# 1. 数据库连接池技术

以我们现在所学技术来做项目的话，我们会在每个DAO方法中都执行一遍获取连接池对象和释放连接资源的操作。但在实际项目中，数据访问量可能非常大，则这样的做法会频繁地创建对象和释放资源。（因为Servlet就是多线程的，多用户并发访问Servlet就会产生多个线程访问下层的service和DAO层，因而多个线程访问数据库）。

但是，数据库连接资源是十分宝贵的，频繁创建和释放对象会耗费很大资源，降低了系统的效率。该问题就可通过数据库连接池来解决。可以在应用启动时，就预先创建好多个数据库连接，并把这些连接对象缓存起来，这个缓存的地方就是“数据库连接池”。当后续需要使用连接对象时，直接从池中取出一个对象提供给应用使用，当应用使用完毕后，再把对象“还回”到数据库连接池中。这样，实现了效率的提高。

## 1.1 模拟数据库连接池——理解原理

上面我们了解了连接池的基本原理，其实我们完全可据此自己编写代码模拟一个简陋的连接池。编码很简单，所谓的连接池就是内存中用于存放连接对象的缓存，使用数组或集合存储起来即可。注意的一点是，当提供给程序使用完后，不能调用close()方法关闭掉，而应该提供一个方法回收对象，这样实现重复利用。

因此一个简陋的数据库连接池代码如下：

|  |
| --- |
| **package** com.company;  **import** java.sql.Connection; **import** java.util.Collections; **import** java.util.LinkedList; **import** java.util.List;  *// 一个简单的数据库连接池* **public class** SimpleConnPool {  **private** SimpleConnPool() {}  *// 维护一个缓存，就是连接池，用于存储连接对象* **private static** List<Connection> *pool* = Collections.*synchronizedList*(**new** LinkedList<>()); *// 在多线程环境中，为了线程安全，所以这样获得线程安全的集合。* **static** {  *// 创建10个连接对象放入缓存中* **for** (**int** i = 0; i < 10; i++) {  **try** {  *pool*.add(JdbcUtil.*getConnection*());  } **catch** (Exception e) {  e.printStackTrace();  }  }  }   *// 提供一个获得数据库连接的方法* **public static** Connection getConnection() {  **if** (*pool*.size() > 0) {  **return** *pool*.remove(0); *// remove会移除并返回元素* } **else** {  **throw new** RuntimeException(**"No enough connections!"**);  }  }   *// 在提供一个“还回”数据库连接的方法* **public static void** returnConnection(Connection conn) {  **if** (conn != **null**) {  *pool*.add(conn); *// 添加到连接池中* }  } } |

如何使用这个“连接池”呢？简单例子：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try** {  conn = SimpleConnPool.*getConnection*();  stmt = conn.prepareStatement(**"UPDATE account SET money = money - 100 WHERE name = 'A'"**);  stmt.executeUpdate();  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  JdbcUtil.*release*(**null**, stmt); *// 关闭其他资源* SimpleConnPool.*returnConnection*(conn); *// 还回连接* } } |

实际上，JDK中提供了“javax.sql.DataSource”类以规范化数据库连接池的开发，否则各种使用规则不一的数据库连接池将带来复杂和麻烦。这个类就叫“数据源”，以后都将使用该规范来使用数据库连接池。为了编写符合Java规范的数据库连接池，我们的连接池类需要实现javax.sql.DataSource接口。实现该接口需要实现很多方法，由于我们是模拟了解其原理，所以现在我们不管其他的，只实现其中的getConnection方法即可，和之前的模拟代码大同小异，代码如下：

|  |
| --- |
| **package** com.company;  **import** javax.sql.DataSource; **import** java.io.PrintWriter; **import** java.sql.Connection; **import** java.sql.SQLException; **import** java.sql.SQLFeatureNotSupportedException; **import** java.util.Collections; **import** java.util.LinkedList; **import** java.util.List; **import** java.util.logging.Logger;  **public class** SimpleStandardPool **implements** DataSource {  **private static** List<Connection> *pool* = Collections.*synchronizedList*(**new** LinkedList<>());  **static** {  *// 创建10个连接对象放入缓存中* **for** (**int** i = 0; i < 10; i++) {  **try** {  *pool*.add(JdbcUtil.*getConnection*());  } **catch** (Exception e) {  e.printStackTrace();  }  }  }  @Override  **public** Connection getConnection() **throws** SQLException {  **if** (*pool*.size() > 0) {  **return** *pool*.remove(0); *// remove会移除并返回元素* } **else** {  **throw new** RuntimeException(**"No enough connections!"**);  }  }  // 其他方法省略...... } |

问题来了，DataSource接口中并没有提供“还回”连接的方法，那么连接如何还回呢？显然，我们自行在SimpleStandardPool实现类中添加一个returnConnection方法是没用的，因为按照规范，程序只能使用DataSource接口，而接口是无法调用到实现类自定义的方法的。

怎么办呢？我们以往使用连接对象完毕后，都会调用Connection接口的close()方法关闭连接，用户都会自觉这么做，但现在使用连接池是不能调用close()方法关闭连接对象的。如果说，用户调用close()方法后，并不是关闭连接对象，而是执行我们的将连接对象放回连接池的操作语句，那不就解决了！而且使用完毕调用close()方法这也符合用户使用惯例。

那么如何实现呢！如何让close()方法转而执行我们的代码？这就是本章的难点了。先明确我们的目的：使调用close()方法，不是执行关闭连接，而是执行我们的语句，达到将连接放回池中的效果。

这其实不就是想要改变对象的执行方法吗？而我们在Java SE中学习的代理正是解决这个问题的！因此这里的案例是应用代理的很好地一个案例，如果有代理陌生了，可回过头复习一下Java SE中的代理知识。

在代理中我们讲到可以使用静态代理和动态代理，其中每种代理又分为基于接口的代理和基于子类的代理。在这里，由于Connection是一个接口，因此适合使用基于接口的代理。

我们先使用静态代理进行实现，则代理类代码如下：

|  |
| --- |
| **package** com.company;  *// 1. 实现与委托类（这里的委托类实际上就是MySQL/Oracle/SqlServer等JDBC中Connection的实现类）相同的接口。 // 这个接口就是Connection。* **public class** MyConnection **implements** Connection {  **private** Connection **connection**; *// 2. 定义一个实例变量，引用被装饰对象。这里直接使用接口类型，通用。* **private** List<Connection> **pool**; *// 这个是维护的连接池对象。因为为了安全性我们之前把pool定义为private的，这里必须通过下面的构造函数传递进来并维护   // 3. 使用构造函数将被装饰对象注入进来。  // 这里也把连接池传递过来了，为了维护，上面说过了。* **public** MyConnection(Connection connection, List<Connection> pool) {  **this**.**connection** = connection;  **this**.**pool** = pool;  }   *// 5. 需要改写的方法，改写即可。* @Override  **public void** close() **throws** SQLException {  **pool**.add(**connection**); *// 这里就将连接对象还回池中* }   *// 4. 下面就是众多的无需改写的方法，直接返回connection的方法即可。还有很多方法，省略了。* @Override  **public** Statement createStatement() **throws** SQLException {  **return connection**.createStatement();  }  // 省略...... } |

这时，我们的SimpleStandardPool连接池类的getConnection方法中，也要将连接对象包装成MyConnection对象返回，这样以后用户调用连接的close()方法，才会真正地调用我们的方法。更改的代码如下：

|  |
| --- |
| **public** Connection getConnection() **throws** SQLException {  **if** (*pool*.size() > 0) {  Connection connection = *pool*.remove(0);  **return new** MyConnection(connection, *pool*);  } **else** {  **throw new** RuntimeException(**"No enough connections!"**);  } } |

虽然返回的还是Connection，但它的本质已经变了，调用close()方法时就会执行归还连接操作。

使用这个连接池时，可查看它真正的类型：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  DataSource ds = **null**;  **try** {  ds = **new** SimpleStandardPool();  conn = ds.getConnection();  System.***out***.println(conn.getClass().getName()); *// 输出的结果是com.company.MyConnection,这才是真正的类型* } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  JdbcUtil.*release*(conn, stmt); *// 这时可正常关闭。但实际上其中的conn.close()调用的方法是还回连接* } } |

