Programación

Bloque 08 - Acceso a bases de datos relacionales

Índice

1.- Introducción

En bloques anteriores hemos aprendido a utilizar ficheros para persistir los datos de forma que sobrevivan a las ejecuciones del programa.

Esta técnica, sin embargo, exige que el programador realice "manualmente" gran parte de las labores.

Un método mejor para la persistencia es emplear un sistema de gestión de bases de datos. Un sistema de estas características ofrece una gran serie de ventajas para el programador:

- Gestión de los ficheros en disco
- Empleo de índices para búsquedas rápidas.
- Transacciones y aislamiento en caso de uso concurrente de los datos.
- Una librería de consulta más o menos potente que liberar al programador de tareas de bajo nivel.

En este bloque revisaremos el uso de bases de datos relacionales, que son las más extendidas y empleadas actualmente.

2.- Conceptos básicos

A fin de poder emplear bases de datos relacionales en nuestras aplicaciones debemos conocer la forma en que aquellas estructuran y almacenan la información. En esta sección describiremos los conceptos básicos y términos empleados en las bases de datos relacionales.

Las bases de datos relacionales almacenan la información empleando el concepto de *tabla*.

Una tabla es una estructura bidimensional en la cual las columnas representarían los distintos atributos o campos y las filas representan cada una de las instancias o registros. Si miramos una tabla fila a fila, cada una de ella contiene los distintos valores de los campos que corresponden a lo que clásicamente se conoce como un registro.

Las distintas columnas o campos tienen un tipo o dominio, que indica para cada una, el conjunto de datos válidos que pueden contenerse en las celdas de dicha columna. Ejemplos típicos son entero, real, texto, etc.

Las tablas normalmente se agrupan en estructuras denominadas bases de datos, esquemas, modelo de datos o catálogo, según el sistema de gestión de bases de datos.

Un Sistema de Gestión de Bases de Datos Relacionales (SGBDR) es un programa o conjunto de programas que se encargan de mantener y gestionar esta información.

El acceso a un SGBDR normalmente está restringido a ciertos usuarios cuyo acceso suele estar protegido con contraseña u otro mecanismo de autenticación similar

La lengua común para trabajar con estos SGBDR es el lenguaje SQL, que es más o menos estándar, aunque cada fabricante añade sus peculiaridades, añadiendo sus propios tipos de columna (INT en lugar de INTEGER, por ejemplo), sus propias extensiones al lenguaje con el uso de sentencias u opciones nuevas y su propia interpretación del funcionamiento de algunas sentencias. Por lo tanto, además de un conocimiento general de SQL hay que tener un conocimento específico del dialecto empleado por el SGBDR concreto que se va a emplear desde nuestra aplicación o utilizar una forma de SQL que sea lo más agnóstica posible sobre el SGBDR sobre el que se va a emplear, aunque esto no siempre es posible.

Otra opción, que no veremos en este bloque pero que sí se trata en otros módulos del ciclo es el empleo de librerías intermedias, por ejemplo Hibernate, que se encargan de "traducir" los comandos enviados al SGBDR para adaptarlos a las peculiaridades del mismo.

3.- Acceso a SGBDR en Java

Para acceder a los SGBDR, Java ofrece una librería o conjunto de clases denominado JDBC (Java DataBase Connectivity - Conectividad a Bases de Datos en Java)

JDBC tiene dos partes bien diferenciadas, lo que hace que sea versátil, aunque también algo más pesado de programar:

- Interfaz con la aplicación (API). Proporciona una serie de clases para realizar consultas, modificaciones y obtener y procesar resultados. Todas estas clases están localizadas en el paquete `java.sql`
- Controladores o drivers. Los controladores o drivers son librerías de clases que implementan el protocolo de comunicación con un SGBDR determinado. Implementan el protocolo necesario para intercambiar información con el SGBDR.

En las siguientes secciones describiremos ambas partes.

4.- Conexión con el SGBDR

Para poder conectar con un SGBDR determinado es necesario disponer del driver o controlador correspondiente al SGBDR y a la versión del mismo, ya que es posible que versiones nuevas introduzcan problemas o incompatibilidades en la comunicación.

