# day18 JDBC第二天

## 今日内容

* 事务
* 连接池
* ThreadLocal
* BaseServlet自定义Servlet父类（只要求会用，不要求会写）
* DBUtils 🡪 commons-dbutils

# 事务

* 事务的四大特性：ACID；
* mysql中操作事务
* jdbc中操作事务

## 事务概述

为了方便演示事务，我们需要创建一个account表：

|  |
| --- |
| **CREATE TABLE account(**  **id INT PRIMARY KEY AUTO\_INCREMENT,**  **NAME VARCHAR(30),**  **balance NUMERIC(10.2)**  **);**  **INSERT INTO account(NAME,balance) VALUES('zs', 100000);**  **INSERT INTO account(NAME,balance) VALUES('ls', 100000);**  **INSERT INTO account(NAME,balance) VALUES('ww', 100000);**  **SELECT \* FROM account;** |

###### 1　什么是事务

银行转账！张三转10000块到李四的账户，这其实需要两条SQL语句：

* 给张三的账户减去10000元；
* 给李四的账户加上10000元。

如果在第一条SQL语句执行成功后，在执行第二条SQL语句之前，程序被中断了（可能是抛出了某个异常，也可能是其他什么原因），那么李四的账户没有加上10000元，而张三却减去了10000元。这肯定是不行的！

你现在可能已经知道什么是事务了吧！事务中的多个操作，要么完全成功，要么完全失败！不可能存在成功一半的情况！也就是说给张三的账户减去10000元如果成功了，那么给李四的账户加上10000元的操作也必须是成功的；否则给张三减去10000元，以及给李四加上10000元都是失败的！

###### 2　事务的四大特性（ACID）

**面试！**

事务的四大特性是：

* 原子性（Atomicity）：事务中所有操作是不可再分割的原子单位。事务中所有操作要么全部执行成功，要么全部执行失败。
* 一致性（Consistency）：事务执行后，数据库状态与其它业务规则保持一致。如转账业务，无论事务执行成功与否，参与转账的两个账号余额之和应该是不变的。
* 隔离性（Isolation）：隔离性是指在并发操作中，不同事务之间应该隔离开来，使每个并发中的事务不会相互干扰。
* 持久性（Durability）：一旦事务提交成功，事务中所有的数据操作都必须被持久化到数据库中，即使提交事务后，数据库马上崩溃，在数据库重启时，也必须能保证通过某种机制恢复数据。

###### 3　MySQL中的事务

在默认情况下，MySQL每执行一条SQL语句，都是一个单独的事务。如果需要在一个事务中包含多条SQL语句，那么需要开启事务和结束事务。

* 开启事务：**start transaction**；
* 结束事务：**commit**或**rollback**。

在执行SQL语句之前，先执行strat transaction，这就开启了一个事务（事务的起点），然后可以去执行多条SQL语句，最后要结束事务，commit表示提交，即事务中的多条SQL语句所做出的影响会持久化到数据库中。或者rollback，表示回滚，即回滚到事务的起点，之前做的所有操作都被撤消了！

下面演示zs给li转账10000元的示例：

|  |
| --- |
| START TRANSACTION;  UPDATE account SET balance=balance-10000 WHERE id=1;  UPDATE account SET balance=balance+10000 WHERE id=2;  ROLLBACK; |
| START TRANSACTION;  UPDATE account SET balance=balance-10000 WHERE id=1;  UPDATE account SET balance=balance+10000 WHERE id=2;  COMMIT; |
| START TRANSACTION;  UPDATE account SET balance=balance-10000 WHERE id=1;  UPDATE account SET balance=balance+10000 WHERE id=2;  quit; |

## JDBC事务

在jdbc中处理事务，都是通过Connection完成的！

### 同一事务中所有的操作，都在使用同一个Connection对象！

###### 1　JDBC中的事务

Connection的三个方法与事务相关：

* setAutoCommit(boolean)：设置是否为自动提交事务，如果true（默认值就是true）表示自动提交，也就是每条执行的SQL语句都是一个单独的事务，如果设置false，那么就相当于开启了事务了；**con.setAutoCommit(false)表示开启事务！！！**
* commit()：提交结束事务；**con.commit();表示提交事务**
* rollback()：回滚结束事务。**con.rollback();表示回滚事务**

jdbc处理事务的代码格式：

try {

con.setAutoCommit(false);//开启事务…

….

