

LPS: Acceso a Bases de Datos



Federico Peinado
www.federicopeinado.es

Depto. de Ingeniería del Software e
Inteligencia Artificial
disia.fdi.ucm.es

Facultad de Informática
www.fdi.ucm.es

Universidad Complutense de Madrid
www.ucm.es

Bases de datos

- ◉ Base de datos (BBDD)
 - Conjunto de información organizada de forma independiente a su tratamiento y a los detalles de su almacenamiento físico
- ◉ Modelo de datos
 - Estructura lógica de los datos y conjunto de operaciones que permiten crearlos, consultarlos y modificarlos
 - El más utilizado es el relacional (datos organizados en tablas)
- ◉ Lenguaje de acceso
 - El más utilizado es SQL (creación, consulta y modificación de BBDD relacionales)
- ◉ Sistema gestor de bases de datos (SGBD)
 - Aplicación para crear, mantener y consultar BBDD
 - El código de las aplicaciones que usan BBDD no debería depender de los detalles particulares de los SGBDs
 - Necesidad de algún API estándar para conectar con SGBDs

Tecnologías de acceso

- ◉ Existen muchas, habitualmente cada proveedor de SGBDs ofrece una propia para acceder a sus BBDD
- ◉ Protocolos propietarios
 - Dependen del proveedor del SGBD
 - Dependen del lenguaje de programación que se quiera utilizar
- ◉ Plataforma Microsoft
 - ODBC (Open DataBase Connectivity)
 - Protocolo de conexión basado en parte del estándar SQL/CLI (API en C)
 - OLE/ADO DB
 - ADO .NET
 - ADO .NET + LINQ
- ◉ Plataforma Java
 - JDBC (Java DataBase Connectivity)

JDBC



JDBC

⊙ Paquete Java ubicado en `java.sql`

- Versiones 1.0, 2.0, 3.0 y 4.0
(la actual, ya incluida en el JDK 6)
- Proporciona una *pasarela JDBC* ⇔ *ODBC*

⊙ Mecanismos básicos

1. Conectarse a una *fuentes de datos*
 - Para ello la fuente de datos debe incluir un driver JDBC (u ODBC, necesitando entonces usar la pasarela)
2. *Consultar/cambiar tablas* en las BBDD de la fuente
 - Se usa SQL “embebido” en los parámetros (*Strings*) de ciertos métodos definidos en las clases de este paquete
 - El resultado son objetos Java, definidos en el paquete

Diseño de JDBC

◉ Cliente BD ↔ Servidor BD ↔ BD

- El cliente no necesita saber nada sobre como funcionan los controladores de BBDD
 - Programación independiente de la plataforma
- JDBC tiene dos capas
 1. API de JDBC
 2. API del Administrador de Controladores JDBC
 - Los distribuidores de BBDD deben construir sus controladores (*drivers*) siguiendo los requisitos de esta segunda API

Diseño de JDBC

- ◉ Debido a su diseño, los programadores sólo necesitan conocer la capa Java/JDBC
- ◉ Esquema de comunicación (entre JDBC y una base de datos)
 - Aplicación Java ↔ Administrador de Controladores de JDBC ↔
 1. ↔ Pasarela JDBC/ODBC ↔ Controlador ODBC ↔ Base de datos
 2. ↔ Controlador JDBC suministrado por el fabricante ↔ Base de datos

Controladores JDBC

◉ Controlador Tipo 1

- Traduce JDBC a ODBC y utiliza el controlador ODBC para comunicar con la base de datos
- Oracle incluye uno de estos controladores en el JDK: *la pasarela JDBC/ODBC*
- *Requiere la configuración específica de un controlador ODBC...*

◉ Controlador Tipo 2

- Escrito parcialmente en Java y en código nativo
- Comunica la API del cliente con la base de datos
- *Requiere instalación previa de software específico de la plataforma junto con una librería Java asociada...*

Controladores JDBC

- ◉ Controlador Tipo 3
 - Librería cliente de Java puro
 - Usa protocolo independiente de la BD para enviar peticiones al servidor (y luego éste traduce)
- ◉ Controlador Tipo 4
 - Librería de Java puro
 - Traduce peticiones JDBC a un protocolo de base de datos específico
- ◉ Las distribuidores de bases de datos más habituales suministran controladores tipo 3 ó 4 con sus productos
 - Otras compañías también los desarrollan
 - Estudiaremos cómo utilizar los de tipo 1 y 3