4.1.- Configuración del driver

Los drivers se proporcionan habitualmente por el fabricante en formato JAR. Un driver hay que colocarlo de forma que el sistema Java lo pueda encontrar y cargar cuando lo necesite. En el caso de una aplicación en consola habría que colocarlo en el CLASSPATH. En caso de Eclipse se deben colocar en el Build Path, en las propiedades del proyecto. Si el sistema no es capaz de localizar las clases del driver no será capaz de conectar con la base de datos.

4.2.- URLs de JDBC

JDBC emplea, para establecer una conexión con un SGBDR, URLs (Uniform Resource Locator) al mismo estilo que se emplean para localizar páginas web.

El formato exacto de la URL depende del SGBDR en cuestión pero suele seguir la forma más o menos que sigue:

```
jdbc:sgbdr://host?parametros
```

donde:

- `jdbc` es fijo e indica que la URL es específica de JDBC.
- `sgbdr` es un nombre que corresponde al sgbdr a emplear (oracle, mysql, etc.)
- `host` es el nombre o IP del host donde está localizado el SGBDR
- `parametros` son parámetros variados de la conexión (nombre de la base de datos, usuario y password, etc.)

4.3.- Establecimiento de conexión

Para establecer una conexión se emplea la clase `DriverManager` que es la clase que se encarga de gestionar la conexión de la aplicación con el driver JDBC adecuado.

De los métodos que ofrece, el más útil para nosotros es `getConnection()` que proporciona una conexión a un SGBDR determinado. Para ello recibe una URL de conexión y opcionalmente parámetros.

Este método obtiene un objeto de la clase `Connection`, que representa una sesión o conexión con la base de datos. Este objeto se empleará a continuación para lanzar comandos al SGBDR.

Si no se puede establecer una sesión con el SGBDR (por muchas y múltiples razones) se lanzará una excepción (chequeada) `SQLException`.

4.3.1.- Ejemplo de conexión: MariaDB

MariaDB es un clon (fork) de MySQL que se realizó después de que Oracle comprara Sun Microsystems, empresa que mantenía MySQL (y Java, por cierto), temiendo que Oracle cancelara el proyecto debido a la competencia que pudiera hacerle a sus propios productos de bases de datos. Aunque el temor se demostró infundado, hoy en día MariaDB es la opción preferida por los defensores del software libre y lo podemos encontrar en la mayoría de distribuciones Linux susituyendo a MySQL.

En MySQL la URL básica tiene el formato:

```
jdbc:mariadb://host:puerto/basedatos?user=usuario&password=pass
```

donde:

- `host` es el host en el que está localizada la instancia de MariaDB a la que nos queremos conectar. Si está en el mismo ordenador se puede emplear localhost o 127.0.0.1
- `puerto` es el puerto en el que escucha MariaDB. Usualmente es el 3306 y en este caso no hace falta indicarlo (ni los dos puntos situados delante) pero si se está empleando cualquier otro puerto se debe indicar
- `basedatos` es la base de datos (esquema) que contiene las tablas a las que queremos acceder. Sólo se puede acceder a las tablas de un esquema con una conexión. Si se quieren acceder a tablas de otro esquema hay que emplear otra conexión al mismo servidor. Si el esquema no existe se producirá un error de conexión.
- `usuario` es el nombre de usuario a emplear para la conexión
- `pass` es la password correspondiente al usuario anterior.

Por ejemplo, para conectar a la base de datos `clientes`, localizada en el host `db.mysite.org`, con usuario `cliente` y password `12345`, la url de conexión debería ser:

```
jdbc:mariadb://db.mysite.org/clientes?user=cliente&password=12345
```

4.3.2.- Ejemplo de conexión: SQLite

SQLite es un SGBDR peculiar en el sentido de que es una librería en lugar de un programa completo. La librería implementa un motor SQL completo que simula un SGBDR. La base de datos completa está contenida en un único archivo.

SQLite es el "SGBDR" más utilizado del mundo ya que está incluido en todos los teléfonos móviles y viene "incrustado" en muchas aplicaciones que lo emplean para sus necesidades de almacenamiento de datos en lugar de recurrir a pesados y a veces costosos SGBDR.

La URL de SQLite, como el motor en si, es muy simple:

```
jdbc:sqlite:ruta
```

donde `ruta` es la ruta al archivo que contiene la base de datos.