写完上面的静态代理就会发现这样的缺点：MyConnection类需要实现的方法太多了，哪些不需要修改的方法也要重新实现一遍，比较繁杂。

因此我们这里再使用接口的动态代理解决问题。既然用动态代理，现在我们直接在SimpleStandardPool中写动态代理代码即可，无需创建其他类。这时getConnection返回的是一个代理类对象，在代理中，只“改写”close()方法。则代码写为：

|  |
| --- |
| **package** com.company;  **import** javax.sql.DataSource; // 省略导包...  **public class** SimpleStandardPool **implements** DataSource {  **private static** List<Connection> *pool* = Collections.*synchronizedList*(**new** LinkedList<>());  **static** {  *// 创建10个连接对象放入缓存中* **for** (**int** i = 0; i < 10; i++) {  **try** {  *pool*.add(JdbcUtil.*getConnection*());  } **catch** (Exception e) {  e.printStackTrace();  }  }  }  @Override  **public** Connection getConnection() **throws** SQLException {  **if** (*pool*.size() > 0) {  Connection connection = *pool*.remove(0);  *// 下面使用动态代理，不直接返回connection连接对象，而是返回代理对象。* **return** (Connection) Proxy.*newProxyInstance*(connection.getClass().getClassLoader(),  connection.getClass().getInterfaces(), **new** InvocationHandler() {  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  *// 这其中就针对close()方法，将连接还回。* **if** (**"close"**.equals(method.getName())) {  *pool*.add(connection);  } **else** {  *// 如果是别的方法，还是使用connection原来的方法。* **return** method.invoke(connection, args);  }  **return null**;  }  });  } **else** {  **throw new** RuntimeException(**"No enough connections!"**);  }  }   @Override  **public** Connection getConnection(String username, String password) **throws** SQLException {  **return null**;  }  // 其他的方法省略... } |

通过上述的代码也较好地实现了我们想要的功能。通过上面的案例，应该能较好地理解代理了，代理的目的就是希望修改和扩展原有的功能而不希望改变原来类的代码。

这里就不使用继承MySQL的Connection实现类来实现代理了，一是因为继承Connection实现类比较复杂，二是因为通用性不好，如果将来使用Oracle或者SqlServer，则分别还要写代理类继承Oracle或者SqlServer的Connection实现类，这也体现了使用接口的好处，实现通用性。

## 1.2 使用数据库连接池

自己实现功能全面的、可靠的数据库连接池是比较麻烦的，因此开发中一般使用第三方连接池，例如下面讲的DBCP等。

第三方做的好的数据库连接池提供的功能完整，能很好地管理数据库连接池，例如可灵活设置初始连接数目、最大连接数等等（比如当前连接数不够时，再新创建几个连接池对象，当连接池数目达到配置的最大值时，必须等待其他连接池使用完毕才进行分配），因此使用连接池的作用是很明显的，不仅能高效地使用资源，还能有效地配置、管理和监控资源的使用。

## 1.3 DBCP连接池

DBCP是Apache的开源连接池实现，要使用DBCP，除了要在项目中引入DBCP的jar包外，还要引入DBCP的两个依赖包：commons-pool.jar和commons-logging.jar，这都是Apache的产品。这些文件可自行访问Apache了解并下载。为了锻炼大家寻找资源的能力，这里不提供详细信息了。

项目中，注意除了要引入上面的三个jar包，还不能忘了JDBC驱动包，这肯定是必须要的，否则无法访问数据库了。

使用DBCP时，一般使用properties配置文件来配置数据库连接等参数，例如我们在src下新建一个db.properties文件，内容和解释如下：

|  |
| --- |
| *# 配置JDBC驱动类* **driverClassName**=**com.mysql.jdbc.Driver** *# 连接的URL* **url**=**jdbc:mysql://114.55.86.230:3306/mytest** *# 用户名* **username**=**root** *# 密码* **password**=**123qwe!@#** *# 初始化连接数，即连接池初始化时就创建的连接数* **initialSize**=**5** *# 最大空闲连接* **maxIdle**=**8** *# 最小空闲连接* **minIdle**=**2** *# 同一时间可分配的最大连接数* **maxTotal**=**10** *# 最大等待时间，以毫秒为单位 # 这个意思是一次连接被使用的最大时间。 # 例如下面配置为600000，表示如果一个连接在60秒内还没被还回池中，就抛出异常。 # 如果该值设置为-1就是无限等待，DBCP的默认配置就是无限等待* **maxWaitMillis**=**600000** *# url的连接属性。书写的格式为“属性名=值” # 由于user和password两个属性在前面指定了，这里不需要包含 # 当然这些属性也能在url中指定，效果是一样的* **connectionProperties**=**useUnicode=true;characterEncoding=utf8;useSSL=false** *# 指定连接的事务是否是自动提交的* **defaultAutoCommit**=**true** *# 指定连接的事务隔离级别 # 可用的值：NONE，READ\_UNCOMMITTED，READ\_UNCOMMITTED，REPEATABLE\_READ，SERIALIZABLE* **defaultTransactionIsolation**=**READ\_UNCOMMITTED** |

上面是常用的DBCP配置项，更多内容可参见官方文档：http://commons.apache.org/proper/commons-dbcp/configuration.html。

下面我们写一个工具类来使用DBCP，这样可以从该工具类DBCPUtil中拿到数据库连接并使用。代码中，使用DBCP的BasicDataSourceFactory类得到DataSource数据源，进而获取数据库连接：

|  |
| --- |
| **package** com.company;  **import** org.apache.commons.dbcp2.BasicDataSourceFactory;  **import** javax.sql.DataSource; **import** java.io.InputStream; **import** java.sql.Connection; **import** java.sql.ResultSet; **import** java.sql.SQLException; **import** java.sql.Statement; **import** java.util.Properties;  **public class** DBCPUtil {  **private static** DataSource *dataSource*;  **static** {  **try** {  *// 读取配置文件，初始化数据源* Properties props = **new** Properties();  InputStream in = DBCPUtil.**class**.getClassLoader().getResourceAsStream(**"db.properties"**);  props.load(in);  *// BasicDataSourceFactory.createDataSource返回的是BasicDataSource类型对象，而BasicDataSource实现了DataSource接口，因此可赋值给dataSource  dataSource* = BasicDataSourceFactory.*createDataSource*(props);  } **catch** (Exception e) {  **throw new** ExceptionInInitializerError(e);  }   }   *// 提供获得数据源对象的方法* **public static** DataSource getDataSource() {  **return** *dataSource*;  }   *// 提供获得数据库连接的方法* **public static** Connection getConnection() {  **try** {  **return** *dataSource*.getConnection();  } **catch** (SQLException e) {  **throw new** RuntimeException(**"获取数据库连接失败！"**);  }  }   *// 下面提供资源的释放方法，和以前的JDBCUtil中释放代码一样，但以后就不用JDBCUtil了，直接使用该类即可。  // 需要知道的是，释放资源中，虽然代码和以前基本一致，但是调用connection.close()方法实际上是把数据库连接返回给DBCP数据库连接池，这个就是前面讲的原理，不再多说  // 唯一需要注意的是，最后的connection不再赋值为null，因为连接对象不是被关闭了。   /\*\*  \* 释放资源  \** ***@param conn*** *连接资源  \** ***@param stmt*** *statement资源  \** ***@param rs*** *结果集资源  \*/* **public static void** release(Connection conn, Statement stmt, ResultSet rs) {  **if** (rs != **null**) {  **try** {  rs.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  rs = **null**;  }  **if** (stmt != **null**) {  **try** {  stmt.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  stmt = **null**;  }  **if** (conn != **null**) {  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  *// 注意这里不再执行conn = null; 这条语句* }  }   */\*\*  \* 释放资源  \** ***@param conn*** *连接资源  \** ***@param stmt*** *statement资源  \*/* **public static void** release(Connection conn, Statement stmt) {  *// 这里需要调用上面的方法，最后一个参数给null即可。  release*(conn, stmt, **null**);  } } |

使用一下：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  **try** {  conn = DBCPUtil.*getConnection*();  System.***out***.println(conn.getClass().getName()); *// 输出连接对象的实际类型名称* } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  DBCPUtil.*release*(conn, **null**);  } } |

发现输出的connection类型是“org.apache.commons.dbcp2.PoolingDataSource$PoolGuardConnectionWrapper”，说明连接池返回的是包装后（或者说是代理后）的连接对象，这个原理和我们之前是一样的，基本上说明了调用close()方法是将连接资源还回池中。如果有兴趣，可看DBCP源码。

如果不了解原理的话，使用者可能有疑惑，到底关不关闭连接，关连接不符合连接池的实现目的，不关连接又疑惑连接是怎么还回去的。之前我就有这样的疑惑，现在了解了实现原理，就茅塞顿开了。

为了验证数据库连接池有效果，我们可以在maing函数中获得连接后打一个断点使程序“暂停”，这时在MySQL中执行“show processlist”查看当前所有的数据库连接，可以看到连接池管理的所有连接的状态。这证明了连接池有效。

像DBCP一样，连接池支持使用配置文件进行数据库连接的配置，这样的好处是使用灵活，如果项目发布时需要更改数据源配置，只要修改配置文件即可，无需重新修改、编译代码和打包项目。以后会常使用配置文件提高系统的灵活性，这些功能自己实现也是很简单的，只是读取文件信息再代码中进行设置即可。

除了DBCP之外，还有其他数据库连接池，例如Druid、HikariCP、C3P0（C3P0现在不推荐使用）等。这些连接池的使用大同小异，只要按文档进行配置、拿到DataSource数据源即可使用，读者可自行学习。总之，以后的项目中都会使用连接池技术，这时必不可少的。

