Spring操作数据库的框架整合

本文档讲Spring与数据库的整合，包括Spring对JDBC的支持、对ORM框架Hibernate的支持和对数据库事务处理的控制。在讲之前，请先在项目中引入如下相关的包：

|  |
| --- |
| spring-jdbc-5.0.2.RELEASE.jar （JDBC支持包）  spring-tx-5.0.2.RELEASE.jar （数据库事物支持包）  spring-orm-4.3.7.RELEASE.jar （对ORM的支持包，主要就是对Hibernate的支持）  mysql-connector-java-5.1.40-bin.jar （MySQL的JDBC驱动，这个一定不能忘记！只要用Java操作MySQL数据库，必须引入此包） |

# 1. Spring对JDBC的支持

## 1.1 Spring管理数据源

要用JDBC操作数据库，首先要拿到Connection连接对象或者DataSource数据源对象。使用了Spring后，就是用Spring来管理连接或者数据源对象。由于DataSource是Java提供的标准规范，且能实现连接池功能，因此只要将DataSource交由Spring管理即可，而不是Connection由Spring管理，Connection依然由具体的连接池管理，Spring只负责创建好DataSource。

这里，依然用druid框架作为数据库连接池，因此项目中再引入包：

|  |
| --- |
| druid-1.1.6.jar |

那么如何让Spring管理dataSource？只要在容器中注册好dataSource对象，即进行如下配置即可：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!-- 配置DruidDataSource这个bean，让Spring管理DataSource -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close"**>  *<!-- url -->* <**property name="url" value="jdbc:mysql://114.55.86.230:3306/mytest"** />  *<!-- 数据库的连接属性。在这里每个配置直接用分号隔开即可，不再用&符号，否则在XML中，&要转义为&amp; druid会自动根据分号分隔再拼接数据库的连接属性 -->* <**property name="connectionProperties" value="useUnicode=false;characterEncoding=UTF8;useSSL=false"** />  *<!-- 用户名 -->* <**property name="username" value="root"** />  *<!-- 密码 -->* <**property name="password" value="123qwe!@#"** />  *<!-- 初始化时建立物理连接的个数。  官方文档说：初始化发生在显示调用init方法，或者第一次getConnection时。  因此上面配置了init-method="init"，就意味着Spring创建本bean对象时就初始化连接了。-->* <**property name="initialSize" value="1"** />  *<!-- 最大连接池数量 -->* <**property name="maxActive" value="10"** />  *<!-- 最小连接池数量 -->* <**property name="minIdle" value="1"** />  *<!-- 获取连接时最大等待时间，单位毫秒 -->* <**property name="maxWait" value="60000"** />  *<!-- 下面是bean中的其他可配置的属性，具体可参见druid官方文档。 -->* <**property name="timeBetweenEvictionRunsMillis" value="60000"** />  <**property name="minEvictableIdleTimeMillis" value="300000"** />   <**property name="testWhileIdle" value="true"** />  <**property name="testOnBorrow" value="false"** />  <**property name="testOnReturn" value="false"** />   <**property name="poolPreparedStatements" value="true"** />  <**property name="maxOpenPreparedStatements" value="20"** />   <**property name="asyncInit" value="true"** />  </**bean**>   <**bean id="personDao" class="com.sd.dao.PersonDaoImpl"**>  <**constructor-arg index="0" ref="dataSource"** />  </**bean**> </**beans**> |

上述bean中的一些配置时参考druid官当示例来的，总之都是装配DruidDataSource的属性。具体可参考官方文档。值得说明的是，属性中并没有配置驱动类，即driverClassName，这是因为官方文档说“这一项可配可不配，如果不配置druid会根据url自动识别dbType，然后选择相应的driverClassName”。因此上述就没有配置。

这样，Spring中就配置好了数据源，可以测试运行一下能否拿到数据源：

|  |
| --- |
| **public static void** main(String[] args) **throws** SQLException {  ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  *// 根据dataSource这个id拿到数据源对象* DataSource ds = (DataSource) ac.getBean(**"dataSource"**);  Connection connection = ds.getConnection();  System.***out***.println(connection);  ac.close(); } |

程序运行正常，说明配置的数据源没有问题。上述代码中还是使用了“面向接口编程”，因为使用DataSource接口接收了数据源对象，根本不需要关注使用了什么样的数据库连接池获得数据源的，使用其他的数据库连接池也是一样的。

## 1.2 使用PropertyPlaceholderConfigurer

上述我们将数据库连接信息配置在了Spring的XML中。实际应用中，XML中的内容比较复杂，不便改动，因此有时会将数据库连接信息单独放在一个配置文件中（例如src目录下的db.properties文件中）：

|  |
| --- |
| **driverClassName**=**com.mysql.jdbc.Driver username**=**root password**=**123qwe!@# url**=**jdbc:mysql://114.55.86.230:3306/mytest connectionProperties**=**useUnicode=false;characterEncoding=UTF8;useSSL=false** |

这样看起来十分直观，改动时也方便。

那么Spring中如何使用上述配置文件中的键值信息呢？这就需要在Spring的XML文件中配置一个PropertyPlaceholderConfigurer的bean，其中的locations属性可指定要引用的配置文件。这样，在Spring的XML中就能用“${配置文件中的键}”的方式来引用配置文件中对应的值信息了（即Spring中用占位符代替）。例如：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!-- 配置PropertyPlaceholderConfigurer这个bean -->* <**bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"**>  *<!-- 引入配置文件 -->* <**property name="locations" value="classpath:db.properties"**/>  </**bean**>  *<!-- 配置DataSource -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close"**>  *<!-- 其中就可引用配置文件的信息 -->* <**property name="url" value="${url}"** />  <**property name="connectionProperties" value="${connectionProperties}"** />  <**property name="username" value="${username}"** />  <**property name="password" value="${password}"** />  *<!-- 其他......-->*  </**bean**> </**beans**> |