…

con.commit();//try的最后提交事务

} catch() {

con.rollback();//回滚事务

}

|  |
| --- |
| **public** **void** transfer(**boolean** b) {  Connection con = **null**;  PreparedStatement pstmt = **null**;    **try** {  con = JdbcUtils.*getConnection*();  //手动提交  con.setAutoCommit(**false**);    String sql = "update account set balance=balance+? where id=?";  pstmt = con.prepareStatement(sql);    //操作  pstmt.setDouble(1, -10000);  pstmt.setInt(2, 1);  pstmt.executeUpdate();    // 在两个操作中抛出异常  **if**(b) {  **throw** **new** Exception();  }    pstmt.setDouble(1, 10000);  pstmt.setInt(2, 2);  pstmt.executeUpdate();    //提交事务  con.commit();  } **catch**(Exception e) {  //回滚事务  **if**(con != **null**) {  **try** {  con.rollback();  } **catch**(SQLException ex) {}  }  **throw** **new** RuntimeException(e);  } **finally** {  //关闭  JdbcUtils.*close*(con, pstmt);  }  } |

###### 2　保存点（了解）

保存点是JDBC3.0的东西！当要求数据库服务器支持保存点方式的回滚。

校验数据库服务器是否支持保存点！

|  |
| --- |
| boolean b = con.getMetaData().supportsSavepoints(); |

保存点的作用是允许事务回滚到指定的保存点位置。在事务中设置好保存点，然后回滚时可以选择回滚到指定的保存点，而不是回滚整个事务！**注意，回滚到指定保存点并没有结束事务！！！只有回滚了整个事务才算是结束事务了！**

Connection类的设置保存点，以及回滚到指定保存点方法：

* 设置保存点：Savepoint setSavepoint()；
* 回滚到指定保存点：void rollback(Savepoint)。

|  |
| --- |
| /\*  \* 李四对张三说，如果你给我转1W，我就给你转100W。  \* ==========================================  \*  \* 张三给李四转1W（张三减去1W，李四加上1W）  \* 设置保存点！  \* 李四给张三转100W（李四减去100W，张三加上100W）  \* 查看李四余额为负数，那么回滚到保存点。  \* 提交事务  \*/  @Test  **public** **void** fun() {  Connection con = **null**;  PreparedStatement pstmt = **null**;    **try** {  con = JdbcUtils.*getConnection*();  //手动提交  **con.setAutoCommit(false);**    String sql = "update account set balance=balance+? where name=?";  pstmt = con.prepareStatement(sql);    //操作1（张三减去1W）  pstmt.setDouble(1, -10000);  pstmt.setString(2, "zs");  pstmt.executeUpdate();    //操作2（李四加上1W）  pstmt.setDouble(1, 10000);  pstmt.setString(2, "ls");  pstmt.executeUpdate();    // 设置保存点  Savepoint sp = con.setSavepoint();    //操作3（李四减去100W）  pstmt.setDouble(1, -1000000);  pstmt.setString(2, "ls");  pstmt.executeUpdate();    //操作4（张三加上100W）  pstmt.setDouble(1, 1000000);  pstmt.setString(2, "zs");  pstmt.executeUpdate();    //操作5（查看李四余额）  sql = "select balance from account where name=?";  pstmt = con.prepareStatement(sql);  pstmt.setString(1, "ls");  ResultSet rs = pstmt.executeQuery();  rs.next();  **double** balance = rs.getDouble(1);  　　　　　　//如果李四余额为负数，那么回滚到指定保存点  **if**(balance < 0) {  con.rollback(sp);  System.*out*.println("张三，你上当了！");  }    //提交事务  con.commit();  } **catch**(Exception e) {  //回滚事务  **if**(con != **null**) {  **try** {  con.rollback();  } **catch**(SQLException ex) {}  }  **throw** **new** RuntimeException(e);  } **finally** {  //关闭  JdbcUtils.*close*(con, pstmt);  }  } |

## 事务隔离级别

###### 事务的并发读问题

* 脏读：读取到另一个事务未提交数据；
* 不可重复读：两次读取不一致；
* 幻读（虚读）：读到另一事务已提交数据。

###### 2　并发事务问题

因为并发事务导致的问题大致有5类，其中两类是更新问题，三类是读问题。

* 脏读（dirty read）：**读到另一个事务的未提交更新数据，即读取到了脏数据；**
* 不可重复读（unrepeatable read）：**对同一记录的两次读取不一致，因为另一事务对该记录做了修改；**
* 幻读（虚读）（phantom read）：**对同一张表的两次查询不一致，因为另一事务插入了一条记录；**

脏读

事务1：张三给李四转账100元

事务2：李四查看自己的账户

* t1：事务1：开始事务
* t2：事务1：张三给李四转账100元
* t3：事务2：开始事务
* t4：事务2：李四查看自己的账户，看到账户多出100元（脏读）
* t5：事务2：提交事务
* t6：事务1：回滚事务，回到转账之前的状态

不可重复读

事务1：酒店查看两次1048号房间状态

事务2：预订1048号房间

* t1：事务1：开始事务
* t2：事务1：查看1048号房间状态为空闲
* t3：事务2：开始事务
* t4：事务2：预定1048号房间
* t5：事务2：提交事务
* t6：事务1：再次查看1048号房间状态为使用
* t7：事务1：提交事务