## 1.4 利用WEB服务器管理数据源（JNDI）

主流的WEB服务器内置了数据源的实现，例如在“Tomcat/lib”目录中，就有“tomcat-dbcp.jar”包，即内置了DBCP的包，可直接供所有应用使用。这样，只要在WEB应用中配置一下，就能拿到DBCP管理的数据源。

JNDI，即“Java Naming and Directory Interface”，意思是“Java命名和目录接口”。JNDI可以理解为一个容器，其中存储键值对，键就是“目录+名称”，值就是存储的对象。即我们根据“目录+名称”可以在JNDI容器中获得对应的对象。这类似于Windows的注册表，注册表项就是根据“路径+名称”找到一个特定的注册表值的，因为前面有路径，因此名称在不同路径是可重复的。

下面我们直接讲解如何为Tomcat中的WEB应用配置JDNI数据源。

配置步骤：

（1）将JDBC驱动包复制一份到“Tomcat/lib”目录下。这样Tomcat和其中所有应用都能使用该包。

（2）为WEB应用配置JNDI数据源。在应用的META\_INF目录下，建立一个context.xml配置文件，其内容和解释如下：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**Context**>  <**Resource name="jdbc/Test" auth="Container" type="javax.sql.DataSource" driverClassName="com.mysql.jdbc.Driver" username="root" password="123qwe!@#" url="jdbc:mysql://114.55.86.230:3306/mytest" connectionProperties="useSSL=false"** />  *<!--  上述的Resource属性中：  name表示放到JNDI容器中的名称（不是目录）  auth="Container"表示对象是由容器来创建的  type="javax.sql.DataSource"表示创建DataSource类型的对象  其余的属性，就是DBCP中的连接属性了，和之前讲的是一致的。  -->* </**Context**> |

如果这时在Tomcat中启动应用，那么DBCP数据源就已经存在于JNDI的内存中了。

（3）最后，在应用中，通过JNDI的API获得数据源。核心代码如下：

|  |
| --- |
| Context initContext = **new** InitialContext(); Context envContext = (Context) initContext.lookup(**"java:/comp/env"**); *// 这是找到JNDI的路径对应的Context* DataSource ds = (DataSource) envContext.lookup(**"jdbc/Test"**); *// 这是找到名称对应的Context，即数据源* |

上述代码中，“路径 + 名字”也可以一起写：

|  |
| --- |
| DataSource ds = (DataSource) initContext.lookup(**"java:/comp/env/jdbc/Test"**); |

# 2. JDBC框架

## 2.1 获取元数据Metadata

数据库或表的定义信息，或者结果集的一些信息。

有时想获得数据库连接对象的一些信息，例如使用的数据库名称、版本、结果集的列信息等。而元数据就包含了数据库相关对象的定义信。

JDBC提供了三种元数据对象，分别是数据库元数据、参数元数据和结果集元数据。见案例。

案例1：数据库元数据。通过Connection对象的getMetaData方法即可得到DatabaseMetaData，可从中获取数据库相关信息。

代码：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  **try** {  conn = DBCPUtil.*getConnection*();  *// 得到数据库元对象* DatabaseMetaData metaData = conn.getMetaData();  System.***out***.println(metaData.getUserName()); *//获得用户名* System.***out***.println(metaData.getURL()); *//获得连接字符串url* System.***out***.println(metaData.getDatabaseProductName()); *// 数据库名称* System.***out***.println(metaData.getDatabaseProductVersion()); *// 数据库版本号* System.***out***.println(metaData.getDriverName()); *// 驱动名称* System.***out***.println(metaData.getDriverVersion()); *// 驱动版本* System.***out***.println(metaData.isReadOnly()); *// 是否是只读的* } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  DBCPUtil.*release*(conn, **null**);  } } |

案例2：参数元数据，可调用PreparedStatement对象的getParameterMetaData()方法得到元数据对象。可获得参数个数。

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement ps = **null**;  **try** {  conn = DBCPUtil.*getConnection*();  ps = conn.prepareStatement(**"insert into account values(?,?)"**);  *// 得到参数元数据对象* ParameterMetaData pmd = ps.getParameterMetaData();  System.***out***.println(pmd.getParameterCount()); *// 获得参数个数。输出是2.* } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  DBCPUtil.*release*(conn, ps);  } } |

案例3：结果集元数据。通过ResultSet的getMetaData()方法得到，可获得当前列的个数和名称等信息。

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement ps = **null**;  ResultSet rs = **null**;  **try** {  conn = DBCPUtil.*getConnection*();  ps = conn.prepareStatement(**"select** *\** **from account"**);  rs = ps.executeQuery();  ResultSetMetaData rsmd = rs.getMetaData();  System.***out***.println(rsmd.getColumnCount()); *// getColumnCount()获得结果集的列数  // 下面遍历列，获得列名、数据类型等信息。由于索引从1开始，因此遍历就从1开始* **for**(**int** i = 1; i <= rsmd.getColumnCount(); i++) {  String columnLabel = rsmd.getColumnLabel(i); *// 获得列名* String columnTypeName = rsmd.getColumnTypeName(i); *// 获得列的数据类型名称* String columnClassName = rsmd.getColumnClassName(i); *// 获得该列数据类型对应的Java类型* System.***out***.println(columnLabel + **" | "** + columnTypeName + **" | "** + columnClassName);  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  DBCPUtil.*release*(conn, ps, rs);  } } |

这里提一下ResultSetMetaData提供的getColumnType(int column)方法，该方法也是获得指定列的数据类型的，但返回值是int类型，这是因为java.sql.Types类中定义了数据类型的常量字段值，例如VARCHAR类型对应的数值是12，这样就能找到对应关系。

当然，在开发中一般用不到元数据，只有在编写JDBC框架时才会用到这些信息，下面我们就编写自己的一个JDBC框架，简化开发，其中就需要用到元数据。

## 2.2 编写自己的JDBC框架

编写此JDBC框架的目的是为了简化代码开发。因为我们发现，实际上前面的数据库操作，可以总结为两类，一类是执行DML增删改操作，执行executeUpdate方法；一类是执行DQL操作，执行executeQuery方法。它们分别是数据更新和查询操作，其中的语句都是类似的，不同的只是SQL语句和预编译参数的不同，对于查询操作来说，还有一个结果集需要处理，除此之外，基本操作都是一样的，例如获得连接、执行SQL和释放资源等。

因此，为了高效开发，我们可以把更新和查询操作分别抽取到一个DBAssist类中，该类提供update()和query()方法，其中update()可执行增删改操作，query()方法可执行查询操作。

封装DBAssist时，为了能拿到数据库连接，我们可以通过构造函数给该类注入一个DataSource数据源，以此拿到连接对象。开发者使用时需要先传递一个数据源对象。

写update()方法时，只要传递SQL和预编译的可变参数即可，方法中，可通过参数元数据判断传递的参数个数是否正确。因此目前DBAssist代码如下（包含了释放资源的代码，和之前的一样）：

|  |
| --- |
| **package** com.company;  **import** javax.sql.DataSource; **import** java.sql.\*;  **public class** DBAssist {   **private** DataSource **dataSource**;   **public** DBAssist(DataSource dataSource) {  **this**.**dataSource** = dataSource;  }   */\*\*  \* 数据更新操作  \** ***@param sql*** *要执行的SQL  \** ***@param params*** *传递的预编译参数。要按照顺序传递。由于不知道是什么类型，因此用Object可变参数数组。  \** ***@return*** *更新的行数  \*/* **public int** update(String sql, Object... params) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try** {  conn = **dataSource**.getConnection();  stmt = conn.prepareStatement(sql);  *// 通过参数元数据判断参数是否正确* ParameterMetaData pmd = stmt.getParameterMetaData();  **int** num = pmd.getParameterCount();  **if** (num > 0) {  **if** (params == **null**) {  **throw new** RuntimeException(**"Parameters Required!"**); *// 需要占位符* }  **if** (params.**length** != num) {  **throw new** RuntimeException(**"Parameters don't match!"**); *// 参数个数不匹配* }  *// 排除了异常情况，这时可以把预编译参数设置到SQL中。* **for** (**int** i = 0; i < num; i++) {  stmt.setObject(i + 1, params[i]);  }  }  *// 最后执行返回结果* **return** stmt.executeUpdate();  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  } **finally** {  release(conn, stmt);  }  }   */\*\*  \* 释放资源  \** ***@param conn*** *连接资源  \** ***@param stmt*** *statement资源  \** ***@param rs*** *结果集资源  \*/* **private void** release(Connection conn, Statement stmt, ResultSet rs) {  **if** (rs != **null**) {  **try** {  rs.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  rs = **null**;  }  **if** (stmt != **null**) {  **try** {  stmt.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  stmt = **null**;  }  **if** (conn != **null**) {  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  *// 注意这里不再执行conn = null; 这条语句* }  }   */\*\*  \* 释放资源  \** ***@param conn*** *连接资源  \** ***@param stmt*** *statement资源  \*/* **private void** release(Connection conn, Statement stmt) {  *// 这里需要调用上面的方法，最后一个参数给null即可。* release(conn, stmt, **null**);  } } |