Por ejemplo, para conectar con la base de datos contenida en el archivo `clientes.db` (es convención emplear la extensión `.db` para las bases de datos SQLite) que está almacenado dentro de la carpeta `database`, la URL sería:

```
jdbc:sqlite:database/clientes.db
```

4.4.- Cierre de conexiones

A fin de conservar recursos y evitar problemas de funcionamiento por agotamiento de los mismos es necesario el cerrar una conexión cuando ya no se necesite más.

Esto se puede hacer de forma manual empleando el método `close()` de la clase `Connection` o se puede hacer de forma automática abriendo la conexión dentro de un try con recursos, ya que `Connection` implementa el interfaz `AutoCloseable`.

5.- Sentencias

Una vez que tenemos establecida una conexión, la interacción con el SGBDR se realiza mediante *sentencias*. Una sentencia es un comando que se manda a la base de datos y que puede tener o no un resultado.

El resultado de una sentencia puede variar desde ninguno (cuando se llama a un procedimiento almacenado, por ejemplo) a un número (cuando la sentencia es UPDATE o DELETE) o una tabla (cuando la sentencia es SELECT). Mas adelante veremos como tratar los resultados.

Las sentencias pueden ser de tres tipos:

- Sentencias "normales". En este caso la instruccion SQL se debe proporcionar completa en formato cadena. Es la forma más rápida pero presenta algunos problemas de seguridad. Implementan el interfaz `Statement`
- Sentencias "preparadas". En este caso la instrucción SQL se proporciona con "huecos" en los lugares en que más tarden se incluirán parámetros de la consulta. Estos huecos funcionan de forma similar a los marcadores en `printf` o `String.format`. Posteriormente se rellenarán esos huecos y se colocarán los valores reales a emplear, antes de enviar la sentencia al SGBDR. Implementan el interfaz `PreparedStatement` (que hereda de `Statement`)
- Sentencias "llamables". Implementan el interfaz `CallableStatement` (que también hereda de `Statement`) y se emplean para invocar procedimientos almacenados. Proporcionan funcionalidad para pasar los parámetros necesarios a estos procedimientos y obtener los resultados. Este tipo de sentencias no las vamos a detallar en esta documentación

Para trabajar con sentencias hay que obtener un objeto que implemente el interfaz deseado (`Statement` o `PreparedStatement`). Estos objetos se obtienen a partir de la conexión y deben ser cerrados también una vez no se necesiten para evitar pérdida de recursos. **Al cerrar una conexión no se cierran las sentencias que se han creado a partir de ellas. Hay que cerrarlas manualmente.**

Para cerrar una conexión podremos emplear las mismas técnicas que para las conexiones (cerrado manual o try con recursos)

5.1.- Sentencias normales

Las sentencias normales son objetos que implementan el interfaz `Statement`.

Para obtener estos objetos hay que emplear un objeto `Connection` ya existente.

Para crear una sentencia emplearemos el método `createStatement()` de la clase `Connection`:

```
Statement sentencia = conexion.createStatement();
```

Hay que hacer notar que la sentencia se crea sin ningún SQL en concreto. Esto se indicará más tarde.

Para realizar una consulta se emplea el método executeQuery, de la forma:

```
String sql = "SELECT * FROM Clientes WHERE DNI='11111111A'";
ResultSet resultado = sentencia.executeQuery(sql);
```

`executeQuery()` toma como parámetro una cadena que debe contener una sentencia SQL `SELECT`. Cualquier otra sentencia produciría un error. El método devuelve un objeto que implementa el interfaz `ResultSet`. Este objeto se emplearía para recorrer el resultado de la consulta y extraer información del mismo. Más adelante veremos cómo se emplea este interfaz.

Cualquier error en la sentencia SQL provocará una excepción `SQLException`. No se considera un error que una consulta correcta no devuelva ningún resultado, por lo que en este caso NO se lanzará una excepción.

Si la sentencia SQL no es una consulta (`SELECT`) debemos utilizar el método `executeUpdate()` en lugar de `executeQuery()`.

```
String sql = "UPDATE Clientes SET nombre = 'Paco' WHERE DNI='11111111A'";
int resultado = sentencia.executeUpdate(sql);
```

`executeUpdate()` toma una cadena con la sentencia, como `executeQuery()` pero devuelve un entero. Éste número indica el número de filas que se han visto afectadas por la sentencia. Por ejemplo, si la sentencia era `DELETE` y se han eliminado 8 filas, `executeUpdate()` devolverá 8.

