JDBC 中驱动加载的过程分析（上）

江苏 无锡 缪小东

本篇从 java.sql.Driver 接口、java.sql.DriveManager 类以及其它开源数据库的驱动类讨论 JDBC 中驱动加载的全过程以及 JDBC 的 Framework 如何做到“可插拔”的细节。

本篇包含了很多部分的内容。如类加载器、本地方法、对象锁、类锁、按功能或者状态分离锁、安全机制，对这些内容没有深入讨论！详情可以继续关注本博客！我在上篇主要关注驱动管理器的初始化、连接的建立、驱动的注册、驱动的遍列、驱动的取消注册以及 DriverManager 中的日志操作。

一、Driver 接口

```
//Driver.java
package java.sql;
public interface Driver {
    Connection connect(String url, java.util.Properties info) throws SQLException;
    boolean acceptsURL(String url) throws SQLException;
    DriverPropertyInfo[] getPropertyInfo(String url, java.util.Properties info) throws SQLException;
    int getMajorVersion();                    //返回驱动的主版本号
    int getMinorVersion();                    //返回驱动的次版本号
    boolean jdbcCompliant();                  //是否兼容于 JDBC 标准
}
```

以上就是 JDBC 中的 Driver 接口，它是任何数据库提供商的驱动类必须实现的接口，驱动类必须实现该接口中的所有方法！简单吧！

它之所以是一个接口，就是 OO 中经常谈到的“依赖倒转原则（DIP－Dependence Inverse Principle）”的具体应用了！在 DriverManager 类中可以看到：它使用的驱动都是 Driver 接口，从而依赖与高层，不依赖于实现。这样可以使用 JDBC Framework 管理和维护不同 JDBC 提供商的数据库驱动。

JDBC Framework 中允许加载多个数据库的驱动！相应地，一般建议数据库提供商的驱动必须小一点，从而保证在加载多个驱动后不会占用太多的内存。

在 Driver 接口中共有以上六个方法。其中红色的两个相对很重要，它是供 DriverManager 调用的，其它四个很简单的方法！下面简单讲述前两个方法的意义！

Connection connect(String url, java.util.Properties info) throws SQLException 方法是数据库提供商的驱动必须实现的方法，它主要是使用指定的 URL 和与具体提供商相关的信息建立一个连接。

boolean acceptsURL(String url) throws SQLException 方法也是数据库提供商的驱动必须实现的方法，主要判定某个该驱动是否介绍该 URL。（一般在建立连接之前调用。详情将 DriverManager 类）

二、DriverManager 类

DriverManager 类是整个 JDBC 的起点！利用它可以创建连接，从而完成后续的操作。在 JDBC 中 DriverManager 是一个相对比较复杂的类，因此我们按其只能分为几类介绍。本篇将 DriverManager 中的方法分为 3 类：1.初始化；2.驱动的注册、查询、取消注册；3.建立连接；4.日志相关。

下面就看看它的源代码吧！

```
//DriverManager.java      1.45 05/11/17
```

```
package java.sql;
import sun.misc.Service;
import java.util.Iterator;

class DriverInfo {
    Driver          driver;
    Class           driverClass;
    String          driverClassName;
    public String toString() {
       return ("driver[className=" + driverClassName + "," + driver + "]");
    }
}

public class DriverManager {
    // Prevent the DriverManager class from being instantiated.
    private DriverManager(){}
```

以上是其代码的前面部分。主要是包的定义、相关文件的导入、类的定义以及一个私有化的构造器——即该类不可以实例化，只可以调用其静态方法，相当于一个工具类——一个管理驱动的工具类！还有一个就是一个辅助类 DriverInfo，它包装了驱动类 Driver，包含驱动类的类和驱动类的名称。