抽取query()方法时，也需要传递SQL和预编译参数。query与update不同的是query需要处理结果集，并且应该返回处理完毕的数据对象以便调用者使用。query()方法中拿到了结果集如何处理，并封装成需要的对象返回呢？很遗憾，框架中query方法是无法知道的，只有调用者知道如何处理。那如何解决这个问题呢？我们参考之前的动态代理用的策略设计模式就知道如何解决了。

这里我们就使用策略模式，策略模式提供一个接口，然后让调用者去实现这个接口。在query中，能拿到结果集但不知如何处理，因此使用策略模式提供一个ResultSetHandler接口，接口中有一个handle方法，方法的参数是ResultSet结果集，返回值是Object。具体结果集是怎么处理的，由用户自行实现接口。这时，query方法还需要传递一个ResultSetHandler对象，然后query中调用这个对象的handle方法，向handle方法传递结果集对象，并将handle方法的返回值返回（正是由于不知道返回什么类型，所以接口方法返回值用Object）。

这样就顺理成章了，query能拿到结果集，然后调用handle方法传递参数，开发者按照接口实现自己的处理类，其中能拿到ResultSet处理并封装返回对象，而这个对象也正是query返回的。最终，调用者能得到正确的结果对象。

代码如下：

（1）ResultSetHandler接口，用于声明策略：

|  |
| --- |
| **package** com.company;  **import** java.sql.ResultSet;  **public interface** ResultSetHandler {  Object handle(ResultSet rs); } |

（2）query的代码：

|  |
| --- |
| */\*\*  \* 执行的查询语句，返回查询结果  \** ***@param sql*** *执行的SQL  \** ***@param rsh*** *ResultSetHandler对象，用于处理结果集  \** ***@param params*** *预编译可变参数。可变参数必须是方法的最后一个参数，因此这里放在最后  \** ***@return*** *返回封装的数据对象  \*/* **public** Object query(String sql, ResultSetHandler rsh, Object... params) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  ResultSet rs = **null**;  **try** {  conn = **dataSource**.getConnection();  stmt = conn.prepareStatement(sql);  *// 通过参数元数据判断参数是否正确。代码和上面一样* ParameterMetaData pmd = stmt.getParameterMetaData();  **int** num = pmd.getParameterCount();  **if** (num > 0) {  **if** (params == **null**) {  **throw new** RuntimeException(**"Parameters Required!"**); *// 需要占位符* }  **if** (params.**length** != num) {  **throw new** RuntimeException(**"Parameters don't match!"**); *// 参数个数不匹配* }  *// 排除了异常情况，这时可以把预编译参数设置到SQL中。* **for** (**int** i = 0; i < num; i++) {  stmt.setObject(i + 1, params[i]);  }  }  *// 执行查询* rs = stmt.executeQuery();  *// 调用ResultSetHandler的处理方法，并返回其返回值即可。就这个是重点。* **return** rsh.handle(rs);  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  } **finally** {  release(conn, stmt, rs);  } } |

这样我们就将DBAsist开发好了，现在用一下。代码如下：

|  |
| --- |
| **package** com.company;  **import** org.junit.Test;  **import** java.sql.ResultSet; **import** java.sql.SQLException; **import** java.util.ArrayList; **import** java.util.List;  *// DAO测试类* **public class** DAOTest {  *// DBAssist对象* **private** DBAssist **dbAssist** = **new** DBAssist(DBCPUtil.*getDataSource*());   *// 测试update* @Test  **public void** testUpdate() {  **int** rows = **dbAssist**.update(**"insert into account(name, money) values(?,?)"**, **"张三"**, 1000);  System.***out***.println(rows);  }   *// 测试query* @Test  **public void** testQuery() {  *// 下面使用了匿名内部类处理Account。Account实体类自行编写。* List<Account> accounts = (List<Account>)**dbAssist**.query(**"select name, money from account"**, **new** ResultSetHandler() {  @Override  **public** Object handle(ResultSet rs) {  List<Account> accountList = **new** ArrayList<>();  **try** {  **while** (rs.next()) {  Account account = **new** Account();  account.setName(rs.getString(**"name"**));  account.setMoney(rs.getInt(**"money"**));  accountList.add(account);  }  **return** accountList;  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }  });  System.***out***.println(accounts);  } } |

现在我们发现代码简洁很多，使我们只关注业务本身。为了使用者更加方便，我们最好能提供一些常用ResultSetHandler实现类。例如我们写一个BeanHandler实现类，该实现类能将结果集封装成指定的类型返回，但只适用于结果集只有一条的情况，如果结果集有多条，则只返回第一条结果。前提是指定类的字段名和数据库的列名一致才可封装（否则比较麻烦，需要一一指定实体类字段和数据表字段的对应关系。以后的很多框架也会默认要求实体类字段和数据库表字段一致才会正确封装）。则代码如下（需要利用反射）：

|  |
| --- |
| **package** com.company;  **import** java.lang.reflect.Field; **import** java.sql.ResultSet;  **public class** BeanHandler **implements** ResultSetHandler {   **private** Class **clazz**;   *// 怎么知道使用者的目标类型呢？用构造函数传递过来字节码对象即可。* **public** BeanHandler(Class clazz) {  **this**.**clazz** = clazz;  }   @Override  **public** Object handle(ResultSet rs) {  *// 待返回的数据对象* Object obj = **null**;  **try** {  **if** ( rs.next() ) {  *// 创建对象* obj = **clazz**.newInstance();  *// 循环获得对象的属性。并设置对象的属性值* Field[] fields = **clazz**.getDeclaredFields();  **for** (Field field : fields) {  field.setAccessible(**true**); *// 使私有字段可访问* field.set(obj, rs.getObject(field.getName())); *// 设置对应的值* }  }  **return** obj;  } **catch** (Exception e) {  **throw new** RuntimeException(e);  }  } } |

这样，要获得封装好的一列查询数据只需要这样：

|  |
| --- |
| @Test **public void** testQuery() {  Account accounts = (Account) **dbAssist**.query(**"select name, money from account where name=?"**, **new** BeanHandler(Account.**class**), **"张三"**);  System.***out***.println(accounts); } |

同样，我们还可提供一个BeanListHandler处理类，能处理结果集的多条数据并封装为指定泛型数据类型的集合。BeanListHandler则写为（大部分代码同BeanHandler）：

|  |
| --- |
| **package** com.company;  **public class** BeanListHandler **implements** ResultSetHandler {   **private** Class **clazz**;  *// 传递目标类型* **public** BeanListHandler(Class clazz) {  **this**.**clazz** = clazz;  }   @Override  **public** Object handle(ResultSet rs) {  *// 待返回的数据对象。因为可能有多条数据，因此是一个集合。* List<Object> list = **new** ArrayList<>();  Object obj = **null**; *// 待加入的对象* **try** {  **while** ( rs.next() ) {  *// 创建对象* obj = **clazz**.newInstance();  *// 循环获得对象的属性。并设置对象的属性值* Field[] fields = **clazz**.getDeclaredFields();  **for** (Field field : fields) {  field.setAccessible(**true**); *// 使私有字段可访问* field.set(obj, rs.getObject(field.getName())); *// 设置对应的值* }  list.add(obj); *// 将对象加入集合中* }  **return** list; *// 返回集合* } **catch** (Exception e) {  **throw new** RuntimeException(e);  }  } } |

这样，查询多条结果集也很简单：

|  |
| --- |
| @Test **public void** testQuery() {List<Account> accounts = (List<Account>) **dbAssist**.query(**"select name, money from account"**, **new** BeanListHandler(Account.**class**));  System.***out***.println(accounts); } |

开发完DBAssist后，完全可以把相关的类一起导出成为jar包，以便别的项目使用，目前DBAssist并没有依赖别的项目代码。使用该框架最直接的感受应该是：框架让我们更专注于业务实现，代码简洁有效，而无需过多考虑下层的实现细节。

其实自己做框架首先要转变的是自己的思想，要认清自己现在的身份。例如编写框架时，那你的身份就是一名框架设计者，这时要考虑的是开发者的需求，开发者如果使用你的框架，你的框架应该提供什么样的功能。而这时不能再以自己作为码农搬砖时的开发思维来开发框架，否则你设计的东西是无法提供给人很好地使用的。我见过同学写的DAO层代码，他根本不考虑作为其他Service层开发者来讲需要的是什么，直接按照自己的思维想当然地写完DAO层，不出所料他的DAO是无法供给其他开发者使用的，设计的也不好，没有这种作为架构师的思维。如果读者想避免出现这种情况（在我理解就是所谓的“搬砖思维”），就需要理解面向对象、理解需求、考虑设计和实现较好封装性的代码。