配置文件的解析工作就是由PropertyPlaceholderConfigurer完成的。上述中，PropertyPlaceholderConfigurer这个bean没有配置id，这是因为以后根本不再需要使用这个bean，因此无需配置id。

引入配置文件还可直接使用“context:property-placeholder”标签，例如（要引用context名称空间）：

|  |
| --- |
| *<!-- 导入外部的properties文件 -->* <**context:property-placeholder location="classpath:db.properties"** /> |

## 1.3 基于JDBC的操作

接下来，就可进行JDBC编程了。简单示例如下：

（1）创建数据库表：

|  |
| --- |
| **create table** person(**id varchar**, **name varchar**); |

（2）Person实体类：

|  |
| --- |
| **package** com.sd.domain;  **import** java.io.Serializable;  **public class** Person **implements** Serializable {  **private** String **id**;  **private** String **name**;    *// getter setter* } |

（3）PersonDao接口：

|  |
| --- |
| **package** com.sd.dao;  **import** com.sd.domain.Person;  **public interface** PersonDao {  **void** insert(Person person); } |

（4）PersonDaoImpl实现（这里提供构造方法注入DataSource，这样Spring就能通过该方式注入。当然提供setter方法也行）：

|  |
| --- |
| **package** com.sd.dao;  **import** com.sd.domain.Person;  **import** javax.sql.DataSource; **import** java.sql.Connection; **import** java.sql.PreparedStatement; **import** java.sql.SQLException;  **public class** PersonDaoImpl **implements** PersonDao {   **private** DataSource **ds**;   *// 注入数据源* **public** PersonDaoImpl(DataSource ds) {  **this**.**ds** = ds;  }   @Override  **public void** insert(Person person) {  Connection connection = **null**;  PreparedStatement statement = **null**;  **try** {  Connection conn = **ds**.getConnection();  statement = conn.prepareStatement(**"insert into person values(?, ?)"**);  statement.setString(1, person.getId());  statement.setString(2, person.getName());  statement.executeUpdate();  } **catch** (SQLException ex) {  ex.printStackTrace();  } **finally** {  **if** (statement != **null**) {  **try** {  statement.close();  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }  **if** (connection != **null**) {  **try** {  connection.close();  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }  }  } } |

（5）接下来，applicationContext.xml中的配置是这样的：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!-- 配置DataSource -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close"**>  *<!-- 这其中属性的配置和之前相同...... -->*  </**bean**>  *<!-- 装配PersonDaoImpl -->* <**bean id="personDao" class="com.sd.dao.PersonDaoImpl"**>  *<!-- 注入dataSource -->* <**constructor-arg index="0" ref="dataSource"** />  </**bean**> </**beans**> |

（6）这样，基本的步骤就做好了，主类直接拿到personDao测试是否能插入数据到数据库中即可。代码如下：

|  |
| --- |
| **public static void** main(String[] args) **throws** SQLException {  ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  PersonDao personDao = (PersonDao) ac.getBean(**"personDao"**);  Person person = **new** Person();  person.setId(**"123"**);  person.setName(**"张三"**);  personDao.insert(person);  ac.close(); } |

程序最终成功运行并插入了数据。这是最基本的使用，读者也可自行再集成DbUtils的使用（提示：这时把QueryRunner加入到bean，PersonDaoImpl中提供构造方法以注入QueryRunner即可）。

从案例也可看出，自己写的代码也是比较容易就能集成到Spring中的。Spring就是这么神奇，编写代码时基本不需要考虑Spring是怎么运行的，也不需要关注Spring框架中的API，只要写好基本的功能后，能够把bean加入到Spring容器中，注入相应的属性，基本上就交由Spring管理了。

## 1.4 使用JdbcTemplate

Spring本身就提供了类似DbUtils的操作，也是进行了封装，毕竟使用上述单纯的Connection操作太麻烦了。因为JDBC编程实际上是符合“模板模式”的。模板模式其实就是“静态代码 + 静态参数”，即代码都是固定的，不同的只是参数，DbUtils就是这样封装的。

那么Spring本身提供了什么类用于简化JDBC操作呢？

（1）Spring提供了一个JdbcTemplate类用于简化JDBC操作。用法和DBUtils十分类似，也是调用update、query等方法。

JdbcTemplate中提供了构造方法：JdbcTemplate(DataSource ds)，且JdbcTemplate的父类也提供了dataSource属性及其setter方法。因此，JdbcTemplate有这两种方式来注入dataSource数据源属性。

（2）Spring提供了JdbcDaoSupport类，其中提供了获得连接等方法。如果要控制事务，可从中拿到连接。JdbcDaoSupport中可注入dataSource，也可注入jdbcTemplate再间接注入dataSource。

一般，使用时，自己写的DAO类继承JdbcDaoSupport类，这样自己的DAO中就能注入dataSource了 。然后自己的DAO中通过getJdbcTemplate()方法拿到JdbcTemplate对象进行JDBC操作。说明：利用JdbcDaoSupport类，只需要注入dataSource，不需要再注入jdbcTemplate属性，因为setDataSource()方法会主动根据注入的数据源创建JdbcTemplate对象，并给jdbcTemplate属性赋值（可看源码），因此我们可直接这样使用。

