# Spring开发笔记

# Spring的概述

## 什么是Spring

Spring是一个开源框架，Spring是于2003年兴起的一个轻量级的Java开发框架，由Rod Johnson创建。简单来说，Spring是一个分层的JavaSE/EEfull-stack(一站式)轻量级开源框架。

Spring是一个开源框架，Spring是于2003年兴起的一个轻量级的Java开发框架，由Rod Johnson在其著作Expert One-On-One J2EE Development and Design中阐述的部分理念和原型衍生而来。它是为了解决企业应用开发的复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许使用者选择使用哪一个组件，同时为J2EE应用程序开发提供集成的框架。Spring使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。**Spring的核心是控制反转(IoC)和面向切面(AOP)。简单来说，Spring是一个分层的JavaSE/EEfull-stack(一站式)轻量级开源框架**。

为什么说Spring是一个一站式的轻量级开源框架呢？EE开发可分成三层架构，针对JavaEE的三层结构，每一层Spring都提供了不同的解决技术。

* WEB层：SpringMVC
* 业务层：Spring的IoC
* 持久层：Spring的JDBCTemplate(Spring的JDBC模板，ORM模板用于整合其他的持久层框架)

从上面的简要介绍中，我们要知道Spring的核心有两部分：

* IoC：控制反转。   
  举例来说，在之前的操作中，比方说有一个类，我们想要调用类里面的方法(不是静态方法)，就要创建类的对象，使用对象调用方法实现。对于Spring来说，Spring创建对象的过程，不是在代码里面实现的，而是交给Spring来进行配置实现的。
* AOP：面向切面编程。   
  之前，讲Struts2框架的拦截器时，我们就已稍微讲了一下，在Spring学习过程中，我们会着重来讲它。但是本文并不会过多阐述它，下文再讲。

## 为什么Spring

* 方便解耦，简化开发。   
  Spring就是一个大工厂，可以将所有对象的创建和依赖关系的维护，交给Spring管理。
* AOP编程的支持   
  Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能。
* 声明式事务的支持   
  只需要通过配置就可以完成对事务的管理，而无须手动编程。
* 方便程序的测试   
  Spring对Junit4支持，可以通过注解方便的测试Spring程序。
* 方便集成各种优秀的框架   
  Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架(如：Struts2、Hibernate、MyBatis、Quartz等)的直接支持。
* 降低JavaEE API的使用难度   
  Spring对JavaEE开发中非常难用的一些API(JDBC、JavaMail、远程调用等)，都提供了封装，使这些API应用难度大大降低。

## Spring的版本

我这儿使用的Spring的版本是spring-framework-4.2.4.RELEASE。

# Spring的入门案例

## IOC的底层实现原理

IOC：Inversion of Control，控制反转。指的是对象的创建权反转(交给)给Spring，其作用是实现了程序的解耦合。也可这样解释：获取对象的方式变了。对象创建的控制权不是“使用者”，而是“框架”或者“容器”。   
用更通俗的话来说，IOC就是指对象的创建，并不是在代码中用new操作new出来的，而是通过Spring进行配置创建的。其底层实现原理是**XML配置文件+SAX解析+工厂设计模式**。   
就拿持久层(也即dao(data access object，数据访问对象)层)的开发来说，官方推荐做法是先创建一个接口，然后再创建接口对应的实现类。   
先创建一个Userdao接口

public interface UserDao {

public void add();

}

* 1
* 2
* 3

再创建Userdao接口的UserDaoImpl实现类

public class UserDaoImpl implements UserDao {

public void add() {

balabala......

}

}

* 1
* 2
* 3
* 4
* 5

接着我们在service层调用dao层，核心代码如下：

// 接口 实例变量 = new 实现类

UserDao dao = new UserDaoImpl();

dao.add();

* 1
* 2
* 3

可发现缺点：service层和dao层耦合度太高了。解决方法是使用工厂模式进行解耦合操作。   
创建一个工厂类，在工厂类中提供一个方法，返回实现类的对象。

public class Factory {

// 提供返回实现类对象的方法

public static UserDao getUserDaoImpl() {

return new UserDaoImpl();

}

}

* 1
* 2
* 3
* 4
* 5
* 6

然后在service层调用dao层的核心代码就变为：

UserDao dao = Factory.getUserDaoImpl();

dao.add();

* 1
* 2

如若这样做，会发现又产生了一个缺点：service层和工厂类又耦合了。所以使用工厂模式进行解耦合也只是一种权宜之计。下面我就来简单讲讲Spring IOC的底层实现原理：

* 配置文件中可能会有如下配置信息：

<bean id="userDaoImpl" class="cn.itcast.dao.impl.UserDaoImpl" />

* + 1
* 也是要创建一个工厂类，在工厂类中提供一个返回实现类对象的方法，但并不是直接new实现类，而是使用SAX解析配置文件，根据标签bean中的id属性值得到对应的class属性值，使用反射创建实现类对象。
* public class Factory {
* public static UserDao getUserDaoImpl() {
* // 1.使用SAX解析得到配置文件内容
* // 直接根据id值userDaoImpl得到class属性值
* String classvalue = "class属性值";
* // 2.使用反射得到对象
* Class clazz = Class.forName(classvalue);
* UserDaoImpl userDaoImpl = (UserDaoImpl)lazz.newInstance();
* return userDaoImpl;
* }

}

* + 1
  + 2
  + 3
  + 4
  + 5
  + 6
  + 7
  + 8
  + 9
  + 10
  + 11

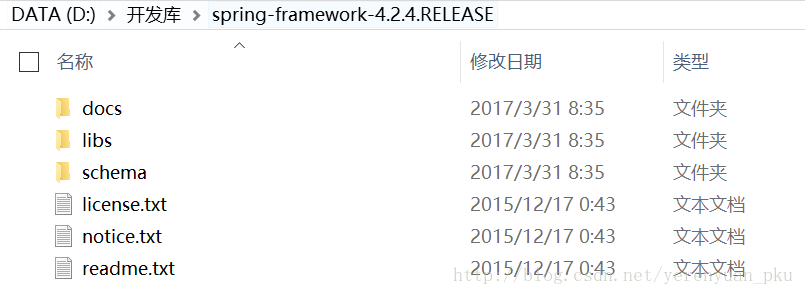
## 面向对象设计的七大原则

这里我稍微讲一下面向对象设计的七大原则，不必强记，重在理解。

1. 单一职责原则（Single Responsibility Principle）：每一个类应该专注于做一件事情。
2. 里氏替换原则（Liskov Substitution Principle）：超类存在的地方，子类是可以替换的。
3. 依赖倒置原则（Dependence Inversion Principle）：实现尽量依赖抽象，不依赖具体实现。
4. 接口隔离原则（Interface Segregation Principle）：应当为客户端提供尽可能小的单独的接口，而不是提供大的总的接口。
5. 迪米特法则（Law Of Demeter）：又叫最少知识原则，一个软件实体应当尽可能少的与其他实体发生相互作用。
6. 开闭原则（Open Close Principle）：面向扩展开放，面向修改关闭。
7. 组合/聚合复用原则（Composite/Aggregate Reuse Principle CARP）：尽量使用组合/聚合达到复用，尽量少用继承。原则： 一个类中有另一个类的对象。

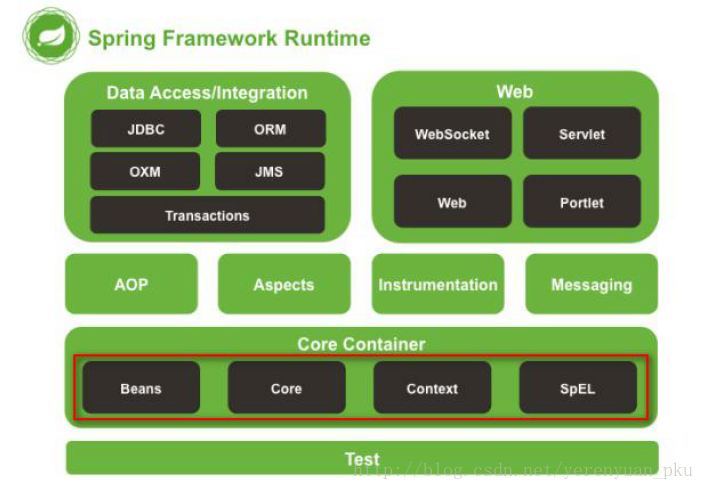
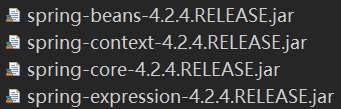
## Spring的IOC入门

### 步骤一：下载Spring的开发包

Spring的官网是[http://spring.io](http://spring.io" \t "_blank)。Spring的开发包的下载地址是<http://repo.springsource.org/libs-release-local/org/springframework/spring>，上面说过，我下载的是spring-framework-4.2.4.RELEASE。解压缩之后，可发现Spring开发包的目录结构如下：   


* docs：API和开发规范
* libs：Jar包和源码
* schema：约束

### 步骤二：创建Web项目，引入Spring的开发包

从下图可知：   
  
由于我们只是初次入门Spring，所以也只是使用Spring的基本功能，所以需要使用到下面的这4个Jar包：   
  
除此之外，还要导入Spring支持的日志Jar包：   
这里写图片描述

### 步骤三：编写相关的类，在类中创建方法

在src目录下创建一个cn.itcast.ioc包，并在该包下创建一个User类。

public class User {

public void add() {

System.out.println("add.................");

}

}

### 步骤三：创建Spring配置文件

注意：Spring配置文件的名称和位置没有固定要求，一般建议把该文件放到src下面，名称可随便写，官方建议写成applicationContext.xml。但我觉得这个文件名称太长了，所以决定写为bean1.xml。然后我们还需要在配置文件中引入约束，Spring学习阶段的约束是schema约束。那么问题来了，这个约束又该怎么写呢？可参考docs\spring-framework-reference\html目录下的xsd-configuration.html文件，在其内容最后找到如下内容：

<?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">

<bean id="foo" class="x.y.Foo">

<meta key="cacheName" value="foo"/>

<property name="name" value="Rick"/>

</bean>

</beans>

然后将其复制黏贴到配置文件bean1.xml中，将其修改为：

<?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">

</beans>

### 步骤四：在配置文件中配置对象的创建

<!-- 1.配置user对象的创建 -->

<bean id="user" class="cn.itcast.ioc.User"></bean>

### 步骤五：编写测试程序

我们要在Spring中写代码来实现获取bean1.xml文件中配置的对象（该段代码不要求重点掌握，只是用在测试中而已）。这段代码主要用来解析Spring配置文件得到对象，但这个过程不需要我们写代码实现，Spring封装了一个对象帮我们进行了这些操作，这个对象叫ApplicationContext，它就能实现这个功能。   
在cn.itcast.ioc包下创建一个TestIOC单元测试类，如下：

public class TestIOC {

// 得到配置的user对象

@Test

public void demo1() {

// 1.加载Spring配置文件，把配置文件中的对象进行创建

ApplicationContext context =

new ClassPathXmlApplicationContext("bean1.xml"); // classpath：类路径，src目录下的文件最终要编译到类路径下

// 2.根据配置文件的id得到user对象

User user = (User) context.getBean("user");

System.out.println(user);

user.add();

}

}

**注意：classpath为类路径，src目录下的文件最终要编译到类路径下**。

# Spring的bean管理

通俗一点说，Spring的bean管理即指创建对象时不需要new操作代码实现，而是交给Spring进行配置完成。   
Spring进行bean管理有两种方式：

1. 使用配置文件方式实现
2. 使用注解方式实现

本文将重点放在第一种方式上，后一种方式后面会讲。

## Spring实例化bean的三种方式

### 使用无参构造（重点）

创建对象时候，调用类里面的无参数的构造方法实现。那么Spring配置文件中又该怎样写呢？基本类似于如下写法：

<!-- 1.配置user对象的创建 -->

<bean id="user" class="cn.itcast.ioc.User"></bean>

### 使用静态工厂（了解）

创建一个工厂类，在工厂类中提供一个静态的方法，这个方法返回类的对象；调用工厂类的方法时候，直接使用类名.方法名称即可以调用。下面举例来说明。   
在src目录下创建一个cn.itcast.bean包，并在该包下创建一个Bean1类。

public class Bean1 {

public void bean1() {

System.out.println("bean1..........");

}

}

然后在该包下创建一个Bean1Factory工厂类。

public class Bean1Factory {

// 静态方法

public static Bean1 getBean1() {

return new Bean1();

}

}

接着Spring配置文件中应向下面这样配置：

<!-- 2.使用静态工厂创建对象 -->

<bean id="bean1" class="cn.itcast.bean.Bean1Factory" factory-method="getBean1"></bean>

最后在该包下创建一个TestIOC单元测试类。

public class TestIOC {

@Test

public void demo1() {

// 1.加载Spring配置文件，把配置文件中的对象进行创建

ApplicationContext context =

new ClassPathXmlApplicationContext("bean1.xml");

// 2.根据配置文件的id得到user对象

Bean1 bean1 = (Bean1) context.getBean("bean1");

System.out.println(bean1);

}

}

### 使用实例工厂（了解）

创建一个工厂类，在工厂类里面提供一个普通的方法，这个方法返回类对象；调用工厂类的方法时候，创建工厂类对象，使用对象调用方法即可。下面也举例来说明。   
在src目录下的cn.itcast.bean包下创建一个Bean2类。

public class Bean2 {

public void bean2() {

System.out.println("bean2..........");

}

}

然后在该包下创建一个Bean2Factory工厂类。

public class Bean2Factory {

public Bean2 getBean2() {

return new Bean2();

}

}

接着Spring配置文件中应向下面这样配置：

<!-- 3.使用实例工厂创建对象 -->

<!-- 3.1先创建工厂对象 -->

<bean id="bean2Factory" class="cn.itcast.bean.Bean2Factory"></bean>

<!-- 3.2再使用工厂对象创建bean2对象 -->

<bean id="bean2" factory-bean="bean2Factory" factory-method="getBean2"></bean>

最后将TestIOC单元测试类的代码修改为：

public class TestIOC {

@Test

public void demo1() {

// 1.加载Spring配置文件，把配置文件中的对象进行创建

ApplicationContext context =

new ClassPathXmlApplicationContext("bean1.xml");

// 2.根据配置文件的id得到user对象

Bean2 bean2 = (Bean2) context.getBean("bean2");

System.out.println(bean2);

}

}

## Spring配置文件中bean标签常用的属性

Spring配置文件中bean标签常用的属性有以下四种：

1. id属性：根据id属性值得到配置对象。   
   在Spring配置文件中会有多个bean标签，但它们的id属性值是不能相同的。Bean起名字时，在约束中采用的是ID约束——唯一，而且名字必须以字母开始，可以使用字母、数字、连字符、下划线、句号、冒号等，但id属性值不能有特殊符号。
2. class属性：要创建对象的类的全路径。
3. scope属性：bean的作用范围。   
   scope属性共有以下5个属性：
   * singleton：创建的对象是单例的，也是scope属性的默认值。   
     下面我来举例说明它。将TestIOC单元测试类的代码修改为：
   * public class TestIOC {
   * // 得到配置的user对象
   * @Test
   * public void demo1() {
   * // 1.加载Spring配置文件，把配置文件中的对象进行创建
   * ApplicationContext context =
   * new ClassPathXmlApplicationContext("bean1.xml");
   * // 2.根据配置文件的id得到user对象
   * User user1 = (User) context.getBean("user");
   * User user2 = (User) context.getBean("user");
   * System.out.println(user1);
   * System.out.println(user2);
   * }

}

单元测试以上方法，一切就尽在不言中。其实，此时Spring配置文件中有关如下bean的配置：

<bean id="user" class="cn.itcast.ioc.User"></bean>

就相当于：

<bean id="user" class="cn.itcast.ioc.User" scope="singleton"></bean>

* + prototype：创建的对象是多实例的。   
    也可举例来说明它。将Spring配置文件中有关如下bean的配置：

<bean id="user" class="cn.itcast.ioc.User"></bean>

修改为：

<bean id="user" class="cn.itcast.ioc.User" scope="prototype"></bean>

测试单元测试类的方法就能明白了。

* + globalSession：用在**单点登录(即SSO，single sign on)**上。

1. name属性：name属性的功能和id属性是一样的。name属性和id属性区别是：在id属性值里面不能有特殊符号，在name属性值里面可以添加特殊符号。

## bean的生命周期的配置

通过配置<bean>标签上的init-method作为bean的初始化的时候执行的方法，配置destroy-method作为bean的销毁的时候执行的方法。销毁方法想要执行，需要是单例创建的Bean而且在工厂关闭的时候，Bean才会被销毁。

## Spring中Bean的属性注入

实际上，有关Bean的属性注入共有三种方式，下面我分别加以简单的说明：

set方法注入   
用代码可表示如下：

public class Book {

private String bookname;

public void setBookname(String bookname) {

this.bookname = bookname;

}

}

Book book = new Book();

book.setBookName("Java编程思想");

有参数构造注入   
用代码可表示如下：

public class Book {

private String bookname;

public Book(String bookname) {

this.bookname = bookname;

}

}

Book book = new Book("代码大全");

1. 接口注入   
   先编写一个接口：
2. public interface Dao {
3. public void add(String name);

}

再编写这个接口的实现类：

public class DaoImpl implements Dao {

private String name;

public void add(String name) {

this.name = name;

}

}

但在Spring框架里面，只支持前两种方式，即set方法注入和有参数构造注入。下面我来举例分别演示。

### 构造方法的方式注入属性

在src目录下创建一个cn.itcast.property包，并在该包下编写一个Book实体类。

public class Book {

private String bookname;

public Book(String bookname) {

this.bookname = bookname;

}

public void testBook() {

System.out.println("book.............." + bookname);

}

}

接着在Spring配置文件中对以上JavaBean添加如下配置：

<!-- 4.使用有参数的构造注入属性 -->

<bean id="book" class="cn.itcast.property.Book">

<!-- 使用标签，name：为属性的名字；value：为属性的值 -->

<constructor-arg name="bookname" value="beautifulMan\_美美侠"></constructor-arg>

</bean>

最后在该包下编写一个TestIOC单元测试类：

public class TestIOC {

@Test

public void demo1() {

// 1.加载Spring配置文件，把配置文件中的对象进行创建

ApplicationContext context =

new ClassPathXmlApplicationContext("bean1.xml");

Book book = (Book) context.getBean("book");

book.testBook();

}

}

自己自行测试去吧！

### set方法的方式注入属性

我们同样在cn.itcast.property包下编写一个Person实体类，在类中定义属性，并生成set方法。

public class Person {

// 1.定义一个属性

private String username;

// 2.生成这个属性的set方法

public void setUsername(String username) {

this.username = username;

}

public void testperson() {

System.out.println("person.............." + username);

}

}

然后在Spring配置文件中，使用bean标签创建对象，在bean标签里面使用property标签注入属性。即在Spring配置文件中对以上JavaBean添加如下配置，

<!-- 5.使用set方法进行注入属性 -->

<bean id="person" class="cn.itcast.property.Person">

<!--

使用property标签注入属性值

name：类属性名称

value属性：往属性中注入的值

-->

<property name="username" value="李阿昀"></property>

</bean>

最后将TestIOC单元测试类的代码修改为：

public class TestIOC {

@Test

public void demo1() {

// 1.加载Spring配置文件，把配置文件中的对象进行创建

ApplicationContext context =

new ClassPathXmlApplicationContext("bean1.xml");

Person person = (Person)context.getBean("person");

person.testperson();

}

}

自己自行测试去吧！

## Spring的属性注入：对象类型的注入

在实际开发中，我们要提交表单到action里面去，然后在action里面调用service层的方法，接着在service层里面调用dao层的方法。在这里，我假设在service层里面调用dao层的方法，所以需要在servcie层里面创建dao层的对象实现调用。   
先在src目录下创建一个cn.itcast.dao包，并在该包下编写一个UserDao类。

public class UserDao {

public void add() {

System.out.println("dao................");

}

}

然后在src目录下再创建一个cn.itcast.service包，并在该包下编写一个UserService类。

public class UserService {

public void add() {

System.out.println("service.........");

// 调用dao

balabala......