下面就开始 DriverManager 类重要方法的介绍吧！

1.初始化

```
private static java.util.Vector drivers = new java.util.Vector();    //保存多个驱动的聚集
private static boolean initialized = false;                          //是否初始化的标记，默认当然是否了

//  真正的初始化方法
static void initialize() {
    if (initialized) {      return;      }          //已经初始化就返回！（初始化了就算了）
    initialized = true;                            //设置此标识符，表示已经完成初始化工作
    loadInitialDrivers();                          //初始化工作主要是完成所有驱动的加载
    println("JDBC DriverManager initialized");
}

//初始化方法中完成加载所有系统提供的驱动的方法
private static void loadInitialDrivers() {
    String drivers;
    try {
       drivers = (String) java.security.AccessController.doPrivileged(
                                                  new sun.security.action.GetPropertyAction("jdbc.drivers"));
            //得到系统属性"jdbc.drivers"对应的驱动的驱动名（这可是需要许可的哦！）。
    } catch (Exception ex) {
          drivers = null;
    }
    Iterator ps = Service.providers(java.sql.Driver.class);          //从系统服务中加载驱动
```

```
        while (ps.hasNext()) {                                  //加载这些驱动，从而实例化它们
            ps.next();
        }
        println("DriverManager.initialize: jdbc.drivers = " + drivers);
        if (drivers == null) {      return;          }          //系统属性未指定驱动则返回
        while (drivers.length() != 0) {                         //循环过程，讲解见下面
            int x = drivers.indexOf(':');
            String driver;
            if (x < 0) {
                driver = drivers;
                drivers = "";
            } else {
                driver = drivers.substring(0, x);
                drivers = drivers.substring(x+1);
            }
            if (driver.length() == 0) {       continue;       }
            try {
                println("DriverManager.Initialize: loading " + driver);
                Class.forName(driver, true, ClassLoader.getSystemClassLoader());
                        //加载这些驱动，下篇会讲解其细节
            } catch (Exception ex) {
                println("DriverManager.Initialize: load failed: " + ex);
            }
        }//end of while
        //系统属性"jdbc.drivers"可能有多个数据库驱动，这些驱动的名字是以“:”分隔开的，
        //上面的过程就是将此以“:”分隔的驱动，依次遍列，然后调用 Class.forName 依次加载
}


private static Object logSync = new Object();              //对象锁

//下面是一个辅助方法，用于向日志中写入信息！
public static void println(String message) {
   synchronized (logSync) {              //很重要的一致性编程的方法，见下面
        if (logWriter != null) {                    //设置日志才可以进行下面的写入信息
            logWriter.println(message);             //向 logger 中写入信息
            logWriter.flush();
        }
   }
}
//以上蓝色的属性和方法，是一致性编程（Concurent Programming）中的重要方法。
//首先明确我们在向日志写入信息的时候，是可以调用其它非写入日志的方法的，
//只是不同的客户不能同时调用该写入方法－－一个客户正在写入，其它必须等待写完
//假如我们机械地使用 synchronized(this)或 synchronized 该写入方法时，必然会导致效率低
//一般地，当类的中多个方法可以分为多个不同组，这些组的方法互相之间不干扰时，
//可以为每个组指定一个自己的锁，限制同一个方法被多个客户使用，从而保证该方法的
//一致性，保证无必要的 synchronized 方法！
```

```
//关于一致性编程，请多关注博客中的文章
```

