. Cuando la fuente de datos devuelve una conexión, mantiene una asociación entre el objeto `Connection` devuelto y el objeto `PooledConnection`, de modo que las llamadas sobre el objeto `Connection` son delegadas en el objeto`PooledConnection`.

La interfaz `javax.sql.PooledConnection` tiene dos métodos más para manejar la asociación entre la conexión y la conexión de reserva. Recuerde que cuando se cierra el objeto `Connection`, la reserva de conexiones debe ser notificada, de modo que la conexión de reserva pueda ser reutilizada:

```
public void addConnectionEventListener(ConnectionEventListener listener)

public void removeConnectionEventListener(ConnectionEventListener listener)
```

Estos dos métodos son utilizados por la reserva de conexiones para añadir y eliminar escuchantes de eventos de conexión.

La interfaz javax.sql.ConnectionEventListener

La interfaz `ConnectionEventListener` tiene dos métodos:

```
public void connectionClosed(ConnectionEvent event)
```

Este método es invocado cuando la aplicación llama al método `close()`. La reserva de conexiones marca la conexión para su reutilización.

```
public void connectionErrorOccured(ConnectionEvent event)
```

Este método es invocado cuando ocurren errores fatales de conexión. La reserva de conexión puede cerrar la conexión en este evento y eliminarlo de la reserva.

La clase javax.sql.ConnectionEvent

Esta clase representa los eventos relacionados con conexiones.

Observe que este mecanismo de manejo de eventos es similar a los eventos AWT. La reserva de conexiones añade escuchantes de eventos de conexión a la conexión de reserva y los escuchantes de conexión son notificados cuando tienen lugar los eventos de conexión.

Implementación de reservas de conexión

El API JDBC no especifica ningún algoritmo concreto para reserva de conexiones. Las interfaces que hemos analizado anteriormente son ganchos para implementar una reserva de conexiones. En una

implementación típica, se suceden los siguientes pasos cuando invocamos `getConnection()` en la fuente de datos:

- La fuente de datos primero comprueba un objeto `PooledConnection` libre. Si hay uno, la reserva de conexiones devuelve un objeto `PooledConnection` a la fuente de datos. La fuente datos invoca entonces `getConnection()` en el objeto `PooledConnection`, que devuelve un objeto`Connection`.
- Si no hay un objeto `PooledConnection` libre disponible en la reserva de conexiones, la fuente de datos utiliza una fuente de datos de reserva de conexiones para obtener una nueva conexión de reserva.

Como podemos ver, la interfaz `javax.sql.ConnectionPoolDataSource`es una factoría para objetos `javax.sql.PooledConnection`. además, la interfaz`javax.sql.PooledConnection`es una factoría para objetos java.sql.Connection. `javax.sql.DataSource` implementa la reserva de conexiones y coordina las conexiones de reserva y las conexiones regulares con la ayuda de las interfaces examinadas.

Como una conexión es meramente un asa para una conexión de reserva (que alberga la conexión de red), una conexión puede ser creada y cerrada varias veces utilizando la misma conexión de reserva. Además, el API cliente (las interfaces `javax.sql.DataSource` y `java.sql.Connection`) se mantiene igual porque los detalles de la reserva de conexiones nunca son expuestos a la aplicación.

Una vez más, la principal ventaja de este enfoque es su sencillez, tanto para los desarrolladores de aplicaciones como para los administradores de sistemas. Los administradores de sistemas pueden activar/configurar la reserva de conexiones independientemente de todas las aplicaciones.

Transacciones distribuidas

Al examinar la interfaz `java.sql.Connection`, analizamos brevemente las transacciones. La interfaz `java.sql.Connection` tiene dos métodos, `commit()` y `rollback()`, para realizar y deshacer cambios, respectivamente. Controlamos los límites (el inicio y el final) de la transacción invocando `setAutoCommit(false)` al principio de la transacción e invocando el método `rollback()` o el método `commit()` para finalizar la transacción.

Tales transacciones son denominadas transacciones locales. En una transacción local, las aplicaciones cliente ejecutan operaciones de lectura y escritura/actualización de base de datos sobre un objeto `Connection`.

En determinadas situaciones, sin embargo, tenemos más de un cliente (por ejemplo, dos servlets diferentes o componentes EJB) participando en una transacción. Alternativamente, el cliente puede que tenga que ejecutar dos operaciones de base de datos en múltiples bases de datos en la misma transacción. ¿Cómo podemos implementar estas características?

Aquí es cuando intervienen las transacciones distribuidas. El paquete `javax.sql`, junto con **Java Transaction API (JTA)** puede utilizarse para implementar transacciones distribuidas.

En esta sección, realizaremos un análisis de las transacciones distribuidas y veremos las interfaces relevantes de JDBC y JTA.

¿Qué es una transacción?

Las aplicaciones de empresa a menudo requieren acceso concurrente a datos distribuidos compartidos por múltiples componentes, para ejecutar operaciones en datos. Tales aplicaciones deben mantener la

integridad de datos (tal y como están definidos por las reglas de empresa de la aplicación) en las siguientes circunstancias:

- **Acceso distribuido a un único recurso de datos**
  Por ejemplo, podemos tener dos componentes de aplicación que implementan dos partes de una transacción de empresa; ambos utilizan la misma base de datos pero diferentes conexiones.
- **Acceso a recursos distribuidos desde un único componente de aplicación**
  Por ejemplo, un componente que actualiza múltiples bases de datos en una única transacción de empresa. Esta operación requeriría múltiples objetos `Connection`.

En ambos casos, los métodos proporcionados por el API JDBC central no son adecuados. Esto se debe a que los métodos proporcionados en `java.sql.Connection` para iniciar y finalizar transacciones están asociados a un único objeto `Connection`. En las situaciones anteriores, sin embargo, no ocurre esto.

El concepto de transacción y de gestor de transacciones (o servicio de procesamiento de transacciones) simplifica el desarrollo de tales aplicaciones al mantenimiento de la integridad de datos en una unidad de trabajo como se describe en las propiedades ACID, garantizando que una transacción nunca queda incompleta, que los datos nunca son inconsistentes, que las transacciones concurrentes con independientes, que los efectos de una transacción son persistentes.

En la siguiente sección aprenderá que representan las siglas ACID y cómo las propiedades ACID son mantenidas por los sistemas de procesamiento de transacciones.

Antecedentes

En el dominio del procesamiento de transacciones distribuidas, el modelo X/Open Distributed Transaction Processing (DTP) es el modelo más adoptado para la construcción de aplicaciones transaccionales. Casi todos los vendedores que desarrollan programas relacionados con el procesamiento de transacciones, bases de datos relacionales y colas de mensajes, se ajustan a las interfaces definidas en el modelo DTP.

Este modelo define tres componentes:

- Programas de aplicación
- Gestores de recursos
- Gestor de transacciones

Este modelo también especifica interfaces funcionales entre programas de aplicación y el gestor de transacciones (conocido como **interfaz TX**), y entre el gestor de transacciones y los gestores de recursos (la **interfaz XA**). Utilizando los gestores de transacciones que se ajusten a TX y los gestores de recursos que se ajusten a XA (como las bases de datos), podemos implementar transacciones con el protocolo bifásico de realizar y recuperar para cumplir los requisitos para las transacciones.

El servicio de transacción de objetos, Object Transaction Service (OTS), es otro modelo de procesamiento de transacciones distribuidas especificado por el grupo de gestión de objetos, Object Management Group (OMG). Este modelo está basado en el modelo X/Open DTP y sustituye las interfaces funcionales TX y XA por interfaces CORBA IDL (Common Object Request Broker Architecture Interface Definition Language). En este modelo, los diversos objetos se comunican mediante llamadas de método CORBA sobre Internet Inter ORB Protocol (IIOP). IIOP ha sido especificado por OMG específicamente para implementar soluciones CORBA en Internet. El modelo OTS es interoperable con el modelo X/Open DTP. Una aplicación que utilice objetos transaccionales podría utilizar la interfaz XT con el gestor de transacciones para la demarcación de transacciones.

En la arquitectura J2EE, el JDBC y el Java Transaction API proporciona interfaces para implementar transacciones distribuidas. Además, el servicio de transacciones Java, Java Transaction Service (JTS) especifica una representación de OTS.

Procesamiento de transacciones – Conceptos

Antes de proceder a examinar los API relevantes de JDBC y JTA, analicemos brevemente algunos de los conceptos asociados a las transacciones distribuidas:

- Demarcación de transacciones
- Contexto de transacciones y propagación
- Alistamiento de recursos
- Realización en dos fases

Comenzaremos con la demarcación de transacciones.

Demarcación de transacciones

Una transacción puede ser especificada por lo que se conoce como demarcación de transacciones. La demarcación de transacción permite que el trabajo realizado por componentes distribuidos se asocie a una transacción global. Por ejemplo, para transacciones locales en el momento que creamos una transacción e invocamos `setAutoCommit (false)`, iniciamos una transacción. De un modo similar, una llamada a `commit()` o `rollback()` finaliza la transacción.

Esto no es así para transacciones distribuidas. El enfoque más común a la demarcación es tomar el thread que ejecuta las operaciones y marcarlo para el procesamiento de transacción. Esto es lo que se denomina **demarcación programática**. La transacción establecida de este modo puede ser suspendida desmarcando el thread y retomada más tarde marcando de nuevo el thread. Para marcar y retomar la transacción, el contexto de transacción debe ser propagado explícitamente hasta el thread en el que retoma la transacción. Sin embargo, con sistemas de procesamiento de transacciones como los contenedores EJB, esto se convierte en responsabilidad del contenedor.

La demarcación de transacción finaliza después de una solicitud de realizar o deshacer al gestor de transacciones. La solicitud realizar (commit) dirige todos los gestores de recursos participantes a ejecutar los efectos de las operaciones de la transacción permanente. La solicitud de deshacer (rollback) da instrucciones a los gestores de recursos para que deshagan los efectos de todas las operaciones realizadas sobre la transacción.

Una alternativa a la demarcación programática es la **demarcación declarativa**. En esta técnica, los componentes son marcados como transaccionales en el momento de despliegue. Esto tiene dos implicaciones:

- Primero, la responsabilidad de demarcación cambia pasa de la aplicación al contenedor que alberga el componente. Por este motivo, esta técnica también es denominada **demarcación controlada por contenedor.**
- Segundo, la demarcación es pospuesta desde el tiempo de construcción de aplicación (estático) al tiempo de despliegue del componente (dinámico).

Los sistemas de procesamiento de transacciones basados en componentes tales como servidores de aplicación basado en la especificación Enterprise JavaBeans y Microsoft Transaction Server cumplen la demarcación declarativa.

Contexto de transacciones y propagación

Puesto que en una transacción participan múltiples componentes de aplicación y recursos, es necesario que el gestor de transacciones establezca y mantenga el estado de la transacción según ocurre. Esto se realiza normalmente en forma de contexto de transacción.

El contexto de transacción es una asociación entre las operaciones transaccionales en los recursos y los componentes que invocan las operaciones. Durante el transcurso de una transacción, todos los hilos que participan en la transacción comparten el contexto de transacción. Así el contexto de transacción envuelve lógicamente todas las operaciones ejecutadas sobre recursos transaccionales durante una transacción. El gestor de transacciones subyacente mantiene normalmente el contexto de transacción de forma transparente. Por ejemplo, un cliente Java o un componente EJB que implemente transacciones no necesita explícitamente propagar el contexto. El contenedor subyacente J2EE lo maneja.

Alistamiento de recursos

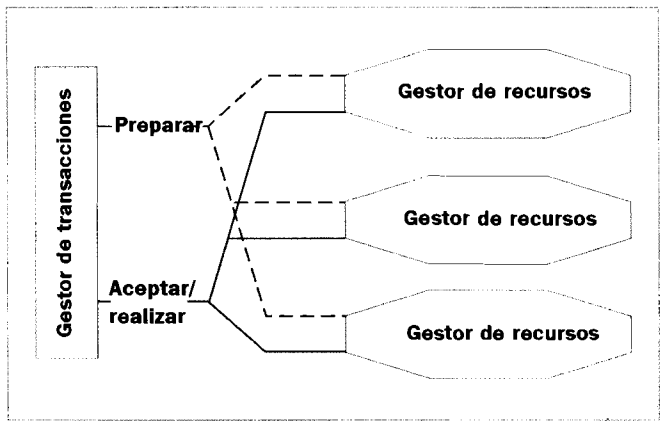
El alistamiento de recursos es el proceso mediante el cual los gestores de recursos informan al gestor de transacciones para que participen en una transacción. Este proceso permite al gestor de transacciones registrar todos los recursos que participan en una transacción. El gestor de transacciones utiliza esta información para coordinar el trabajo transaccional ejecutado por los gestores de recursos y para dirigir el protocolo bifásico de realización y recuperación.

Al final de una transacción (después de aceptar o deshacer), el gestor de transacciones libera los recursos. Después, la asociación entre las transacciones y los recursos no se mantiene.

Aceptar en dos fases

Este protocolo entre el gestor de transacciones y todos los recursos alistados para una transacción certifica que todos los gestores de recursos acepten (`commit`) la transacción o que todos la aborten. En este protocolo, cuando la aplicación solicita que la transacción sea aceptada, el gestor de transacciones emite una solicitud de **preparación** a todos los gestores de recursos implicados. Cada uno de estos recursos enviará una respuesta a su vez indicando si está preparado o no para realizar su operación.

Únicamente cuando todos los gestores de recursos están preparados para aceptar, el gestor de transacciones emite una solicitud de **commit** a todos los gestores de recursos. De no ser así, el gestor de transacciones emite una solicitud de deshacer y la transacción será deshecha. El siguiente diagrama ilustra estas fases:



Las líneas de puntos indican el proceso de preparación durante el cual cada gestor de recursos manifiesta su voluntad de aceptar. Después de que todos los gestores de recursos indiquen que están preparados para aceptar, el gestor de transacciones emite la llamada de aceptar que aparece en el diagrama con una línea continua.

¿Cómo puede el gestor de recursos o el gestor de transacciones solicitar deshacer la operación cuando la aplicación ha decidido que la transacción debe ser aceptada? Esto puede pasar, por ejemplo, cuando uno de los recursos que participan en la transacción no es alcanzable debido a fallos en la red o en el sistema. Esto también podría ocurrir si uno de los recursos que bloquea ciertos datos ya no puede mantener el bloqueo debido a una expiración del plazo. La fase de aceptación en dos fases certifica que cada recurso participante puede realmente aceptar y realizar la transacción.

El protocolo bifásico de aceptación tiene sus limitaciones. Por ejemplo, no puede recuperarse de casos en los que un gestor de recursos ha indicado que está preparado para aceptar, pero no ha conseguido aceptar (debido a fallos internos). En este caso, los recursos que se aceptaron con anterioridad a este fallo no pueden ser deshechos. Una situación similar surge cuando hay un fallo en la red durante el proceso de aceptación.

Construcción de bloques de sistemas de procesamiento de transacciones

Cualquier arquitectura típica de procesamiento de transacciones implica un gestor de transacciones y gestor de recursos para cada recurso. Estos componentes abstraen la mayoría de los temas específicos de transacciones de las aplicaciones y comparten la responsabilidad de implementación de transacciones.

Componentes de aplicación

Los componentes de aplicación son clientes para los recursos transaccionales. Éstos son los programas con los que el desarrollador de aplicaciones implementa las transacciones de empresa.

Con la ayuda del gestor de transacciones, estos componentes crean transacciones globales, propagan el contexto de transacción si es necesario y operan sobre los recursos transaccionales dentro del alcance de estas transacciones. Estos componentes no son responsables de la implementación de la semántica para preservar las propiedades ACID de las transacciones. Como parte de la lógica de aplicación, estos componentes generalmente deciden si aceptar o deshacer las transacciones. Las siglas ACID representan Atomic (atómico), Consistent (consistente), Isolated (aislado) y Durable (duradero).

Las siguientes son las responsabilidades habituales de un componente de aplicación:

- Crear y demarcar transacciones
- Propagar contextos de transacciones
- Operar sobre datos mediante gestores de recursos

Gestores de recursos

Un gestor de recursos es un componente que gestiona sistemas de almacenamiento de datos estables y persistentes, y participa en los protocolos bifásicos de aceptación y recuperación junto con el gestor de transacciones. Los ejemplos son sistemas de bases de datos, colas de mensajes, etc.

Además, un gestor de recursos es normalmente un driver o un envoltorio sobre un sistema de almacenamiento, con interfaces que operan sobre los datos (para los componentes de aplicación). Este componente puede también registrar, directa o indirectamente, recursos con el gestor de transacciones de

modo que el gestor de transacciones puede registrar todos los recursos que participen en una transacción. Este proceso se denomina alistamiento de recursos. El gestor de recursos también debe implementar mecanismos suplementarios (por ejemplo, registro) que hagan posible la recuperación.

Los gestores de recursos proporcionan dos grupos de interfaces: un grupo para que los componentes de aplicación obtengan conexiones y ejecuten operaciones sobre los datos, y el otro grupo para que el gestor de transacciones participe en el protocolo bifásico de aceptación recuperación.

Las siguientes son las típicas responsabilidades de los gestores de recursos:

- Alistar recursos con el gestor de transacciones
- Participar en el protocolo bifásico de aceptación y recuperación

Gestor de transacciones

El gestor de transacciones es el componente central de un entorno de procesamiento de transacciones. Sus principales responsabilidades son crear transacciones cuando éstas sean solicitadas por los componentes de aplicación, permitir el alistamiento y "delistamiento" de recursos, y ejecutar el protocolo de aceptación/recuperación con los gestores de recursos.

Una típica aplicación transaccional empieza con la emisión de una solicitud a un gestor de transacciones para iniciar una transacción. En respuesta, el gestor de transacciones inicia una transacción y la asocia con el hilo llamante. El gestor de transacciones también establece un contexto de transacción. Todos los componentes de aplicación y los hilos que participan en la transacción comparten el contexto de transacción. El hilo que inicialmente emitió la solicitud para iniciar la transacción, o, si el gestor de transacciones lo permite, cualquier otro hilo, puede finalmente completar la transacción emitiendo una solicitud de aceptar o deshacer.

Antes de que sea completada una transacción, cualquier número de componentes y/o hilos puede ejecutar operaciones transaccionales obre cualquier número de recursos transaccionales conocidos para el gestor de transacciones. Si el gestor de transacciones lo permite, una transacción puede ser suspendida y después retomada antes de ser completada finalmente.

Una vez que la aplicación emite la solicitud de aceptar, el gestor de transacciones prepara todos los recursos para una operación de aceptación (dirigiendo un voto) y, basándose en la situación de los recursos (si están o no preparados), emite una solicitud de aceptar o deshacer.

Las siguientes son las típicas responsabilidades de un gestor de transacciones:

- Establecer y mantener el contexto de transacción
- Mantener la asociación entre una transacción y los recursos participantes
- Iniciar y dirigir el protocolo bifásico de aceptación y recuperación con los gestores de recursos
- Realizar llamadas de sincronización a los componentes de aplicación antes de iniciar y después de finalizar el proceso bifásico de aceptación y recuperación.

Transacciones distribuidas JDBC

Gracias a la sección anterior, ha quedado claro que el apoyo de las transacciones distribuidas es una necesidad absoluta para construir aplicaciones distribuidas de empresa. Debido a J2EE es una tecnología destinada a construir tales aplicaciones, JDBC incluye apoyo a transacciones distribuidas. El apoyo a las transacciones distribuidas en JDBC especifica las responsabilidades de un driver de base de datos (equivalente a un gestor de recursos en el examen anterior).

El paquete javax.sql incluye dos interfaces: `javax.sql.DataSource` y `javax.sql.XAConnection`. Observe que, por la convención seguida por el modelo X/Open DTP, las interfaces que comienzan con las letras "XA" representan las interfaces para gestores de recursos, es decir, drivers de base de datos o servidores de aplicación que se ajustan a transacciones distribuidas.

La interfaz javax.sql.XADataSource

Esta es una fuente de datos transaccional. Esta interfaz es una factoría para objetos `javax.sql.XAConnection`, y es similar a la interfaz `javax.sql.DataSource`. Sin embargo, en lugar de crear objetos `javax.sql.Connection`, una implementación de esta interfaz crea objetos `javax.sql.XAConnection`. `javax.sql.XAConnection` es una interfaz `Connection` transaccional.

Los siguientes son los métodos clave de esta interfaz:

```
public javax.sql.XAConnection getXAConnection()
                              throws java.sql.SQLException
public javax.sql.XAConnection getXAConnection (String user, String password)
                              throws java.sql.SQLException
```

Los métodos restantes de esta interfaz son los mismos que los de la interfaz `javax.sql.DataSource`.

La interfaz javax.sql.XAConnection

Esta interfaz proporciona apoyo a las transacciones distribuidas. Esta interfaz amplía la interfaz `javax.sql.PooledConnection` y naturalmente apoya todos los métodos especificados en la interfaz `javax.sql.PooledConnection`. Además de estos métodos, la interfaz `javax.sql.XAConnection` especifica el siguiente método adicional:

```
public javax.transaction.xa.XAResource getXAResource()
                                       throws java.sql.SQLException
```

¿Cuál es el propósito de este método? Para comprender mejor el objetivo de este método, consideremos los conceptos de procesamiento de transacciones examinados en la sección anterior.

Los componentes de aplicación en transacciones distribuidas pueden acceder a los datos a través de múltiples conexiones de bases de datos. Es decir, una transacción consiste en diversas conexiones, con cada conexión operando (leyendo, actualizando o eliminando) en un determinado conjunto de recursos de base de datos. Como hemos visto en la sección anterior, cada recurso debe ser alistado, de modo que el gestor de transacciones pueda registrar todos los recursos que hayan podido ser modificados durante la transacción. El JTA especifica la interfaz `javax.Transaction.xa.XAResource` para este propósito. Para cada objeto `Connection`, el gestor de transacciones obtiene un `javax.Transaction.xa.XAResource` y lo alista en la transacción actual.

El paquete `javax.transaction.xa` es una parte del API Java Transaction (JTA).

Antes de continuar para analizar cómo podemos implementar transacciones distribuidas, debemos considerar una interfaz adicional.

La interfaz javax.Transaction.UserTransaction

La demarcación de transacciones programáticas se lleva a cabo con el uso de la interfaz `UserTransaction`. Ésta es innecesaria en entornos controlados por contenedor como contenedores EJB.

Esta interfaz no forma parte de JDBC pero sí del API Java Transaction (JTA). Podemos acceder a este API online en http://java.sun.com/products/jta/javadocs-1.0.1/index.html.

La interfaz `UserTransaction` encapsula la mayor parte de la funcionalidad de un gestor de transacciones y son los vendedores de servidores de aplicación (o software intermediario transaccional) quienes deben implementarla. Los métodos expuestos por esta interfaz proporcionan los medios para iniciar y finalizar transacciones explícitamente:

```
public void begin() throws NotSupportedException, SystemException
```

Con la ayuda de este método, podemos crear una nueva transacción. `NotSupportedException` y `SystemException` son parte del paquete `javax.transaction`. La excepción `NotSupportedException` es lanzada cuando el gestor de transacciones no puede iniciar una transacción. Como ejemplo, esto puede ocurrir si estuviéramos intentando iniciar una segunda transacción dentro del mismo hilo. La excepción `SystemException` es lanzada en caso de errores inesperados.

```
public void commit() throws RollbackException, HeuristicMixedException,
                            HeuristicRollbackException,
                            java.lang.SecurityException,
                            java.lang.IllegalStateException, SystemException
```

`commit()` acepta explícitamente todas las operaciones de base de datos ejecutadas hasta el momento en la transacción actual. El gestor de transacciones subyacente utiliza el proceso bifásico de aceptación y recuperación para completar el proceso de aceptación.

```
public void rollback() throws java.lang.IllegalStateException,
                              java.lang.SecurityException,
                              SystemException
```

Podemos utilizar este método para deshacer la transacción actual.