我们这里基于1.3节的示例用JdbcTemplate继续改进。

（1）PersonDao：

|  |
| --- |
| **package** com.sd.dao;  **import** com.sd.domain.Person; **import** java.util.List;  **public interface** PersonDao {  *// 添加人员* **void** add(Person person);  *// 查询全部* List<Person> queryAll();  *// 按id查询* Person findById(String id); } |

（2）PersonDaoImpl：

|  |
| --- |
| **package** com.sd.dao;  **import** com.sd.domain.Person; **import** org.springframework.jdbc.core.RowMapper; **import** org.springframework.jdbc.core.support.JdbcDaoSupport;  **import** java.sql.ResultSet; **import** java.sql.SQLException; **import** java.util.List;  **public class** PersonDaoImpl **extends** JdbcDaoSupport **implements** PersonDao {   @Override  **public void** add(Person person) {  getJdbcTemplate().update(**"INSERT INTO person VALUES(?,?)"**,  person.getId(),  person.getName());  }   @Override  **public** List<Person> queryAll() {  **return** getJdbcTemplate().query(**"SELECT id,name FROM person"**, **new** RowMapper<Person>() {  @Override  **public** Person mapRow(ResultSet rs, **int** i) **throws** SQLException {  *// 这里的rs是当前行的结果集。index就是当前行  // 因此处理时无需调用rs.next()。直接处理好一行数据即可* Person person = **new** Person();  person.setId(rs.getString(**"id"**));  person.setName(rs.getString(**"name"**));  **return** person;  }  });  }   @Override  **public** Person findById(String id) {  List<Person> personList = getJdbcTemplate().query(**"SELECT id,name FROM person WHERE id=?"**,  **new** RowMapper<Person>() {  @Override  **public** Person mapRow(ResultSet rs, **int** i) **throws** SQLException {  Person person = **new** Person();  person.setId(rs.getString(**"id"**));  person.setName(rs.getString(**"name"**));  **return** person;  }  },  id  );  **return** personList.size() == 0 ? **null** : personList.get(0);  } } |

可以看出，JDBCTemplate的用法和DBUtils用法很相似，增删改都是用upate()方法，查询用query()方法。只是JdbcTemplate中，处理查询的数据时使用的是RowMapper接口，从mapRow()方法中得到的是一行的数据，只需要处理一行数据即可，返回值是List集合。因此当返回单个对象时，就需要判断下size()是否为0。由于这样的设计，mapRow()的针对同一个对象的处理代码基本是一致的（如上面的findById和queryAll方法），当然也可以直接把重复代码写在内部类中重复利用。

（3）applicationContext.xml文件。分析：这时只要给PersonDaoImpl注入dataSource即可。

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!-- 配置PropertyPlaceholderConfigurer这个bean -->* <**bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"**>  *<!-- 引入配置文件 -->* <**property name="locations" value="classpath:db.properties"**/>  </**bean**>  *<!-- 配置DataSource -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close"**>  *<!-- 属性...... -->*  </**bean**>   *<!-- 配置personDao -->* <**bean id="personDao" class="com.sd.dao.PersonDaoImpl"**>  *<!-- 注入dataSource -->* <**property name="dataSource" ref="dataSource"** />  </**bean**> </**beans**> |

（4）测试：

|  |
| --- |
| **public static void** main(String[] args) **throws** SQLException {  ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  PersonDao personDao = (PersonDao) ac.getBean(**"personDao"**);  List<Person> people = personDao.queryAll();  System.***out***.println(people);  Person person = personDao.findById(**"123"**);  System.***out***.println(person.getName());  ac.close(); } |

# 2. Spring的事物管理

## 2.1 关于事物管理

从开发者角度来说，事物管理分为“编程式事物管理”和“声明式事物管理”。解释如下：

（1）编程式事物管理：

之前我们使用的都是编程式事物管理。编程式事物管理就是在代码中手动处理事物。比如在JDBC中可通过设置connection.setAutoCommit(false)来手动控制事物。在Hibernate中，我们调用的beginTransaction()和commit()方法就是手动处理事物。

编程式事务管理可以对事务进行详细地控制，是细粒度的事物控制，比较灵活。缺点是开发时比较繁琐。

（2）声明式事物管理：

声明式事物管理就是对要控制的事物进行“声明”，比如可将事物控制配置在文件中（或者通过注解的方式配置），可按需更改或者移除。这种事物是粗粒度的事物控制。

前面说过，在三层结构中，事务的控制应该放到Service层的方法中。因为一项业务的操作可能涉及多个DAO对象，若此项业务成功执行，则要求其中所有的DAO操作都执行成功，否则应该都操作失败。所以若采用声明式事物管理，则要将事物声明配置在Service层中。

Spring中就提供了声明式事物管理，其核心是基于AOP给方法添加事物控制。Spring中用声明式事物管理器类来实现此功能。比如JDBC使用DataSourceTransactionManager事物管理器；Hibernate使用HibernateTransactionManager事物管理器。

如果不用Spring的声明式事务管理，如果想在Service层实现事务，就需要像之前一样，使用ThreadLocal类来进行处理，是比较麻烦的。而使用Spring后，能让开发者不再关注事务处理，只要告诉Spring容器某些方法采用什么样的事务处理策略，那么Spring就能自动完成事务在Service方法上的处理，当然具体的CRUD操作还是由开发者来做的。

实际上使用Spring与JDBC等数据库操作进行整合，目的并不是为了使用JdbcTemplate这些辅助类，而是为了使用Spring的声明式事务管理。正是因为Spring的声明式事务处理，才把AOP发挥到极致。现在，Spring的声明式事务处理是大部分企业中应用处理事务的一个平台。