2.驱动的注册、查询、取消注册

```
//向 DriverManager 注册指定的驱动。驱动这么注册请阅读下篇！
        public static synchronized void registerDriver(java.sql.Driver driver) throws SQLException {
            if (!initialized) {         initialize(); }          //注册前必须初始化了
            DriverInfo di = new DriverInfo();                    //创建一个新的驱动信息类
            di.driver = driver;
            di.driverClass = driver.getClass();
            di.driverClassName = di.driverClass.getName();       //以上填入注册驱动的信息
            drivers.addElement(di);                              //将此驱动信息假如驱动聚集中
            println("registerDriver: " + di);
        }

//
        public static synchronized Driver getDriver(String url) throws SQLException {
            println("DriverManager.getDriver(\"" + url + "\")");
            if (!initialized) {         initialize();            }       //同样必须先初始化
            //本地方法，得到调用此方法的类加载器
            ClassLoader callerCL = DriverManager.getCallerClassLoader();
            // 遍列所有的驱动信息，返回能理解此 URL 的驱动
            for (int i = 0; i < drivers.size(); i++) {                   //遍列驱动信息的聚集
                DriverInfo di = (DriverInfo)drivers.elementAt(i);
                // 调用者在没有权限加载此驱动时会忽略此驱动
                if ( getCallerClass(callerCL, di.driverClassName ) != di.driverClass ) {
                    println("       skipping: " + di);
                    continue;
                }
                try {
                    println("       trying " + di);
                    if (di.driver.acceptsURL(url)) {                     //驱动能理解此 URL 时，返回此驱动
                        println("getDriver returning " + di);
                        return (di.driver);
                    }
                } catch (SQLException ex) {
                    // Drop through and try the next driver.
                }
            }
            println("getDriver: no suitable driver");
            throw new SQLException("No suitable driver", "08001");
        }

//从 DriverManager 中取消注册某个驱动。Applet 仅仅能够取消注册从它的类加载器加载的驱动
        public static synchronized void deregisterDriver(Driver driver) throws SQLException {
```

```
            ClassLoader callerCL = DriverManager.getCallerClassLoader();
            println("DriverManager.deregisterDriver: " + driver);
            int i;
            DriverInfo di = null;
            for (i = 0; i < drivers.size(); i++) {
                di = (DriverInfo)drivers.elementAt(i);
                if (di.driver == driver) {      break;      }          //找到了某个驱动则返回，同时返回 i 值
            }
            if (i >= drivers.size()) {                                  //全部遍列完，度没有找到驱动则返回
                println("       couldn't find driver to unload");
                return;
            }
            //找到此驱动，但调用者不能加载此驱动，则抛出异常
            if ( getCallerClass(callerCL, di.driverClassName ) != di.driverClass ) {
                throw new SecurityException();
            }
            // 在以上所有操作后，可以删除此驱动了
            drivers.removeElementAt(i);
        }

//得到当前所有加载的 JDBC 驱动的枚举**
        public static synchronized java.util.Enumeration getDrivers() {
            java.util.Vector result = new java.util.Vector();
            if (!initialized) {         initialize();          }       //该类没有初始化时，必须完成初始化工作
                                                                        //详情请阅读初始化部分
            ClassLoader callerCL = DriverManager.getCallerClassLoader();       //得到当前类的类加载器
            for (int i = 0; i < drivers.size(); i++) {                          //遍列所有的驱动
                DriverInfo di = (DriverInfo)drivers.elementAt(i);               //得到某个具体的驱动
                // 假如调用者没有许可加载此驱动时，忽略该驱动
                if ( getCallerClass(callerCL, di.driverClassName ) != di.driverClass ) {
                    println("       skipping: " + di);
                    continue;
                }
                result.addElement(di.driver);               //将可以加载的驱动加入要返回的结果集
            }
            return (result.elements());                     //返回结果集
        }

    private static native ClassLoader getCallerClassLoader();
    //获得当前调用者的类装载器的本地方法（关于本地方法 JNI 请关注本博客后续文章）

// 返回类对象。我们使用 DriverManager 的本地方法 getCallerClassLoader()得到调用者的类加载器
    private static Class getCallerClass(ClassLoader callerClassLoader, String driverClassName) {
            // callerClassLoader 为类加载器，driverClassName 为驱动的名称
        Class callerC = null;
        try {
            callerC = Class.forName(driverClassName, true, callerClassLoader);
```

```
            //使用指定的类装载器定位、加载指定的驱动类，
            //true 代表该驱动类在没有被初始化时会被初始化，返回此类
        }catch (Exception ex) {
            callerC = null;                    // being very careful
        }
        return callerC;
    }
```