De los tres métodos anteriores, el primero marca el inicio del límite de una transacción, mientras que el segundo y tercer método finalizan explícitamente una transacción. Estos métodos son utilizados en la demarcación explícita de transacciones.

```
public void setRollbackOnly() throws java.lang.IllegalStateException,
                                     SysemException
```

`setRollbackOnly()` permite a una aplicación aplicar deshacer y la transacción no debe ser aceptada desde ese momento. Después de invocar este método, si intentamos invocar el método `commit()` anterior, la aplicación recibirá la excepción `javax.Transaction.RollbackException`.

```
public int getStatus() throws SystemException
```

Este método devuelve el estado actual de la transacción. Estos valores de estado están definidos en la interfaz `javax.Transaction.Status`.

```
public void setTransactionTimeout(int seconds) throws SystemException
```

Podemos especificar el intervalo de plazo de tiempo máximo para la transacción con `setTransactionTimeout()`. Esto permite al gestor de transacciones reclamarla transacción pendiente y deshacer todos los cambios en caso de que el cliente siga inactivo más allá de este intervalo.

Pasos para la implementación de transacciones distribuidas

Los siguientes pasos son necesarios para la demarcación programática de transacciones. En transacciones controladas por contenedor (declarativas), el único paso a seguir es la configuración.

Configuración

El primer paso es configurar el servidor de aplicación y el driver de base de datos, siguiendo el procedimiento descrito por su servidor de aplicación. Similares a la interfaz `javax.sql.ConnectionPoolDataSource` y `javax.sql.PooledConnection`, las dos interfaces anteriores deben ser implementadas por vendedores de servidor de aplicación y vendedores de driver de base de datos. Por lo tanto, el entorno de servidor de aplicación debe ser configurado para utilizar la interfaz `javax.sql.XADataSource`. Sin embargo, la mayoría de vendedores de servidor de aplicación J2EE tienen sus propias implementaciones de este API, en cuyo caso puede no ser necesario que realicemos esta configuración.

Iniciar una transacción

Para iniciar una transacción, debemos obtener una objeto que implemente la interfaz `javax.Transaction.UserTransaction`. El enfoque estándar para recuperar este objeto es ejecutar una búsqueda en el servicio JNDI. El siguiente fragmento de código ilustra esta situación:

```
import javax.sql.*;            //  Import the javax.sql package
import javax.naming.*;         //  Import the JNDI API
import javax.transaction.*;    //  Import the JTA

//  ...

   try {
      Context initialContext = new InitialContext();
      UserTransaction ut = (UserTransaction)
                  initialContex.lookup("javax.transaction.UserTransaction");

      ut.begin()
   }
```

Fíjese en que importamos el paquete `javax.Transaction` además de los paquetes `javax.sql` y `javax.naming`.

En el fragmento anterior, la búsqueda de JNDI busca `javax.Transaction.UserTransaction`. Éste es el nombre estándar asociado a la interfaz `javax.Transaction.UserTransaction`. El servidor de aplicación, que implementa esta interfaz, asocia un objeto que implementa esta interfaz en el servicio JNDI.

Operaciones de base de datos

Aparte de ciertas restricciones sobre la invocación de determinados métodos sobre el objeto `java.sql.Connection`, el resto del procedimiento para la búsqueda de JNDI de un objeto de fuente de datos, para crear conexiones, y ejecutar instrucciones SQL, es el mismo. El siguiente fragmento de código no ofrece una instantánea:

```
DataSource dataSource = null;
   UserTransaction ut = null;

    // Crear un contexto
    ...

   try {
      Context context = new InitialContext();
       dataSource = (DataSource) context.lookup("jdbc/x");
       ut = (UserTransaction)
              context.lookup("javax.transaction.UserTransaction");
   } catch (NamingException ne) {
```

```
        // Error al crear el contexto o en la búsqueda.
    }

    Connection connection = dataSource.getConnection();

    // ...

    // Ejecutar operaciones de base de datos habituales

    // ...

    // Deshacer la transacción en caso de cualquier fallo de condición de empresa
    if(...){
       // Obtener la UserTransaction.
       ut.setRollbackOnly()
    }

    connection.close()

    // Realizar algo que utilice otra fuente de datos y otro
    // objeto de conexión
    doSomeThing();

    // ...
```

El código anterior realiza operaciones similares a las que realizaríamos en transacciones de base de datos. Sin embargo, la implementación de la lógica de aplicación puede decidir producir una condición para deshacer la transacción.

Veamos qué ocurre cuando invocamos el método `getConnection()` en la fuente de datos. Con nuestro servidor de aplicación configurado para transacciones distribuidas, el objeto de fuente de datos utiliza la implementación subyacente de la interfaz `javax.sql.XADataSource` para crear objetos `javax.sql.XAConnection`. Fíjese en que un `XAConnection` es un tipo especial de `PooledConnection` (`javax.sql.XAConnection` amplía la interfaz `javax.sql.PooledConnection`). `DataSource`, por lo tanto, obtiene una conexión de este objeto `javax.sql.XAConnection` invocando el método `getConnection` y la devuelve a la aplicación. Observe que esto es exactamente lo que sucede con la reserva de conexiones.

Hay un paso adicional: el alistamiento de recursos. Antes de devolver una conexión, el gestor subyacente de transacciones obtiene el objeto `javax.transaction.XAResource` de `XAConnection` y alista recursos para la transacción. Los paquetes `javax.transaction` y `javax.transaction.xa` cuentan con otros paquetes que asisten en el alistamiento de recursos y en el proceso bifásico de aceptación y recuperación. Sin embargo, como desarrollador de aplicaciones, sólo nos interesa la interfaz `javax.transaction.UserTransaction`.

Finalizar una transacción

El procedimiento para finalizar una transacción es similar el modo en el que iniciamos una transacción. Ahora invocamos el método `commit()` en el objeto `UserTransaction` para finalizar la transacción. Puede que necesitemos realizar una búsqueda JNDI adicional para recuperar el objeto `UserTransaction` si la transacción se está cerrando desde un método u objeto diferente al que inició la transacción.

Precauciones especiales

En una transacción, no se pueden ejecutar las siguientes operaciones:

- No debemos invocar los métodos `commit()` y `rollback()` en objetos `Connection`. Esto se debe a que el objeto de transacción de usuario controla los límites de la transacción. Estos métodos lanzan la excepción `java.sql.SQLException` cuando son invocados en una transacción.
- No debemos permitir la autoaceptación (auto-commit) en objetos `Connection`. Observe que auto-commit es contrario a la noción de transacciones. Cuando recuperamos una conexión bajo una transacción, la conexión tendrá auto-commit desactivado por defecto y si intentamos activarlo invocando el método `setAutoCommit()`, lanza la excepción `java.sql.SQLException`.

Objetos RowSet

El rasgo final del paquete `javax.sql` que analizaremos es el objeto `RowSet`.

En términos sencillos un rowset (grupo de filas) es un componente que se ajusta a JavaBean que encapsula el acceso a base de datos incluido el resultado. La interfaz `javax.sql.RowSet` representa la funcionalidad básica para un rowset. Para comprobar los beneficios de utilizar un rowset, consideremos un simple acceso a bases de datos para tomar un grupo de filas. Veamos primero cómo conseguiríamos esto sin utilizar un rowset:

```
// Crear una conexión
connection = DriverManager.getConnection("jdbc:cloudscape:Movies");
connection.setAutoCommit(false);

// Crear una conexión
Statement statement = connection.createStatement();

// Ejecutar una consulta
ResultSet rs = statement.executeQuery("SELECT * FROM CATALOG");

while(rs.next()) {
   // Desplazar y procesar los resultados
}
```

En este enfoque, hay al menos tres objetos que representan diferentes aspectos de una operación de búsqueda: un objeto `java.sql.Connection` para conectividad de base de datos, un objeto `java.sql.Statement` para ejecutar una instrucción y un objeto `java.sql.ResultSet` para albergar los valores devueltos.

En su lugar, consideremos un enfoque en el que la misma operación de consulta sea encapsulada por un único objeto:

```
JdbcRowSet rowSet = new JdbcRowSet();

rowSet.setCommand("SELECT * FROM CATALOG WHERE TYPE = 'ROMANCE'");
rowSet.setUrl("jdbc:clouscape:Movies");
rowSet.setUsername("sa");
rowSet.setPassword("");
rowSet.execute();

while(rowSet.next()) {
    // Desplazar y procesar los resultados
}
```

El código anterior ilustra una posible aplicación de objetos `RowSet`. En este código, esta clase `JdbcRowSet` es parte del paquete `sun.jdbc.rowset`. Este paquete es una implementación de referencia

de la especificación RowSet. Puede descargarla junto con ejemplo en http://developer.java.sun.com/developer/earlyAccess/crs/.

Aparte de esta encapsulación, la principal característica de un grupo de filas es que un objeto rowset puede utilizarse como componente JavaBean. Como bean, un RowSet proporciona un conjunto de mutadores para establecer propiedades y un grupo complementario de accesores para obtener los valores de esas propiedades. En el ejemplo anterior, invocamos programáticamente los configuradores para configurar las propiedades del rowset. En su lugar, podríamos utilizar un entorno de programación visual para crear un objeto RowSet y establecer sus propiedades.

Los objetos RowSet también se ajustan a los eventos de estilo JavaBean. Los objetos RowSet generan eventos cuando tienen lugar ciertos cambios en el estado del grupo de filas. Los objetos interesados pueden registrarse como escuchantes para estos eventos y que les sea notificado a medida que van sucediendo los eventos.

> *En esencia, un objeto RowSet es un componentes JavaBeans; tiene ciertas propiedades que pueden ser manipuladas por configuradores y una serie de eventos que pueden escuchar los objetos interesados. Un objeto RowSet también encapsula los resultados de ejecutar una instrucción.*

La interfaz javax.sql.RowSet puede ser implementada como una capa separada encima de una capa JDBC existente. La implementación de referencia de Sun es de hecho una capa de este tipo y podemos utilizarla encima de cualquier driver JDBC que se adapte a JDBC 2.1.

La interfaz javax.sql.RowSet

```
public interface RowSet extends java.sql.ResultSet
```

La interfaz javax.sql.RowSet amplía la interfaz java.sql.ResultSet. Esto se debe a que un rowset también encapsula un grupo de filas (resultados) separados de las propiedades. Como hemos visto, la interfaz java.sql.ResultSet ofrece un rico espectro de métodos para acceder a filas de resultados. Por lo tanto, todos los métodos que son aplicables para una lista de resultados lo son también para un grupo de filas (rowset).

Propiedades

La interfaz javax.sql.RowSet proporciona una serie de propiedades JavaBeans que pueden ser fijadas en el periodo de diseño. Con la ayuda de estas propiedades, el objeto RowSet puede conectar a una fuente de datos y recuperar una lista de resultados en el periodo de ejecución. Los siguientes son algunos de los parámetros comúnmente requeridos:

- URL: un URL JDBC.
- Nombre de fuente de datos: Nombre de la fuente de datos. Dependiendo de la implementación, el grupo de filas puede utilizar bien un URL o bien un nombre de fuente de datos para obtener una conexión.
- Nombre de usuario: Nombre de usuario para obtener una conexión.
- Contraseña de usuario: Contraseña para obtener una conexión.
- Aislamiento de transacción: Nivel de aislamiento de transacción. Ésta es una propiedad de java.sql.Connection que especifica un conjunto de constantes para nivel de aislamiento.
- Comando: La instrucción SQL. El comando también puede ser un string de instrucciones preparadas.

Eventos

Los objetos `RowSet` pueden generar tres tipos de eventos:

- Eventos de movimiento de cursos: estos son generados cuando el cursor es desplazado. Por ejemplo, invocar un método `previous()` en el objeto `RowSet` generaría un evento.
- Eventos de cambio de fila: estos eventos son generados cuando las filas de un grupo de filas son alteradas. Por ejemplo, eliminar una fila generaría un evento.
- Eventos de cambio de grupo de filas: estos eventos son generados cuando todos los contenidos del grupo de filas cambian. Por ejemplo, evocar `execute()` cambiaría el contenido completo de un grupo de filas.

Para que los objetos de aplicación sean notificados de estos eventos, el API JDBC especifica la interfaz `javax.sql.RowSetListener`:

```
public interface RowSetListener extends java.util.EventListener
```

Esta interfaz especifica tres métodos:

```
public void cursorMoved(RowSetEvent event)
public void rowChanged(RowSetEvent event)
public void rowSetChanged(RowSetEvent event)
```

El objeto `RowSet` invoca estos métodos cuando tiene lugar uno de los eventos enumerados.

Los objetos de aplicación que implementan la interfaz `javax.sql.RowSetListener` pueden registrar y "desregistrar" estos eventos con rowset, utilizando los siguientes métodos en `javax.sql.RowSet`:

```
public void addRowSetListener(RowSetListener listener);
public void removeRowSetListener(RowSetListener listener);
```

Estos métodos son similares a los que encontramos en AWT y Swing. Podemos añadir dinámicamente más de un escuchante. También podemos eliminar estos escuchantes dinámicamente.

Cuando los objetos `Listeners` (escuchantes) son notificados, reciben el objeto `RowSet` encapsulado en `javax.sql.RowSetEvent`.

Ejecución de comandos y resultados

Después de establecer las propiedades y los escuchantes, podemos invocar el método execute() en el grupo de filas para poblar los resultados y utilizar cualquiera de los métodos especificados en la interfaz `java.sql.ResultSet` para desplazar y modificar los resultados. El método `execute()` obtiene internamente una conexión de base de datos, prepara una instrucción y crea una lista de resultados.

Crear y utilizar los objetos `RowSet` es muy sencillo y sigue el modelo JavaBeans. Para poblar una lista de resultados, hay tres tareas que es preciso realizar:

- Crear un objeto `RowSet`
- Configurar sus propiedades
- Ejecutarlo

Tipos de objetos RowSet

El API JDBC no especifica interfaces estándar que implementen la interfaz `javax.sql.RowSet`. La entrega de primer acceso de Sun, sin embargo, identifica tres posibles implementaciones. Aunque en la

actualidad éstas no forman parte de la especificación, en el futuro podrían ser añadidas a la especificación como parte de una serie de interfaces revisadas.

Las siguientes son tres posibles implementaciones de esta interfaz:

- Rowsets caché (`sun.jdbc.rowset.CachedRowSet`)
- Rowsets JDBC (`sun.jdbc.rowset.JdbcRowSet`)
- Rowsets Web (`sun.jdbc.rowset.WebRowSet`)

Estas implementaciones están especificadas en el paquete `sun.jdbc.rowset`, que no forma parte del API JDBC . En la actualidad, Sun ofrece una implementación de referencia de estos objetos `RowSet` en una entrega Early Access (en la actualidad, entrega 4), que puede ser descargada de la página Web de Sun Java Developer Connection (`http://java.sun.com/jdbc/`). Esta entrega de primer acceso incluye implementaciones de estos tres tipos de objeto `RowSet`.

Todas las clases anteriores se extienden desde `sun.jdbc.rowset.BaseRowset` e implementan la interfaz `javax.sql.RowSet`. La clase `BaseRowSet` proporciona la implementación común para propiedades, eventos y métodos para establecer parámetros para objetos `RowSet` (puesto que los objetos rowset implementan la interfaz `java.sql.ResultSet`).

Aparte de la implementación de referencia, algunos de los vendedores de drivers JDBC están empezando a apoyar las implementaciones para estos tipos de `RowSet`. Si desea una lista de drivers que se ajusten a rowset, visite http://industry.java.sun.com/products/jdbc/drivers/.

En las siguientes subsecciones, analizaremos brevemente cada una de estas implementaciones `RowSet`.

La implementación CachedRowSet

Una lista de resultados JDBC es un objeto conectado. Es decir, siempre que la lista de resultados esté abierta (y antes de que invoquemos el método `close()` sobre ella), la lista de resultados mantiene una conexión con la base de datos. Esto es aceptable si mantuviéramos la lista de resultados en cortos intervalos. Sin embargo, cuando pretendemos utilizar los objetos ResultSet para periodos significativos de tiempo, mantener la conexión consumirá niveles inaceptables de recursos de red. Por ejemplo, si una lista de resultados tiene 1000 filas y la aplicación Web está presentando 10 resultados por página, puede que necesitemos mantener el objeto `Connection` durante un marco de tiempo significativamente largo. Esto no es deseable. Para atajar este problema, tendríamos que transferir los resultados a objetos de adaptación, de modo que podamos liberar la instrucción y la conexión inmediatamente después de ejecutar la instrucción.

El rowset caché trata este problema. Una vez ha sido creado el rowset caché y los resultados han sido poblados, la instrucción y los objetos `Connection` pueden ser cerrados. El rowset caché mantiene las filas de resultados de forma "desconectada". Cuando actualizamos o modificamos el `RowSet`, el rowset caché conecta con la base de datos y ejecuta las actualizaciones en la base de datos y, por lo tanto, limita el coste de recursos.

La clase `sun.jdbc.rowset.CachedRowSet` define un `RowSet` caché.

```
public class CachedRowSet extends BaseRowSet
                          implements javax.sql.RowSet,
                                     javax.sql.RowSetInternal,
                                     java.io.Serializable,
                                     java.lang.Cloneable
```

Un rowset caché es un contenedor para filas de datos desconectado, serializable, clonable y desplazable. Amplía `sun.jdbc.rowset.CachedRowSet` e implementa las interfaces `javax.sql.RowSet`,

`RowSetInternal`, `java.io.Serializable` y `java.lang.Cloneable`. Está desconectado y, por lo tanto, no alberga un objeto `Connection`. Puesto que es serializable y clonable, podemos crear copias de nuestro rowset caché y transferirlas por la conexión. Por ejemplo, un componente EJB puede crear y devolver un rowset caché a un cliente en la red. El cliente puede modificar el `RowSet` y enviarlo de vuelta al componente, y el componente puede realizar actualizaciones. Para que las aplicaciones poblen y accedan/modifiquen el estado de un rowset caché, el rowset caché también implementa la interfaz `javax.rowset.RowSetInternal`.

El objeto `CachedRowSet` es apropiado para dos escenarios como mínimo:

- La aplicación cliente pretende mantener los resultados durante un intervalo de tiempo significativamente largo. Esto es así para las aplicaciones Web así como para las aplicaciones cliente-servidor que ejecutan consultas con un número de filas de resultados posiblemente alto.
- El cliente no tiene ni la capacidad ni los recursos para conectar con una base de datos. Por ejemplo, los dispositivos redificados como PDA y otros clientes infradotados no tienen los recursos necesarios para conectar con bases de datos. En estos casos, una aplicación de lado servidor puede crear un rowset caché y enviarlo por la red al cliente infradotado. El cliente puede desplazar el rowset y puede también guardarlo para un uso posterior. El cliente puede enviar una copia de los resultados de vuelta a la aplicación de lado servidor siempre que la actualice.

En efecto, el rowset caché nos proporciona la capacidad para crear listas de resultados "descargables".

El procedimiento para crear y utilizar un rowset caché es similar al que hemos visto al principio de esta sección:

```
CachedRowSet rowSet = new CachedRowSet();

rowSet.setCommand("SELECT * FROM CATALOG WHERE TYPE = 'ROMANCE'");
rowSet.setUrl("jdbc:cloudscape:Movies;create=true");
rowSet.setUrl("jdbc:cloudscape:Movies");
rowSet.setUsername("sa");
rowSet.setPassword("");
rowSet.execute();

while(rowSet.next()) {
   // Desplazar los resultados.
}
```

El rowset caché se puebla cuando se invoca el método `execute()`.

Los métodos `updateXXX()` son los especificados en la interfaz `java.sql.ResultSet` y están destinados a listas de resultados actualizables.

Puesto que el rowset caché está desconectado, la base de datos no aplica ningún control de concurrencia en los datos albergados por el rowset caché. Para mantener una concurrencia óptima, la implementación actual de un rowset caché cachetiza el valor original de los datos. Esta copia representa el estado del rowset caché cuando se invoca el método `execute()`. Podemos acceder a los datos originales invocando los métodos `getOriginal()` y `getOriginalRow()` sobre el objeto `CachedRowSet`. Estos métodos devuelven objetos `ResultSet` que se corresponden con el rowset caché y la fila actual respectivamente.

Puesto que el rowset está desconectado, debemos invocar el método `acceptChanges()` para actualizar el rowset:

```
rowSet.acceptChanges();
```

Este método restablece una conexión con la base de datos y realiza las actualizaciones necesarias a la base de datos. Si los datos originales albergados en `CachedRowSet` no coinciden con los datos de la base de datos, este método no actualiza la base de datos con los valores cambiados. Observe que este método lanza la excepción `SQLException` en caso de fallos al efectuar los cambios.

La implementación RowSet JDBC

El **rowset JDBC** es un rowset conectado. El objetivo del rowset JDBC es proporcionar una capa tipo JavaBeans encima de `java.sql.ResultSet`. Está definida en la clase `sun.jdbc.rowset.JdbcRowSet`:

```
public class JdbcRowSet extends BaseRowSet implements javax.sql.RowSet
```

En esta especificación de clase, a diferencia de un rowset caché, un rowset JDBC está conectado, no es serializable y no es clonable. Proporciona las características que proporcione la interfaz subyacente `java.sql.ResultSet`, con la ventaja añadida de presentarse a sí misma como un JavaBean.

La implementación RowSet Web

En el momento de escribir este libro, la especificación rowset no es definitiva y la implementación está siendo distribuida como entrega de primer acceso. La implementación de la entrega de primer acceso no está completa en sí misma. En esta sección, vamos a centrarnos por lo tanto en los conceptos que se encuentran detrás del rowset Web, sin entrar en detalles de programación.

El siguiente apartado representa la estructura deseada de este tipo rowset. El rowset Web está destinado a aplicaciones de base Web. Por este motivo, el modo primario en que se desvía del rowset caché (que amplía) está en que se comunica con otros componentes mediante el uso de XML sobre HTTP. La clase `sun.jdbc.rowset.WebRowSet` representa una implementación cliente-servidor de la interfaz `RowSet`.

```
public class WebRowSet extends CachedRowSet
```

Se espera que la implementación rowset Web incluya la clase anterior, un servlet Java y un protocolo para transmitir datos tabulares utilizando XML como formato de datos.

La situación ideal para utilizar un rowset Web es la siguiente: un cliente Web (posiblemente con apoyo Java o JavaScript) recupera una lista de resultados utilizando un objeto `sun.odbc.rowset.WebRowSet`. Cuando se invoca el método `execute()` en el rowset Web, invoca un servlet en el lado servidor (que ejecuta el acceso a la base de datos) para poblar una lista de resultados. Una vez es actualizado el rowset, el objeto `WebRowSet` envía los datos actualizados a una implementación de lado servidor del rowset como datos XML sobre el protocolo HTTP. La implementación de lado servidor actualiza entonces la base de datos. Los objetos WebRowSet utilizan cachetización incremental de datos, de modo que los datos pueden ser recuperados en fracciones desde el lado servidor. Mientras el cliente navega por el rowset Web, algunos datos adicionales son descargados desde el lado servidor.