Spring中，事务其实就是一个切面，其中有开启事务、提交事务、回滚事务等方法，这是Spring已经提供好的，而Service中的方法就是目标方法。最终，Spring会利用AOP形成代理方法处理事务。下面的案例进行具体的讲解。

## 2.2 Spring声明式事物管理案例（JDBC + XML配置）

基于第一章的例子，先添加Service层，因为以后会在此层实现Spring的声明式事物管理。

（1）新建service包，其中新建PersonService接口。提供add2Person方法。该业务方法会在数据库表中添加两个人员信息：

|  |
| --- |
| **package** com.sd.service;  **public interface** PersonService {  **void** add2Person(); } |

（2）PersonServiceImpl的实现：

|  |
| --- |
| **package** com.sd.service;  **import** com.sd.dao.PersonDao; **import** com.sd.domain.Person; **import** java.util.UUID;  **public class** PersonServiceImpl **implements** PersonService {   *// 依赖UserDao。面向接口编程* **private** PersonDao **personDao**;   *// // setter方法，以供注入* **public void** setPersonDao(PersonDao personDao) {  **this**.**personDao** = personDao;  }   @Override  **public void** add2Person() {  *// 保存第一个学生* Person person1 = **new** Person();  person1.setId(UUID.*randomUUID*().toString());  person1.setName(**"张三"**);  **personDao**.add(person1);   *// 中间故意让程序出错* **int** i = 9 / 0;   *// 保存第二个学生* Person person2 = **new** Person();  person2.setId(UUID.*randomUUID*().toString());  person2.setName(**"张三"**);  **personDao**.add(person2);  } } |

（3）如果不考虑事务，则XML配置为（在原来的基础上配置service的bean，并引用dao）：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!-- 配置PropertyPlaceholderConfigurer -->* <**bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"**>  *<!-- 引入配置文件 -->* <**property name="locations" value="classpath:db.properties"**/>  </**bean**>  *<!-- 配置DataSource -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close"**>  *<!-- 属性...... -->*  </**bean**>  *<!-- 配置personDao -->* <**bean id="personDao" class="com.sd.dao.PersonDaoImpl"**>  *<!-- 注入dataSource -->* <**property name="dataSource" ref="dataSource"** />  </**bean**>  *<!-- 配置personService的bean -->* <**bean id="personService" class="com.sd.service.PersonServiceImpl"**>  *<!-- 注入personDao -->* <**property name="personDao" ref="personDao"** />  </**bean**> </**beans**> |

（4）测试方法为：

|  |
| --- |
| **public static void** main(String[] args) **throws** SQLException {  ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  PersonService personService = (PersonService) ac.getBean(**"personService"**);  personService.add2Person();  ac.close(); } |

运行程序发现达不到控制事务的目的，数据库中只保存了“张三”。

那么，如果要考虑事务，要求Service中每个方法中的操作都在同一个事务中，怎么做呢？即新增的两个人员要么全部插入成功，要么全部失败，把事务的控制放在Service层。

此时Spring的XML中要这样配置：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd"**>  <**bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"**>  <**property name="locations" value="classpath:db.properties"**/>  </**bean**>  *<!-- 配置DataSource -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close"**>  *<!-- 数据源属性配置...... -->*  </**bean**>  *<!-- 配置personDao -->* <**bean id="personDao" class="com.sd.dao.PersonDaoImpl"**>  *<!-- 注入dataSource -->* <**property name="dataSource" ref="dataSource"** />  </**bean**>  *<!-- 配置personService -->* <**bean id="personService" class="com.sd.service.PersonServiceImpl"**>  *<!-- 注入personDao -->* <**property name="personDao" ref="personDao"** />  </**bean**>   *<!-- 上述的配置和之前是相同的。下面进行声明式事务的配置 -->   <!-- 1. 配置事务管理器类，这里配置的是JDBC的事务管理器 -->* <**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  *<!-- 其中引用数据源 -->* <**property name="dataSource" ref="dataSource"** />  </**bean**>  *<!-- 2. 进行事务的声明，即配置Spring如何管理事物 -->  <!-- transaction-manager引用上面的事务管理器-->* <**tx:advice id="txAdvice" transaction-manager="transactionManager"**>  <**tx:attributes**>  *<!-- attributes节点中配置详细。 -->  <!--  tx:method节点中配置某些方法应用事务时执行的具体策略  name指定对哪些方法应用事物，这里运用了通配符\*  isolation表示事务的隔离级别，这里是默认的；  propagation表示事务的传输级别，这里也是默认的；  read-only属性用于设置是否是只读事务。  -->* <**tx:method name="add\*" isolation="DEFAULT" propagation="REQUIRED" read-only="false"** />  <**tx:method name="save\*" isolation="DEFAULT" propagation="REQUIRED" read-only="false"** />  <**tx:method name="update\*" isolation="DEFAULT" propagation="REQUIRED" read-only="false"** />  <**tx:method name="delete\*" isolation="DEFAULT" propagation="REQUIRED" read-only="false"** />  *<!-- 对于查询操作，可设置只读为true，只读事务，不允许写 -->* <**tx:method name="query\*" isolation="DEFAULT" propagation="REQUIRED" read-only="true"** />  </**tx:attributes**>  </**tx:advice**>  *<!-- 3. 进行AOP的配置。 -->* <**aop:config**>  *<!-- 切入点表达式，定位到Service层中的所有方法 -->* <**aop:pointcut id="pt" expression="execution(\* com.sd.service.PersonServiceImpl.\*(..))"** />  *<!-- 指定事务的声明和切入点之间的关联配置。这样Spring才能为匹配的切入点方法织入事务控制的相关代码 -->* <**aop:advisor advice-ref="txAdvice" pointcut-ref="pt"**/>  </**aop:config**> </**beans**> |