3.建立连接

在 JDBC 程序中一般使用 DriverManager.getConnection 方法返回一个连接。该方法有多个变体，它们都使用了具体驱动类的 connect 方法实现连接。下面是连接的核心方法。

```
private static Connection getConnection(String url, java.util.Properties info, ClassLoader callerCL)
                                                                                    throws SQLException {
    //当类加载器为 null 时，必须检查应用程序的类加载器
    //其它在 rt.jar 之外的 JDBC 驱动类可以从此加载驱动    /*
    synchronized(DriverManager.class) {                //同步当前 DriverManger 的类
        if(callerCL == null) {        callerCL = Thread.currentThread().getContextClassLoader();      }
        //得到当前线程的类加载器（此句的真正含义请关注后续线程相关的文章）
    }
    if(url == null) {        throw new SQLException("The url cannot be null", "08001");     }
    println("DriverManager.getConnection(\"" + url + "\")");
    if (!initialized) {        initialize();    }            //必须初始化，将默认的驱动加入
    //  遍列当前的所有驱动，并试图建立连接
    SQLException reason = null;
    for (int i = 0; i < drivers.size(); i++) {
        DriverInfo di = (DriverInfo)drivers.elementAt(i);
        //  假如调用者没有许可加载该类，忽略它
        if ( getCallerClass(callerCL, di.driverClassName ) != di.driverClass ) {
            //当驱动不是被当前调用者的类加载器加载时忽略此驱动
            println("        skipping: " + di);
            continue;
        }
        try {
            println("        trying " + di);
            Connection result = di.driver.connect(url, info);          //调用某个驱动的连接方法建立连接
            if (result != null) {                                    //在建立连接后打印连接信息且返回连接
                println("getConnection returning " + di);
                return (result);
            }
        } catch (SQLException ex) {
            if (reason == null) {        reason = ex;          }            //第一个错误哦
        }
    }
```

```
        //以上过程要么返回连接，要么抛出异常，当抛出异常会给出异常原因，即给 reason 赋值
        //在所有驱动都不能建立连接后，若有错误则打印错误且抛出该异常
        if (reason != null)       {
            println("getConnection failed: " + reason);
            throw reason;
        }
        //若根本没有返回连接也没有异常，否则打印没有适当连接，且抛出异常
        println("getConnection: no suitable driver");
        throw new SQLException("No suitable driver", "08001");
    }

    //以下三个方法是上面的连接方法的变体，都调用了上面的连接方法
        public static Connection getConnection(String url, java.util.Properties info) throws SQLException {
            ClassLoader callerCL = DriverManager.getCallerClassLoader();
                //没有类加载器时就是该调用者的类加载器
            return (getConnection(url, info, callerCL));
        }

        public static Connection getConnection(String url, String user, String password) throws SQLException {
            java.util.Properties info = new java.util.Properties();
            ClassLoader callerCL = DriverManager.getCallerClassLoader();
            if (user != null) {       info.put("user", user);  }
            if (password != null) {         info.put("password", password);  }
            return (getConnection(url, info, callerCL));
        }

        public static Connection getConnection(String url) throws SQLException {
            java.util.Properties info = new java.util.Properties();
            ClassLoader callerCL = DriverManager.getCallerClassLoader();
            return (getConnection(url, info, callerCL));
        }
```

4.日志相关

DriverManager 中与日志相关的方法有好几个，主要分为已被 deprecated 的 Stream 相关的方法，和建议使用的 Reader、Writer 相关的方法。（对应于 java IO 的字符流和字节流哦！因为写入日志的信息一般为字符流，所以废弃了与字节流相关的方法）

```
        //常数。允许设置 logging stream
        final static SQLPermission SET_LOG_PERMISSION = new SQLPermission("setLog");
        private static int loginTimeout = 0;
        private static java.io.PrintWriter logWriter = null;                    //写入的字符流
        private static java.io.PrintStream logStream = null;                    //写入的字节流

        //设置驱动在试图连接（log）时等待的最大时间
        public static void setLoginTimeout(int seconds) {         loginTimeout = seconds;       }
```

```
    public static int getLoginTimeout() {              return (loginTimeout);      }

    public static java.io.PrintWriter getLogWriter() {          //得到 LogWriter
        return logWriter;
    }

    public static void setLogWriter(java.io.PrintWriter out) {      //设置字符流
    SecurityManager sec = System.getSecurityManager();              //取得安全管理器
    if (sec != null) {      sec.checkPermission(SET_LOG_PERMISSION);  }
                            //检查是否具有日志写入的权限，有权限则继续，否则抛出异常！
        logStream = null;
        logWriter = out;
    }

    public static void setLogStream(java.io.PrintStream out) {            //设置字节流
        SecurityManager sec = System.getSecurityManager();
        if (sec != null) {        sec.checkPermission(SET_LOG_PERMISSION);          }
        logStream = out;
    if ( out != null )
        logWriter = new java.io.PrintWriter(out);                       //将字节流包装为字符流
    else
        logWriter = null;
    }

    public static java.io.PrintStream getLogStream() {      //得到字节流
        return logStream;
    }

}
```

以上对应于《教你建立简单 JDBC 程序的》DriverManager.getConnection()方法。

下篇我们将关注：数据库提供商如何注册自己的驱动，即关注 Class.forName()方法。以及“可插拔”等概念！