Aunque la especificación y la implementación de esta clase están actualmente incompletas, el objetivo de esta clase `WebRowSet` es el siguiente.

En Internet, no siempre es posible conectar a un servidor de base de datos. Los proxys y los cortafuegos a menudo no lo permitirán. Incluso si las configuraciones de proxy/cortafuego lo permite, abrir una base de datos en Internet supone una amenaza a la seguridad. Para evitarlo, la implementación de lado servidor del rowset Web actuará como proxy para la base de datos. Por lo tanto, la implementación de lado cliente utiliza la implementación de lado servidor para recuperar y actualizar resultados. La principal implicación de esta arquitectura es que convierte a los clientes de base de datos en infradotados y los clientes pueden utilizar el protocolo HTTP (en lugar de protocolos de base de datos basados en TCP/IP).

Cuando el cliente Web crea una lista de resultados Web y ejecuta un comando, la implementación del lado cliente del rowset Web envía esta solicitud a la implementación de lado servidor sobre HTTP. La implementación de lado servidor conectará con la base de datos, ejecutará el comando y poblará el

`RowSet`. La implementación de lado servidor escribirá entonces los datos en un documento XML y los enviará de vuelta a la implementación de lado servidor sobre HTTP.

Este proceso se repite cuando el cliente modifica el rowset e invoca `acceptChanges()`. La implementación de lado cliente enviará una copia XML del rowset Web a la implementación de lado servidor sobre HTTP y la implementación de lado servidor realizará las actualizaciones.

Este mecanismo cuenta con HTTP y XML para el intercambio de comandos y datos entre implementaciones de lado cliente y de lado servidor. Esto elimina la necesidad de que el cliente conecte con el servidor de base de datos en Internet.

Basándonos en este enfoque, podemos esperar implementaciones más refinadas para clientes infradotados.

Resumen

El propósito de este capítulo ha sido proporcionar un análisis de la programación de bases de datos utilizando el API JDBC e introducir entonces ciertos conceptos avanzados relacionados con transacciones y bases de datos que son más relevantes en una plataforma de empresa como J2EE.

Como mencionamos al principio de este capítulo, el objetivo de la primera parte del capítulo ha sido analizar cómo ejecutar típicas operaciones SQL utilizando este API. El API JDBC es bastante extenso y ofrece más funciones de las que podemos cubrir en un capítulo. La mayoría de estas funciones (como actualizaciones de lotes, listas de resultados desplazables, etc.) son comparativamente nuevas y la mayoría de los vendedores de bases de datos sólo están empezando a apoyar estas características. Le hemos animado a que consulte la documentación del vendedor antes de pensar en implementar estas funciones.

Para resumir, en la primera parte de este capítulo, hemos analizado los siguientes puntos:

- Drivers de base de datos y cómo cargar drivers JDBC
- Crear tablas e insertar datos
- Instrucciones preparadas
- Representación entre SQL y tipos Java
- Actualizaciones de lotes
- Listas de resultados desplazables
- Puntos de salvaguardia

En la segunda parte de este capítulo, hemos analizado cuatro funciones avanzadas del API JDBC:

- Las aplicaciones cliente JDBC no necesitan estar cableadas para drivers JDBC específicos y para utilizar los URL JDBC específicos de vendedor. En su lugar, el objeto DataSource, combinado con la asociación y búsqueda de base JNDI, separa los detalles de base de datos específicos de vendedor de las aplicaciones de cliente.
- El paquete javax.sql traslada la responsabilidad de reserva de conexiones a los servidores de aplicación J2EE y drivers JDBC. Con este enfoque, la reserva de conexiones consiste sólo en configurar el servidor de aplicación; no es necesario implementar reserva de conexiones de adaptación, ya que este mecanismo ofrece un medio más fiable de reserva de conexión.
- El procesamiento de transacciones distribuidas nunca ha sido tan sencillo. Utilizando la interfaz `javax.Transaction.UserTransaction`, podemos demarcar explícitamente transacciones, mientras todavía utilizamos el API JDBC estándar para acceso a bases de datos.

- La interfaz RowSet y sus implementaciones son un complemento para el API JDBC. El objetivo de RowSet es proporcionar un acceso a los datos más flexible, similar a bean. Sun también está estudiando los objetos de datos Java, Java Data Objects (JDO), probablemente basados en los objetos de datos activos de Microsoft, Active Data Objects. Se espera que los JDO complementen el API JDBC y proporcionen acceso a base de datos a un nivel más alto sin utilizar SQL. El API JDBC está en desarrollo en la actualidad, y podemos esperar funciones más avanzadas de acceso a datos en el futuro. Consulte las novedades en http://java.sun.com/products/jdbc/.

En el siguiente capítulo analizaremos las aplicaciones Web y los contenedores que las gestionan.

5
Introducción a los contenedores Web

Como hemos visto en el capítulo 1, hay fundamentalmente dos tipos de clientes en la arquitectura J2EE: **clientes Web** y **clientes de aplicación**. Los clientes de aplicación se remontan a la arquitectura cliente-servidor tradicional en la que los clientes dirigen la interacción del usuario (normalmente vía GUI), así como la mayor parte de la lógica de aplicación (incluido el acceso a bases de datos). Por este motivo, los clientes de aplicación se conocen también como **clientes robustos**. Los clientes robustos procesan la lógica de aplicación de forma local. En una arquitectura multi-capa, los clientes de aplicación pueden delegar parte de la lógica de aplicación y acceso a bases de datos en los componentes de la capa media (como Enterprise JavaBeans). A pesar de esta distribución de la lógica de aplicación en los componentes de la capa media, los clientes de aplicación autónomos se mantienen robustos y requieren instalación en cada instalación de usuario.

Con la llegada de Internet, los clientes Web sustituyeron a muchos clientes de aplicación autónomos. El principal motor de este cambio es la naturaleza de los clientes Web. En las arquitecturas basadas en clientes Web, la capa de interacción con el usuario está separada de la capa de cliente tradicional. Los navegadores Web gestionan la interacción de usuario pero dejan el resto a las aplicaciones del lado servidor, incluido la lógica para controlar la interfaz de usuario, la interacción con los componentes de la capa media y el acceso a las bases de datos. Para un usuario final, el navegador es el cliente para todas las aplicaciones de base Web. Puesto que este tipo de cliente de objetivo general no impone ningún requisito especial (aparte del acceso a la red) en la instalación cliente, los clientes Web también son denominados **clientes infradotados**.

Los siguientes rasgos caracterizan a los clientes Web habituales:

- Un navegador Web o aplicación similar gestiona la interacción de usuario; ésta es la capa cliente.
- HTML (con JavaScript y/o DHTML) o XML (XSLT) es utilizado para crear la interfaz de usuario.

- HTTP(S) es el protocolo de intercambio de información utilizado por clientes y aplicaciones. Los programas de aplicación en el lado servidor ejecutan la lógica de aplicación en nombre de los clientes navegador.

La arquitectura J2EE ofrece un modelo de programación flexible y rico en funciones para construir aplicaciones Web dinámicas. Como hemos visto en el capítulo 1, la arquitectura J2EE proporciona **contenedores Web**, el Java Servlet API y JavaServer Pages API para la construcción y gestión de las aplicaciones Web. El contenedor Web ofrece el entorno básico de periodo de ejecución y un marco para proporcionar apoyo de periodo de ejecución para las aplicaciones Web. El servlet Java y las tecnologías JSP forman el material central para el desarrollo de las aplicaciones Web.

Este capítulo nos introduce en los contenedores Web de J2EE y en los principios básicos del desarrollo de aplicaciones Web, en tres aspectos diferentes:

- Analizaremos la anatomía básica de un contenedor Web. Empezando con una breve introducción a HTTP, para después centrarnos en los conceptos de los contenedores Web, los servlets Java y las páginas JSP. Continuaremos conociendo estos temas en capítulos posteriores.
- Haremos un recorrido por la creación de una sencilla pero completa aplicación Web.
- Analizaremos, en detalle, el funcionamiento interno de un ejemplo de aplicación Web.

Comenzaremos examinando cómo se comunican clientes y servidores utilizando el protocolo HTTP.

El protocolo HTTP

En el desarrollo de aplicaciones distribuidas, el protocolo de comunicación determina la naturaleza de clientes y servidores y la relación entre ellos. Esto también se cumple en el caso de aplicaciones de base Web. La complejidad de muchas de las características de nuestro navegador Web y del servidor Web depende del protocolo subyacente, es decir, el **Hypertext Transfer Protocol** (**HTTP**).

> **HTTP en un protocolo de aplicación (generalmente implementado sobre conexiones TCP/IP). Es un protocolo sin estado basado en solicitudes y respuestas.**

Los clientes (como navegadores Web) envían solicitudes al servidor (como el servidor Web de una memoria online) para recibir información (como descargar un catálogo) o para iniciar un procesamiento específico en el servidor (como realizar una petición).

Métodos de solicitud HTTP

Como protocolo de aplicación, HTTP define los tipos de solicitudes que los clientes pueden enviar a los servidores y los tipos de respuestas que los servidores pueden enviar a los clientes. El protocolo también especifica cómo están estructuradas estas solicitudes y respuestas.

HTTP/1.0 especifica tres tipos de métodos de solicitud: **GET**, **POST** y **HEAD**. HTTP/1.1 tiene cinco métodos de solicitud adicionales: **OPTIONS**, **PUT**, **TRACE**, **DELETE** y **CONNECT**. De estos, las solicitudes GET y POST cumplen la mayoría de los requisitos más comunes de desarrollo de aplicaciones.

El método de solicitud GET

La solicitud GET es el método de solicitud más sencillo y más frecuentemente utilizado. Se utiliza normalmente para acceder a recursos estáticos como documentos HTML e imágenes. Las solicitudes

GET pueden utilizarse para recuperar información dinámica, incluyendo parámetros de consulta en el URL de solicitud. Por ejemplo, podemos enviar un parámetro `nombre`, con un valor de `Joe`, adjuntado a un URL como http://www.domain.com?nombre=Joe. El servidor Web puede utilizar el valor de este parámetro para enviar contenido específico a un cliente.

El método de solicitud POST

El método de solicitud POST se utiliza comúnmente para acceder a recursos dinámicos. Las solicitudes POST se utilizan habitualmente para transmitir información que depende de la solicitud o cuando una gran cantidad de información compleja debe ser enviada al servidor.

La solicitud POST permite la encapsulación de mensajes multiparte en el cuerpo de la solicitud. Por ejemplo, podemos utilizar las solicitudes POST para subir archivos de texto o binarios. De un modo similar, podemos utilizar las solicitudes POST en applets para enviar objetos Java serializables o incluso bytes primigenios al servidor Web. Las solicitudes POST ofrecen una opción más amplia que las solicitudes GET en términos de contenidos de una solicitud.

Existen ciertas diferencias entre las solicitudes GET y POST. Con las solicitudes GET, los parámetros de solicitud son transmitidos como una cadena de consulta adjunta el URL de solicitud. En el caso de las solicitudes POST, los parámetros de solicitud son transmitidos en el cuerpo de la solicitud. Esto tiene dos consecuencias. En primer lugar, puesto que la solicitud GET contiene la información completa de solicitud adjunta el mismo URL, permite a los navegadores guardar la dirección de la página y volver a visitarla más tarde. Dependiendo del tipo y de la sensibilidad de los parámetros de la solicitud, esto puede ser o no deseable. En segundo lugar, algunos servidores pueden imponer restricciones sobre la longitud del URL de solicitud. Esto limita la cantidad de información que puede ser adjuntada al URL de solicitud. Observe que HTTP/1.1 no impone ningún límite superior en la longitud. Sin embargo, no podemos garantizar los servidores/clientes HTTP se adapten a excesivas longitudes.

El método de solicitud HEAD

Un cliente envía una solicitud HEAD cuando sólo desea ver las cabeceras de una respuesta, como `Contenido-Tipo` o `Contenido-Longitud`.

Las solicitudes de método GET y HEAD no deberían ser programadas para modificar la información en el servidor. Estas solicitudes pueden ser presentadas múltiples veces sin cambiar los datos en el servidor y como tales están destinadas únicamente a la *recuperación* de información.

Junto con el tipo de solicitud, la aplicación de cliente también especifica el recurso que necesita como parte de la cabecera de la solicitud. Por ejemplo, cuando introducimos http://java.sun.com/index.html. En la barra de dirección de nuestro navegador, el navegador envía la solicitud GET al servidor Web identificado por `java.sun.com` para el recurso `index.html`. En el léxico HTTP, un identificador de recursos uniformes, **Uniform Resource Identifier (URI)**, especifica un recurso. El URI es el URL, excluyendo el nombre del dominio. En nuestro ejemplo, el recurso es el archivo `index.html` localizado en el documento raíz del servidor Web que sirve al dominio `java.sun.com`.

La respuesta HTTP

Cuando un servidor recibe una solicitud HTTP, responde con el estado de la respuesta y alguna meta-información que describe la respuesta. Todos estos elementos son parte de la **cabecera de respuesta**. Excepto en el caso de la solicitud HEAD, el servidor también envía el contenido que corresponde al recurso especificado en la solicitud. Si un navegador envía una solicitud a http://java.sun.com/index.html, recibe el contenido del archivo `index.html` como parte del mensaje.

Los campos de cabecera del contenido de la respuesta contienen información útil que los clientes pueden querer comprobar en determinadas circunstancias. Los campos habituales de cabecera incluyen: `Fecha`, `Contenido-Tipo` y `Expira`. Por ejemplo, el campo de cabecera `Expira` de una página puede ser fijado en la misma fecha que el campo de cabecera `Fecha` para indicar que los navegadores no deben cachetizar la página. Las aplicaciones Web con contenido sensible al tiempo pueden necesitar establecer estos campos.

Los servidores y clientes que se comunican utilizando HTTP hacen uso de **Multi-Purpose Internet Mail Extensions (MIME)** para indicar el tipo de contenido de los cuerpos de solicitud y respuesta. Los tipos MIME incluyen `texto/html` e `imagen/gif`. La primera parte de la cabecera indica el tipo de datos (por ejemplo, texto e imagen), mientras que la segunda parte indica la extensión estándar (como `html` para texto y `gif` para imagen). MIME es una extensión de un protocolo de e-mail diseñado para permitir el intercambio de diferentes tipos de archivos de datos. Los servidores HTTP utilizan cabeceras MIME al principio de cada transmisión y los navegadores utilizan esta información para decidir cómo analizar y generar el contenido. Los navegadores también utilizan cabeceras MIME al transmitir datos en el cuerpo de las solicitudes, para describir el tipo de datos que se envía. Por ejemplo, el tipo de codificación MIME por defecto para solicitudes POST es `application/x-www-form-urlencoded`.

Las características clave de HTTP que es preciso comprender son:

- HTTP es un protocolo sin estado muy sencillo y ligero.
- El cliente siempre inicia la solicitud. El servidor nunca puede realizar una conexión de retrollamada al cliente.
- HTTP requiere que el cliente establezca conexiones previas a cada solicitud y que el servidor cierre la conexión después de enviar la respuesta. Esto garantiza que un cliente no puede mantener una conexión después de recibir la respuesta. Tanto el cliente como el servidor puede finalizar prematuramente una conexión.

HTTP tenía como objetivo original servir información estática, por ello necesitamos considerar ciertos temas adicionales para poder construir aplicaciones dinámicas que comuniquen con clientes utilizando HTTP. En este capítulo, veremos cómo enfrentarse a algunos de estos temas utilizando contenedores Web J2EE.

Contenedores Web y aplicaciones Web

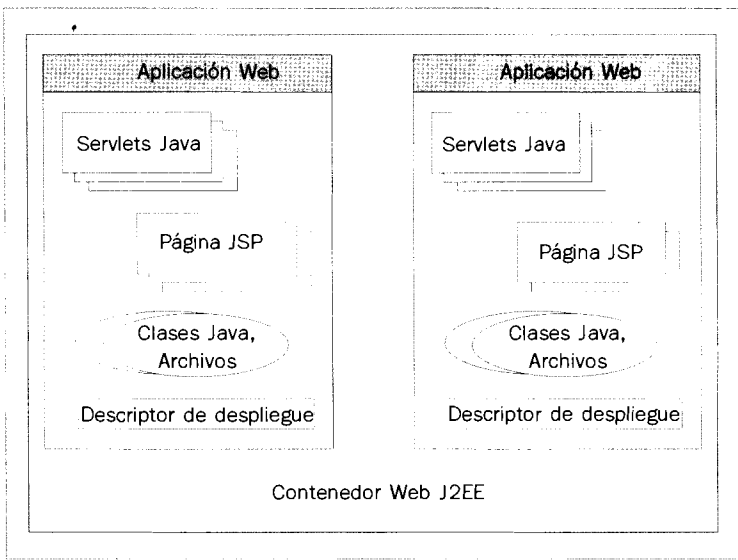
Las aplicaciones Web son aplicaciones de lado servidor. Los requisitos más esenciales para el desarrollo de aplicaciones de lado servidor son los siguientes:

- **Un modelo de programación y un API**
  Un modelo de programación para aplicaciones de lado servidor especifica cómo desarrollar aplicaciones.
- **Soporte de periodo de ejecución de lado servidor**
  Esto incluye apoyo para servicios de red aplicables y periodo de ejecución para ejecutar las aplicaciones.
- **Soporte de implementación**
  La implementación es el proceso de instalación de la aplicación en el servidor. Este proceso también incluye la configuración de los componentes de la aplicación, como especificar los parámetros de inicialización y especificar cualquier base de datos.

Para cumplir estos requisitos al construir y ejecutar aplicaciones Web, la especificación J2EE proporciona los elementos siguientes:

- **Servlets Java y Páginas JavaServer**
  Los servlets Java y las Páginas JavaServer (JSP) son los bloques de construcción para el desarrollo de aplicaciones Web en J2EE. En el léxico de J2EE, los servlets Java y las páginas JSP son conocidos como **componentes Web.**
- **Aplicaciones Web**
  Una aplicación Web es una colección de servlets Java, páginas JSP, clases ayudantes y librerías de clases, recursos estáticos como documentos HTML, XHTML o XML, e imágenes.
- **Un contenedor Web para albergar aplicaciones Web**
  Un contenedor Web es esencialmente un periodo de ejecución Java que proporciona una implementación del API Servlet Java y facilidades para las páginas JSP. El contenedor Web es responsable de inicializar, invocar y gestionar el ciclo de vida de los servlets Java y las Páginas JavaServer.
- **Una estructura de empaquetado y descriptores de implementa ción**
  La especificación J2EE define una estructura de empaquetado para las aplicaciones Web. La especificación también define un descriptor de implementación para cada aplicación Web. El descriptor de implementación es un archivo HTML que permite la adaptación de aplicaciones Web en el tiempo de despliegue.

El siguiente diagrama muestra el contenedor Web de J2EE con dos aplicaciones desplegadas en el mismo contenedor:



En esta sección, analizaremos algunos de los rasgos básicos de servlets, páginas JSP y descriptores de implementación. El objetivo de este análisis es ofrecer una visión completa de lo que supone el desarrollo de aplicaciones Web. En posteriores capítulos, examinaremos el proceso con más detalle.

Servlets Java

Los servlets Java permiten que la lógica de aplicación sea integrada en el proceso de solicitudes y respuestas HTTP y que sea especificada en el API Java Servlet Especificación 2.3 disponible en http://java.sun.com/products/servlet/.

Los servlets Java son programas pequeños de lado servidor independientes de la plataforma que amplían programáticamente la funcionalidad del servidor Web. En API Java Servlet proporciona un marco sencillo para construir aplicaciones en estos servidores Web.

Para comprender mejor la función de la lógica de aplicación en el proceso de solicitud-respuesta, consideremos un servidor de correo de base Web. Cuando nos registramos en nuestro servidor de correo de base Web favorito, el servidor debe poder enviar una página con vínculos a nuestro correo, nuestras carpetas de correo, nuestro libro de direcciones, etc. Esta información es dinámica, en el sentido en que cada usuario espera ver su buzón individual. Para generar este contenido, el servidor debe ejecutar una lógica de aplicación compleja para recuperar el correo y componer una página.

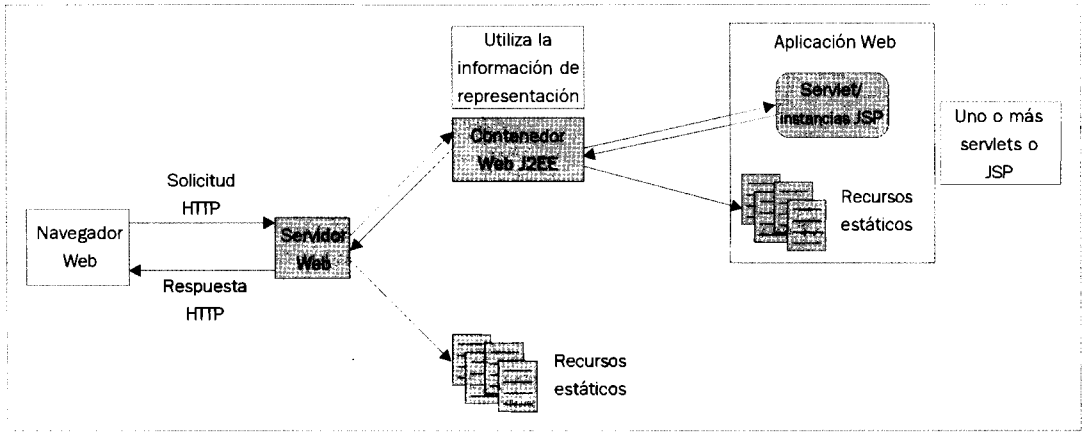
Los clientes pueden enviar información de contexto o específica del cliente al servidor en las solicitudes pero, ¿cómo puede el servidor decidir cómo debe ser generado el contenido? HTTP no define un medio estándar para integrar la lógica de aplicación durante la fase de generación de respuestas. No existe un modelo de programación especificado para estas tareas. HTTP define cómo pueden los clientes solicitar información y cómo pueden responder los servidores pero no interviene en cómo puede ser generada la respuesta.

Aquí es donde se hacen evidentes los beneficios de tecnologías de lado servidor tales como los servlets Java y las páginas JavaServer. Con estas tecnologías, podemos integrar la lógica de la aplicación de adaptación durante el procesamiento de respuestas y las etapas de generación de respuestas.

Los servlets Java no son aplicaciones invocadas por usuarios. En su lugar, el contenedor Web en el que se despliega la aplicación Web que contiene los servlets invoca los servlets basándose en solicitudes entrantes HTTP. Cuando un servlet ha sido invocado, el contenedor Web envía la información de la solicitud entrante al servlet, de modo que el servlet pueda procesarla y generar una respuesta dinámica. El contenedor Web actúa como interfaz con el servidor Web aceptando solicitudes para servlets y transmitiendo respuestas de vuelta al servidor Web.