Se da un caso especial cuando la sentencia es `INSERT` y la tabla sobre la que opere la sentencia tenga campos autogenerados (o autoincrementales). En este caso se suele requerir el conocer inmediatamente los valores que ha asignado el SGBDR a estos campos autogenerados. Para ello hay que emplear una variante de `executeUpdate()` de la forma:

```
String sql = "INSERT INTO Paquetes (nombre) VALUES ('Juan Vazquez')";
int resultado = sentencia.executeUpdate(sql, Statement.RETURN_GENERATED_KEYS);
```

En este caso, la tabla `Paquetes` contendría un campo, `id`, autogenerado. Al insertar el nuevo registro no proporcionamos valor a esta campo dado que el SGBDR lo generará automáticamente. Si queremos obtener el valor que se ha generado empleamos la variante de `executeUpdate()` que toma dos parámetros.

Para acceder al valor del campo generado hay que emplear justo a continuación de las dos líneas anteriores una tercera:

```
ResultSet claves = sentencia.getGeneratedKeys();
```

Esto obtendría un `ResultSet` que podemos emplear para consultar los valores de las campos que se acaban de autogenerar. Volveremos sobre esto más adelante cuando veamos los `ResultSet`.

Un problema que presentan este tipo de sentencias es que la inclusión de parámetros requiere de métodos de tratamiento de cadenas, lo cual puede resultar en un problema de seguridad (ataques de tipo SQL Injection) si las cadenas no se "sanitizan" antes.

Supongamos que tenemos un metodo `getUsuarioById`, que recibe un `id` entero y localiza al usuario cuyo campo `ID` vale precisamente este valor. Esta es una operación muy frecuente al trabajar con bases de datos. Este método, como parte de su implementación, debería emplear una sentencia SQL en la cual una parte no es fija sino que depende del parámetro `id`. Habría que hacer algo asi como:

```
String sql = "SELECT * FROM Usuario WHERE id = " + id;
```

o

```
String sql = String.format("SELECT * FROM Usuario WHERE id = %d",
id);
```

El problema se presenta cuando el parámetro es de tipo texto. Deberíamos hacer algo como:

```
String sql = "SELECT * FROM Usuario WHERE nombre LIKE '%" + nombre
+ "%'";
```

o

```
String sql = "SELECT * FROM Usuario WHERE nombre LIKE '%%%s%%'",
nombre);
```

El problema viene si un usuario malicioso prepara un nombre tal que asi:

```
String nombre = "';DELETE * FROM Usuario";
```

En este caso, al unir las dos cadenas tendríamos:

```
SELECT * FROM Usuario WHERE nombre LIKE '%';DELETE * FROM Usuario;
%'
```

Que en realidad son dos sentencias. La primera es un `SELECT` y la segunda es un `DELETE` que elimina por completo todos los registros de la tabla `Usuario`

Para evitar este tipo de problemas es conveniente emplear:

```
Statement.enquoteLiteral()
```

Este método toma cualquier cadena y la "sanitiza" de forma que las comillas y otros carácteres especiales se escapan de forma que no puedan interferir con la ejecución normal de las sentencias. En nuestro caso haríamos

```
String sql = "SELECT * FROM Usuario WHERE nombre LIKE '%" +
Statement.enquoteLiteral(nombre) + "%'";
```

o

```
String sql = "SELECT * FROM Usuario WHERE nombre LIKE '%%%s%%'",
Statement.enquoteLiteral(nombre));
```

5.2.- Sentencias preparadas

Estas sentencias que, como recordarás, implementan el interfaz `PreparedStatement`, permiten especificar de una forma descriptiva las sentencias SQL.

Para crear un objeto que implemente el interfaz `PreparedStatement` hay que emplear el método `prepareStatement` de la clase `Connection` en la forma:

```
PreparedStatement sentencia = conexion.prepareStatement(sql);
```

o

```
PreparedStatement sentencia = conexion.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS)
```

Esta última forma hay que emplearla si la sentencia provoca la generación de valores autoincrementales o autogenerados y se desea recuperarlos (ver sección anterior)

En este caso, la sentencia SQL emplea un formato especial ya que se pueden incorporar en lugares que necesiten un valor el carácter especial `?` en lugar del mismo. Estos sitios donde aparecen los caracteres `?` se denominan huecos (placeholders en inglés)

De esta manera se especifica a la librería sítios dentro de la sentencia que pueden variar. Posteriormente se deberán dar valores específicos para estos sitios antes de enviar la sentencia al SGBDR.