Programación con JDBC

- ◉ Hay varias alternativas, cada una con sus requisitos
- ◉ Oracle
 - Añadir librería con el controlador al proyecto
 - `ojdbc14_g.jar` (Oracle Instant Client)
- ◉ MySQL
 - Añadir librería con el controlador al proyecto
 - `mysql-connector-java-xx.jar` (MySQL Connector/J)
- ◉ Access
 - Configurar el controlador ODBC para poder acceder a la base de datos (*nombreControlador*)
 1. Panel de Control → Herramientas administrativas → Orígenes de datos (ODBC) → ODBC Data Source Administrator
 2. Añadir [Microsoft Access Driver (*.mdb)]
 3. Seleccionar base de datos

Programación con JDBC

◉ En el programa

- Registrar el controlador cargando su clase
 - `System.setProperty("jdbc.drivers", driver);`
 - Se pueden registrar varios drivers separados por ":"
 - `Class.forName(String driver)`
 - Registro manual
- Ejemplos
 - `sun.jdbc.odbc.JdbcOdbcDriver`
 - `oracle.jdbc.driver.OracleDriver`

Programación con JDBC

◉ Para conectar con la base de datos hay que especificar la fuente de datos mediante una sintaxis similar a una URL

- **`jdbc:nombreSubprotocolo:otrosElementos`**
 - `nombreSubprotocolo` selecciona el controlador concreto
 - `otrosElementos` depende del subprotocolo
- Ejemplos
 - `jdbc:oracle:thin:@ipServidor:puerto:nombreBD`
 - `jdbc:odbc:nombreControlador`

Programación con JDBC

◎ ODBC

- `Connection conn = DriverManager.getConnection(String url)`

◎ Oracle

- `Connection conn = DriverManager.getConnection(String url, String user, String password)`

◎ **conn.close()**

- Cierra la conexión
- Es necesario cerrar las conexiones una vez que se han terminado de utilizar para liberar recursos (tanto en la aplicación como en el servidor)

Programación con JDBC

- ◎ Usa archivos de configuración (tipo Properties)
 - Fichero `database.properties`
 - `jdbc.drivers, jdbc.url, jdbc.username, jdbc.password`
 - Ejemplo Oracle
 - `jdbc.drivers=oracle.jdbc.driver.OracleDriver`
 - `jdbc.url=jdbc:oracle:thin:@localhost:1521:test`
 - `jdbc.username=test`
 - `jdbc.password=test`
 - Ejemplo Access
 - `jdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver`
 - `jdbc.url=jdbc:odbc:TEST`
 - `jdbc.username=`
 - `jdbc.password=`

Ejecución de sentencias SQL

- ◉ Primero, hay que crear un objeto Statement (sentencia SQL)
 - `Statement stmt = conn.createStatement ();`
- ◉ Después podemos hacer consultas
 - `ResultSet executeQuery(String sql)`
 - Realiza “select”
 - Devuelve los registros en un ResultSet
 - `int executeUpdate(String sql)`
 - Realiza “insert, update, delete, create, drop...”
 - Devuelve *num* de registros afectados de 0 a 1
 - `boolean execute(String sql)`
 - El modo “auto-commit” funciona por defecto, es decir, cada sentencia se ejecuta dentro de una única transacción

Ejecución de sentencias SQL

© `java.sql.ResultSet`

- Tabla de resultados por filas
- cursor entre filas, inicialmente antes de la primera
- **`boolean next()`**
 - Pasa el cursor a la siguiente fila
 - ***false***, si se llega al final de la tabla
 - ***true***, caso contrario
- *Tipo* **`getTipo(int numColumna)`**
- *Tipo* **`getTipo(String nombreColumna)`**

Ejecución de sentencias SQL

```
/**
 * Tabla "Prueba" con columna "cadena" de tipo VARCHAR.
 * Seleccionar las columnas de la tabla prueba
 */
ResultSet rset = stmt.executeQuery ("SELECT * FROM
    Prueba");

/**
 * Iterar a lo largo de la tabla obtenida,
 * imprimiendo las tuplas
 */
while (rset.next ()) {
    System.out.println (rset.getString (1));
    //System.out.println (rset.getString ("cadena"));
}
// Liberación de recursos, tanto ResultSet como Connection
rset.close();
stmt.close();
```