}

}

如果我们使用最原始的方式在service层里面调用dao层的方法，那么UserService类中add()方法应这样写：

public void add() {

System.out.println("service.........");

// 原始方式，调用dao

UserDao dao = new UserDao();

dao.add();

}

在Spring里面，我们就应该这么玩了。我们的最终目的是在service层里面得到dao层对象。所以步骤为：

第一步，让dao作为service的一个属性。

// 1.让dao作为service的一个属性

private UserDao userDao;

第二步，生成dao属性的set方法。

// 2.生成dao属性的set方法

public void setUserDao(UserDao userDao) {

this.userDao = userDao;

}

这时，UserService类的代码就变成：

public class UserService {

// 1.让dao作为service的一个属性

private UserDao userDao;

// 2.生成dao属性的set方法

public void setUserDao(UserDao userDao) {

this.userDao = userDao;

}

public void add() {

System.out.println("service.........");

userDao.add();

}

}

在Spring配置文件中进行配置和注入。

<!-- 6.注入对象的属性 -->

<!-- 6.1先创建dao对象 -->

<bean id="userDao" class="cn.itcast.dao.UserDao"></bean>

<!-- 6.2再创建service对象 -->

<bean id="userService" class="cn.itcast.service.UserService">

<!-- 在servcie里面注入userDao属性

name属性：service对象里面的userDao属性的名称

注入dao对象，不能写value属性，要写ref属性：dao配置的bean的id值

-->

<property name="userDao" ref="userDao"></property>

</bean>

## 名称空间p的属性注入方式——Spring2.x版本后提供的方式

在src目录下创建一个cn.itcast.propertydemo包，并在该包下编写一个Orders实体类。

public class Orders {

private String oname;

public void setOname(String oname) {

this.oname = oname;

}

public void testorders() {

System.out.println("orders................" + oname);

}

}

接下来我们需要在Spring核心配置文件中的schema约束位置定义p名称空间。

xmlns:p="http://www.springframework.org/schema/p"

紧接着，我们需要在Spring核心配置文件中添加如下配置：

<!-- 7.p(property，属性)名称空间的注入 -->

<bean id="orders" class="cn.itcast.propertydemo.Orders" p:oname="去你妈的！！！"></bean>

最后将cn.itcast.property包下的TestIOC单元测试类的代码修改为：

public class TestIOC {

@Test

public void demo1() {

// 1.加载Spring配置文件，把配置文件中的对象进行创建

ApplicationContext context =

new ClassPathXmlApplicationContext("bean1.xml");

Orders orders = (Orders)context.getBean("orders");

orders.testorders();

}

}

测试即可。   
**结论——使用p名称空间：**

* **普通属性：p:属性名称=”…”**
* **对象类型的属性：p:属性名称-ref=”…”**

## 注入复杂属性

### 注入数组类型的属性

将cn.itcast.propertydemo包下的Orders类的代码修改为：

public class Orders {

private String oname;

public void setOname(String oname) {

this.oname = oname;

}

// 1.数组类型的属性

private String[] arrays;

public void setArrays(String[] arrays) {

this.arrays = arrays;

}

public void testorders() {

// System.out.println("orders................" + oname);

System.out.println("数组：" + Arrays.toString(arrays));

}

}

然后需要在Spring核心配置文件中添加如下配置：

<bean id="orders" class="cn.itcast.propertydemo.Orders"> <!-- 创建对象 -->

<!-- 数组类型 -->

<property name="arrays"> <!-- 注入属性 -->

<list>

<value>叶子</value>

<value>liayun</value>

<value>杰哥</value>

</list>

</property>

</bean>

最后将cn.itcast.property包下的TestIOC单元测试类的代码修改为：

public class TestIOC {

@Test

public void demo1() {

// 1.加载Spring配置文件，把配置文件中的对象进行创建

ApplicationContext context =

new ClassPathXmlApplicationContext("bean1.xml");

Orders orders = (Orders)context.getBean("orders");

orders.testorders();

}

}

测试即可。

### 注入List集合类型的属性

将cn.itcast.propertydemo包下的Orders类的代码修改为：

public class Orders {

private String oname;

public void setOname(String oname) {

this.oname = oname;

}

// 2.list类型的属性

private List<String> list;

public void setList(List<String> list) {

this.list = list;

}

public void testorders() {

// System.out.println("orders................" + oname);

System.out.println("list：" + list);

}

}

然后需要在Spring核心配置文件中添加如下配置：

<bean id="orders" class="cn.itcast.propertydemo.Orders"> <!-- 创建对象 -->

<!-- list类型 -->

<property name="list">

<list>

<value>叶子</value>

<value>李昀玲</value>

<value>杰哥</value>

</list>

</property>

</bean>

cn.itcast.property包下的TestIOC单元测试类的代码勿须修改，直接测试即可。

### 注入Set集合类型的属性

将cn.itcast.propertydemo包下的Orders类的代码修改为：

public class Orders {

private String oname;

public void setOname(String oname) {

this.oname = oname;

}

// 3.set类型的属性

private Set<String> keyset;

public void setKeyset(Set<String> keyset) {

this.keyset = keyset;

}

public void testorders() {

// System.out.println("orders................" + oname);

System.out.println("set：" + keyset);

}

}

然后需要在Spring核心配置文件中添加如下配置：

<bean id="orders" class="cn.itcast.propertydemo.Orders"> <!-- 创建对象 -->

<!-- set类型 -->

<property name="keyset">

<set>

<value>蝙蝠侠</value>

<value>钢铁侠</value>

<value>美美侠</value>

</set>

</property>

</bean>

其实，以上配置也可以写为：

<bean id="orders" class="cn.itcast.propertydemo.Orders"> <!-- 创建对象 -->

<!-- set类型 -->

<property name="keyset">

<list>

<value>蝙蝠侠</value>

<value>钢铁侠</value>

<value>美美侠</value>

</list>

</property>

</bean>

cn.itcast.property包下的TestIOC单元测试类的代码勿须修改，直接测试即可。

### 注入Map集合类型的属性

将cn.itcast.propertydemo包下的Orders类的代码修改为：

public class Orders {

private String oname;

public void setOname(String oname) {

this.oname = oname;

}

// 4.map类型

private Map<String, String> map;

public void setMap(Map<String, String> map) {

this.map = map;

}

public void testorders() {

// System.out.println("orders................" + oname);

System.out.println("map：" + map);

}

}

然后需要在Spring核心配置文件中添加如下配置：

<bean id="orders" class="cn.itcast.propertydemo.Orders"> <!-- 创建对象 -->

<!-- map类型 -->

<property name="map">

<map>

<entry key="username" value="潘金莲"></entry>

<entry key="password" value="1314"></entry>

<entry key="address" value="明初"></entry>

</map>

</property>

</bean>

cn.itcast.property包下的TestIOC单元测试类的代码勿须修改，直接测试即可。

### 注入Properties类型的属性

将cn.itcast.propertydemo包下的Orders类的代码修改为：

public class Orders {

private String oname;

public void setOname(String oname) {

this.oname = oname;

}

// 5.properties

private Properties properties;

public void setProperties(Properties properties) {

this.properties = properties;

}

public void testorders() {

// System.out.println("orders................" + oname);

System.out.println("properties：" + properties);

}

}

然后需要在Spring核心配置文件中添加如下配置：

<bean id="orders" class="cn.itcast.propertydemo.Orders"> <!-- 创建对象 -->

<!-- properties类型 -->

<property name="properties">

<props>

<prop key="name">宫本一郎</prop>

<prop key="address">日本</prop>

</props>

</property>

</bean>

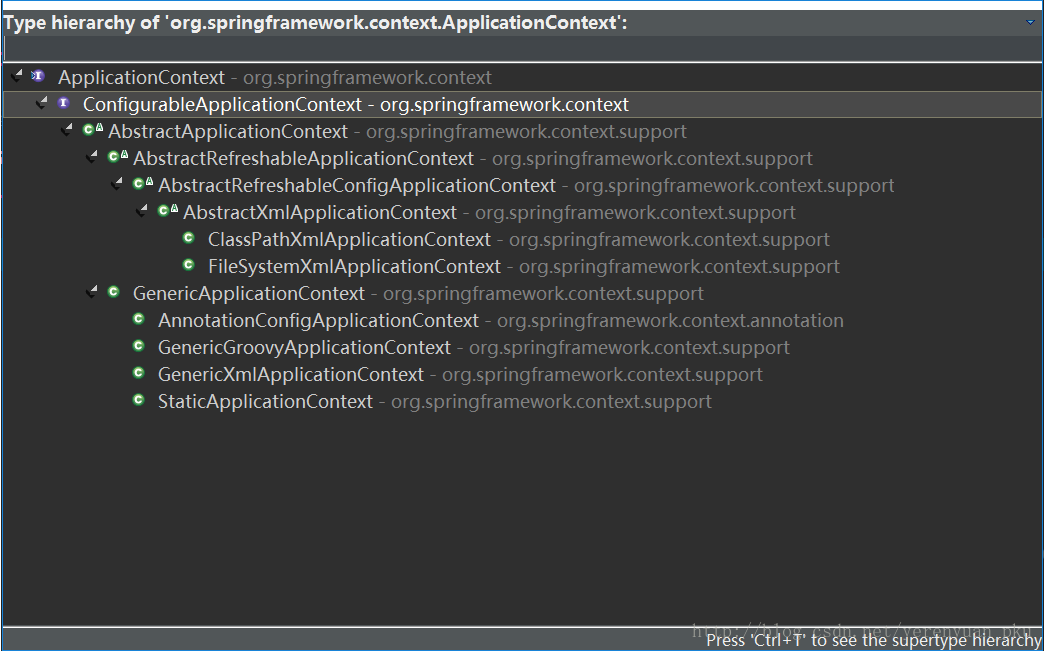
cn.itcast.property包下的TestIOC单元测试类的代码勿须修改，直接测试即可。

## IoC和DI的区别

* IoC：控制反转，即把对象的创建交给Spring进行管理。所以Spring IoC容器是用来创建对象，管理依赖关系的。
* DI(Dependency Injection)：依赖注入，即在创建对象的过程中，向类里面的属性中设置值。
* IoC和DI的关系：依赖注入不能单独存在，须在控制反转基础之上完成，用更通俗点的话来说，就是注入类里面的属性值，不能直接注入，须创建类的对象再完成注入。

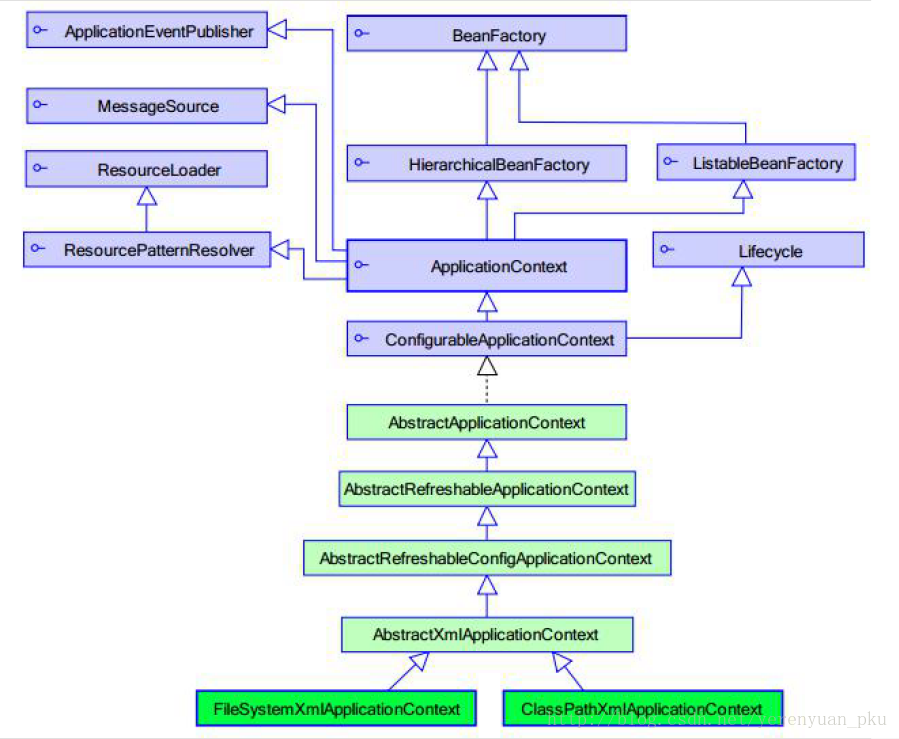
## Spring中的工厂

### ApplicationContext

ApplicationContext接口有两个实现类，如下图：   


* ClassPathXmlApplicationContext：加载的是类路径下的Spring配置文件
* FileSystemXmlApplicationContext：加载的是本地磁盘下的Spring配置文件

### BeanFactory

下面应该是BeanFactory类的继承体系图吧！   
  
稍微了解一下就好，这个类在实际开发中我们并不需要用到。

### ApplicationContext和BeanFactory的区别

虽然使用这两个对象都可以加载Spring的配置文件，并创建配置文件中的对象。但他俩还是有区别的，最主要的区别是：

* 使用applicationContext操作时，可把Spring里面的配置文件中的对象都进行创建。
* 使用BeanFactory对象操作时，在调用getBean方法的时候进行对象的创建。

# Spring整合Web项目的底层原理

在实际的开发中，我们一般会使用SSH(即Struts2、Spring、Hibernate)进行开发，然后创建Spring的配置文件，使用applicationContext对象加载配置文件，继而创建对象。在真正的开发中，我们一般不会直接写applicationContext代码加载配置文件。

## Spring整合Web项目的思想

在服务器启动的时候，加载Spring配置文件，并创建对象。

## Spring整合Web项目的底层原理

Spring整合Web项目用到以下两个技术：

* 使用ServletContext对象(在服务器启动时创建)
* 使用监听器

在服务器启动的时候，每个项目都会创建一个ServletContext对象，而且每个项目只有一个ServletContext对象。在ServletContext对象创建的时候，使用监听器可以监听到其创建。当我们监听到ServletContext对象创建时，Spring就会帮我们做一件事情——加载Spring配置文件，并把配置文件中的对象进行创建，对象创建之后，放到ServletContext域里面去，最终我们要使用创建的对象，可从ServletContext域里面获取出来。

# Spring的bean管理（注解方式）

注解：代码中的特殊标记，注解可以使用在类、方法、属性上面，使用注解可实现一些基本的功能。注解的写法是@注解名称(属性=属性值)。

## 使用注解创建对象

**第一步，创建Web项目，引入Spring的开发包**   
除了导入Spring基本的Jar包外(可参考《Spring的概述》一文)，还须导入Spring注解的Jar包，如下：   
这里写图片描述  
**第二步，编写相关的类**   
在src目录下创建一个cn.itcast.anno包，并在该包下编写一个User类。

public class User {

public void add() {

System.out.println("add....................");

}

}

**第三步，创建Spring配置文件**

在Spring配置文件中引入约束，如下：

<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"

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">

在Spring配置文件中做些事情，即开启注解扫描。

<!-- 开启注解的扫描。到配置的包里面扫描类、方法、属性上面是否有注解 -->

<context:component-scan base-package="cn.itcast"></context:component-scan>

注意：也可以这样开启注解扫描，如下：

<context:annotation-config></context:annotation-config>

但是这种开启注解扫描的方式，只会扫描属性上面的注解。实际开发中用到的并不多！故不推荐使用。

**第四步，在创建对象所在的类上面使用注解实现**

@Component(value="user") // 类似于<bean id="user" class="..." />

public class User {

public void add() {

System.out.println("add....................");

}

}

如若注解里面属性名称是value，则可以省略，所以上面的User类亦可这样写为：

@Component("user") // 类似于<bean id="user" class="..." />

public class User {

public void add() {

System.out.println("add....................");

}

}

**第五步，编写测试类**   
在cn.itcast.anno包下编写一个TestDemo单元测试类。

public class TestDemo {

@Test

public void testUser() {

ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");

User user = (User) context.getBean("user");

System.out.println(user);

user.add();

}

}

测试即可。

## Spring的bean管理中常用的注解

### @Component(作用在类上)

创建对象的注解有四个：

1. @Component
2. @Repository：用于对DAO实现类进行标注(持久层)。
3. @Service：用于对Service实现类进行标注(业务层)。
4. @Controller：用于对Controller实现类进行标注(WEB层)。

后三个注解是Spring中提供的@Component的三个衍生注解(功能目前来讲是一样的)，它们是为了让标注类本身的用途更清晰，Spring在后续的版本中会对其进行增强。

### bean的作用范围的注解

@Scope

* singleton：单例，默认值
* prototype：多例

所以我们可通过注解设置创建对象是单例或者还是多实例的。这样User类的代码亦可写为：

@Service("user")

@Scope("singleton")

public class User {

public void add() {

System.out.println("add....................");

}

}

## 使用注解注入属性（对象）

这儿，我举个例子来说明如何使用注解注入(对象类型的)属性。   
先创建业务层中的UserService类：

public class UserService {

public void add() {

System.out.println("service...........");

}

}

再创建持久层中的UserDao类：

public class UserDao {

public void add() {

System.out.println("dao................");

}

}

以上两个类都创建在cn.itcast.anno包中。我们要实现的目的是在UserService类里面调用UserDao类的方法，这样我们就要在UserService类里面得到UserDao类的对象。之前是采用xml配置文件的方式来注入属性的，本文将使用注解的方式完成注入属性的操作。

1. 在UserService类里面定义UserDao类型属性

private UserDao userDao;

在UserService类里面定义UserDao类型的属性，由于是使用注解的方式，故不需要手动生成set方法。

1. 进行注解方式实现属性注入

创建UserDao类的对象和UserService类的对象

@Service("userService")

public class UserService {

private UserDao userDao;

public void add() {

System.out.println("service...........");

userDao.add();

}

}

@Repository("userDao")

public class UserDao {

public void add() {

System.out.println("dao................");

}

}

在UserService类里面注入UserDao类的对象，使用注解来实现。首先我使用@Autowired注解来实现。

@Service("userService")

public class UserService {

@Autowired

private UserDao userDao;

public void add() {

System.out.println("service...........");

userDao.add();

}

}

注意：使用注解@Autowired，它不是根据名字去找Dao，而是默认按类型进行装配。   
当然了，也可使用@Resource注解来实现，如下：

@Service("userService")

public class UserService {

@Resource(name="userDao")

private UserDao userDao;

public void add() {

System.out.println("service...........");

userDao.add();

}

}

注意，使用@Resource注解，它默认是按名称进行注入的。在实际开发中，我们也是使用@Resource注解来注入属性的，注解@Autowired用到的并不多。

# AOP的概述

## 什么是AOP

Spring是用来解决实际开发中的一些问题的，AOP解决了OOP中遇到的一些问题，是OOP的延续和扩展。我们可从以下三个方面来理解AOP：

1. 扩展功能不是通过修改源代码而实现的。   
   可通过Struts2框架中的拦截器来理解。
2. AOP采用横向抽取机制实现。   
   要理解横向抽取机制，就必须认识纵向抽取机制。例如有如下的一个类：
3. public class User {

public void add() {

添加的逻辑...

}

}

现在我们在add()方法中要扩展一个功能，即日志添加功能，添加完该功能之后，可记录在什么时候添加了哪个用户。想到的最原始的方法就是直接修改源代码。

public class User {

public void add() {

添加的逻辑...

直接写添加日志记录的代码以实现...

}

}

很显然这是一种愚蠢的做法，并且这儿还有一个原则——**修改功能一般不是直接修改源代码来实现的**。顺其自然地就要来讲纵向抽取机制了，这时我们可编写一个BaseUser类。

public class BaseUser {

public void wirtelog() {

记录日志的逻辑...

}

}

接下来让User类继承BaseUser类，如下：

public class User extends BaseUser {

public void add() {

添加的逻辑...

// 记录日志

super.wirtelog();

}

}

这样是不是就万事大吉了呢？你懂的！因为当父类的方法名称变化时，子类调用的方法也必然要进行修改。最后终于要讲横向抽取机制了，横向抽取机制分为两种情况，下面分别加以简单阐述。   
**有接口情况的横向抽取机制：**   
例如有一个如下接口：

public interface UserDao {

public void add();

}

接口的一个实现类如下：

public class UserDaoImpl implements UserDao {

public void add() {

...

}

}

我们现在即可使用动态代理技术增强类里面的方法，即创建接口的实现类代理对象，并增强UserDaoImpl类里面的add()方法。   
**无接口情况的横向抽取机制：**   
例如有一个如下User类：

public class User {

public void add() {

...

}

}

我们也可使用动态代理技术增强类里面的方法，即创建被增强方法所在类的子类代理对象，并增强User类里面的add()方法。

1. AOP底层使用动态代理实现
   * 有接口情况：创建接口实现类代理对象
   * 没有接口情况：创建增强方法所在类的子类代理对象

## 为什么学习AOP

在不修改源代码的情况下，即可对程序进行增强。AOP可以进行权限校验、日志记录、性能监控和事务控制。

## Spring的AOP的由来

AOP最早是由AOP联盟的组织提出的，他们制定了一套规范。Spring将AOP思想引入到框架中，且必须遵守AOP联盟的规范。

## Spring的AOP的底层实现

Spring的AOP的底层用到了两种代理机制：

* JDK的动态代理：针对实现了接口的类产生代理。
* Cglib的动态代理：针对没有实现接口的类产生代理，应用的是底层的字节码增强的技术，生成当前类的子类对象。

## AOP开发中的相关术语

以下是比较专业的术语。

* Joinpoint(连接点)：所谓连接点是指那些被拦截到的点。在Spring中，这些点指的是方法，因为Spring只支持方法类型的连接点。
* Pointcut(切入点)：所谓切入点是指我们要对哪些Joinpoint进行拦截的定义。
* Advice(通知/增强)：所谓通知是指拦截到Joinpoint之后所要做的事情就是通知。通知分为前置通知、后置通知、异常通知、最终通知和环绕通知(切面要完成的功能)。
* Aspect(切面)：是切入点和通知的结合。
* Target(目标对象)：代理的目标对象(要增强的类)
* Weaving(织入)：是指把增强应用到目标对象来创建新的代理对象的过程。Spring采用动态代理织入，而AspectJ采用编译期织入和类装载期织入。
* Proxy(代理)：一个类被AOP织入增强后，就产生一个结果代理类。
* Introduction(引介)：引介是一种特殊的通知在不修改类代码的前提下，Introduction可以在运行期为类动态地添加一些方法或Field。



例如有如下一个类：

public class User {

public void add() {

}

public void update() {

}

public void delete() {

}

}

下面是用比较通俗易懂的话来阐述AOP开发中的常见的相关术语：

* 连接点：在User类里面有3个方法，这3个方法都可以被增强，类里面的哪些方法可以被增强，这些方法就可被成为连接点。
* 切入点：在一个类中可以有很多的方法被增强，在实际操作中，如若只增强了类里面的add方法，则实际增强的方法被称为切入点。
* 增强/通知：比如增强User类里面的add方法，在add方法中添加了日志功能，这个日志功能就称为增强。   
  通知类型：
  + 前置通知：在增强的方法执行之前进行操作。
  + 后置通知：在增强的方法执行之后进行操作。
  + 环绕通知：在增强的方法执行之前和执行之后进行操作。
  + 最终通知：增强了两个方法，执行第一个方法，执行第二个方法，在第二个方法执行之后进行操作。   
    也可理解为后置通知后面执行的通知或者**无论目标方法是否出现异常，最终通知都会执行**。
  + 异常通知：程序出现异常之后执行的通知。
* 切面：把增强应用到切入点的过程。即把具体增强的逻辑用到具体的方法上面的过程。
* 目标对象：增强的方法所在的类，即要增强的类。
* Weaving(织入)：是指把增强应用到目标对象的过程。即把把advice应用到target的过程。

# Spring的基于AspectJ的AOP开发

## @AspectJ的简介

AspectJ是一个面向切面的框架，它扩展了Java语言。AspectJ定义了AOP语法所以它有一个专门的编译器用来生成遵守Java字节编码规范的Class文件。AspectJ是一个基于Java语言的AOP框架。**Spring2.0以后新增了对AspectJ切点表达式的支持**。@AspectJ是AspectJ1.5新增的功能，通过JDK5注解技术，允许直接在Bean类中定义切面。新版本Spring框架，建议使用AspectJ方式来开发AOP，使用AspectJ需要导入Spring AOP和AspectJ相关的Jar包。   
从上面的阐述中，我们应认识到AspectJ并不是Spring框架的一部分，而是一个单独的面向切面的框架，只不过它经常和Spring框架一起使用进行AOP的操作而已。   
使用AspectJ方式来开发AOP共有两种方式：