另外，每个框架中都有一些自己的约定，例如配置文件的位置、实体类和数据库字段的对应关系等，按照这样的约定才能较好的使用框架。所谓“约定优于编码”，代码不可能把所有奇葩的情况都考虑到来提升系统的灵活性，遵守约定也能提高开发效率。

## 2.3 DbUtils组件

commons-dbutils 是 Apache 组织的一个开源类库，它对JDBC进行了简单的封装。开发者使用DbUtils能提高开发效率，简化代码。

Apache的DbUtils项目地址：<http://commons.apache.org/proper/commons-dbutils/> 。可自行下载工具包，然后在Java项目中引入“commons-dbutils-1.6.jar”这个JAR包。当然，不能忘了引入数据库驱动包。

其实，我们写的DBAssist框架就是仿照DbUtils写的，因此现在再看DbUtils将会非常简单。

使用DbUtils的一般步骤：

（1）创建QueryRunner对象，可使用无参构造，或者使用带参构造QueryRunner(DataSource ds)注入一个数据源来创建对象；

（2）调用QueryRunner对象的update()和query()方法进行数据库操作。其实QueryRunner对象相当于我们之前的DBAssist对象，和之前的使用基本一致。

如果查看QueryRunner对象中提供的方法，会发现它提供了很多重载方法，总结下来实际上QueryRunner就提供了三个方法：

batch(String sql, Object[][] params)；执行批处理；

query(String sql, ResultSetHandler<T> rsh, Object... params)；执行查询；

update(String sql, Object... params)：执行更新语句。

它们均提供了一个带connection参数的重载，这个重载的重要作用会在后面的事务中会重点讲。一般我们注入了DataSource后，只需要使用上述的方法即可，至于query和update方法，和我们之前使用是类似的，演示如下：

|  |
| --- |
| **package** com.company;  **import** org.apache.commons.dbutils.QueryRunner; **import** org.apache.commons.dbutils.ResultSetHandler; **import** org.junit.Test;  **import** java.sql.ResultSet; **import** java.sql.SQLException; **import** java.util.ArrayList; **import** java.util.List;  *// DAO测试类* **public class** DAOTest {  *// QueryRunner对象* **private** QueryRunner **queryRunner** = **new** QueryRunner(DBCPUtil.*getDataSource*());   *// 测试update* @Test  **public void** testUpdate() {  **int** rows = 0;  **try** {  rows = **queryRunner**.update(**"insert into account(name, money) values(?,?)"**, **"张三"**, 1000);  } **catch** (SQLException e) {  e.printStackTrace();  }  System.***out***.println(rows);  }   *// 测试query* @Test  **public void** testQuery() {  List<Account> accounts = **null**;  **try** {  accounts = **queryRunner**.query(**"select name, money from account"**, **new** ResultSetHandler<List<Account>>() {  @Override  **public** List<Account> handle(ResultSet rs) **throws** SQLException {  List<Account> accountList = **new** ArrayList<>();  **while** (rs.next()) {  Account account = **new** Account();  account.setName(rs.getString(**"name"**));  account.setMoney(rs.getInt(**"money"**));  accountList.add(account);  }  **return** accountList;  }  });  } **catch** (SQLException e) {  e.printStackTrace();  }  System.***out***.println(accounts);  }   *// 测试batch* @Test  **public void** testBatch() {  QueryRunner qr = **new** QueryRunner();  *// batch批处理也适用于相似的增删改操作。* **try** {  *// batch的第三个参数是一个二维数组，二维数组中有几个一维数组，就表示执行几次SQL，而一维数组中的数据就是设置SQL参数值的。* **queryRunner**.batch( **"insert into account values(?,?)"**, **new** Object[][]{  {**"张三"**, 1000},  {**"李四"**, 2000},  {**"王五"**, 3000}  });  } **catch** (SQLException e) {  e.printStackTrace();  }  } } |

上述再需要注意一点是DbUtils中query和ResultSetHandler使用了泛型，而不是我们直接使用的Object类型。其中ResultSetHandler接口代码是这样的：

|  |
| --- |
| **public interface** ResultSetHandler<T> {T handle(ResultSet rs) **throws** SQLException; } |

所以query中也返回了泛型。这样我们写自己的ResultSetHandler实现时，就像上面的代码一样，接口实现类可以实现指定的泛型类型，该泛型类型就是我们想要的类型。当然，DbUtils中也提供了BeanHandler和BeanListHandler实现类，效果和之前是相同的，例如：

|  |
| --- |
| @Test **public void** testQuery() {  List<Account> accounts = **null**;  **try** {  accounts = **queryRunner**.query(**"select name, money from account"**, **new** BeanListHandler<Account>(Account.**class**));  } **catch** (SQLException e) {  e.printStackTrace();  }  System.***out***.println(accounts); } |

除此之外，DbUtils还提供了其他一些ResultSetHandler实现类，不过不是很常用，读者可自行了解，但多数情况下，大不了还是自己实现ResultSetHandler进行处理，按照自己的需求自行封装，没什么难的。

上述使用DbUtils实现了简单的对象关系映射（对象关系映射，即ORM，Object Relational Mapping），把数据库中的数据封装到对象中，实现用面向对象的方法操作数据。以后还会学习更强大的ORM框架，比如Hibernate和Mybatis。

# 3. 专题——使用DbUtils实现事务控制

## 3.1 ThreadLocal相关知识

Java文档中说“ThreadLocal提供了线程局部变量”。什么意思呢？那么ThreadLocal到底是什么，有什么作用呢？通过下面对ThreadLocal代码的模拟我们就能知道ThreadLocal是什么东西。

其实，ThreadLocal的代码可以理解为是这样设计的：

|  |
| --- |
| **public class** ThreadLocal {   **private** Map<Runnable,Object> **map** = **new** HashMap<Runnable,Object>();   **public void** set(Object obj){  **map**.put(Thread.*currentThread*(), obj);  }   **public void** remove(){  **map**.remove(Thread.*currentThread*());  }   **public** Object get(){  **return map**.get(Thread.*currentThread*());  } } |

上面的模拟代码很简单，其实意思就是为每个线程（Runnable）提供一个存储对象的地方，每个线程只能获得该线程之前设置过的对象，各个线程互不干扰，不会获得其他线程所有的对象数据。

真实的ThreadLocal虽然代码和上面模拟的不同，但我们使用其功能时可以这么理解。ThreadLocal同样提供了set()、remove()和get()方法，效果是类似的。所以说理解ThreadLocal是比较简单的，但是ThreadLocal这个类非常重要，能实现线程和数据的绑定。

ThreadLocal相当于一个容器，各线程只能拿到自己的局部变量，因此ThreadLocal“提供了线程局部变量”。

下面使用代码演示一下ThreadLocal的特性。

首先我们创建一个线程类，该类接收LocalThread对象，并在run()方法中设置值：

|  |
| --- |
| **package** com.company;  **public class** MyThread **extends** Thread {   **private** ThreadLocal **threadLocal**;   *// 接收传递过来的ThreadLocal对象* **public** MyThread(ThreadLocal threadLocal) {  **this**.**threadLocal** = threadLocal;  }   @Override  **public void** run() {  *// 使用ThreadLocal对象设置一个局部变量值* **threadLocal**.set(**"B"**);  } } |

下面是主类的演示：

|  |
| --- |
| **public class** ThreadLocalDemo {  **public static void** main(String[] args) {  *// 创建ThreadLocal对象* ThreadLocal threadLocal = **new** ThreadLocal();  *// 在当前主线程中设置一个“A”值。* threadLocal.set(**"A"**);   *// 启动新线程，为了使用同一个ThreadLocal，将ThreadLocal作为参数传进去。* Thread myThread = **new** MyThread(threadLocal);  myThread.start(); *// 开启新线程  // 在本线程中获得ThreadLocal的局部变量。* System.***out***.println(threadLocal.get());  } } |

我们发现，输出的值只会是A，而不是myThread线程设置的B。即主线程是无法获得myThread线程的局部变量的。

## 3.2 使用DbUtils实现事务控制

在前面简单的实例中都没有考虑如何进行事务控制，如果现在要对转账业务使用事务控制，利用DbUtils应该在三层架构中实现呢？

首先我们搭建“转账业务”的三层架构。按照我们之前的说法，先开发domain中的实体类，再开发Service接口，这样的开发流程最清晰，更能知道自己应该做什么。

（1）Account实体类（在com.company.domain中）：

|  |
| --- |
| **package** com.company.domain;  **import** java.io.Serializable;  **public class** Account **implements** Serializable {  **private** String **name**;  **private float money**;   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public float** getMoney() {  **return money**;  }   **public void** setMoney(**float** money) {  **this**.**money** = money;  }   @Override  **public** String toString() {  **return "Account{"** +  **"name='"** + **name** + **'\''** +  **", money="** + **money** +  **'}'**;  } } |