Por ejemplo, si recordamos uno de los ejemplos de la sección anterior tendríamos:

```
String sql = "SELECT * FROM Usuario WHERE id = " + id;

ResultSet rs = sentencia.executeQuery(sql);
```

Empleando `PreparedStatement` se haría de la siguiente forma:

```
String sql = "SELECT * FROM Usuario WHERE id = ?";

PreparedStatement sentencia = conexion.prepareStatement(sql);

// Aqui se asignaría valor al primer (y unico hueco)

ResultSet rs = sentencia.executeQuery();
```

Como se puede ver en la primera línea se especifica la sentencia pero el lugar que ocuparía el ID real a buscar se sustituye por el carácter `?`.

Mas tarde se especificará el valor que va a ocupar ese hueco ("rellenandolo") antes de enviar la sentencia.

Antes de ver cómo se rellenan los huecos conviene indicar que los métodos para enviar las sentencias son ligeramente distintos a los vistos para `Statement`. En este caso tenemos `executeQuery()` y `executeUpdate()` pero sin parámetros, ya que no se necesitan puesto que el SQL se ha especificado al crear la sentencia. Lo que devuelven ambos y su significado sigue siendo el mismo, sin embargo.

Para rellenar los huecos, el interfaz `PreparedStatement` especifica una serie de métodos `setXXX`, donde XXX es un tipo de columna (`Int`, `String`, `Float`, `Double`, etc). Concretamente lo que vamos a emplear en este curso son:

- `setBoolean`, para especificar un valor booleano (`true` o `false`)
- `setDouble`, para especificar un valor real.
- `setInt`, para especificar un entero
- `setLong`, para especificar un entero grande (64 bits)
- `setNull`, para especificar que el parámetro vale NULL
- `setString`, para especificar una cadena

Todos los métodos tienen dos parámetros. El primero es un entero que indica la posición del "hueco" que se está especificando con el método. **Cuidado porque se comienza por 1, esto es, el primer hueco tiene la posición 1, no la 0 como estamos acostumbrados.** Si se especifica una posición incorrecta obtendremos `SQLException`.

El segundo parámetro es el valor a especificar para el hueco y será del tipo compatible. La excepción es `setNull`, en el cual el segundo parámetro debe ser uno de los valores especificados en la clase `java.sql.Types`. Hay que especificar el valor correspondiente al valor del campo o columna donde se va a emplear.

Cualquiera de los `setXXX` puede lanzar `SQLException` si se intenta dar un valor que no es compatible con el hueco en cuestión, según las reglas del SQL del SGBDR empleado.

El uso de `PreparedStatement` es más cómodo que el de las sentencias "normales", especialmente cuando se trabaja con un número de campos o columnas mayor. Nosotros lo vamos a emplear como método preferente.

6.- Proceso de resultados

Cuando se ejecutan consultas (Sentencias SELECT), se obtiene un resultado que es una tabla virtual, esto es, una tabla que sólo existe momentaneamente hasta que se finaliza su procesamiento, momento en que se descarta.

En JDBC esta tabla virtual se modela mediante el interfaz `ResultSet`. El interfaz `ResultSet` permite recorrer esta tabla virtual extrayendo los valores de las distintas "celdas" del resultado.

Un `ResultSet` funciona como un cursor. La tabla virtual se va recorriendo fila a fila (no se puede acceder a toda la tabla simultáneamente). Cuando se está en una fila se puede acceder directamente a los contenidos de las celdas de esa fila.

El movimiento entre filas se realiza siempre hacia adelante empleando el método `next()`. `next()` avanza el cursor a la siguiente fila y devuelve un valor booleano. Si este valor es true indica que el cursor se ha situado en una fila válida. Si es false indica que NO está en una fila válida (lo que indica que se ha llegado al final de la tabla). Un detalle importante es que un `ResultSet` recién creado no tiene el cursor situado en la primera fila sino **antes** de la primera fila, por lo que hay que emplear `next()` para colocarlo en la primera fila. Si el primer `next()` devuelve false significa que el resultado no contiene ninguna fila, lo cual es un resultado perfectamente válido ya que no todas las consultas devuelven filas.