En comparación con CGI y extensiones de servidor de propiedad como NSAPI o ISAPI, el marco de servlet proporciona una mejor abstracción del paradigma de solicitud y respuesta HTTP especificando un API de programación para encapsular solicitudes y respuestas. Además, los servlets tienen todas las ventajas del lenguaje de programación Java, incluyendo la independencia de la plataforma. Las aplicaciones basadas en servlets Java pueden ser desplegadas en cualquier servidor Web con contenedores Web incorporados (dentro del proceso) o basados en conectores (fuera del proceso), con independencia del sistema operativo y de la plataforma de hardware.

Para entender cómo un servlet interactúa con un servidor Web a través de un contenedor Web, consideremos el proceso básico de invocación con que el servidor Web está recibiendo una solicitud HTTP. Como hemos visto antes, el protocolo HTTP está basado en un modelo de solicitudes y respuestas. Un cliente conecta con un servidor Web y envía una solicitud HTTP en la conexión. Basado en un URL de solicitud, la siguiente secuencia de eventos tiene lugar en una secuencia típica (las flechas hacia la derecha indican solicitudes y las flechas hacia la izquierda indican respuestas):



- El servidor Web tiene que descifrar si la solicitud entrante corresponde a una aplicación Web en el contenedor Web. Esto requiere un entendimiento implícito entre el servidor Web y el contenedor Web. Los contenedores Web utilizan la noción de **contexto de servlet** para identificar las aplicaciones Web. Este contexto puede ser especificado cuando la aplicación es desplegada sobre el contenedor.

- Si el contenedor Web puede manejar la solicitud, el servidor Web la delega en él. Podemos pensar en este proceso como el servidor Web invocando un método en el contenedor Web, pasándolo a la información de la solicitud.

- Una vez que el contenedor Web ha recibido la solicitud, decide qué aplicación debe manejarla. En una aplicación Web de J2EE, una solicitud HTTP puede ser representada en un servlet, una página JSP o cualquier recurso estático. Esta representación está basada en patrones URL. Los recursos estáticos incluyen páginas HTML/XML, imágenes y archivos de clase applet. Todos estos recursos son parte de la aplicación Web. Cuando empaquetamos y desplegamos una aplicación Web, especificamos esta información de representación. El contenedor Web utiliza esta información de representación para representar cada solicitud entrante en un servlet, página JSP o recurso estático. Si el recurso es representado en un recurso estático, el contenedor Web simplemente pasa el recurso tal y como está al servidor Web. Esto conforma el cuerpo de la respuesta que el servidor Web envía al navegador.

- Basándose en la información de representación, el contenedor Web determina si la solicitud debe ser manejada por un servlet. Si la solicitud debe ser manejada por un servlet, el contenedor crea o localiza una instancia de servlet y delega la solicitud. En próximos capítulos, veremos cómo el contenedor Web lleva a cabo este proceso y qué factores lo gobiernan.

- Cuando el contenedor delega la solicitud en un servlet, también pasa objetos que encapsulan la solicitud HTTP y la respuesta HTTP en la instancia servlet. Para el servlet, estos objetos representan las corrientes de solicitud y respuesta del navegador. El servlet puede leer la información de la solicitud y escribir una respuesta para estas corrientes. Para escribir una respuesta, el servlet puede utilizar los métodos `println()` del objeto asociado `java.io.PrintWriter`. Esto es equivalente a escribir contenido en la conexión ya abierta desde el navegador Web.

Hay dos pasos necesarios para construir y desplegar una aplicación basada en un servlet:

- Escribir los servlets con la lógica de aplicación requerida. Ésta es la fase de despliegue.

- Proporcionar un contexto y la información opcional de representación de patrón URL durante la fase de despliegue. Esta información de representación es utilizada para identificar el servlet apropiado para manejar una solicitud.

Páginas JavaServer

La generación de contenido dinámico puede conseguirse mediante generación de contenido programático o generación de contenido basado en una plantilla. Los servlets Java están dentro de la primera categoría y las páginas JavaServer (JSP) pertenecen a la segunda. Las páginas JSP están especificadas en la especificación 1.2 de JavaServer Pages, disponible en http://java.sun.com/products/jsp/

Las páginas JavaServer son una extensión de la tecnología de servlets Java. Sin embargo, en contraste con los servlets, que son puros programas Java, las páginas JSP son documentos de base textual. Una página JSP contiene dos partes:

- HTML o XML para el contenido estático
- Etiquetas y scriptlets JSP escritos en Java para encapsular la lógica que genera el contenido dinámico

Puesto que una página JSP proporciona una representación general de contenido que puede producir múltiples vistas dependiendo del resultado de las etiquetas y scriptlets JSP, una página JSP actúa como una plantilla para producir contenido.

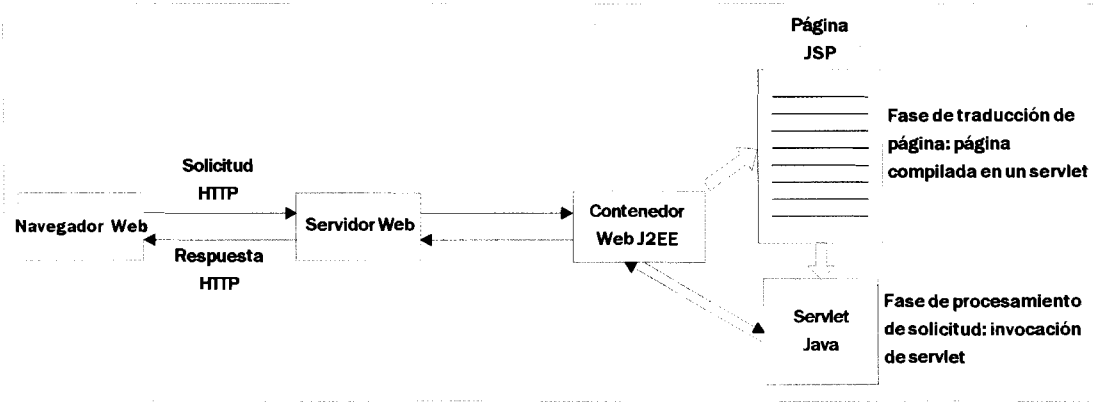
Una plantilla es una página de anotación (como HTML) con guardasitios especiales (etiquetas y scriptlets JSP) integrados. Estos guardasitios contienen información de procesamiento para el procesador de plantilla (o generador de contenido). En una página JSP, la anotación usual nos permite definir la estructura estática y el contenido de una página. Las etiquetas y scriptlets adicionales integrados en la página nos permiten incluir lógica de programación para ser ejecutada durante la generación de páginas.

La ventaja clave de esta tecnología es que ayuda a mantener el diseño del contenido y las actividades de desarrollo en combinación libre con el diseño y desarrollo de la lógica de aplicación. Si desarrollamos el contenido únicamente con servlets, las dos actividades estarán estrechamente entrelazadas. Esto no es deseable porque estas aplicaciones son más difíciles de mantener.

En algunas tecnologías controladas por plantilla como Active Server Pages (ASP), las plantillas son evaluadas en el periodo de ejecución. Es decir, el procesador de plantilla convierte dinámicamente las etiquetas especiales en contenido normal. Éste es un proceso de periodo de ejecución y cada vez que se solicita una página, el procesador de plantilla interpreta la plantilla una y otra vez.

A diferencia de ASP, la tecnología JSP está basada en la compilación de páginas. En lugar de interpretar las páginas JSP, el contendor Web la convierte en una clase de servlet y la compila. Este proceso ocurre habitualmente cuando el contenedor Web invoca una página JSP por primera vez en respuesta a una solicitud o en el arranque del contenedor. Los contenedores Web también nos permiten precompilar páginas JSP en servlets. La mayoría de los contenedores disponibles en la actualidad repiten este proceso siempre que se modifica la página JSP. Esto se denomina fase de traducción de página. El contenedor invoca el servlet generado/compilado para solicitudes posteriores a la JSP. Ésta es la fase de procesamiento de solicitud.

En la figura siguiente se muestran las diferentes fases (la flecha blanca representa el proceso de compilación de página):



Una vez compilada la página JSP en un servlet, el resto del procesamiento de la solicitud y la generación de respuesta es igual al proceso descrito previamente en el caso de los servlets.

Esta arquitectura nos lleva a aplicaciones muy útiles y flexibles utilizando páginas JSP y servlets:

- Podemos utilizar sólo servlets para lógica de aplicación y generación de contenido
- Podemos utilizar también páginas JSP para lógica de aplicaciones y generación de contenido
- Podemos combinar las anteriores con la lógica de aplicación manejada por servlets y generación de contenido manejada por páginas JSP

Descriptores de despliegue

Los descriptores de despliegue son una parte integrante de las aplicaciones Web J2EE. Ayudan a gestionar la configuración de las aplicaciones Web una vez desplegadas. Para los contenedores Web, el descriptor de despliegue es un archivo XML llamado `Web.xml` almacenado en el directorio `/WEB-INF` de la aplicación Web. La especificación de Java Servlets proporciona una definición de tipo de documento (DTD) para el descriptor de despliegue. Éste está disponible en http://java.sun.com/dtd/Web-app_2_3.dtd.

Un descriptor de despliegue tiene varios propósitos:

- **Inicialización de parámetros para servlets y aplicaciones Web**
  Esto nos permite minimizar el nivel de inmodificabilidad de los valores de inicialización en nuestras aplicaciones Web. Por ejemplo, si nuestro servlet requiere acceso a una base de datos, el mejor lugar para especificar los detalles, como el registro y la contraseña, es el descriptor de despliegue. Esto nos permite configurar nuestra aplicación sin tener que recompilar los servlets.
- **Definiciones de servlet/JSP**
  Cada servlet o página JSP precompilada utilizada en nuestra aplicación Web debe estar definida en el descriptor de despliegue. Esta entrada incluye el nombre del servlet o JSP, la clase del servlet o JSP, y una descripción opcional.
- **Representaciones de servlet/JSP**
  Los contenedores Web utilizan esta información para representar solicitudes entrantes a servlets y páginas JSP. Conocerá más detalles sobre estas representaciones en el capítulo 9.
- **Tipos MIME**
  Puesto que cada aplicación Web puede contener varios tipos de contenido, podemos especificar los tipos MIME para cada tipo en el descriptor de despliegue.

- **Seguridad**
  Podemos conseguir el control de acceso para nuestra aplicación utilizando el descriptor de despliegue. Por ejemplo, podemos especificar si nuestra aplicación Web requiere un registro y, si es así, cómo debería ser esa página de registro y qué función debe tener el usuario.

El descriptor de despliegue puede también puede ser utilizado para adaptar otros elementos incluido las páginas bienvenida, páginas de error y configuración de sesión. Aprenderemos más sobre estos temas en el capítulo 9.

Estructura de aplicaciones Web

Una aplicación Web consta de cuatro partes:

- Un directorio público
- Un archivo `WEB-INF/Web.xml`
- Un directorio `WEB-INF/classes`
- Un directorio `WEB-INF/lib`

El área pública es la raíz de la aplicación, excluyendo el directorio `WEB-INF`. Si está familiarizado con servidores Web como Apache, éste es equivalente al directorio `htdocs` donde tendríamos todos nuestros archivos HTML. El contenedor Web puede servir a cualquiera de los archivos del área pública.

El directorio `WEB-INF` es un área privada y el contenedor no servirá contenidos de este directorio a los usuarios. Los archivos de éste están destinados únicamente al uso del contenedor. `WEB-INF` contiene, entre otras cosas, el descriptor de despliegue, un directorio `classes` y un directorio `lib`. El directorio `classes` es utilizado para almacenar servlets compilados y otras clases de utilidad. Si una aplicación ha empaquetado archivos JAR (por ejemplo, un API tercero empaquetado como un archivo JAR), pueden ser copiados en el directorio `lib`. El contenedor Web utiliza estos dos directorios para ubicar servlets y otras clases dependientes.

Tipos de contenedores Web

Existen fundamentalmente tres formas de configurar contenedores Web. Recordemos del Capítulo1 que un contenedor Web bien implementa los servicios HTTP subyacentes o delega estos servicios a servidores Web externos:

- **Contenedor Web en un servidor de aplicación J2EE**
  La mayoría de los servidores comerciales de aplicaciones J2EE, como WebLogic, Inprise Application Server, iPlanet Application Server, WebSphere Application Server incluyen en la actualidad contenedores Web integrados.

- **Contenedores Web integrados en servidores Web**
  Éste es el caso de los servidores Web Java puro que contienen contenedores Web integrados. Jakarta Tomcat (en http://jakarta.apache.org), que es la implementación de referencia del contenedor Web, también está dentro de esta categoría. Tomcat incluye un servidor Web junto con un contenedor Web.

- **Contenedor Web en un periodo de ejecución independiente**
  En el universo exterior a J2EE, éste es el escenario más común. Los servidores Web como Apache o Microsoft IIS requieren un periodo de ejecución distinto Java para ejecutar servlets y un plug-in

de servidor Web para integrar el periodo de ejecución con el servidor Web. Los sistemas de procesamiento de servlet/JSP comercialmente disponibles como JRun de Allaire y ServletExec de New Atlanta (ahora parte de la familia de productos Unify's eWave) ofrecen plug-ins para integrarlos con los servidores Web.

La elección de un tipo de contenedor depende por completo de los requisitos de nuestra aplicación. Por ejemplo, si sólo estuviéramos interesados en construir una nueva aplicación Web de base Java, elegiríamos la segunda o la tercera configuración. Sin embargo, con la unificación de las normas de procesamiento Web y de empresa mediante J2EE, la primera configuración se está convirtiendo cada vez en la más común.

Una sencilla aplicación Web

Ahora que hemos visto una amplia descripción de la arquitectura de contenedores Web de J2EE y de aplicaciones Web, construyamos una aplicación Web de modo que podamos hacer un recorrido por los aspectos básicos del desarrollo de aplicaciones Web con J2EE.

Nuestra aplicación Web hará dos cosas:

- Sugerir al usuario un nombre una dirección de e-mail
- Imprimir un saludo basado en la hora del día

Aunque existen diferentes combinaciones de servlets y páginas JSP que podrían utilizarse para desarrollar esta aplicación, utilizaremos la siguiente:

- Una página HTML con un formulario para recabar el nombre y la dirección de e-mail del usuario
- Un servlet para procesar la solicitud y generar HTML para mostrar el mensaje de bienvenida
- Un descriptor de despliegue

Una vez hayamos completado este ejercicio, comprobará lo sencillo e intuitivo que es el proceso de construcción de una aplicación Web.

Prepara el contenedor Web

Para construir, desplegar y ejecutar nuestra aplicación Web necesitamos un contenedor Web (anteriormente conocido como motor de servlet/JSP) que cumpla con la especificación Java Servlet 2.3. hay diferentes productos comerciales disponibles en el mercado pero en éste y en los siguientes capítulos recomendamos que utilice la implementación de referencia J2EE de http://java.sun.com.

Crear el archivo HTML

Cree un archivo HTML con el siguiente contenido y guárdelo como `%BOOK_HOME%\Ch05\greeting\index.html`.

```
<html>
   <head>
      <title>ProJava Registration</title>
   </head>
```

```
    <body>
      <h1>Welcome</h1>
      <form action="/greeting/servlet/GreetingServlet" method="POST">
        <p>Your Name <input type="text" size="40" name="name" /></p>
        <p>Your Email <input type="text" size="40" name="email" />
        <input type="submit" VALUE="Submit" /></p>
      </form>
    </body>
</html>
```

Observe el contenido de la etiqueta `<form>`. Esta etiqueta tiene una solicitud POST para un formulario que recoge los valores para los parámetros `nombre` y `email`.

Crear un servlet

El siguiente paso es crear un servlet. Este servlet procesa la solicitud HTTP POST presentada desde index.html. Cree una clase Java llamada `GreetingServlet` y guárdela como `%BOOK_HOME%\Ch05\greeting\GreetingServlet.java`.

No es necesario que nos preocupemos por poner nuestro servlet en el directorio classes en esta fase porque la herramienta que utilizaremos para construir nuestra aplicación Web realizará este proceso por nosotros.

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Calendar;
import java.util.GregorianCalendar;
public class GreetingServlet extends HttpServlet {

  protected void doPost (HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException {
    String name = request.getParameter("name");
    String email = request.getParameter("email");
    String message = null;
    GregorianCalendar calendar = new GregorianCalendar();
    if(calendar.get(Calendar.AM_PM) == Calendar.AM) {
      message = "Good Morning";
    } else {
      message = "Good Afternoon";
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<html>");
    out.println("<body>");
    out.println("<p>" + message + ", " + name + "</p>");

    out.println("<p>  Thanks for registering your email (" + email +
                ") with us.</p>");
    out.println("<p> - The Pro Java Team. </p>");

    out.println("</body>");
```

```
        out.println("</html>");

        out.close();
    }
}
```

Para poder compilar el servlet, debe tener `j2ee.jar` (que se encuentra en `%J2EE_HOME%\lib\`) en su `CLASSPATH`. Ejecute el siguiente comando desde `%BOOK_HOME%\Ch05\greeting\` para compilar el servlet:

```
javac -classpath %J2EE_HOME%\lib\j2ee.jar;. GreetingServet.java
```

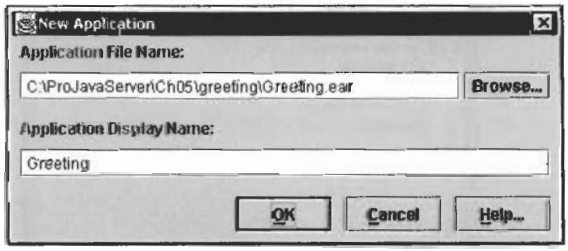
Construir la aplicación Web

En este punto, debe tener los siguientes archivos:

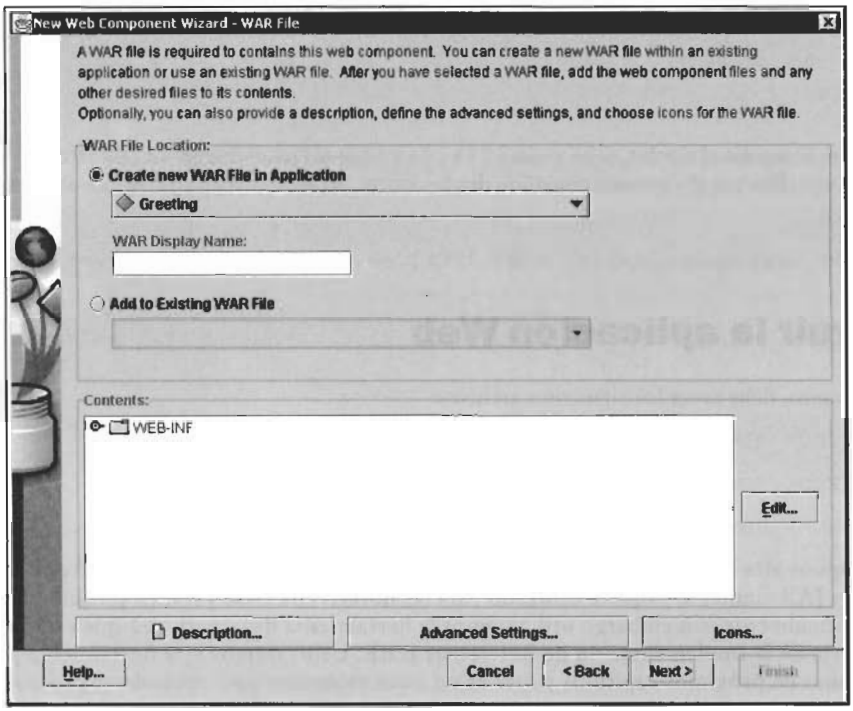
- `%BOOK_HOME%\Ch05\greeting\index.html`
- `%BOOK_HOME%\Ch05\greeting\GreetingServlet.java`
- `%BOOK_HOME%\Ch05\greeting\GreetingServlet.class`

Los componentes Web en J2EE están empaquetados en un archivo **WAR, Web Application Archive**, que es un JAR similar al paquete utilizado para las librerías de clase Java. Es posible crear archivos WAR manualmente, sin embargo utilizaremos la herramienta `deploytool` que está incorporada como parte de la Implementación de Referencia J2EE. Utilizaremos esta herramienta para crear aplicaciones en próximos capítulos y éste es un buen momento para aprender algunas de sus características básicas.

Arranque la herramienta de implementación de J2EE ejecutando el archivo `deploytool.bat` en el directorio `%J2EE_HOME%\bin`. Cree una nueva aplicación utilizando el botón `New Application` o seleccionando `New | Application` del menú `File`. Nombre a la aplicación `Greeting`:



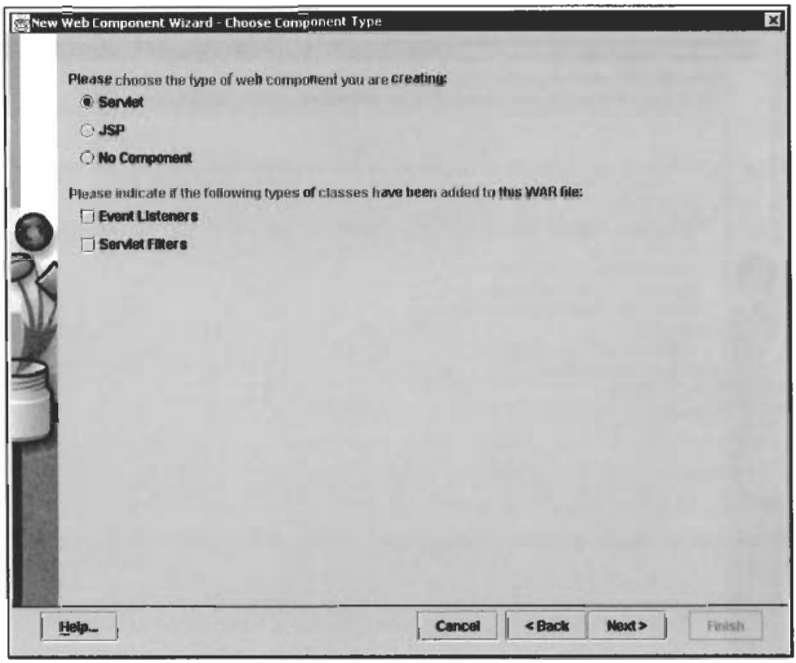
Nuestra aplicación será parte de esta aplicación Enterprise. Seleccione la aplicación `Greeting` en el árbol de la izquierda y añada un nuevo `Web Component` utilizando el menú `File` (Archivo). Salte la primera pantalla de `New Component Wizard` (`Asistente para nuevo componente`) hasta que llegue a la pantalla `WAR File`:



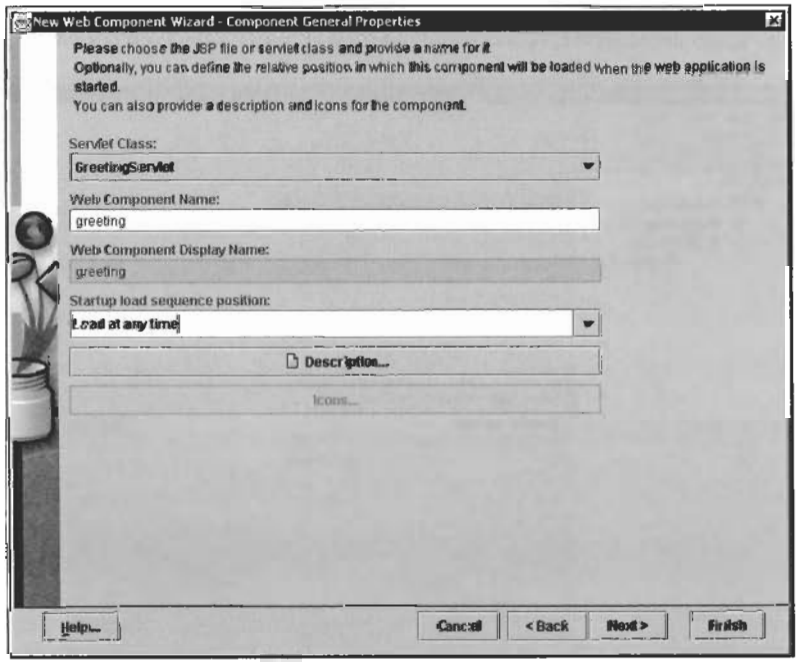
Pulse el botón `Edit` (Editar) y añada `GreetingServlet.class` y `index.htnl` al componente Web:



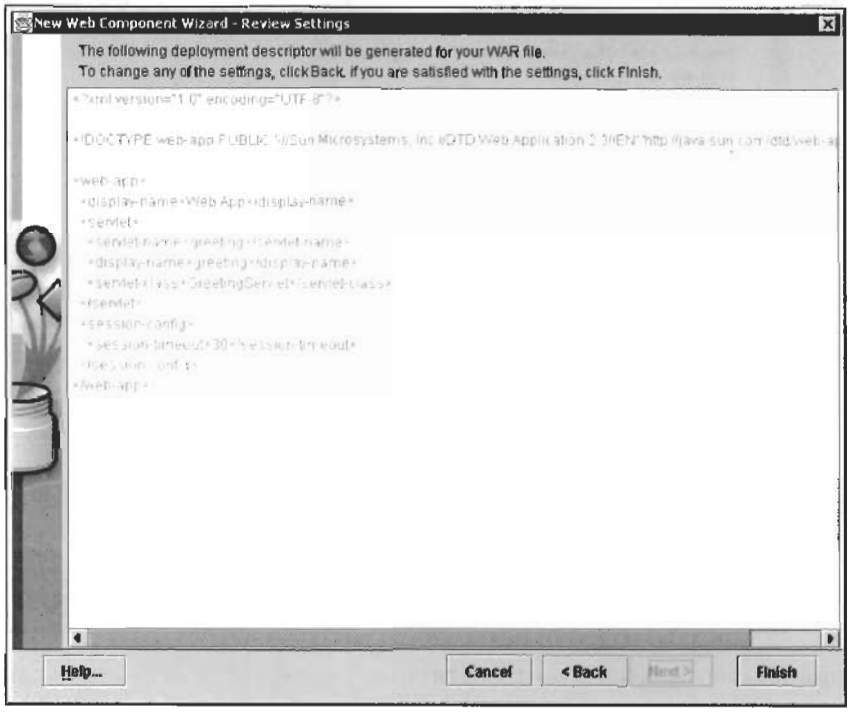
Avance hasta la siguiente pantalla del asistente y seleccione desplegar un solo `Servlet`:



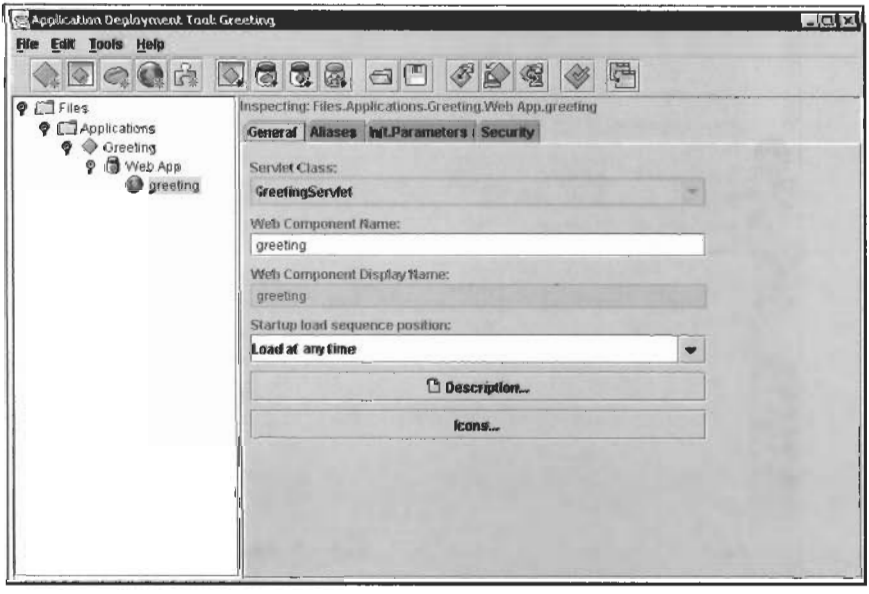
Vaya a la siguiente pantalla. Nombre al servlet `greeting` y elija el archivo de clase `GreetingServlet`:



Esto es todo lo que necesitamos para establecer el asistente. Puede pulsar el botón `Finish` (Finalizar) ahora o seguir hasta el final del asistente para ver el descriptor de despliegue:



Una vez haya salido del asistente, verá el componente Web recién creado en la ventana principal:



Esto completa la creación de nuestra primera aplicación Web.

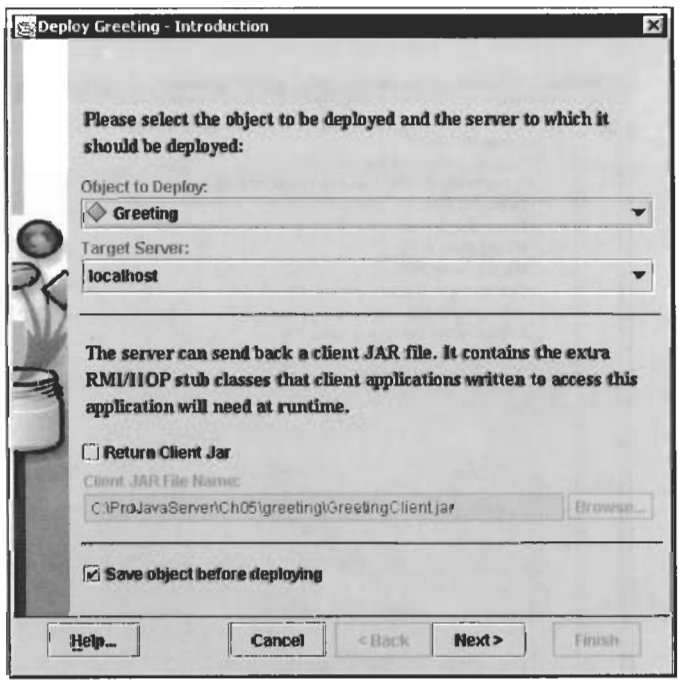
Desplegar la aplicación Web

El siguiente paso es **desplegar** nuestra aplicación Web recién creada sobre un servidor de modo que los navegadores clientes puedan acceder a ella.

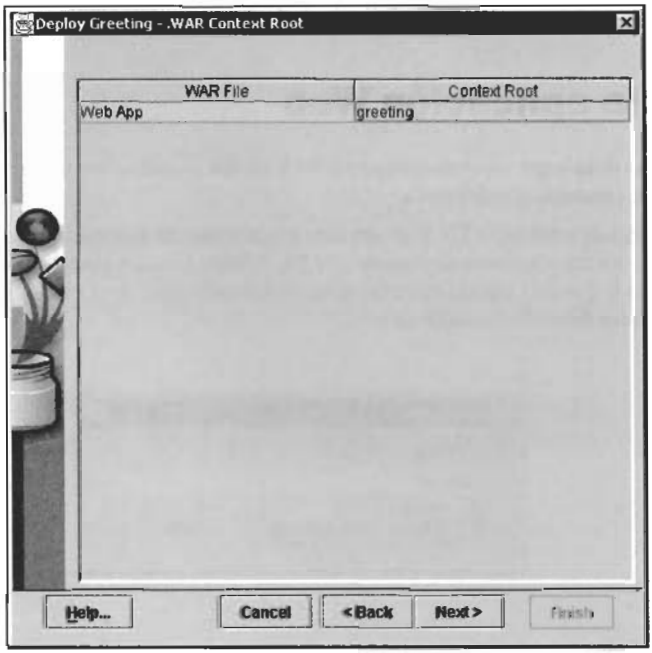
Si la implementación de referencia J2EE el servidor no está siendo ejecutada, necesita comenzar ejecutando `j2ee -verbose` desde dentro de `%J2EE_HOME%\bin`. Conecte entonces la herramienta de despliegue al servidor que está siendo ejecutado, seleccionando `Add Server` del menú `File` (Archivo) y añadiendo un servidor llamado `localhost`:



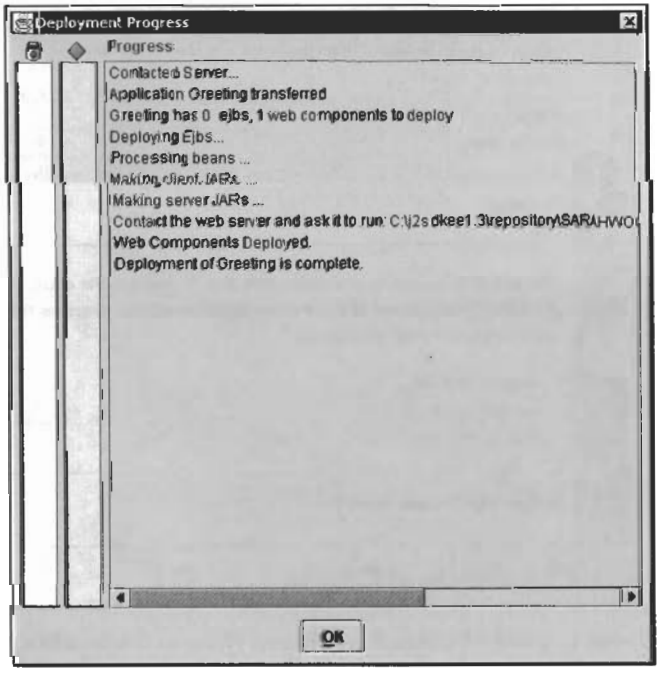
Ahora podemos desplegar el componente Web. Selección `Deploy` (Desplegar) del menú `Tools` (Herramientas) para iniciar el asistente de despliegue:



Asegúrese de que `Greeting` está seleccionado mientras el objeto es desplegado y pulse `Next` (Siguiente):



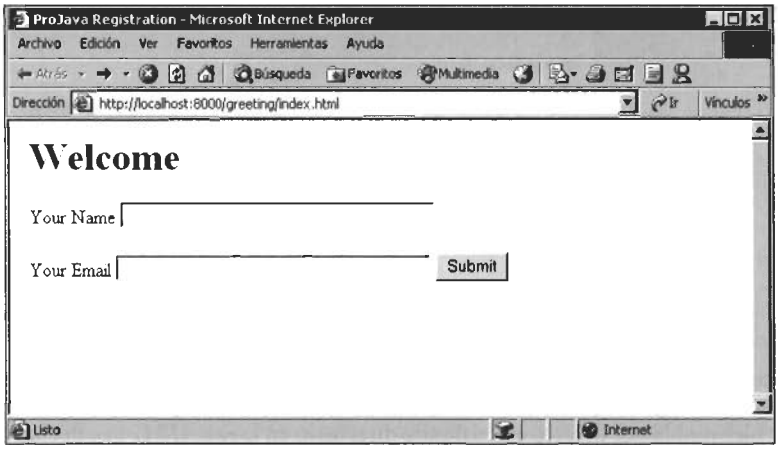
Fije `Context Root` en `greeting`; así se configura el URL para acceder a la aplicación. Seleccione `Next` y después `Finish` para iniciar el proceso de despliegue:



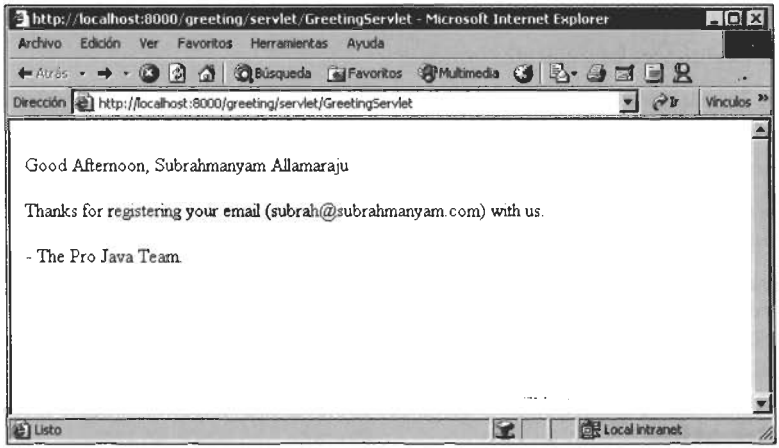
Ahora estamos preparados para probar nuestra aplicación.

Ejecutar la aplicación Web

El servidor ya debe estar ejecutándose; abra su navegador y vaya a http://localhost8000/greeting/index.html, donde debe ver algo así:



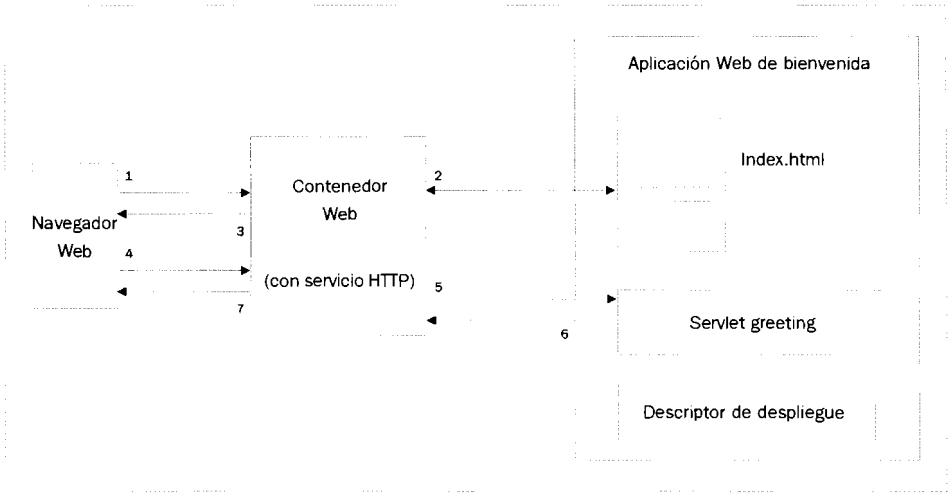
Introduzca su nombre y dirección de e-mail, y haga clic en `Submit`. Su navegador debe mostrar la siguiente bienvenida del equipo `The Pro Java Server`. El verdadero mensaje de bienvenida depende del momento del día en que ejecute esta aplicación:



La siguiente fase es comprender cómo funciona esta aplicación.

Cómo funciona la aplicación

La figura que aparece a continuación muestra lo que ocurre entre el navegador Web y la aplicación Web. Las flechas numeradas indican la secuencia en la que solicitudes y respuestas son procesadas:



En respuesta a la primera solicitud que el navegador realiza al servidor (http://localhost:8000/greeting/index.html), el contenedor Web sirve el index.html bajo \greeting\index.html.

En la actual configuración, la ruta URL `greeting` representa una aplicación Web situada en `\webapps\greeting`. En general, los servidores y contenedores Web consideran `index.html` como el archivo de bienvenida por defecto en cualquier directorio en la ruta URL. Las flechas numeradas (1), (2) y (3) muestran esta secuencia.

La segunda solicitud (http://localhost:8080/greeting/servlet/GreetingServlet) realizada el contenedor Web es la solicitud POST HTTP con los parámetros nombre y email. Basándose en la ruta `greeting/servlet` y la definición del servlet del descriptor de despliegue, el contenedor Web delega esta solicitud en `GreetingServlet`. (Examinaremos el descriptor de despliegue muy pronto.)

El servlet realiza entonces las siguientes tareas:

- Extrae los parámetros de solicitud: Esto conlleva extraer los parámetros `"nombre"` y `"email"` de la solicitud
- Ejecuta la lógica de aplicación: En nuestro ejemplo, esto supone generar un mensaje de bienvenida
- Genera la respuesta: Después de procesar la solicitud, el servlet genera un documento HTML con el mensaje de bienvenida

El navegador recibe la respuesta así generada por el servlet. Esta secuencia aparece representada por las flechas numeradas (4), (5), (6) y (7). Brevemente, así es como los contenedores Web delegan solicitudes en los servlet en las aplicaciones Web. Aprenderemos más detalles sobre este proceso en los capítulo 6 y 7.

El servlet Greeting

No se preocupe por comprender cada línea de código en `GreetingServlet.java`. Nuestro objetivo es simplemente obtener una visión general de las funciones de un servlet.

Importar los paquetes de servlet

El API Java Servlet consiste en dos paquetes: `javax.servlet`, y `javax.servlet.http`. Observe que todos los paquetes `javax` son parte de Java 2 SDK, Enterprise Edition.

De estos dos paquetes, el paquete `javax.servlet` contiene clases e interfaces de servlet que son independientes de HTTP. El resto de las clases que son específicas para HTTP forman parte del paquete `javax.servlet.http`.

Además de los dos anteriores, la clase `GreetingServlet` también necesita importar clases de los paquetes`java.io`y`java.util`:

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Calendar;
import java.util.GregorianCalendar;
```

Declaración de clase

Todos los servlets son necesarios para implementar la interfaz `javax.servlet.Servlet`. Sin embargo, para los servlets en aplicaciones Web, la clase `javax.servlet.http.HttpServlet` proporciona una implementación de esta interfaz. Por lo tanto, la clase `GreetingServlet` amplía la clase `HttpServlet`:

```
public class GreetingServlet extends HttpServlet {
   ...
}
```

Revisar la solicitud POST HTTP

En el servlet `GreetingServlet`, el método `doPost()` maneja la solicitud POST HTTP. Este método toma un objeto `HttpServletRequest` que encapsula la información contenida en la solicitud y un objeto`HttpServletResponse` que encapsula la respuesta HTTP.

Nuestra implementación del método `doPost()` incluye dos tareas: extraer los parámetros `FORM` de la solicitud HTTP y generar la respuesta.

Extraer parámetros de HttpServletRequest

Podemos utilizar el método `getParameter()` de la interfaz`HttpServletRequest` para extraer parámetros de la solicitud HTTP:

```
String name = request.getParameter("name");
String email = request.getParameter("email");
```

La primera llamada extrae el parámetro llamado `name` en la etiqueta `<FORM>` del archivo `index.html`. Aquí tiene un resumen:

```
<p>Your Name <input type="text" size="40" name="name"></p>
```

El método `getParameter()` examina todos los parámetros disponibles en la solicitud y extrae un parámetro llamado `name`. De un modo similar, podemos extraer el parámetro `email`. Observe que estos parámetros no necesitan ser extraídos en ningún orden determinado. Por ejemplo, podemos extraer primero el parámetro `email` y después extraer el parámetro `name`.

Así termina la parte de procesamiento de solicitudes. El siguiente paso es preparar la respuesta.

Generar respuesta

El siguiente paso es preparar el contenido dinámico que debe ser enviado al usuario. En la respuesta, nos interesa mostrar un mensaje de bienvenida, como el siguiente:

Good Afternoon, Subrahmanyam Allamaraju

Thanks for registering your email (subrah@subrahmanyam.com) with us.

- The Pro Java Team.

En este ejemplo, el servlet utiliza la corriente de respuesta asociada al objeto respuesta para imprimir directamente el contenido. Observe que las páginas JSP ofrecen una mejor alternativa para la generación de contenido dinámico. Sin embargo, para mayor simplicidad, utilizaremos los servlets en su lugar.

El siguiente fragmento de código calcula un mensaje de bienvenida dependiendo del momento del día en que se emita. Observe que ésta es la única lógica de aplicación ejecutada en esta aplicación. Como veremos en los Capítulos 7 y 8, podemos aplicar una lógica de aplicación más compleja (como el acceso a una base de datos) en este punto:

```
String message = null;
     GregorianCalendar calendar = new GregorianCalendar();
     if(calendar.get(Calendar.AM_PM) == Calendar.AM) {
        message = "Good Morning";
     } else {
        message = "Good Afternoon";
     }
```

Observe que ésta es una forma algo tosca para un mensaje de bienvenida. El servlet utiliza un objeto `java.util.GregorianCalendar` para saber si el momento del día corresponde a AM o PM, y consecuentemente, prepara la respuesta con una cadena de mensaje.

Después de procesar este mensaje, el servlet está preparado para generar la respuesta. Este proceso conlleva los siguientes pasos:

- Establecer un tipo de contenido (MIME) para la respuesta; en nuestro ejemplo es `text/html`:

```
response.setContentType("text/html");
```

- Obtener un objeto `java.io.PrintWriter` del objeto respuesta:

```
PrintWriter out = response.getWriter();
```

- Imprimir la respuesta utilizando el objeto `PrintWriter`:

```
out.println("<html>");
out.println("<body>");
out.println("<p>" + message + ", " + name + "</p>");

out.println("<p> Thanks for registering your email (" + email +
            ") with us.</p>");

out.println("<p> - The Pro Java Team. </p>");
out.println("</body>");
out.println("</html>");
```

- Cerrando el objeto `PrintWriter`:

```
out.close();
```

Estos pasos dan como resultado el documento HTML que aparece en el navegador en respuesta a la solicitud POST.

El descriptor de despliegue

En nuestra aplicación de ejemplo, el archivo de descriptor de despliegue (`Web.xml`) es creado para nosotros por `deploytool`. Si busca en `%J2EE_HOME%\public_html\greeting\`, encontrará un archivo llamado `original.war`. Éste es el archivo WAR para nuestra aplicación Web `greeting` y, como un WAR es un JAR, podemos examinar los contenidos de `original.war` utilizando la herramienta estándar `jar`. Una herramienta zip, como WinZip, también es capaz de entender archivos WAR. Si utiliza una de estas herramientas para examinar `original.war`, encontrará la siguiente estructura:

```
\greeting
          \index.html
          \WEB-INF
                   \Web.xml
                   \classes
                            \GreetingServlet.class
```

Observe que la herramienta de despliegue ha situado nuestro servlet en el directorio correcto. Esto pone de manifiesto una de las ventajas de utilizar esta herramienta para crear aplicaciones, en lugar de crearla de forma manual.

Puede abrir entonces `Web.xml` y examinar su contenido. Recuerde que este archivo incluye la información de configuración para nuestra aplicación Web. La primera línea del descriptor de despliegue es una declaración XML especificando la versión de XML y la codificación utilizada:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Esto va seguido por la declaración del tipo de documento especificando el URL para el DTD utilizado para esta aplicación:

```
<!DOCTYPE Web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
                         'http://java.sun.com/dtd/Web-app_2_3.dtd'>
```

En el caso de las aplicaciones Web, el DTD es parte de la especificación J2EE para aplicaciones Web y es estándar para todas nuestras aplicaciones Web.