对同一记录的两次查询结果不一致！

幻读

事务1：对酒店房间预订记录两次统计

事务2：添加一条预订房间记录

* t1：事务1：开始事务
* t2：事务1：统计预订记录100条
* t3：事务2：开始事务
* t4：事务2：添加一条预订房间记录
* t5：事务2：提交事务
* t6：事务1：再次统计预订记录为101记录
* t7：事务1：提交

　　对同一表的两次查询不一致！

不可重复读和幻读的区别：

* 不可重复读是读取到了另一事务的更新；
* 幻读是读取到了另一事务的插入（MySQL中无法测试到幻读）；

###### 3　四大隔离级别

4个等级的事务隔离级别，在相同数据环境下，使用相同的输入，执行相同的工作，根据不同的隔离级别，可以导致不同的结果。不同事务隔离级别能够解决的数据并发问题的能力是不同的。

1　SERIALIZABLE（串行化）

* 不会出现任何并发问题，因为它是对同一数据的访问是串行的，非并发访问的；
* 性能最差；

2　REPEATABLE READ（可重复读）（MySQL）

* 防止脏读和不可重复读，不能处理幻读问题；
* 性能比SERIALIZABLE好

3　READ COMMITTED（读已提交数据）（Oracle）

* 防止脏读，没有处理不可重复读，也没有处理幻读；
* 性能比REPEATABLE READ好

4　READ UNCOMMITTED（读未提交数据）

* 可能出现任何事务并发问题
* 性能最好

**MySQL的默认隔离级别为REPEATABLE READ，这是一个很不错的选择吧！**

###### 5　MySQL隔离级别

MySQL的默认隔离级别为Repeatable read，可以通过下面语句查看：

|  |
| --- |
| select @@tx\_isolation |

也可以通过下面语句来设置当前连接的隔离级别：

|  |
| --- |
| set transaction isolationlevel [4先1] |

###### 6　JDBC设置隔离级别

con. setTransactionIsolation(int level)

参数可选值如下：

* Connection.TRANSACTION\_READ\_UNCOMMITTED；
* Connection.TRANSACTION\_READ\_COMMITTED；
* Connection.TRANSACTION\_REPEATABLE\_READ；
* Connection.TRANSACTION\_SERIALIZABLE。

事务总结：

* 事务的特性：ACID；
* 事务开始边界与结束边界：开始边界（con.setAutoCommit(false)），结束边界（con.commit()或con.rollback()）；
* 事务的隔离级别： READ\_UNCOMMITTED、READ\_COMMITTED、REPEATABLE\_READ、SERIALIZABLE。多个事务并发执行时才需要考虑并发事务。

# 数据库连接池

**池参数（所有池参数都有默认值）**：

初始大小：10个

最小空闲连接数：3个

增量：一次创建的最小单位（5个）

最大空闲连接数：12个

最大连接数：20个

最大的等待时间：1000毫秒

**四大连接参数**

连接池也是使用四大连接参数来完成创建连接对象！

**实现的接口**

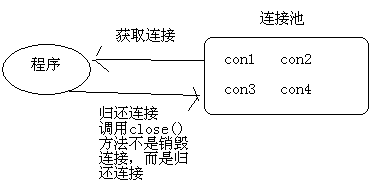
连接池必须实现：javax.sql.DataSource接口！

连接池返回的Connection对象，它的close()方法与众不同！调用它的close()不是关闭，而是把连接归还给池！

## 数据库连接池

###### 1　数据库连接池的概念

用池来管理Connection，这可以重复使用Connection。有了池，所以我们就不用自己来创建Connection，而是通过池来获取Connection对象。当使用完Connection后，调用Connection的close()方法也不会真的关闭Connection，而是把Connection“归还”给池。池就可以再利用这个Connection对象了。



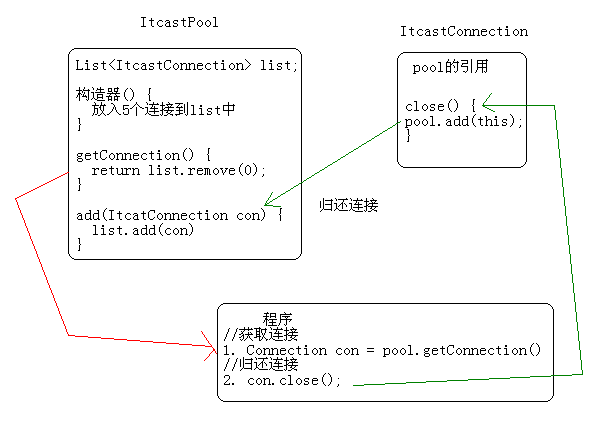
###### 2　JDBC数据库连接池接口（DataSource）

　　Java为数据库连接池提供了公共的接口：javax.sql.DataSource，各个厂商可以让自己的连接池实现这个接口。这样应用程序可以方便的切换不同厂商的连接池！