Una vez el cursor está en una fila, `ResultSet` proporciona una serie de métodos `getXXX` para acceder a los datos de dicha fila, donde XXX se corresponden con los tipos de las columnas a leer. Los que emplearemos nosotros son:

- `getBoolean`, para leer un campo booleano (true o false)
- `getDouble`, para leer un campo real.
- `getInt`, para leer un campo entero
- `getLong`, para leer un campo entero largo
- `getString`, para leer una cadena

Todos los métodos tienen dos versiones. Una de ellas recibe como parámetro un entero y la otra recibe como parámetro una cadena. En la versión con entero, este especifica la posición, dentro del resultado, de la columna correspondiente a la celda a leer, **comenzando por 1, no por 0**. Así, para leer el dato de la primera columna (de la fila actual) se emplearía 1, 2 para la segunda y así sucesvivamente. En la versión con cadena, ésta especifica el *nombre* de la columna a leer, tal y como se ha definido en la sentencia SELECT que generó el resultado. Cuidado cuando una columna sea calculada o cuando tenga el mismo nombre en varias tablas en la misma consulta, ya que a veces el nombre no es intuitivo. Para paliar este problema se recomienda utilizar alias en las columnas (clausula AS) para dar nombres más sencillos de usar en las consultas.

Un caso especial que hay que procesar con cuidado y cuyo funcionamiento no es intuitivo es cuando estamos leyendo un campo nulable, esto es, un campo que admite NULL como valor. En estos casos, ya que los campos numéricos, de carácter y booleanos no admiten valores null, no se puede emplear simplemente `getXXX` ya que este método no puede devolver valores null. Cuando empleemos `getString` no tenemos problema puesto que éste método si puede devolver un valor `null`. Por lo tanto, cuando empleemos `getInt`, `getDouble`, `getBoolean` o `getLong` sobre campos que permitan contener valores nulos deberemos emplear el método `wasNull()` **justo después de leer el campo empleando `getXXXX`**. `wasNull`() devolverá `true` si el campo recién leido valía NULL en realidad o `false` si no valía NULL (en cuyo caso el valor leido mediante `getXXXX` es correcto). Es un procedimiento un poco extraño pero es el que tenemos, asi que aplícalo con cuidado.

Los `ResultSet` hay que cerrarlos una vez se ha terminado con ellos para liberar recursos, ya sea empleando el método `close()` o usándolos dentro de un `try` con recursos.

Otro punto donde hay que ser cuidadoso es que, normalmente, cuando una sentencia devuelve un nuevo `ResultSet`, éste invalida el anterior obtenido mediante dicha sentencia. Por lo tanto, si no se ha cerrado es posible que nos lance una excepción al intentar interactuar con él.

Otra derivada de esto último es que si se necesita trabajar con dos (o mas) `ResultSet` al mismo tiempo, cada uno de ellos debe provenir de una sentencia distinta o pueden ocurrir problemas inesperados.

6.1.- Proceso de claves autogeneradas

Como comentamos anteriormente, en el caso de campos autogenerados o autoincrementales, es posible obtener los valores que el SGBDR ha generado automáticamente después de hacer una sentencia `INSERT`, empleando `Statement.RETURN_GENERATED_KEYS`

Una vez que ejecutamos la sentencia, podemos emplear `getGeneratedKeys()` para acceder a un `ResultSet` con los valores de las claves generadas. Este resultset es algo particular ya que no se corresponde con una consulta SQL que hayamos escrito nosotros sino que se recupera desde el driver JDBC.

Este `ResultSet` tendrá una única fila y tantas columnas como campos autogenerados se hayan visto involucrados en la sentencia `INSERT` de la que procede (normalmente será uno pero podrían ser mas).

Asimismo no conocemos los nombres de las columnas por lo que hay que acceder a ellas por posición.

7.- Referencias

- Documentación del paquete java.sql en JDK 17 (https://docs.oracle.com/en/java/javase/17/docs/api/java.sql/java/sql/package-summary.html)
- Formato de la URL JDBC para diferentes SGBDR (https://www.baeldung.com/java-jdbc-url-format)