La verdadera definición de los servlets aparece entre etiquetas `<Web-app>`. Dentro de la etiqueta `<servlet>`, fíjese en las tres entradas para el nombre del servlet, el nombre de presentación del servlet, y la clase cualificada Java. La etiqueta `<servlet>` permite al contenedor Web conocer el nombre utilizado para referirse a un servlet:

```
<Web-app>
   <display-name>Web App</display-name>
   <servlet>
      <servlet-name>greeting</servlet-name>
      <display-name>greeting</display-name>
      <servlet-class>GreetingServlet</servlet-class>
   </servlet>
   <session-config>
      <session-timeout>30</session-timeout>
   </session-config>
</Web-app>
```

Recuerde que FORM en index.html envía la solicitud a un URL relativo, `/greeting/servlet/GreetingServlet`. El valor de `<servlet-name>` es un alias para el verdadero nombre de clase. Este rasgo nos permite cambiar los archivos de clase servlet cambiando nuestras páginas HTML u otros servlets. También podemos desplegar el mismo servlet más de una vez en la misma aplicación Web. Para conseguirlo, sólo necesitamos utilizar nombres diferentes.

Resumen

Este capítulo ofrece un análisis de los contenedores Web, sin entrar en muchos detalles sobre los API subyacentes. El propósito de este capítulo es introducirle en el desarrollo de aplicaciones Web utilizando servlets Java. Además de ilustrar el modelo de programación de servlets Java con la ayuda de una sencilla aplicación, este capítulo cubre los siguientes puntos:

- Los principios básicos de HTTP: los diferentes tipos de solicitudes y detalles del paradigma de solicitudes y respuestas HTTP
- Los servlets Java, las páginas JSP y los descriptores de despliegue
- Los pasos necesarios para la construcción de una sencilla aplicación Web para recabar datos de los usuarios y generar una respuesta utilizando la implementación de referencia J2EE

Aunque estas áreas ilustran el modelo básico de desarrollo de aplicaciones, hay otras diversas características necesarias para construir útiles aplicaciones Web de calidad de producción. En los siguientes capítulos profundizaremos en las tecnologías de servlets y JSP. Más concretamente, en los siguientes cuatro capítulos trataremos:

- En el capítulo 6, examinaremos el API Servlet con detalle. Además, este capítulo también analizará el ciclo de vida de los servlets y el API de solicitud y respuesta.
- El capítulo 7 le presentará los conceptos de sesiones y contexto, y su función esencial para el desarrollo de aplicaciones Web. Este capítulo también estudiará cómo utilizar dos o más servlets o páginas JSP en coordinación.
- El capítulo 8 le presentará los filtros. Los filtros le permiten intervenir en el proceso en el que el contenedor invoca los servlets y las páginas JSP, o sirve contenido estático.
- El capítulo 9 analizará detenidamente los descriptores de despliegue y cómo utilizarlos para proporcionar seguridad a las aplicaciones.

Estos capítulos le mostrarán el desarrollo de varias aplicaciones Web. Estas muestras se centrarán en diferentes aspectos del API servlet e incorporarán lo que ha aprendido en este capítulo.

6
Programación de servlets

Los servlets son bloques de construcción básicos para construir interfaces de base Web para aplicaciones. La tecnología de servlets proporciona un modelo de programación común que es también la base de las Páginas JavaServer. En capítulo anterior, hemos visto un análisis de las aplicaciones Web y los contenedores Web, en el que hemos repasado los requisitos de programación y estructura básica de las aplicaciones Web. Después nos hemos centrado en presentar los principios básicos de los contenedores Web y las aplicaciones Web, y en cómo las aplicaciones Web pueden ser programadas para mejorar el proceso de solicitud-respuesta HTTP.

El objetivo de los siguientes capítulos es presentar el **API Java Servlet 2.3**, que puede ser descargado desde http://java.sun.com/products/servlet. Este API incluye dos paquetes: `javax.servlet` y `javax.servlet.http`.

> *Fíjese en que el paquete javax.servlet tiene dos subpaquetes para JSP (javax.servlet.jsp) y las etiquetas de adaptación JSP (jsp.servlet.jsp.tagext). Estos paquetes serán analizados a partir del capítulo 10.*

Analizaremos el API Java Servlet en cuatro pasos:

- Implementación de servlets
- Solicitudes y respuestas
- Sesión, contexto y colaboración de servlet
- Filtrado de solicitudes y respuestas

> *En este capítulo, nuestra intención es analizar los pasos primero y segundo; el tercer paso será analizado en el capítulo 7 y el cuarto en el capítulo 8. Adicionalmente, estudiaremos los aspectos de despliegue en el capítulo 9.*

En este capítulo, examinaremos los siguientes puntos:

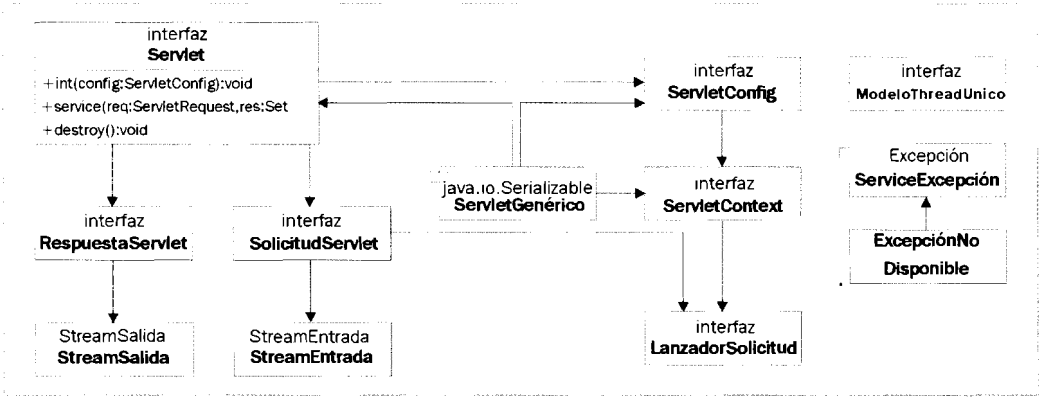
- Clases e interfaces para la implementación de servlets, incluido las excepciones de servlet
- Configuración de servlets
- El ciclo de vida de los servlets
- Solicitudes y respuestas
- El modelo de programación del servlet

Fíjese en que este capítulo no pretende ser una referencia exhaustiva para los paquetes `javax.servlet` y `javax.servlet.http`. En cambio, analizaremos estos paquetes y los conceptos que los respaldan, concentrándonos en cómo utilizar el API durante el desarrollo de las aplicaciones. La documentación del API Servlet debe tenerlo a cerca mientras lee este capítulo. Puede acceder a esta documentación online en http://java.sun.com/products/servlet/2.3/javadoc/index.html.

Análisis del API Java Servlet

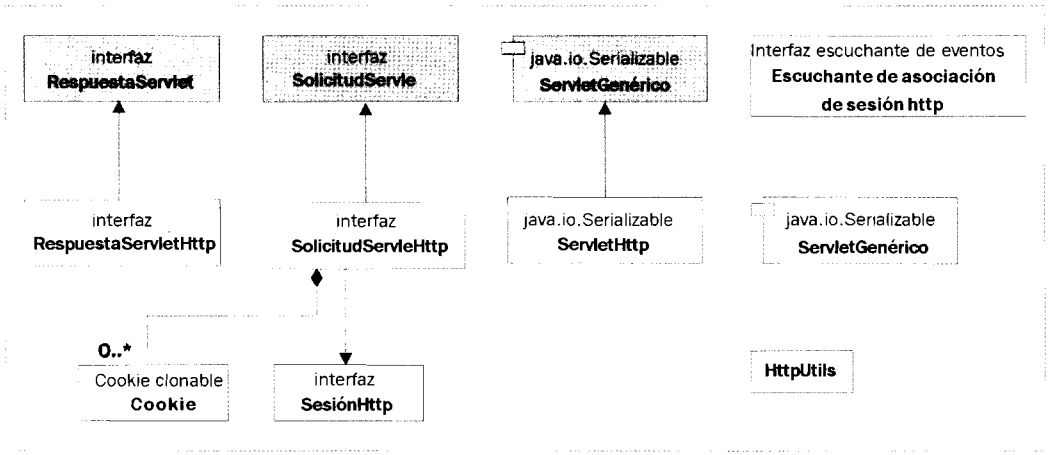
Como ya hemos visto, el API Servlet está especificado en dos paquetes de extensión Java: `javax.servlet` y `javax.servlet.http`. Las clases e interfaces contenidas en el paquete `javax.servlet` son independientes del protocolo, mientras que el segundo paquete, `javax.servlet.http`, contiene clases e interfaces que son específicas de HTTP. Observe que algunas de las clases/interfaces en `javax.servlet.http` amplían las especificadas en el paquete `javax.servlet`.

La siguiente figura muestra el diagrama de clase para algunas de las clases centrales del paquete `javax.servlet` y algunas de sus asociaciones:



Las flechas continuas representan asociaciones, mientras que las flechas discontinuas indican dependencias. Una asociación puede ser herencia, composición o incluso navegabilidad. La dependencia indica algún tipo de combinación entre clases. Por ejemplo, la flecha discontinua que va desde la interfaz javax.servlet.Servlet hasta javax.servlet.ServletRequest indica que el primero depende del último y cualquier cambio en el primero depende del último, y cualquier cambio en la última definición de interfaz provoca cambios en el primero.

La siguiente figura muestra algunas de las clases/interfaces del paquete `javax.servlet.http`:



Observe que el contenedor mantiene todas estas asociaciones, siempre y cuando esté utilizando el paquete `javax.servlet.http`. Sin embargo, si está construyendo servlets especializados que amplíen las clases del paquete `javax.servlet`, tendrá que mantener algunas de las asociaciones.

Estos diagramas no representan el API Servlet completo. Para una mayor sencillez, estas figuras sólo destacan aquellas clases/interfaces que son necesarias para proporcionar un análisis de este API. Presentaremos la parte restante del API durante los siguientes capítulos.

¿Cuál es el propósito de todas estas interfaces y clases? Consideremos las siguientes cuestiones:

- ¿Cómo gestiona el contenedor Web el ciclo de vida de los servlets? ¿Cómo pueden los servlets participar en este ciclo de vida?
- ¿Cómo implementamos la lógica de aplicación? La lógica de aplicación en servlets es la lógica de programación para responder a las solicitudes HTTP. Dependiendo de la arquitectura y complejidad de las aplicaciones, ésta podría ser acceder a bases de datos, invocar los servicios de diversos EJB u otros componentes de capa media, o utilizar JMS para enviar mensajes a otras aplicaciones de empresa, etc.
- ¿Cómo leemos solicitudes HTTP y cómo generamos respuestas HTTP?
- ¿Cómo tratamos las excepciones en los servlets?
- ¿Cómo pueden los servlets interactuar con su entorno?

No intentaremos responder a estas preguntas en este momento pero, después de leer este capítulo, podremos responderlas.

La siguiente tabla ofrece un análisis del API Java Servlet, que será examinado con detalle en breve. En esta tabla, varias intefaces/clases están agrupadas según su propósito y uso (que estudiaremos en un momento). Las interfaces aparecen en cursiva, mientras que las clases (ya sean abstractas o no) y excepciones aparecen en fuente normal. Las adiciones al API Servlet de la versión 2.3 están marcadas con un asterisco (*):

Propósito	Clase/Interfaz
Implementación de servlet	javax.servlet.Servlet
	javax.servlet.SingleThreadModel
	javax.servlet.GenericServlet
	javax.servlet.http.HttpServlet
Configuración de servlet	javax.servlet.ServletConfig
Excepciones servlet	javax.servlet.ServletException
	javax.servlet.UnavailableException
Solicitudes y respuestas	javax.servlet.http.HttpServletRequest
	javax.servlet.http.HttpServletRequestWrapper(*)
	javax.servlet.http.HttpServletResponse
	javax.servlet.http.HttpServletResponseWrapper
	javax.servlet.ServletInputStream (*)
	javax.servlet.ServletOutputStream
	javax.servlet.ServletRequest
	javax.servlet.ServletRequestWrapper (*)
	javax.servlet.ServletResponse
	javax.servlet.ServletResponseWrapper (*)
Localización de sesión	javax.servlet.http.HttpSession
	javax.servlet.http.HttpSessionActivationListener (*)
	javax.servlet.http.HttpSessionAttributeListener (*)
	javax.servlet.http.HttpSessionBindingListener
	javax.servlet.http.HttpSessionBindingEvent
	javax.servlet.http.HttpSessionEvent (*)
	javax.servlet.http.HttpSessionListener (*)
Contexto de servlet	javax.servlet.ServletContext
	javax.servlet.ServletContextAttributeEvent (*)
	javax.servlet.ServletContextAttributeListener (*)
	javax.servlet.ServletContextEvent (*)
	javax.servlet.ServletContextListener (*)
Colaboración de servlet	javax.servlet.RequestDispatcher
Filtrado	javax.servlet.Filter (*)
	javax.servlet.FilterChain (*)
	javax.servlet.FilterConfig (*)
Varios	javax.servlet.http.Cookie
	javax.servlet.http.HttpUtils

Los servidores Web con las implementaciones de Java Servlet, como Tomcat, WebLogic, Orion, iPlanet, JRun, etc., utilizan la mayoría de las interfaces y clases que aparecen en la tabla anterior. De hecho, aparte de las diversas interfaces escuchadoras de la tabla, sólo queda una clase abstracta (`javax.servlet.http.HttpServlet`) y una interfaz (`javax.servlet.Filter`) para implementar cuando construya aplicaciones Web. El contenedor crea el resto de los objetos de periodo de ejecución que son parte de su aplicación. ¿Por qué ocurre así?

Un contenedor Web es un marco de aplicación con un entorno de periodo de aplicación. En el marco, los objetos de aplicaciones son creados e invocados por el periodo de ejecución. Mientras invoca los métodos sobre los objetos de aplicación, el contenedor tiene que construir los objetos para los argumentos. Por ejemplo, consideremos el `GreetingServlet` del capítulo anterior. En este servlet, el contenedor crea y pasa los objetos `HttpServletRequest` y `HttpServletResponse` como argumentos al método `doPost()`:

```
proteced void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletExceotion, IOException {
```

Del mismo modo, el contenedor crea todos los demás objetos (como `ServletConfig`, `ServletContext`, `HttpSession`, etc.) a los que un servlet puede acceder, directa o indirectamente.

La tabla anterior clasifica las clases/interfaces de los paquetes `javax.servlet` y `javax.servlet.http`, basándose en su función en el desarrollo de aplicaciones Web. Por lo tanto, examinemos ahora esta clasificación más detenidamente:

- **Implementación de servlet**
  Las clases/interfaces de esta categoría están destinadas a implementar servlets. Por ejemplo, la clase `GreetingServlet` del capítulo anterior amplía la clase `HttpServlet`. Para desarrollar nuestra propios servlets, implementaríamos uno o más métodos de estas clases/interfaces. Cuando desplegamos servlets, el contenedor Web invoca estos métodos para controlar su ciclo de vida y para ejecutar la lógica de aplicación.

- **Configuración de servlet**
  La interfaz `ServletConfig` pertenece a esta categoría. El API Servlet ofrece varios medios de acceder al objeto `ServletConfig` asociado a un servlet. Este objeto proporciona acceso a ciertos parámetros de inicialización que pueden ser configurados al desplegar un servlet. Demostraremos cómo se puede realizar esto más adelante, en este mismo capítulo.

- **Excepciones servlet**
  El API Java Servlet especifica dos excepciones: `ServletException` y `UnavailableExcepction`. Habitualmente, los servlets lanzan estas excepciones, que el contenedor captura luego para ejecutar un apropiado manejo de errores.

- **Solicitudes y respuestas**
  Hay cuatro interfaces y dos clases abstractas en esta categoría. Estos objetos proporcionan métodos para acceder a la conexión cliente y las corrientes subyacentes de entrada y salida asociadas a ésta. Utilizando estos objetos, podemos leer datos de la entrada y escribir datos de vuelta al cliente.

  Además de las anteriores, la especificación Servlet 2.3 introduce clases envoltorio para las clases `ServletRequest`, `ServletResponse`, `http.HttpServletRequest` y `http.HttpServletResponse` (como hemos visto en la tabla en las nombres de clases terminados en "Wrapper"). Aunque todas estas clases son abstractas, no contienen en realidad ninguna funcionalidad. Examinaremos la función de estas clases envoltorio más adelante en este capítulo.

- **Localización de sesión**
  La localización de sesión es una de las partes esenciales del desarrollo de aplicaciones Web. Como veremos en el capítulo 7, que es el protocolo fundamental para las aplicaciones Web, no tiene

estado. Como resultado, las aplicaciones Web no pueden reconocer múltiples solicitudes del mismo cliente HTTP originándose del mismo lugar. La noción de **sesión** proporciona esta abstracción. En términos sencillos, una sesión le permite agrupar solicitudes en un grupo reunido. Además, la gestión de sesiones también implica la asociación de datos con cada sesión. Al API Servlet especifica la interfaz `javax.servlet.HttpSession` para proporcionar esta abstracción. Aprenderemos más sobre sesiones en el próximo capítulo.

- **Contexto de servlet**
  La noción de contexto de servlet está íntimamente asociada a la noción de aplicación Web. La interfaz `javax.servlet.ServletContext` permite a los servlets de una aplicación compartir datos. También ofrece métodos con los que los servlets pueden acceder al contenedor Web host. Utilizando el objeto `ServletContext`, un servlet puede registrar eventos, obtener referencias URL a los recursos y fijar y almacenar atributos a los que puedan acceder otros servlets del contexto. Trataremos el contexto de servlet en el siguiente capítulo.

- **Colaboración de servlet**
  El API Servlet también proporciona una interfaz (`RequestDispatcher`) con la que un servlet puede invocar otro servlet, una JSP o incluso un recurso estático como una página HTML. Este mecanismo le ayuda a controlar el flujo de lógica de múltiples servlets y páginas JSP programáticamente; lo estudiaremos con detalle en el capítulo 7.

- **Filtrado**
  El API Servlet tiene un mecanismo con el que podemos introducir código (llamado filtro) para participar en el proceso solicitud/respuesta del contenedor. Los filtros son una nueva característica del API Java Servlet versión 2.3. Generalmente, no crean solicitudes/respuestas; en su lugar, modifican o adaptan solicitudes y respuestas desde y hacia un recurso Web, que puede tener contenido estático o dinámico. Estudiaremos los filtros más detenidamente en el capítulo 8.

- **Otros API varios**
  En esta categoría, encontrará dos clases: `http.Cookie`, que puede utilizar para crear cookies y `http.HttpUtils`, que proporciona métodos ayudantes varios. Encontrará algunos ejemplos de estas clases en el siguiente capítulo.

En las siguientes secciones, analizaremos detenidamente las cuatro primeras categorías.

Implementación de servlet

Comencemos examinando de cerca las clases e interfaces utilizadas para implementar servlets.

La interfaz servlet

```
public interface Servlet
```

Esta interfaz especifica el contrato entre el contenedor Web y un servlet. En el paradigma orientado al objeto, un objeto puede comunicar con otro siempre que el primer objeto pueda referenciar al segundo objeto con una interfaz conocida (no es necesario que conozca el nombre de la verdadera clase de implementación). En el caso del API Servlet, `javax.servlet.Servlet` es la interfaz que los contenedores utilizan para referenciar a servlets.

Cuando escriba un servlet, debe implementar esta interfaz directa o indirectamente. Probablemente siempre implemente la interfaz indirectamente ampliando la clase `javax.servlet.GenericServlet` o la clase `javax.servlet.http.HttpServlet`.

Al implementar la interfaz `javax.servlet.Servlet`, los siguientes cinco métodos deben ser implementados:

```
public void init(ServletConfig config)
public void service(ServletRequest request, ServletResponse response)
public void destroy()
public ServletConfig getServletConfig()
public String getServletInfo()
```

El método init()

```
public void init(ServletConfig config) throws ServletException
```

Una vez que el servlet ha sido instanciado, el contenedor Web invoca el método `init()`. El objetivo de este método es permitir a un servlet realizar cualquier inicialización requerida antes de ser invocado en solicitudes HTTP. El contenedor pasa un objeto de tipo `ServletConfig` al método `init()` y, como veremos más tarde, un servlet puede acceder a sus datos de configuración utilizando el objeto `ServletConfig`. El método `init()` lanza una excepción `ServletException` en el caso de que no se complete de forma normal.

La especificación Servlet garantiza que el método `init()` será invocado exactamente una vez en una instancia dada del servlet y el método `init()` podrá completar (siempre que no lance una `ServletException`) antes de que ninguna solicitud sea pasada al servlet.

Algunas de las tareas típicas que pueden ser implementadas en el método `init()` son:

- Leer los datos de configuración de recursos persistentes como archivos de configuración
- Leer parámetros de inicialización utilizando el objeto `ServletConfig`
- Inicializar actividades antiguas como registrar un driver de base de datos, una reserva de conexión o un servicio de registro

El método service()

```
public void service(ServletRequest request, ServletResponse response)
                throws ServletException, IOException
```

Éste es el punto de entrada para ejecutar la lógica de aplicación en un servlet. El contenedor invoca este método en respuesta a solicitudes entrantes. Sólo después de que el servlet haya sido inicializado con éxito, será invocado el método `service()`. El método `service()` acepta dos argumentos, implementado las interfaces `javax.servlet.Servlet` y `javax.servlet.ServletResponse` respectivamente. El objeto solicitud proporciona métodos para acceder a los datos originales de solicitud y el objeto respuesta proporciona métodos con los que el servlet puede construir una respuesta.

El método destroy()

```
public void destroy()
```

Este contenedor invoca este método antes de eliminar una instancia de servlet fuera de servicio. Esto puede suceder si necesita liberar memoria o si el servidor Web está siendo cerrado. Antes de que el

contenedor invoque este método, dará tiempo a los restantes hilos `service()` para que terminen de ejecutar (sujetos a un periodo de tiempo límite), de modo que el método `destroy()` no sea invocado mientras se produce una llamada `service()`. Después de invocar el método `destroy()`, el contenedor no encamina dirige el servlet. Las actividades que pueden ser implementadas en el método `destroy()` incluyen:

- Realizar tareas de limpieza, como cerrar cualquier recurso abierto, cerrar una reserva de conexiones, o incluso informar a otra aplicación/sistema que el servlet ya no está de servicio
- Mantener cualquier estado asociado a un servlet

El método getServletConfig()

```
public ServletConfig getServletConfig()
```

Este método debe ser implementado para devolver el `ServletConfig` que fue pasado al servlet durante el método `init()`.

El método getServletInfo()

```
public String getServletInfo()
```

Este método debe devolver un objeto `String` que contenga información sobre el servlet (por ejemplo, autor, fecha de creación, descripción, etc.). Ésta está disponible para el contenedor Web, en caso de deseara mostrar, por ejemplo, una lista de servlets instalados junto con sus descripciones.