（2）编写业务接口：

|  |
| --- |
| **package** com.company.service;  **public interface** BusinessService {  */\*\*  \* 转账  \** ***@param srcAccountName*** *转出账户  \** ***@param targetAccountName*** *转入账户  \** ***@param money*** *金额  \*/* **void** transfer(String srcAccountName, String targetAccountName, **float** money); } |

（3）DAO接口

由于要实现业务，DAO层不能把这个转账业务分开，否则不能使用同一个连接对象，将无法实现事务。因此DAO层目前也只能这样写：

|  |
| --- |
| **package** com.company.dao;  **public interface** AccountDao {  *// 转账* **void** transfer(String srcAccountName, String targetAccountName, **float** money); } |

（4）DAO实现（com.company.dao.impl包下）。

如果按照我们之前的写法，则写为：

|  |
| --- |
| **package** com.company.dao.impl;  **import** com.company.dao.AccountDao; **import** com.company.util.DBCPUtil; **import** org.apache.commons.dbutils.QueryRunner;  **import** java.sql.SQLException;  **public class** AccountDaoImpl **implements** AccountDao {  **private** QueryRunner **queryRunner** = **new** QueryRunner(DBCPUtil.*getDataSource*());  @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  **try** {  **queryRunner**.update(**"update account set money=money-? where name=?"**, money, srcAccountName);  **queryRunner**.update(**"update account set money=money+? where name=?"**, money, targetAccountName);  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  } } |

（5）Service实现（com.company.service.impl包下）。

|  |
| --- |
| **package** com.company.service.impl;  **import** com.company.dao.AccountDao; **import** com.company.dao.impl.AccountDaoImpl; **import** com.company.service.BusinessService;  **public class** BusinessServiceImpl **implements** BusinessService {   **private** AccountDao **accountDao** = **new** AccountDaoImpl();   @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  **accountDao**.transfer(srcAccountName, targetAccountName, money);  } } |

（6）测试：

|  |
| --- |
| **package** com.company;  **import** com.company.service.BusinessService; **import** com.company.service.impl.BusinessServiceImpl; **import** org.junit.Test;  **public class** AppTest {  BusinessService **businessService** = **new** BusinessServiceImpl();  *// 测试转账* @Test  **public void** testTransfer() {  **businessService**.transfer(**"张三"**, **"李四"**, 100);  } } |

运行程序即可转账。但是在上面的DAO层代码中，并没有处理事务，这是不允许的，如果两次update()方法有一个出现异常，则没有事务的保证下，金额将出现我们之前讲的错误（我们可以使用int i = 1/0;这个代码进行模拟）。

因此我们现在讲上述DAO实现改为带有事务的版本，标记此版为第一版。

如何改呢？首先创建QueryRunner对象时要使用无参构造，进而使用带有connection的update方法，使两次使用的对象是同一个对象，这样才能进行事务控制。

为什么不能使用传递数据源的QueryRunner对象呢？因为这时会直接使用不带connection参数的update（或query、batch等）重载，而且对于不带connection参数的重载，DbUtils会自动在update等方法执行结束时关闭从数据源获得的connection对象，这样的话，每次调用update等方法时，使用的连接对象是不同的，这样就无法进行事务等的控制了（因为事务控制要在同一个连接对象中进行）。而对于带connection的重载，DbUtils不会关闭此连接对象，而是等待开发者手动关闭，因此DbUtils类专门提供了close()方法用于关闭connecion对象。这些都是看完DbUtils源码知道的，很简单，读者可阅览。

也就是说，如果使用的是带有数据源的QueryRunner对象，则使用update等方法时，方法会自动帮你关闭连接资源，无需开发者关心，但是不利于事务的控制；而如果使用不带数据源的QueryRunner，自己传递connection对象来调用update等方法，那么需要自己主动关闭connection对象，这时便于进行连接的事务等的控制。

经过上面的说明，我们就能理解为什么需要自己传递连接对象来实现事务了注意的是要自己关闭连接对象。因此，这里我们将DAO实现代码改为：

|  |
| --- |
| **package** com.company.dao.impl;  **import** com.company.dao.AccountDao; **import** com.company.util.DBCPUtil; **import** org.apache.commons.dbutils.DbUtils; **import** org.apache.commons.dbutils.QueryRunner;  **import** java.sql.Connection; **import** java.sql.SQLException;  **public class** AccountDaoImpl **implements** AccountDao {  *// 无数据源的QueryRunner。* **private** QueryRunner **queryRunner** = **new** QueryRunner();   @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  Connection connection = **null**;  **try** {  connection = DBCPUtil.*getConnection*(); *// 获得连接* connection.setAutoCommit(**false**); *// 开启事务  // 下面两个方法使用同一个连接对象* **queryRunner**.update(connection, **"update account set money=money-? where name=?"**, money, srcAccountName);  **queryRunner**.update(connection, **"update account set money=money+? where name=?"**, money, targetAccountName);  connection.commit(); *// 提交事务* } **catch** (SQLException e) {  **try** {  DbUtils.*rollback*(connection); *// 回滚事务。这里使用了DbUtils提供的回滚方法，其中实现和我们写的是一样的，也是先判断connection不为null再回滚，效果一样的，这里直接使用DbUtils的方法。* } **catch** (SQLException e1) {  e1.printStackTrace();  }  **throw new** RuntimeException(e);  } **finally** {  **try** {  DbUtils.*close*(connection); *// 这里也直接使用DbUtils提供的close()方法关闭连接。* } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }  } } |

这样的话就实现好事务了，再测试即可。

但是，这个第一版的代码是完全不合格的！为什么呢？有什么问题呢？

我们分层的目的是什么呢？对于DAO只用来CRUD。为什么要把上面两个update放在一个事务中呢？这实际上是业务的要求。因此事务控制实际上应该写在Service中。

总之：事务要求都是业务上的要求，应在Service中控制。

这时我们进行如下的更改：

（1）DAO接口：其实对于转张就需要两个方法，一是查询功能，二是更新功能，而不提供转账功能。这样DAO才是就只做CRUD：

|  |
| --- |
| **package** com.company.dao;  **import** com.company.domain.Account;  **public interface** AccountDao {  */\*\*  \* 根据姓名查询账户  \** ***@param accountName*** *姓名  \** ***@return*** *账户数据  \*/* Account findAccountByName(String accountName);   */\*\*  \* 更新账户数据  \** ***@param account*** *账户数据  \*/* **void** updateAccount(Account account); } |

（2）DAO实现（现在不考虑事务，使用数据源）：

|  |
| --- |
| **package** com.company.dao.impl;  **import** com.company.dao.AccountDao; **import** com.company.domain.Account; **import** com.company.util.DBCPUtil; **import** org.apache.commons.dbutils.QueryRunner; **import** org.apache.commons.dbutils.handlers.BeanHandler;  **import** java.sql.SQLException;  **public class** AccountDaoImpl **implements** AccountDao {   **private** QueryRunner **queryRunner** = **new** QueryRunner(DBCPUtil.*getDataSource*());   @Override  **public** Account findAccountByName(String accountName) {  **try** {  **return queryRunner**.query(**"select name, money from account where name=?"**, **new** BeanHandler<Account>(Account.**class**), accountName);  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   @Override  **public void** updateAccount(Account account) {  **try** {  **queryRunner**.update(**"update account set money=? where name=?"**, account.getMoney(), account.getName());  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  } } |

（3）最后BusinessServiceImpl中修改代码为：

|  |
| --- |
| **package** com.company.service.impl;  **import** com.company.dao.AccountDao; **import** com.company.dao.impl.AccountDaoImpl; **import** com.company.domain.Account; **import** com.company.service.BusinessService;  **public class** BusinessServiceImpl **implements** BusinessService {   **private** AccountDao **accountDao** = **new** AccountDaoImpl();   @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  *// 查询到转账双方信息* Account srcAccount = **accountDao**.findAccountByName(srcAccountName);  Account targetAccount = **accountDao**.findAccountByName(targetAccountName);  *// 更新金额* srcAccount.setMoney(srcAccount.getMoney() - money);  targetAccount.setMoney(targetAccount.getMoney() + money);  *// 更新账户* **accountDao**.updateAccount(srcAccount);  **accountDao**.updateAccount(targetAccount);  } } |

这样写完后，各层职能明确，更加符合面向对象的思想。但是还没有考虑事务的处理。如果Service层更新Account对象时有一个发生异常，没有事务控制的话，也会发生金额错误。如果加上事务处理，如何修改代码以实现在Service层加上事务控制呢？要想在同一个事务中，那么Service中执行的DAO操作必须在同一个connection中。

为了实现在DAO中使用同一个connection，只能在DAO实现类中，让使用者（即Service）注入一个connection，这时这个connection是在Service层中创建的。