Ejecución de sentencias SQL

◉ Información sobre objetos `java.sql.ResultSet`

- Interfaz `java.sql.ResultSetMetaData`

- `getColumnCount()` `numColumnas` → 1..n

- `columnName(i)`, `getColumnTypeName(i)`

◉ Información sobre tablas de usuario

```
DatabaseMetaData dbmd = conn.getMetaData();  
// MySQL, Access y HSQLDB  
ResultSet rs = dbmd.getTables( null, null,  
                               null, new  
                               String[] {"TABLE"} );  
  
// Oracle  
ResultSet rs = dbmd.getTables( null,  
                               "NombreUsuario",  
                               null, new  
                               String[] {"TABLE"} );
```

Sentencias preparadas

- ◉ En lugar de construir una sentencia cada vez que el usuario lanza una consulta, se puede preparar una vez y usarla muchas veces
 - Se gana en rendimiento ya que el servidor puede “cachear” el plan de ejecución de la consulta
 - Se gana en seguridad ya que los *drivers* habitualmente se encargan de procesar los parámetros de la consulta preparada para evitar ataques del tipo “SQL Injection”

```
String consulta = "SELECT precio FROM libros  
WHERE autor = ?";  
PreparedStatement stmtCP =  
    conn.prepareStatement(consulta);  
stmtCP.setString(1, "Pepe");  
ResultSet rs = stmtCP.executeQuery();
```

Resultados con desplazamiento

⊙ Para moverse hacia adelante y hacia atrás a través de los datos, e incluso, saltar a una posición concreta del conjunto de resultados

- `Statement stat = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY)`
- `ResultSet rs = stat.executeQuery(query);`
- `rs.previous()` → fila anterior
- `rs.relative(n)` → incremento relativo
- `rs.absolute(n)` → posición n
- `int n = rs.getRow()` → fila actual

Resultados actualizables

◉ Se pueden actualizar los datos obtenidos con `executeQuery`

- `Statement stat = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE)`
- `ResultSet rs = stat.executeQuery(query);`
 - `double precio = rs.getDouble("precio");`
`rs.updateDouble("precio", precio + incremento);`
 - `rs.updateRow();` → actualiza la base de datos
 - `rs.deleteRow();` → borra la fila actual de la base de datos
 - `rs.insertRow();` → inserta una fila

Mantenimiento de conexiones

- ⊙ Las conexiones a la BBDD son "costosas"
 - Cliente: El driver reserva una cantidad de memoria (a veces > 1 MB) para gestionar una conexión... además tiene que realizar cierto trabajo para obtener la conexión
 - Servidor: El número de conexiones está limitado → existe un número máximo de conexiones permitidas
- ⊙ Un programa sencillo establece una única conexión al comienzo del mismo y la cierra cuando termina
 - Demasiado costoso con múltiples consultas
- ⊙ Las conexiones están pensadas para ser utilizadas por múltiples consultas
 - Hay que utilizar una conexión para cada grupo de consultas
 - No hay que tener una única conexión permanente abierta
- ⊙ En las aplicaciones reales se suele utilizar un "pool" de conexiones para optimizar y gestionar adecuadamente las conexiones de BBDD

POO y persistencia

- ◎ La POO a veces necesita persistencia
- ◎ ¿Qué tipo de datos haríamos persistentes?
 - Objetos
 - Sería muy directo y útil
 - Las SGBDOO no están maduras, hay estándares pobres, etc.
 - Tablas (modelo relacional)
 - Hay SGBDs variados (pero necesitan desacoplamiento)
 - Las operaciones son siempre las mismas (hay uniformidad)
- ◎ Solución: ORM (Object/Relational Mapping)
 - Realiza la conversión de objetos (Java, principalmente) a tablas (datos en modelo relacional)
 - Transparente, automático, independiente de SGBD (portable), aumenta productividad, etc.

Armazones de persistencia

- ◉ Un armazón (*framework*) software es un conjunto de clases que proporciona funcionalidad genérica para solucionar un problema
 - No son solo librerías, imponen el flujo de control del programa
 - Se especializan/generalizan/personalizan según el problema particular
- ◉ Algunos armazones para hacer ORM
 - Hibernate
<https://www.hibernate.org>
 - Apache iBATIS
<http://ibatis.apache.org>
 - Enterprise JavaBeans
<http://java.sun.com/products/ejb>
 - iPersist
<http://ipersist.sourceforge.net>
 - Apache Cayenne
<http://cayenne.apache.org>

Críticas, dudas, sugerencias...



Federico Peinado
www.federicopeinado.es