这时，再运行主类，事务就会起作用，数据库不会增加数据。若把int a = 9 / 0; 这行代码去掉，则程序会成功添加两个人员的数据。

注意：tx名称空间引用的是“xmlns:tx="http://www.springframework.org/schema/tx"”，因为还有一个名称空间也叫tx，但是不是用来配置事物的，用错了则程序无法正常运行出错。

配置的说明：

（1）在AOP配置切面之前，先配置了JDBC的事务管理器类和事务的声明。如果采用Hibernate，则Hibernate的事务管理器和JDBC的是不同的。事务的声明就是指明哪些方法应用什么样的事务策略，例如隔离级别、事务传输级别、事务是读写事务还是只读事务等。

（2）在AOP的配置中，先配置了切入点表达式，后面配置了切入点表达式与事务的声明的联系，这样才会匹配表达式应用事务。这里有个问题要说明，为什么没有配置事务的切面类，也没有在aop:config中配置事务的通知呢？原因是Spring的声明式事务的初衷就是让开发者无需关注事务，如果让开发者配置通知，则开发者必然要知道切面类运行的具体原理，需要知道前置通知配置成什么方法才能开启事务、后置通知配置成什么方法才能提交事务、异常通知配置成什么方法才能回滚事务，这意味着开发者要对Spring的内部运行机制非常熟悉，而这并不是声明式事务想要的。

关于更具体的，下面再说。

## 2.3 声明式事务配置的说明

一、Spring声明式事务管理的框架

Spring中，用于管理事务的一个顶级接口是PlatformTransactionManager接口，其中提供了如下方法：

|  |
| --- |
| **public interface** PlatformTransactionManager {  TransactionStatus getTransaction(@Nullable TransactionDefinition var1) **throws** TransactionException;   **void** commit(TransactionStatus var1) **throws** TransactionException;   **void** rollback(TransactionStatus var1) **throws** TransactionException; } |

分别用于获取事务、提交事务和回滚事务。Spring中，还有一个名叫AbstractPlatformTransactionManager的抽象类实现了这个接口，该接口中有些方法是抽象的，而有些方法是有实现的。最终，具体的事务管理器类（例如DataSourceTransactionManager和HibernateTransactionManager等）继承了AbstractPlatformTransactionManager抽象类，实现了其中的抽象方法，也就是最终执行获取事务、提交和回滚事务时，是由具体的事务管理器来实现的。

为什么要这样设计呢？因为抽象类中实现了一些通用的方法，但是一些和具体JDBC框架相关的事务处理，还是要交给特定的事务管理器去做，如果要自己实现事务管理器，只要继承AbstractPlatformTransactionManager抽象类，实现抽象方法即可。这其中也体现了接口和抽象类的作用，接口用于面向接口编程，抽象类实现了一些通用的方法，不必事务管理器重复实现。

因此，我们在Spring中注册事务管理器类就是为了让Spring知道使用哪个具体的事务管理器类处理最终的事务提交等操作。

二、事务声明中注意的几点

tx:method中，用于对不同的方法指定不同的事务策略，其中的属性可指定如下事项：

（1）事务的隔离级别

使用isolation属性可指定事务的隔离级别，即防止脏读、幻读和不可重复读等，可设置如下的值：

DEFAULT：默认的隔离级别

READ\_UNCOMMITTED：脏读、不可重复读、虚读都有可能发生

READ\_COMMITTED：防止脏读发生，不可重复读、虚读都有可能发生

REPEATABLE\_READ：防止脏读、不可重复读的发生，虚读有可能发生

SERIALIZABLE：防止脏读、不可重复读、虚读的发生

开发时使用默认的DEFAULT即可。

（2）事务的传输属性

使用propagation属性可指定事务的传播属性。事务的传播属性就是用来解决事务的嵌套问题的。

什么是事务的嵌套？例如Service中的A、B、C方法都在一个独立的事务中，但A方法调用了Service中的B和C方法，这样就是A方法的事务中嵌套了B和C两个事务，此时就是事务的嵌套。

常用如下事务的传播属性值来处理事务的嵌套：

REQUIRED：这是默认值。把嵌套的事务加入到已有的事务中，即把方法当做一个整体进行事务处理。例如A中的B和C会被处理在同一个事务中，要么都成功，要么都失败。

REQUIRES\_NEW：嵌套的每个事务都将开启一个新的事务，也就是说B和C的执行不干扰，B和C都有可能成功。

一般开发中使用默认值即可。

由于isolation和propagation基本都使用默认值，因此这两个属性也可不配置。

（3）只读事务和读写事务

read-only属性用于控制事务是只读事务还是读写事务。

如果read-only为false（默认值），表示该事务是读写事务，在该事务中可进行写数据库的操作；

如果read-only为true，表示该事务是只读事务。只读事务就是本事务只进行数据库查询操作，在查询时会进行加锁，其他事务在此期间不能对数据库修改。简单的说，只读事务用于查询，不允许修改数据库数据。在开发时应该控制事务的这个属性，查询时应该设置read-only为true。

## 2.4 注解配置声明式事物管理（JDBC + 注解）

Spring的声明式事务也可采用注解来配置，开发者使用时只需要在Service的方法上使用@Transactional注解即可控制Service方法的事务。

现在我们把上述案例改为用注解配置。注解可简化XML的配置，也能使Service方法的事务控制看上去更加直观。但是并不是说使用了注解，就意味着把所有bean的配置都写在注解中，很多情况下都是结合XML使用。