因此现在的DAO实现代码如下：

|  |
| --- |
| **package** com.company.dao.impl;  **public class** AccountDaoImpl **implements** AccountDao {  **private** Connection **connection**;  **private** QueryRunner **queryRunner** = **new** QueryRunner(); *// 使用无参构造    // 注入connection* **public** AccountDaoImpl(Connection connection) {  **this**.**connection** = connection;  }   *// 下面的两个方法都使用带connection参数的重载。* @Override  **public** Account findAccountByName(String accountName) {  **try** {  **return queryRunner**.query(**connection**, **"select name, money from account where name=?"**, **new** BeanHandler<Account>(Account.**class**), accountName);  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   @Override  **public void** updateAccount(Account account) {  **try** {  **queryRunner**.update(**connection**, **"update account set money=? where name=?"**, account.getMoney(), account.getName());  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  } } |

而把获得连接和事务控制写在Service中：

|  |
| --- |
| **package** com.company.service.impl;  **import** com.company.dao.AccountDao; **import** com.company.dao.impl.AccountDaoImpl; **import** com.company.domain.Account; **import** com.company.service.BusinessService; **import** com.company.util.DBCPUtil; **import** org.apache.commons.dbutils.DbUtils;  **import** java.sql.Connection; **import** java.sql.SQLException;  **public class** BusinessServiceImpl **implements** BusinessService {   @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  Connection connection = **null**;  **try** {  connection = DBCPUtil.*getConnection*();  connection.setAutoCommit(**false**); *// 开启事务* AccountDao accountDao = **new** AccountDaoImpl(connection); *// 这时AccountDao就在方法中维护了。  // 下面是转账代码* Account srcAccount = accountDao.findAccountByName(srcAccountName);  Account targetAccount = accountDao.findAccountByName(targetAccountName);  srcAccount.setMoney(srcAccount.getMoney() - money);  targetAccount.setMoney(targetAccount.getMoney() + money);  accountDao.updateAccount(srcAccount);  **int** i = 9/0;  accountDao.updateAccount(targetAccount);  *// 最后提交事务* connection.commit();  } **catch** (Exception e) {  **try** {  DbUtils.*rollback*(connection);  } **catch** (SQLException e1) {  e1.printStackTrace();  }  **throw new** RuntimeException(e);  } **finally** {  **try** {  DbUtils.*close*(connection);  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }  } } |

这样就把事务控制写在了Service中了。经过测试，事务控制是有效的。

这就是第二个版本。但是这个版本还是不好，因为Service中用到了JDBC的API，而这部分API应该只在DAO中用到的，这样导致Service中代码看起来比较混乱，不利于解耦。那怎么解决呢？又要在DAO中使用同一个连接，又不希望在Service中使用JDBC的API，这个矛盾问题如何解决呢？

我们使用“第三版”解决这个问题。

在第三版中我们使用一个中间类TransactionManager管理这个事务，这个类就称为事务管理器。在类中，我们利用ThreadLocal存储当前线程的连接，由于事务用到同一个连接，那么他们肯定需要在一个线程中，因此将连接对象与线程进行绑定，即使有多个线程同时访问数据库，也不会发生混乱。我们的事务管理器代码如下，其中就提供了管理连接和事务的方法：

|  |
| --- |
| **package** com.company.util;  **import** java.sql.Connection; **import** java.sql.SQLException;  *// 事务管理器。* **public class** TransactionManager {  *// 一个只能存储Connection的ThreadLocal* **private static** ThreadLocal<Connection> *threadLocal* = **new** ThreadLocal<>();   *// 得到与当前线程绑定的连接对象* **public static** Connection getConnection() {  Connection connection = (Connection) *threadLocal*.get();  **if** (connection == **null**) {  *// 如果拿不到连接，就从池中获得一个连接，并绑定到当前线程。* connection = DBCPUtil.*getConnection*();  *threadLocal*.set(connection);  }  **return** connection;  }   *// 开启事务的方法* **public static void** startTransaction() {  Connection connection = *getConnection*(); *// 调用本身的getConnection方法拿到连接* **try** {  connection.setAutoCommit(**false**); *// 开启事务* } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   *// 提交事务的方法* **public static void** commitTransaction() {  Connection connection = *getConnection*();  **try** {  connection.commit();  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   *// 回滚事务的方法* **public static void** rollbackTransaction() {  Connection connection = *getConnection*();  **try** {  connection.rollback();  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   *// 关闭连接，释放资源的方法* **public static void** release() {  Connection connection = *getConnection*();  **try** {  connection.close();  *// 关闭后，需要把连接与线程解绑，调用remove方法移除对象即可。  threadLocal*.remove();  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  } } |

那么DAO实现类这时需要从TransactionManager获得连接：

|  |
| --- |
| **package** com.company.dao.impl;  **import** com.company.dao.AccountDao; **import** com.company.domain.Account; **import** com.company.util.TransactionManager; **import** org.apache.commons.dbutils.QueryRunner; **import** org.apache.commons.dbutils.handlers.BeanHandler;  **import** java.sql.SQLException;  **public class** AccountDaoImpl **implements** AccountDao {   **private** QueryRunner **queryRunner** = **new** QueryRunner(); *// 使用无参构造  // 这时也无需注入Connection   // 下面使用TransactionManager.getConnection()获得连接。* @Override  **public** Account findAccountByName(String accountName) {  **try** {  **return queryRunner**.query(TransactionManager.*getConnection*(), **"select name, money from account where name=?"**, **new** BeanHandler<Account>(Account.**class**), accountName);  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   @Override  **public void** updateAccount(Account account) {  **try** {  **queryRunner**.update(TransactionManager.*getConnection*(), **"update account set money=? where name=?"**, account.getMoney(), account.getName());  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  } } |

然后Service的实现能够使用和以前一样的结构，只要加上事务的处理即可：

|  |
| --- |
| **package** com.company.service.impl;  **import** com.company.dao.AccountDao; **import** com.company.dao.impl.AccountDaoImpl; **import** com.company.domain.Account; **import** com.company.service.BusinessService; **import** com.company.util.TransactionManager;  **public class** BusinessServiceImpl **implements** BusinessService {  *// 还是在类中维护accountDao* AccountDao **accountDao** = **new** AccountDaoImpl();  @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  **try** {  TransactionManager.*startTransaction*(); *// 开始事务  // 下面是转账代码* Account srcAccount = **accountDao**.findAccountByName(srcAccountName);  Account targetAccount = **accountDao**.findAccountByName(targetAccountName);  srcAccount.setMoney(srcAccount.getMoney() - money);  targetAccount.setMoney(targetAccount.getMoney() + money);  **accountDao**.updateAccount(srcAccount);  **accountDao**.updateAccount(targetAccount);  TransactionManager.*commitTransaction*(); *// 提交事务* } **catch** (Exception e) {  TransactionManager.*rollbackTransaction*(); *// 回滚* e.printStackTrace();  } **finally** {  TransactionManager.*release*(); *// 释放资源* }  } } |

代码既简洁又有效。可以实现事务控制。

那么是不是这个版本就完美了呢？不是的。开发者应该将主要精力放在业务开发中，如果Service中多个方法都需要控制事务，那么就要为每个方法写事务控制的代码（例如上面通过事务控制器进行控制的代码），导致工作量大而重复性高的代码出现。而且，这些方法有时还会提出其他的功能扩展，例如要统计执行时间、记录日志等等，而有时又要求去掉这些功能。如果每个方法都像事务这样进行编码控制，是没有灵活性的，只会增大工作量和重复性操作的工作量，耗费精力。而编程上有一个开闭原则：不能轻易改变已被使用的代码；功能上应该根据需求的发展区不断扩展。

其实，我们可以把事务、方法性能和日志的统计这些功能看做是方法实现中的某一些层面，这个层面就是所谓的“横切面”，即以不同的方面来看待方法。我们希望能灵活地配置这些切面，实现功能的扩展，这就是“面向切面编程”，即AOP。这样，开发者只需关注业务实现即可，因为事务、日志这些东西并不是业务导向真正要求的。

现在我们就来进行AOP编程，实现在AOP中处理Service的事务，进而让Service中只需要编写业务代码。如何进行AOP编程呢？这就需要用到我们前面学习的代理，因为代理可以扩展被代理类的所有方法，因此这里利用代理，可以为Service中的每个方法添加事务控制的切面编程。