###### 3　自定义连接池（ItcastPool）

　　分析：ItcastPool需要有一个List，用来保存连接对象。在ItcastPool的构造器中创建5个连接对象放到List中！当用人调用了ItcastPool的getConnection()时，那么就从List拿出一个返回。当List中没有连接可用时，抛出异常。

　　我们需要对Connection的close()方法进行增强，所以我们需要自定义ItcastConnection类，对Connection进行装饰！即对close()方法进行增强。因为需要在调用close()方法时把连接“归还”给池，所以ItcastConnection类需要拥有池对象的引用，并且池类还要提供“归还”的方法。



ItcastPool.java

|  |
| --- |
| **public** **class** ItcastPool **implements** DataSource {  **private** **static** Properties *props* = **new** Properties();  **private** List<Connection> list = **new** ArrayList<Connection>();  **static** {  InputStream in = ItcastPool.**class**.getClassLoader()  .getResourceAsStream("dbconfig.properties");  **try** {  *props*.load(in);  Class.*forName*(*props*.getProperty("driverClassName"));  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  **public** ItcastPool() **throws** SQLException {  **for** (**int** i = 0; i < 5; i++) {  Connection con = DriverManager.*getConnection*(  *props*.getProperty("url"), *props*.getProperty("username"),  *props*.getProperty("password"));  ItcastConnection conWapper = **new** ItcastConnection(con, **this**);  list.add(conWapper);  }  }    **public** **void** add(Connection con) {  list.add(con);  }  **public** Connection getConnection() **throws** SQLException {  **if**(list.size() > 0) {  **return** list.remove(0);  }  **throw** **new** SQLException("没连接了");  }  ......  } |

ItcastConnection.java

|  |
| --- |
| **public** **class** ItcastConnection **extends** ConnectionWrapper {  **private** ItcastPool pool;    **public** ItcastConnection(Connection con, ItcastPool pool) {  **super**(con);  **this**.pool = pool;  }  @Override  **public** **void** close() **throws** SQLException {  pool.add(**this**);  }  } |

## DBCP

###### 1　什么是DBCP？

DBCP是Apache提供的一款开源免费的数据库连接池！

Hibernate3.0之后不再对DBCP提供支持！因为Hibernate声明DBCP有致命的缺欠！DBCP因为Hibernate的这一毁谤很是生气，并且说自己没有缺欠。

###### 2　DBCP的使用

|  |
| --- |
| **public** **void** fun1() **throws** SQLException {  BasicDataSource ds = **new** BasicDataSource();  ds.setUsername("root");  ds.setPassword("123");  ds.setUrl("jdbc:mysql://localhost:3306/mydb1");  ds.setDriverClassName("com.mysql.jdbc.Driver");    ds.setMaxActive(20);  ds.setMaxIdle(10);  ds.setInitialSize(10);  ds.setMinIdle(2);  ds.setMaxWait(1000);    Connection con = ds.getConnection();  System.*out*.println(con.getClass().getName());  con.close();  } |

###### 3　DBCP的配置信息

下面是对DBCP的配置介绍：

|  |
| --- |
| #基本配置  driverClassName=com.mysql.jdbc.Driver  url=jdbc:mysql://localhost:3306/mydb1  username=root  password=123  #初始化池大小，即一开始池中就会有10个连接对象  默认值为0  initialSize=0  #最大连接数，如果设置maxActive=50时，池中最多可以有50个连接，当然这50个连接中包含被使用的和没被使用的（空闲）  #你是一个包工头，你一共有50个工人，但这50个工人有的当前正在工作，有的正在空闲  #默认值为8，如果设置为非正数，表示没有限制！即无限大  maxActive=8  #最大空闲连接  #当设置maxIdle=30时，你是包工头，你允许最多有20个工人空闲，如果现在有30个空闲工人，那么要开除10个  #默认值为8，如果设置为负数，表示没有限制！即无限大  maxIdle=8  #最小空闲连接  #如果设置minIdel=5时，如果你的工人只有3个空闲，那么你需要再去招2个回来，保证有5个空闲工人  #默认值为0  minIdle=0  #最大等待时间  #当设置maxWait=5000时，现在你的工作都出去工作了，又来了一个工作，需要一个工人。  #这时就要等待有工人回来，如果等待5000毫秒还没回来，那就抛出异常  #没有工人的原因：最多工人数为50，已经有50个工人了，不能再招了，但50人都出去工作了。  #默认值为-1，表示无限期等待，不会抛出异常。  maxWait=-1  #连接属性  #就是原来放在url后面的参数，可以使用connectionProperties来指定  #如果已经在url后面指定了，那么就不用在这里指定了。  #useServerPrepStmts=true，MySQL开启预编译功能  #cachePrepStmts=true，MySQL开启缓存PreparedStatement功能，  #prepStmtCacheSize=50，缓存PreparedStatement的上限  #prepStmtCacheSqlLimit=300，当SQL模板长度大于300时，就不再缓存它  connectionProperties=useUnicode=true;characterEncoding=UTF8;useServerPrepStmts=true;cachePrepStmts=true;prepStmtCacheSize=50;prepStmtCacheSqlLimit=300  #连接的默认提交方式  #默认值为true  defaultAutoCommit=true  #连接是否为只读连接  #Connection有一对方法：setReadOnly(boolean)和isReadOnly()  #如果是只读连接，那么你只能用这个连接来做查询  #指定连接为只读是为了优化！这个优化与并发事务相关！  #如果两个并发事务，对同一行记录做增、删、改操作，是不是一定要隔离它们啊？  #如果两个并发事务，对同一行记录只做查询操作，那么是不是就不用隔离它们了？  #如果没有指定这个属性值，那么是否为只读连接，这就由驱动自己来决定了。即Connection的实现类自己来决定！  defaultReadOnly=false  #指定事务的事务隔离级别  #可选值：NONE,READ\_UNCOMMITTED, READ\_COMMITTED, REPEATABLE\_READ, SERIALIZABLE  #如果没有指定，那么由驱动中的Connection实现类自己来决定  defaultTransactionIsolation=REPEATABLE\_READ |

## C3P0

###### 1　C3P0简介

　　C3P0也是开源免费的连接池！C3P0被很多人看好！

###### 2　C3P0的使用

　　C3P0中池类是：ComboPooledDataSource。

|  |
| --- |
| **public** **void** fun1() **throws** PropertyVetoException, SQLException {  ComboPooledDataSource ds = **new** ComboPooledDataSource();  ds.setJdbcUrl("jdbc:mysql://localhost:3306/mydb1");  ds.setUser("root");  ds.setPassword("123");  ds.setDriverClass("com.mysql.jdbc.Driver");    ds.setAcquireIncrement(5);  ds.setInitialPoolSize(20);  ds.setMinPoolSize(2);  ds.setMaxPoolSize(50);    Connection con = ds.getConnection();  System.*out*.println(con);  con.close();  } |

配置文件要求：

* 文件名称：必须叫c3p0-config.xml
* 文件位置：必须在src下

c3p0也可以指定配置文件，而且配置文件可以是properties，也可骒xml的。当然xml的高级一些了。但是c3p0的配置文件名必须为c3p0-config.xml，并且必须放在类路径下。

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <c3p0-config>  <default-config>  <property name=*"jdbcUrl"*>jdbc:mysql://localhost:3306/mydb1</property>  <property name=*"driverClass"*>com.mysql.jdbc.Driver</property>  <property name=*"user"*>root</property>  <property name=*"password"*>123</property>  <property name=*"acquireIncrement"*>3</property>  <property name=*"initialPoolSize"*>10</property>  <property name=*"minPoolSize"*>2</property>  <property name=*"maxPoolSize"*>10</property>  </default-config>  <named-config name=*"oracle-config"*>  <property name=*"jdbcUrl"*>jdbc:mysql://localhost:3306/mydb1</property>  <property name=*"driverClass"*>com.mysql.jdbc.Driver</property>  <property name=*"user"*>root</property>  <property name=*"password"*>123</property>  <property name=*"acquireIncrement"*>3</property>  <property name=*"initialPoolSize"*>10</property>  <property name=*"minPoolSize"*>2</property>  <property name=*"maxPoolSize"*>10</property>  </named-config>  </c3p0-config> |

　　c3p0的配置文件中可以配置多个连接信息，可以给每个配置起个名字，这样可以方便的通过配置名称来切换配置信息。上面文件中默认配置为mysql的配置，名为oracle-config的配置也是mysql的配置，呵呵。

|  |
| --- |
| **public** **void** fun2() **throws** PropertyVetoException, SQLException {  ComboPooledDataSource ds = **new** ComboPooledDataSource();  Connection con = ds.getConnection();  System.*out*.println(con);  con.close();  } |
| **public** **void** fun2() **throws** PropertyVetoException, SQLException {  ComboPooledDataSource ds = **new** ComboPooledDataSource("orcale-config");  Connection con = ds.getConnection();  System.*out*.println(con);  con.close();  } |

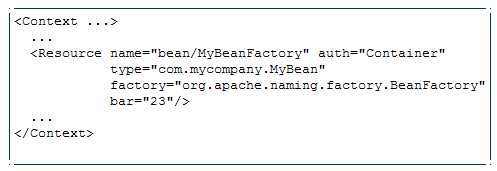
## Tomcat配置连接池

###### 1　Tomcat配置JNDI资源

JNDI（Java Naming and Directory Interface），Java命名和目录接口。JNDI的作用就是：在服务器上配置资源，然后通过统一的方式来获取配置的资源。

我们这里要配置的资源当然是连接池了，这样项目中就可以通过统一的方式来获取连接池对象了。

下图是Tomcat文档提供的：



配置JNDI资源需要到<Context>元素中配置<Resource>子元素：

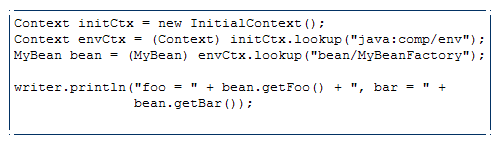
* name：指定资源的名称，这个名称可以随便给，在获取资源时需要这个名称；
* factory：用来创建资源的工厂，这个值基本上是固定的，不用修改；
* type：资源的类型，我们要给出的类型当然是我们连接池的类型了；
* bar：表示资源的属性，如果资源存在名为bar的属性，那么就配置bar的值。对于DBCP连接池而言，你需要配置的不是bar，因为它没有bar这个属性，而是应该去配置url、username等属性。

|  |
| --- |
| <Context>  <Resource name="mydbcp"  type="org.apache.tomcat.dbcp.dbcp.BasicDataSource"  factory="org.apache.naming.factory.BeanFactory"  username="root"  password="1234"  driverClassName="com.mysql.jdbc.Driver"  url="jdbc:mysql://127.0.0.1/mydb"  maxIdle="3"  maxWait="5000"  maxActive="5"  initialSize="3"/>  </Context> |
| **c3p0使用这一个配置**  <Context>  <Resource name="myc3p0"  factory="org.apache.naming.factory.BeanFactory"  type="com.mchange.v2.c3p0.ComboPooledDataSource"  jdbcUrl="jdbc:mysql://localhost:3306/mydb"  user="root"  password="1234"  driverClass="com.mysql.jdbc.Driver"  maxPoolSize="20"  minPoolSize ="5"  initialPoolSize="10"  acquireIncrement="2"/>  </Context> |

###### 2　获取资源

配置资源的目的当然是为了获取资源了。只要你启动了Tomcat，那么就可以在项目中任何类中通过JNDI获取资源的方式来获取资源了。

下图是Tomcat文档提供的，与上面Tomcat文档提供的配置资源是对应的。



获取资源：

Context：javax.naming.Context；

InitialContext：javax.naming.InitialContext；

lookup(String)：获取资源的方法，其中”java:comp/env”是资源的入口（这是固定的名称），获取过来的还是一个Context，这说明需要在获取到的Context上进一步进行获取。”bean/MyBeanFactory”对应<Resource>中配置的name值，这回获取的就是资源对象了。

|  |
| --- |
| Context cxt = new InitialContext();  DataSource ds = (DataSource)cxt.lookup("java:/comp/env/mydbcp");  Connection con = ds.getConnection();  System.out.println(con);  con.close(); |
| Context cxt = new InitialContext();  Context envCxt = (Context)cxt.lookup("java:/comp/env");  DataSource ds = (DataSource)envCxt.lookup("mydbcp");  Connection con = ds.getConnection();  System.out.println(con);  con.close(); |

上面两种方式是相同的效果。

## 修改JdbcUtils

因为已经学习了连接池，那么JdbcUtils的获取连接对象的方法也要修改一下了。

JdbcUtils.java

|  |
| --- |
| **public** **class** JdbcUtils {  **private** **static** DataSource *dataSource* = **new** ComboPooledDataSource();  **public** **static** DataSource getDataSource() {  **return** *dataSource*;  }  **public** **static** Connection getConnection() {  **try** {  **return** *dataSource*.getConnection();  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  } |

# ThreadLocal

Thread 🡪 人类

Runnable 🡪 任务类

|  |  |
| --- | --- |
| key | value |
| thread1 | aaa |
| thread2 | bbb |
| thread3 | ccc |

###### 1　ThreadLocal API

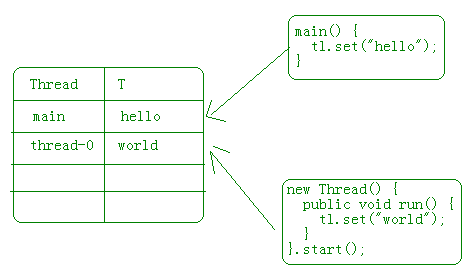
ThreadLocal类只有三个方法：

* void set(T value)：保存值；
* T get()：获取值；
* void remove()：移除值。

###### 2　ThreadLocal的内部是Map

ThreadLocal内部其实是个Map来保存数据。虽然在使用ThreadLocal时只给出了值，没有给出键，其实它内部使用了当前线程做为键。

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



# BaseServlet

###### 1　BaseServlet的作用

在开始客户管理系统之前，我们先写一个工具类：BaseServlet。

我们知道，写一个项目可能会出现N多个Servlet，而且一般一个Servlet只有一个方法（doGet或doPost），如果项目大一些，那么Servlet的数量就会很惊人。

为了避免Servlet的“膨胀”，我们写一个BaseServlet。它的作用是让一个Servlet可以处理多种不同的请求。不同的请求调用Servlet的不同方法。我们写好了BaseServlet后，让其他Servlet继承BaseServlet，例如CustomerServlet继承BaseServlet，然后在CustomerServlet中提供add()、update()、delete()等方法，每个方法对应不同的请求。

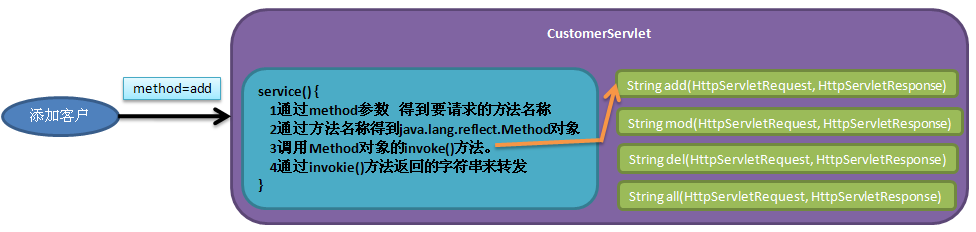


###### 2　BaseServlet分析

我们知道，Servlet中处理请求的方法是service()方法，这说明我们需要让service()方法去调用其他方法。例如调用add()、mod()、del()、all()等方法！具体调用哪个方法需要在请求中给出方法名称！然后service()方法通过方法名称来调用指定的方法。

无论是点击超链接，还是提交表单，请求中必须要有method参数，这个参数的值就是要请求的方法名称，这样BaseServlet的service()才能通过方法名称来调用目标方法。例如某个链接如下：

<a href=”/xxx/CustomerServlet?method=add”>添加客户</a>



###### 3　BaseServlet代码

|  |
| --- |
| **public** **class** BaseServlet **extends** HttpServlet {  /\*  \* 它会根据请求中的方法名，来决定调用本类的哪个方法  \*/  **protected** **void** service(HttpServletRequest req, HttpServletResponse res)  **throws** ServletException, IOException {  req.setCharacterEncoding("UTF-8");  res.setContentType("text/html;charset=utf-8");  // 例如：http://localhost:8080/demo1/xxx?m=add  String methodName = req.getParameter("method");// 它是一个方法名称    // 当没用指定要调用的方法时，那么默认请求的是execute()方法。  **if**(methodName == **null** || methodName.isEmpty()) {  methodName = "execute";  }  Class c = **this**.getClass();  **try** {  // 通过方法名称获取方法的反射对象  Method m = c.getMethod(methodName, HttpServletRequest.**class**,  HttpServletResponse.**class**);  // 反射方法目标方法，也就是说，如果methodName为add，那么就调用add方法。  String result = (String) m.invoke(**this**, req, res);  // 通过返回值完成请求转发  **if**(result != **null** && !result.isEmpty()) {  req.getRequestDispatcher(result).forward(req, res);  }  } **catch** (Exception e) {  **throw** **new** ServletException(e);  }  }  } |

# DBUtils

###### 1　DBUtils简介

DBUtils是Apache Commons组件中的一员，开源免费！

DBUtils是对JDBC的简单封装，但是它还是被很多公司使用！

DBUtils的Jar包：dbutils.jar

###### 2　DBUtils主要类

* DbUtils：都是静态方法，一系列的close()方法；
* QueryRunner：
* update()：执行insert、update、delete；
* query()：执行select语句；
* batch()：执行批处理。

###### 3　QueryRunner之更新

QueryRunner的update()方法可以用来执行insert、update、delete语句。

1. 创建QueryRunner

构造器：QueryRunner();

1. update()方法

int update(Connection con, String sql, Object… params)

|  |
| --- |
| @Test  **public** **void** fun1() **throws** SQLException {  QueryRunner qr = **new** QueryRunner();  String sql = "insert into user values(?,?,?)";  qr.update(JdbcUtils.*getConnection*(), sql, "u1", "zhangSan", "123");  } |

还有另一种方式来使用QueryRunner

1. 创建QueryRunner

构造器：QueryRunner(DataSource)

1. update()方法

int update(String sql, Object… params)

这种方式在创建QueryRunner时传递了连接池对象，那么在调用update()方法时就不用再传递Connection了。

|  |
| --- |
| @Test  **public** **void** fun2() **throws** SQLException {  QueryRunner qr = **new** QueryRunner(JdbcUtils.*getDataSource*());  String sql = "insert into user values(?,?,?)";  qr.update(sql, "u1", "zhangSan", "123");  } |

###### 4　ResultSetHandler

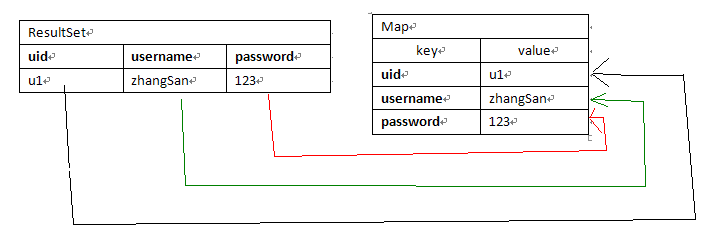
我们知道在执行select语句之后得到的是ResultSet，然后我们还需要对ResultSet进行转换，得到最终我们想要的数据。你可以希望把ResultSet的数据放到一个List中，也可能想把数据放到一个Map中，或是一个Bean中。

DBUtils提供了一个接口ResultSetHandler，它就是用来ResultSet转换成目标类型的工具。你可以自己去实现这个接口，把ResultSet转换成你想要的类型。

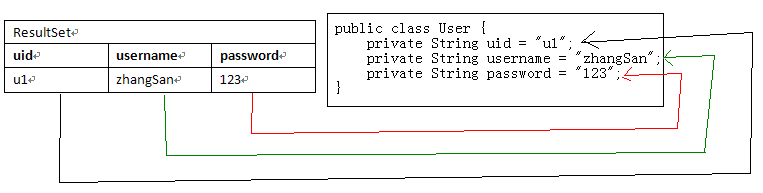
DBUtils提供了很多个ResultSetHandler接口的实现，这些实现已经基本够用了，我们通常不用自己去实现ResultSet接口了。

* MapHandler：单行处理器！把结果集转换成Map<String,Object>，其中列名为键！
* MapListHandler：多行处理器！把结果集转换成List<Map<String,Object>>；
* BeanHandler：单行处理器！把结果集转换成Bean，该处理器需要Class参数，即Bean的类型；
* BeanListHandler：多行处理器！把结果集转换成List<Bean>；
* ColumnListHandler：多行单列处理器！把结果集转换成List<Object>，使用ColumnListHandler时需要指定某一列的名称或编号，例如：new ColumListHandler(“name”)表示把name列的数据放到List中。
* ScalarHandler：单行单列处理器！把结果集转换成Object。一般用于聚集查询，例如select count(\*) from tab\_student。

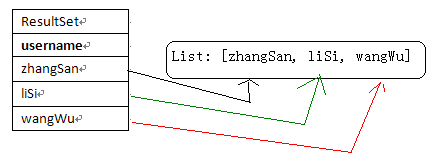
Map处理器



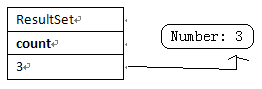
Bean处理器



Column处理器



Scalar处理器



###### 5　QueryRunner之查询

QueryRunner的查询方法是：

public <T> T query(String sql, ResultSetHandler<T> rh, Object… params)

public <T> T query(Connection con, String sql, ResultSetHandler<T> rh, Object… params)

query()方法会通过sql语句和params查询出ResultSet，然后通过rh把ResultSet转换成对应的类型再返回。

|  |
| --- |
| @Test  **public** **void** fun1() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "select \* from tab\_student where number=?";  Map<String,Object> map = qr.query(sql, **new** MapHandler(), "S\_2000");  System.*out*.println(map);  }    @Test  **public** **void** fun2() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "select \* from tab\_student";  List<Map<String,Object>> list = qr.query(sql, **new** MapListHandler());  **for**(Map<String,Object> map : list) {  System.*out*.println(map);  }  }    @Test  **public** **void** fun3() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "select \* from tab\_student where number=?";  Student stu = qr.query(sql, **new** BeanHandler<Student>(Student.**class**), "S\_2000");  System.*out*.println(stu);  }    @Test  **public** **void** fun4() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "select \* from tab\_student";  List<Student> list = qr.query(sql, **new** BeanListHandler<Student>(Student.**class**));  **for**(Student stu : list) {  System.*out*.println(stu);  }  }    @Test  **public** **void** fun5() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "select \* from tab\_student";  List<Object> list = qr.query(sql, **new** ColumnListHandler("name"));  **for**(Object s : list) {  System.*out*.println(s);  }  }    @Test  **public** **void** fun6() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "select count(\*) from tab\_student";  Number number = (Number)qr.query(sql, **new** ScalarHandler());  **int** cnt = number.intValue();  System.*out*.println(cnt);  } |

###### 5　QueryRunner之批处理

QueryRunner还提供了批处理方法：batch()。

我们更新一行记录时需要指定一个Object[]为参数，如果是批处理，那么就要指定Object[][]为参数了。即多个Object[]就是Object[][]了，其中每个Object[]对应一行记录：

|  |
| --- |
| @Test  **public** **void** fun10() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "insert into tab\_student values(?,?,?,?)";  Object[][] params = **new** Object[10][];//表示 要插入10行记录  **for**(**int** i = 0; i < params.length; i++) {  params[i] = **new** Object[]{"S\_300" + i, "name" + i, 30 + i, i%2==0?"男":"女"};  }  qr.batch(sql, params);  } |