1. 基于AspectJ的xml配置文件的方式
2. 基于AspectJ的注解的方式

只不过本文讲解的是基于AspectJ的xml配置文件的方式，下文再讲第二种方式。

## Spring使用AspectJ进行AOP的开发：XML的方式

**第一步，引入相应的Jar包**   
上面我说过，除了导入最基本的Jar包外，使用AspectJ还需要导入Spring AOP和AspectJ相关的Jar包。

* Spring的传统AOP的开发的包：
  + spring-aop-4.2.4.RELEASE.jar
  + aopalliance-1.0.jar
* AspectJ的开发包
  + aspectjweaver-1.8.7.jar
  + spring-aspects-4.2.4.RELEASE.jar

**第二步，编写目标类**   
在src目录下创建一个cn.itcast.aop包，并在该包下编写一个目标类。

public class Book {

public void add() {

System.out.println("book add.................");

}

}

**第三步，创建增强的类以及增强的方法**

public class MyBook {

// 前置通知

public void before1() {

System.out.println("before........");

}

}

我们现在要求在Book类里面的add方法之前执行MyBook类里面的before1的方法。   
**第四步，在Spring配置文件中进行配置**

1. 在Spring配置文件中引入aop约束
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop" xsi:schemaLocation="
5. http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans.xsd
7. http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

1. 配置两个类对象
2. <!-- book对象 -->
3. <bean id="book" class="cn.itcast.aop.Book"></bean>

<bean id="myBook" class="cn.itcast.aop.MyBook"></bean>

1. 配置AOP操作，即需要配置切入点和切面。
2. <!-- 配置AOP的操作 -->
3. <aop:config>
4. <!-- 配置切入点，对Book类里面的所有方法都增强 -->
5. <aop:pointcut expression="execution(\* cn.itcast.aop.Book.\*(..))" id="pointcut1"></aop:pointcut>
6. <!-- 配置切面 aop:aspect标签里面使用属性ref，ref属性值写增强类的bean的id值 -->
7. <aop:aspect ref="myBook">
8. <!--
9. 增强类型
10. method属性：增强类的方法名称
11. pointcut-ref属性：切入点的id值
12. -->
13. <!-- 前置通知 -->
14. <aop:before method="before1" pointcut-ref="pointcut1"></aop:before>
15. </aop:aspect>

</aop:config>

**第五步，编写一个单元测试类并进行测试**   
在cn.itcast.aop包下编写一个TestDemo单元测试类。

public class TestDemo {

@Test

public void testUser() {

ApplicationContext context = new ClassPathXmlApplicationContext("bean2.xml");

Book book = (Book) context.getBean("book");

book.add();

}

}

其实我们也可以整合Junit单元测试，Spring对Junit4进行了支持，可以通过注解方便的测试Spring程序，所以就不必写那么麻烦的单元测试类了。首先导入如下Jar包：   
这里写图片描述  
然后编写如下的单元测试类：

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("/bean2.xml") // 或者也可写为：@ContextConfiguration("classpath:bean2.xml")

public class TestDemo {

@Autowired

private Book book;

@Test

public void demo1() {

book.add();

}

}

### 演示其他通知类型

先将MyBook增强类的代码修改为：

public class MyBook {

// 前置通知

public void before1() {

System.out.println("before........");

}

// 后置通知

public void after11() {

System.out.println("after...........");

}

// 环绕通知

public void around1(ProceedingJoinPoint joinPoint) throws Throwable {

System.out.println("方法之前执行...........");

// 让被增强的方法执行

joinPoint.proceed();

System.out.println("方法之后执行...........");

}

}

然后再在Spring配置文件中进行配置。

<!-- 配置AOP的操作 -->

<aop:config>

<!-- 配置切入点，对Book类里面的所有方法都增强 -->

<aop:pointcut expression="execution(\* cn.itcast.aop.Book.\*(..))" id="pointcut1"></aop:pointcut>

<!-- 配置切面 aop:aspect标签里面使用属性ref，ref属性值写增强类的bean的id值 -->

<aop:aspect ref="myBook">

<!--

增强类型

method属性：增强类的方法名称

pointcut-ref属性：切入点的id值

-->

<!-- 前置通知 -->

<aop:before method="before1" pointcut-ref="pointcut1"></aop:before>

<!-- 后置通知 -->

<aop:after-returning method="after11" pointcut-ref="pointcut1"></aop:after-returning>

<!-- 环绕通知 -->

<aop:around method="around1" pointcut-ref="pointcut1"></aop:around>

</aop:aspect>

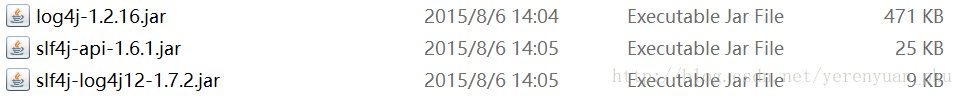
</aop:config>

## 切入点表达式

通过execution函数，可以定义切点的方法切入。   
语法为：execution(<访问修饰符>?<返回类型><方法名>(<参数>)<异常>)。   
例如：

* 匹配所有类的public方法：execution(public \*.\*(..))
* 匹配指定包下所有类的方法：execution(\* cn.itcast.dao.\*(..))，但不包含子包
* execution(\* cn.itcast.dao..\*(..))，..\*表示包、子孙包下所有类。
* 匹配指定类所有方法：execution(\* cn.itcast.service.UserService.\*(..))
* 匹配实现特定接口的所有类的方法：execution(\* cn.itcast.dao.GenericDAO+.\*(..))
* 匹配所有save开头的方法：execution(\* save\*(..))
* 匹配所有类里面的所有的方法：execution(\* \*.\*(..))

# Log4j操作

在项目开发中，我们通常会导入类似这样的日志Jar包：   
  
关于这些日志Jar包之间的关系我也搞的不是很清楚，只能作简单的记录。   
使用Log4j，可以查看到当前运行程序中对象创建的过程，也可以看到更详细的信息。Log4j适合使用在程序调试中。   
例如，在本文中讲解上面的Web项目使用Log4j。   
首先导入log4j的jar包，如下：   
这里写图片描述  
然后添加log4j配置文件——log4j.properties，添加到src目录下面，该文件内容如下：

### direct log messages to stdout ###

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.err

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

### direct messages to file mylog.log ###

log4j.appender.file=org.apache.log4j.FileAppender

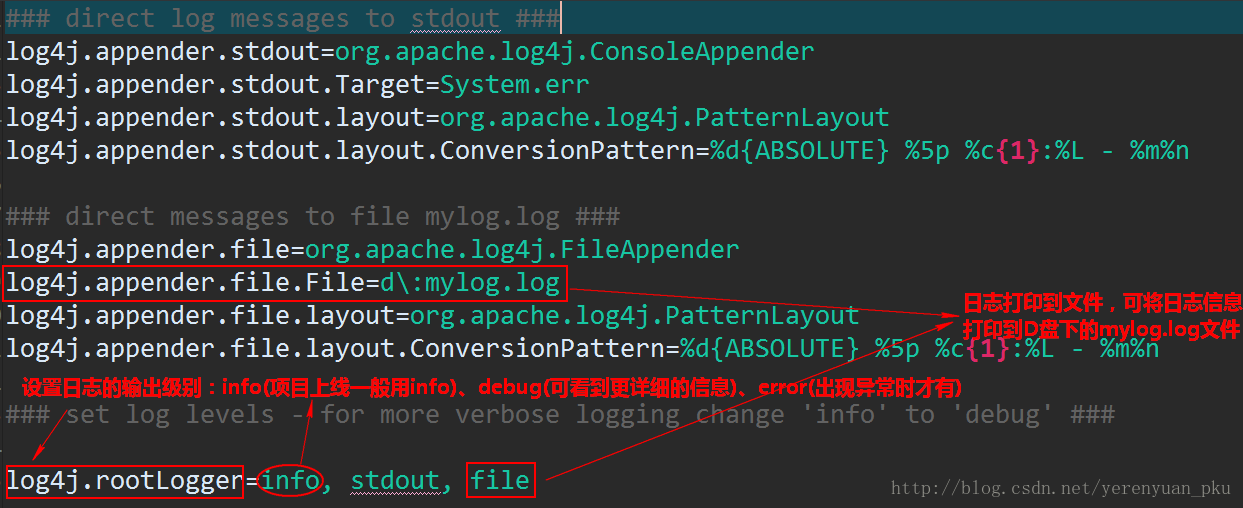
log4j.appender.file.File=d\:mylog.log

log4j.appender.file.layout=org.apache.log4j.PatternLayout

log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

### set log levels - for more verbose logging change 'info' to 'debug' ###

log4j.rootLogger=info, stdout, file

粗略讲解如下图：   


# IoC配置文件和注解混合使用

实际开发中，我们都会混合使用IoC配置文件和注解，一般是使用配置文件方式创建对象，使用注解方式注入属性。当然了，你亦可另辟蹊径。下面我举个例子来演示。   
在src目录下创建一个cn.itcast.xmlanno包，并在该包下编写一个BookDao类、PersonDao类、BookService类。

* BookDao类
* public class BookDao {
* public void update() {
* System.out.println("book dao.............");
* }

}

* PersonDao类
* public class PersonDao {
* public void update() {
* System.out.println("person dao.............");
* }

}

* BookService类
* public class BookService {
* public void update() {
* System.out.println("service............");
* }

}

接着创建对象，使用配置文件实现。

<bean id="bookService" class="cn.itcast.xmlanno.BookService"></bean>

<bean id="bookDao" class="cn.itcast.xmlanno.BookDao"></bean>

<bean id="personDao" class="cn.itcast.xmlanno.PersonDao"></bean>

也有人说创建对象，能用注解就用注解，而不要写上面这种乱七八糟的东西。我觉得也蛮有道理的。   
然后在BookService类里面注入BookDao类以及PersonDao类的对象，使用注解方式。

public class BookService {

@Resource(name="bookDao")

private BookDao bookDao;

@Resource(name="personDao")

private PersonDao personDao;

public void update() {

System.out.println("service............");

bookDao.update();

personDao.update();

}

}

注意，不要忘了开启注解的扫描：

<context:component-scan base-package="cn.itcast"></context:component-scan>

最后再在cn.itcast.xmlanno包下编写一个TestDemo单元测试类。

public class TestDemo {

@Test

public void testBook() {

ApplicationContext context = new ClassPathXmlApplicationContext("bean3.xml");

BookService bookService = (BookService) context.getBean("bookService");

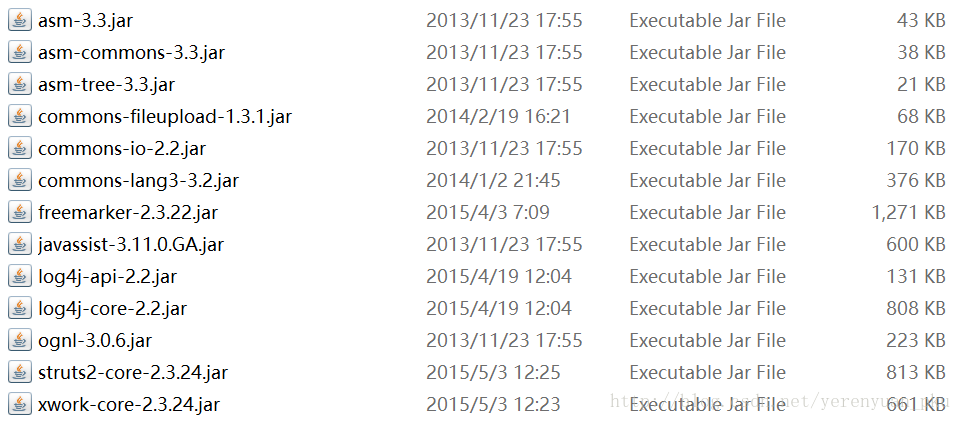
bookService.update();

}

}

测试即可。

# Spring整合Web项目

现在我来介绍如何使用Spring整合Web项目。在真正的开发中，都会在action中调用service层的代码，service层再调用dao层的代码。下面我就举个例子来演示。   
首先创建Web项目，引入Struts2的开发包，可参考struts-2.3.24中的apps下的一些示例代码，其中struts2-blank.war是一个Struts2的空的工程。我们只需要将struts2-blank.war解压后进入到WEB-INF下的lib中查看，即可知道要导入哪些Struts2的开发包。   
  
然后在src目录下创建一个cn.itcast.action包，并在该包下编写一个Action类。

public class UserAction extends ActionSupport {

@Override

public String execute() throws Exception {

System.out.println("action.........");

return NONE;

}

}

接着就要完成该UserAction类的配置，在src目录下引入Strust2框架的核心配置文件——struts.xml，其内容如下：

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE struts PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"

"http://struts.apache.org/dtds/struts-2.3.dtd">

<struts>

<package name="demo" extends="struts-default" namespace="/">

<action name="user" class="cn.itcast.action.UserAction"></action>

</package>

</struts>

不要忘了配置核心过滤器，打开web.xml，在web.xml文件中添加如下配置：

<!-- 配置Struts2的过滤器 -->

<filter>

<filter-name>struts2</filter-name>

<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>struts2</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

接着再在src目录下创建一个cn.itcast.service包，并在该包下编写一个UserService类。

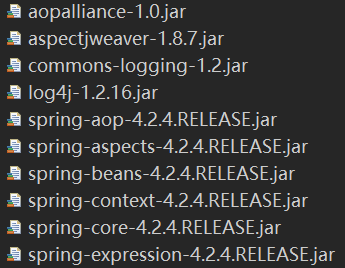
public class UserService {

public void add() {

System.out.println("service............");

}

}

上面基本上将Struts2框架的开发环境搭建好了，接下来就要使用Spring整合该Web项目了。   
首先导入Spring框架开发的Jar包：   
  
既然上面已讲了log4j日志包的操作，这里还是在src目录下引入log4j.properties文件。

### direct log messages to stdout ###

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.err

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

### direct messages to file mylog.log ###

log4j.appender.file=org.apache.log4j.FileAppender

log4j.appender.file.File=d\:mylog.log

log4j.appender.file.layout=org.apache.log4j.PatternLayout

log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

### set log levels - for more verbose logging change 'info' to 'debug' ###

log4j.rootLogger=info, stdout, file

然后再在src目录下引入Spring的核心配置文件——bean1.xml，创建并管理UserService类的对象。

<?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"

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">

<bean id="userService" class="cn.itcast.service.UserService"></bean>

</beans>

接着改写UserAction类的代码，让其调用service层的代码。

public class UserAction extends ActionSupport {

@Override

public String execute() throws Exception {

System.out.println("action.........");

ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");

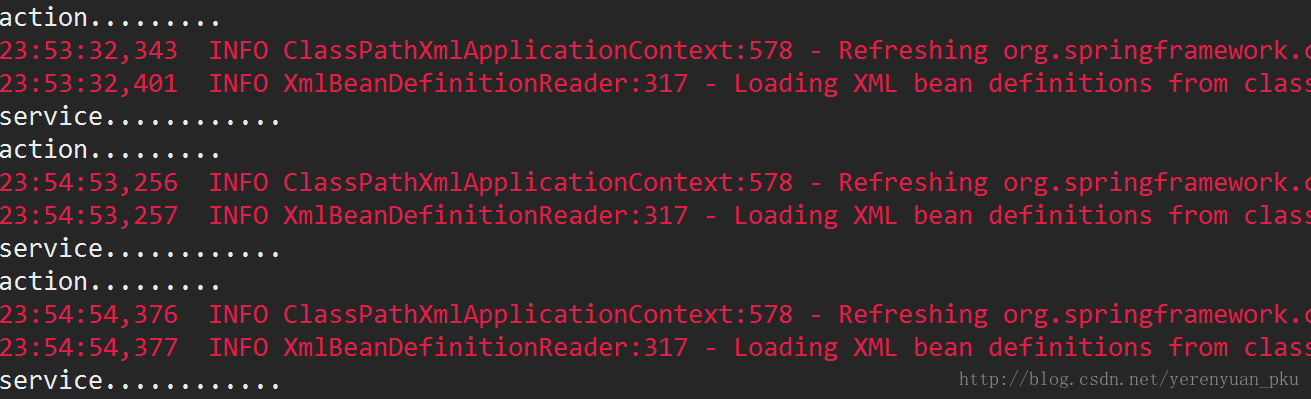
UserService userService = (UserService) context.getBean("userService");

userService.add();

return NONE;

}

}

最后我们在浏览器地址栏中输入地址http://localhost:8080/Spring\_day02\_web/user.action进行访问，并多次刷新，在Eclipse控制台中可以看到如下信息：   
  
这就产生了一个问题：由于action对象是一个多实例的对象，所以每次访问action都会创建一个action对象。虽然功能是可以实现，但是性能很低下。为了解决这个问题，需要使用到2个技术：

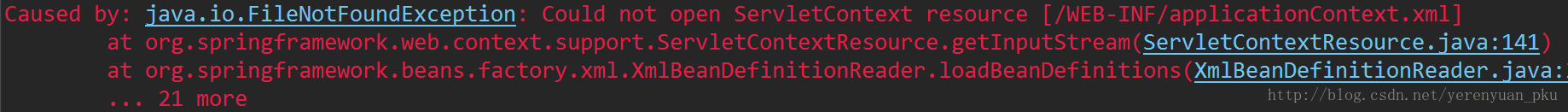
* ServletContext对象
* 监听器

在服务器启动的时候创建ServletContext对象，使用监听器可以知道ServletContext对象在什么时候创建，然后加载配置文件，创建配置的对象，把创建的对象放到ServletContext域里面。但在Spring里面不需要我们写这些操作的代码，因为它已经帮我们进行了封装，Spring里面封装了一个监听器，我们只需要配置监听器就可以了。使用Spring里面的监听器的时候，需要导入一个jar包，这个jar包用于整合Web项目。   
这里写图片描述  
接着在web.xml文件中配置Spring的监听器，这样在服务器启动的时候，就会加载Spring的配置文件了。

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

这时我们启动服务器，会在Eclipse的控制台中看到如下异常信息：   
  
这是因为服务器启动的时候，会默认到WEB-INF下面找寻名称是applicationContext.xml的配置文件。要解决这个异常，就要手动配置让服务器找寻哪个Spring配置文件，可在web.xml文件中添加如下配置：

<!-- 指定找到Spring配置文件的位置和名称 -->

<context-param>

<param-name>contextConfigLocation</param-name>

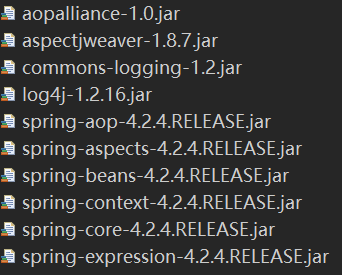
<param-value>classpath:bean1.xml</param-value>

</context-param>

这样问题就解决了。但我们还要明白Spring整合Web项目还没完，也不用急，下文我会接着介绍的。

# Spring使用AspectJ进行AOP的开发：注解方式

之前我已讲过Spring使用AspectJ通过配置文件的方式来进行AOP的开发，现在就来讲怎样使用注解方式进行AOP的开发。

* 创建一个Web项目， 引入相关的jar包。所要导入的jar包如下：   
  
* 引入Spring的配置文件。主要引入AOP的约束：
* <?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: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/aop
* http://www.springframework.org/schema/aop/spring-aop.xsd">

</beans>

* 编写目标类。   
  在Web项目的src目录下创建一个cn.itcast.aop包，并在该包下编写一个Book类：
* public class Book {
* public void add() {
* System.out.println("book add.................");
* }

}

* 编写切面类。   
  再在cn.itcast.aop包下编写一个MyBook增强类：
* public class MyBook {
* public void before1() {
* System.out.println("before........");
* }

}

* 在增强的类上面使用@Aspect注解
* @Aspect
* public class MyBook {
* public void before1() {
* System.out.println("before........");
* }

}

* 在增强类的方法里面，使用注解配置通知类型：
* @Aspect
* public class MyBook {
* // 前置通知
* // \*：方法的访问修饰符，也可写为execution(public void cn.itcast.aop.Book.\*(..))，但一般都不会用
* @Before("execution(\* cn.itcast.aop.Book.\*(..))")
* public void before1() {
* System.out.println("before........");
* }

}

AspectJ的AOP的注解：

* + @Aspect：定义切面类的注解
  + @Before：前置通知，相当于BeforeAdvice
  + @AfterReturning：后置通知，相当于AfterReturningAdvice
  + @Around：环绕通知，相当于MethodInterceptor
  + @AfterThrowing：抛出通知，相当于ThrowAdvice
  + @After：最终final通知，不管是否异常，该通知都会执行
  + @Pointcut：定义切入点的注解
* 开启AspectJ的注解

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

* 创建Book类和MyBook类的两个对象(使用配置文件)
* <bean id="book" class="cn.itcast.aop.Book"></bean>

<bean id="myBook" class="cn.itcast.aop.MyBook"></bean>

* 最后在cn.itcast.aop包下编写一个TestDemo单元测试类
* public class TestDemo {
* @Test
* public void testBook() {
* ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");
* Book book = (Book) context.getBean("book");
* book.add();
* }

}

测试即可。

# Spring的JDBC模板

Spring是一个分层的JavaSE/EEfull-stack(一站式)轻量级开源框架。它针对JavaEE三层中的每一层都提供了不同的解决技术，在dao层，Spring提供了JDBC模板的技术，可对数据库进行CRUD操作。Spring提供了很多持久层技术的模板类简化了编程，如下图：   
  
我再次加以说明：Spring框架对不同的持久层技术做了封装，如对传统的JDBC使用JdbcTemplate进行了封装，对Hibernate框架使用HibernateTemplate进行了封装。JdbcTemplate对JDBC进行了简单封装，使用类似于dbutils，但是使用并没有dbutils方便，只是提供了一种实现的方式而已。下面来演示使用JdbcTemplate模板类实现CRUD操作。

## 使用JdbcTemplate模板类实现CRUD操作

首先创建数据库和表，如下：

create database spring\_lee;

use spring\_lee;

DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (

`username` varchar(100) DEFAULT NULL,

`password` varchar(100) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

使用JdbcTemplate模板类还须导入jar包，先引入JdbcTemplate的jar包：   
这里写图片描述  
但我们要知道spring-jdbc-4.2.4.RELEASE.jar包才是最主要的。除此之外还须导入MySQL数据库驱动的jar包。

## 添加操作

在src目录下创建一个cn.itcast.jdbcTemplate包，并在该包下编写一个JdbcTemplateDemo1单元测试类。现在要编写一个add方法来测试添加操作。

public class JdbcTemplateDemo1 {

// 1.添加操作

@Test

public void add() {

// 1.设置数据库相关信息(JDBC模板依赖连接池获得数据库连接，所以必须先构造连接池)

DriverManagerDataSource dataSource = new DriverManagerDataSource(); // 数据源

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///spring\_lee");

dataSource.setUsername("root");

dataSource.setPassword("yezi");

// 2.做添加的操作

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

String sql = "insert into user values(?,?)";

int rows = jdbcTemplate.update(sql, "liayun", "lee");

System.out.println(rows);

}

}

注意：JDBC模板依赖连接池获得数据库连接，所以必须先构造连接池，然后再创建JdbcTemplate模板类对象。而且还须用到JdbcTemplate模板类的update方法：   
这里写图片描述  
这个方法中有两个参数：

* 第一个参数是sql语句。
* 第二个参数是传递的参数值，Object类型的可变参数。

## 修改操作

现在要在单元测试类中编写一个update方法来测试修改操作。

public class JdbcTemplateDemo1 {

// 2.修改操作

@Test

public void update() {

// 1.设置数据库相关信息

DriverManagerDataSource dataSource = new DriverManagerDataSource(); // 数据源

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///spring\_lee");

dataSource.setUsername("root");

dataSource.setPassword("yezi");

// 实现修改操作

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

String sql = "update user set password=? where username=?";

int rows = jdbcTemplate.update(sql, "9999", "liayun");

System.out.println(rows);

}

}

## 删除操作

现在要在单元测试类中编写一个delete方法来测试删除操作。

public class JdbcTemplateDemo1 {

// 3.删除操作

@Test

public void delete() {

// 1.设置数据库相关信息

DriverManagerDataSource dataSource = new DriverManagerDataSource(); // 数据源

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///spring\_lee");

dataSource.setUsername("root");

dataSource.setPassword("yezi");

// 实现删除操作

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

String sql = "delete from user where username=?";

int rows = jdbcTemplate.update(sql, "liayun");

System.out.println(rows);

}

}

## 查询操作

### 查询表中的记录数

现在要在单元测试类中编写一个testCount方法来测试查询表中记录数的操作。

public class JdbcTemplateDemo1 {

// 查询表记录数

@Test

public void testCount() {

// 1.设置数据库相关信息

DriverManagerDataSource dataSource = new DriverManagerDataSource(); // 数据源

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///spring\_lee");

dataSource.setUsername("root");

dataSource.setPassword("yezi");

// 2.创建JdbcTemplate模板类的对象

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

// 3.sql语句

String sql = "select count(\*) from user";

// 4.调用JdbcTemplate模板类里面的方法

// 返回int类型

int count = jdbcTemplate.queryForObject(sql, Integer.class);

System.out.println(count);

}

}

在查询表中记录数的操作时，用到了JdbcTemplate模板类里面的queryForObject方法，如下：   
这里写图片描述  
这个方法中有两个参数：

* 第一个参数：sql语句
* 第二个参数：返回类型的class

### 查询返回对象

使用JdbcTemplate模板类进行查询操作的时候，还是比较麻烦的。前面我也说过JdbcTemplate对JDBC进行了简单封装，使用类似于dbutils，但是使用并没有dbutils方便，只是提供了一种实现的方式而已。我是为何这么说呢？因为在dbutils里面帮我们编写好了一些实现类，使用这些实现类可以封装结果，这些实现类都实现了接口ResultSetHandler；使用JdbcTemplate模板类进行查询操作返回数据结果的时候，虽然在JdbcTemplate模板类中有个接口，但是并没有提供实现类，故还需要自己编写实现类来封装结果。   
好了，接下来复习一下编写JDBC最原始的代码做查询操作，基本功不要忘记了啊！   
首先在cn.itcast.jdbcTemplate包下编写一个User类。

public class User {

private String username;

private String password;

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

@Override

public String toString() {

return "User [username=" + username + ", password=" + password + "]";

}

}

然后再在该包下编写一个JdbcTemplateDemo2单元测试类，编写JDBC最原始的代码做查询操作。

public class JdbcTemplateDemo2 {

// jdbc最原始的代码做查询操作

@Test

public void testJDBC() {

Connection conn = null;

PreparedStatement psmt = null;

ResultSet rs = null;

try {

Class.forName("com.mysql.jdbc.Driver");

conn = DriverManager.getConnection("jdbc:mysql:///spring\_lee", "root", "yezi");

String sql = "select \* from user where username=?";

// 对sql进行预编译操作

psmt = conn.prepareStatement(sql);

psmt.setString(1, "mary");

// 执行sql

rs = psmt.executeQuery();

// 遍历结果

while (rs.next()) {

String username = rs.getString("username");

String password = rs.getString("password");

User user = new User();

user.setUsername(username);

user.setPassword(password);

System.out.println(user);

}

} catch (Exception e) {

} finally {

try {

rs.close();

psmt.close();

conn.close();

} catch (Exception e2) {

}

}

}

}

接下来，就来讲如何使用JdbcTemplate模板类进行查询操作并返回一个User类的对象。其中就要用到JdbcTemplate模板类里面的queryForObject方法：   
这里写图片描述  
这个方法有3个参数：

* 第一个参数：sql语句
* 第二个参数：RowMapper接口
  1. 之前使用dbutils进行查询时，返回结果有ResultSetHandler接口，但是在dbutils里面有其对应的实现类。
  2. 使用JdbcTemplate模板类的时候，虽然提供了RowMapper接口，但是这个接口没有实现类，需要自己进行实现，然后进行数据封装。
* 第三个参数：可变参数

现在要在JdbcTemplateDemo2单元测试类中编写一个testObject方法来测试查询时返回一个User类对象的操作。

public class JdbcTemplateDemo2 {

// 1.查询返回对象

@Test

public void testObject() {

// 1.设置数据库相关信息

DriverManagerDataSource dataSource = new DriverManagerDataSource(); // 数据源

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///spring\_lee");

dataSource.setUsername("root");

dataSource.setPassword("yezi");

// 2.创建JdbcTemplate对象

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

// 3.sql语句

String sql = "select \* from user where username=?";

// 4.调用方法执行

// 第二个参数是接口，RowMapper，要自己创建一个类实现这个接口，在里面进行封装

User user = jdbcTemplate.queryForObject(sql, new MyRowMapper(), "mary");

System.out.println(user);

}

}

注意，还要编写RowMapper接口的一个实现类。在以上JdbcTemplateDemo2.java源代码文件中编写RowMapper接口的一个MyRowMapper实现类。

// T：表示要将数据封装到哪个类的对象中

class MyRowMapper implements RowMapper<User> {

// 实现接口里面的方法，在方法中实现数据封装

// 第一个参数：返回的结果集，第二个参数是当前的行数(即第几行)

@Override

public User mapRow(ResultSet rs, int rows) throws SQLException {

// 从结果集得到数据

String username = rs.getString("username");

String password = rs.getString("password");

// 封装到对象里面

User user = new User();

user.setUsername(username);

user.setPassword(password);

return user;

}

}

### 查询返回List集合

现在要在JdbcTemplateDemo2单元测试类中编写一个testList方法来测试查询时返回List集合的操作。

public class JdbcTemplateDemo2 {

// 2.返回List集合

@Test

public void testList() {

// 1.设置数据库相关信息

DriverManagerDataSource dataSource = new DriverManagerDataSource(); // 数据源

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///spring\_lee");

dataSource.setUsername("root");

dataSource.setPassword("yezi");

// 2.查询操作

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

String sql = "select \* from user";

List<User> list = jdbcTemplate.query(sql, new MyRowMapper());

System.out.println(list);

}

}

在进行查询并返回List集合的操作时，须用到JdbcTemplate模板类里面的query方法：   
这里写图片描述  
或   
这里写图片描述

# Spring配置连接池

在实际开发中，一般都会用Spring配置C3P0连接池，所以下面我就来重点介绍在Spring中如何配置C3P0连接池。   
首先引入Spring的配置文件，主要是引入约束：

<?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:aop="http://www.springframework.org/schema/aop"

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/aop

http://www.springframework.org/schema/aop/spring-aop.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd">

</beans>

接着导入Spring的基本jar包，除此之外，还要导入C3P0的jar包：   
  
试想要是以前在Web项目中配置C3P0连接池，势必会写这样的代码：

ComboPooledDataSource dataSource = new ComboPooledDataSource();

dataSource.setDriverClass(driverClass);

dataSource.setJdbcUrl(jdbcUrl);

dataSource.setUser(user);

dataSource.setPassword(password);

而现在我们就可以在Spring配置文件(bean2.xml)中配置C3P0连接池了，即在Spring配置文件中添加如下配置：

<!-- 配置C3P0连接池 -->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"></property>

<property name="jdbcUrl" value="jdbc:mysql:///spring\_lee"></property>

<property name="user" value="root"></property>

<property name="password" value="yezi"></property>

</bean>

现在举例来演示如何在Spring中配置C3P0连接池了。我的思路是这样的：**创建一个UserService类和一个UserDao类，然后在UserService类里面调用UserDao类的方法，在UserDao类中使用JdbcTemplate模板类进行数据库CRUD操作，并且用上C3P0连接池**。   
先在src目录下创建一个cn.itcast.c3p0包，并在该包下编写一个UserDao类。

public class UserDao {

// 在Dao里面要得到JdbcTemplate对象

private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

// 添加操作，使用JdbcTemplate模板来实现添加

public void add() {

String sql = "insert into user values(?,?)";

jdbcTemplate.update(sql, "李阿昀", "lee");

}

}

再在该包下编写一个UserService类，并在UserService类里面调用UserDao类的add方法。

public class UserService {

private UserDao userDao;

public void setUserDao(UserDao userDao) {

this.userDao = userDao;

}

public void add() {

userDao.add();

}

}

那么Spring核心配置文件就应该像下面这样配置：

<!-- 配置service和dao以及它们的注入 -->

<bean id="userService" class="cn.itcast.c3p0.UserService">

<property name="userDao" ref="userDao"></property>

</bean>

<bean id="userDao" class="cn.itcast.c3p0.UserDao">

<property name="jdbcTemplate" ref="jdbcTemplate"></property>

</bean>

<!-- 配置JdbcTemplate模板类的对象 -->

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

<!-- 注入dataSource，因为在其源代码中dataSource属性有其对应的set方法，故可直接注入 -->

<property name="dataSource" ref="dataSource"></property>

</bean>

从上面的配置可看出：UserDao中注入了JdbcTemplate对象，JdbcTemplate对象里面又注入了dataSource。   
最后再在该包下编写一个TestDemo单元测试类。

public class TestDemo {

@Test

public void testBook() {

ApplicationContext context = new ClassPathXmlApplicationContext("bean2.xml");

UserService userService = (UserService) context.getBean("userService");

userService.add();

}

}

# 事务的回顾

## 什么是事务

事务是逻辑上的一组操作，组成这组操作的各个逻辑单元，要么一起成功，要么一起失败。

## 事务的特性

* 原子性：强调事务的不可分割。
* 一致性：事务的执行的前后数据的完整性保持一致。
* 隔离性：一个事务执行的过程中，不应该受到其他事务的干扰。
* 持久性：事务一旦结束，数据就持久化到数据库。

## 如果不考虑隔离性会引发的安全性问题

* 脏读：一个事务读到了另一个事务的未提交的数据。
* 不可重复读：一个事务读到了另一个事务已经提交的update的数据，导致多次查询的结果不一致。
* 虚读：一个事务读到了另一个事务已经提交的insert的数据，导致多次查询的结果不一致。

## 解决读问题：设置事务的隔离级别

* 未提交读：脏读、不可重复读和虚读都有可能发生。
* 已提交读：避免脏读，但是不可重复读和虚读有可能发生。
* 可重复读：避免脏读和不可重复读，但是虚读有可能发生。
* 串行化的：避免以上所有读问题。   
  **mysql数据库的默认隔离级别就是可重复读**。

# Spring进行事务操作常用的API

## PlatformTransactionManager：平台事务管理器

Spring进行事务操作时候，主要使用一个PlatformTransactionManager接口，它表示事务管理器，即真正管理事务的对象。   
Spring针对不同的持久化框架，提供了不同PlatformTransactionManager接口的实现类，如下：   


## TrancactionDefinition：事务定义信息

事务定义信息有：

* 隔离级别
* **传播行为**
* 超时信息
* 是否只读

## TrancactionStatus：事务的状态

记录事务的状态。

## Spring的这组接口是如何进行事务管理的

平台事务管理器根据事务定义的信息进行事务的管理，事务管理的过程中产生一些状态，将这些状态记录到TrancactionStatus里面。

## 事务的传播行为

PROPAGION\_XXX：事务的传播行为。

* 保证在同一个事务中   
  **PROPAGION\_REQUIRED**：支持当前事务，如果不存在，就新建一个(**默认**)   
  PROPAGION\_SUPPORTS：支持当前事务，如果不存在，就不使用事务   
  PROPAGION\_MANDATORY：支持当前事务，如果不存在，就抛出异常
* 保证没有在同一个事务中   
  **PROPAGION\_REQUIRES\_NEW**：如果有事务存在，挂起当前事务，创建一个新的事务   
  PROPAGION\_NOT\_SUPPORTED：以非事务方式运行，如果有事务存在，挂起当前事务   
  PROPAGION\_NEVER：以非事务方式运行，如果有事务存在，抛出异常   
  **PROPAGION\_NESTED**：如果当前事务存在，则嵌套事务执行

关于事务的传播行为，我真的是一点都不了解啊！希望随着时间的推移，能够真心明白。

## Spring的声明式事务管理方式

Spring进行声明式事务配置的方式有两种：

1. 基于xml配置文件方式
2. 基于注解方式

这两种方式我都会讲解，但无论使用什么方式进行Spring的事务操作，首先要配置一个事务管理器。

# 搭建转账的环境

现在我举例来演示Spring如何进行声明式事务的配置。例子就是模拟银行转账，首先要搭建好转账的环境。   
第一步，创建数据库表。

DROP TABLE IF EXISTS `account`;

CREATE TABLE `account` (

`id` int(11) DEFAULT NULL,

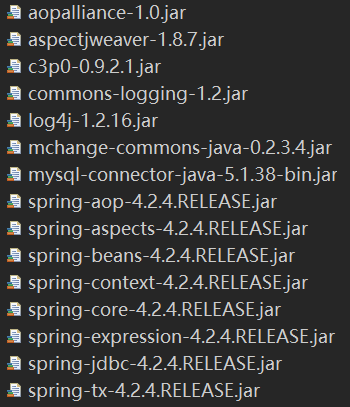
`username` varchar(100) DEFAULT NULL,

`salary` int(11) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `account` VALUES ('1', '小郑', '10000');

INSERT INTO `account` VALUES ('2', '小谭', '10000');

第二步，创建一个Web项目，并引入Spring的相关jar包。   
  
第三步，创建业务层和DAO层的类。   
在Web项目的src目录下创建一个cn.itcast.tx包，并在该包下编写业务层和DAO层的类。

* 业务层——BookService.java
* public class BookService {
* private BookDao bookDao;
* public void setBookDao(BookDao bookDao) {
* this.bookDao = bookDao;
* }

}

* DAO层——BookDao.java
* public class BookDao {
* private JdbcTemplate jdbcTemplate;
* public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
* this.jdbcTemplate = jdbcTemplate;
* }

}

第四步，配置业务层和DAO层的类。

<?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:aop="http://www.springframework.org/schema/aop"

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/aop

http://www.springframework.org/schema/aop/spring-aop.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd">

<!-- 配置C3P0连接池 -->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"></property>

<property name="jdbcUrl" value="jdbc:mysql:///spring\_lee"></property>

<property name="user" value="root"></property>

<property name="password" value="yezi"></property>

</bean>

<!-- 创建service和dao的对象 -->

<bean id="bookService" class="cn.itcast.tx.BookService">

<!-- 注入dao -->

<property name="bookDao" ref="bookDao"></property>

</bean>

<bean id="bookDao" class="cn.itcast.tx.BookDao">

<!-- 注入JdbcTemplate模板类的对象 -->

<property name="jdbcTemplate" ref="jdbcTemplate"></property>

</bean>

<!-- 创建JdbcTemplate模板类的对象 -->

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

<!-- 注入dataSource -->

<property name="dataSource" ref="dataSource"></property>

</bean>

</beans>

第五步，转账的具体实现，实现小郑转账1000元给小谭。   
JavaEE中DAO层做的事情主要是对数据库进行操作，在DAO层里面一般不写业务操作，一般写单独操作数据库的方法。所以BookDao类的代码要修改为：

public class BookDao {

private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

// 小郑少1000

public void lessMoney() {

String sql = "update account set salary=salary-? where username=?";

jdbcTemplate.update(sql, 1000, "小郑");

}

// 小谭多1000

public void moreMoney() {

String sql = "update account set salary=salary+? where username=?";

jdbcTemplate.update(sql, 1000, "小谭");

}

}

JavaEE中Service层写具体的业务操作，所以BookService类的代码要修改为：

public class BookService {

private BookDao bookDao;

public void setBookDao(BookDao bookDao) {

this.bookDao = bookDao;

}

// 转账的业务

public void accountMoney() {

// 1.小郑少1000

bookDao.lessMoney();

// 2.小谭多1000

bookDao.moreMoney();

}

}

第六步，编写一个测试类。   
在cn.itcast.tx包下编写一个TestDemo单元测试类。

public class TestDemo {

@Test

public void testAccount() {

ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");

BookService bookService = (BookService) context.getBean("bookService");

bookService.accountMoney();

}

}

测试以上方法即可实现小郑转账1000元给小谭。现在我来演示一个问题，在BookService类中调用BookDao类的两个方法构成了转账业务，但是如果小郑少了1000元之后，这时突然出现异常，比如银行断电，就会出现小郑的钱少了，而小谭的钱没有多，钱丢失了的情况。

public class BookService {

private BookDao bookDao;

public void setBookDao(BookDao bookDao) {

this.bookDao = bookDao;

}

// 转账的业务

public void accountMoney() {

// 1.小郑少1000

bookDao.lessMoney();

int x = 10 / 0; // 模拟银行断电的情况(出现的异常)

// 2.小谭多1000

bookDao.moreMoney();

}

}

这时应该怎么解决这个问题呢？就可使用事务来解决。Spring中进行事务的操作主要有两种方式：

1. 第一种：编程式事务管理(这种了解就行，不用掌握)
2. **第二种：声明式事务管理**
   * **基于xml配置文件方式**
   * **基于注解方式**

## Spring的声明式事务管理——XML方式：思想就是AOP

基于xml配置文件的方式来进行声明式事务的操作，不需要进行手动编写代码，通过一段配置完成事务管理。下面我在搭建好的转账环境下演示它。   
第一步，配置事务管理器。   
之前，我就讲过Spring针对不同的持久化框架，提供了不同PlatformTransactionManager接口的实现类，如下：   
  
所以我们需要在Spring的配置文件中添加如下配置：

<!-- 1.配置事务的管理器 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<!-- 指定要对哪个数据库进行事务操作 -->

<property name="dataSource" ref="dataSource"></property>

</bean>

第二步，配置事务的增强，即指定对哪个事务管理器进行增强。故需要向Spring的配置文件中添加如下配置：

<!-- 2.配置事务的增强，指定对哪个事务管理器进行增强 -->

<tx:advice id="txadvice" transaction-manager="transactionManager">

<tx:attributes>

<!--

表示来配置你要增强的方法的匹配的一个规则，

注意：只须改方法的命名规则，其他都是固定的！

propagation：事务的传播行为。

-->

<tx:method name="account\*" propagation="REQUIRED"></tx:method>

<!-- <tx:method name="insert\*" propagation="REQUIRED"></tx:method> -->

</tx:attributes>

</tx:advice>

第三步，配置切入点和切面。这步须向Spring的配置文件中添加如下配置：

<!-- 3.配置切入点和切面(最重要的一步) -->

<aop:config>

<!-- 切入点 -->

<aop:pointcut expression="execution(\* cn.itcast.tx.BookService.\*(..))" id="pointcut1"/>

<!-- 切面，即表示把哪个增强用在哪个切入点上 -->

<aop:advisor advice-ref="txadvice" pointcut-ref="pointcut1"/>

</aop:config>

这时测试TestDemo单元测试类的testAccount方法即可。

## Spring的声明式事务的注解方式

基于注解方式来进行声明式事务的操作会更加简单，在实际开发中我们也会用的比较多。下面我在搭建好的转账环境下演示它。   
第一步，配置事务管理器。

<!-- 1.配置事务管理器 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"></property>

</bean>

第二步，开启事务注解。

<!-- 2.开启事务的注解 -->

<tx:annotation-driven transaction-manager="transactionManager"></tx:annotation-driven>

以上配置添加完毕之后，Spring核心配置文件的内容就为：

<?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:aop="http://www.springframework.org/schema/aop"

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/aop

http://www.springframework.org/schema/aop/spring-aop.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd">

<!-- 配置C3P0连接池 -->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"></property>

<property name="jdbcUrl" value="jdbc:mysql:///spring\_lee"></property>

<property name="user" value="root"></property>

<property name="password" value="yezi"></property>

</bean>

<!-- 1.配置事务管理器 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"></property>

</bean>

<!-- 2.开启事务的注解 -->

<tx:annotation-driven transaction-manager="transactionManager"></tx:annotation-driven>

<!-- 创建service和dao的对象 -->

<bean id="bookService" class="cn.itcast.tx.BookService">

<!-- 注入dao -->

<property name="bookDao" ref="bookDao"></property>

</bean>

<bean id="bookDao" class="cn.itcast.tx.BookDao">

<!-- 注入JdbcTemplate模板类的对象 -->

<property name="jdbcTemplate" ref="jdbcTemplate"></property>

</bean>

<!-- 创建JdbcTemplate模板类的对象 -->

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

<!-- 注入dataSource -->

<property name="dataSource" ref="dataSource"></property>

</bean>

</beans>

第三步，在具体使用事务的方法所在的类上面添加注解：@Transactional。即BookService类应修改为：

@Transactional

public class BookService {

private BookDao bookDao;

public void setBookDao(BookDao bookDao) {

this.bookDao = bookDao;

}

// 转账的业务

public void accountMoney() {

// 1.小郑少1000

bookDao.lessMoney();

int x = 10 / 0; // 模拟银行断电的情况(出现的异常)

// 2.小谭多1000

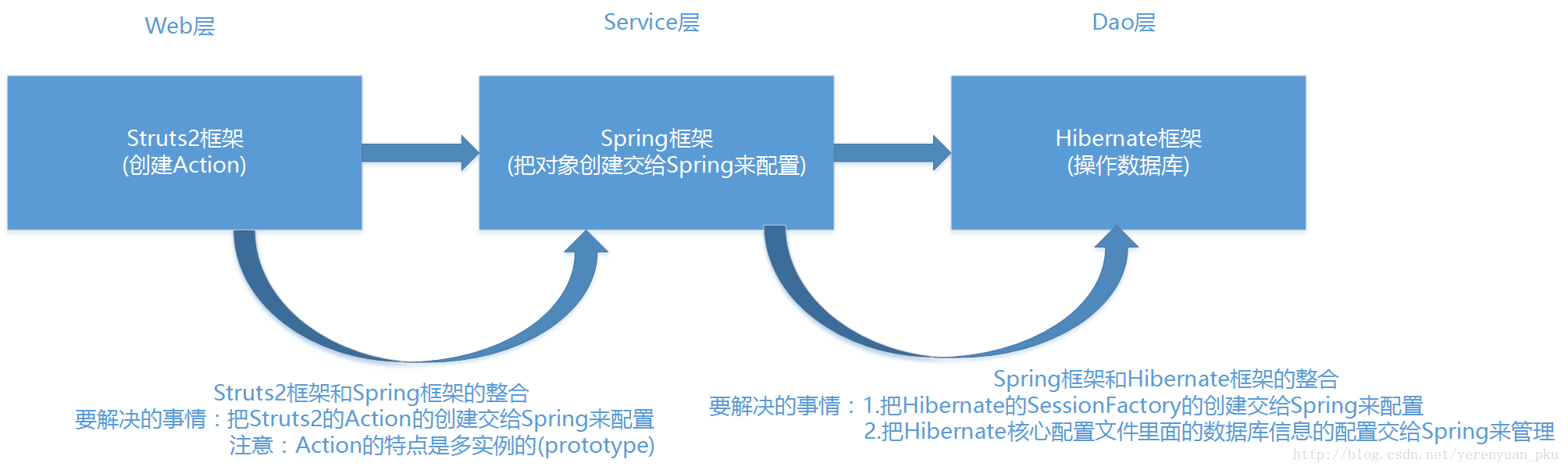
bookDao.moreMoney();

}

}

**注意：千万不要忘记这一步。**   
这时测试TestDemo单元测试类的testAccount方法即可。

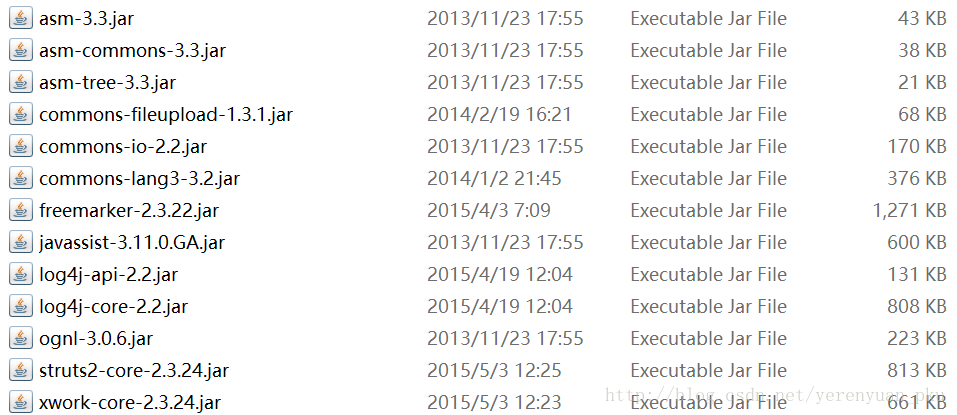
# SSH框架整合思想

三大框架应用在JavaEE三层结构，每一层都用到了不同的框架。SSH框架整合的思想可用下图来表示：   


# SSH框架的整合方式一：零障碍整合(带有Hibernate配置文件)

我们先整合Struts2和Spring这两个框架，再整合Spring和Hibernate这两个框架。

## 整合Struts2和Spring框架

整合Struts2和Spring这两个框架要解决的事情就是把Struts2的Action的创建交给Spring来配置。   
第一步，创建Web项目，导入Struts2框架相关的jar包。   
要进行Struts2的基本的开发，可以参考struts-2.3.24中的apps下的一些示例代码，其中struts2-blank.war是一个Struts2的空的工程。我们只需要将struts2-blank.war解压后进入到WEB-INF下的lib中查看。   
  
除此之外，对于Struts2还需要了解以下jar包：

* struts2-convention-plugin-2.3.24.jar：Struts2注解的开发包
* struts2-json-plugin-2.3.24.jar：Struts2整合AJAX返回JSON数据
* **struts2-spring-plugin-2.3.24.jar**：Struts2整合Spring的插件包。在本文中，我只导入了这个jar包。

第二步，编写一个Action。   
在src目录下创建一个cn.itcast.action包，并在该包下创建一个UserAction。

public class UserAction extends ActionSupport {

@Override

public String execute() throws Exception {

System.out.println("action..........");

return NONE;

}

}

第三步，创建Struts2的核心配置文件，该核心配置文件的位置是在src目录下面，名称是struts.xml。并且在该核心配置文件配置UserAction。

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE struts PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"

"http://struts.apache.org/dtds/struts-2.3.dtd">

<struts>

<package name="demo" extends="struts-default" namespace="/">

<action name="user" class="cn.itcast.action.UserAction"></action>

</package>

</struts>

此时UserAction类由Struts2自己创建。   
**注意：Struts2和Spring框架整合的时候，Struts2核心配置文件的名称和位置是有固定要求的，即名称只能是struts.xml，且必须位于src目录下面。**   
第四步，配置Struts2的过滤器。即在web.xml配置文件中添加如下配置：

<!-- 配置Struts2的过滤器 -->

<filter>

<filter-name>struts2</filter-name>

<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>

</filter>

<filter-mapping>

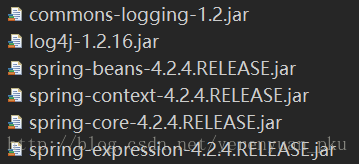
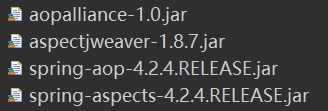
<filter-name>struts2</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

第五步，导入Spring框架相关的jar包。

* Spring框架基本开发的jar包：   
  
* AOP开发相关的jar包：   
  
* 与JDBC开发相关的jar包：   
  这里写图片描述
* 与事务管理的开发相关jar包：   
  这里写图片描述
* 整合Hibernate的jar包：   
  这里写图片描述
* 整和Web项目的jar包：   
  这里写图片描述

第六步，创建Spring的核心配置文件——bean1.xml。

* 引入约束
* <?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:aop="http://www.springframework.org/schema/aop"
* 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/aop
* http://www.springframework.org/schema/aop/spring-aop.xsd
* http://www.springframework.org/schema/tx
* http://www.springframework.org/schema/tx/spring-tx.xsd">

</beans>

* 配置Spring的监听器，即在web.xml配置文件中添加如下配置：
* <listener>
* <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

* 指定Spring配置文件的位置，即在web.xml配置文件中添加如下配置：
* <context-param>
* <param-name>contextConfigLocation</param-name>
* <param-value>classpath:bean1.xml</param-value> <!-- 因为bean1.xml配置文件在src目录下，故可直接这样写 -->

</context-param>

第七步，把UserAction交给Spring进行配置，即在Spring配置文件中添加如下配置：

<!-- 配置action对象 -->

<bean id="userAction" class="cn.itcast.action.UserAction" scope="prototype"></bean>

这时，在Spring配置文件——bean1.xml中配置了UserAction对象，在Struts2配置文件——struts.xml中也配置了UserAction对象。很显然，这种做法是不可取的。解决思路是只需要在Spring配置文件里面配置UserAction对象，而不要在Struts2配置文件中配置。所以，Struts2配置文件应该改为：

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE struts PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"

"http://struts.apache.org/dtds/struts-2.3.dtd">

<struts>

<package name="demo" extends="struts-default" namespace="/">

<!-- 在action标签里面不要写action全路径，class属性值写在Spring配置action对象的bean标签的id值 -->

<action name="user" class="userAction"></action>

</package>

</struts>

到这里，Struts2和Spring框架已经整合好了，现在就需要测试是否真如我们所愿了。   
在src目录下创建一个cn.itcast.dao包，并在该包下编写Dao层的一个类——UserDao.java。

public class UserDao {

public void add() {

System.out.println("dao..............");

}

}

然后在src目录下创建一个cn.itcast.service包，并在该包下编写Service层的一个类——UserService.java，并在该类中调用UserDao类里面的方法。

public class UserService {

private UserDao userDao;

public void setUserDao(UserDao userDao) {

this.userDao = userDao;

}

public void add() {

System.out.println("service..............");

userDao.add();

}

}

接着在UserAction类里面调用UserService类里面的方法。

public class UserAction extends ActionSupport {

private UserService userService;

public void setUserService(UserService userService) {

this.userService = userService;

}

@Override

public String execute() throws Exception {

System.out.println("action..........");

userService.add();

return NONE;

}

}

相应地，Spring的核心配置文件就要修改为：

<?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:aop="http://www.springframework.org/schema/aop"

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/aop

http://www.springframework.org/schema/aop/spring-aop.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd">

<!-- 配置action对象 -->

<bean id="userAction" class="cn.itcast.action.UserAction" scope="prototype">

<property name="userService" ref="userService"></property>

</bean>

<bean id="userService" class="cn.itcast.service.UserService">

<property name="userDao" ref="userDao"></property>

</bean>

<bean id="userDao" class="cn.itcast.dao.UserDao"></bean>

</beans>

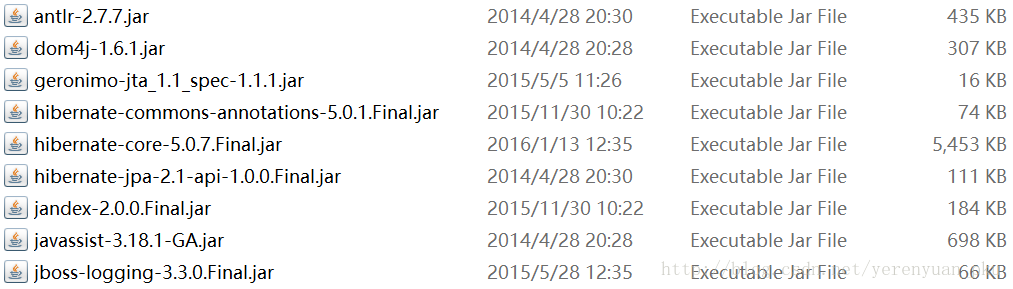
最后，在浏览器地址栏中输入地址http://localhost:8080/Spring\_day04\_demo1/user.action，即可在Eclipse控制台看到如下输出结果：

action……….   
service…………..   
dao…………..

## 整合Spring和Hibernate框架

我之前也说过，要整合Spring和Hibernate框架，须解决下面两件事情：

1. 把Hibernate核心配置文件里面的数据库信息的配置交给Spring来管理。
2. 把Hibernate里面的SessionFactory的创建交给Spring来管理。

下面我就来详细介绍Spring框架整合Hibernate框架的具体实现。   
第一步，导入Hibernate开发相关的jar包。   
首先导入lib/required目录下所有的jar包，因为在该目录下，包含运行Hibernate项目必须的jar包。   
  
再导入mysql数据库的驱动jar包：   
这里写图片描述  
接着再导入日志相关的jar包：   
  
其中log4j的包由Spring导入。   
在进行这一步时，可能会遇到以下两个问题：

1. 导入Struts2和Hibernate的jar包时有jar包冲突问题(两个jar包功能相同，但版本不一样)，即在Struts2里面有一个这样的jar包：   
   这里写图片描述  
   在Hibernate里面也有这样的一个jar包：   
   这里写图片描述  
   解决方式很简单了，就是删除那个低版本的jar包。
2. Spring整合持久化层框架时，还须导入如下jar包：   
   这里写图片描述

第二步，搭建Hibernate开发环境。

1. 创建实体类。在src目录下创建一个cn.itcast.entity包，并在该包下编写一个User实体类。
2. public class User {
3. private Integer uid;
4. private String username;
5. private String address;
6. public Integer getUid() {
7. return uid;
8. }
9. public void setUid(Integer uid) {
10. this.uid = uid;
11. }
12. public String getUsername() {
13. return username;
14. }
15. public void setUsername(String username) {
16. this.username = username;
17. }
18. public String getAddress() {
19. return address;
20. }
21. public void setAddress(String address) {
22. this.address = address;
23. }

}

1. 配置实体类映射关系，即在cn.itcast.entity包中创建User.hbm.xml映射文件：
2. <?xml version="1.0"?>
3. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5. <!-- Generated 2017-4-5 14:44:57 by Hibernate Tools 3.4.0.CR1 -->
6. <hibernate-mapping>
7. <class name="cn.itcast.entity.User" table="t\_user">
8. <id name="uid" type="java.lang.Integer">
9. <column name="uid" />
10. <generator class="native" />
11. </id>
12. <property name="username" type="java.lang.String">
13. <column name="username" />
14. </property>
15. <property name="address" type="java.lang.String">
16. <column name="address" />
17. </property>
18. </class>

</hibernate-mapping>

1. 创建Hibernate核心配置文件，即在src目录下引入hibernate.cfg.xml文件。
2. <?xml version="1.0" encoding="UTF-8"?>
3. <!DOCTYPE hibernate-configuration PUBLIC
4. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
5. "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
6. <hibernate-configuration>
7. <session-factory>
8. <!-- 配置关于数据库连接的四个项：driverClass url username password -->
9. <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>
10. <property name="hibernate.connection.url">jdbc:mysql:///hibernateTest</property>
11. <property name="hibernate.connection.username">root</property>
12. <property name="hibernate.connection.password">yezi</property>
13. <!-- 可以将向数据库发送的SQL语句显示出来 -->
14. <property name="hibernate.show\_sql">true</property>
15. <!-- 格式化SQL语句 -->
16. <property name="hibernate.format\_sql">true</property>
17. <!-- hibernate的方言 -->
18. <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
19. <!-- 自动创建表 -->
20. <property name="hibernate.hbm2ddl.auto">update</property>
21. <mapping resource="cn/itcast/entity/User.hbm.xml" />
22. </session-factory>

</hibernate-configuration>

**注意：如果只是单纯使用Hibernate框架，那么核心配置文件名称必须是hibernate.cfg.xml且位置必须在src目录下。但Hibernate和Spring框架整合的时候，Hibernate核心配置文件的名称和位置是没有固定要求的。**

第三步，把Hibernate核心配置文件中的数据库配置的信息交给Spring进行配置。即把Hibernate核心配置文件中的数据库信息的配置去掉，改在Spring的核心配置文件中配置：

<!-- 配置连接池 -->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"></property>

<property name="jdbcUrl" value="jdbc:mysql:///spring\_lee"></property>

<property name="user" value="root"></property>

<property name="password" value="yezi"></property>

</bean>

第四步，把Hibernate里面的SessionFactory的创建交给Spring来配置。具体思路为：服务器启动的时候，加载Spring配置文件，继而把配置文件中的对象都创建完毕。故可把SessionFactory对象的创建也在Spring配置文件进行配置。又因为创建SessionFactory对象的代码不是简简单单new出来的，而是多行代码实现的，核心代码如下：

// 加载核心配置文件

config = new Configuration().configure();

// 创建SessionFactory对象

sessionFactory = config.buildSessionFactory();

所以我们不可能在Spring配置文件中通过一个<bean>标签就简简单单地进行配置。Spring框架针对上面这种情况，帮我们封装了一个类，配置这个类的对象就可以创建SessionFactory对象了。所以我们应在Spring配置文件中添加如下内容：

<!-- 配置sessionFactory的创建 -->

<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

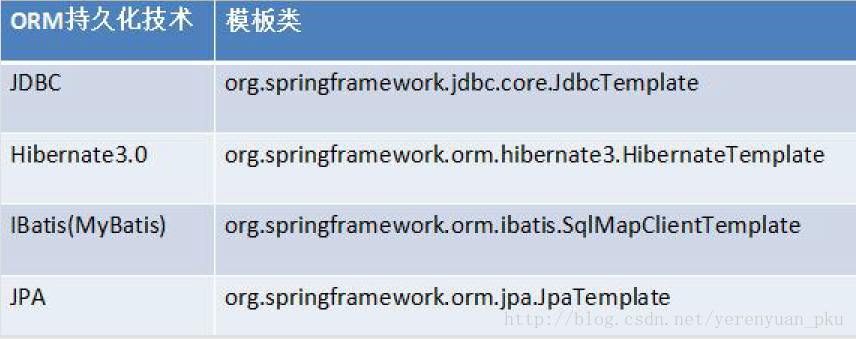
<!-- 1.指定数据库位置 -->

<property name="dataSource" ref="dataSource"></property>

<!-- 2.指定使用的Hibernate核心配置文件位置 -->

<property name="configLocations" value="classpath:hibernate.cfg.xml"></property>

</bean>

第五步，之前我就讲过Spring针对JavaEE三层结构中的每一层都提供了不同的技术，在dao层，Spring提供了JDBC模板的技术，可对数据库进行CRUD操作。Spring提供了很多持久层技术的模板类简化了编程，如下图：   
  
现在我们就要在dao层里面使用HibernateTemplate模板类，至于它是什么，我待会将做简单介绍。具体操作为：向UserDao里面注入HibernateTemplate模板类对象，并调用HibernateTemplate模板类里面的save方法向数据库表中添加一条记录。这样UserDao类的代码就应修改为：

public class UserDao {

// 在Dao里面注入HibernateTemplate模板对象

private HibernateTemplate hibernateTemplate;

public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {

this.hibernateTemplate = hibernateTemplate;

}

public void add() {

// 向表中添加记录

// 使用hibernateTemplate实现添加

// HibernateTemplate hibernateTemplate = new HibernateTemplate(sessionFactory);

// hibernateTemplate.save(entity);

System.out.println("dao..............");

User user = new User();

user.setUsername("李阿昀");

user.setAddress("湖北天门");

hibernateTemplate.save(user);

}

}

当然了，Spring配置文件的修改是必不可少的啦！所以应向Spring配置文件添加如下配置：

<bean id="userDao" class="cn.itcast.dao.UserDao">

<property name="hibernateTemplate" ref="hibernateTemplate"></property>

</bean>

<!-- 创建HibernateTemplate模板对象 -->

<bean id="hibernateTemplate" class="org.springframework.orm.hibernate5.HibernateTemplate">

<!-- 注入sessionFactory -->

<property name="sessionFactory" ref="sessionFactory"></property>

</bean>

第六步，配置Spring的事务管理。

1. 配置事务管理器
2. <!-- 1.配置事务的管理器 -->
3. <bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">
4. <property name="sessionFactory" ref="sessionFactory"></property>

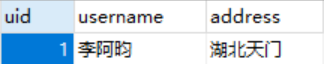
</bean>

1. 注解事务管理的开启
2. <!-- 2.开启事务注解 -->

<tx:annotation-driven transaction-manager="transactionManager"></tx:annotation-driven>

1. 在业务层添加一个@Transactional注解
2. @Transactional
3. public class UserService {
4. private UserDao userDao;
5. public void setUserDao(UserDao userDao) {
6. this.userDao = userDao;
7. }
8. public void add() {
9. System.out.println("service..............");
10. userDao.add();
11. }

}

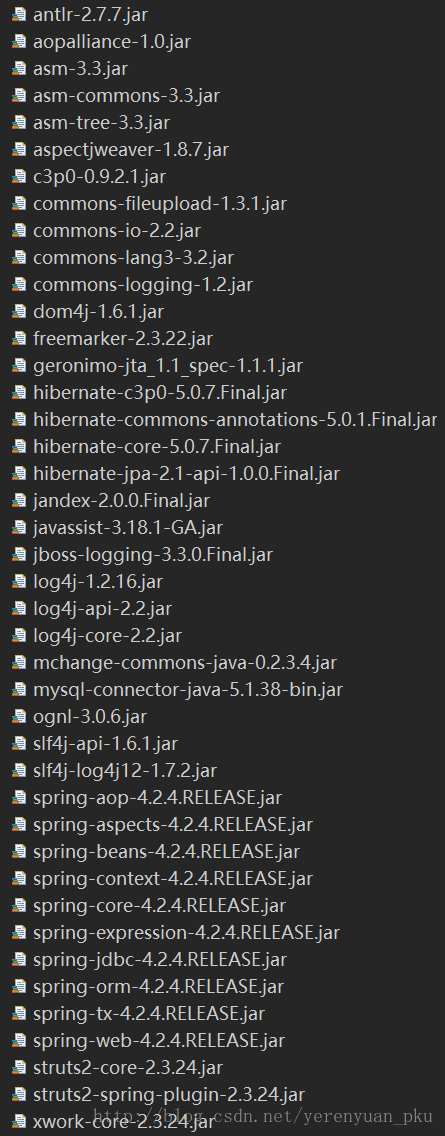
这时启动tomcat服务器，就会发现数据库中多了一个t\_user表，这是理所应当的啦！接着在浏览器地址栏中输入地址http://localhost:8080/Spring\_day04\_demo1/user.action进行访问，即可看到t\_user表多了一条记录，如下：   
  
至此，hibernate-5.0.7+struts-2.3.24+spring-4.2.4三大框架就已整合完毕！

## HibernateTemplate介绍

HibernateTemplate对Hibernate框架进行了封装，直接调用HibernateTemplate里面的方法即可实现对数据库进行CRUD的操作。   
HibernateTemplate常用的方法如下：

* Serializable save(Object entity)：添加操作
* void update(Object entity)：修改操作
* void delete(Object entity)：删除操作
* <T> T get(Class<T> entityClass, Serializable id)：根据id查询
* <T> T load(Class<T> entityClass, Serializable id)：根据id查询
* List find(String queryString, Object... values)：查询操作的方法

## SSH框架整合过程总结

1. 导入相关的jar包。   
     
   一共有42个jar包。
2. 搭建Struts2的开发环境
   * 创建Action，引入struts.xml配置文件，并配置Action
   * 配置Struts2框架的过滤器
3. 搭建Hibernate的开发环境
   * 创建实体类
   * 配置实体类和数据库表之间的映射关系
   * 创建Hibernate核心配置文件，主要是引入映射配置文件
4. 搭建Spring的开发环境
   * 创建Spring核心配置文件
   * 让Spring配置文件在服务器启动时候进行加载
     + 配置监听器
     + 指定Spring配置文件的位置
5. Struts2和Spring框架的整合
   * 把Struts2的Action的创建交给Spring来配置(记住Action是多实例的对象)。
   * 在struts.xml配置文件中的action标签的class属性里面写上bean的id值。
6. Spring和Hibernate框架的整合
   * 把Hibernate核心配置文件里面的数据库信息的配置交给Spring来管理。
   * 把Hibernate里面的SessionFactory的创建交给Spring来管理。
7. 在dao里面使用HibernateTemplate模板类
   * 在dao里面注入HibernateTemplate模板类对象
   * 在HibernateTemplate模板对象中注入SessionFactory
8. 配置Spring的事务管理

# SSH框架的整合方式二：不带Hibernate配置文件

Spring整合Hibernate框架的时候，可以不用写Hibernate的核心配置文件。下面我就举个例子来演示。   
首先复制以上SSH项目，再在其基础上进行修改。   
然后查看之前写的Hibernate核心配置文件，可知Hibernate核心配置文件包含如下内容：

1. 连接数据库必要的参数
2. Hibernate的属性
3. 连接池的配置
4. 映射文件的引入

Spring整合Hibernate框架的时候，虽然可以不用写Hibernate的核心配置文件，但还是要把Hibernate核心配置文件中的基本信息配置和映射文件的引入放到Spring配置文件中进行配置。所以Spring配置文件的内容应修改为：

<?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:aop="http://www.springframework.org/schema/aop"

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/aop

http://www.springframework.org/schema/aop/spring-aop.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd">

<!-- 配置连接池 -->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"></property>

<property name="jdbcUrl" value="jdbc:mysql:///spring\_lee"></property>

<property name="user" value="root"></property>

<property name="password" value="yezi"></property>

</bean>

<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

<property name="dataSource" ref="dataSource"></property>

<!-- <property name="configLocations" value="classpath:hibernate.cfg.xml"></property> -->

<!-- 如果不写Hibernate的核心配置文件，你就不需要写上面这行配置了 -->

<!-- 1.把Hibernate的基本信息的配置(写在这儿) -->

<property name="hibernateProperties">

<props>

<prop key="hibernate.show\_sql">true</prop>

<prop key="hibernate.format\_sql">true</prop>

<prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>

<prop key="hibernate.hbm2ddl.auto">update</prop>

</props>

</property>

<!-- 2.把映射文件引入 -->

<property name="mappingResources">

<list>

<value>cn/itcast/entity/User.hbm.xml</value>

<!-- <value>...</value> -->

</list>

</property>

</bean>

<bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">

<property name="sessionFactory" ref="sessionFactory"></property>

</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>

<bean id="userAction" class="cn.itcast.action.UserAction" scope="prototype">

<property name="userService" ref="userService"></property>

</bean>

<bean id="userService" class="cn.itcast.service.UserService">

<property name="userDao" ref="userDao"></property>

</bean>

<bean id="userDao" class="cn.itcast.dao.UserDao">

<property name="hibernateTemplate" ref="hibernateTemplate"></property>

</bean>

<bean id="hibernateTemplate" class="org.springframework.orm.hibernate5.HibernateTemplate">

<property name="sessionFactory" ref="sessionFactory"></property>

</bean>

</beans>

至此，hibernate-5.0.7+struts-2.3.24+spring-4.2.4三大框架就又整合完毕！两种方式任君选择！

# Activity工作流

# 工作流的概念

先看下面两张图：   


  
对以上两张图进行说明：

1. 假设这两张图就是华谊兄弟的请假流程图
2. 图的组成部分：
   * 人物：范冰冰、冯小刚、王中军
   * 事件（动作）：请假、批准、不批准

通过以上分析我们就可以抽象成：   
  
接下来给出工作流的书面化概念：

工作流(Workflow)，就是“业务过程的部分或整体在计算机应用环境下的自动化”，它主要解决的是“使在多个参与者之间按照某种预定义的规则传递文档、信息或任务的过程自动进行，从而实现某个预期的业务目标，或者促使此目标的实现”。

对于第一次接触工作流的小伙伴来说，觉得难以理解，也无可厚非，说得好像我自己就能深刻理解一样，我也只是将学习Activiti工作流框架中的一些知识点记录下来而已，也希望能和大家讨论。

不管了，下面也给出工作流管理系统的概念：

工作流管理系统(Workflow Management System, WfMS)是一个软件系统，它完成工作量的定义和管理，并按照在系统中预先定义好的工作流逻辑进行工作流实例的执行。工作流管理系统不是企业的业务系统，而是为企业的业务系统的运行提供了一个软件的支撑环境。

除此之外，工作流管理联盟(WfMC，Workflow Management Coalition)也给出了关于工作流管理系统的定义：

工作流管理系统是一个软件系统，它通过执行经过计算的流程定义去支持一批专门设定的业务流程。工作流管理系统被用来定义、管理和执行工作流程。

而工作流管理系统的目标为：

管理工作的流程以确保工作在正确的时间被期望的人员所执行——在自动化进行的业务过程中插入人工的执行和干预。

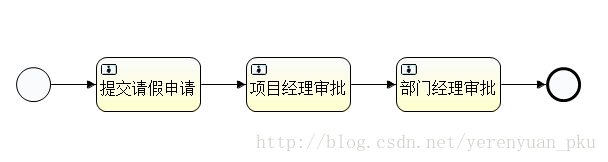
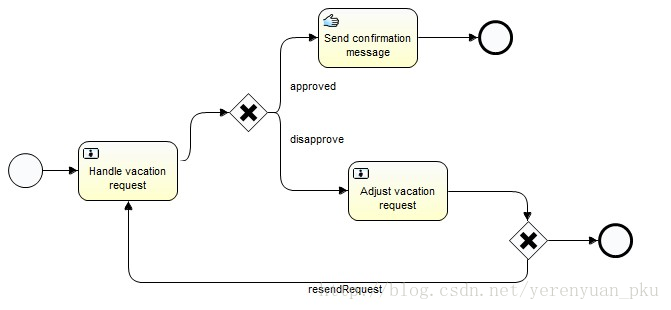
说完工作流，不可避免就要阐述一下工作流框架的概念，**工作流框架即用于处理工作流相关问题的框架**。常见的工作流框架有：

1. activiti5.13
2. JBPM4.4
3. OSWorkflow

我本人使用的是activiti5.13这个工作流框架。

# Activiti介绍

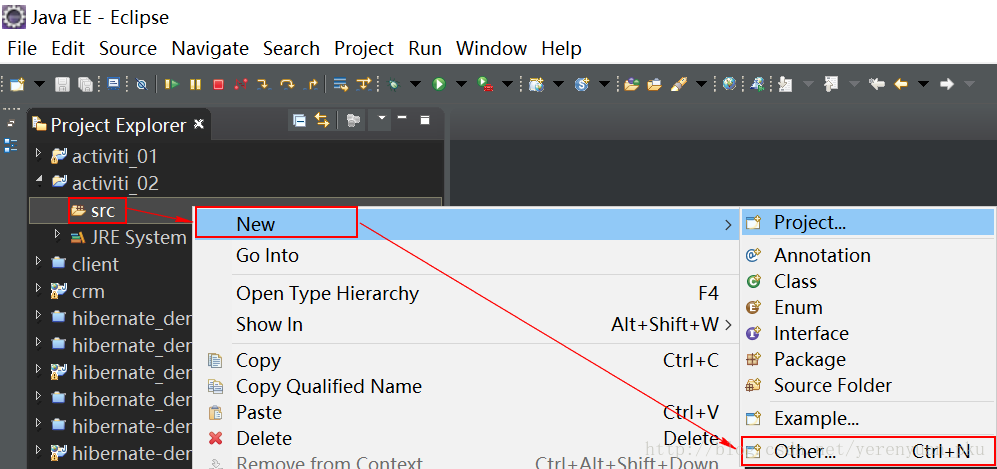
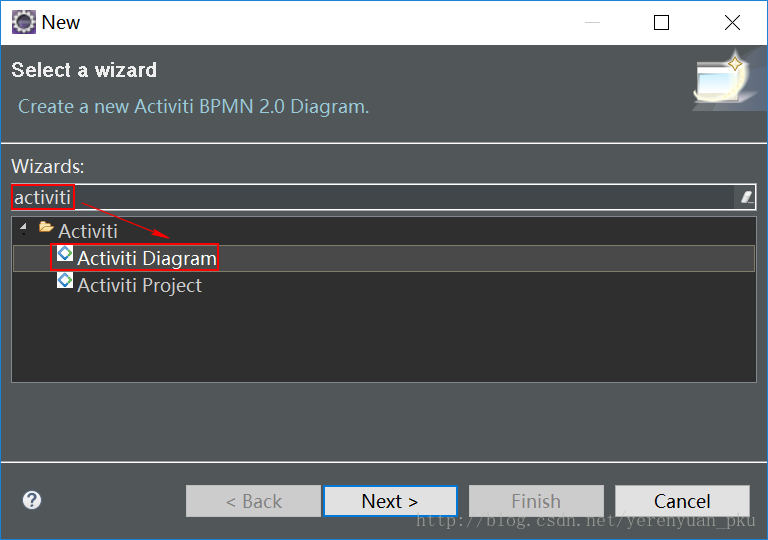
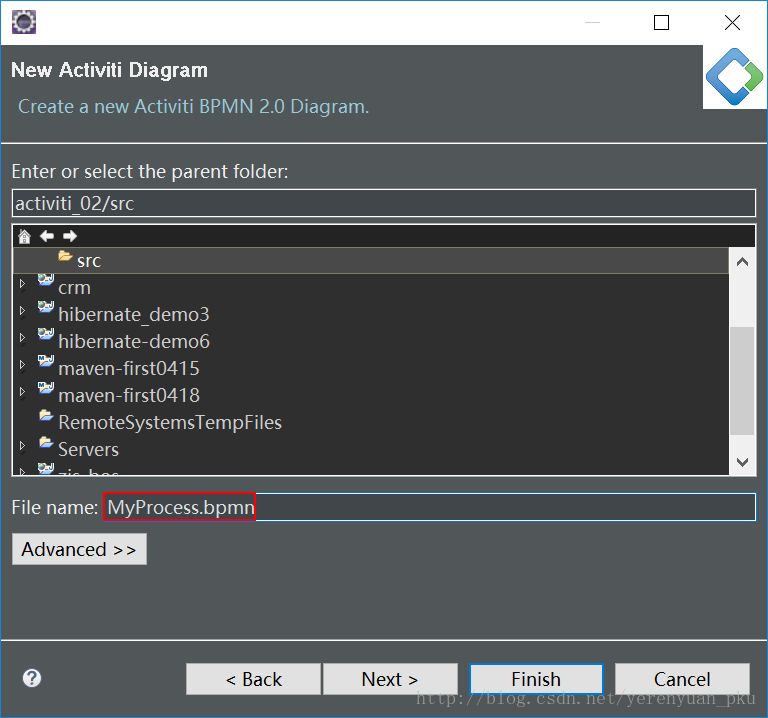
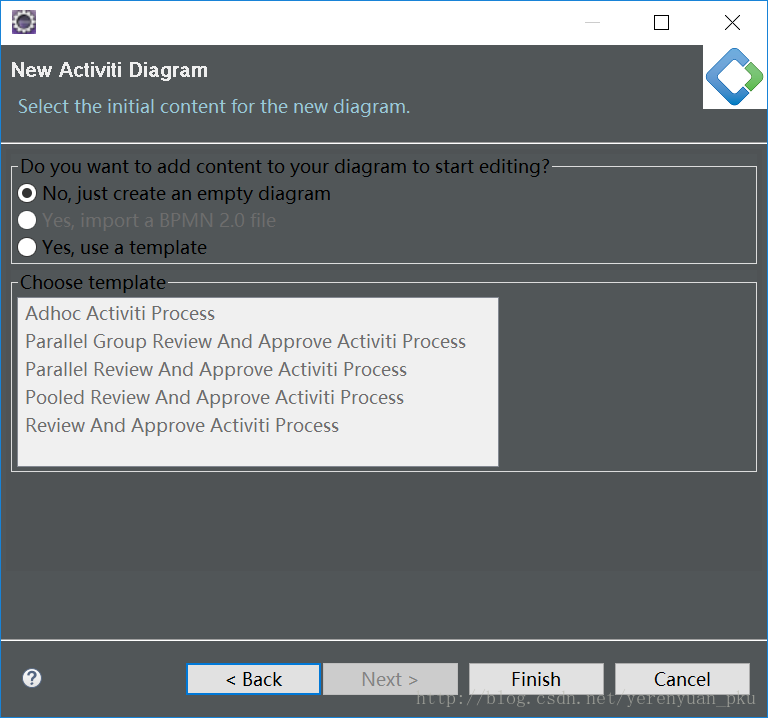
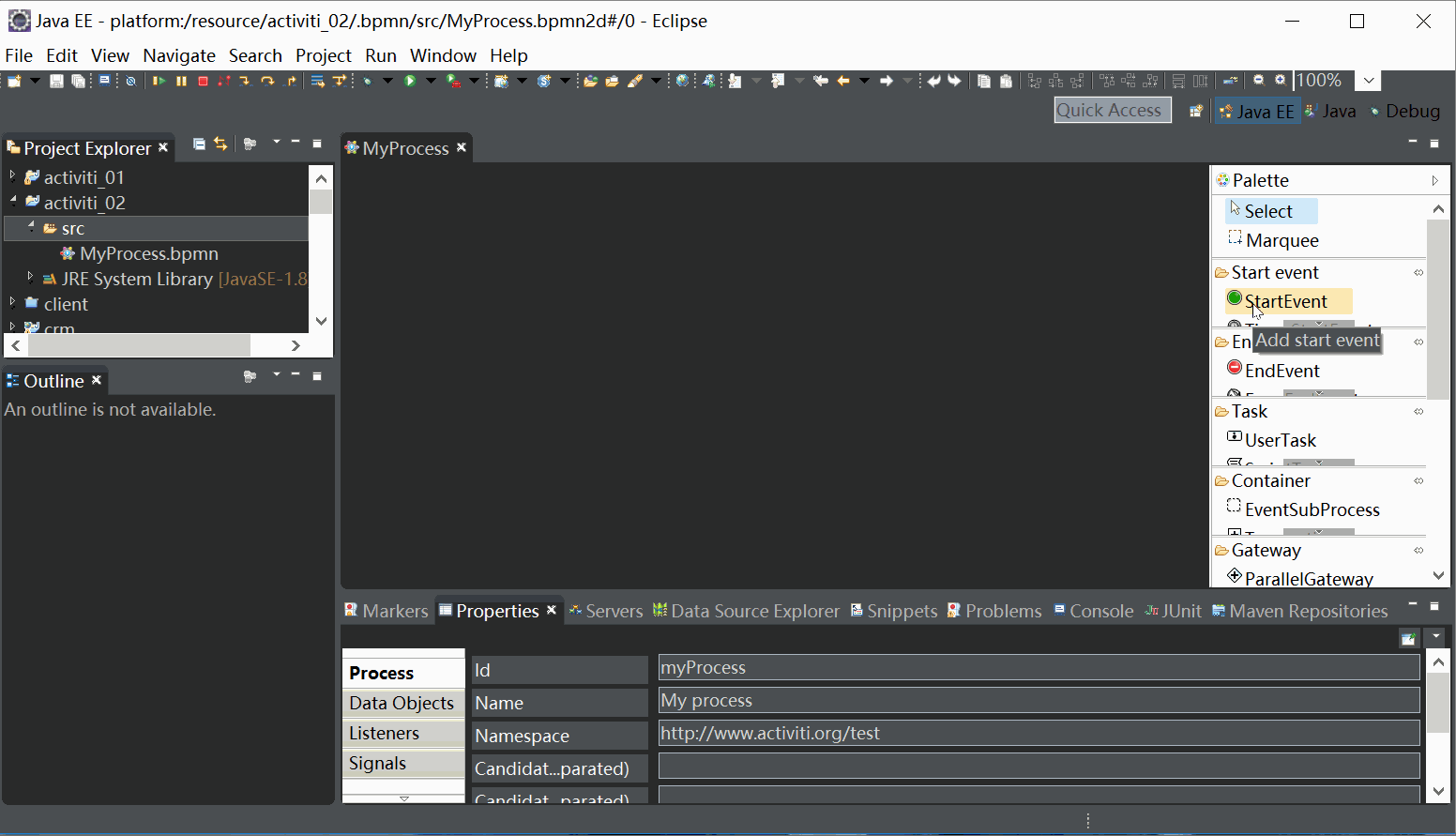
Activiti5是由Alfresco软件在2010年5月17日发布的业务流程管理（BPM）框架，它是覆盖了业务流程管理、工作流、服务协作等领域的一个开源的、灵活的、易扩展的可执行流程语言框架。Activiti基于Apache许可的开源BPM平台，创始人Tom Baeyens是JBoss JBPM的项目架构师，它的特色是提供了eclipse插件，开发人员可以通过插件直接绘画出业务流程图。   
大家可能听说过业务流程图一嘴，这里给出两个业务流程图：

* 请假流程图   
  
* 不知道什么情况的业务流程图   
  

## Activiti框架的目录结构

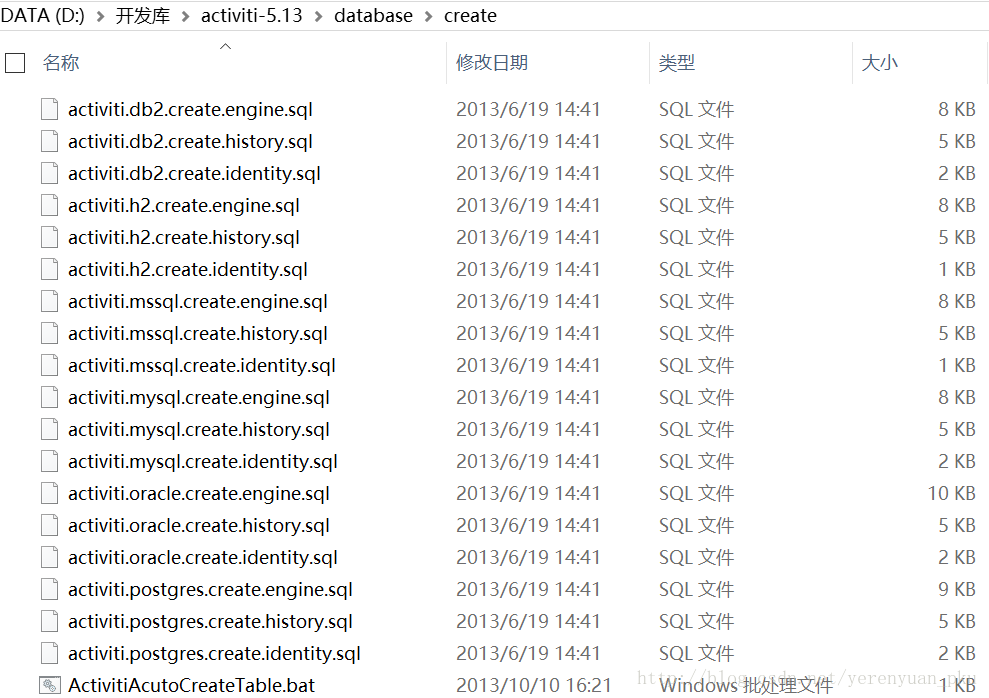
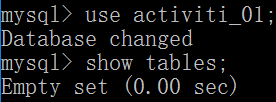
Activiti框架的目录结构如下图所示：   
  
你会发现bin目录是空的。database目录下有3个目录，一个目录是create，里面存放的是建表语句，一个目录是drop，里面存放的是删除表的语句，最后一个目录是upgrade，里面存放的是升级Activiti的语句。libs目录下是一些jar包，最核心的jar包是activiti-engine-5.13.jar。wars目录下存放的是Activiti框架官方的学习demo，我们初次学习Activiti框架必然要借鉴其中的案例。   
我想大家可能会好奇Activiti框架里面为何会有一些建表语句。因为工作流框架底层是有一套数据库提供支持的，针对不同的数据库提供不同的sql建表语句。Activiti5.13框架对应23张表，JBPM4.4框架对应18张表，开发人员不需要自己编写sql语句操作这些表，框架底层会生成sql语句操作。Activiti5.13框架底层使用mybatis框架操作数据库，JBPM框架底层使用hibernate框架操作数据库。

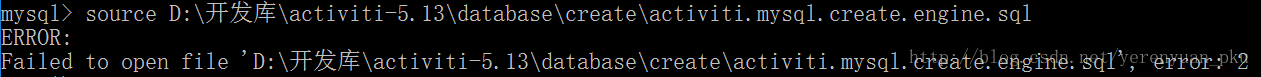
## 安装activiti插件——流程设计器插件

要在eclipse上安装activiti插件，可参考我的文章[4.5版本eclipse安装activiti插件](http://blog.csdn.net/yerenyuan_pku/article/details/71246773)！   
下面我就来用这个插件设计一个请假流程图：   
【第一步】，建一个普通的java项目，例如activiti\_02，在src目录下右键→New→Other...   
  