我们使用工厂设计模式，利用一个BusinessServiceFactory工厂类创建符合要求的Service实现以供使用。其中提供了getBusinessService(boolean proxy)方法，传递true就表示返回代理事务处理的类，否则返回原实现对象。当然，工厂类也可根据配置文件返回需要的对象，这个更多的工作读者可以去做。示例代码如下：

|  |
| --- |
| **package** com.company.util;  **import** com.company.service.BusinessService; **import** com.company.service.impl.BusinessServiceImpl;  **import** java.lang.reflect.InvocationHandler; **import** java.lang.reflect.Method; **import** java.lang.reflect.Proxy;  **public class** BusinessServiceFactory {  **public static** BusinessService getBusinessService(**boolean** proxy) {  BusinessService businessService = **new** BusinessServiceImpl();  **if** (proxy) {  *// 进行代理，返回代理后的对象* **return** (BusinessService) Proxy.*newProxyInstance*(businessService.getClass().getClassLoader(), businessService.getClass().getInterfaces(), **new** InvocationHandler() {  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  *// 给每个方法加上事务的代码，就是把原来写在Service实现类方法中的事务移到这里  // 在执行原方法的前后执行事务代码* Object rtValue = **null**; *// 调用原来方法的返回值* **try** {  TransactionManager.*startTransaction*(); *// 开始事务  // 下面调用原本Service中的业务方法* rtValue = method.invoke(businessService, args);  TransactionManager.*commitTransaction*(); *// 提交事务* } **catch** (Exception e) {  TransactionManager.*rollbackTransaction*(); *// 回滚* e.printStackTrace();  } **finally** {  TransactionManager.*release*(); *// 释放资源* }  **return** rtValue;  }  });  }  *// proxy为假，直接返回原对象* **return** businessService;  } } |

而此时Service实现类就只关注业务，去掉其余代码，重归干净：

|  |
| --- |
| **package** com.company.service.impl;  **import** com.company.dao.AccountDao; **import** com.company.dao.impl.AccountDaoImpl; **import** com.company.domain.Account; **import** com.company.service.BusinessService; **import** com.company.util.TransactionManager;  **public class** BusinessServiceImpl **implements** BusinessService {  AccountDao **accountDao** = **new** AccountDaoImpl();  @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  *// 这里直接写业务代码* Account srcAccount = **accountDao**.findAccountByName(srcAccountName);  Account targetAccount = **accountDao**.findAccountByName(targetAccountName);  srcAccount.setMoney(srcAccount.getMoney() - money);  targetAccount.setMoney(targetAccount.getMoney() + money);  **accountDao**.updateAccount(srcAccount);  **accountDao**.updateAccount(targetAccount);  } } |

其他类是不用改变的。

这样，在测试类中，如果使用处理事务的代理类，就会有事务处理：

|  |
| --- |
| **package** com.company;  **import** com.company.service.BusinessService; **import** com.company.util.BusinessServiceFactory; **import** org.junit.Test;  **public class** AppTest {  *// 返回经过事务代理的实现类对象* BusinessService **businessService** = BusinessServiceFactory.*getBusinessService*(**true**);  *// 测试转账* @Test  **public void** testTransfer() {  **businessService**.transfer(**"张三"**, **"李四"**, 100);  } } |

这样就实现了AOP编程，使得核心代码中全是和业务相关的代码，只关心业务代码，这是面向切面编程的核心思想。且扩展也方便，例如要想再为方法添加执行时长方面的扩展，只要在代理时添加该功能即可：

|  |
| --- |
| **package** com.company.util;  **import** com.company.service.BusinessService; **import** com.company.service.impl.BusinessServiceImpl;  **import** java.lang.reflect.InvocationHandler; **import** java.lang.reflect.Method; **import** java.lang.reflect.Proxy;  **public class** BusinessServiceFactory {  **public static** BusinessService getBusinessService(**boolean** proxy) {  BusinessService businessService = **new** BusinessServiceImpl();  **if** (proxy) {  *// 进行代理，返回代理后的对象* **return** (BusinessService) Proxy.*newProxyInstance*(businessService.getClass().getClassLoader(), businessService.getClass().getInterfaces(), **new** InvocationHandler() {  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  *// 记录开始执行的时间* **long** startMillis = System.*currentTimeMillis*();  Object rtValue = **null**;  **try** {  TransactionManager.*startTransaction*(); *// 开始事务  // 下面调用原本Service中的业务方法* rtValue = method.invoke(businessService, args);  TransactionManager.*commitTransaction*(); *// 提交事务* } **catch** (Exception e) {  TransactionManager.*rollbackTransaction*(); *// 回滚* e.printStackTrace();  } **finally** {  TransactionManager.*release*(); *// 释放资源* }  *// 打印执行的时间* System.***out***.println(**"方法"** + method.getName() + **"执行花费时间："** + (System.*currentTimeMillis*() - startMillis) + **"毫秒"**);  **return** rtValue;  }  });  }  **return** businessService;  } } |

以上，就是我们的第四个版本的案例了，也是最终案例，实现了我们想要的最舒服的效果：各层职责明确、关注业务、代码简洁、低耦合、可扩展性高。

其实，本标题也不能起名叫“使用DbUtils实现事务控制”，实际上只要是在实际的项目中，不管是用什么技术，都应该在Service中进行事务控制，本节只是利用了DbUtils而已。关于本节内容，特别是代理和AOP思想，以后的Spring框架使用中也会有更深刻的应用和体会，可以再回头看看本节！本节这些都是一层一层地进行讲解过来的！

最后不得不提一下讲解这些知识的王昭廷老师，真是讲的太好了。他从“代理 - 数据库连接池的实现原理 - 自己利用连接池实现框架 - DBUtils - 事务控制和AOP”一路讲下来，真的是一气呵成，功力深厚，让人心无困惑，真是好奇此人怎么学的！这也鼓励我继续努力！

# 附1：JDBC连接SQL Server数据库

步骤如下：

（1）下载适用于SQL Server的JDBC驱动，网址：<https://www.microsoft.com/zh-CN/download/details.aspx?id=11774> 。下载tar.gz压缩包即可。下载前可看看页面上的说明。在“系统要求”中可知道，若使用Java8，则可使用Sqljdbc42.jar包，支持的数据库版本是SQL Server2008——2016。

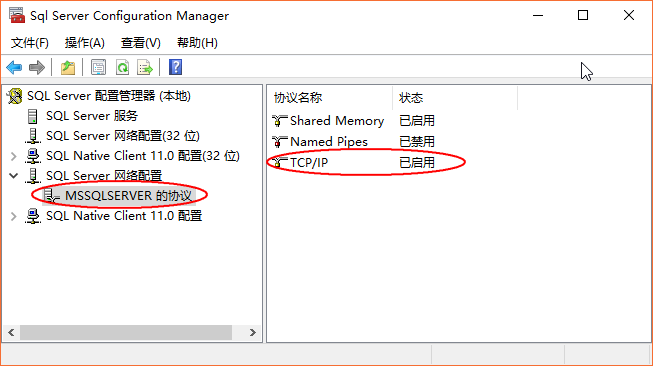
（2）解压压缩包，拿到jre8目录中的jar文件，引用到项目中。

（3）然后即可正常写Java代码。注意的是：数据库连接url格式为：jdbc:sqlserver://IP地址:端口;databaseName=数据库名；需要加载的驱动类为：com.microsoft.sqlserver.jdbc.SQLServerDriver。SQLServer默认端口是1433。

（4）Java代码中使用sa账户和密码验证。这时可成功连接数据库。示例代码：

|  |
| --- |
| Class.*forName*(**"com.microsoft.sqlserver.jdbc.SQLServerDriver"**); Connection conn = DriverManager.*getConnection*(**"jdbc:sqlserver://localhost:1433;databaseName=MyTest"**, **"sa"**, **"123456"**); |

有时不能正常连接数据库，则检查SQL Server服务是否打开了，并且还要保证SQL Server的网络配置正常：打开“Sql Server Configuration Manager”，设置“MSSQLSERVER的协议”中，TCP/IP为“启用”状态，如图：



若使用外界的计算机访问本地数据库，即远程访问，那么本机也需要在防火墙中配置1433端口为“允许入站”，方法和MySQL配置时方法一致。

另外，学习过.Net的人还会使用Windows连接验证方式，这样无需使用密码，但是不推荐这样做，并且这样做较麻烦。步骤如下：

（1）数据库连接url使用Windows验证，即加上“integratedSecurity=true”，比如：“jdbc:sqlserver://localhost:1433;databaseName=MyTest;integratedSecurity=true”。

（2）Java代码中无需写用户名和密码，即：“DriverManager.getConnection(url)”即可。

（3）仅仅上述步骤会出错。还需要向Windows系统注册一个“sqljdbc\_auth.dll”的文件。该文件在压缩包的auth文件夹中已经提供了。根据自己电脑位数选择合适的版本注册。有两种注册的方法：1.把该dll所在文件夹添加为Path环境变量；2.把该dll文件添加到“C:/Windows/System32”文件夹中。

温馨提示：连接SQL Server数据库时，也能使用DBUtils组件和连接池技术（DBCP和C3P0）。只是需要注意下述问题：

使用DBUtils时，SQL Server不能正确识别处理预编译的SQL语句，需要在创建QueryRunner对象时，提供一个knowParaType参数，此参数类型是boolean，设置为true即可。比如：

QueryRunner qr = new QueryRunner(true)；

当使用数据库连接池提供的DataSource数据源时，也别忘了提供此参数：

QueryRunner qr = new QueryRunner(ds, true);