在本例中，DAO、Service类及其依赖的属性可使用@Repository、@Service和@Resource注解配置，但是DAO由于继承了JdbcDaoSupport，需要注入dataSource，使用注解是不好配置的。因此这里将DAO改为不再继承JdbcDaoSupport类，而是直接在DAO中提供jdbcTemplate属性，以便使用注解注入。并不非要用JdncDaoSupport类，因为采用了Spring的声明式事务管理后，事务将由Spring管理，不再需要使用JdncDaoSupport类获得连接来控制事务了，因此这里直接使用jdbcTemplate没有任何问题。

要想DAO中能注入jdbcTemplate属性，则Spring容器中要有jdbcTemplate这个bean，因此这个jdbcTemplate还是要用XML来加入到bean。同样，XML中还要加入数据库连接池的bean，因为这两个bean无法通过注解配置进来，这就是XML要配合注解使用。另外，XML中还要配置关于使用事务注解等的相关内容，因此applicationContext.xml中内容现在如下：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd"**>  *<!-- 引入db.properties配置文件 -->* <**bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"**>  <**property name="locations" value="classpath:db.properties"**/>  </**bean**>  *<!-- 配置DataSource -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close"**>  *<!-- 数据源的属性...... -->*  </**bean**>  *<!-- 这里虽然不再需要配置Service和DAO了，但需要配置JdbcTemplate，DAO需要使用 -->* <**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  *<!-- 注入dataSource属性 -->* <**property name="dataSource" ref="dataSource"** />  </**bean**>   *<!-- 开启注解扫描 -->* <**context:component-scan base-package="com.sd"** />  *<!-- 以下是用注解进行声明式事务管理所需的XML配置 -->   <!-- 1. 还是要先配置事务管理器 -->* <**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  *<!-- 引用数据源 -->* <**property name="dataSource" ref="dataSource"** />  </**bean**>  *<!-- 2. 开启自动创建声明式事务，这样事务的注解才有效 -->* <**tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true"** />  *<!-- 这样，事务的声明和AOP配置就不需要了，改用注解解决。 -->* </**beans**> |

按照之前的说法，则PersonDaoImpl改为如下：

|  |
| --- |
| **package** com.sd.dao;  *// 使用Repository注解把类加入到bean* @Repository(**"personDao"**) **public class** PersonDaoImpl **implements** PersonDao {   @Resource(name = **"jdbcTemplate"**)  **private** JdbcTemplate **jdbcTemplate**;   *// 提供jdbcTemplate的getter和setter* **public** JdbcTemplate getJdbcTemplate() {  **return jdbcTemplate**;  }   **public void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {  **this**.**jdbcTemplate** = jdbcTemplate;  }   @Override  **public void** add(Person person) {  getJdbcTemplate().update(**"INSERT INTO person VALUES(?,?)"**,  person.getId(),  person.getName());  }  *// 下面方法和之前是一样的，无改动......* } |

然后把Service加入bean，且使用@Transactional控制事务即可：

|  |
| --- |
| **package** com.sd.service;  **import** com.sd.dao.PersonDao; **import** com.sd.domain.Person; **import** org.springframework.stereotype.Service; **import** org.springframework.transaction.annotation.Transactional; **import** javax.annotation.Resource; **import** java.util.UUID;  *// 将本service加入到bean* @Service(**"personService"**)  **public class** PersonServiceImpl **implements** PersonService {   *// 依赖UserDao。面向接口编程* @Resource(name = **"personDao"**)  **private** PersonDao **personDao**;   *// 无需提供getter和setter方法即可注入personDao* @Override  @Transactional(readOnly = **false**) *// 事务控制* **public void** add2Person() {  *// 保存第一个学生* Person person1 = **new** Person();  person1.setId(UUID.*randomUUID*().toString());  person1.setName(**"张三"**);  **personDao**.add(person1);   *// 中间故意让程序出错* **int** i = 9 / 0;   *// 保存第二个学生* Person person2 = **new** Person();  person2.setId(UUID.*randomUUID*().toString());  person2.setName(**"张三"**);  **personDao**.add(person2);  } } |

最后运行主类测试（主类无改动）。

@Transactional中可使用readOnly、isolation、propagation等属性，意义和我们之前讲的是一样的。另外，@Transactional注解定义在方法上时，是对当前方法应用事物，而定义到类上时，是对类中所有的方法都应用此事物。

说明：

上述代码中，如果Service层方法不使用@Transactional注解，则不会应用事务，但是JDBC默认是自动提交事务的，因此执行代码会默认更改数据库（如果代码没有错误的话。本例的代码有错误，运行程序后，数据库表中也会添加一条出错之前的“张三”）。

下面我们会讲Spring与Hibernate的集成，这时就不一样了。因为Hibernate中事务默认情况下是手动提交的，所以如果不加@Transactional的配置，则方法不会应用事务，这样执行程序后，无论代码执行是否出错，都不会对数据库中的数据产生影响，因为事务不会提交到数据库。这点可在下节中自行测试。

## 2.5 Spring与Hibernate的集成（注解）

这里讲Spring与Hibernate的集成，直接用注解和XML结合的形式。

首先，Spring的orm包中自带了对Hibernate的支持，类似于对JDBC的支持，该包中也提供了HibernateTemplate、HibernateDaoSupport类等，它们提供了sessionFactory属性以供注入。与JDBC中需要注入dataSource不同，Hibernate需要的就是注入sessionFactory即可操作数据库。同样，HibernateTemplate中也提供了很多使用Hibernate操作数据库的方法。