【第二步】，在弹出的对话框中，在输入项中输入activiti，快速找到Activiti Diagram，选中它，点击Next   
  
【第三步】，在弹出的对话框中，写入流程图的名称，当然了亦可使用默认名称——MyProcess，然后点击Next   
  
【第四步】，在最后弹出的对话框中直接点击Finish   
  
【第五步】，观看以下gif动图，读者即可创建一个请假流程图   


# 创建Activiti框架提供的数据库表

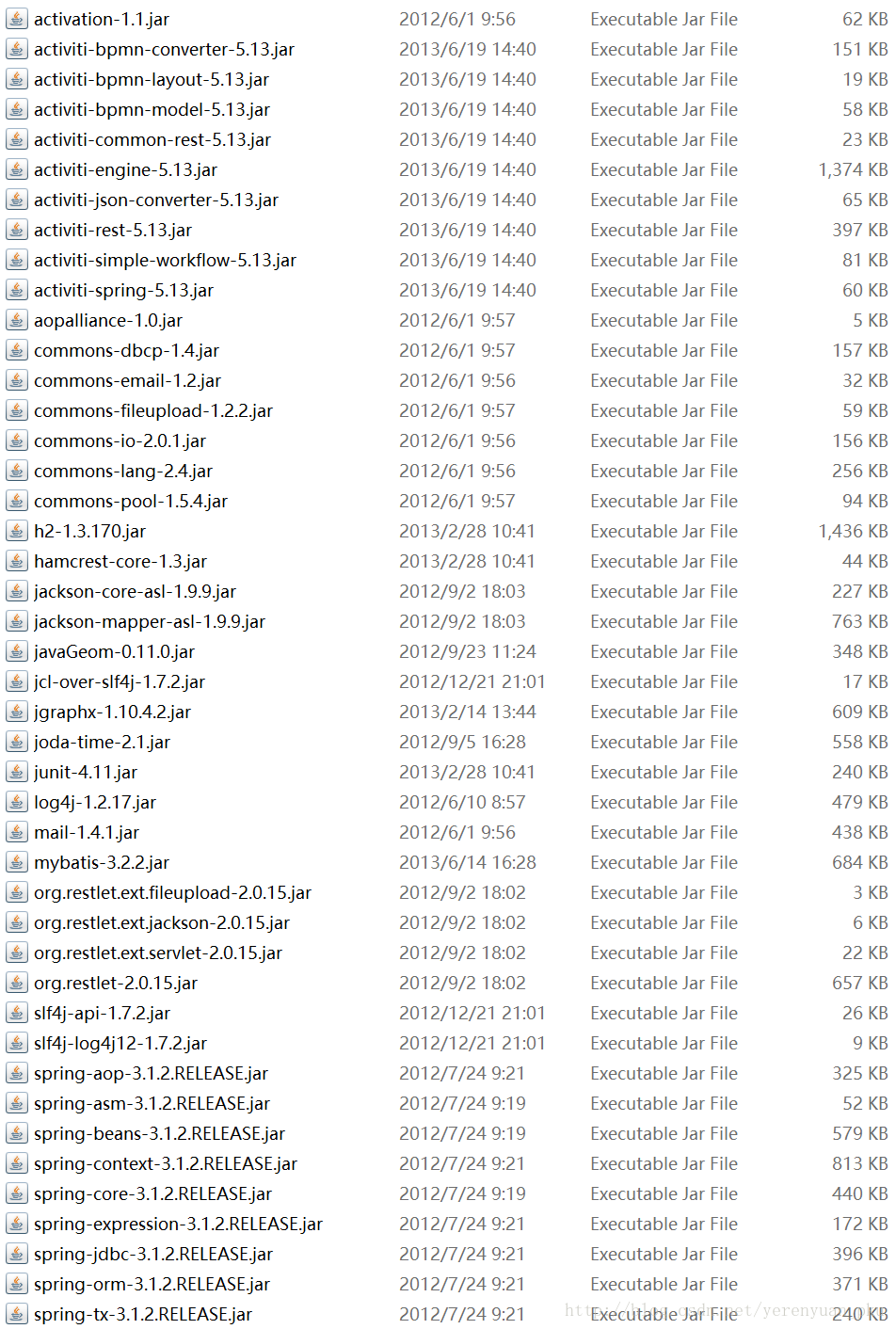
## 使用Activiti框架提供的sql脚本建表

Activiti框架提供了sql脚本文件用于建表，这些sql脚本文件就位于Activiti框架database目录下的create目录中，如下：   
  
下面我就来使用Activiti框架提供的sql脚本建表，步骤如下：   
【第一步】，手动创建一个数据库   
这里写图片描述  
【第二步】，进入数据库，执行框架提供的sql脚本文件   


  
照理来说我们可以使用source命令来执行这些sql脚本文件的，但不知为何，我就是不行。我就没纠结这个问题了，直接将以下三个sql脚本文件拖入Navicat for MySQL图形化工具中。

1. activiti.mysql.create.engine.sql
2. activiti.mysql.create.history.sql
3. activiti.mysql.create.identity.sql

## 使用Activiti框架自动建表

在上面创建好了一个普通的java项目——activiti\_02之后，要使用Activiti框架自动建表，还必须导入Activiti框架所需的jar包，那这些jar包到哪儿去找呢？还记得之前我讲过Activiti框架下的wars目录中存放的是Activiti框架官方的学习demo吗？所以读者可将wars目录中的activiti-rest.war文件解压缩，在activiti-5.13\wars\activiti-rest\WEB-INF\lib目录下可找到Activiti框架所需的所有jar包，如下：   
  
接着在activiti\_02项目下新建一个lib目录，将以上Activiti框架所需的所有jar包导入到lib目录中，除此之外，还要导入MySQL数据库驱动的jar包：   
这里写图片描述  
读者可千万别忘了这个jar包哟！！！

### 在没有提供xml配置文件的情况下使用Activiti框架自动建表

在src目录下创建一个cn.itcast.activiti包，并在该包下编写一个HelloWorld单元测试类，并在该类中编写如下单元测试方法：

public class HelloWorld {

/\*\*

\* 使用activiti框架提供的自动建表方式创建23张表-----没有提供配置文件

\*/

@Test

public void test1() {

// 创建一个流程引擎配置对象

ProcessEngineConfiguration conf =

ProcessEngineConfiguration.createStandaloneProcessEngineConfiguration();

// 设置jdbc连接参数

conf.setJdbcDriver("com.mysql.jdbc.Driver"); conf.setJdbcUrl("jdbc:mysql://localhost:3306/activiti\_01");

conf.setJdbcUsername("root");

conf.setJdbcPassword("yezi");

// 设置自动建表

conf.setDatabaseSchemaUpdate("true");

// 使用配置对象创建一个流程引擎对象，并且在创建过程中可以自动建表

ProcessEngine processEngine = conf.buildProcessEngine();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

执行以上test1方法，即可在activiti\_01数据库中创建好23张表。

### 在提供xml配置文件的情况下使用Activiti框架自动建表

在没有提供xml配置文件的情况时使用Activiti框架自动建表，我是把jdbc连接参数写死在程序中的，想都不要想，这种方式是愚蠢的。更合理的做法是把这些jdbc连接参数配置到一个配置文件中，而不是在java代码中写死。   
在activiti\_02项目下新建一个config源码目录，并在该目录下创建一个activiti-context.xml配置文件，内容如下：

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"

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

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-2.5.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

<!-- 配置一个流程引擎配置对象 -->

<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">

<property name="jdbcDriver" value="com.mysql.jdbc.Driver"></property>

<property name="jdbcUrl" value="jdbc:mysql://localhost:3306/activiti\_01"></property>

<property name="jdbcUsername" value="root"></property>

<property name="jdbcPassword" value="yezi"></property>

<property name="databaseSchemaUpdate" value="true"></property>

</bean>

<!-- 配置一个流程引擎工厂bean，用于创建流程引擎对象 -->

<bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">

<property name="processEngineConfiguration" ref="processEngineConfiguration"></property>

</bean>

</beans>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24

其实，以上id为processEngine的bean可不用配置，当然了配了也没关系，只不过是为后面的学习做铺垫而已，这里无伤大雅啊！   
接着在HelloWorld单元测试类中编写如下单元测试方法：

public class HelloWorld {

/\*\*

\* 使用activiti框架提供的自动建表方式创建23张表-----提供配置文件

\*/

@Test

public void test2() {

// 获得一个流程引擎配置对象

ProcessEngineConfiguration conf = ProcessEngineConfiguration

.createProcessEngineConfigurationFromResource(

"activiti-context.xml", "processEngineConfiguration");

// 使用配置对象创建一个流程引擎对象，并且在创建过程中可以自动建表

ProcessEngine processEngine = conf.buildProcessEngine();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

执行以上test2方法，也可在activiti\_01数据库中创建好23张表。

### 在提供默认配置文件的情况下使用Activiti框架自动建表

在实际开发中，建议在提供默认配置文件的情况下使用Activiti框架自动建表。但须注意：**配置文件必须在类路径的根路径下，配置文件的名称必须为activiti-context.xml或者为activiti.cfg.xml，xml配置文件中必须配置流程引擎配置对象，id必须为processEngineConfiguration，且必须配置流程引擎工厂bean，id必须为processEngine**。   
由此可知，我在config源码目录下编写的activiti-context.xml配置文件完全符合以上要求，activiti-context.xml配置文件的内容为：

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"

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

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-2.5.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

<!-- 配置一个流程引擎配置对象 -->

<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">

<property name="jdbcDriver" value="com.mysql.jdbc.Driver"></property>

<property name="jdbcUrl" value="jdbc:mysql://localhost:3306/activiti\_01"></property>

<property name="jdbcUsername" value="root"></property>

<property name="jdbcPassword" value="yezi"></property>

<property name="databaseSchemaUpdate" value="true"></property>

</bean>

<!-- 配置一个流程引擎工厂bean，用于创建流程引擎对象 -->

<bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">

<property name="processEngineConfiguration" ref="processEngineConfiguration"></property>

</bean>

</beans>

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24

这个activiti-context.xml配置文件就是Activiti核心配置文件，主要配置流程引擎创建工具的基本参数和数据库连接池参数。   
定义数据库配置参数：

* jdbcUrl：数据库的JDBC URL
* jdbcDriver：对应不同数据库类型的驱动
* jdbcUsername：连接数据库的用户名
* jdbcPassword：连接数据库的密码

基于JDBC参数配置的数据库连接会使用默认的MyBatis连接池。下面的参数可以用来配置连接池（来自MyBatis参数）：

* jdbcMaxActiveConnections：连接池中处于被使用状态的连接的最大值。默认为10
* jdbcMaxIdleConnections：连接池中处于空闲状态的连接的最大值
* jdbcMaxCheckoutTime：连接被取出使用的最长时间，超过时间会被强制回收。默认为20000（20秒）
* jdbcMaxWaitTime：这是一个底层配置，让连接池可以在长时间无法获得连接时， 打印一条日志，并重新尝试获取一个连接。（避免因为错误配置导致沉默的操作失败）。默认为20000（20秒）

在这里我给出一个示例数据库配置：   
  
当然了也可以使用javax.sql.DataSource。（比如，Apache Commons的DBCP）：   
  
以上就当作了解，哈哈，我也没这样写过！初次学习Activiti工作流框架的小白也不需要接触到这些配置，真到要用的时候，再回来看呗！   
不说远了，接着再在HelloWorld单元测试类中编写如下单元测试方法：

public class HelloWorld {

/\*\*

\* 使用activiti框架提供的自动建表方式创建23张表-----使用默认配置文件

\*/

@Test

public void test3() {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

执行以上test3方法，同样也可在activiti\_01数据库中创建好23张表。

# 了解Activiti框架提供的23张表

Activiti的后台是有数据库的支持的，所有的表都以ACT\_开头。第二部分是表示用途的两个字母标识。用途也和服务的API对应。

1. ACT\_RE\_\*：’RE’表示repository。这个前缀的表包含了流程定义和流程静态资源 （图片，规则等等）。
2. ACT\_RU\_\*：’RU’表示runtime。这些是运行时的表，包含流程实例，任务，变量，异步任务等运行中的数据。Activiti只在流程实例执行过程中保存这些数据， 在流程结束时就会删除这些记录。这样运行时表可以一直很小且速度很快。
3. ACT\_ID\_\*：’ID’表示identity。这些表包含身份信息，比如用户，组等等。
4. ACT\_HI\_\*：’HI’表示history。这些表包含历史数据，比如历史流程实例，变量，任务等等。
5. ACT\_GE\_\*：通用数据，用于不同场景下。

## 资源库流程规则表

1. act\_re\_deployment：部署信息表
2. act\_re\_model：流程设计模型部署表
3. act\_re\_procdef：流程定义数据表

## 运行时数据库表

1. act\_ru\_execution：运行时流程执行实例表
2. act\_ru\_identitylink：运行时流程人员表，主要存储任务节点与参与者的相关信息
3. act\_ru\_task：运行时任务节点表
4. act\_ru\_variable：运行时流程变量数据表

## 历史数据库表

1. act\_hi\_actinst：历史节点表
2. act\_hi\_attachment：历史附件表
3. act\_hi\_comment：历史意见表
4. act\_hi\_identitylink：历史流程人员表
5. act\_hi\_detail ：历史详情表，提供历史变量的查询
6. act\_hi\_procinst：历史流程实例表
7. act\_hi\_taskinst：历史任务实例表
8. act\_hi\_varinst：历史变量表

## 组织机构表

1. act\_id\_group ：用户组信息表
2. act\_id\_info：用户扩展信息表
3. act\_id\_membership：用户与用户组对应信息表
4. act\_id\_user：用户信息表

这四张表很常见，基本的组织机构管理，关于用户认证方面建议还是自己开发一套，组件自带的功能太简单，使用中有很多需求难以满足。

## 通用数据表

1. act\_ge\_bytearray：二进制数据表
2. act\_ge\_property：属性数据表存储整个流程引擎级别的数据，初始化表结构时会默认插入三条记录

# BPMN

业务流程建模与标注（Business Process Model and Notation，BPMN) ，描述流程的基本符号，包括这些图元如何组合成一个业务流程图（Business Process Diagram）。

# Activiti框架的API使用

首先使用流程设计器插件设计一个请假流程，读者不妨按照如下gif动图来设计：   
在这儿特此作出申明，由于【使用流程设计器插件设计一个请假流程.gif】大小已超过2M的限制，所以未能上传，但读者可点击[**使用流程设计器插件设计一个请假流程.gif**](http://download.csdn.net/detail/yerenyuan_pku/9835457)进行下载并查看，给大家带来一些阅读上的麻烦，还请谅解！！！   
读者在设计请假流程图时，必然要知道Assignee的意思，它指定任务的办理人。恐怕大家可能会有一个疑问：如果像上面那样设计的话，只有张三一个人能提交请假申请，其他人是提交不了申请的。我们现在是为了测试的方便，所以就指定死了，后面我们会有办法来动态地指定。

## 部署流程定义

部署流程定义即将请假规则应用到数据库里面去。部署流程定义操作的数据库表有：部署表(act\_re\_deployment)、流程定义表(act\_re\_procdef)和二进制表(act\_ge\_bytearray)。   
想必大家肯定想知道部署流程定义怎样用代码来实现，我在HelloWorld单元测试类中编写如下单元测试方法：

public class HelloWorld {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 部署流程定义——即把请假规则应用到数据库里面去

\*/

@Test

public void test4() {

// 创建一个部署构建器对象，用于加载流程定义文件(bpmn文件和png文件)

DeploymentBuilder deploymentBuilder = processEngine.getRepositoryService().createDeployment();

deploymentBuilder.addClasspathResource("qjlc.bpmn");

deploymentBuilder.addClasspathResource("qjlc.png");

// 部署，并返回一个部署对象(其实Deployment是一个接口)

Deployment deployment = deploymentBuilder.deploy();

System.out.println(deployment.getId());

}

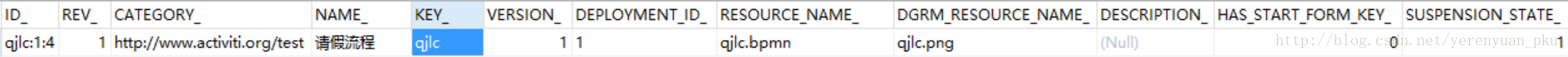
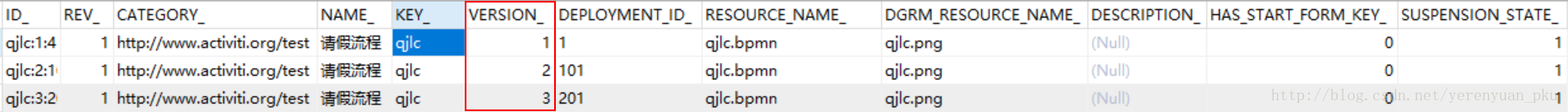
}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

整个Activiti框架最核心的组件是ProcessEngine，部署流程定义就需要用到它，只要是跟工作流相关的任何操作都要使用到流程引擎对象。为了能让接下来编写的所有单元测试方法都能使用到它，故使其成为成员变量。以下这句代码：

Deployment deployment = deploymentBuilder.deploy();

* 1

返回的是一个部署对象，注意Deployment是一个接口。其实只要一调用deploy方法，Activiti框架就会帮我们发出sql语句，来操作数据库表。一旦我们部署一次，对应地就会向部署表(act\_re\_deployment)里面插入一条数据，如下：   
  
同时向流程定义表(act\_re\_procdef)里面插入一条数据，如下：   
  
流程定义表(act\_re\_procdef)里面的KEY\_字段非常关键，KEY\_字段的值是由流程图的id值来决定的。注意：KEY\_这个字段代表的是流程定义的标识，也即说只要KEY\_相同，那么说明它们就是同一个流程，但版本号可能不相同。读者不妨再部署两次，你将看到流程定义表(act\_re\_procdef)就是这样的：   


## 查询流程定义

查询流程定义操作的数据表是流程定义表(act\_re\_procdef)。我在HelloWorld单元测试类中编写如下单元测试方法：

public class HelloWorld {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 查询流程定义

\*/

@Test

public void test5() {

// 流程定义查询对象，用于查询流程定义表（act\_re\_procdef）

ProcessDefinitionQuery query = processEngine.getRepositoryService().createProcessDefinitionQuery();

// 以下查询的是所有的流程定义

List<ProcessDefinition> list = query.list();

for (ProcessDefinition pd : list) {

System.out.println(pd.getId() + " " + pd.getName() + " " + pd.getVersion());

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

以上查询的是所有的流程定义，我们亦可根据流程定义的key来过滤，如下：

public class HelloWorld {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 查询流程定义

\*/

@Test

public void test5() {

// 流程定义查询对象，用于查询流程定义表（act\_re\_procdef）

ProcessDefinitionQuery query = processEngine.getRepositoryService().createProcessDefinitionQuery();

// 根据流程定义的key来过滤

query.processDefinitionKey("qjlc");

List<ProcessDefinition> list = query.list();

for (ProcessDefinition pd : list) {

System.out.println(pd.getId() + " " + pd.getName() + " " + pd.getVersion());

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

万一除了要根据流程定义的key来过滤，还要排序，咋办？以码明示：

public class HelloWorld {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 查询流程定义

\*/

@Test

public void test5() {

// 流程定义查询对象，用于查询流程定义表（act\_re\_procdef）

ProcessDefinitionQuery query = processEngine.getRepositoryService().createProcessDefinitionQuery();

// 根据流程定义的key来过滤

query.processDefinitionKey("qjlc");

// 添加排序条件

query.orderByProcessDefinitionVersion().desc();

List<ProcessDefinition> list = query.list();

for (ProcessDefinition pd : list) {

System.out.println(pd.getId() + " " + pd.getName() + " " + pd.getVersion());

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21

上面是根据流程定义表(act\_re\_procdef)的版本号来降序排列的！那万一我们还要分页查询呢？以码明示：

public class HelloWorld {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 查询流程定义

\*/

@Test

public void test5() {

// 流程定义查询对象，用于查询流程定义表（act\_re\_procdef）

ProcessDefinitionQuery query = processEngine.getRepositoryService().createProcessDefinitionQuery();

// 根据流程定义的key来过滤

query.processDefinitionKey("qjlc");

// 添加排序条件

query.orderByProcessDefinitionVersion().desc();

// 分页查询（伪代码）

query.listPage("从哪开始查", "查几条");

List<ProcessDefinition> list = query.list();

for (ProcessDefinition pd : list) {

System.out.println(pd.getId() + " " + pd.getName() + " " + pd.getVersion());

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23

## 启动流程实例

什么是流程实例？根据某个流程定义的一次具体执行过程，就是一个流程实例。流程定义和流程实例是一对多的关系。在本例中，根据请假流程定义来具体地请一次假，就是启动流程实例了。   
启动流程实例操作的数据表有流程实例表(act\_ru\_execution)、任务表(act\_ru\_task)。我在HelloWorld单元测试类中编写如下单元测试方法：

public class HelloWorld {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 启动流程实例

\*/

@Test

public void test6() {

String processDefinitionId = "qjlc:2:104"; // 流程定义id

ProcessInstance processInstance = processEngine.getRuntimeService()

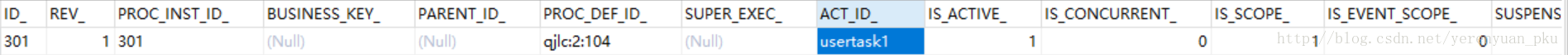
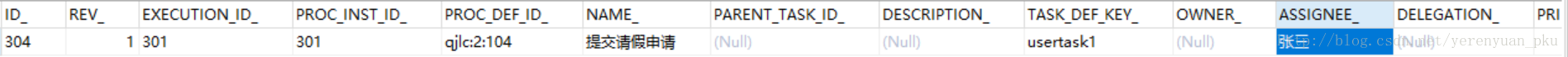
.startProcessInstanceById(processDefinitionId); // 根据请假流程定义来具体地请一次假，即启动流程实例

System.out.println(processInstance.getId());

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

运行以上方法，流程实例表(act\_ru\_execution)里面就会插入一条数据，如下：   
  
ACT\_ID\_字段的值意味着流程向下推进到哪个地步了，上面表中ACT\_ID\_字段的值是usertask1，表示流程推进到【提交请假申请】这一步了。   
除此之外，任务表(act\_ru\_task)里面也会插入一条数据，如下：   
  
从上表可知，张三有一个任务——提交请假申请要办理。

## 查询任务

查询任务操作的数据表是任务表(act\_ru\_task)。我在HelloWorld单元测试类中编写如下单元测试方法：

public class HelloWorld {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 查询任务

\*/

@Test

public void test7() {

// 任务查询对象，操作的是任务表(act\_ru\_task)

TaskQuery query = processEngine.getTaskService().createTaskQuery();

// 根据任务的办理人过滤

query.taskAssignee("张三"); // 只查询张三的任务，其他人的任务不查

// query.taskAssignee("李四");

// query.taskAssignee("王五");

List<Task> list = query.list();

for (Task task : list) {

System.out.println(task.getId() + "\t" + task.getName() + "\t" + task.getAssignee());

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21

上面只查询了张三的任务，其他人的任务没查，因为从任务表(act\_ru\_task)知道张三有一个任务——提交请假申请要办理嘛。当流程一步一步向下推进，任务也会不断发生变化，具体地就要根据任务的办理人来过滤了。

## 办理任务

办理任务操作的数据表有任务表(act\_ru\_task)、流程实例表(act\_ru\_execution)。我在HelloWorld单元测试类中编写如下单元测试方法：

public class HelloWorld {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 办理任务

\*/

@Test

public void test8() {

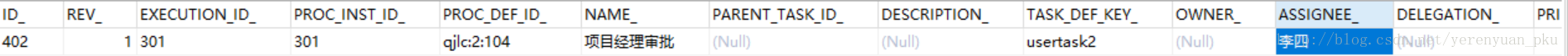
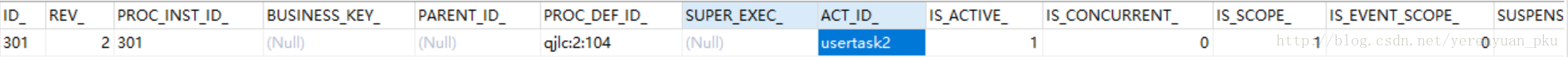
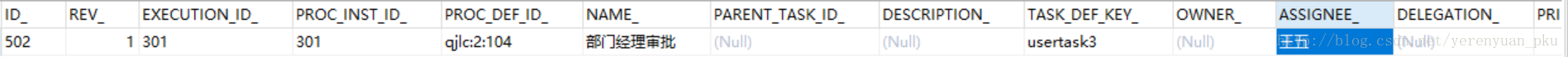
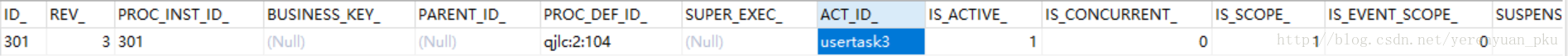
String taskId = "304"; // 任务的id

processEngine.getTaskService().complete(taskId);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

我们查询出张三的任务之后，张三就要办理它，办理完之后，流程向下推进到【项目经理审批】这一步，故任务表(act\_ru\_task)就要发生变化，如下：   
  
从上表可知，李四现在有一个任务——项目经理审批要办理了。除此之外，流程实例表(act\_ru\_execution)也要发生变化，如下：   
  
上面表中ACT\_ID\_字段的值是usertask2，就已经表示流程推进到【项目经理审批】这一步了。   
现在我们就要明确一个概念，流程一步一步向下推进，并不是我们去控制的，而是由工作流框架来帮我们推进的。我们要做的事就是将任务查出来，把它办理完，办理完之后，它会自动地由工作流框架来推进到下一个任务，所以，由工作流框架负责任务一步一步地向下推进，因为我们当时已经把流程部署进去了，也即说这个规则工作流框架是知道的，所以，我们只需要将任务查出来，把它办理完。流程实例表(act\_ru\_execution)也要发生变化，ACT\_ID\_这个字段的值更新了，因为流程向下推进了一步，所以ACT\_ID\_这个字段的值也需要更新。   
下面就很简单了，将李四的任务查询出来，然后办理之。这样流程向下推进到【部门经理审批】这一步，故任务表(act\_ru\_task)就要发生变化，如下：   
  
从上表可知，王五现在有一个任务——部门经理审批要办理了。除此之外，流程实例表(act\_ru\_execution)也要发生变化，如下：   
  
上面表中ACT\_ID\_字段的值是usertask3，就已经表示流程推进到【部门经理审批】这一步了。   
接着再将王五的任务查询出来，然后办理之。这样一来，任务表(act\_ru\_task)和流程实例表(act\_ru\_execution)就没有任何数据了，整个请假流程就走完了。

## 部署流程定义的两种方式

部署流程定义其实有两种方式，第一种方式是加载单个的流程定义文件，正如我之前所讲解的那样。下面再来讲一下这种方式，大家可以加深印象。   
在cn.itcast.activiti包下再新建一个ActivitiAPITest单元测试类，并在该类中编写如下单元测试方法：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 部署流程定义

\*/

@Test

public void test1() {

DeploymentBuilder deploymentBuilder = processEngine

.getRepositoryService().createDeployment();

// 方式一：加载单个的流程定义文件

deploymentBuilder.addClasspathResource("qjlc.bpmn");

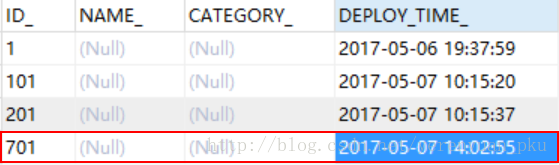
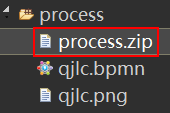
deploymentBuilder.addClasspathResource("qjlc.png");

deploymentBuilder.deploy();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

运行以上方法，即可在部署表(act\_re\_deployment)里面新增一条记录，如下：   
  
部署流程定义的第二种方式是加载zip压缩文件。我们可以将process源码目录下的两个流程定义文件压缩为一个**zip格式**的压缩文件，比如process.zip。   
  
如此一来，就要将test1方法修改为：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 部署流程定义

\*/

@Test

public void test1() {

DeploymentBuilder deploymentBuilder = processEngine

.getRepositoryService().createDeployment();

// 方式二：加载zip压缩文件

ZipInputStream zipInputStream = new ZipInputStream(this.getClass()

.getClassLoader().getResourceAsStream("process.zip")); // 从类路径下读取process.zip压缩文件，并把它包装成一个输入流

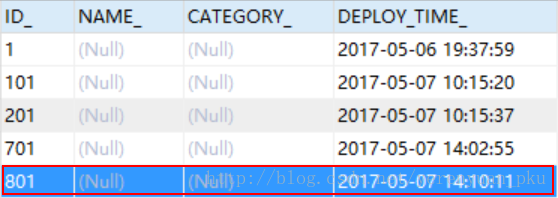
deploymentBuilder.addZipInputStream(zipInputStream );

deploymentBuilder.deploy();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

运行以上方法，又在部署表(act\_re\_deployment)里面新增一条记录，如下：   


## 查询部署信息

在ActivitiAPITest单元测试类中编写如下单元测试方法：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 查询部署信息

\*/

@Test

public void test2() {

// 部署查询对象，查询部署表

DeploymentQuery query = processEngine.getRepositoryService().createDeploymentQuery();

List<Deployment> list = query.list();

for (Deployment deployment : list) {

System.out.println(deployment.getId() + "\t" + deployment.getDeploymentTime());

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

运行以上方法即可查询出部署表(act\_re\_deployment)中所有的记录。

## 删除部署信息

删除部署信息时，同时对应操作的数据库表有部署表(act\_re\_deployment)、流程定义表(act\_re\_procdef)和二进制表(act\_ge\_bytearray)。我在ActivitiAPITest单元测试类中编写如下单元测试方法：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 删除部署信息

\*/

@Test

public void test3() {

String deploymentId = "801"; // 部署id

processEngine.getRepositoryService().deleteDeployment(deploymentId);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

运行以上方法，部署表(act\_re\_deployment)里面ID\_为801的部署信息被删除掉了，附带着流程定义表(act\_re\_procdef)里面DEPLOYMENT\_ID\_为801的流程定义信息也被删掉了，当然了二进制表(act\_ge\_bytearray)里面DEPLOYMENT\_ID\_为801的两条记录同样也被删除掉了。   
void deleteDeployment(String deploymentId);方法有一个重载方法：

* void deleteDeployment(String deploymentId, boolean cascade);   
  cascade：是否级联删除，若cascade=false，则不级联删除；若cascade=true，则级联删除。

先将cascade置为false，为了便于测试，我先启动流程定义id为qjlc:1:4的流程实例， 从流程定义表(act\_re\_procdef)中可以很明显的看出该实例的DEPLOYMENT\_ID\_字段的值为1，如下：   
  
启动流程定义id为qjlc:1:4的流程实例的代码为：

public class HelloWorld {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 启动流程实例

\*/

@Test

public void test6() {

String processDefinitionId = "qjlc:1:4"; // 流程定义id

ProcessInstance processInstance = processEngine.getRuntimeService()

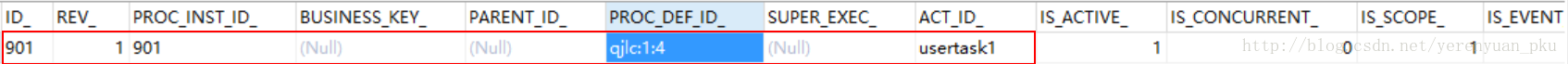
.startProcessInstanceById(processDefinitionId); // 根据请假流程定义来具体地请一次假，即启动流程实例

System.out.println(processInstance.getId());

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

运行以上方法，流程实例表(act\_ru\_execution)里面就会插入一条数据，如下：   
  
ACT\_ID\_字段的值意味着流程向下推进到哪个地步了，上面表中ACT\_ID\_字段的值是usertask1，表示流程推进到【提交请假申请】这一步了。   
除此之外，任务表(act\_ru\_task)里面也会插入一条数据，如下：   
  
从上表可知，张三有一个任务——提交请假申请要办理。   
启动流程实例完毕之后，把ActivitiAPITest单元测试类中的test3方法修改为：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 删除部署信息

\* 删除部署信息时，同时对应操作的数据库表有act\_re\_deployment、act\_re\_procdef、act\_ge\_bytearray

\*/

@Test

public void test3() {

String deploymentId = "1"; // 部署id

boolean cascade = false; // 是否级联删除，false表示不级联删

processEngine.getRepositoryService().deleteDeployment(deploymentId, cascade);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

运行以上方法，这样当删除部署id为1的部署信息时，就会抛出一个org.apache.ibatis.exceptions.PersistenceException异常，可见并没有删除成功，因为有外键约束。   
我们就想删除成功呢？则可以将cascade置为true，但不建议这么做。再次把ActivitiAPITest单元测试类中的test3方法修改为：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 删除部署信息

\* 删除部署信息时，同时对应操作的数据库表有act\_re\_deployment、act\_re\_procdef、act\_ge\_bytearray

\*/

@Test

public void test3() {

String deploymentId = "1"; // 部署id

boolean cascade = true;

processEngine.getRepositoryService().deleteDeployment(deploymentId, cascade);

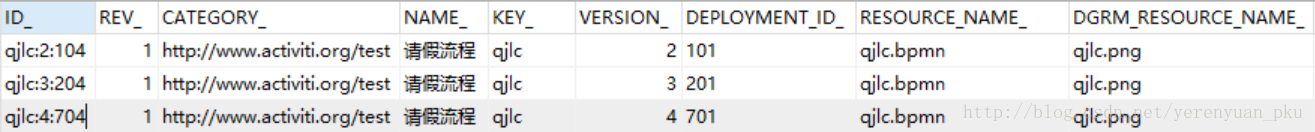
}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

这样当删除部署id为1的部署信息时，不禁发现部署表(act\_re\_deployment)里面ID\_为1的部署信息被删除掉了，附带着流程定义表(act\_re\_procdef)里面DEPLOYMENT\_ID\_为1的流程定义信息也被删掉了，还有流程实例表(act\_ru\_execution)和任务表(act\_ru\_task)中PROC\_DEF\_ID\_字段的值为qjlc:1:4的记录也被删除掉了。

## 获得流程定义文件名称和输入流

假设现在有这样一个需求：查询最新版本的流程定义数据。给出流程定义表(act\_re\_procdef)，如下：   
  
要实现这样一个需求，特简单，在ActivitiAPITest单元测试类中编写如下测试方法：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 查询最新版本的流程定义数据

\*/

@Test

public void test4() {

// 流程定义查询对象，查询的是流程定义表(act\_re\_procdef)

ProcessDefinitionQuery query = processEngine.getRepositoryService().createProcessDefinitionQuery();

// 最新版本过滤

query.latestVersion();

List<ProcessDefinition> list = query.list();

for (ProcessDefinition processDefinition : list) {

System.out.println(processDefinition.getId());

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

好了，回到这一小节的主题，关于如何获得流程定义文件名称和输入流，我个人总结为两种方式。   
【第一种方式】，根据客户端传过来的部署id进行获取。在ActivitiAPITest单元测试类中编写如下测试方法：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 查询一次部署对应的流程定义文件名称和输入流

\* @throws IOException

\*/

@Test

public void test5() throws IOException {

String deploymentId = "201"; // 部署id

// 获得两个流程定义文件的名称

List<String> names = processEngine

.getRepositoryService().getDeploymentResourceNames(deploymentId);

for (String name : names) {

System.out.println(name);

// 获得两个流程定义文件对应的输入流

InputStream in = processEngine

.getRepositoryService().getResourceAsStream(deploymentId, name);

// 读取输入流写到指定的本地磁盘上

FileUtils.copyInputStreamToFile(in, new File("F:\\" + name));

in.close();

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

运行以上方法，除了在Eclipse控制台打印两个流程定义文件的名称，F盘上也会生成两个流程定义文件：

* qjlc.bpmn
* qjlc.png

【第二种方式】，根据客户端传过来的流程定义id进行获取。在ActivitiAPITest单元测试类中编写如下测试方法：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 获得文件名称和输入流

\* @throws IOException

\*/

@Test

public void test6() throws IOException {

String processDefinitionId = "qjlc:2:104"; // 流程定义id

// 直接获得png图片的名称

// 根据流程定义id查询流程定义对象

ProcessDefinitionQuery query = processEngine.getRepositoryService().createProcessDefinitionQuery();

query.processDefinitionId(processDefinitionId);

ProcessDefinition processDefinition = query.singleResult();

// 根据流程定义对象获得png图片的名称

String pngName = processDefinition.getDiagramResourceName();

// 直接获得png图片对应的输入流

InputStream pngStream = processEngine.getRepositoryService().getProcessDiagram(processDefinitionId);

// 读取输入流写到指定的本地磁盘上

FileUtils.copyInputStreamToFile(pngStream, new File("F:\\" + pngName));

pngStream.close();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

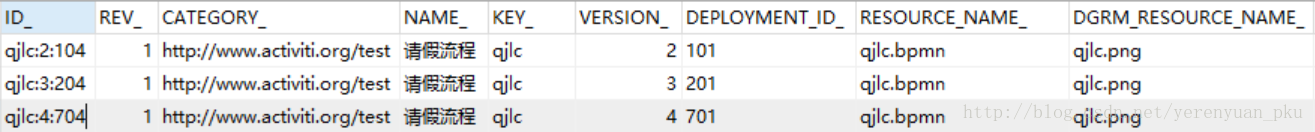
这种方式只是获取到png图片的名称和其对应的输入流。

## 流程实例操作（启动、查询、删除）

### 启动流程实例

启动流程实例可分为两种方式：

* 方式一：根据流程定义的id来启动流程实例
* 方式二：根据流程定义的key来启动流程实例，建议使用

先讲第一种方式，我们之前启动流程实例时就是使用的这种方式。现在再讲一遍加深印象。给出流程定义表(act\_re\_procdef)，如下：   
  
现在我们想启动流程定义id为qjlc:2:104的流程实例，可在ActivitiAPITest单元测试类中编写如下测试方法：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 启动流程实例

\*/

@Test

public void test7() {

String processDefinitionId = "qjlc:2:104"; // 流程定义的id

// 方式一：根据流程定义的id来启动流程实例

ProcessInstance processInstance = processEngine.getRuntimeService()

.startProcessInstanceById(processDefinitionId);

System.out.println(processInstance.getId());

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

再讲第二种方式，这种方式也是被推荐使用的，即根据流程定义的key来启动流程实例，该方式可以自动选择最新版本的流程定义来启动流程实例。以码明示，将ActivitiAPITest单元测试类中的test7方法改为：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 启动流程实例

\*/

@Test

public void test7() {

String processDefinitionKey = "qjlc"; // 流程定义的key

// 方式二：根据流程定义的key来启动流程实例(建议)——可以自动选择最新版本的流程定义来启动流程实例

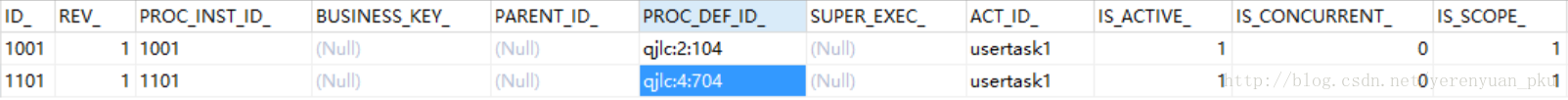
ProcessInstance processInstance = processEngine.getRuntimeService().startProcessInstanceByKey(processDefinitionKey);

System.out.println(processInstance.getId());

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

运行以上方法，启动的是流程定义id为qjlc:4:704的流程实例，给出流程实例表(act\_ru\_execution)，如下：   


### 查询流程实例

查询流程实例操作的是流程实例表(act\_ru\_execution)。在ActivitiAPITest单元测试类中编写如下测试方法：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 查询流程实例

\*/

@Test

public void test8() {

// 流程实例查询对象，操作的是流程实例表(act\_ru\_execution)

ProcessInstanceQuery query = processEngine.getRuntimeService().createProcessInstanceQuery();

List<ProcessInstance> list = query.list();

for (ProcessInstance processInstance : list) {

System.out.println(processInstance.getId());

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

### 删除流程实例

何谓删除流程实例？举个例子，某人把请假流程启动之后，又不想请假了，那意味着后面的人就不用帮他审批了，所以就需要把这个流程实例删除掉。如要删除流程实例id为1001的那个流程实例，则可在ActivitiAPITest单元测试类中编写如下测试方法进行测试：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 删除流程实例

\*/

@Test

public void test9() {

String processInstanceId = "1001"; // 流程实例id

String deleteReason = "不请假了"; // 删除原因，任君写

processEngine.getRuntimeService().deleteProcessInstance(processInstanceId, deleteReason);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

运行以上方法，流程实例id为1001的流程实例被删除掉了，附带着任务表(act\_ru\_task)里面EXECUTION\_ID\_字段为1001的那条记录也被删除掉了。

## 任务操作（查询、办理）

### 查询任务

查询任务对应操作的数据库表是任务表(act\_ru\_task)。在ActivitiAPITest单元测试类中编写如下测试方法进行测试：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 查询任务

\*/

@Test

public void test10() {

// 任务查询对象，对应操作的数据库表是任务表(act\_ru\_task)

TaskQuery query = processEngine.getTaskService().createTaskQuery();

query.taskAssignee("张三");

List<Task> list = query.list();

for (Task task : list) {

System.out.println(task.getId() + "\t" + task.getName());

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

上面只查询了张三的任务，其他人的任务没查，因为从任务表(act\_ru\_task)知道张三有一个任务——提交请假申请要办理嘛。当流程一步一步向下推进，任务也会不断发生变化，具体地就要根据任务的办理人来过滤了。

### 办理任务

办理任务操作的数据表有任务表(act\_ru\_task)、流程实例表(act\_ru\_execution)。我在ActivitiAPITest单元测试类中编写如下单元测试方法：

public class ActivitiAPITest {

ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

/\*\*

\* 办理任务

\*/

@Test

public void test11() {

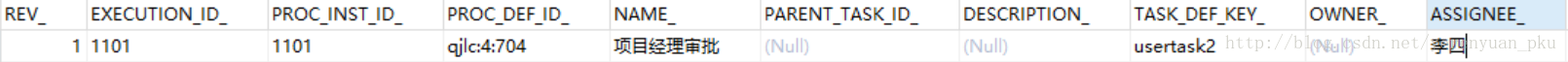
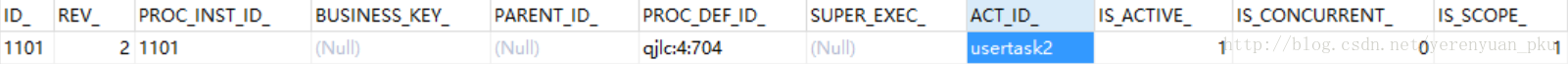
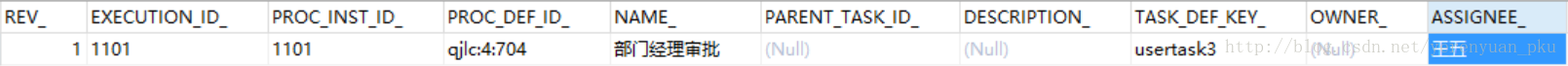
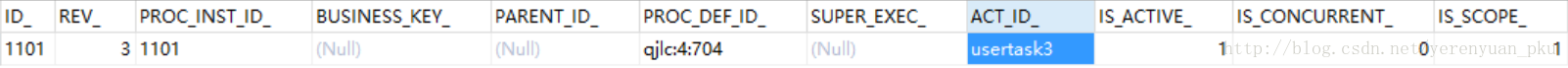
String taskId = "1104"; // 任务id

processEngine.getTaskService().complete(taskId);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

我们查询出张三的任务之后，张三就要办理它，办理完之后，流程向下推进到【项目经理审批】这一步，故任务表(act\_ru\_task)就要发生变化，如下：   
  
从上表可知，李四现在有一个任务——项目经理审批要办理了。除此之外，流程实例表(act\_ru\_execution)也要发生变化，如下：   
  
上面表中ACT\_ID\_字段的值是usertask2，就已经表示流程推进到【项目经理审批】这一步了。   
现在我们就要明确一个概念，流程一步一步向下推进，并不是我们去控制的，而是由工作流框架来帮我们推进的。我们要做的事就是将任务查出来，把它办理完，办理完之后，它会自动地由工作流框架来推进到下一个任务，所以，由工作流框架负责任务一步一步地向下推进，因为我们当时已经把流程部署进去了，也即说这个规则工作流框架是知道的，所以，我们只需要将任务查出来，把它办理完。流程实例表(act\_ru\_execution)也要发生变化，ACT\_ID\_这个字段的值更新了，因为流程向下推进了一步，所以ACT\_ID\_这个字段的值也需要更新。   
下面就很简单了，将李四的任务查询出来，然后办理之。这样流程向下推进到【部门经理审批】这一步，故任务表(act\_ru\_task)就要发生变化，如下：   
  
从上表可知，王五现在有一个任务——部门经理审批要办理了。除此之外，流程实例表(act\_ru\_execution)也要发生变化，如下：   
  
上面表中ACT\_ID\_字段的值是usertask3，就已经表示流程推进到【部门经理审批】这一步了。   
接着再将王五的任务查询出来，然后办理之。这样一来，任务表(act\_ru\_task)和流程实例表(act\_ru\_execution)就没有任何数据了，整个请假流程就走完了。

# 总结activiti中的几个对象

* 几个和流程相关的对象
  + Deployment：部署对象，和部署表(act\_re\_deployment)对应
  + ProcessDefinition：流程定义对象，和流程定义表(act\_re\_procdef)对应
  + ProcessInstance：流程实例对象，和流程实例表(act\_ru\_execution)对应
  + Task：任务对象，和任务表(act\_ru\_task)对应
* 几个Service对象
  + RepositoryService：操作部署、流程定义等静态资源信息
  + RuntimeService：操作流程实例，启动流程实例、查询流程实例、删除流程实例等动态信息
  + TaskService：操作任务，查询任务、办理任务等和任务相关的信息
  + HistoryService：操作历史信息的，查询历史信息
  + IdentityService：操作用户和组
* 几个Query对象
  + DeploymentQuery：对应查询部署表(act\_re\_deployment)
  + ProcessDefinitionQuery：对应查询流程定义表(act\_re\_procdef)
  + ProcessInstanceQuery：对应查询流程实例表(act\_ru\_execution)
  + TaskQuery：对应查询任务表(act\_ru\_task)