## JDK动态代理

###### 1　什么是JDK动态代理

刚刚写ItcastConnection时爽么？因为Connection中的方法太多了，每个都要写，所以很累吧。累点到是没什么，可以完成功能就是好的。但是不是什么时候可以用代理的，有时你可能会遇到要代理的东西，只有在运行时才能知道，所以你不可能先把代理写出来！这时就需要使用动态代理。

JDK动态代理是JavaSE中一个高级特性，不是那么好理解的，但是它可是框架们的“秘密武器”。你要是可以理解它，那么将来在学习框架时，你就会知道框架是怎么完成一些“神奇功能”的。

**动态代理的作用：在运行时生成一个实现了指定接口的对象。**

　　例如在运行时生成一个对象，这个对象实现了Connection接口。

###### 2　JDK动态代理之Hello World

我们要写一个程序，这个程序会在运行时动态的生成一个对象，这个对象会实现Connection接口。

Connectoin c = (Connection)Proxy.newInstance(Connetion.class);

上面代码只是示意代码，不能编译通过的。

上面代码有个问题：生成一个实现了指定接口的对象，但是我们知道实现接口，需要为接口中每个方法添加实现内容，那么这个动态代理对象它是怎么实现Connection接口中的方法的呢？也就是说，我现在如果调用了代理对象的close()方法，它会执行什么呢？这就是问题！

想生成代理对象，还需要提供实现内容！

别的先别去管，先来看一个接口：InvocationHandler。

|  |
| --- |
| **class** HelloWorldHandler **implements** InvocationHandler {  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  System.*out*.println("Hello 动态代理!");  **return** **null**;  }  } |

InvoactionHandler接口只有一个方法，invoke()！方法的参数你不要去管是什么意思，只需要知道它只有一个方法，名字叫invoke()，一会儿再去讨论它参数的含义。

　　Connectoin c = (Connection)Proxy.newInstance(Connetion.class, new HelloWorldHandler());

　　上面代码还是示意代码，不能编译通过。

我们这回在创建代理对象时，多给出了一个参数，是一个接口的实现类。实现类中有一条输出语句“Hello 动态代理！”，现在生成的代理对象是Connection接口的实现类对象，你调用代理对象的任何方法都会调用HelloWorldHandler的invoke()方法，即输出“Hello 动态代理！”。

Connectoin c = (Connection)Proxy.newInstance(Connetion.class, new HelloWorldHandler());

c.close();

c.toString();

c.createStatement();

还是示意代码！

上面示意代码中调用了三个方法，无论哪个方法都会输出“Hello 动态代理！”。现在你知道InvocationHandler接口的作用了吧。

|  |
| --- |
| **public** **void** fun2() **throws** SQLException {  ClassLoader loader = Thread.*currentThread*().getContextClassLoader();  Class[] interfaces = {Connection.**class**};  InvocationHandler h = **new** HelloWorldHandler();  Connection con = (Connection)Proxy.*newProxyInstance*(loader, interfaces, h);  con.close();  con.toString();  con.createStatement();  } |

###### 3　真正的代理

虽然我们学会了动态代理，但还没有真正的代理。真正的代理是需要一个真正的连接对象，然后我们的代理对象使用它来完成任务。为了说明这个真正的代理，需要写几个类：

|  |
| --- |
| **public** **interface** Waiter {  **public** **void** serve();  } |
| **public** **class** WaiterImpl **implements** Waiter {  **public** **void** serve() {  System.*out*.println("服务...");  }  } |

上面代码中写了一个Waiter接口，和一个WaiterImpl，它是Waiter接口的实现类。现在我们要写一个WaiterImpl的代理类。

|  |
| --- |
| **public** **class** WaiterProxy **implements** Waiter {  **private** Waiter waiter;  **public** WaiterProxy(Waiter waiter) {  **this**.waiter = waiter;  }  **public** **void** serve() {  waiter.serve();  }  } |

上面代理中，WaiterProxy就是一个代理类，当然，这个代理类没有实际的意义，因为它没有做任何的改变，所以没有意思！通常代理类是这样的，它会去实现一个接口，但它还需要一个该接口的实现类对象，然后所有实现都使用这个对象来完成。象是上面的代理中，WaiterProxy是一个代理类，它实现了Waiter接口，而且它还需要一个Waiter类型的对象，然后所有的实现都是代理这个对象功能。但是通常代理类会对被代理的对象的一些行为做一些改动，我们的例子中没有做。

上面的WaiterProxy虽然是一个代理类，但它不是动态代理。下面是通过JDK动态代理来生成一个代理对象。

|  |
| --- |
| **public** **class** WaiterHandler **implements** InvocationHandler {  **private** Waiter waiter;  **public** WaiterHandler(Waiter waiter) {  **this**.waiter = waiter;  }  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  **return** method.invoke(waiter, args);  }  } |

|  |
| --- |
| ClassLoader loader = Thread.*currentThread*().getContextClassLoader();  Class[] interfaces = {Waiter.**class**};  Waiter watier = **new** WaiterImpl();  InvocationHandler h = **new** WaiterHandler(watier);  Waiter proxy = (Waiter)Proxy.*newProxyInstance*(loader, interfaces, h);  proxy.serve(); |

###### 4　使用动态代理完成连接池

|  |
| --- |
| **public** **class** ItcastDataSource **implements** DataSource {  **private** String username;  **private** String password;  **private** String url;  **private** String driverClassName;    **private** List<Connection> list = **new** ArrayList<Connection>();  **private** **boolean** flag = **true**;  **private** **void** init() **throws** SQLException {  flag = **false**;  **try** {  Class.*forName*(driverClassName);  } **catch**(ClassNotFoundException e) {  **throw** **new** RuntimeException(e);  }  **for**(**int** i = 0; i < 5; i++) {  **final** Connection con = DriverManager.*getConnection*(url, username, password);  ClassLoader l = Thread.*currentThread*().getContextClassLoader();  Class[] ins = {Connection.**class**};  InvocationHandler h = **new** InvocationHandler() {  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  **if**(method.getName().equals("close")) {  list.add((Connection)proxy);  **return** **null**;  } **else** {  **return** method.invoke(con, args);  }  }  };  Connection proxy = (Connection)Proxy.*newProxyInstance*(l, ins, h);  list.add(proxy);  }  }    **public** Connection getConnection() **throws** SQLException {  **if**(flag) {  init();  }  **if**(list.size() > 0) {  **return** list.remove(0);  }  **throw** **new** RuntimeException();  }  ......  } |