接下来我们进行Spring与Hibernate的整合配置。其中DAO中，仍然使用注入hibernateTemplate的方式，而不是使用继承HibernateDaoSupport的方式。以下是步骤：

（1）引入Hibernate相关jar包。

（2）在com.sd.domain中创建hibernate映射文件Person.hbm.xml（对应的Person类还是之前的）：

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"** *?>* **<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>*** <**hibernate-mapping package="com.sd.domain"**>  <**class name="Person" table="person"**>  <**id name="id" column="id"**>  <**generator class="uuid"** />  </**id**>  <**property name="name" column="name"** />  </**class**> </**hibernate-mapping**> |

（3）编写applicationContext.xml文件。

分析：在该文件中，除了要配置dataSource，还要配置hibernate需要的sessionFactory，因为DAO操作（包括其中的hibernateTemplate）都是需要sessionFactory的。

sessionFactory是要依赖dataSource的，在这个新版本Spring中，只推荐在注册sessionFactory时引用dataSource，并在Spring的XML中维护Hibernate的相关配置！这样就不再需要Hibernate的主配置文件了！

随后，注册HibernateTemplate时再引用sessionFactory，并且在Hibernate事务管理器中引用sessionFactory，而不是dataSource。

最终，与Hibernate整合时，applicationContext.xml一般配置如下（开启注解）：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd"**>  *<!-- 引入db.properties配置文件 -->* <**bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"**>  <**property name="locations" value="classpath:db.properties"**/>  </**bean**>  *<!-- 配置DataSource(使用druid) -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close"**>  *<!-- 数据库连接池属性...... -->*  </**bean**>   *<!-- 配置sessionFactory -->* <**bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"**>  *<!-- 其中引用dataSource -->* <**property name="dataSource" ref="dataSource"** />  *<!-- 由于不再使用hibernate.cfg.xml主配置文件，因此一些Hibernate配置信息放到Spring中配置 -->  <!-- 这里就利用其中的属性进行配置 -->  <!-- 1. 指明映射文件所在的路径 -->* <**property name="mappingDirectoryLocations"**>  <**list**>  *<!-- list可配置多个，这里就是在domain目录下 -->* <**value**>classpath:com/sd/domain</**value**>  </**list**>  </**property**>  *<!-- 2. 在hibernateProperties中配置hibernate的其他属性 -->* <**property name="hibernateProperties"**>  <**props**>  *<!-- 方言 -->* <**prop key="hibernate.dialect"**>org.hibernate.dialect.MySQL5Dialect</**prop**>  *<!-- 显示SQL -->* <**prop key="hibernate.show\_sql"**>true</**prop**>  *<!-- 格式化SQL -->* <**prop key="hibernate.format\_sql"**>true</**prop**>  *<!-- 自动建表 -->* <**prop key="hibernate.hbm2ddl.auto"**>update</**prop**>  *<!-- 其他等等...... -->* </**props**>  </**property**>  </**bean**>  *<!-- 配置hibernateTemplate -->* <**bean id="hibernateTemplate" class="org.springframework.orm.hibernate5.HibernateTemplate"**>  *<!-- 注入sessionFactory属性 -->* <**property name="sessionFactory" ref="sessionFactory"** />  </**bean**>  *<!-- 开启注解扫描 -->* <**context:component-scan base-package="com.sd"** />  *<!-- 以下是用注解进行声明式事务管理所需的XML配置 -->   <!-- 这里就是配置hibernate事务管理器 -->* <**bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager"**>  *<!-- 这里引用sessionfactory,而不是数据源dataSource -->* <**property name="sessionFactory" ref="sessionFactory"** />  </**bean**>  *<!-- 开启自动创建声明式事务 -->* <**tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true"** /> </**beans**> |

上述中，使用LocalSessionFactoryBean类配置了sessionFactory，不能直接使用Hibernate的实现类SessionFactoryImpl，这是因为Hibernate本身的SessionFactoryImpl类没有属性可供注入dataSource，因此使用Spring的orm包中提供的对Hibernate支持的类LocalSessionFactoryBean。

装配LocalSessionFactoryBean时，还设置了映射文件目录、Hibernate相关设置等，以此代替了hibernate.cfg.xml的作用。mappingDirectoryLocations属性指明了映射文件的所在目录，不用像原来hibernate.cfg.xml中要一一指定每个映射文件了。另外需要注意的是，在Spring中配置Hibernate属性时，最好都加上“hibernate”前缀，例如“hibernate.show\_sql”，否则配置可能不生效！！！

（4）PersonDaoImpl改为如下：

|  |
| --- |
| **package** com.sd.dao;  **import** com.sd.domain.Person; **import** org.springframework.orm.hibernate5.HibernateTemplate; **import** org.springframework.stereotype.Repository; **import** javax.annotation.Resource; **import** java.util.List;  *// 使用Repository注解把类加入到bean* @Repository(**"personDao"**) **public class** PersonDaoImpl **implements** PersonDao {   *// 注入hibernateTemplate* @Resource(name = **"hibernateTemplate"**)  **private** HibernateTemplate **hibernateTemplate**;   @Override  **public void** add(Person person) {  **hibernateTemplate**.save(person); *// 添加* }   @Override  **public** List<Person> queryAll() {  **return hibernateTemplate**.loadAll(Person.**class**); *// 查询全部* }   @Override  **public** Person findById(String id) {  **return hibernateTemplate**.get(Person.**class**, id); *// 按主键查询* } } |

上述代码中就使用了hibernateTemplate提供的方法。

写到这里就结束了，因为其余的代码根本不需要改变，Service还是像之前一样使用@Transactional进行事务的处理，注意把Service层类加入bean即可。运行程序就会采用Hibernate框架操作。

实际中，也可能不使用hibernateTemplate，而还是直接使用sessionFactory，这样还是使用Hibernate中的API，而不是Spring的API，因为使用Spring的API不利于解耦，意味着DAO要依赖Spring。

如果直接使用sessionFactory，则DAO代码更改如下（applicationContext.xml中也不需要配置jdbcTemplate这个bean了）：

|  |
| --- |
| **package** com.sd.dao;  **import** com.sd.domain.Person; **import** org.hibernate.SessionFactory; **import** org.springframework.stereotype.Repository; **import** javax.annotation.Resource; **import** java.util.List;  @Repository(**"personDao"**) **public class** PersonDaoImpl **implements** PersonDao {   *// 为了解耦，这里不使用hibernateTemplate，而是使用sessionFactory  // sessionFactory已经在Spring XML中配置过了，直接注入即可* @Resource(name = **"sessionFactory"**)  **private** SessionFactory **sessionFactory**;   @Override  **public void** add(Person person) {  **sessionFactory**.getCurrentSession().save(person);  }   @Override  **public** List<Person> queryAll() {  **return sessionFactory**.getCurrentSession()  .createQuery(**"from Person"**)  .list();  }   @Override  **public** Person findById(String id) {  **return sessionFactory**.getCurrentSession().get(Person.**class**, id);  } } |

其他代码不变，运行主类测试成功。

关于上述的代码，需要说明以下几点：

（1）由于使用了Spring管理事务，因此代码中无需进行事务处理，无需调用Hibernate中与Transaction相关的API。

（2）上述代码中使用了getCurrentSession()方式获得当前session进行操作，但是我们的applicationContext.xml中，sessionFactory这个bean中的hibernateProperties属性并没有配置“current\_session\_context\_class”为“thread”这个设置，即：

|  |
| --- |
| <**prop key="hibernate.current\_session\_context\_class"**>thread</**prop**> |

为什么也能在DAO中使用sessionFactory.getCurrentSession()呢？相反，如果进行了上述的配置，Hibernate操作还会报错：

|  |
| --- |
| HibernateException: save is not valid without active transaction |

即无法激活事务。这是怎么一回事呢？

原来，在Hibernate整合到Spring后，Spring就会以“SpringSessionContext”这个类作为“hibernate.current\_session\_context\_class”的默认值，这是因为Spring要处理Hibernate的事务，则当前session应该是绑定到SpringSessionContext类上，由Spring去管理，而不是原来hibernate中的“thread”去管理，因此设置hibernate.current\_session\_context\_class为thread后，Spring就无法进行事务管理，就会报错。在Spring中，“SpringSessionContext”是“hibernate.current\_session\_context\_class”的默认值，可无需设置，当然也可显式设置：

|  |
| --- |
| *<!-- 线程方式得到session(基于Spring中的配置，该设置可省略) -->* <**prop key="hibernate.current\_session\_context\_class"**>org.springframework.orm.hibernate5.SpringSessionContext</**prop**> |

因此，我们在上述代码中可以直接调用getCurrentSession方法获得session对象，同样也无需进行session的关闭，这些是Spring完成了。

由此发现，应用Spring后，DAO中的代码简洁了许多，既无需处理事务，也无需关心会话的关闭。

## 2.6 HibernateCallback和回调

上述使用HibernateTemplate时，就是使用了Spring提供的封装Hibernate操作的API。但有时如果需要直接使用Hibernate本身提供的操作数据库的API（如session.createQuery、session.get等方法），怎么办呢？

Spring的HibernateTemplate中就提供了一个名叫“execute”的方法，用于执行Hibernate原生的操作，execute方法的原型是这样的：

|  |
| --- |
| **public** <T> T execute(HibernateCallback<T> action) **throws** DataAccessException |

该方法需要一个HibernateCallback类型的对象，而HibernateCallback是一个接口，提供了一个doInHibernate的方法：

|  |
| --- |
| **package** org.springframework.orm.hibernate5; @FunctionalInterface **public interface** HibernateCallback<T> {@Nullable  T doInHibernate(Session session) **throws** HibernateException; } |

也就是说，要使用execute方法，必须传递一个HibernateCallback对象，且实现其中的doInHibernate方法。其实这就是之前讲的“策略模式”。doInHibernate方法需要开发者自己编写，其中就是要执行的Hibernate操作的“原生”方法。doInHibernate传递了session对象以供使用，开发者可直接使用session进行操作，根本不用考虑如何获得session，当然使用了声明式事务处理后也无需考虑事务的处理。

其实最终框架会调用我们编写的方法，其中的泛型就是我们想返回的数据类型，如果是查询的话，只要返回查询封装好的数据即可。例如可将findById方法改写为：

|  |
| --- |
| **public** Person findById(String id) {  **return hibernateTemplate**.execute(**new** HibernateCallback<Person>() {  @Nullable  @Override  **public** Person doInHibernate(Session session) **throws** HibernateException {  **return** session.get(Person.**class**, id);  }  }); } |

其实，HibernateCallback就是一个回调，就是Spring框架预留的一个接口，由开发者实现，最终框架还会调用这部分开发者编写的代码。

回调的代码并不是开发者自己调用的，这样的回调很常见。例如在写JavaScript时，你给一个按钮注册了一个单击事件的代码，但这部分代码并不是你调用的，而是浏览器监听到事件后，调用你编写的代码。