La clase GenericServlet

```
public abstract class GenericServlet implements Servlet, ServletConfig,
                                                Serializable
```

La clase `GenericServlet` proporciona una implementación básica de la interfaz `Servlet`. Ésta es una clase abstracta y todas las subclases deben implementar el método `service()`. Esta clase abstracta tiene los siguientes métodos además de los declarados en `javax.servlet.Servlet` y `javax.servlet.ServletConfig`:

```
public init()
public void log(String message)
public void log(String message, Throwable t)
```

El método init (`ServletConfig config`) almacena el objeto `ServletConfig` en una variable de instancia privada transitoria (llamada `config`). Puede utilizar el método `getServletConfig()` para acceder a este objeto. Sin embargo, si elige ignorar este método, debe incluir una llamada a `super.init(config)`. Alternativamente, puede ignorar el método sin argumento `init()` de la clase `GenericServlet`.

La clase `GenericServlet` también implementa la interfaz `ServletConfig`. Esto permite al desarrollador de servlet invocar los métodos `ServletConfig` directamente sin tener que obtener primero un objeto `ServletConfig`. Estos métodos son `getInitParameter()`, `getInitParameterNames()`, `getServletContext()` y `getServletName()`. Cada uno de estos métodos delega las llamadas en los métodos respectivos del objeto `ServlerConfig` almacenado.

La clase `GenericConfig` también incluye dos métodos para escribir a un registro de servlet, que invoca los métodos correspondientes en `ServletContext`. El primer método, `log (String msg)`, escribe el nombre del servlet y el argumento msg en el registro del contenedor Web. El otro método, `log (String msg, Throwable cause)`, incluye una traza de pila para la excepción dada `Throwable` además del nombre del servlet y el mensaje. La verdadera implementación del mecanismo de registro es específica del contenedor, aunque la mayoría de los contendores utilizan archivos de textos para los registros.

La interfaz SingleThreadModel

```
public interface SingleThreadModel
```

El API Servlet especifica una **interfaz de registro** especial llamada `javax.servlet.SingleThreadModel`.

Durante la vida de un servlet que no implementa esta interfaz, el contenedor puede enviar múltiples solicitudes de servicio en diferentes hilos a una única instancia. Esto significa que la implementación del método `service()` debe estar a salvo de hilos. Sin embargo, ¿cuál es la alternativa si el método `service()` no está a salvo de hilos?

El API Java Servlet especifica la interfaz `SingleThreadModel` con este propósito. Los servlets pueden implementar la interfaz `SingleThreadModel` (además de implementar la interfaz `javax.servlet.Servlet` o ampliar una de sus clases de implementación) con el objetivo de informar al contenedor de que debe asegurarse de que sólo un hilo está ejecutando el método `service()` del servlet en un momento dado.

Para servlets `SingleThreadModel`, los contenedores deben seguir uno de los siguientes enfoques para asegurar que cada instancia de servlet es invocada en un hilo independiente:

- **Reserva de instancias**
  En este enfoque, el contenedor mantiene una reserva de instancias de servlet. Para cada solicitud entrante, el contenedor destina una instancia de servlet de la reserva y sobre la conclusión del servicio, el contenedor devuelve la instancia a la reserva.
- **Serialización de solicitudes**
  En este enfoque, el contenedor mantiene una única instancia del servlet. Sin embargo, como el contenedor no puede enviar múltiples solicitudes a la instancia a la vez, el contenedor serializa las solicitudes. Esto significa que las nuevas solicitudes se mantendrán a la espera mientras que la solicitud actual es servida.

En realidad, una combinación de estos dos enfoques es más pragmática, de modo que el contendor podría mantener un número razonable de instancias en la reserva, mientras que todavía serializaría las solicitudes si el número de solicitudes excediera el número de instancias en la reserva.

Fíjese en que el modelo de hilo único (`SingleThreadModel`) es intensivo en recursos, especialmente si se espera un gran número de solicitudes concurrentes para el servlet. El efecto del `SingleThreadModel` es que el contenedor invoca el método `service()` en un bloque sincronizado. Esto es equivalente a utilizar la palabra clave `synchronized` para el método `service()` del servlet. Consecuentemente, cuando hay cientos o incluso miles de solicitudes concurrentes en el contenedor Web, el contenedor puede serializar las solicitudes a la misma instancia del servlet o crear tantas instancias como solicitudes. En el primer caso, la capacidad del contendor para procesar las solicitudes concurrentes se verá severamente perturbada debido a la serialización. En el último caso, el contenedor se encuentra con más asignación de objeto (que incluye más sobregasto de creación de objetos y gasto de memoria).

Sin embargo, en casos en los que sólo algunas instrucciones del método `service()` no están a salvo de hilos, debe considerar reducir el alcance de la sincronización y sincronizar explícitamente esos bloques utilizando la palabra clave `synchronized`. Cuanto más cortos sean los bloques, mejor puede ser el rendimiento de este enfoque.

> **Siempre debemos considerar la posibilidad de volver a diseñar nuestras aplicaciones para evitar el modelo de hilo único, `SingleThreadModel`, o la sincronización de hilos. Si no podemos evitarlas, es importante ser consciente de las implicaciones de rendimiento. Con cualquier código de sincronización de hilos está bloqueando potencialmente un hilo de contenedor Web.**

La clase HttpServlet

```
public abstract class HttpServlet extends GenericServlet implements Serializable
```

La clase `HttpServlet` amplía `GenericServlet` y proporciona una implementación específica de HTTP de la interfaz Servlet. Ésta será probablemente la clase que todos los servlets ampliarán. Esta clase especifica los siguientes métodos:

```
public void service(ServletRequest request, ServletResponse response)
protected void service(HttpServletRequest request, HttpServletResponse response)
protected void doGet(HttpServletRequest request, HttpServletResponse response)
protected void doPost(HttpServletRequest request, HttpServletResponse response)
protected void doHead(HttpServletRequest request, HttpServletResponse response)
protected void doDelete(HttpServletRequest request, HttpServletResponse response)
protected void doOptions(HttpServletRequest request, HttpServletResponse response)
protected void doPut(HttpServletRequest request, HttpServletResponse response)
protected void doTrace(HttpServletRequest request, HttpServletResponse response)
protected long getLastModified(HttpServletRequest request)
```

Los métodos service()

El `HttpServlet` tiene dos variantes de este método:

```
public void service(ServletRequest request, ServletResponse response)
          throws ServletException, IOException
```

Ésta es una implementación del método `service()` en `GenericServlet`. Este método emite los objetos de solicitud y respuesta a `HttpServletRequest` y `HttpServletResponse`, e invoca el siguiente método sobrecargado `service()`. Por lo tanto, no debe ignorar el método anterior.

```
protected void service(HttpServletRequest request, HttpServletResponse response)
          throws ServletException, IOException
```

Este método sobrecargado toma objetos solicitud y respuesta específicos de HTTP, y es invocado por el primer método anterior. `HttpServlet` implementa este método para ser un lanzador de solicitudes HTTP. Como verá más adelante en este mismo capítulo, la interfaz `javax.servlet.ServletRequest` proporciona un método `getMethod()` que devuelve el tipo de método HTTP asociado a la solicitud. Por ejemplo, para solicitudes GET, este método devuelve `"GET"` como una cadena. El método `service()` utiliza esta cadena para delegar la solicitud en uno de los métodos `doXXX()`. `Javax.servlet.http.HttpServlet` proporciona implementaciones por defecto a todos estos métodos.

En general, debe evitar ignorar este método ya que afecta al comportamiento por defecto del método `service()`. La única situación que requiere ignorar este método se produce cuando se quiere cambiar el comportamiento por defecto o cuando se quiere incluir procesamiento adicional común a todos los métodos `doXXX()`. Incluso en esos casos, debe considerar incluir una llamada a `super.service()` en sus servlets.

La secuencia de llamadas de método cuando el contenedor recibe una solicitud para un servlet es la siguiente:

- El contenedor invoca el método `service()`
- El método público `service()` invoca el método protegido `service()` después de emitir los argumentos a `HttpServletRequest` y `HttpServletResponse` respectivamente
- El método protegido `service()` invoca uno de los métodos `doXXX()`, dependiendo del tipo del método de solicitud HTTP

Los métodos doXXX()

La clase `HttpServlet` implementa los siguientes métodos protegidos, uno para cada método de solicitud HTTP:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
protected void doHead(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
protected void doDelete(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
protected void doOptions(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
protected void doPut(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
protected void doTrace(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
```

La firma de cada uno de los métodos `doXXX()` es la misma que la del método protegido `service()` anterior; cada uno toma argumentos `HttpServletRequest` y `HttpServletResponse`, y lanza `ServletException` y `IOException`.

La clase `HttpServlet` proporciona implementaciones adecuadas para los métodos TRACE y OPTIONS, y no es necesario que sus servlets ignoren `doTrace()` y `doOptions()`.

Para los otros cinco métodos, la clase `HttpServlet` proporciona implementaciones que devuelven errores HHTP. Para el caso de contenedores sometidos a HTTP 1.0, estos métodos devuelven un error HTTP con código de estado 400, indicando que la solicitud enviada por el cliente es sintácticamente incorrecta. Para los contenedores que se ajustan a HTTP 1.1, estos métodos devuelven un error HTTP con código de estado 405, indicando que el método HTTP solicitado no está permitido para este servlet. Esta clase utiliza el método `getProtocol()` de la interfaz `javax.servlet.ServletRequest` para determinar el protocolo.

Dependiendo de su aplicación, debe determinar los métodos HTTP que deben ser apoyados por su servlet e ignorar consecuentemente los métodos `doXXX()` correspondientes (excepto los de los métodos TRACE y OPTIONS).

El método getLastModified()

```
protected long getLastModified(HttpServletRequest req)
```

Este método implementa una operación condicional *get*. Por ejemplo, debe devolver el momento en el que el servlet fue modificado por última vez en milisegundos desde el 1 de Enero, 1970 00:00:00 GMT. La implementación por defecto devuelve un número negativo (-1) indicando que el momento de modificación es desconocido.

HTTP 1.1 alberga la noción de solicitudes condicionales GET. La cabecera `If-Modified-Since` es una de las cabeceras que convierte a una solicitud en condicional. Los clientes que se ajustan a HTTP 1.1 pueden enviar esta cabecera con un campo de fecha (por ejemplo: `If-Modified-Since: Sat, 01 Jan 2000 00:00:00 GMT`). Esta cabecera indica al servidor que si el recurso solicitado no ha sido modificado desde la fecha indicada en la cabecera, el contenedor Web no necesita invocar el método `doGet()`, puesto que todas las versiones actuales de navegadores de uso común como Internet Explorer 5.5, Netscape 6.0, y Opera 5.1 se ajustan a HTTP 1.1 y se aprovechan de esta cabecera.

Sus servlets pueden ignorar este método para controlar la memorización en caché de páginas generadas por el método `doGet()`.

Configuración de servlets

En el API Java Servlet, los objetos `javax.servlet.ServletConfig` representan la configuración de un servlet. La información de configuración contiene parámetros de inicialización (un conjunto de pares nombre/valor), el nombre del servlet y un objeto `javax.servlet.ServletContext`, que ofrece la información del servlet sobre el contenedor. Los parámetros de inicialización y el nombre de un servlet pueden ser especificados en el descriptor de despliegue (el archivo `Web.xml`), por ejemplo:

```
<Web-app>
   <servlet>
      <servlet-name>Admin</servlet-name>
      <servlet-class>com.wrox.admin.AdminServlet</servlet-class>
      <init-param>
          <param-name>email</param-name>
          <param-value>admin@admin.wrox.com</param-value>
      </init-param>
      <init-param>
          <param-name>helpURL</param-name>
          <param-value>/admin/help/index.html</param-value>
      </init-param>
   </servlet>
<Web-app>
```

Este ejemplo registra un servlet de nombre `admin` y especifica dos parámetros de inicialización, `email` y `helpURL`. El contenedor Web lee esta información, y la pone a disposición de `com.wrox.admin.AdminServlet` mediante el objeto `javax.servlet.ServletConfig`. Si queremos cambiar estos parámetros, podemos hacerlo sin tener que volver a compilar el servlet. Aprenderemos más sobre este enfoque en el capítulo 9.

El mecanismo del descriptor de despliegue fue introducido en la especificación Servlet 2.2. Los contenedores Web (llamados entonces motores de servlet) que se ajustan a versiones antiguas de la especificación, proporcionan medios específicos del proveedor (como archivos apropiados) para especificar los parámetros de inicialización.

La interfaz ServletConfig

```
public interface ServletConfig
```

Como hemos ya visto cuando hemos analizado la clase `GenericServlet`, esta interfaz especifica los siguientes métodos:

```
public String getInitParameter(String name)
public Enumeration getInitParameterNames()
public ServletContext getServletContext()
public String getServletName()
```

El método getInitParameter()

```
public String getInitParameter(String name)
```

Este método devuelve el valor de un parámetro llamado inicialización o `null` si el parámetro especificado no existe. En el ejemplo anterior, al invocar el método `getInitParameter` con `"email"` como argumento, se obtiene el valor `"admin.@admin.wrox.com"`.

El método getInitParameterNames()

```
public Enumeration getInitParameterNames()
```

Este método devuelve una enumeración de todos los parámetros de inicialización de un servlet. Utilizando esta enumeración, puede obtener los nombres de todos los parámetros de enumeración sucesivamente. Si no hay parámetros de inicialización especificados, este método devuelve una enumeración vacía. En el ejemplo anterior, invocar el método `getInitParameterNames()` devuelve una enumeración que contiene dos objetos `String`: `"email"` y `"helpURL"`.

El método getServletContext()

```
public ServletContext getServletContext()
```

Este método devuelve una referencia al objeto `ServletContext` asociado con la aplicación Web. `javax.servlet.ServletContext` es analizado con más detalles en el siguiente capítulo.

El método getServletName()

```
public String getServletName()
```

Este método devuelve el nombre asignado a un servlet en su descriptor de despliegue. Si no hay ningún nombre especificado, éste devuelve en su lugar el nombre de clase de servlet.

Obtener una referencia a ServletConfig

En el API Java Servlet, un servlet puede obtener una referencia al objeto `javax.servlet.ServletConfig` de los siguientes modos:

Durante la inicialización del servlet

Como hemos analizado anteriormente, los métodos `init()` de la interfaz `Servlet` y la interfaz `GenericServlet` tienen un argumento de tipos `javax.servlet.ServletConfig`. Durante la inicialización de un servlet, el contenedor Web crea este argumento y lo pasa el método `init()`. Cuando sustituye el método `init()`, puede acceder al objeto `javax.servlet.ServletConfig`.

Sin embargo, cuando ignora el método anterior `init()` en la clase `GenericServlet`, debe invocar explícitamente `super.init()` como sigue:

```
public init(ServletConfig config) {
   super.init(config);

   // Inicialización aquí
}
```

La llamada a `super.init(config)` asegura que la clase `GenericServlet` recibe una referencia al objeto `ServletConfig`. La implementación de la clase `GenericServlet` mantiene en realidad una referencia al objeto `ServletConfig` (como una variable de instancia transitoria) y requiere que `super.init(config)` sea invocado en subclases.

Utilizar el método getServletConfig()

Los servlets también pueden acceder al objeto `ServletConfig` invocando el método `getServletConfig()`. Este método está especificado en la interfaz `javax.servlet.Servlet`.

Alternativamente, los servlets que amplían el `GenericServlet` o su subclase `HttpServlet` también pueden invocar los métodos de la interfaz `ServletConfig` directamente. Esto se debe a que `GenericServlet` también implementa la interfaz `ServletConfig`.

Excepciones servlet

El paquete `javax.servlet` especifica dos clases de excepciones: `javax.servlet.ServletException` y `javax.servlet.UnavailableException`.

La clase ServletException

```
public class ServletException extends java.lang.Exception
```

Ésta es una excepción genérica que puede ser lanzada por los métodos `init()`, `service()`, `doXXX()` y `destroy()`. La clase proporciona los siguientes constructores:

```
public ServletException()
public ServletException(String message)
```

Al crear objetos de tipo `ServletException`, puede integrar cualquier excepción de aplicación (llamada **causa-raíz**). Los contenedores utilizan la excepción causa-raíz con propósitos de registro. Por ejemplo, puede integrar una excepción `javax.sql.SQLException` en una excepción `javax.servlet.ServletException`. Existen dos constructores adicionales para apoyar estas excepciones causa-raíz:

```
public ServletException(Throwable cause)
public ServletException(String message, Throwable cause)
```

El método `getRootCause()` devuelve la excepción causa-raíz:

```
public Throwable getRootCause()
```

La clase UnavailableException

```
public class UnavailableException extends ServletException
```

`javax.servlet.UnavailableException` es un tipo especial de excepción servlet. Puesto que este servlet amplía `javax.servlet.ServletException`, todos los métodos de servlet que pueden lanzar una excepción `javax.servlet.Exception` también pueden lanzar una excepción `javax.servlet.UnavailableException`.

El objetivo es indicar al contenedor Web que el servlet no está disponible bien de forma temporal o permanentemente.

Esta clase especifica los siguientes constructores:

```
public UnavailableException(String message)
```

Éste construye una nueva excepción permanentemente indisponible con el mensaje dado:

```
public UnavailableException(String message, int seconds)
```

Esto construye una nueva excepción no disponible con el mensaje dado. El argumento `seconds` indica la duración en segundos durante la cual el servlet no está disponible.

Analizaremos más tarde el comportamiento exacto del contenedor en fallos temporales y permanentes.

El ciclo de vida de los servlets

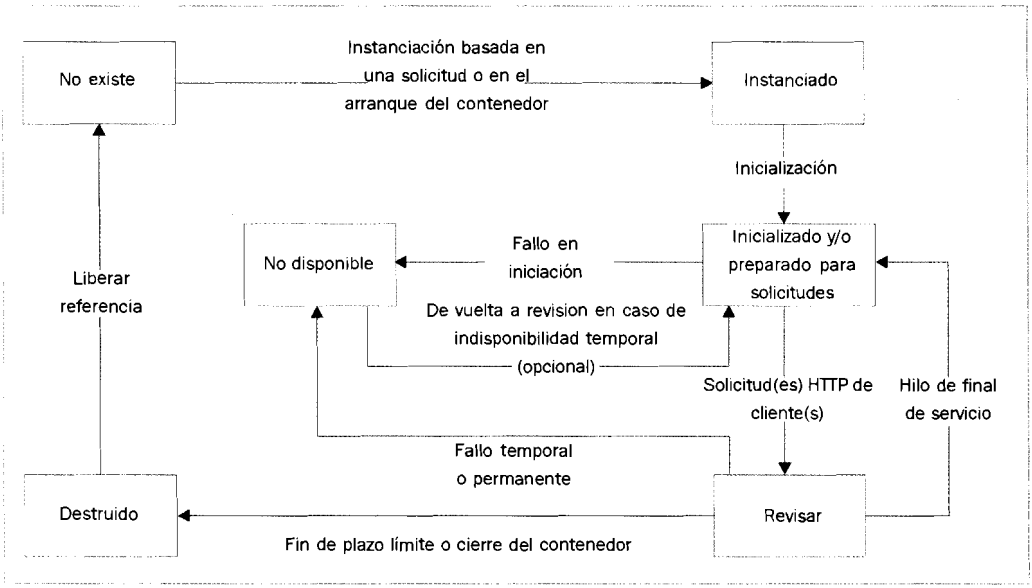
Como hemos visto anteriormente, el contenedor es un periodo de ejecución que gestiona los servlets. De entre las diversas responsabilidades de un contenedor, la gestión del ciclo de vida es la más esencial. En el caso de los servlets, los eventos de ciclo de vida están especificados en la interfaz `javax.servlet.Servlet` del API Servlet. Aunque la gestión del ciclo de vida es una responsabilidad del contenedor, como desarrolladores de servlets, debemos asegurarnos de que nuestros servlets siguen el modelo de ciclo de vida y de que no son implementados en un modo que contradiga esta condición.

Los métodos de la interfaz Servlet relevantes para el ciclo de vida del servlet son `init()`, `service()` y `destroy()`. El ciclo de vida empieza cuando el contenedor invoca el método init() y termina cuando el contenedor invoca el método `destroy()`.

El ciclo de vida de un servlet consiste en las siguientes etapas fundamentales:

- **Instanciación**
  El contenedor Web crea una instancia del servlet
- **Inicialización**
  El contenedor invoca el método `init()` de la instancia
- **Revisión**
  Si el contenedor tiene una solicitud para el servlet, invoca el método `service()` de la instancia del servlet
- **Destrucción**
  Antes de destruir la instancia, el contendor invoca el método destroy() de la instancia del servlet
- **No disponible**
  La instancia es destruida y marcada para la recolección de residuos

El diagrama de estado UML muestra las posibles transiciones en el ciclo de vida del servlet:



El contenedor crea una instancia de servlet en respuesta a una solicitud HTTP entrante o en el arranque del contenedor. Después de la instanciación, el contenedor inicializa la instancia invocando su método `init()`. Después de la inicialización, la instancia de servlet está preparada para servir solicitudes entrantes. El propósito de este proceso de inicialización es cargar cualquier parámetro de inicialización requerido para el servidor. Veremos cómo se puede conseguir esto en la siguiente sección.

Durante el proceso de inicialización, una instancia de servlet puede lanzar una excepción `ServletException` o `UnavailableException`. `UnavailableException` es una subclase de `ServletException`. Mientras que `ServletException` puede ser utilizada para indicar fallos generales de inicialización (como fallos al encontrar los parámetros de inicialización), `UnavailableException` se utiliza para informar sobre la no-disponibilidad de la instancia para servir solicitudes. Por ejemplo, si su servlet depende de un servidor RMI y está verificando si el servidor es alcanzable para revisión, su instancia de servlet puede lanzar una `UnavailableException` para indicar que está temporal o permanentemente indisponible. Si su instancia de servlet determina que la no-disponibilidad debe ser temporal, puede que así lo indique mientras construye la excepción `UnavailableException`, especificando el número de segundos de indisponibilidad al constructor de la excepción. Cuando tiene lugar este fallo, el contenedor suspende todas las solicitudes a su servlet durante el periodo especificado y lo devuelve a su estado de disponibilidad al final del periodo. Si no especifica internamente una configuración de indisponibilidad, el servlet estará indisponible cuando se reinicie el contenedor. Una de las aplicaciones de esta excepción tiene lugar cuando uno de sus componentes o sistemas de segundo plano no está accesible temporalmente. En general, si su lógica de aplicación es tal que ciertos fallos pueden ser resueltos reintentándolo después de un rato, puede utilizar esta excepción.

El contenedor garantiza que, antes de que se invoque el método `service()`, el método `init()` podrá ser completado y que, antes de que el servlet sea destruido, su método `destroy()` será invocado.

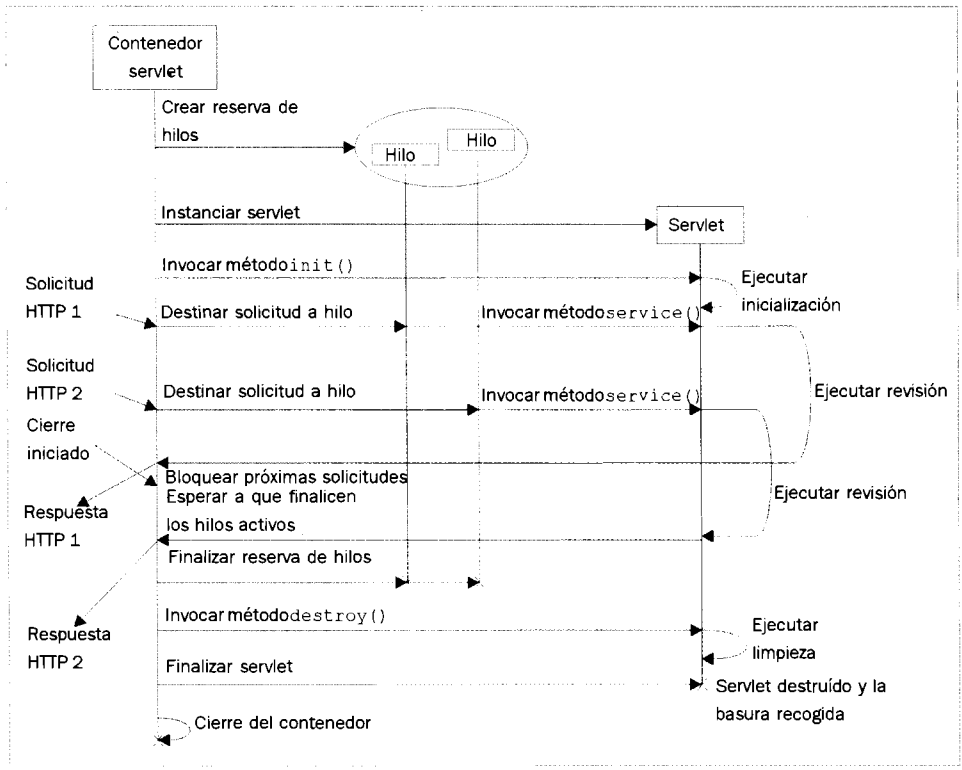
El servlet puede lanzar `ServletException` o `UnavailableException` durante su método `service()`, en cuyo caso el contenedor suspenderá las solicitudes para esa instancia temporal o permanentemente. Es importante que diseñe sus servlets considerando fallos temporales y permanentes.

En teoría, no hay nada que impida que un contenedor Web ejecute el ciclo de vida completo del servlet cada vez que un servlet es solicitado. En la práctica, los contenedores Web cargan e inicializan los servlets durante el arranque del contenedor o cuando el servlet es invocado por primera vez y mantienen esa instancia de servlet en la memoria para revisar todas las solicitudes que recibe. El contenedor puede decidir en cualquier momento liberar la referencia de servlet, finalizando así el ciclo de vida del servlet. Esto podría ocurrir, por ejemplo, si el servlet no ha sido invocado durante un tiempo o si el contenedor se está cerrando. Cuando esto sucede, el contenedor invoca el método `destroy()`.

En el modelo habitual de ciclo de vida de servlet, el contenedor Web crea una única instancia de cada servlet. Pero, ¿qué ocurre si el método `service()` del servlet está todavía ejecutándose cuando el contenedor Web recibe otra solicitud? Para los servlets que no implementan la interfaz `javax.servlet.SingleThreadModel`, el contenedor invoca la misma instancia de servlet en cada hilo de solicitud. Por lo tanto, siempre es posible que el método `service()` sea ejecutado en más de un hilo, siendo necesario que el método `service()` esté a salvo de hilos. Aparte de no acceder a recursos a salvo de hilos (como escribir a archivos), debemos también considerar mantener nuestros servlets sin estado (es decir, no definir ningún atributo en sus clases de servlet). Al definir variables de instancias en nuestros servlets, debemos asegurarnos de que tales variables son manipuladas de modo seguro.

Hemos analizado previamente el rendimiento de los servlets que implementan la interfaz `javax.servlet.SingleThreadModel`. Recuerde que, para estos servlets, el contenedor puede serializar solicitudes o mantener una reserva de instancias de servlet y destinar cada solicitud a una instancia diferente de la reserva, o utilizar una combinación de estas dos.

El siguiente diagrama muestra la secuencia de eventos que representan un servlet durante su carga, revisando dos solicitudes en rápida sucesión y siendo después descargado cuando el servidor se cierra:



Existe todavía otra situación en la que un contenedor podría crear más de una instancia de un servlet: sería el caso en el que una clase servlet fuera añadida en el descriptor de despliegue más de una vez, posiblemente con diferentes parámetros de inicialización.

Puede que encuentre los anteriores enfoques de instanciación bastante complejos. Para evitar problemas, sus servlets no deben asumir la instanciación. Por ejemplo, no debemos asumir que la misma instancia de servlet se utilice para todas las solicitudes/clientes o que sólo un hilo ejecute un método `service()` a la vez.

El ciclo de vida de un servlet – *FreakServlet*

Consideremos ahora un ejemplo, para comprender mejor el ciclo de vida del servlet. El ejemplo consiste en un único servlet llamado `FreakServlet`, que demuestra los diversos estados de un típico servlet, incluido la indisponibilidad.

Introduzca la siguiente fuente en `%BOOK_HOME%\Ch06\freakservlet\src\FreakServlet.java`, donde `BOOK_HOME` es el directorio raíz para su código fuente:

```
// Importar paquetes de servlet
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import javax.servlet.UnavailableException;

// Importar otros paquetes Java
import java.io.IOException;
import java.io.PrintWriter;

public class FreakServlet extends HttpServlet {
   java.util.Vector states;
   java.util.Random random;
   int waitInterval;
   public static final int DEFAULT WAIT INTERVAL = 10;

   public FreakServlet() {
      states = new java.util.Vector();
      random = new java.util.Random();
      waitInterval = DEFAULT WAIT INTERVAL;
      states.add(createState("Instantiation"));
   }

   public void init() throws ServletException {
      states.add(createState("Initialization"));
      String waitIntervalString =
         getServletConfig().getInitParameter("waitInterval");
      if (waitIntervalString != null) {
         waitInterval = new Integer(waitIntervalString).intValue();
      }
   }

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                     throws ServletException, IOException {

      if (random.nextBoolean()) {
         // No disponible durante waitInterval segundos
         states.add(createState("Unavailable from doGet"));
         throw new UnavailableException("Unavailable from doGet", waitInterval);
```

```
    }

    states.add(createState("Service"));

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // enviar acuse al navegador
    out.println("<html>");
    out.println("<meta http-equiv=\"Pragma\" content=\"no-cache\">");
    out.println("<head><title>");

    out.println("FreakServlet: State History");
    out.println("</title></head>");
    out.println("<body>");
    out.println("<h1>FreakServlet: State History</h1>");

    out.println("<a href=\"/lifeCycle/servlet/freak\">Reload</a></p>");

    for (int i = 0; i < states.size(); i++) {
       out.println("<p> " + states.elementAt(i) + "</p>");
    }
    out.println("</body></html>");
    out.close();
  }

  public void destroy() {
    states.add(createState("Destroy"));
    system.out.println("Flushing state history of LifeCycleTest servlet.");
    for (int i = 0; i < states.size(); i++) {
       system.out.println(states.elementAt(i).toString());
    }
  }

  private String createState(String message) {
    return "[" + (new java.util.Date()).toString() + "] " + message;
  }
}
```

Crear un archivo `Web.xml` (descriptor de despliegue) en `%BOOK_HOME%\Ch06\freakServlet\WEB-INF\` con las siguientes definiciones `<servlet>` y `<error-page>`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE Web-app
     PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
     "http://java.sun.com/j2ee/dtds/Web-app 2 3.dtd">

<Web-app>
  <servlet>
     <servlet-name>freak</servlet-name>
     <display-name>freak</display-name>
     <servlet-class>FreakServlet</servlet-class>
     <init-param>
        <param-name>waitInterval</param-name>
        <param-value>5</param-value>
     </init-param>
  </servlet>

  <error-page>
     <exception-type>javax.servlet.UnavailableException</exception-type>
     <location>/unavailable.html</location>
```

```
    </error-page>
</Web-app>
```

Ahora compile el servlet en el directorio %BOOK_HOME%Ch06\freakServlet\WEB-INF\classes. Consulte el capítulo anterior para más instrucciones.

Copie también el siguiente HTML en %BOOK_HOME%Ch06\freakServlet\unavailable.html:

```
<html>
   <meta http-equiv="Pragma" content="no-cache">
   <head>
      <title>FreakServlet Unavailable</title>
   </head>

   <body>
      <h1>FreakServlet Unavailable</h1>

      <p>FreakServlet is temporarily unavailable. Please try
          <a href="/lifeCycle/servlet/freak">again</a>.</p>
   </body>
</html>
```

Ahora necesitamos crear nuestro archivo WAR para el despliegue en la Implementación de Referencia J2EE (cubriremos el asistente de aplicación Web más adelante en el capítulo pero, por ahora, sólo desplegaremos nuestro servlet del modo más fácil posible).

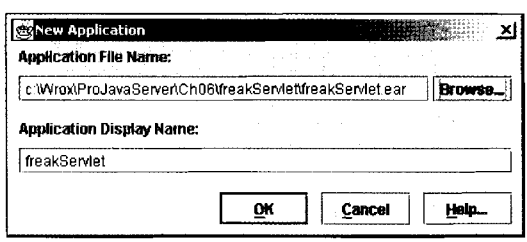
Asegúrese de que tiene los siguientes archivos en las siguientes posiciones:

```
/unavailable.html
/WEB-INF/
          Web.xml
          classes/
                    FreakServlet.class
```

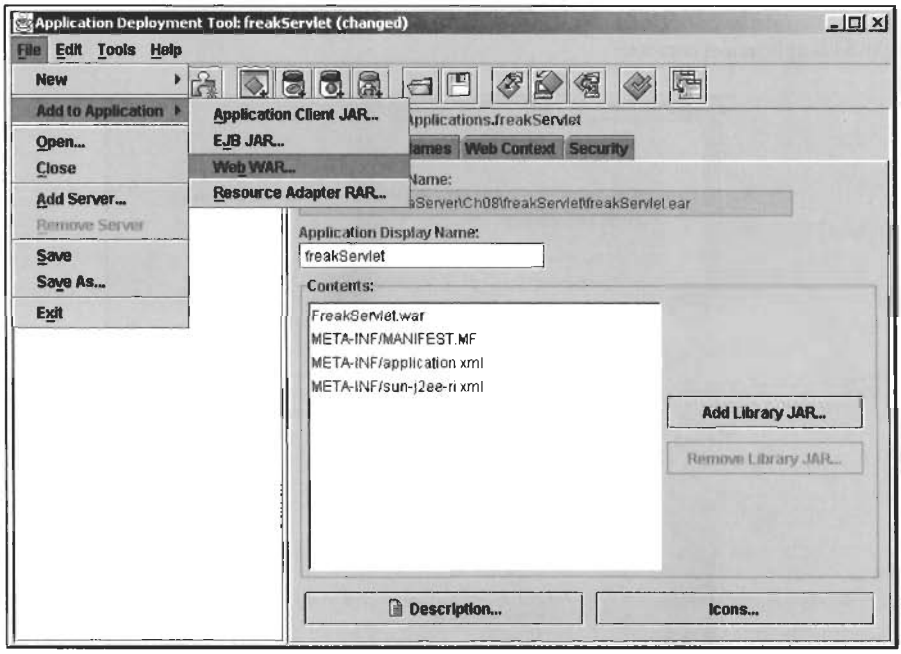
Cree ahora un archivo WAR utilizando el comando jar:

```
jar -cf FreakServlet.war unavailable.html WEB-INF
```

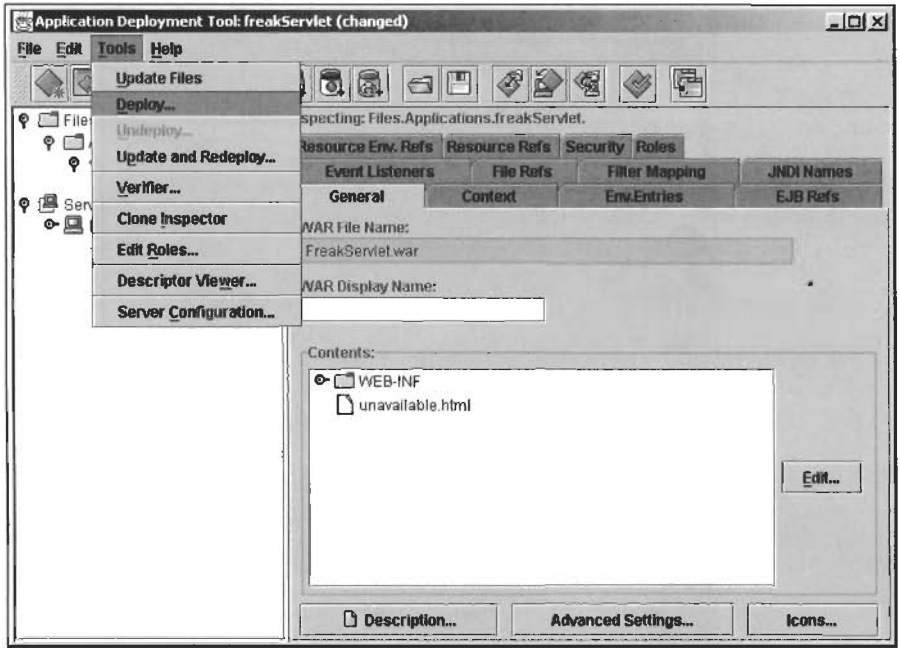
Ahora estamos preparados para desplegar el archivo WAR. Inicie el servidor de la Implementación de Referencia J2EE ejecutando el archivo `j2ee.bat` en el directorio %J2EE_HOME%\bin, ejecute luego el archivo `deploytool.bat` en el mismo directorio, para iniciar la herramienta de despliegue. Cuando se cargue la herramienta de despliegue, cree una nueva aplicación llamada `freakServlet`:



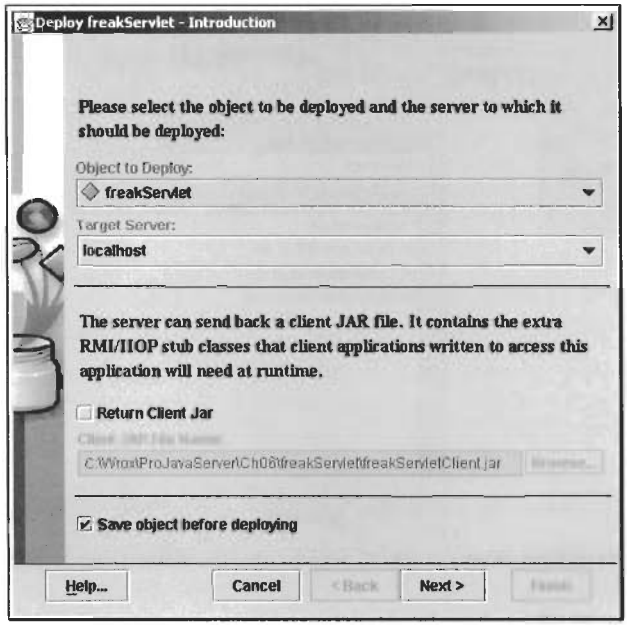
Añada después nuestro archivo WAR a la nueva aplicación utilizando la opción `Add to Application` (Agregar a la aplicación) del menú `File` (Archivo):



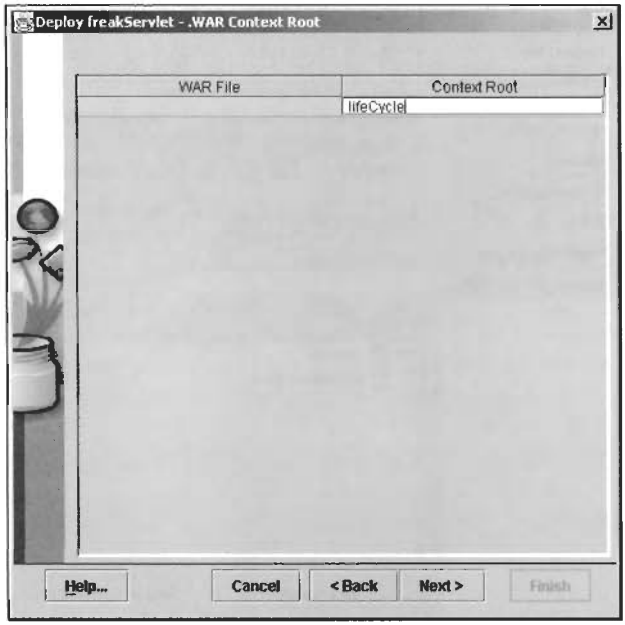
Seleccione el archivo WAR en el cuadro de diálogo abierto para cargar el archivo WAR en la aplicación. Ahora podemos sencillamente desplegar nuestra aplicación eligiendo `Deploy` (Desplegar) del menú `Tools` (Herramientas):



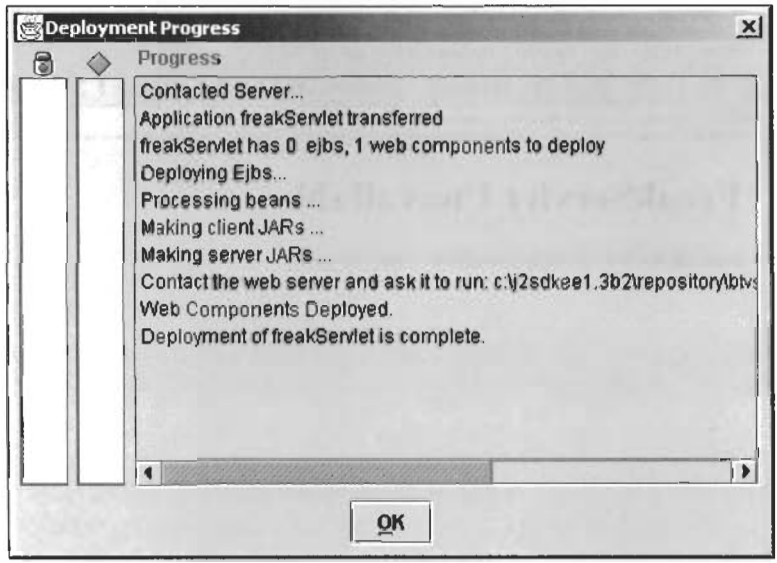
No es necesario que hagamos nada en la primera pantalla excepto volver a comprobar si estamos desplegando la aplicación correcta:



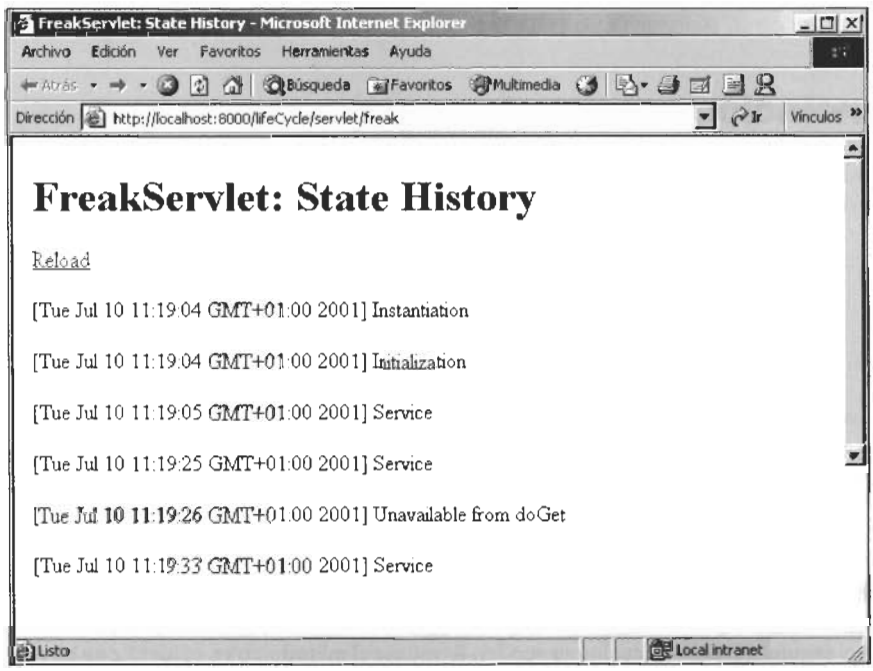
En la siguiente pantalla, necesitamos proporcionar el contexto que utilizaremos para navegar nuestra aplicación. Configúrelo en `lifeCycle`:



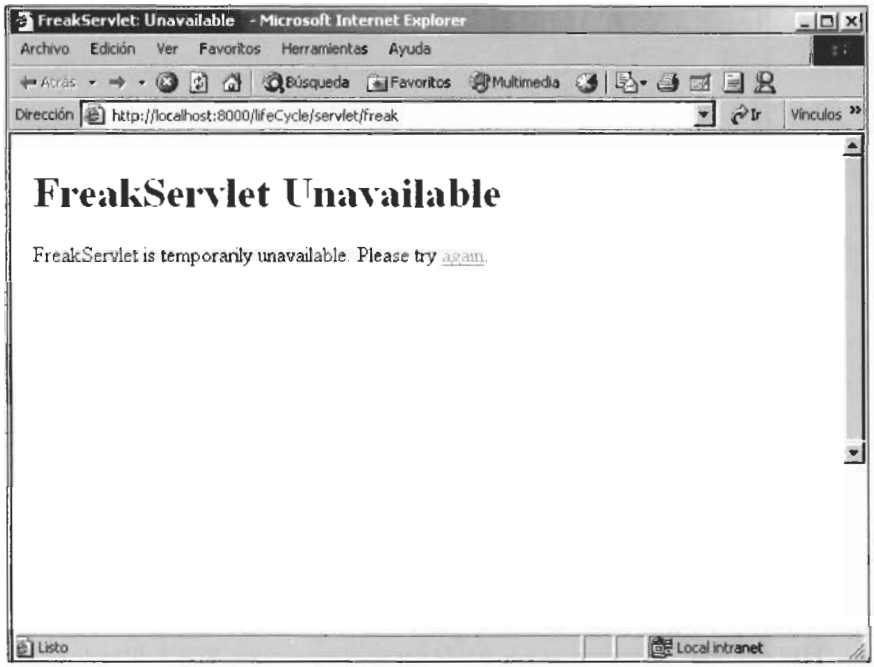
Proceda hasta la pantalla final y seleccione `Finish` para desplegar la aplicación en el servidor:



Ahora ya está preparado para probar este servlet. Introduzca http://localhost:8000/lifeCycle/servlet/freak para cargar el servlet. Vuelva a cargar el URL algunas veces utilizando el botón `Actualizar` de su navegador. Hay dos tipos de respuestas que puede ver en el navegador. Verá la lista de estados en el ciclo de vida del `FreakServlet`, como sigue:



o verá la respuesta cuando el servlet introduzca el estado no disponible:



En ambos casos, haga clic en el vínculo `Actualizar` en el navegador para invocar este servlet varias veces. Vemos ahora cómo son generados estos mensajes.

Instanciación

Este estado corresponde a la construcción de la instancia del servlet por el contenedor. Remítase al constructor de `FreakServlet`:

```
public FreakServlet() {
    states = new java.util.Vector();
    random = new java.util.Random();
    waitInterval = DEFAULT_WAIT_INTERVAL;
    states.add(createState("Instantiation"));
}
```

Este constructor inicializa un vector y un generador de números aleatorios. En este ejercicio, el vector `states` es utilizado para almacenar los estados conformen se van sucediendo. Cada estado es representado como una cadena precedida por una cadena de fecha. El generador de números aleatorios es utilizado para lanzar de forma aleatoria `UnavailableException` desde el método `doGet()`.

Inicialización

Este es el segundo estado en cualquier servlet. Remítase al método `init()` del `FreakServlet`: