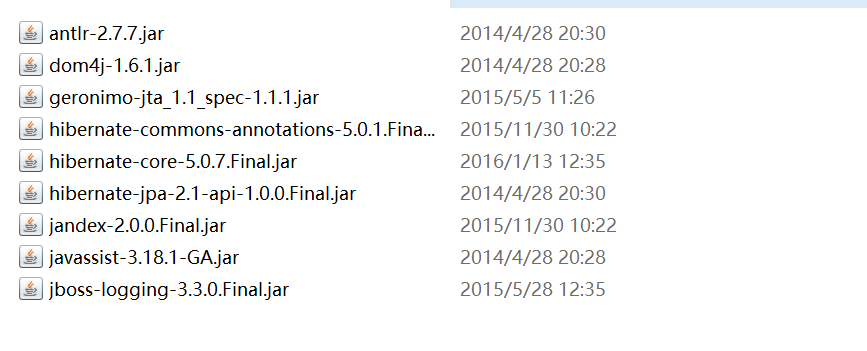
# 1.Hibernate开发环境搭建

Hibernate版本：hibernate-release-5.0.7.Final

整体步骤：

## 1.下载源码，引入jar文件





Log4j日志的三个包



数据库驱动包：



## 2. 写配置

\* hibernate.cfg.xml 主配置

\* Users.hbm.xml 映射配置

## 3.Api

Configuration SessionFactory Transaction Session Query

## 4.创建类与表结构的映射

#### 1. 在JavaBean所在的包下创建映射的配置文件

\* 默认的命名规则为：实体类名.hbm.xml

\* 在xml配置文件中引入约束（引入的是hibernate3.0的dtd约束，不要引入4的约束）

|  |
| --- |
| 映射的dtd文件：  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"> |

#### 2. 如果不能上网，编写配置文件是没有提示的，需要自己来配置

先复制http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd --> window --> preferences --> 搜索xml --> 选择xml catalog --> 点击add --> 现在URI --> 粘贴复制的地址 --> 选择location，选择本地的DTD的路径

#### 3. 编写映射的配置文件

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  <hibernate-mapping>  <class name="com.qf.model.Customer" table="cst\_customer">  <id name="cust\_id" column="cust\_id">  <generator class="native"/>  </id>  <property name="cust\_name" column="cust\_name"/>  <property name="cust\_user\_id" column="cust\_user\_id"/>  <property name="cust\_create\_id" column="cust\_create\_id"/>  <property name="cust\_source" column="cust\_source"/>  <property name="cust\_industry" column="cust\_industry"/>  <property name="cust\_level" column="cust\_level"/>  <property name="cust\_linkman" column="cust\_linkman"/>  <property name="cust\_phone" column="cust\_phone"/>  <property name="cust\_mobile" column="cust\_mobile"/>  </class>  </hibernate-mapping> |

## 5.编写Hibernate核心的配置文件

#### 1. 在src目录下，创建名称为hibernate.cfg.xml的配置文件

#### 2. 在XML中引入DTD约束

|  |
| --- |
| <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd"> |

#### 3. 打开：资料/hibernate-release-5.0.7.Final/project/etc/hibernate.properties，可以查看具体的配置信息

a.必须配置的4大参数

#hibernate.connection.driver\_class com.mysql.jdbc.Driver

#hibernate.connection.url jdbc:mysql:///test

#hibernate.connection.username gavin

#hibernate.connection.password

b.数据库的方言（必须配置的）

#hibernate.dialect org.hibernate.dialect.MySQLDialect

c. 可选的配置

#hibernate.show\_sql true

#hibernate.format\_sql true

#hibernate.hbm2ddl.auto update

d.引入映射配置文件（一定要注意，要引入映射文件，框架需要加载映射文件）

\* <mapping resource="com/qf/ model /Customer.hbm.xml"/>

#### 4. 具体的配置如下

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  <hibernate-configuration>  <session-factory>  <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>  <property name="hibernate.connection.url">jdbc:mysql:///hibernate\_day01</property>  <property name="hibernate.connection.username">root</property>  <property name="hibernate.connection.password">root</property>  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>  <mapping resource="com/qf/model/Customer.hbm.xml"/>  </session-factory>  </hibernate-configuration> |

## 第六步：编写Hibernate入门代码

1. 具体的代码如下

|  |
| --- |
| /\*\*  \* 测试保存客户  \*/  @Test  public void testSave(){  // 先加载配置文件  Configuration config = new Configuration();  // 默认加载src目录下的配置文件  config.configure();  // 创建SessionFactory对象  SessionFactory factory = config.buildSessionFactory();  // 创建session对象  Session session = factory.openSession();  // 开启事务  Transaction tr = session.beginTransaction();  // 编写保存代码  Customer c = new Customer();  // c.setCust\_id(cust\_id); 已经自动递增  c.setCust\_name("测试名称");  c.setCust\_mobile("110");  // 保存客户  session.save(c);  // 提交事务  tr.commit();  // 释放资源  session.close();  factory.close();  } |

## 技术分析之：Hibernate常用的配置文件

#### Hibernate配置文件之映射配置文件

1. 映射文件，即Stu.hbm.xml的配置文件

<hibernate-mapping package="model" >

如果配置 package，name的取值可以是简单类名 name="Person"

\* <class>标签 -- 用来将类与数据库表建立映射关系

\* name -- 类的全路径

\* table -- 表名.(类名与表名一致,那么table属性也可以省略)

\* catalog -- 数据库的名称，基本上都会省略不写

\*dynamic-insert="false" 是否支持动态生成insert语句

\*dynamic-update="false" 是否支持动态生成update语句

如果设置true，hibernate底层将判断提供数据是否为null，如果为null，insert或update语句将没有此项。

\* <id>标签 -- 用来将类中的属性与表中的主键建立映射，id标签就是用来配置主键的。

\* name -- 类中属性名

\* column -- 表中的字段名.(如果类中的属性名与表中的字段名一致,那么column可以省略.)

\* length -- 字段的长度，如果数据库已经创建好了，那么length可以不写。如果没有创建好，生成表结构时，length最好指定。是整形时 length不起作用。

\* <property> -- 用来将类中的普通属性与表中的字段建立映射.

\* name -- 类中属性名

\* column -- 表中的字段名.(如果类中的属性名与表中的字段名一致,那么column可以省略.)

\* length -- 数据长度

\* type -- 表中列的类型。默认hibernate自己通过getter获得类型，一般情况不用设置

\* Hibernate的数据类型 type="string"

\* Java的数据类型 type="java.lang.String"

#### 基本数据与包装类型

* 基本数据类型和包装类型对应hibernate的映射类型相同
* 基本类型无法表达null、数字类型的默认值为0。
* 包装类默认值是null。当对于默认值有业务意义的时候需要使用包装类。

#### 类型对应

|  |  |  |
| --- | --- | --- |
| Java数据类型 | Hibernate数据类型 | 标准SQL数据类型 (PS:对于不同的DB可能有所差异) |
| byte、java.lang.Byte | byte | TINYINT |
| short、java.lang.Short | short | SMALLINT |
| int、java.lang.Integer | integer | INGEGER |
| long、java.lang.Long | long | BIGINT |
| float、java.lang.Float | float | FLOAT |
| double、java.lang.Double | double | DOUBLE |
| java.math.BigDecimal | big\_decimal | NUMERIC |
| char、java.lang.Character | character | CHAR(1) |
| boolean、java.lang.Boolean | boolean | BIT |
| java.lang.String | string | VARCHAR |
| boolean、java.lang.Boolean | yes\_no | CHAR(1)('Y'或'N') |
| boolean、java.lang.Boolean | true\_false | CHAR(1)('Y'或'N') |
| java.util.Date、java.sql.Date | date | DATE |
| java.util.Date、java.sql.Time | time | TIME |
| java.util.Date、java.sql.Timestamp | timestamp | TIMESTAMP |
| java.util.Calendar | calendar | TIMESTAMP |
| java.util.Calendar | calendar\_date | DATE |
| byte[] | binary | VARBINARY、BLOB |
| java.lang.String | text | CLOB |
| java.io.Serializable | serializable | VARBINARY、BLOB |
| java.sql.Clob | clob | CLOB |
| java.sql.Blob | blob | BLOB |
| java.lang.Class | class | VARCHAR |
| java.util.Locale | locale | VARCHAR |
| java.util.TimeZone | timezone | VARCHAR |
| java.util.Currency | currency | VARCHAR |

### Hibernate配置文件之核心配置文件

#### 1. 核心配置文件的两种方式

a. 第一种方式是属性文件的形式，即properties的配置文件

hibernate.properties

hibernate.connection.driver\_class=com.mysql.jdbc.Driver

缺点

不能配置加载映射的配置文件，需要手动编写代码去加载

b．第二种方式是XML文件的形式，开发都会选择这种方式(使用)

hibernate.cfg.xml

<property name="hibernate.connection.driver\_class" >com.mysql.jdbc.Driver</property>

优点

格式比较清晰

编写有提示

可以在该配置文件中加载映射的配置文件（最主要的）

#### 2. 关于hibernate.cfg.xml的配置文件方式

必须有的配置

数据库连接信息:

hibernate.connection.driver\_class -- 连接数据库驱动程序

hibernate.connection.url -- 连接数据库URL

hibernate.connection.username -- 数据库用户名

hibernate.connection.password -- 数据库密码

方言:

hibernate.dialect -- 操作数据库方言

可选的配置

hibernate.show\_sql -- 显示SQL

hibernate.format\_sql -- 格式化SQL

hibernate.hbm2ddl.auto -- 通过映射转成DDL语句

create -- 每次都会创建一个新的表.---测试的时候

create-drop -- 每次都会创建一个新的表,当执行结束之后,将创建的这个表删除.---测试的时候

update -- 如果有表,使用原来的表.没有表,创建一个新的表.同时更新表结构.

validate -- 如果有表,使用原来的表.同时校验映射文件与表中字段是否一致如果不一致就会报错.

加载映射

如果XML方式：<mapping resource="cn/qf/hibernate/model/User.hbm.xml" />

## 技术分析之Hibernate常用的接口和类

### Configuration类和作用

#### 1. Configuration类

Configuration对象用于配置并且启动Hibernate。

Hibernate应用通过该对象来获得对象-关系映射文件中的元数据，以及动态配置Hibernate的属性，然后创建SessionFactory对象。

简单一句话：加载Hibernate的配置文件，可以获取SessionFactory对象。

#### 2. Configuration类的其他应用（了解）

加载配置文件的种类，Hibernate支持xml和properties类型的配置文件，在开发中基本都使用XML配置文件的方式。

如果采用的是properties的配置文件，那么通过Configuration configuration = new Configuration();就可以加载配置文件

但是需要自己手动加载映射文件

例如：config.addResource("cn/qf/model/Student.hbm.xml");

如果采用的XML的配置文件，通过Configuration configuration = new Configuration().configure();加载配置文件

类路径:

Java项目的类路径 是bin目录

Javaweb项目的类路径是WEB-INF/classes

### SessionFactory

（二级缓存 被redis等缓存数据库给代替了。）

**1. 是工厂类，是生成Session对象的工厂类**

**2. SessionFactory类的特点**

由Configuration通过加载配置文件创建该对象。

SessionFactory对象中保存了当前的数据库配置信息和所有映射关系以及预定义的SQL语句。同时，SessionFactory还负责维护Hibernate的二级缓存。

预定义SQL语句

使用Configuration类创建了SessionFactory对象是，已经在SessionFacotry对象中缓存了一些SQL语句

常见的SQL语句是增删改查（通过主键来查询）

这样做的目的是效率更高

一个SessionFactory实例对应一个数据库，应用从该对象中获得Session实例。

SessionFactory是线程安全的，意味着它的一个实例可以被应用的多个线程共享。

SessionFactory是重量级的，意味着不能随意创建或销毁它的实例。如果只访问一个数据库，只需要创建一个SessionFactory实例，且在应用初始化的时候完成。

SessionFactory需要一个较大的缓存，用来存放预定义的SQL语句及实体的映射信息。另外可以配置一个缓存插件，这个插件被称之为Hibernate的二级缓存，被多线程所共享

**3. 总结**

一般应用使用一个SessionFactory,最好是应用启动时就完成初始化。

### 编写HibernateUtil的工具类

1. 具体代码如下

|  |
| --- |
| public class HibernateUtil {  private static final Configuration cfg;  private static final SessionFactory factory;  static{  // 给常量赋值  // 加载配置文件  cfg = new Configuration().configure();  // 生成factory对象  factory = cfg.buildSessionFactory();  }  // 获取Session对象  public static Session openSession(){  return factory.openSession();  }  } |

### Session接口

**1. 概述**

Session是在Hibernate中使用最频繁的接口。也被称之为持久化管理器。它提供了和持久化有关的操作，比如添加、修改、删除、加载和查询实体对象

Session 是应用程序与数据库之间交互操作的一个单线程对象，是 Hibernate 运作的中心

Session是线程不安全的

所有持久化对象必须在 session 的管理下才可以进行持久化操作

Session 对象有一个一级缓存，显式执行 flush 之前，所有的持久化操作的数据都缓存在 session 对象处

持久化类与 Session 关联起来后就具有了持久化的能力

**2. 特点**

不是线程安全的。应避免多个线程使用同一个Session实例

Session是轻量级的，它的创建和销毁不会消耗太多的资源。应为每次客户请求分配独立的Session实例

Session有一个缓存，被称之为Hibernate的一级缓存。每个Session实例都有自己的缓存

**3. 常用的方法**

save(obj)保存一个对象

delete(obj) 删除一个对象

get(Class,id) 根据id查询

load(Class,id)

update(obj) 更新对象

saveOrUpdate(obj) -- 保存或者修改（如果没有数据，保存数据。如果有，修改数据）

createQuery() -- HQL语句的查询的方式

### Transaction接口

**1. Transaction是事务的接口**

**2. 常用的方法**

commit() -- 提交事务

rollback() -- 回滚事务

ThreadLocal和当前线程绑定的一个类

开启事务在业务逻辑层

**3. 特点**

Hibernate框架默认情况下事务不自动提交.需要手动提交事务

如果没有开启事务，那么每个Session的操作，都相当于一个独立的事务

**4. 如果想在Hibernate的框架中来设置隔离级别，需要在hibernate.cfg.xml的配置文件中通过标签来配置**

通过：hibernate.connection.isolation = 4 来配置

取值

1—Read uncommitted isolation

2—Read committed isolation

4—Repeatable read isolation(mysql数据库默认的方案)

8—Serializable isolation

### 丢失更新的问题

1. 如果不考虑隔离性，也会产生写入数据的问题，这一类的问题叫丢失更新的问题。

2. 例如：两个事务同时对某一条记录做修改，就会引发丢失更新的问题。

A事务和B事务同时获取到一条数据，同时再做修改

如果A事务修改完成后，提交了事务

B事务修改完成后，不管是提交还是回滚，如果不做处理，都会对数据产生影响

3. 解决方案有两种

#### 悲观锁

采用的是数据库提供的一种锁机制，如果采用做了这种机制，在SQL语句的后面添加 for update 子句

当A事务在操作该条记录时，会把该条记录锁起来，其他事务是不能操作这条记录的。

只有当A事务提交后，锁释放了，其他事务才能操作该条记录

#### 乐观锁

采用版本号的机制来解决的。会给表结构添加一个字段version=0，默认值是0

当A事务在操作完该条记录，提交事务时，会先检查版本号，如果发生版本号的值相同时，才可以提交事务。同时会更新版本号version=1.

当B事务操作完该条记录时，提交事务时，会先检查版本号，如果发现版本不同时，程序会出现错误。

4. 使用Hibernate框架解决丢失更新的问题

悲观锁

使用session.get(Customer.class, 1,LockMode.UPGRADE); 方法

乐观锁

1.在对应的JavaBean中添加一个属性，名称可以是任意的。例如：private Integer version; 提供get和set方法

2.在映射的配置文件中，提供<version name="version"/>标签即可。

# 2. Spring开发环境搭建

## 步骤一:下载Spring的开发包:

官网：http://spring.io/

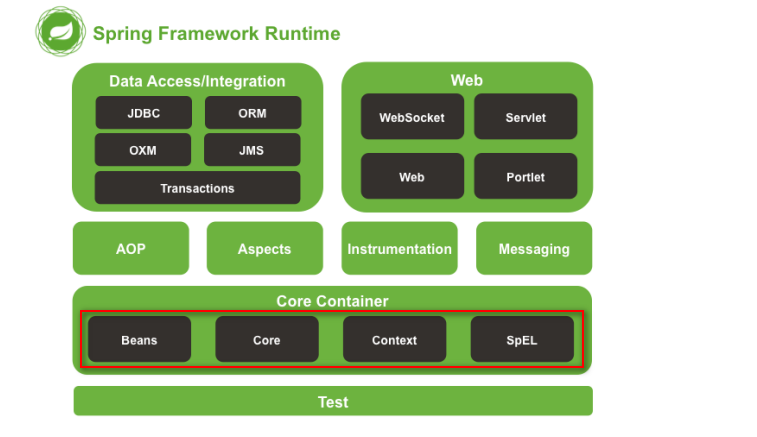
下载地址：http://repo.springsource.org/libs-release-local/org/springframework/spring解压:(Spring目录结构:)

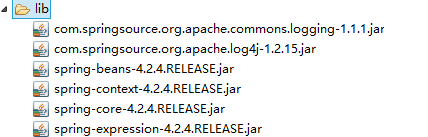
\* docs :API和开发规范.

\* libs :jar包和源码.

\* schema :约束.

## 步骤二:创建web项目,引入Spring的开发包:





4个核心包2个日志包

## 步骤三:创建目标类

* 提供UserService接口和实现类
* 获得UserService实现类的实例

之前开发中，直接new一个对象即可。

学习spring之后，将由Spring创建对象实例--> IoC 控制反转（Inverse of Control）

之后需要实例对象时，从spring工厂（容器）中获得，需要将实现类的全限定名称配置到xml文件中

|  |
| --- |
| **public** **interface** UserService {    **public** **void** addUser();  } |
| **public** **class** UserServiceImpl **implements** UserService {  @Override  **public** **void** addUser() {  System.*out*.println("add user");  }  } |

## 步骤四:编写配置文件

**引入相关配置文件: applicationContext.xml和log4j.properties日志配置文件**

**Spring的配置文件：applicationContext.xml：**

* 位置：任意，开发中一般在classpath下（src）
* 名称：任意，开发中常用applicationContext.xml
* 内容：添加schema约束

|  |
| --- |
| <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=*"userServiceId"* class=*"com.qf.ioc.userServiceImpl"></bean>*  </beans> |

## 步骤五:编写测试类

**ApplicationContext接口（了解）**

\* 使用ApplicationContext工厂的接口，使用该接口可以获取到具体的Bean对象

\* 该接口下有两个具体的实现类

\* ClassPathXmlApplicationContext -- 加载类路径下的Spring配置文件

\* FileSystemXmlApplicationContext

@Test

// Spring的方式：

**public** **void** demo2(){

// 创建Spring的工厂类：

ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("applicationContext.xml");

//获得内容 --不需要自己new，都是从spring容器获得

UserService userService = (UserService) applicationContext.getBean("userServiceId");

userService.addUser();

}

注意：Spring工厂默认调用无参数的构造函数来实例化对象。

## DI属性依赖注入

#### 构造方法注入

##### 目标类

|  |
| --- |
| **public** **class** User {    **private** Integer uid;  **private** String username;  **private** Integer age;    **public** User(Integer uid, String username) {  **super**();  **this**.uid = uid;  **this**.username = username;  }    **public** User(String username, Integer age) {  **super**();  **this**.username = username;  **this**.age = age;  } |

##### spring配置

|  |
| --- |
| <!-- 构造方法注入  \* <constructor-arg> 用于配置构造方法一个参数argument  name ：参数的名称  value：设置普通数据  ref：引用数据，一般是另一个bean id值    index ：参数的索引号，从0开始 。如果只有索引，匹配到了多个构造方法时，默认使用第一个。  type ：确定参数类型  例如：使用名称name  <constructor-arg name="username" value="jack"></constructor-arg>  <constructor-arg name="age" value="18"></constructor-arg>  例如2：【类型type 和 索引 index】  <constructor-arg index="0" type="java.lang.String" value="1"></constructor-arg>  <constructor-arg index="1" type="java.lang.Integer" value="2"></constructor-arg>  -->  <bean id=*"userId"* class=*"com.qf.constructor.User"* >  <constructor-arg index=*"0"* type=*"java.lang.String"* value=*"1"*></constructor-arg>  <constructor-arg index=*"1"* type=*"java.lang.Integer"* value=*"2"*></constructor-arg>  </bean> |

### setter方法注入

|  |
| --- |
| <!-- setter方法注入  \* 普通数据  <property name="" value="值">  等效  <property name="">  <value>值  \* 引用数据  <property name="" ref="另一个bean">  等效  <property name="">  <ref bean="另一个bean"/>    -->  <bean id=*"personId"* class=*"com.qf.setter.Person"*>  <property name=*"pname"* value=*"小苍"*></property>  <property name=*"age"*>  <value>1234</value>  </property>    <property name=*"homeAddr"* ref=*"homeAddrId"*></property>  <property name=*"companyAddr"*>  <ref bean=*"companyAddrId"*/>  </property>  </bean>    <bean id=*"homeAddrId"* class=*"com.qf.setter.Address"*>  <property name=*"addr"* value=*"北京"*></property>  <property name=*"tel"* value=*"****110*** *"*></property>  </bean>  <bean id=*"companyAddrId"* class=*"com.qf.setter.Address"*>  <property name=*"addr"* value=*"北京海淀"*></property>  <property name=*"tel"* value=*"120"*></property>  </bean> |

### P命令空间[了解]

**名称空间p的属性注入的方式:Spring2.x版本后提供的方式.**

* 对“setter方法注入”进行简化，替换<property name="属性名">，而是在

<bean p:属性名="普通值" p:属性名-ref="引用值">

* p命名空间使用前提，必须添加命名空间

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

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

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=*"personId"* class=*"com.qf.p.Person"*  p:pname=*"小泽"* p:age=*"22"*  p:homeAddr-ref=*"homeAddrId"* p:companyAddr-ref=*"companyAddrId"*>  </bean>    <bean id=*"homeAddrId"* class=*"com.qf.p.Address"*  p:addr=*"东莞"* p:tel=*"666666"*>  </bean>  <bean id=*"companyAddrId"* class=*"com.qf.p.Address"*  p:addr=*"日本"* p:tel=*"333333"*>  </bean> |

### SpEL的方式的属性注入[了解]

**Spring3.x版本后提供的方式.**

* 对<property>进行统一编程，所有的内容都使用value

<property name="" value="#{表达式}">

#{123}、#{'jack'} ： 数字、字符串

#{beanId} ：另一个bean引用

#{beanId.propName} ：操作数据

#{beanId.toString()} ：执行方法

#{T(类).字段|方法} ：静态方法或字段

|  |
| --- |
| <!--  <property name="cname" value="#{'jack'}"></property>  <property name="cname" value="#{customerId.cname.toUpperCase()}"></property>  通过另一个bean，获得属性，调用的方法  <property name="cname" value="#{customerId.cname?.toUpperCase()}"></property>  ?. 如果对象不为null，将调用方法  -->  <bean id=*"customerId"* class=*"com.qf.spel.Customer"* >  <property name=*"cname"* value=*"#{customerId.cname?.toUpperCase()}"*></property>  <property name=*"pi"* value=*"#{T(java.lang.Math).PI}"*></property>  </bean> |

### 集合注入

|  |
| --- |
| <!--  集合的注入都是给<property>添加子标签  数组：<array>  List：<list>  Set：<set>  Map：<map> ，map存放k/v 键值对，使用<entry>描述  Properties：<props> <prop key=""></prop> 【】    普通数据：<value>  引用数据：<ref>  -->  <bean id=*"collDataId"* class=*"com.qf.coll.CollData"* >  <property name=*"arrayData"*>  <array>  <value>hello wolrd</value>  <value>jack</value>  <value>rose</value>  </array>  </property>    <property name=*"listData"*>  <list>  <value>芙蓉</value>  <value>菊花</value>  <value>如花</value>  </list>  </property>    <property name=*"setData"*>  <set>  <value>芙蓉</value>  <value>菊花</value>  <value>如花</value>  </set>  </property>    <property name=*"mapData"*>  <map>  <entry key=*"jack"* value=*"杰克"*></entry>  <entry key=*"rose"* value=*"肉丝"*></entry>  </map>  </property>    <property name=*"propsData"*>  <props>  <prop key=*"高富帅"*>冠希哥</prop>  <prop key=*"白富美"*>阿凤姐</prop>  <prop key=*"男屌丝"*>菊花信</prop>  </props>  </property>  </bean> |

## Spring的分配置文件的开发

一种:创建工厂的时候加载多个配置文件:

ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml","applicationContext2.xml");

二种:在一个配置文件中包含另一个配置文件：

<import resource="applicationContext2.xml"></import>

## 技术分析之Spring框架的IOC功能之注解的方式

### Spring框架的IOC之注解方式的快速入门

### 1. 步骤一：导入注解开发所有需要的jar包

引入IOC容器必须的6个jar包

多引入一个：Spring框架的AOP的jar包，spring-aop的jar包

### 2. 步骤二：创建对应的包结构，编写Java的类

UserService -- 接口

UserServiceImpl -- 具体的实现类

### 3. 步骤三：引入约束

在src的目录下，创建applicationContext.xml的配置文件，然后引入约束。注意：因为现在想使用注解的方式，那么引入的约束发生了变化

\* 需要引入context的约束，具体的约束如下

<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"> <!-- bean definitions here -->

</beans>

### 4. 步骤四：在applicationContext.xml配置文件中开启组件扫描

Spring的注解开发:组件扫描

<context:component-scan base-package="com.qf.demo1"/>

注意：可以采用如下配置

<context:component-scan base-package="com.qf"/> 这样是扫描com.qf包下所有的内容

### 5. 步骤五：在UserServiceImpl的实现类上添加注解

@Component(value="userService") -- 相当于在XML的配置方式中 <bean id="userService" class="...">

### 6. 步骤六：编写测试代码

public class SpringDemo1 {

@Test

public void run1(){

ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");

UserService us = (UserService) ac.getBean("userService");

us.save();

}

}

## Spring框架中Bean管理的常用注解

### **1. @Component:组件**.(作用在类上)

### 2. Spring中提供@Component的三个衍生注解:(功能目前来讲是一致的)

@Controller -- 作用在WEB层

@Service -- 作用在业务层

@Repository -- 作用在持久层

说明：这三个注解是为了让标注类本身的用途清晰，Spring在后续版本会对其增强

### 3. 属性注入的注解(说明：使用注解注入的方式可以,可以不用提供set方法)

如果是注入的普通类型，可以使用value注解

@Value -- 用于注入普通类型

如果注入的是对象类型，使用如下注解

@Autowired -- 默认按类型进行自动装配

如果想按名称注入

@Qualifier -- 强制使用名称注入

@Resource -- 相当于@Autowired和@Qualifier一起使用

强调：Java提供的注解

属性使用name属性

### 4.Bean的作用范围和生命周期的注解

#### 1. Bean的作用范围注解

注解为@Scope(value="prototype")，作用在类上。值如下：

singleton -- 单例，默认值

prototype -- 多例

#### 2. Bean的生命周期的配置（了解）

注解如下：

@PostConstruct -- 相当于init-method

@PreDestroy -- 相当于destroy-method

## spring aop编程：【掌握】

### 导包：spring的传统AOP的开发的包

\* spring-aop-4.2.4.RELEASE.jar

\* com.springsource.org.aopalliance-1.0.0.jar

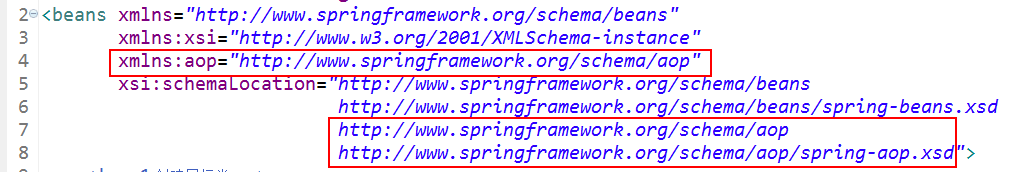
\* aspectJ的开发包

\* com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

\* spring-aspects-4.2.4.RELEASE.jar

* 从spring容器获得目标类，如果配置aop，spring将自动生成代理。
* 要确定目标类，aspectj 切入点表达式，导入jar包

### spring配置



|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns: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"*>  <!-- 1 创建目标类 -->  <bean id=*"userServiceId"* class=*"com.qf.c\_spring\_aop.UserServiceImpl"*></bean>  <!-- 2 创建切面类（通知） -->  <bean id=*"myAspectId"* class=*"com.qf.c\_spring\_aop.MyAspect"*></bean>  <!-- 3 aop编程  3.1 导入命名空间  3.2 使用 <aop:config>进行配置  proxy-target-class="true" 声明时使用cglib代理  <aop:pointcut> 切入点 ，从目标对象获得具体方法  <aop:advisor> 特殊的切面，只有一个通知 和 一个切入点  advice-ref 通知引用  pointcut-ref 切入点引用  3.3 切入点表达式  execution(\* com.qf.c\_spring\_aop.\*.\*(..))  选择方法 返回值任意 包 类名任意 方法名任意 参数任意    -->  <aop:config proxy-target-class=*"true"*>  <aop:pointcut expression=*"execution(\* com.qf.c\_spring\_aop.\*.\*(..))"* id=*"myPointCut"*/>  <aop:advisor advice-ref=*"myAspectId"* pointcut-ref=*"myPointCut"*/>  </aop:config>  </beans> |

# 3.AspectJ

## 介绍

* AspectJ是一个基于Java语言的AOP框架
* Spring2.0以后新增了对AspectJ切点表达式支持
* @AspectJ 是AspectJ1.5新增功能，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

* 主要用途：自定义开发

## 切入点表达式【掌握】

1.execution() 用于描述方法 【掌握】

语法：execution(修饰符 返回值 包.类.方法名(参数) throws异常)

修饰符，一般省略

public 公共方法

\* 任意

返回值，不能省略

void 返回没有值

String 返回值字符串

\* 任意

包，[省略]

com.qf.crm 固定包

com.qf.crm.\*.service crm包下面子包任意 （例如：com.qf.crm.service）

com.qf.crm.. crm包下面的所有子包（含自己）

com.qf.crm.\*.service.. crm包下面任意子包，固定目录service，service目录任意包

类，[省略]

UserServiceImpl 指定类

\*Impl 以Impl结尾

User\* 以User开头

\* 任意

方法名，不能省略

addUser 固定方法

add\* 以add开头

\*Do 以Do结尾

\* 任意

(参数)

() 无参

(int) 一个整型

(int ,int) 两个

(..) 参数任意

throws ,可省略，一般不写。

**execution(public \* com.qf.service.impl.\*.\*(..))**

## AspectJ 通知类型

* aop联盟定义通知类型，具有特性接口，必须实现，从而确定方法名称。
* aspectj 通知类型，只定义类型名称。方法格式。
* 个数：掌握1中。

before:前置通知(应用：各种校验)

在方法执行前执行，如果通知抛出异常，阻止方法运行

afterReturning:后置通知(应用：常规数据处理)

方法正常返回后执行，如果方法中抛出异常，通知无法执行

必须在方法执行后才执行，所以可以获得方法的返回值。

around:环绕通知(应用：十分强大，可以做任何事情)

方法执行前后分别执行，可以阻止方法的执行

必须手动执行目标方法

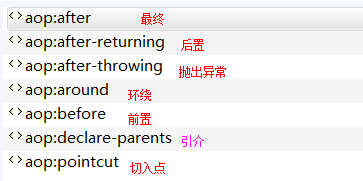
afterThrowing:抛出异常通知(应用：包装异常信息)

方法抛出异常后执行，如果方法没有抛出异常，无法执行

after:最终通知(应用：清理现场)

方法执行完毕后执行，无论方法中是否出现异常

|  |
| --- |
| 环绕  try{  //前置：before  //手动执行目标方法  //后置：afterRetruning  } catch(){  //抛出异常 afterThrowing  } finally{  //最终 after  } |



## 基于xml

### 步骤一：创建JavaWEB项目，引入具体的开发的jar包

\* 先引入Spring框架开发的基本开发包

\* 再引入Spring框架的AOP的开发包

\* spring的传统AOP的开发的包

\* spring-aop-4.2.4.RELEASE.jar

\* com.springsource.org.aopalliance-1.0.0.jar

\* aspectJ的开发包

\* com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

\* spring-aspects-4.2.4.RELEASE.jar

### 步骤二：创建Spring的配置文件，引入具体的AOP的schema约束

|  |
| --- |
| <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"> |

### 步骤三：编写切面类

|  |
| --- |
| /\*\*  \* 切面类，含有多个通知  \*/  **public** **class** MyAspect {    **public** **void** myBefore(JoinPoint joinPoint){  System.*out*.println("前置通知 ： " + joinPoint.getSignature().getName());  }    **public** **void** myAfterReturning(JoinPoint joinPoint,Object ret){  System.*out*.println("后置通知 ： " + joinPoint.getSignature().getName() + " , -->" + ret);  }    **public** Object myAround(ProceedingJoinPoint joinPoint) **throws** Throwable{  System.*out*.println("前");  //手动执行目标方法  Object obj = joinPoint.proceed();    System.*out*.println("后");  **return** obj;  }    **public** **void** myAfterThrowing(JoinPoint joinPoint,Throwable e){  System.*out*.println("抛出异常通知 ： " + e.getMessage());  }    **public** **void** myAfter(JoinPoint joinPoint){  System.*out*.println("最终通知");  }  } |

### 步骤四：spring配置

**编写目标类，切面类 aop的配置**

|  |
| --- |
| <!-- 1 创建目标类 -->  <bean id=*"userServiceId"* class=*"com.qf.d\_aspect.a\_xml.UserServiceImpl"*></bean>  <!-- 2 创建切面类（通知） -->  <bean id=*"myAspectId"* class=*"com.qf.d\_aspect.a\_xml.MyAspect"*></bean>  <!-- 3 aop编程  <aop:aspect> 将切面类 声明“切面”，从而获得通知（方法）  ref 切面类引用  <aop:pointcut> 声明一个切入点，所有的通知都可以使用。  expression 切入点表达式  id 名称，用于其它通知引用  -->  <aop:config>  <aop:aspect ref=*"myAspectId"*>  <aop:pointcut expression=*"execution(\* com.qf.d\_aspect.a\_xml.UserServiceImpl.\*(..))"* id=*"myPointCut"*/>    <!-- 3.1 前置通知  <aop:before method="" pointcut="" pointcut-ref=""/>  method : 通知，及方法名  pointcut :切入点表达式，此表达式只能当前通知使用。  pointcut-ref ： 切入点引用，可以与其他通知共享切入点。  通知方法格式：public void myBefore(JoinPoint joinPoint){  参数1：org.aspectj.lang.JoinPoint 用于描述连接点（目标方法），获得目标方法名等  例如：  <aop:before method="myBefore" pointcut-ref="myPointCut"/>  -->    <!-- 3.2后置通知 ,目标方法后执行，获得返回值  <aop:after-returning method="" pointcut-ref="" returning=""/>  returning 通知方法第二个参数的名称  通知方法格式：public void myAfterReturning(JoinPoint joinPoint,Object ret){  参数1：连接点描述  参数2：类型Object，参数名 returning="ret" 配置的  例如：  <aop:after-returning method="myAfterReturning" pointcut-ref="myPointCut" returning="ret" />  -->    <!-- 3.3 环绕通知  <aop:around method="" pointcut-ref=""/>  通知方法格式：public Object myAround(ProceedingJoinPoint joinPoint) throws Throwable{  返回值类型：Object  方法名：任意  参数：org.aspectj.lang.ProceedingJoinPoint  抛出异常  执行目标方法：Object obj = joinPoint.proceed();  例如：  <aop:around method="myAround" pointcut-ref="myPointCut"/>  -->  <!-- 3.4 抛出异常  <aop:after-throwing method="" pointcut-ref="" throwing=""/>  throwing ：通知方法的第二个参数名称  通知方法格式：public void myAfterThrowing(JoinPoint joinPoint,Throwable e){  参数1：连接点描述对象  参数2：获得异常信息，类型Throwable ，参数名由throwing="e" 配置  例如：  <aop:after-throwing method="myAfterThrowing" pointcut-ref="myPointCut" throwing="e"/>  -->  <!-- 3.5 最终通知 -->  <aop:after method=*"myAfter"* pointcut-ref=*"myPointCut"*/>        </aop:aspect>  </aop:config> |

### 步骤五：完成测试

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**)  @ContextConfiguration("classpath:com/qf/test/beans.xml")  **public** **class** TestAop {  @Resource(name="userDao")  **private** UserDao userDao;  @Test  **public** **void** run1(){  userDao.add();  }  } |
|  |

## AOP基于注解

### 引入相关的jar包:

\* spring的传统AOP的开发的包

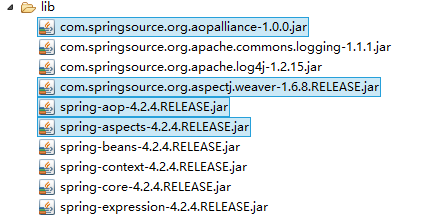
spring-aop-4.2.4.RELEASE.jar

com.springsource.org.aopalliance-1.0.0.jar

\* aspectJ的开发包:

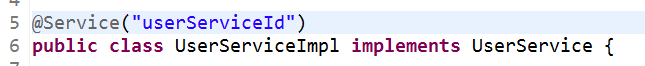
com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

spring-aspects-4.2.4.RELEASE.jar



### 替换bean

|  |
| --- |
| <!-- 1 创建目标类 -->  <bean id=*"userServiceId"* class=*"com.qf.d\_aspect.b\_anno.UserServiceImpl"*></bean>  <!-- 2 创建切面类（通知） -->  <bean id=*"myAspectId"* class=*"com.qf.d\_aspect.b\_anno.MyAspect"*></bean> |





* 注意：扫描

|  |
| --- |
| <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"*  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*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd"*>  <!-- 1.扫描 注解类 -->  <context:component-scan base-package=*"com.qf.d\_aspect.b\_anno"*></context:component-scan> |

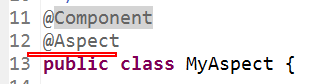
### 替换aop

* 必须进行aspectj 自动代理

|  |
| --- |
| <!-- 2.确定 aop注解生效 -->  <aop:aspectj-autoproxy></aop:aspectj-autoproxy> |

* 声明切面

|  |
| --- |
| <aop:aspect ref=*"myAspectId"*> |



* 替换前置通知

|  |
| --- |
| <aop:before method=*"myBefore"* pointcut=*"execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))"*/> |

|  |
| --- |
| //切入点当前有效  @Before("execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))")  **public** **void** myBefore(JoinPoint joinPoint){  System.*out*.println("前置通知 ： " + joinPoint.getSignature().getName());  } |

* 替换 公共切入点

|  |
| --- |
| <aop:pointcut expression=*"execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))"* id=*"myPointCut"*/> |

|  |
| --- |
| //声明公共切入点  @Pointcut("execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))")  **private** **void** myPointCut(){  } |

* 替换后置

|  |
| --- |
| <aop:after-returning method=*"myAfterReturning"* pointcut-ref=*"myPointCut"* returning=*"ret"* /> |

|  |
| --- |
| @AfterReturning(value="myPointCut()" ,returning="ret")  **public** **void** myAfterReturning(JoinPoint joinPoint,Object ret){  System.*out*.println("后置通知 ： " + joinPoint.getSignature().getName() + " , -->" + ret);  } |

* 替换环绕

|  |
| --- |
| <aop:around method=*"myAround"* pointcut-ref=*"myPointCut"*/> |

|  |
| --- |
| @Around(value = "myPointCut()")  **public** Object myAround(ProceedingJoinPoint joinPoint) **throws** Throwable{  System.*out*.println("前");  //手动执行目标方法  Object obj = joinPoint.proceed();    System.*out*.println("后");  **return** obj;  } |

* 替换抛出异常

|  |
| --- |
| <aop:after-throwing method=*"myAfterThrowing"* pointcut=*"execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))"* throwing=*"e"*/> |

|  |
| --- |
| @AfterThrowing(value="execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))" ,throwing="e")  **public** **void** myAfterThrowing(JoinPoint joinPoint,Throwable e){  System.*out*.println("抛出异常通知 ： " + e.getMessage());  } |

### 切面类

|  |
| --- |
| /\*\*  \* 切面类，含有多个通知  \*/  @Component  @Aspect  **public** **class** MyAspect {    //切入点当前有效  // @Before("execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))")  **public** **void** myBefore(JoinPoint joinPoint){  System.*out*.println("前置通知 ： " + joinPoint.getSignature().getName());  }    //声明公共切入点  @Pointcut("execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))")  **private** **void** myPointCut(){  }    // @AfterReturning(value="myPointCut()" ,returning="ret")  **public** **void** myAfterReturning(JoinPoint joinPoint,Object ret){  System.*out*.println("后置通知 ： " + joinPoint.getSignature().getName() + " , -->" + ret);  }    // @Around(value = "myPointCut()")  **public** Object myAround(ProceedingJoinPoint joinPoint) **throws** Throwable{  System.*out*.println("前");  //手动执行目标方法  Object obj = joinPoint.proceed();    System.*out*.println("后");  **return** obj;  }    // @AfterThrowing(value="execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))" ,throwing="e")  **public** **void** myAfterThrowing(JoinPoint joinPoint,Throwable e){  System.*out*.println("抛出异常通知 ： " + e.getMessage());  }    @After("myPointCut()")  **public** **void** myAfter(JoinPoint joinPoint){  System.*out*.println("最终通知");  }  } |

### spring配置

|  |
| --- |
| <!-- 1.扫描 注解类 -->  <context:component-scan base-package=*"com.qf.d\_aspect.b\_anno"*></context:component-scan>    <!-- 2.确定 aop注解生效 -->  <aop:aspectj-autoproxy></aop:aspectj-autoproxy> |

### aop注解总结

@Aspect 声明切面，修饰切面类，从而获得 通知。

通知

@Before 前置

@AfterReturning 后置

@Around 环绕

@AfterThrowing 抛出异常

@After 最终

切入点

@PointCut ，修饰方法 private void xxx(){} 之后通过“方法名”获得切入点引用

<?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"*>

<!-- 配置目标类 -->

<bean id=*"userService"*class=*"com.qf.aspectj.UserServiceImpl"*></bean>

<!-- 配置通知类 -->

<bean id=*"advice"* class=*"com.qf.aspectj.Myadvice"*></bean>

<!-- aop编程 -->

<aop:config>

<!-- 配置一个公共的切点-->

<aop:pointcut expression=*"execution(\* com.qf.aspectj.\*.\*(..))"* id=*"mycut"*/>

<!-- ref:指向通知类的id -->

<aop:aspect ref=*"advice"*>

<!-- 配置前置通知 -->

<aop:before method=*"before"* pointcut-ref=*"mycut"*/>

<!-- <aop:after-returning配置后置通知

method：配置后置通知的方法名

pointcut-ref：引入某个切点id

returning="res" 配置返回值的形参名

-->

<aop:after-returning method=*"after"* pointcut-ref=*"mycut"* returning=*"res"*/>

<!--<aop:after-throwing 配置异常通知 throwing:配置方法啊的形参名-->

<aop:after-throwing method=*"afterThrowing"* pointcut-ref=*"mycut"* throwing=*"e"*/>

<!--<aop:after 配置最终通知 -->

<aop:after method=*"afterFinal"* pointcut-ref=*"mycut"*/>

<!-- 配置环绕通知 -->

<aop:around method=*"around"* pointcut-ref=*"mycut"*/>

</aop:aspect>

</aop:config>

</beans>

# 4.SpringJDBCTemplate配置

## 1.创建项目,引入jar包

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>aopalliance</groupId>

<artifactId>aopalliance</artifactId>

<version>1.0</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.8.10</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-aspects -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>4.3.7.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>4.3.10.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>4.3.11.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.3.10.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>4.3.10.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>4.3.10.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-expression</artifactId>

<version>4.3.10.RELEASE</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>4.3.9.RELEASE</version>

</dependency>

<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.38</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.1.2</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.14</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.38</version>

</dependency>

<!-- https://mvnrepository.com/artifact/commons-dbcp/commons-dbcp -->

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.4</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-test -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>4.3.10.RELEASE</version>

<scope>test</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/com.mchange/mchange-commons-java -->

<dependency>

<groupId>com.mchange</groupId>

<artifactId>mchange-commons-java</artifactId>

<version>0.2.11</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot</artifactId>

<version>1.5.7.RELEASE</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.mchange/c3p0 -->

<dependency>

<groupId>com.mchange</groupId>

<artifactId>c3p0</artifactId>

<version>0.9.5.2</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>4.3.9.RELEASE</version>

</dependency>

## 2.导入日志:log4j.properties

1.2 创建数据库

数据库名: springjdbc

创建角色表:

CREATE TABLE `role` (

`rid` INT(10) UNSIGNED NOT NULL AUTO\_INCREMENT COMMENT '角色id',

`rname` VARCHAR(20) NOT NULL COMMENT '角色名',

`alias` VARCHAR(20) NOT NULL COMMENT '角色别名',

PRIMARY KEY (`rid`)

)

INSERT INTO role (rid,rname,alias)

VALUES(1,'admin','系统管理员'),(2,'dispatcher','调度员');

**1.3 测试代码**

@Test

public void test1() throws Exception {

//TODO 测试jdbcTemplate简单使用

//1.创建c3p0链接池

ComboPooledDataSource dataSource = new ComboPooledDataSource();

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

dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/springjdbc");

dataSource.setUser("root");

dataSource.setPassword("111");

//创建jdbcTemplate对象

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

//创建sql语句

String sql = "insert into role (rid , rname ,alias) value (? , ?,?);";

jdbcTemplate.update(sql,"3","visitor","游客");

}

## 3.spring管理Template

### 1. 创建Role.java

public class Role {

private Integer cid;

private String rname;

private String alias;

//setter getter

}

### 2. RoleDao.java

public interface RoleDao {

//增

void save(Role role);

//删除

void delete(Integer id);

//改

void update(Role role);

//查

Role getById(Integer id);

//查

int getTotalCount();

//查

List<Role> getAll();

}

### 3. 创建RoleDaoImpl.java

创建数据源(连接池),JdbcTemplate,RoleDao交给Spring容器管理(IOC)

注意: 可以自己在RoleDaoImpl中添加 JdbcTemplate变量,如果不自动装载记得添加变量的set方法,

标准的操作,我们可以让RoleDaoImpl 继承 JdbcDaoSupport, 因为(org.springframework.jdbc.core.support.JdbcDaoSupport)类提供了JdbcTemplate对象以及对应的获取和设置方法.不用自己在实现类中添加JdbcTemplate变量!!

JdbcTemplate // Spring操作数据模板类(工具类)

JdbcTemplate.update(sql,ArgsObj....); //DML

JdbcTemplate.execute(sql) //DDL DCL

//DQL 查询单个

jdbcTemplate.queryForObject(String var1, RowMapper<T> var2, Object... var3);

RowWapper<T> 将结果封装的处理器; 得到Result解析成实体类对象即可!

//查询所有

jdbcTemplate.query(String var1, RowMapper<T> var2, Object... var3);

RoleDaoImple代码:

public class RoleDaoImpl extends JdbcDaoSupport implements RoleDao {

public void save(Role role) {

//TODO 插入数据

String sql = "INSERT INTO role (rname,alias) value (?,?) ;";

getJdbcTemplate().update(sql,role.getRname(),role.getAlias());

}

public void delete(Integer id) {

//TODO 根据id删除

String sql = "delete from role where rid = ? ;";

getJdbcTemplate().update(sql,id);

}

public void update(Role role) {

//TODO 修改role信息

String sql="update role set rname = ? , alias = ? where rid = ? ;" ;

getJdbcTemplate().update(sql,role.getRname(),role.getAlias(),role.getRid());

}

public Role getById(Integer id) {

String sql = "select \* from role where rid = ? ;";

//直接查询

// Role role = getJdbcTemplate().queryForObject(sql, new Object[]{id}, Role.class);

Role role = (Role) getJdbcTemplate().queryForObject(sql, new Object[]{id}, new RowMapper<Object>() {

public Object mapRow(ResultSet resultSet, int index) throws SQLException {

//将一行数据解析成一个java对象

return mapRowHandler(resultSet);

}

});

return role;

}

public int getTotalCount() {

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

Integer count = getJdbcTemplate().queryForObject(sql,Integer.class);

return count;

}

//查询所有

public List<Role> getAll() {

String sql = "select \* from role";

List<Role> list = getJdbcTemplate().query(sql, new RowMapper<Role>() {

public Role mapRow(ResultSet resultSet, int i) throws SQLException {

return mapRowHandler(resultSet);

}

});

return list;

}

private Role mapRowHandler(ResultSet resultSet) throws SQLException{

Role role = new Role();

role.setRid(resultSet.getInt("rid"));

role.setRname(resultSet.getString("rname"));

role.setAlias(resultSet.getString("alias"));

return role;

}

}

### 4. 创建数据库配置文件

文件名:db.properties 位置: src下/ maven项目 resources文件夹下

jdbc.jdbcUrl=jdbc:mysql://localhost:3306/springjdbc

jdbc.driverClass=com.mysql.jdbc.Driver

jdbc.user=root

jdbc.password=111

### 5.编写IOC配置

文件名:applicationContext.xml 位置:src下 / maven项目 resources文件夹下

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

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

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 definitions here -->

<context:component-scan base-package="com.itqf.spring"></context:component-scan>

<context:property-placeholder location="db.properties"></context:property-placeholder>

<bean name="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" p:jdbcUrl="${jdbc.jdbcUrl}"

p:driverClass="${jdbc.driverClass}"

p:user="root"

p:password="111"

/>

<!-- bean jdbcTemplate -->

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

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

</bean>

<bean name="roleDao" class="com.itqf.spring.dao.impl.RoleDaoImpl">

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

</bean>

</beans>

测试:

@Test

public void test2(){

//TODO 测试使用配置文件

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

RoleDao dao = context.getBean("roleDao", RoleDaoImpl.class);

Role byId = dao.getById(2);

System.out.println(byId);

List<Role> all = dao.getAll();

System.out.println("all = " + all);

}

# 5.Spring管理事务

## 1.基于XML配置文件管理事务

### 1.1. TransactionManager

在不同平台，操作事务的代码各不相同，因此spring提供了一个 TransactionManager 接口：

- DateSourceTransactionManager 用于 JDBC 的事务管理

- HibernateTransactionManager 用于 Hibernate 的事务管理

- JpaTransactionManager 用于 Jpa 的事务管理

### 1.2 接口的定义

事务的属性介绍：这里定义了传播行为、隔离级别、超时时间、是否只读

package org.springframework.transaction;

public interface TransactionDefinition {

int PROPAGATION\_REQUIRED = 0; //支持当前事务，如果不存在，就新建一个

int PROPAGATION\_SUPPORTS = 1; //支持当前事务，如果不存在，就不使用事务

int PROPAGATION\_MANDATORY = 2; //支持当前事务，如果不存在，就抛出异常

int PROPAGATION\_REQUIRES\_NEW = 3;//如果有事务存在，挂起当前事务，创建一个新的事物

int PROPAGATION\_NOT\_SUPPORTED = 4;//以非事务方式运行，如果有事务存在，挂起当前事务

int PROPAGATION\_NEVER = 5;//以非事务方式运行，如果有事务存在，就抛出异常

int PROPAGATION\_NESTED = 6;//如果有事务存在，则嵌套事务执行

int ISOLATION\_DEFAULT = -1;//默认级别，MYSQL: 默认为REPEATABLE\_READ级别 SQLSERVER: 默认为READ\_COMMITTED

int ISOLATION\_READ\_UNCOMMITTED = 1;//读取未提交数据(会出现脏读, 不可重复读) 基本不使用

int ISOLATION\_READ\_COMMITTED = 2;//读取已提交数据(会出现不可重复读和幻读)

int ISOLATION\_REPEATABLE\_READ = 4;//可重复读(会出现幻读)

int ISOLATION\_SERIALIZABLE = 8;//串行化

int TIMEOUT\_DEFAULT = -1;//默认是-1，不超时，单位是秒

//事务的传播行为

int getPropagationBehavior();

//事务的隔离级别

int getIsolationLevel();

//事务超时时间

int getTimeout();

//是否只读

boolean isReadOnly();

String getName();

}

### 1.3添加tx命名空间

事务基础组件，包括对 DAO 的支持及 JCA 的集成 修改 applicationContext.xml

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

<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

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-4.2.xsd

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

http://www.springframework.org/schema/context/spring-context-4.2.xsd

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

http://www.springframework.org/schema/aop/spring-aop-4.2.xsd

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

http://www.springframework.org/schema/tx/spring-tx-4.2.xsd">

</beans>

### 1.4 添加事务相关配置

修改applicationContext.xml

<!-- 平台事务管理器 -->

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

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

</bean>

<!-- 通知 -->

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

<tx:attributes>

<!-- 传播行为 -->

<!-- REQUIRED：如果有事务，则在事务中执行；如果没有事务，则开启一个新的事物 -->

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

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

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

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

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

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

<tx:method name="transfer" propagation="REQUIRED" />

<!-- SUPPORTS：如果有事务，则在事务中执行；如果没有事务，则不会开启事物 -->

<tx:method name="find\*" propagation="SUPPORTS" read-only="true" />

<tx:method name="select\*" propagation="SUPPORTS" read-only="true" />

<tx:method name="get\*" propagation="SUPPORTS" read-only="true" />

</tx:attributes>

</tx:advice>

<aop:config>

<aop:pointcut id="txPointCut" expression="execution(\* com..service..\*.\*(..))" />

<aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut"/>

</aop:config>

**配置介绍:**

tx:advice 是用于配置事务相关信息, transaction-manager属性是引入对应类型的事务管理;

​ jdbc/mybatias : DataSourceTransactionManager

​ hibernate: HibernateTransactionManager

​ JPA:JPATransactionManager

​ tx:attributes 此标签所配置的是 哪些方法可以作为事务方法(为后面切点进行补充)

​ tx:method 标签设置具体要添加事务的方法和其他属性

​ name 是必须的,表示与事务属性关联的方法名(业务方法名),对切入点进行细化。通配符\*可以用来指定一批关联到相同的事务属性的方法。如：'get\*'、'handle\*'、'on\*Event'等等.

​ propagation 不是必须的 ，默认值是REQUIRED 表示事务传播行为, 包括REQUIRED,SUPPORTS,MANDATORY,REQUIRES\_NEW,NOT\_SUPPORTED,NEVER,NESTED

​ isolation 不是必须的 默认值DEFAULT

​ timeout 不是必须的 默认值-1(永不超时) 表示事务超时的时间（以秒为单位）

​ read-only 不是必须的 默认值false不是只读的 表示事务是否只读？

​ rollback-for 不是必须的 表示将被触发进行回滚的 Exception(s)；以逗号分开。 如：'com.itqf.MyBusinessException,ServletException'

​ no-rollback-for 不是必须的 表示不被触发进行回滚的 Exception(s)；以逗号分开。 如：'com.foo.MyBusinessException,ServletException'

aop:config标签 设置事务的切点,配置参与事务的类和对应的方法.

注意: aop:config和tx:advice 但是两者并不冲突, aop:config面向切面编程的切点,选择对应的方法进行切入,而tx:adivce是设置事务的相关的属性和描述,换句话说,aop:config选择了对应的切入点,tx:advice是在这些切入点上根据 method name属性再次进行筛选!!!

## 2. 使用注解方式添加事务

除了基于XML的事务配置,Spring还提供了基于注解的事务配置,即通过@Transactional对需要事务增强的Bean接口,实现类或者方法进行标注,在容器中配置基于注解的事务增强驱动,即可启用注解的方式声明事务!

### 2.1使用@Transactional注解

顺着原来的思路,使用@Transactional对基于aop/tx命名空间的事务配置进行改造!

- 修改service类添加@Transactional注解

@Transactional //对业务类进行事务增强的标注

@Service("accountService")

public class AccountServiceImpl implements AccountService {

@Autowired

private AccountDao accountDao;

public void transfer(final Integer from, final Integer to, final Double money) {

accountDao.decreaseMoney(from,money);

int i = 1 / 0 ;

accountDao.increaseMoney(to,money);

}

因为注解本身具有一组默认的事务属性,所以往往只要在需要事务的业务类中添加一个@Transactional注解,就完成了业务类事务属性的配置!

当然,注解只能提供元数据,它本身并不能完成事务切面织入的功能.因此,还需要在Spring的配置中通过一行配置'通知'Spring容器对标注@Transactional注解的Bean进行加工处理!

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

<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

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

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

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.2.xsd

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

http://www.springframework.org/schema/context/spring-context-4.2.xsd

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

http://www.springframework.org/schema/aop/spring-aop-4.2.xsd

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

http://www.springframework.org/schema/tx/spring-tx-4.2.xsd">

<context:component-scan base-package="com.itqf.spring"/>

<context:property-placeholder location="db.properties" />

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

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

</bean>

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

p:driverClass="${jdbc.driverClass}"

p:jdbcUrl="${jdbc.jdbcUrl}"

p:user="${jdbc.user}"

p:password="${jdbc.password}"

/>

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

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

</bean>

<bean id="template" class="org.springframework.transaction.support.TransactionTemplate">

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

</bean>

<!-- <bean id="entityManagerFactory" class="org.springframework.orm.jpa.aspectj.JpaExceptionTranslatorAspect"-->

<!--① 对标注@Transactional注解的Bean进行加工处理,以织入事物管理切面 -->

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

</beans>

### 2.2 @Transactional其他方面介绍

- 关于@Transactional的属性

基于@Transactional注解的配置和基于xml的配置一样,它拥有一组普适性很强的默认事务属性,往往可以直接使用默认的属性.

- 事务传播行为: PROPAGATION\_REQUIRED.

- 事务隔离级别: ISOLATION\_DEFAULT.

- 读写事务属性:读/写事务.

- 超时时间:依赖于底层的事务系统默认值

- 回滚设置:任何运行期异常引发回滚,任何检查型异常不会引发回滚.

默认值可能适应大部分情况,但是我们依然可以可以自己设定属性,具体属性表如下:

- 在何处标注@Transactional注解

@Transactional注解可以直接用于接口定义和接口方法,类定义和类的public方法上.

但Spring建议在业务实现类上使用@Transactional注解,当然也可以添加到业务接口上,但是这样会留下一些容易被忽视的隐患,因为注解不能被继承,所以业务接口中标注的@Transactional注解不会被业务类实现继承.

- 在方法出使用注解

方法出添加注解会覆盖类定义处的注解,如果有写方法需要使用特殊的事务属性,则可以在类注解的基础上提供方法注解,如下:

@Transactional

@Service("accountService")

public class AccountServiceImpl implements AccountService {

@Autowired

private AccountDao accountDao;

//不等于默认值!可以覆盖类注解

@Transactional(readOnly = false , isolation = Isolation.READ\_COMMITTED)

public void transfer(final Integer from, final Integer to, final Double money) {

accountDao.decreaseMoney(from,money);

// int i = 1/0;

accountDao.increaseMoney(to,money);

}

}

使用不同的事务管理器

一般情况下,一个应用仅需要使用一个事务管理器.如果希望在不同的地方使用不同的事务管理,@Transactional注解同样支持!

实现代码:

@Transactional("事务管理器的名字") //此处添加指定事务管理器的名字

@Service("accountService")

public class AccountServiceImpl implements AccountService {

@Autowired

private AccountDao accountDao;

public void transfer(final Integer from, final Integer to, final Double money) {

accountDao.decreaseMoney(from,money);

// int i = 1/0;

accountDao.increaseMoney(to,money);

}

}

对应事务查找事务管理器的名字应该在xml中进行定义!如下:

<!--声明一个事务管理器 -->

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

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

<qualifier value="定义事务管理器的名字,可以被注解查找" />

</bean>

# 6.springMVC配置

## 1.maven项目pom配置:

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

<scope>provided</scope>

</dependency>

<!-- 添加Spring包 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.3.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.3.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>4.3.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>4.3.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>4.3.6.RELEASE</version>

</dependency>

<!-- 为了方便进行单元测试，添加spring-test包 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>4.3.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>4.3.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-expression</artifactId>

<version>4.3.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>4.3.6.RELEASE</version>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.8.5</version>

</dependency>

<!-- jstl -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

## 配置web.xml

<servlet>

//①

<servlet-name>springMVC</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!--SpringMVC配置文件的名字 <servlet-name>-servlet.xml

默认位置:src / resources

如果放在了 src/resources(maven)

contextConfigLocation:classpath:文件名即可!

Web-INF/xx.xml

contextConfigLocation:/WEB-INF/xx.xml

-->

//②

<init-param>

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

<param-value>/WEB-INF/springMVC-servlet.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<!-- 访问DispatcherServlet对应的路径 -->

<servlet-mapping>

<servlet-name>springMVC</servlet-name>

<url-pattern>/</url-pattern> <!--/不过滤jsp防止死循环-->

</servlet-mapping>

-

1./\* 拦截所有请求 css img js jsp html action

2.\*.action/\*.do \*.action 可以拦截以.action结尾的请求

3./ 拦截css js image action请求 不拦截jsp(推荐)

配置解释:

Spring MVC同样需要创建配置文件,但是配置文件的位置由DispatcherServlet的配置决定!默认情况下,Spring MVC配置文件的名字为:<servlet-name>-servlet.xml 位置:src/resources(maven)

当然，我们也可以修改默认配置的名字和位置的引用!如果想修改,我们可以直接在声明DispatcherServlet标签中添添加init-param 即可, init-param的参数名固定

## 3. 配置Spring MVC配置文件

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

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">

<!-- 扫描controller-->

<context:component-scan base-package="com.itqf.springmvc.controller"/>

<!-- 视图解析器 -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<!-- jsp所在的位置-->

<property name="prefix" value="/WEB-INF/jsp/" />

<!-- jsp文件的后缀名-->

<property name="suffix" value=".jsp" />

</bean>

</beans>

## 4.使用原生API向页面传递数据

如果我们想使用servlet原生的API向前端页面传递数据,只要在参数中传入HttpServletRequest, HttpServletResponse,HttpSession等,但是需要在maven中配置jsp,jstl, javax.servlet-api

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>javax.servlet.jsp-api</artifactId>

<version>2.2.1</version>

<scope>provided</scope>

</dependency>

## 5.spring中实现restful风格

Restful风格中有四种数据操作方式,GET(查询获取),POST(新建),PUT(更新),DELETE(删除),但是前端页面的提交方式只支持GET和POST两种,如果想支持另外两种,需要在web.xml文件中增加一个HiddenHttpMethodFilter过滤器.

<filter>

<filter-name>HiddenHttpMethodFilter</filter-name>

<filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filterclass>

</filter>

<filter-mapping>

<filter-name>HiddenHttpMethodFilter</filter-name>

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

</filter-mapping>

同时前端form表单中需要添加如下内容:

<input type="hidden" name="\_method" value="DELETE" />

## 6.处理静态资源

DispatcherServlet中配置了/会过滤除了jsp之外的所有资源,所以当我们想使用静态资源的时候,如html,img等会出错,这个时候要通过配置来过滤掉这些静态资源.

### 1.修改SpringMvc对应配置文件,添加Mvc命名空间和约束

<?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:mvc="http://www.springframework.org/schema/mvc"  
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/mvc  
http://www.springframework.org/schema/mvc/spring-mvc.xsd](http://www.springframework.org/schema/beanshttp://www.springframework.org/schema/beans/spring-beans.xsdhttp://www.springframework.org/schema/contexthttp://www.springframework.org/schema/context/spring-context.xsdhttp://www.springframework.org/schema/mvchttp://www.springframework.org/schema/mvc/spring-mvc.xsd)”>

### 2.添加处理标签

<mvc:annotation-driven /> <!--注解驱动-->  
<mvc:resources mapping="/static/\*\*" location="/static/" ></mvc:resources>

这样我们需要将静态资源全部放在static文件夹下

## 7.配置handlerAdapter和HandlerMapping

当我们需要实现返回json字符串和文件上传功能的时候,需要配置上面的处理器映射器和适配器

<!—处理器映射器-->  
<bean  
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandler  
Mapping" />  
<!--处理器适配器-->  
<bean  
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandler  
Adapter" />  
<mvc:annotation-driven /> 配置这一个可以抵用上面的两句话

## 8.springMVC中配置拦截器

先写一个MyInterceptors实现HandlerInterceptor接口,然后在springmvc配置文件中文件中配置如下内容:

<mvc:interceptors>  
<mvc:interceptor>  
<mvc:mapping path="/user/\*"/>  
<bean class="com.itqf.spring.interceptors.MyInterceptors" />  
</mvc:interceptor>  
</mvc:interceptors>

## 9.springMVC中处理文件上传

### **1.引入jar包**

commons-fileupload.jar commons-io.jar

<!-- https://mvnrepository.com/artifact/commons-fileupload/commons-fileupload -->  
<dependency>  
<groupId>commons-fileupload</groupId>  
<artifactId>commons-fileupload</artifactId>  
<version>1.3.1</version>  
</dependency>  
<!-- https://mvnrepository.com/artifact/commons-io/commons-io -->  
<dependency>  
<groupId>commons-io</groupId>  
<artifactId>commons-io</artifactId>  
<version>2.4</version>  
</dependency>

### 2.配置MultipartResolver

<!--multipartResolverᯈᗝ-->  
<bean id="multipartResolver"  
class="org.springframework.web.multipart.commons.CommonsMultipartResolver"  
p:defaultEncoding="UTF-8"  
p:maxUploadSize="5242880"  
p:uploadTempDir="file:/d:/temp"  
/>

# 7.Maven项目配置

maven安装

### 下载安装

* 下载

从该网站 <http://maven.apache.org/download.cgi> 下载maven3.3.9 版本

* 解压

将maven解压到一个不含有中文和空格的目录中。

bin目录 mvn.bat （以run方式运行项目）、 mvnDebug.bat（以debug方式运行项目 ）

boot目录 maven运行需要类加载器

conf目录 settings.xml 整个maven工具核心配置文件

lib目录 maven运行依赖jar包

### 环境变量配置

电脑上需安装java环境，安装JDK1.7 + 版本 （将JAVA\_HOME/bin 配置环境变量path ）

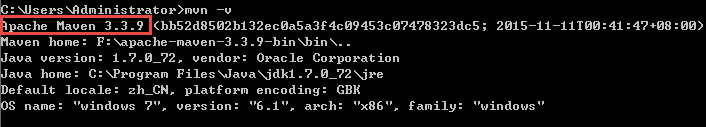
配置 MAVEN\_HOME



将 %MAVEN\_HOME%/bin 加入环境变量 path



通过 mvn -v命令检查 maven是否安装成功，看到maven的版本为3.3.9及java版本为1.7即为安装成功。

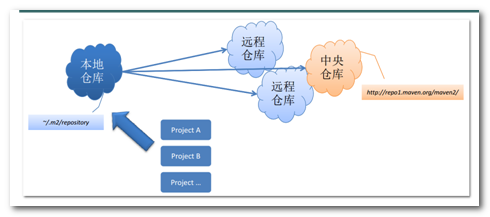


maven仓库

### maven仓库的作用

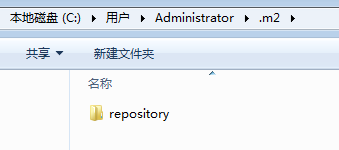
maven的工作需要从仓库下载一些jar包，如下图所示，本地的项目A、项目B等都会通过maven软件从远程仓库（可以理解为互联网上的仓库）下载jar包并存在本地仓库，本地仓库 就是本地文件夹，当第二次需要此jar包时则不再从远程仓库下载，因为本地仓库已经存在了，可以将本地仓库理解为缓存，有了本地仓库就不用每次从远程仓库下载了。

下图描述了maven中仓库的类型：



* 本地仓库 ：用来存储从远程仓库或中央仓库下载的插件和jar包，项目使用一些插件或jar包，优先从本地仓库查找

默认本地仓库位置在 ${user.dir}/.m2/repository，${user.dir}表示windows用户目录。

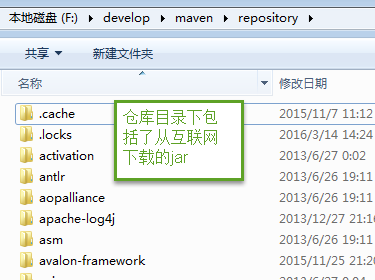


* 远程仓库：如果本地需要插件或者jar包，本地仓库没有，默认去远程仓库下载。

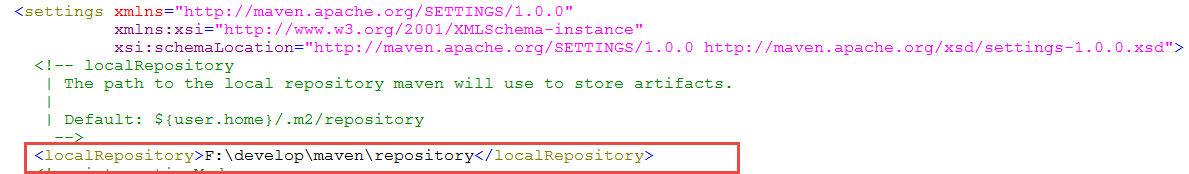
远程仓库可以在互联网内也可以在局域网内。

* 中央仓库 ：在maven软件中内置一个远程仓库地址http://repo1.maven.org/maven2 ，它是中央仓库，服务于整个互联网，它是由Maven团队自己维护，里面存储了非常全的jar包，它包含了世界上大部分流行的开源项目构件。

### 配置本地仓库



在MAVE\_HOME/conf/settings.xml文件中配置本地仓库位置：



### 全局setting与用户setting

maven仓库地址、私服等配置信息需要在setting.xml文件中配置，分为全局配置和用户配置。

在maven安装目录下的有 conf/setting.xml文件，此setting.xml文件用于maven的所有project项目，它作为maven的全局配置。

如需要个性配置则需要在用户配置中设置，用户配置的setting.xml文件默认的位置在：${user.dir} /.m2/settings.xml目录中,${user.dir} 指windows 中的用户目录。

maven会先找用户配置，如果找到则以用户配置文件为准，否则使用全局配置文件。



项目构建

### m2e插件安装配置

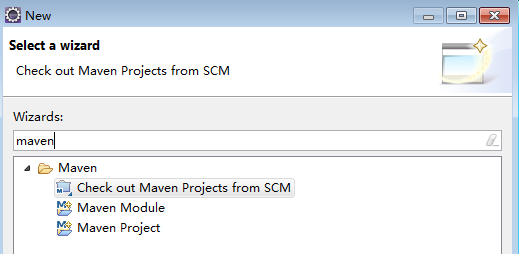
### eclipse与maven

可以在eclipse开发工具中集成maven软件，eclipse是一个开发工具，maven是一个项目管理工具，maven有一套项目构建的规范，在eclipse集成maven软件，最终通过eclipse创建maven工程。

### 插件安装

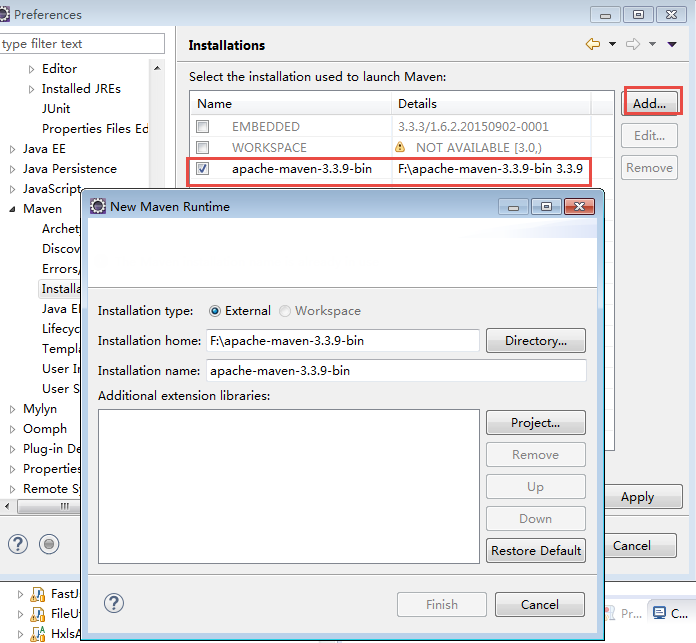
本教程 使用eclipse mars 2版本，此版本自带maven插件不用单独安装。

在新建窗口中可看到maven项目的创建项表示maven插件已存在，如下图：



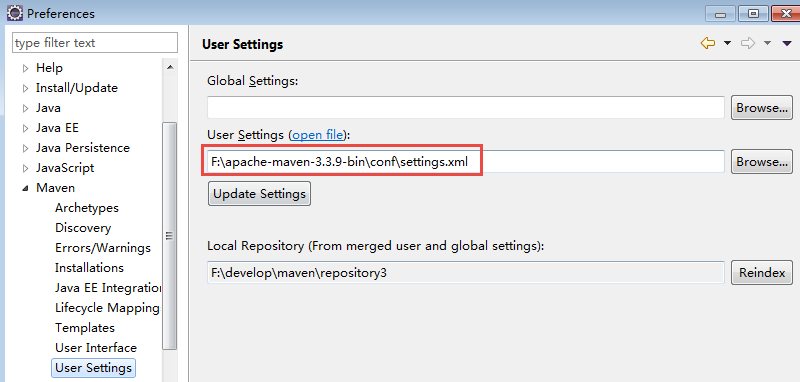
### 指定maven安装目录

一些高版本的eclipse已经内置了maven的安装，下图是eclipse mars2版本中已经内置了maven3.3.3版本，项目为了统一maven的版本不会使用eclipse内置的maven版本，这里我们maven3.3.9。



### User Setting配置

在eclipse中配置使用的maven的setting.xml文件，使用maven安装目录下的setting.xml文件。

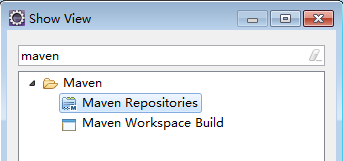


注意：如果修改了 setting.xml文件需要点击上图中的“update settings”按钮对本地仓库重建索引，点击“Reindex”。

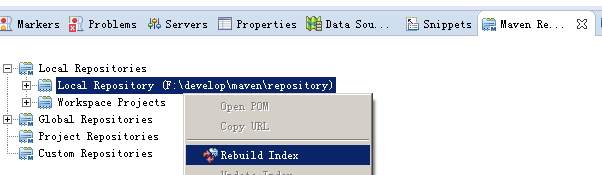
### eclipse浏览仓库

maven配置完成需要测试在eclipse中是否可以浏览maven的本地仓库，如果可以正常浏览maven本地仓库则说明eclipse集成maven已经完成。

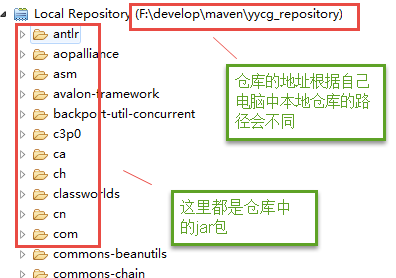
打开eclipse仓库视图，对插件和jar包建立索引



找到Local respository本地仓库项，点击Rebuild index重建索引



重建索引完成点击前边的“+”图标即可查看本地仓库的内容：如下图：



### 定义maven坐标

每个maven工程都需要定义本工程的坐标，坐标是maven对jar包的身份定义，比如：入门程序的坐标定义如下：

<!--项目名称，定义为组织名+项目名，类似包名-->

<groupId>com.qf.maven</groupId>

<!-- 模块名称 -->

<artifactId>maven-first</artifactId>

<!-- 当前项目版本号，snapshot为快照版本即非正式版本，release为正式发布版本 -->

<version>0.0.1-SNAPSHOT</version>

<packaging > ：打包类型

jar：执行package会打成jar包

war：执行package会打成war包

pom ：用于maven工程的继承，通常父工程设置为pom

构建web工程

### 需求

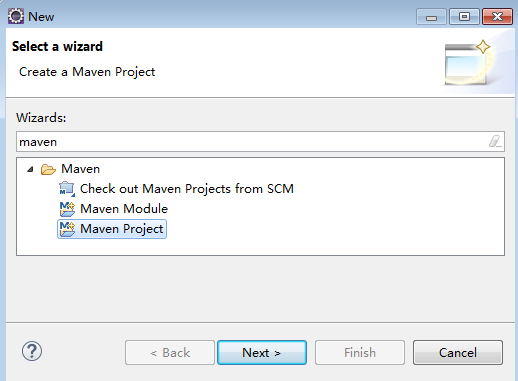
创建一个web工程，实现入门程序的功能。

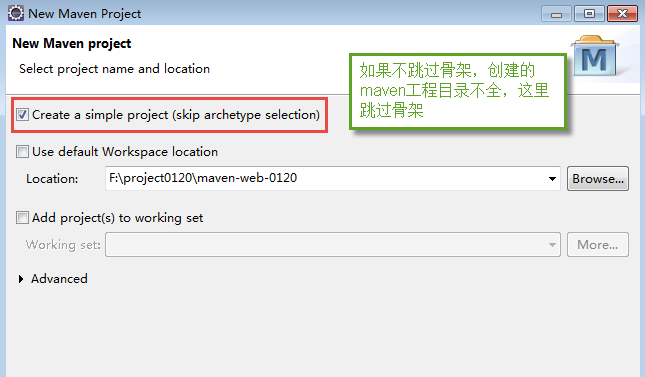
1）添加index.jsp，输出hello world

2）添加一个servlet转发到jsp页面。

### 第一步创建maven工程

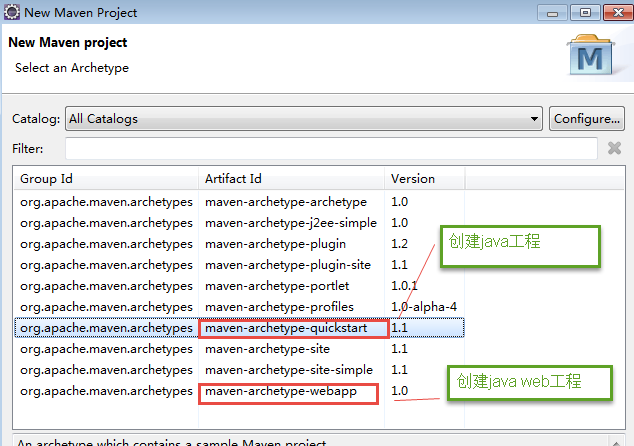
选择 Maven Project





了解一下骨架界面:

当我们不跳过骨架点击“next”会进入骨架选择页面，如果eclipse中配置本地仓库正确则显示出骨架：

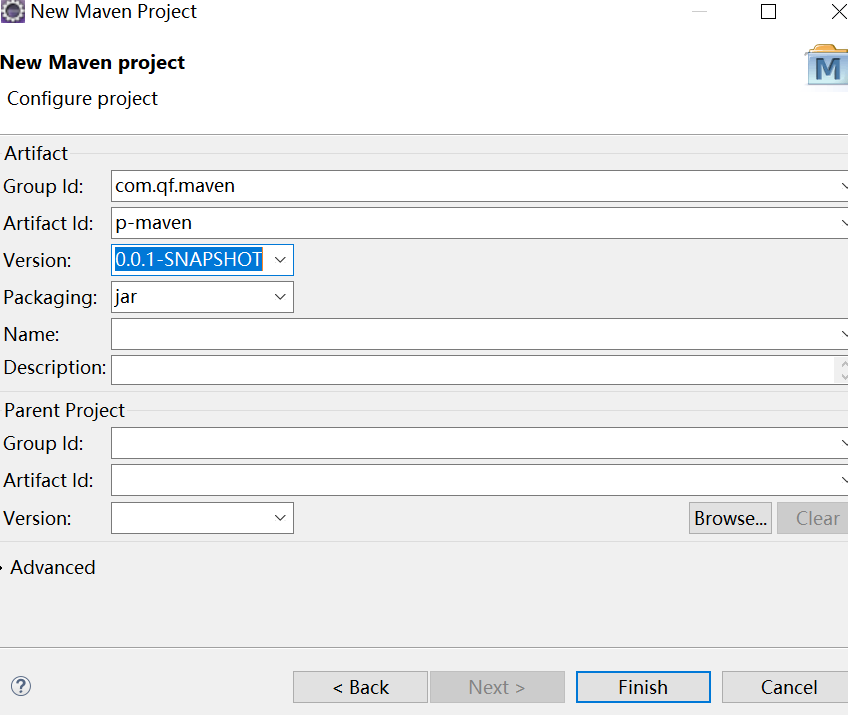


选择webapp会创建一个maven工程（java web工程）

选择quickstart会创建一个maven工程（java工程）

注意：使用骨架的问题是创建的maven工程目录不完整，所以不推荐使用，本教程创建maven工程统一跳过骨架。

### 第二步定义坐标

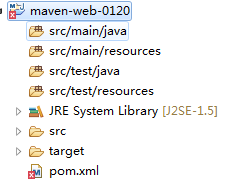


Group Id ：组织名或者公司名 com.qf.maven

Artifact Id:项目名

Version:版本

Packaing:打包方式：java项目选jar web项目war



### 第三步设置编译版本

查看上边工程的编译版本为1.5，本教程 使用jdk1.7，需要设置编译版本为1.7，这里需要使用maven的插件来设置：

在pom.xml中加入：

<!-- 设置项目的编译版本和Tomcat插件 -->

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

</plugins>

<pluginManagement>

<plugins>

<!-- 配置Tomcat插件 -->

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<configuration>

<port>8080</port>

<encoding>UTF-8</encoding>

</configuration>

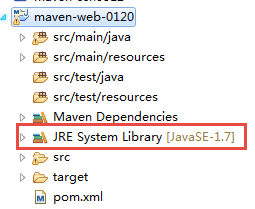
<version>2.2</version>

</plugin>

</plugins>

</pluginManagement>

</build>执行update project，查看编译版本为1.7：



### 第四步定义web.xml

在src/webapp中添加WEB-INF/web.xml文件，内容为：

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

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns=*"http://java.sun.com/xml/ns/javaee"* xmlns:web=*"http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*

id=*"WebApp\_ID"* version=*"2.5"*>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

</web-app>

### 第五步编写servlet

在src/main/java中创建ServletTest

**public** **class** ServletTest **extends** HttpServlet {

@Override

**protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {

**this**.doPost(req, resp);

}

@Override

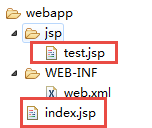
**protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {

req.getRequestDispatcher("/jsp/test.jsp").forward(req, resp);

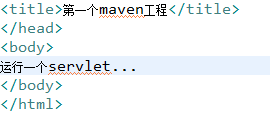
}

}

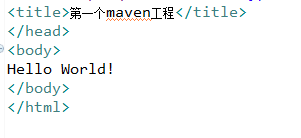
### 第六步编写jsp



test.jsp的内容如下：

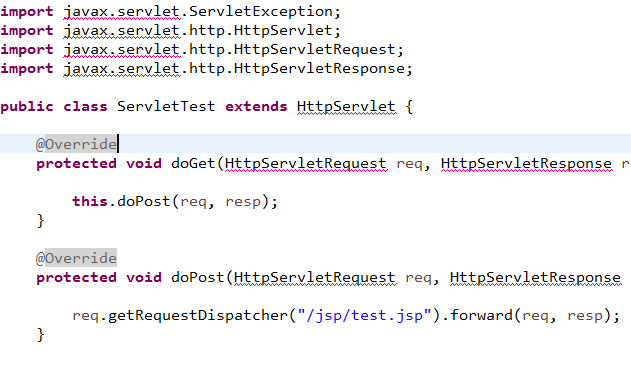


index.jsp的内容如下：



### 第七步添加servlet/jsp的jar包

servlet类中报错，无法解析javax.sevlet....



在maven工程中添加jar的方式是需要在pom.xml中添加servlet/jsp的坐标，maven自动从创建下载servlet/jsp的jar包

编辑pom.xml，如下：

<!-- 添加servlet-api，jsp-api -->

<dependencies>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jsp-api</artifactId>

<version>2.0</version>

<scope>provided</scope>

</dependency>

</dependencies>

### 第八步配置servlet

在web.xml中配置servlet，如下所示：

<!-- 配置servlet -->

<servlet>

<servlet-name>servletTest</servlet-name>

<servlet-class>com.qf.maven.servlet.ServletTest</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>servletTest</servlet-name>

<url-pattern>/test</url-pattern>

</servlet-mapping>

### 运行

cmd命令行进入工程目录，运行tomcat:run命令

### eclipse下使用maven命令

在eclipse下测试命令

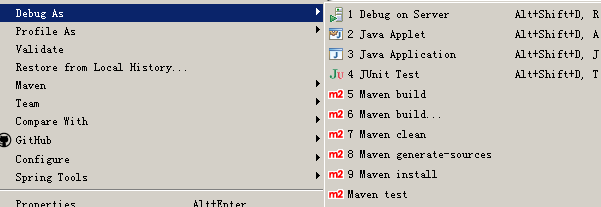
clean

compile

test

package

install



Run as 采用 mvn 命令运行 ，Debug as 采用 mvnDebug 命令调试方式运行（可打断点）

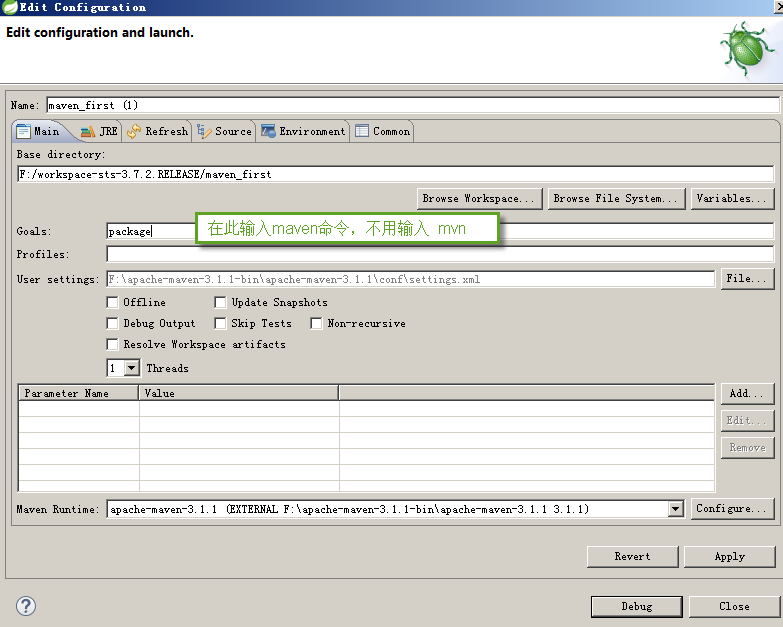
Maven clean 清理target目录

Maven test 执行单元测试

Maven install将工程打包后发布到本地仓库

Maven build 使用之前操作过的命令

Maven build … 手动输入命令内容，如下图：



* 编译命令 mvn compile

编译后 .class文件在 target/classes 下 （这个命令只会对java源程序编译， 不会编译测试代码 ， 编译测试类 mvn test-compile , 编译后.class 文件在 target\test-classes ）

* 测试命令 mvn test

执行所有测试用例方法， 重新编译

* 清除命令 mvn clean

清除target目录 （清除所有编译结果或者打包结果 ），清理后编译。

* 打包命名 mvn package

java项目生成 jar包， web项目生成war包

默认生成jar包名称 ： artifactId-version.jar

* 安装命令

安装命令 mvn install 将工程打包后发布到本地仓库

---- 安装到仓库/groupId/artifactId/version 目录

依赖管理-添加依赖

### 需求

SSM+Maven

### 添加依赖

在pom.xml中添加dependency标签，如下：

<dependency>

<groupId><groupId>

<artifactId></artifactId>

<version></version>

</dependency>

### 查找坐标

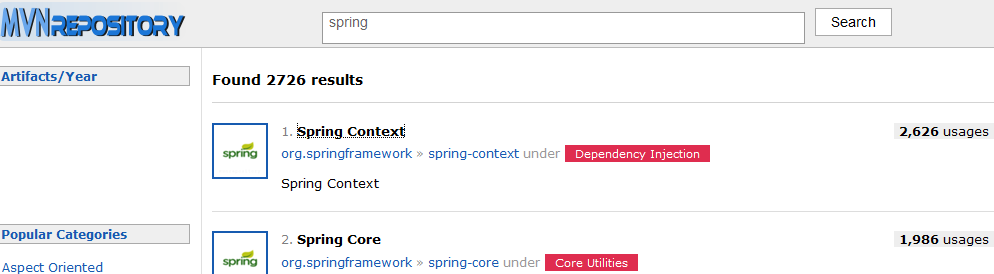
添加依赖需要指定依赖jar包的坐标，但是很多情况我们是不知道jar包的的坐标，可以通过如下方式查询：

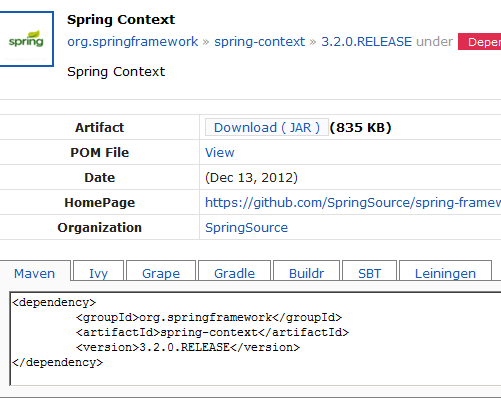
* 方法一：从互联网搜索

http://search.maven.org/

http://mvnrepository.com/

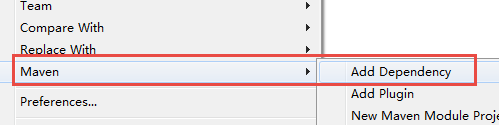
网站搜索示例：

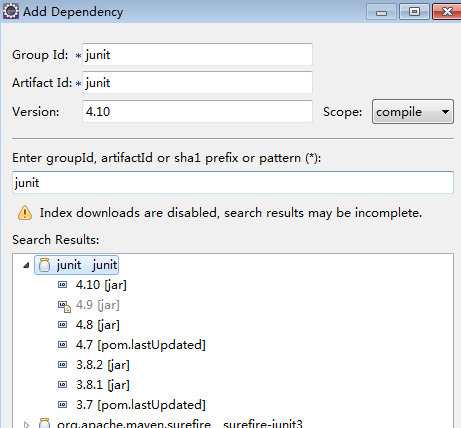




* 方法二：使用maven插件的索引功能

如果在本地仓库有我们要的jar包，可以在pom.xml中邮件添加依赖





### 构建web工程

**添加struts2依赖**

编写web工程的pom.xml文件，添加依赖，如下：

<!-- 依赖struts2 -->

<dependency>

<groupId>org.apache.struts</groupId>

<artifactId>struts2-core</artifactId>

<version>2.3.24</version>

</dependency>

**依赖范围**

**基本概念**

A依赖B，需要在A的pom.xml文件中添加B的坐标，添加坐标时需要指定依赖范围，依赖范围包括：

* compile：编译范围，指A在编译时依赖B，此范围为默认依赖范围。编译范围的依赖会用在编译、测试、运行，由于运行时需要所以编译范围的依赖会被打包。

编译、测试、运行、打包需要

* provided：provided依赖只有在当JDK或者一个容器已提供该依赖之后才使用， provided依赖在编译和测试时需要，在运行时不需要，比如：servlet api被tomcat容器提供。

编译和测试时需要

* runtime：runtime依赖在运行和测试系统的时候需要，但在编译的时候不需要。比如：jdbc的驱动包。由于运行时需要所以runtime范围的依赖会被打包。

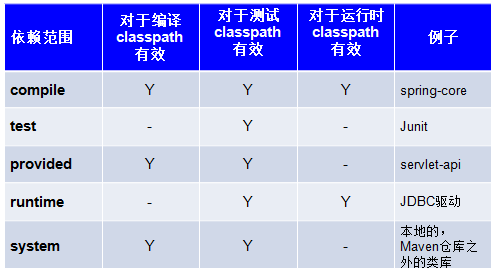
runtime:测试、运行、打包需要

* test：test范围依赖 在编译和运行时都不需要，它们只有在测试编译和测试运行阶段可用，比如：junit。由于运行时不需要所以test范围依赖不会被打包。

test:测试

* system：system范围依赖与provided类似，但是你必须显式的提供一个对于本地系统中JAR文件的路径，需要指定systemPath磁盘路径，system依赖不推荐使用。

system:编译和测试时需要,显示引用，Maven不会检索



在maven-web工程中测试各各scop。

测试总结：

* 默认引入 的jar包 ------- compile 【默认范围 可以不写】（编译、测试、运行 都有效 ）
* servlet-api 、jsp-api ------- provided （编译、测试 有效， 运行时无效 防止和tomcat下jar冲突）
* jdbc驱动jar包 ---- runtime （测试、运行 有效 ）
* junit ----- test （测试有效）

依赖范围由强到弱的顺序是：compile>provided>runtime>test

**测试**

1、向web工程 添加jsp/servlet依赖

<!-- servlet jsp -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jsp-api</artifactId>

<version>2.0</version>

<scope>provided</scope>

</dependency>

package打war观察jsp-api和servlet-api是否在war中存在？

2、向dao工程 添加jdbc依赖

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.6</version>

<scope>runtime</scope>

</dependency>

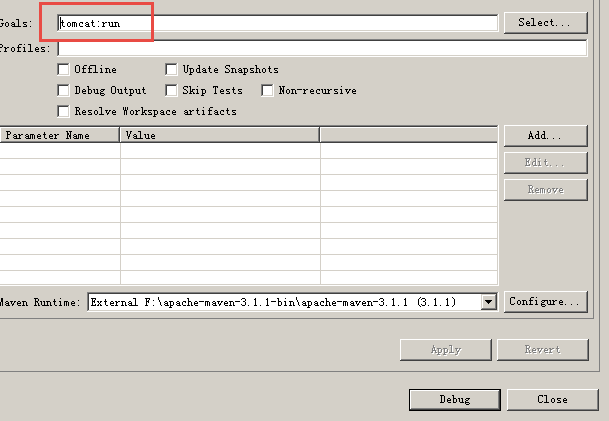
package打war观察mysql-connctor-java是否在war中存在？

完整的pom.xml

maven工程运行调试

### tomcat插件

maven内置tomcat的插件（org.codehaus.mojo. tomcat-maven-plugin），执行tomcat:run命令即可启动tomcat



访问：http://localhost:8080/工程名/...

可以通过配置plugin修改tomcat的访问路径及端口：

<build>

<plugins>

<!-- maven内置 的tomcat6插件 -->

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>tomcat-maven-plugin</artifactId>

<version>1.1</version>

<configuration>

<!-- 可以灵活配置工程路径 -->

<path>/ssh</path>

<!-- 可以灵活配置端口号 -->

<port>8080</port>

</configuration>

</plugin>

</plugins>

</build>

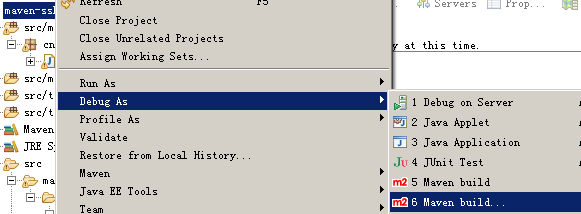
### 端口占用处理

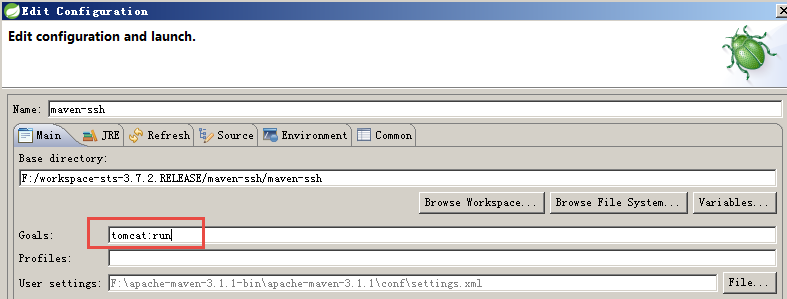
重新执行tomcat:run命令重启工程，重启之前需手动停止 tomcat，否则报下边的错误：

Caused by: java.net.BindException: Address already in use: JVM\_Bind

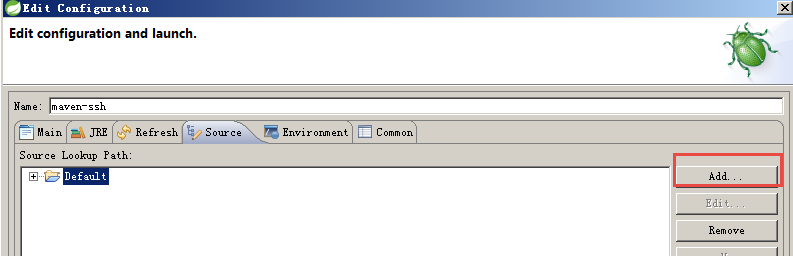
### 断点调试

maven工程断点调试必须采用“Debug As”方式启动，并且需要引入源码才可源码跟踪：

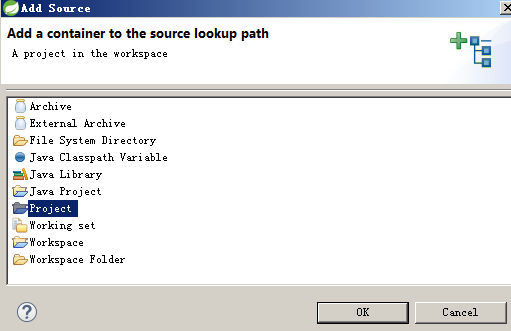


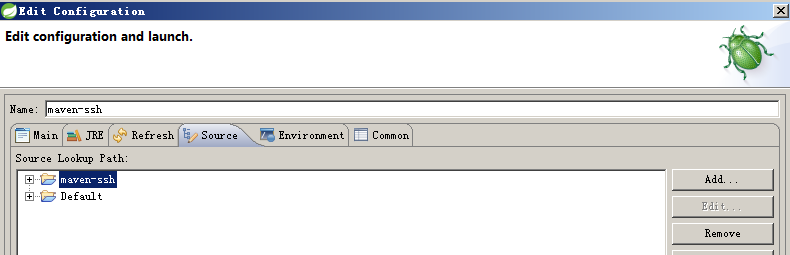


引入源码：

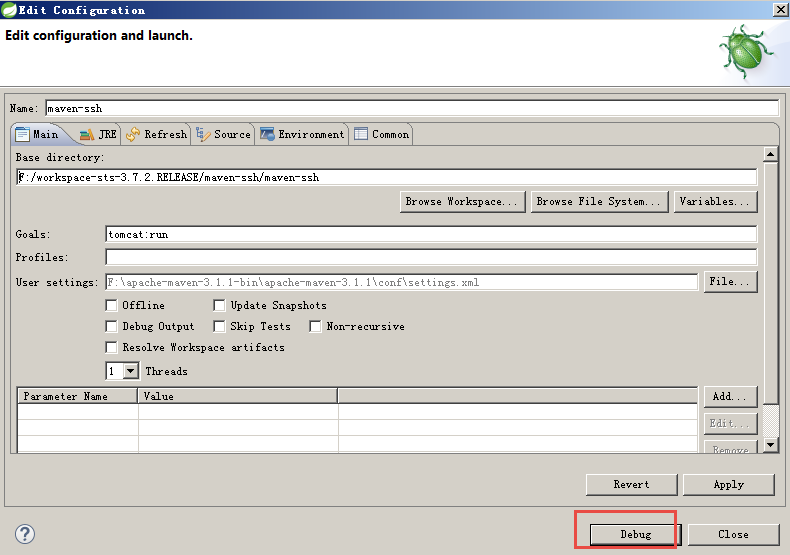


添加，选择本工程：





以debug方式运行：



总结

### maven仓库

1、maven仓库的类型有哪些？

2、maven工程查找仓库的流程是什么？

3、本地仓库如何配置？

### 常用的maven命令

常用 的maven命令包括：

compile：编译

clean：清理

test：测试

package：打包

install：安装

### 坐标定义

在pom.xml中定义坐标，内容包括：groupId、artifactId、version，详细内容如下：

<!--项目名称，定义为组织名+项目名，类似包名-->

<groupId>com.qf.maven</groupId>

<!-- 模块名称 -->

<artifactId>maven-first</artifactId>

<!-- 当前项目版本号，snapshot为快照版本即非正式版本，release为正式发布版本 -->

<version>0.0.1-SNAPSHOT</version>

<packaging > ：打包类型

jar：执行package会打成jar包

war：执行package会打成war包

pom ：用于maven工程的继承，通常父工程设置为pom

### pom基本配置

pom.xml是Maven项目的核心配置文件，位于每个工程的根目录，基本配置如下：

<project > ：文件的根节点 .  
<modelversion > ： pom.xml使用的对象模型版本  
<groupId > ：项目名称，一般写项目的域名  
<artifactId > ：模块名称，子项目名或模块名称  
<version > ：产品的版本号 .

<packaging > ：打包类型，一般有jar、war、pom 等   
<name > ：项目的显示名，常用于 Maven 生成的文档。   
<description > ：项目描述，常用于 Maven 生成的文档

<dependencies> ：项目依赖构件配置，配置项目依赖构件的坐标

<build> ：项目构建配置，配置编译、运行插件等。

### Maven项目工程目录约定

使用maven创建的工程我们称它为maven工程，maven工程具有一定的目录规范，如下：

src/main/java —— 存放项目的.java文件

src/main/resources —— 存放项目资源文件，如spring, hibernate配置文件

src/test/java —— 存放所有单元测试.java文件，如JUnit测试类

src/test/resources —— 测试资源文件

target —— 项目输出位置，编译后的class文件会输出到此目录

pom.xml——maven项目核心配置文件

Project

|-src

| |-main

| | |-java —— 存放项目的.java文件

| | |-resources —— 存放项目资源文件，如spring, hibernate配置文件

|-webapp —— webapp目录是web工程的主目录

|-WEB-INF

|-web.xml

| |-test

| |-java ——存放所有测试.java文件，如JUnit测试类

| |-resources —— 测试资源文件

|-target —— 目标文件输出位置例如.class、.jar、.war文件

|-pom.xml ——maven项目核心配置文件

### Maven管理项目周期

1、Clean:项目构建前的清理操作

2、Default：核心生命周期核心过程：编译、运行、打包等

3、Site：发布战点，生成报告等

### Maven的常用命令：

clean:清空

compile:编译

deploy:上传

test:单元测试

install：添加到本地仓库

tomcat:run 启动tomcat---默认启动Tomact6

package：打包

clean pagckeage:清理并打包

clean compile:清理并编译

实际中，我们会将一些庞大的项目拆分为若干模块进行开发

三层+MVC

拆分：

pojo-------jar

dao----------jar

service------jar

web----------war

### Maven的继承

开发中多个项目有很多一样的jar依赖，那么我们可以采用继承的方式简化各个项目的pom文件

在父类的pom文件依赖共同拥有的jar

注意点:

1、父级项目只能是pom项目

2、子项目是一个Maven Project

### Maven的聚合

Maven允许将一个项目按照多模块开发，整体打包发布

注意点：

1、根项目是一个pom项目

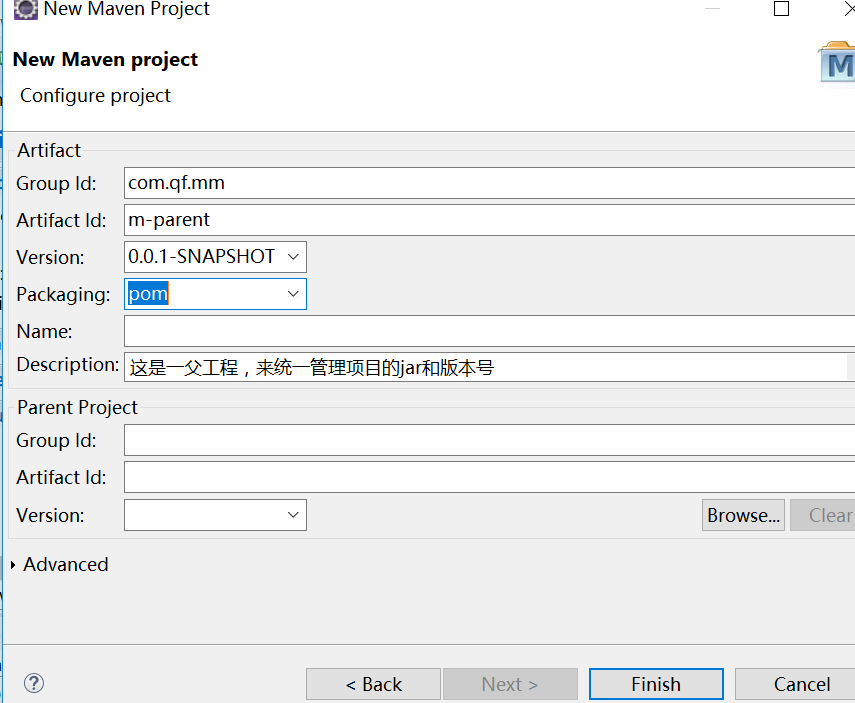
2、子模块是：Maven Model

Maven Model和Maven Project的区别？

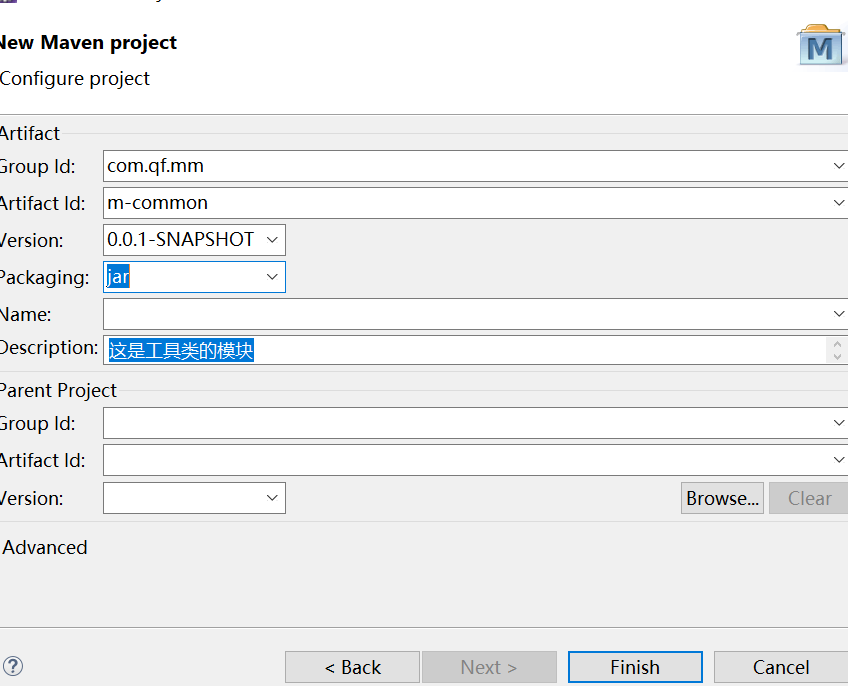
Maven Project独立运行

Maven Model无法独立运行

**第一步：创建父工程m-parent(来统一管理jar包（管理所有模块的公共的jar包）)：打包方式pom**

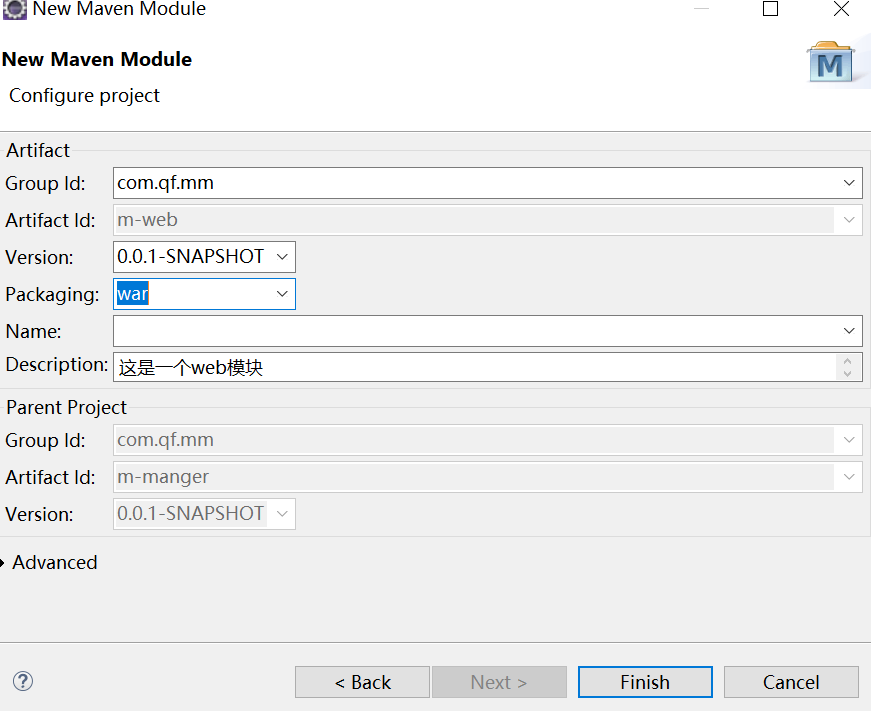


第二步：创建一个工具类的模块m-common： 打包方式 jar

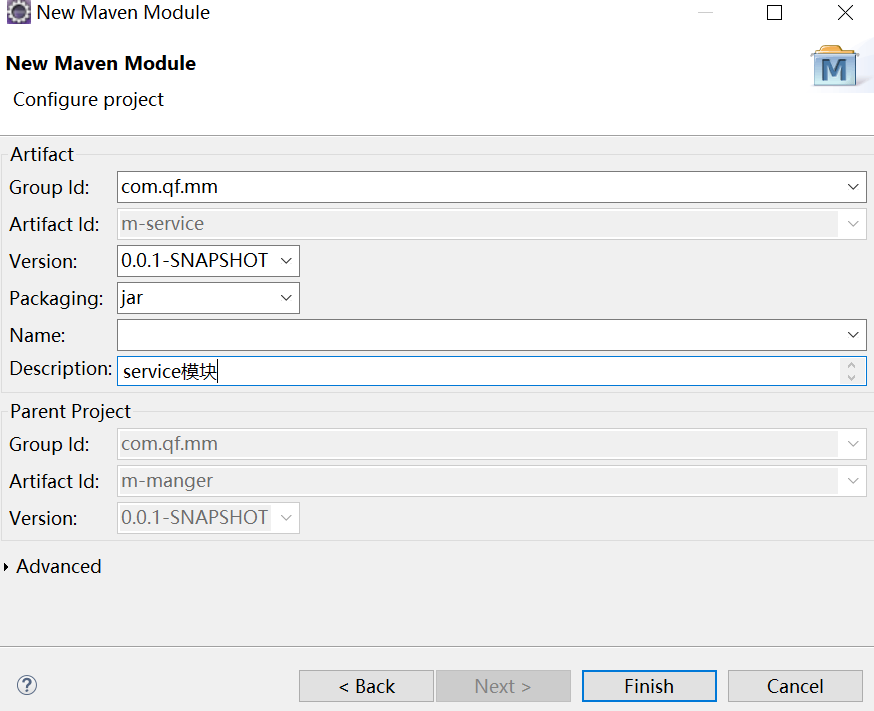


第三步：在建一个m-manger:pom工程把m-web,m-pojo,m-service,m-dao聚合在一个

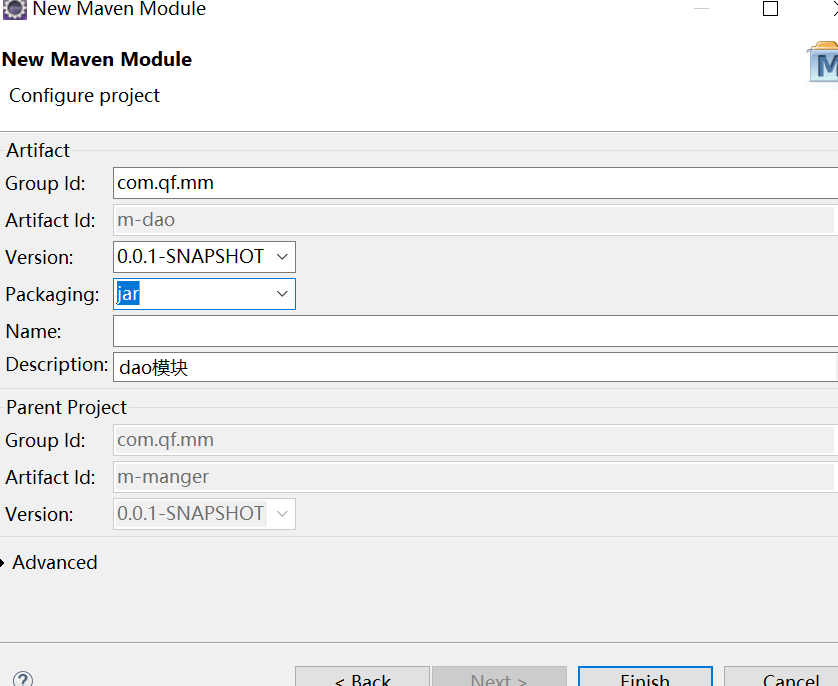
第四步：创建一个maven module 模块：m-web



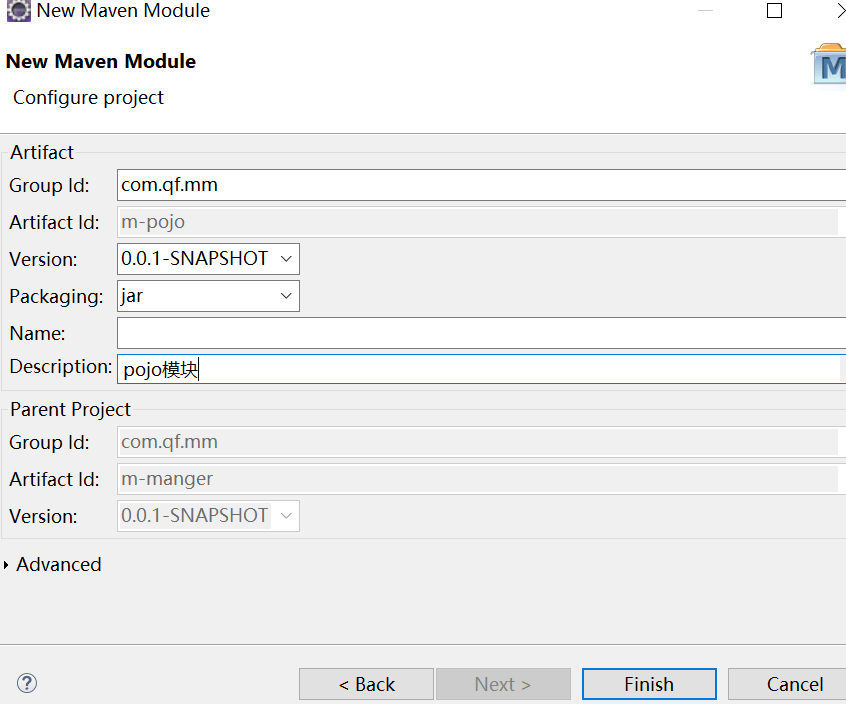
第五步：创建一个maven module 模块：m-service



第六步：创建一个maven module 模块：m-dao



第七步：创建一个maven module 模块：m-pojo



第八步：配置依赖关系：web层依赖service层，service依赖dao,dao依赖pojo,依赖具有传递性

第九步：SSM的配置文件放在m-web模块

第十步：配置web.xml文件

写代码 启动测试

8.MyBatis配置



1. mybatis配置

SqlMapConfig.xml，此文件作为mybatis的全局配置文件，配置了mybatis的运行环境等信息。

mapper.xml文件即sql映射文件，文件中配置了操作数据库的sql语句。此文件需要在SqlMapConfig.xml中加载。

1. 通过mybatis环境等配置信息构造SqlSessionFactory即会话工厂
2. 由会话工厂创建sqlSession即会话，操作数据库需要通过sqlSession进行。
3. mybatis底层自定义了Executor执行器接口操作数据库，Executor接口有两个实现，一个是基本执行器、一个是缓存执行器。
4. Mapped Statement也是mybatis一个底层封装对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中一个sql对应一个Mapped Statement对象，sql的id即是Mapped statement的id。
5. Mapped Statement对sql执行输入参数进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql前将输入的java对象映射至sql中，输入参数映射就是jdbc编程中对preparedStatement设置参数。
6. Mapped Statement对sql执行输出结果进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql后将输出结果映射至java对象中，输出结果映射过程相当于jdbc编程中对结果的解析处理过程。

mybatis下载

mybaits的代码由github.com管理，地址：https://github.com/mybatis/mybatis-3/releases



mybatis-3.2.7.jar----mybatis的核心包

lib----mybatis的依赖包

mybatis-3.2.7.pdf----mybatis使用手册

## 需求

实现以下功能：

根据用户id查询一个用户信息

根据用户名称模糊查询用户信息列表

添加用户

更新用户

删除用户

工程搭建

### 第一步：创建java工程

使用eclipse创建java工程

### 第二步：加入jar包

加入mybatis核心包、依赖包、数据驱动包。



### 第三步：log4j.properties

在classpath下创建log4j.properties如下：

# Global logging configuration

log4j.rootLogger=DEBUG, stdout

# Console output...

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

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

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

mybatis默认使用log4j作为输出日志信息。

### 第四步：SqlMapConfig.xml

在classpath下创建SqlMapConfig.xml，如下：

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

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!-- 和spring整合后 environments配置将废除-->

<environments default=*"development"*>

<environment id=*"development"*>

<!-- 使用jdbc事务管理-->

<transactionManager type=*"JDBC"* />

<!-- 数据库连接池-->

<dataSource type=*"POOLED"*>

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

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"root"* />

</dataSource>

</environment>

</environments>

</configuration>

SqlMapConfig.xml是mybatis核心配置文件，上边文件的配置内容为数据源、事务管理。

### 第五步：po类

Po类作为mybatis进行sql映射使用，po类通常与数据库表对应，User.java如下：

**Public class** User {

**private int** id;

**private** String username;// 用户姓名

**private** String sex;// 性别

**private** Date birthday;// 生日

**private** String address;// 地址

get/set……

### 第六步：sql映射文件

在classpath下的sqlmap目录下创建sql映射文件Users.xml：

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

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"test"*>

</mapper>

namespace ：命名空间，用于隔离sql语句。

### 第七步：加载映射文件

mybatis框架需要加载映射文件，将Users.xml添加在SqlMapConfig.xml，如下：

<mappers>

<mapper resource=*"sqlmap/User.xml"*/>

</mappers>

示例1:根据id查询用户信息

### 映射文件：

在user.xml中添加：

<!-- 根据id获取用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"cn.qf.mybatis.po.User"*>

select \* from user where id = #{id}

</select>

parameterType：定义输入到sql中的映射类型，#{id}表示使用preparedstatement设置占位符号并将输入变量id传到sql。

resultType：定义结果映射类型。

### 测试程序：

**public** **class** Mybatis\_first {

//会话工厂

**private** SqlSessionFactory sqlSessionFactory;

@Before

**public** **void** createSqlSessionFactory() **throws** IOException {

// 配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

// 使用SqlSessionFactoryBuilder从xml配置文件中创建SqlSessionFactory

sqlSessionFactory = **new** SqlSessionFactoryBuilder()

.build(inputStream);

}

// 根据 id查询用户信息

@Test

**public** **void** testFindUserById() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 查询单个记录，根据用户id查询用户信息

User user = sqlSession.selectOne("test.findUserById", 10);

// 输出用户信息

System.out.println(user);

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (sqlSession != **null**) {

sqlSession.close();

}

}

}

}

示例二:根据用户名查询用户信息

### 映射文件：

在user.xml中添加：

<!-- 自定义条件查询用户列表 -->

<select id=*"findUserByUsername"* parameterType=*"java.lang.String"*

resultType=*"cn.qf.mybatis.po.User"*>

select \* from user where username like '%${value}%'

</select>

parameterType：定义输入到sql中的映射类型，${value}表示使用参数将${value}替换，做字符串的拼接。

注意：如果是取简单数量类型的参数，括号中的值必须为value

resultType：定义结果映射类型。

### 测试程序：

// 根据用户名称模糊查询用户信息

@Test

**public** **void** testFindUserByUsername() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 查询单个记录，根据用户id查询用户信息

List<User> list = sqlSession.selectList("test.findUserByUsername", "张");

System.*out*.println(list.size());

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (sqlSession != **null**) {

sqlSession.close();

}

}

}

### 小结

### 5.6.1 parameterType和resultType

parameterType指定输入参数的java类型，可以填写别名或Java类的全限定名。

resultType指定输出结果的java类型，可以填写别名或Java类的全限定名。

### #{}和${}

#{}表示一个占位符号，通过#{}可以实现preparedStatement向占位符中设置值，自动进行java类型和jdbc类型转换。#{}可以有效防止sql注入。 #{}可以接收简单类型值或pojo属性值。 如果parameterType传输单个简单类型值，#{}括号中可以是value或其它名称。

${}表示拼接sql串，通过${}可以将parameterType 传入的内容拼接在sql中且不进行jdbc类型转换， ${}可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，${}括号中只能是value。

### selectOne和selectList

selectOne查询一条记录，如果使用selectOne查询多条记录则抛出异常：

org.apache.ibatis.exceptions.TooManyResultsException: Expected one result (or null) to be returned by selectOne(), but found: 3

at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:70)

selectList可以查询一条或多条记录。

9.SSH整合配置

**Spring4+hibernate5.2.10+SpringMvc+maven整合**

1.创建maven项目，编写pom文件

**把整合框架的相关jar包全都引入**

**<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">**

**<modelVersion>4.0.0</modelVersion>**

**<groupId>com.qf</groupId>**

**<artifactId>ssh</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**<packaging>war</packaging>**

**<properties>**

**<spring.version>4.2.8.RELEASE</spring.version>**

**<gson.version>2.8.0</gson.version>**

**<servlet.version>3.1.0</servlet.version>**

**<jsp.version>2.1</jsp.version>**

**<mysql.version>5.1.38</mysql.version>**

**<c3p0.version>0.9.1.2</c3p0.version>**

**<log4j.version>1.2.17</log4j.version>**

**<jstl.version>1.2</jstl.version>**

**<jackson.version>2.8.9</jackson.version>**

**<fileupload.version>1.3.1</fileupload.version>**

**<common.io.version>2.4</common.io.version>**

**<hibernate.version>5.2.10.Final</hibernate.version>**

**<project.encoding>UTF-8</project.encoding>**

**<fastjson.version>1.2.46</fastjson.version>**

**</properties>**

**<!-- 配置所有依赖 -->**

**<dependencies>**

**<!-- 配置servlet相关 -->**

**<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->**

**<dependency>**

**<groupId>javax.servlet</groupId>**

**<artifactId>javax.servlet-api</artifactId>**

**<version>${servlet.version}</version>**

**<scope>provided</scope>**

**</dependency>**

**<!-- 配置JSP相关 -->**

**<!-- https://mvnrepository.com/artifact/javax.servlet.jsp/jsp-api -->**

**<dependency>**

**<groupId>javax.servlet.jsp</groupId>**

**<artifactId>jsp-api</artifactId>**

**<version>${jsp.version}</version>**

**<scope>provided</scope>**

**</dependency>**

**<!-- 配置JSTL相关 -->**

**<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->**

**<dependency>**

**<groupId>javax.servlet</groupId>**

**<artifactId>jstl</artifactId>**

**<version>${jstl.version}</version>**

**</dependency>**

**<!-- 配置数据库驱动相关 -->**

**<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->**

**<dependency>**

**<groupId>mysql</groupId>**

**<artifactId>mysql-connector-java</artifactId>**

**<version>${mysql.version}</version>**

**</dependency>**

**<!-- 配置spring相关 -->**

**<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-context</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<!-- 配置事务相关 -->**

**<!-- https://mvnrepository.com/artifact/org.springframework/spring-tx -->**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-tx</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<!-- 配置spring-jdbc -->**

**<!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-jdbc</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<!—配置spring-test -->**

**<!-- https://mvnrepository.com/artifact/org.springframework/spring-test -->**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-test</artifactId>**

**<version>${spring.version}</version>**

**<scope>test</scope>**

**</dependency>**

**<!-- 配置AspectJ 相关 -->**

**<!-- https://mvnrepository.com/artifact/org.springframework/spring-aspects -->**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-aspects</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<!-- https://mvnrepository.com/artifact/org.springframework/spring-context-support -->**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-context-support</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<!-- 配置SpringMvc相关 -->**

**<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-webmvc</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<!-- 配置Log4J相关 -->**

**<!-- https://mvnrepository.com/artifact/log4j/log4j -->**

**<dependency>**

**<groupId>log4j</groupId>**

**<artifactId>log4j</artifactId>**

**<version>${log4j.version}</version>**

**</dependency>**

**<!-- c3p0连接池 -->**

**<!-- https://mvnrepository.com/artifact/com.mchange/c3p0 -->**

**<!-- https://mvnrepository.com/artifact/c3p0/c3p0 -->**

**<dependency>**

**<groupId>c3p0</groupId>**

**<artifactId>c3p0</artifactId>**

**<version>${c3p0.version}</version>**

**</dependency>**

**<!-- 配置Gson -->**

**<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->**

**<dependency>**

**<groupId>com.google.code.gson</groupId>**

**<artifactId>gson</artifactId>**

**<version>${gson.version}</version>**

**</dependency>**

**<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->**

**<!-- SpringMvc内置的jackson -->**

**<dependency>**

**<groupId>com.fasterxml.jackson.core</groupId>**

**<artifactId>jackson-databind</artifactId>**

**<version>${jackson.version}</version>**

**</dependency>**

**<!-- 配置文件上传相关 -->**

**<dependency>**

**<groupId>commons-fileupload</groupId>**

**<artifactId>commons-fileupload</artifactId>**

**<version>${fileupload.version}</version>**

**</dependency>**

**<!-- https://mvnrepository.com/artifact/commons-io/commons-io -->**

**<dependency>**

**<groupId>commons-io</groupId>**

**<artifactId>commons-io</artifactId>**

**<version>${common.io.version}</version>**

**</dependency>**

**<!-- 配置hibernate相关 -->**

**<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->**

**<dependency>**

**<groupId>org.hibernate</groupId>**

**<artifactId>hibernate-core</artifactId>**

**<version>${hibernate.version}</version>**

**</dependency>**

**<!-- 配置Spring整合Hibernate相关 -->**

**<!-- https://mvnrepository.com/artifact/org.springframework/spring-orm -->**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-orm</artifactId>**

**<version>${spring.version}</version>**

**</dependency>**

**<!-- 配置fastJson相关 -->**

**<!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->**

**<dependency>**

**<groupId>com.alibaba</groupId>**

**<artifactId>fastjson</artifactId>**

**<version>${fastjson.version}</version>**

**</dependency>**

**</dependencies>**

**<!-- 配置插件相关 -->**

**<!-- 设置项目的编译版本 -->**

**<build>**

**<plugins>**

**<plugin>**

**<groupId>org.apache.maven.plugins</groupId>**

**<artifactId>maven-compiler-plugin</artifactId>**

**<configuration>**

**<source>1.8</source>**

**<target>1.8</target>**

**<encoding>${project.encoding}</encoding>**

**</configuration>**

**</plugin>**

**</plugins>**

**<pluginManagement>**

**<plugins>**

**<!-- 配置Tomcat插件 tomcat7:run tomcat:run -->**

**<plugin>**

**<groupId>org.apache.tomcat.maven</groupId>**

**<artifactId>tomcat7-maven-plugin</artifactId>**

**<version>2.2</version>**

**<configuration>**

**<path>/ssh</path>**

**<port>8080</port>**

**<uriEncoding>${project.encoding}</uriEncoding>**

**</configuration>**

**</plugin>**

**</plugins>**

**</pluginManagement>**

**</build>**

**</project>**

2.把一些必要的配置文件准备好

1.jdbc.properties

​ 2.log4j.properties

​ 3.Springmvc的配置文件

​ 4.hibernate的配置文件

​ （不使用jpa的注解 hibernate的映射文件。）

​ 5.web.xml配置文件

​ 6.Spring的配置文件

3. Spring和hibernate和SpringMvc开始进行整合

hibernate是dao层的框架SpringMvc是web层框架

Spring是一个容器管理，用来管理和集成其他层的框架的

### 3.1.Spring整合web层

spring的配置用监听器加载。

SpringMVC的配置文件让dispatcherServlet去加载：

#### SpringMVC的配置文件：spring-mvc.xml

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

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

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

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

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

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

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

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

<!-- 开发SpringMvc的配置文件 -->

<!-- 开启组件扫描 -->

<context:component-scan base-package="com.qf.controller"></context:component-scan>

<mvc:annotation-driven></mvc:annotation-driven>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/jsp/"></property>

<property name="suffix" value=".jsp" ></property>

</bean>

<!-- 用来过滤静态资源 -->

<mvc:resources location="/static/" mapping="/static/\*\*"></mvc:resources>

<!-- 配置文件上传相关 -->

<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver"

p:defaultEncoding="utf-8"

p:maxUploadSize="1048576"

p:uploadTempDir="file:D:\temp"

>

</bean>

</beans>

#### web.xml配置

<servlet>

<servlet-name>DispatcherServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<!-- contextConfigLocation固定 -->

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

<param-value>classpath:applicationContext-springmvc.xml</param-value>

</init-param>

<!-- 启动服务器就实例化 -->

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>DispatcherServlet</servlet-name>

<!-- 1./\* 拦截所有请求 css img js jsp html action 2.\*.action/\*.do \*.action 可以拦截以.action结尾的请求 3./ 拦截css js image action请求 不拦截jsp(推荐) -->

<url-pattern>/</url-pattern>

</servlet-mapping>

<!-- SPring解决POST请求乱码 -->

<filter>

<filter-name>characterEncodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

<init-param>

<param-name>forceEncoding</param-name>

<param-value>true</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>characterEncodingFilter</filter-name>

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

</filter-mapping>

<!-- form表单只支持 Get和POST 如果想支持rest风格 得配置该过滤器 然后该过滤器会帮我们实现 -->

<filter>

<filter-name>HiddenHttpMethodFilter</filter-name>

<filter-class>

org.springframework.web.filter.HiddenHttpMethodFilter

</filter-class>

</filter>

<filter-mapping>

<filter-name>HiddenHttpMethodFilter</filter-name>

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

</filter-mapping>

#### Spring的配置文件org.springframework.web.context.ContextLoaderListener该监听器去加载

<listener>

<!-- 默认是去WEB-INF下找 applicationContext.xml配置文件-->

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

<context-param>

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

<param-value>classpath:applicationContext-\*.xml</param-value>

</context-param>

### 3.2 Spring整合dao层

就是整合hibernate（hibernate.cfg.xml配置文件不要了。里面的东西要全部写到Spring配置文件中去）,applicationContext-dao.xml,配置数据源，整合hibernate

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

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

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

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

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

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

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

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

<!-- 开发SpringMvc的配置文件 -->

<!-- 开启组件扫描 -->

<context:component-scan base-package="com.qf.dao"></context:component-scan>

<!-- 加载jdbc.properties配置文件 -->

<context:property-placeholder location="classpath:jdbc.properties"/>

<!-- 配置C3P0连接池 -->

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

<property name="user" value="${jdbc.username}"></property>

<property name="password" value="${jdbc.password}"></property>

<property name="jdbcUrl" value="${jdbc.url}"></property>

<property name="driverClass" value="${jdbc.driver}"></property>

<property name="maxPoolSize" value="${jdbc.maxPoolSize}"></property>

<property name="initialPoolSize" value="${jdbc.initialPoolSize}"></property>

</bean>

<!-- 配置SessionFactory -->

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

<!-- 注入数据源 -->

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

<property name="hibernateProperties">

<props>

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

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

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

<prop key="hibernate.dialect">org.hibernate.dialect.MySQL5InnoDBDialect</prop>

<prop key="hibernate.connection.isolation">4</prop>

<!-- session与当前线程绑定不能配置 事务交给了spring管理 -->

</props>

</property>

<!-- xml方式 -->

<!-- <property name="mappingResources">

<list>

<value>User1.hb.xml</value>

<value>User2.hb.xml</value>

<value>User3.hb.xml</value>

<value>User4.hb.xml</value>

</list>

</property> -->

<!-- 引入映射文件 -->

<!-- <property name="mappingLocations">

<value>classpath:/com/pojo/\*.hbm.xml</value>

</property>

-->

<!-- 扫描实体类所在的包 -->

<property name="packagesToScan">

<list>

<value>com.qf.model</value>

</list>

</property>

</bean>

</beans>

### 3.3.Spring整合Service层

applicationContext-service.xml配置事务相关

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

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

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

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

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

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

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

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

<!-- 开发SpringMvc的配置文件 -->

<!-- 开启组件扫描 -->

<context:component-scan base-package="com.qf.service"></context:component-scan>

<!-- 配置事务 -->

<!-- 配置平台事务管理器 -->

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

<!-- 现在注入SessionFactory -->

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

</bean>

<!-- 配置事务通知 -->

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

<tx:attributes>

<tx:method name="select\*" propagation="SUPPORTS" read-only="true"/>

<tx:method name="query\*" propagation="SUPPORTS" read-only="true"/>

<tx:method name="find\*" propagation="SUPPORTS" read-only="true"/>

<tx:method name="search\*" propagation="SUPPORTS" read-only="true"/>

<tx:method name="get\*" propagation="SUPPORTS" read-only="true"/>

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

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

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

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

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

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

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

</tx:attributes>

</tx:advice>

<!-- 配置aop -->

<aop:config proxy-target-class="true">

<aop:pointcut expression="execution( public \* com.qf.service.impl.\*.\*(..))" id="cut"/>

<aop:advisor advice-ref="txAdvice" pointcut-ref="cut"/>

</aop:config>

</beans>

10.SSM整合配置

Spring整合Mybatis

### 整合Service层

applicationContext-service.xml

配置平台事务管理器：

<!-- 配置事务 -->

<!-- 配置平台事务管理器:org.springframework.jdbc.datasource.DataSourceTransactionManager(jdbc/Mybatis) -->

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

<!-- 现在注入dataSource -->

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

</bean>

引入pom文件

Mybatis整合Spring的jar和Mybatis的jar

<!-- https://mvnrepository.com/artifact/org.mybatis/mybatis-spring -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis-spring</artifactId>

<version>1.3.1</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.4.5</version>

</dependency>

### **保留SqlMapConfig.xml配置文件**

SqlMapConfig.xml里面的连接池和事务都去掉，要交给Spring来管理，加载mapper文件和配置别名等其他的配置写在SqlMapConfig.xml配置文件中

1.SqlMapConfig.xml配置文件：

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

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!-- 配置自定义别名 -->

<typeAliases>

<package name="com.model"/>

</typeAliases>

<mappers>

<!-- <mapper resource="com/dao/UserMapper.xml"/> -->

<!-- 注意：此种方法要求mapper接口名称和mapper映射文件名称相同，且放在同一个目录中。 -->

<package name="com.dao"/>

</mappers>

</configuration>

2.dao层文件配置SqslSessionFactory:org.mybatis.spring.SqlSessionFactoryBean：

<!-- 整合Mybatis相关 -->

<!-- 配置SqlSessionFactory -->

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<!-- 注入数据源 -->

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

<!-- 加载SqlMapConfig.xml配置文件 -->

<property name="configLocation" value="classpath:SqlMapConfig.xml"></property>

</bean>

3.配置org.mybatis.spring.mapper.MapperFactoryBean

配置Mapper接口动态生成实现累的配置

org.mybatis.spring.mapper.MapperFactoryBean:mybatis整合spring的包给我们提供了一个工厂类，来获取mapper接口的实现类的

<bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">

<!-- 注入接口 -->

<property name="mapperInterface" value="com.ssm.mapper.UserMapper"></property>

<!-- 注入sqlsessionFactory -->

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

</bean>

4.配置Mapper自动扫描器

org.mybatis.spring.mapper.MapperScannerConfigurer MapperScannerConfigurer

<!-- 配置Mapper接口扫描 -->

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<!-- 配置要扫描的包 -->

<property name="basePackage" value="com.ssm.mapper"></property>

<!-- 如果只有一个数据源 可以不用配置 推荐配置注入sqlSessionFactory-->

<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"></property>

</bean>

### 不保留SqlMapConfig配置文件

1配置SqslSessionFactory: org.mybatis.spring.SqlSessionFactoryBean：

<!-- 整合Mybatis相关 -->

<!-- 配置SqlSessionFactory -->

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<!-- 注入数据源 -->

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

<!-- 引入mapper文件 配置别名 -->

<property name="mapperLocations"value="classpath:com/ssm/mapper/\*.xml"></property>

<property name="typeAliasesPackage" value="com.ssm.model"></property>

</bean>

2配置Mapper自动扫描器MapperScannerConfigurer

<!-- 配置Mapper接口扫描 -->

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<!-- 配置要扫描的包 -->

<property name="basePackage" value="com.ssm.mapper"></property>

<!-- 如果只有一个数据源 可以不用配置 推荐配置注入sqlSessionFactory-->

<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"></property>

</bean>

3.配置平台事务管理器

<!-- 配置事务 -->

<!-- 配置平台事务管理器 -->

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

<!-- 现在注入Datadource -->

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

</bean>

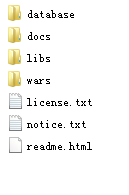
11.activiti整合配置

## 搭建Activti开发环境

### 1、下载activiti

|  |
| --- |
| 我们可以在Activiti的官网上直接下载开发需要的相关jar包  指定官网地址：<http://activiti.org/download.html>  建议使用开发工具为eclipse和IntelliJ IDEA，所需的JDK为1.6+以上。  构建Activiti工程可以使用两种方式，一种是拷贝jar包，一种是使用maven的构建方式。  Activiti构建工程是依赖于spring的环境，故在下载的包中出现了Activiti-spring相关的jar包。 |

5.18如下：



1. **Database：**

activiti运行需要有数据库的支持，支持的数据库有：h2, mysql, oracle, postgres, mssql, db2等，该目录存放activiti的建表脚本。

1. **Docs**

Activiti的帮助文档。

1. **Libs**

Activiti所需要的jar 包

1. **Wars**

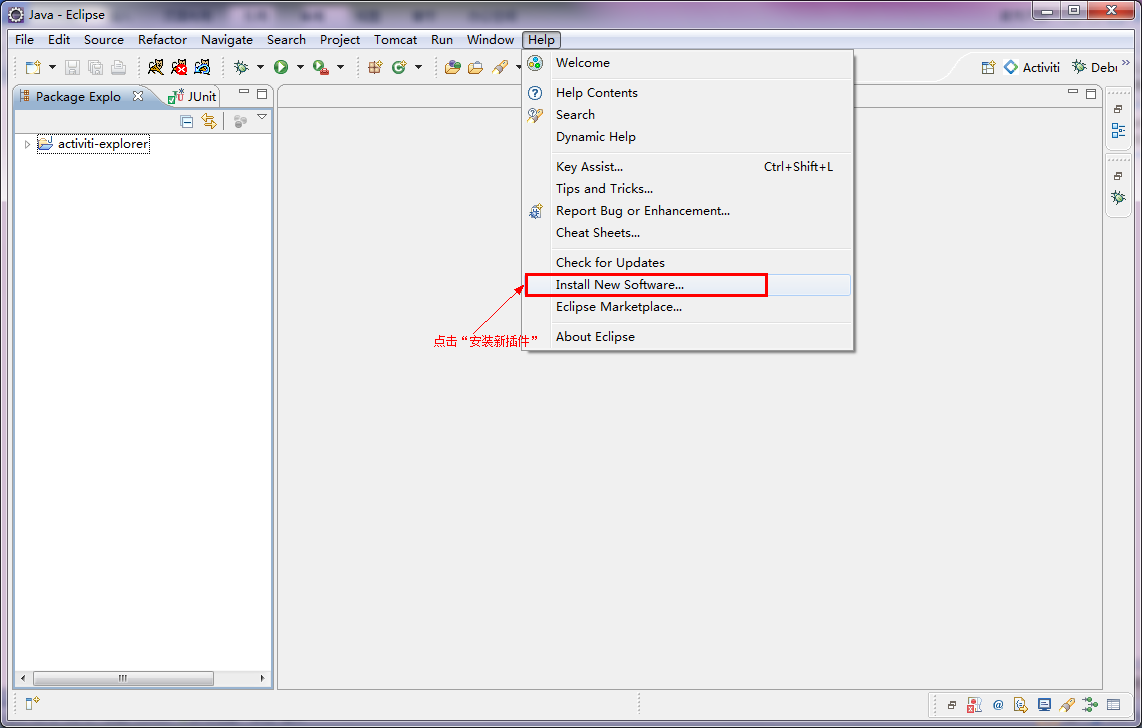
官方自带的示例工程。

### 2、Activiti Designer流程设计器

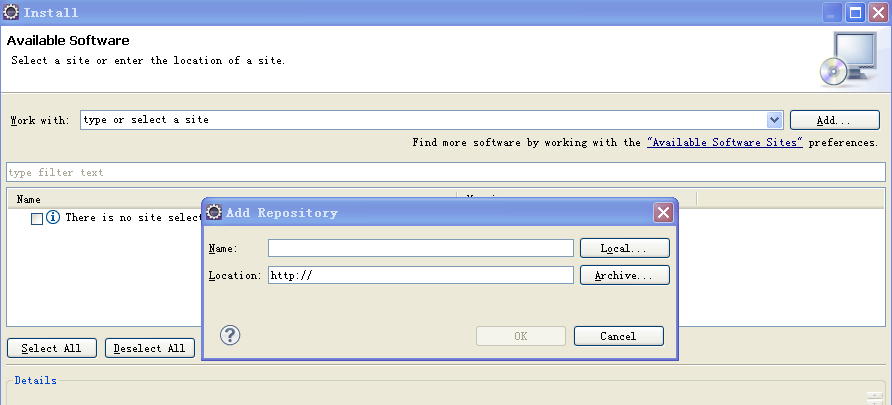
使用Activiti –Designer-eclipse-plugin（activiti流程设计器插件）完成流程的制作。

**网络在线安装**：

1. 打开 Help -> Install New Software. 在如下面板中:



1. 在如下Install界面板中，点击Add按钮：

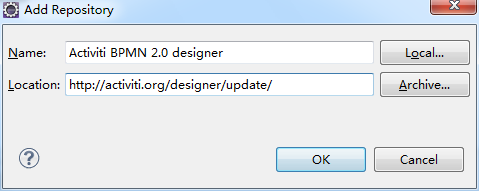


配置新装插件的地址和名称

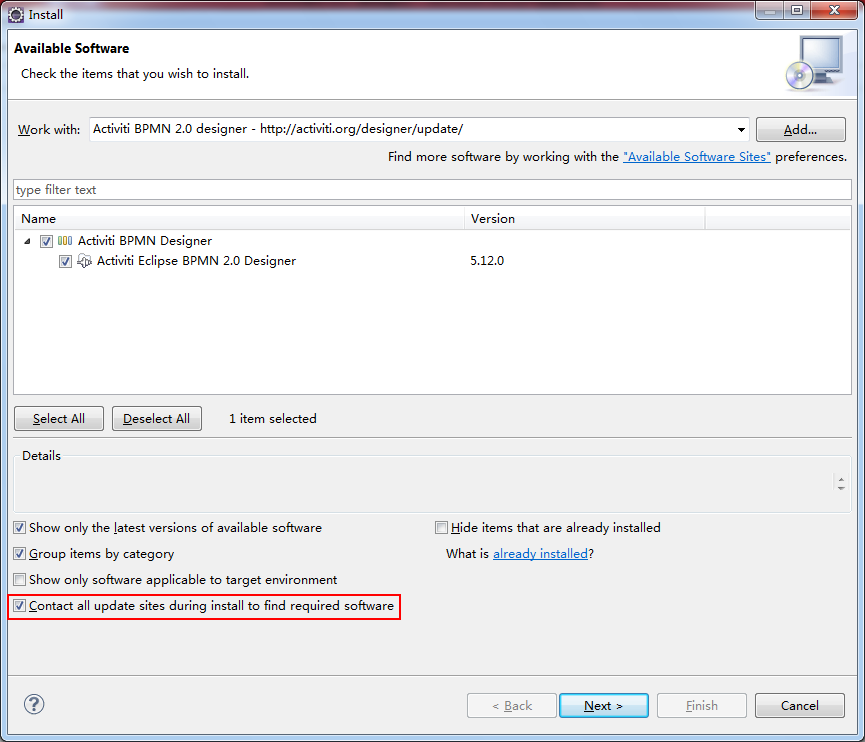
1. 然后填入下列字段

Name: Activiti BPMN 2.0 designer

Location: http://activiti.org/designer/update/



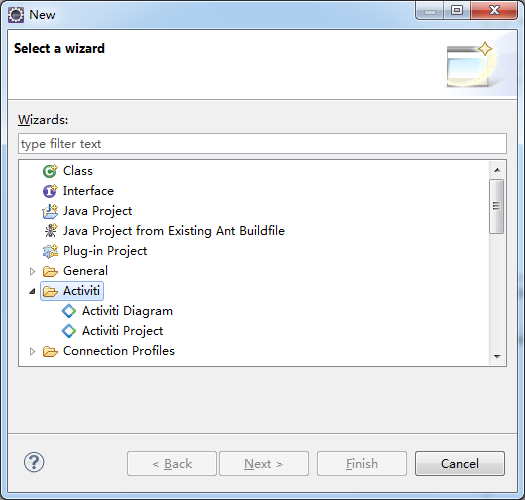
1. 回到Install界面，在面板正中列表中把所有展示出来的项目都勾上：



1. 点击复选框

在Detail部分记得选中 "Contact all updates sites.." , 因为它会检查所有当前安装所需要的插件并可以被Eclipse下载.

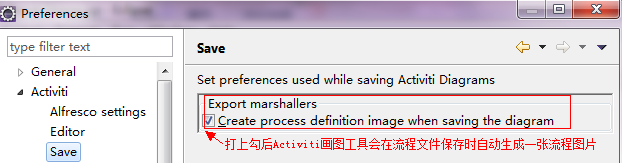
1. 安装完以后，点击新建工程new->Other…打开面板，如果看到下图内容：



说明安装成功了。

#### 补充说明

打开菜单Windows->Preferences->Activiti->Save下流程图片的生成方式:



虽然流程引擎在单独部署bpmn文件时会自动生成图片，但在实际开发过程中，自动生成的图片会导致和BPMN中的坐标有出入，在实际项目中展示流程当前位置图会有问题。

所在完成以上配置后，会由我们自己来管理流程图片。在发布流程时把流程规则文件和流程图片一起上传就行了。

### 3、Activiti所支持数据库类型

Activiti的运行需要数据库支撑，需要安装activiti数据库，支持如下版本：

| Activiti数据库类型 | JDBC URL实例 | 备注 |
| --- | --- | --- |
| h2 | jdbc:h2:tcp://localhost/activiti | 默认配置的数据库 |
| mysql | jdbc:mysql://localhost:3306/activiti?autoReconnect=true | 使用mysql-connector-java驱动测试 |
| oracle | jdbc:oracle:thin:@localhost:1521:xe |  |
| postgres | jdbc:postgresql://localhost:5432/activiti |  |
| db2 | jdbc:db2://localhost:50000/activiti |  |
| mssql | jdbc:sqlserver://localhost:1433/activiti |  |

### 4、创建数据库、数据表

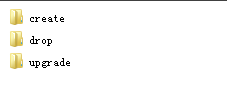
#### 1、创建数据库名：activiti

#### 2、创建表：

#### 3、创建表的方式一：

可通过官方提供的数据库脚本创建表

进入activiti5.18下database目录：



Create目录存放了数据库创建脚本

Drop目录存放了数据库删除脚本

Upgrade目录存放了数据库升级脚本

手动执行Create目录的建库脚本即可完成数据库的创建。

**（本教程采用下边介绍的程序创建数据库方式 ）**

#### 4、创建表的方式二：通过运行java程序创建表。

##### 1、使用eclipse创建maven工程项目。

##### 2、创建pom.xml文件配置环境所需要的依赖包

|  |
| --- |
| <properties>  <activiti-version>5.18.0</activiti-version>  </properties>  <dependencies>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.44</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-engine</artifactId>  <version>${activiti-version}</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-spring</artifactId>  <version>${activiti-version}</version>  </dependency>  <dependency>  <groupId>org.codehaus.groovy</groupId>  <artifactId>groovy-all</artifactId>  <version>2.4.3</version>  </dependency>  <dependency>  <groupId>com.h2database</groupId>  <artifactId>h2</artifactId>  <version>1.3.168</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.7.6</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-jdk14</artifactId>  <version>1.7.6</version>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.11</version>  </dependency>  </dependencies> |

##### 3、创建log4j配置文件

|  |
| --- |
| # Global logging configuration  log4j.rootLogger=DEBUG, stdout  # Console output...  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n |

##### 4、创建activiti.cfg.xml

|  |
| --- |
| <?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:p="http://www.springframework.org/schema/p"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:mvc="http://www.springframework.org/schema/mvc"  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  http://www.springframework.org/schema/mvc  http://www.springframework.org/schema/mvc/spring-mvc.xsd"> <!--配置数据源-->  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  <!--配置数据库的中文乱码-->  <property name="url" value="jdbc:mysql://localhost:3306/activiti?useUnicode=true&amp;characterEncoding=UTF-8"/>  <property name="username" value="root"/>  <property name="password" value="root"/>  <property name="maxActive" value="3"/>  <property name="maxIdle" value="1"/>  </bean>  <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">  <property name="dataSource" ref="dataSource"/>  <property name="databaseSchemaUpdate" value="true"/>  </bean>  </beans> |

processEngineConfiguration用来创建ProcessEngine，在创建ProcessEngine时会执行数据库的操作。

关于*processEngineConfiguration*中的**databaseSchemaUpdate参数，**通过此参数设计activiti数据表的处理策略，参数如下：

false（默认）：检查数据库表的版本和依赖库的版本， 如果版本不匹配就抛出异常。

true: 构建流程引擎时，执行检查，如果需要就执行更新。 如果表不存在，就创建。

create-drop: 构建流程引擎时创建数据库表， 关闭流程引擎时删除这些表。

drop-create：先删除表再创建表。

create: 构建流程引擎时创建数据库表， 关闭流程引擎时不删除这些表。

### 5、编写程序

创建ProcessEngineConfiguration，通过ProcessEngineConfiguration创建ProcessEngine，在创建ProcessEngine时会自动创建数据库。

|  |
| --- |
| //创建ProcessEngineConfiguration  ProcessEngineConfiguration configuration = ProcessEngineConfiguration  .*createProcessEngineConfigurationFromResource*("activiti.cfg.xml")  //通过ProcessEngineConfiguration创建ProcessEngine，此时会创建数据库  ProcessEngine processEngine = configuration.buildProcessEngine();  System.*out*.println(processEngine); |

**说明：**

1、运行以上程序段即可完成activiti数据库创建，通过改变activiti.cfg.xml中

databaseSchemaUpdate参数的值执行不同的数据表处理策略。

2、上边的方法*createProcessEngineConfigurationFromResource在执行时在activiti.cfg.xml中找固定的名称**processEngineConfiguration*

也可以使用重载方法调用，这时可以不用限定*processEngineConfiguration*名称



### 6、数据库表的命名规则

Activiti一共有23张表其中所有的表都是以ACT\_.\*开始的。

* **ACT\_RE\_** *：*RE*代表repository。具有此前缀的表包含*静态\*信息，例如流程定义和流程资源（图像，规则等）。
* **ACT\_RU\_** *：*RU\*代表runtime。这些是包含流程实例，用户任务，变量，作业等的运行时数据的运行时表。Activiti仅在流程实例执行期间存储运行时数据，并在进程实例结束时删除记录。这使得运行时间表保持不间断的快速。
* **ACT\_ID\_** *：*ID\*代表identity。这些表包含身份信息，如用户，组等。
* **ACT\_HI\_** *：*HI\*代表history。这些是包含历史数据的表，例如过去的流程实例，变量，任务等。
* **ACT\_GE\_** \*：general数据，用于各种用例。

**Activiti整合SSM**

* **创建maven项目编写pom文件:**
* **创建包的结构**
* **编写SpringMVc的配置文件：spring-mvc.xml**
* **编写applicationContext-dao.xml、applicationContext-service.xml、jdbc.properties、logback.xml等配置文件**
* **（参考SSM整合）**

### ****POM依赖****

|  |
| --- |
| <properties>  <spring.version>4.2.8.RELEASE</spring.version>  <gson.version>2.8.0</gson.version>  <servlet.version>3.1.0</servlet.version>  <jsp.version>2.1</jsp.version>  <mysql.version>5.1.38</mysql.version>  <c3p0.version>0.9.1.2</c3p0.version>  <log4j.version>1.2.17</log4j.version>  <jstl.version>1.2</jstl.version>  <jackson.version>2.8.9</jackson.version>  <fileupload.version>1.3.1</fileupload.version>  <common.io.version>2.4</common.io.version>  <project.encoding>UTF-8</project.encoding>  <fastjson.version>1.2.46</fastjson.version>  <junit.version>4.12</junit.version>  <mybatis.spring.version>1.3.1</mybatis.spring.version>  <mybatis.version>3.4.5</mybatis.version>  <mybatis.pagehelper.version>5.1.2</mybatis.pagehelper.version>  <logback.version>1.1.7</logback.version>  <slf4j.version>1.7.21</slf4j.version>  <activiti-version>5.18.0</activiti-version>  </properties>  <!-- 配置所有依赖 -->  <dependencies>  <!-- https://mvnrepository.com/artifact/com.github.pagehelper/pagehelper -->  <dependency>  <groupId>com.github.pagehelper</groupId>  <artifactId>pagehelper</artifactId>  <version>${mybatis.pagehelper.version}</version>  </dependency>  <!-- 配置servlet相关 -->  <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>${servlet.version}</version>  <scope>provided</scope>  </dependency>  <!-- 配置JSP相关 -->  <!-- https://mvnrepository.com/artifact/javax.servlet.jsp/jsp-api -->  <dependency>  <groupId>javax.servlet.jsp</groupId>  <artifactId>jsp-api</artifactId>  <version>${jsp.version}</version>  <scope>provided</scope>  </dependency>  <!-- 配置JSTL相关 -->  <!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jstl</artifactId>  <version>${jstl.version}</version>  </dependency>  <!-- 配置数据库驱动相关 -->  <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>${mysql.version}</version>  </dependency>  <!-- 配置spring相关 -->  <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- 配置事务相关 -->  <!-- https://mvnrepository.com/artifact/org.springframework/spring-tx -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-tx</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- 配置spring-jdbc -->  <!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- 诶只spring-test -->  <!-- https://mvnrepository.com/artifact/org.springframework/spring-test -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-test</artifactId>  <version>${spring.version}</version>  <scope>test</scope>  </dependency>  <!-- 配置AspectJ 相关 -->  <!-- https://mvnrepository.com/artifact/org.springframework/spring-aspects -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- https://mvnrepository.com/artifact/org.springframework/spring-context-support -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- 配置SpringMvc相关 -->  <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- 配置Log4J相关 -->  <!-- https://mvnrepository.com/artifact/log4j/log4j -->  <!-- <dependency> <groupId>log4j</groupId> <artifactId>log4j</artifactId>  <version>${log4j.version}</version> </dependency> -->  <!-- c3p0连接池 -->  <!-- https://mvnrepository.com/artifact/com.mchange/c3p0 -->  <!-- https://mvnrepository.com/artifact/c3p0/c3p0 -->  <dependency>  <groupId>c3p0</groupId>  <artifactId>c3p0</artifactId>  <version>${c3p0.version}</version>  </dependency>  <!-- 配置Gson -->  <!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->  <dependency>  <groupId>com.google.code.gson</groupId>  <artifactId>gson</artifactId>  <version>${gson.version}</version>  </dependency>  <!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->  <!-- SpringMvc内置的jackson -->  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>${jackson.version}</version>  </dependency>  <!-- 配置文件上传相关 -->  <dependency>  <groupId>commons-fileupload</groupId>  <artifactId>commons-fileupload</artifactId>  <version>${fileupload.version}</version>  </dependency>  <!-- https://mvnrepository.com/artifact/commons-io/commons-io -->  <dependency>  <groupId>commons-io</groupId>  <artifactId>commons-io</artifactId>  <version>${common.io.version}</version>  </dependency>  <!-- 配置mybatis相关 -->  <!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>${mybatis.version}</version>  </dependency>  <!-- 配置Mybatis整合SPring相关 -->  <!-- https://mvnrepository.com/artifact/org.mybatis/mybatis-spring -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis-spring</artifactId>  <version>${mybatis.spring.version}</version>  </dependency>  <!-- 配置Spring整合orm相关 -->  <!-- https://mvnrepository.com/artifact/org.springframework/spring-orm -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-orm</artifactId>  <version>${spring.version}</version>  </dependency>  <!-- 配置fastJson相关 -->  <!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>${fastjson.version}</version>  </dependency>  <!-- 配置junit单元测试相关 -->  <!-- https://mvnrepository.com/artifact/junit/junit -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>${junit.version}</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>${log4j.version}</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>${slf4j.version}</version>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-classic</artifactId>  <version>1.1.2</version>  </dependency>  <dependency>  <groupId>ch.qos.logback</groupId>  <artifactId>logback-core</artifactId>  <version>1.1.2</version>  </dependency>  <dependency>  <groupId>org.logback-extensions</groupId>  <artifactId>logback-ext-spring</artifactId>  <version>0.1.1</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>jcl-over-slf4j</artifactId>  <version>1.7.12</version>  </dependency>  <!-- 配置POI相关 -->  <dependency>  <groupId>org.apache.poi</groupId>  <artifactId>poi-ooxml</artifactId>  <version>3.5-FINAL</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-engine</artifactId>  <version>${activiti-version}</version>  </dependency>  <dependency>  <groupId>org.activiti</groupId>  <artifactId>activiti-spring</artifactId>  <version>${activiti-version}</version>  </dependency>  </dependencies> |

### ****配置applicationContext-activiti.xml配置文件****

|  |
| --- |
| <?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:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  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*  *http://www.springframework.org/schema/mvc*  *http://www.springframework.org/schema/mvc/spring-mvc.xsd"*>  <!-- 工作流引擎配置bean -->  <bean id=*"processEngineConfiguration"* class=*"org.activiti.spring.SpringProcessEngineConfiguration"*>  <!-- 数据源 -->  <property name=*"dataSource"* ref=*"dataSource"* />  <!-- 使用spring事务管理器 -->  <property name=*"transactionManager"* ref=*"transactionManager"* />  <!-- 数据库策略 -->  <property name=*"databaseSchemaUpdate"* value=*"true"* />  <!-- activiti的定时任务关闭 -->  <property name=*"jobExecutorActivate"* value=*"false"* />  </bean>  <!-- 流程引擎 -->  <bean id=*"processEngine"* class=*"org.activiti.spring.ProcessEngineFactoryBean"*>  <property name=*"processEngineConfiguration"* ref=*"processEngineConfiguration"* />  </bean>    <!-- 资源服务service -->  <bean id=*"repositoryService"* factory-bean=*"processEngine"*  factory-method=*"getRepositoryService"* />  <!-- 流程运行service -->  <bean id=*"runtimeService"* factory-bean=*"processEngine"*  factory-method=*"getRuntimeService"* />  <!-- 任务管理service -->  <bean id=*"taskService"* factory-bean=*"processEngine"*  factory-method=*"getTaskService"* />  <!-- 历史管理service -->  <bean id=*"historyService"* factory-bean=*"processEngine"*  factory-method=*"getHistoryService"* />  <!-- 用户管理service -->  <bean id=*"identityService"* factory-bean=*"processEngine"*  factory-method=*"getIdentityService"* />  <!-- 引擎管理service -->  <bean id=*"managementService"* factory-bean=*"processEngine"*  factory-method=*"getManagementService"* />  </beans> |

* **注意：**
* **StandaloneProcessEngineConfiguration activiti单独使用**
* **SpringProcessEngineConfiguration activiti和Spring进行整合的时候使用**

### 配置web.xml

* 参考以前代码

12.CFX框架整合配置

1. CXF的介绍、安装和配置

### 介绍

CXF是一个开源的webservice框架，提供很多完善功能，可以实现快速开发

CXF支持的协议：SOAP1.1/1.2，REST

CXF支持数据格式：XML，JSON（仅在REST方式下支持）

### 安装和配置

##### 1.2.1 下载地址 <http://cxf.apache.org/download.html>

##### 1.2.2 安装和配置

注意： 安装配置之前需要先安装jdk。

1.将下载的文件进行解压,配置环境变量，新建CXF\_HOME,输入下载文件库的路径，

示例：C:\software\apache-cxf-3.2.1\apache-cxf-3.2.1

2.在系统环境变量的Path追加：第三步C:\software\apache-cxf-3.2.1\apache-cxf-3.2.1\bin目录

3.测试，在cmd下加入wsdl2java –h ·

2 . CXF发布SOAP协议的服务

### 2.1 需求

服务端：发布服务，接收客户端的城市名，返回天气数据给客户端

客户端：发送城市名给服务端，接收服务端的响应信息，打印

### 2.2 实现

**Cxf的pom文件：**

|  |
| --- |
| **<!-- 配置cxf相关 -->**  **<!-- https://mvnrepository.com/artifact/org.apache.cxf/cxf-rt-frontend-jaxws -->**  **<dependency>**  **<groupId>org.apache.cxf</groupId>**  **<artifactId>cxf-core</artifactId>**  **<version>3.2.4</version>**  **</dependency>**  **<dependency>**  **<groupId>org.apache.cxf</groupId>**  **<artifactId>cxf-rt-frontend-jaxws</artifactId>**  **<version>3.2.4</version>**  **</dependency>**  **<dependency>**  **<groupId>org.apache.cxf</groupId>**  **<artifactId>cxf-rt-transports-http-jetty</artifactId>**  **<version>3.2.4</version>**  **</dependency>**  **<!-- https://mvnrepository.com/artifact/org.apache.cxf/cxf-rt-rs-client -->**  **<dependency>**  **<groupId>org.apache.cxf</groupId>**  **<artifactId>cxf-rt-rs-client</artifactId>**  **<version>3.2.4</version>**  **</dependency>**  **<!-- https://mvnrepository.com/artifact/org.apache.cxf/cxf-rt-rs-extension-providers -->**  **<dependency>**  **<groupId>org.apache.cxf</groupId>**  **<artifactId>cxf-rt-rs-extension-providers</artifactId>**  **<version>3.2.4</version>**  **</dependency>**  **<!-- https://mvnrepository.com/artifact/org.apache.cxf/cxf-rt-rs-service-description -->**  **<dependency>**  **<groupId>org.apache.cxf</groupId>**  **<artifactId>cxf-rt-rs-service-description</artifactId>**  **<version>3.2.4</version>**  **</dependency>**  **<!-- for extension providers -->**  **<dependency>**  **<groupId>org.codehaus.jettison</groupId>**  **<artifactId>jettison</artifactId>**  **<version>1.2</version>**  **</dependency>** |

#### 2.2.1 服务端

**JaxWsServerFactoryBean**

1. **设置服务地址**
2. **设置服务服务接口setServiceClass()**
3. **设置服务实例 setServiceBean()**
4. **发布 create()**
5. **测试**

|  |
| --- |
| 1.导入jar包/引入pom文件还可以引入cxf的运行环境 //具体步骤如下：  // 右键项目-properties-Java Build Path--Add External JARs即可  **2.创建SEI接口,要加入@WebService接口**  备注：@BindingType(SOAPBinding.SOAP12HTTP\_BINDING)表示发布SOAP1.2的服务端    @WebService  @BindingType(SOAPBinding.SOAP12HTTP\_BINDING)  public interface WeatherInterface {  public String queryWeather(String cityName);  }  **3.创建SEI接口实现类。**  public class WeatherInterfaceImpl implements WeatherInterface {  @Override  public String queryWeather(String cityName) {  // TODO Auto-generated method stub  if ("北京".equals(cityName)) {  return "冷且霾";  } else {  return "暖且晴";  }  }  }  **4.发布服务(使用JaxWsServerFactoryBean发布)**  备注：endpoint仅支持发布实现类，JaxWsServerFactoryBean支持发布接口。  public class WeatherServer {  //用JaxWsServerFactoryBean发布服务，设置3个参数，1.服务接口；2.服务实现类；3.服务地址；  public static void main(String[] args) {  JaxWsServerFactoryBean factoryBean =new JaxWsServerFactoryBean();  //设置服务接口  factoryBean.setServiceClass(WeatherInterface.class);  //设置服务实现类  factoryBean.setServiceBean(new WeatherInterfaceImpl());  //设置服务地址  factoryBean.setAddress("http://127.0.0.1:8888/weather");  //发布  factoryBean.create();  }  }  **5.测试服务是否发布成功，阅读使用说明书.直接访问即可。** |

#### 2.2.2 客户端

**JaxWsProxyFactoryBean**

1. 设置地址 **setAddress**
2. 设置服务接口**setServiceClass**
3. **获取服务接口实例 create**
4. **调用查询方法**

|  |
| --- |
| **1.生成客户端代码**  1.1 wsdl2java命令是CXF提供的生成客户端的工具，他和wsimport类似，可以根据WSDL生成客户端代码  1.2 wsdl2java常用参数：  -d，指定输出目录  -p，指定包名，如果不指定该参数，默认包名是WSDL的命名空间的倒序  1.3 wsdl2java支持SOAP1.1和SOAP1.2    完整示例：wsdl2java -p com.sky.cxf.weather -d . http://127.0.0.1:8888/weather?wsdl  //根据指定说明书地址生成客户端代码 －p指定生成的包名 －d .表示输出到当前目录  **2.使用说明书，使用生成代码调用服务端**  //使用JaxWsProxyFactoryBean调用服务端，设置2个参数，1.设置服务接口；  public class WeatherClient {  public static void main(String[] args) {  JaxWsProxyFactoryBean factoryBean =new JaxWsProxyFactoryBean();  //设置服务接口  factoryBean.setServiceClass(WeatherInterface.class);  //设置服务地址  factoryBean.setAddress("http://127.0.0.1:8888/weather");  //获取服务接口实例  WeatherInterface weatherInterface =factoryBean.create(WeatherInterface.class);  //调用查询方法  String string = weatherInterface.queryWeather("宝鸡");  System.out.println(string);  }  } |

3. CXF+Spring整合发布SOAP的服务

### 3.1 服务端

#### 3.1.1 第一步：创建web工程，引入cxf的jar包

#### 3.1.2 第二步：创建SEI接口

#### 3.1.3 第三步：创建SEI接口实现类

#### 3.1.4 第四步：配置Spring配置文件 applicationContext-cxf.xml

|  |
| --- |
| <?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:p="http://www.springframework.org/schema/p"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:mvc="http://www.springframework.org/schema/mvc"  xmlns:cxf="http://cxf.apache.org/cxf"  xmlns:jaxrs="http://cxf.apache.org/jaxrs"  xmlns:jaxws="http://cxf.apache.org/jaxws"  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  http://www.springframework.org/schema/mvc  http://www.springframework.org/schema/mvc/spring-mvc.xsd  http://cxf.apache.org/cxf  http://cxf.apache.org/schemas/core.xsd  http://cxf.apache.org/jaxrs  http://cxf.apache.org/schemas/jaxrs.xsd  http://cxf.apache.org/jaxws  http://cxf.apache.org/schemas/jaxws.xsd">    <!-- 服务端 spring配置webservice -->  <!-- 对EndPoint封装 -->  <jaxws:endpoint address="/hello" implementor="com.ws.weather.Hello"></jaxws:endpoint>  <!-- 对JaxServerFactoryBean封装 -->  <jaxws:server address="/queryWeather" serviceName="com.ws.weather.WeatherInterface">  <!-- 配置接口的实现类 -->  <jaxws:serviceBean>  <ref bean="weatherServiceImpl"/>  </jaxws:serviceBean>  </jaxws:server>      <!-- 配置服务实现类 -->  <bean id="weatherServiceImpl" class="com.ws.weather.WeatherInterfaceImpl"></bean>  </beans> |

#### 3.1.5 第五步：配置web.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" version="2.5">  <display-name>s-cxf</display-name>  <welcome-file-list>  <welcome-file>index.html</welcome-file>  <welcome-file>index.htm</welcome-file>  <welcome-file>index.jsp</welcome-file>  <welcome-file>default.html</welcome-file>  <welcome-file>default.htm</welcome-file>  <welcome-file>default.jsp</welcome-file>  </welcome-file-list>    <!-- 配置Spring环境 -->  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:applicationContext-cxf.xml</param-value>  </context-param>  <!-- Bootstraps the root web application context before servlet initialization -->  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener>  **<!-- 配置CXF的Servlet -->**  <servlet>  <servlet-name>CXF</servlet-name>  <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>CXF</servlet-name>  <url-pattern>/ws/\*</url-pattern>  </servlet-mapping>  </web-app> |

#### 3.1.6 第六步：启动tomcat,部署web工程到tomcat

#### 3.1.7 测试服务是否发布成功

* + **WSDL地址规则：**[***http://localhost:***](http://localhost:)**端口号/项目名称/servlet拦截路径/服务名称?wsdl**

示例：<http://localhost/CXFSpring/ws/weather?wsdl>

​ <http://localhost/CXFSpring/ws/hello?wsdl>

### 3.2 客户端

#### 3.2.1 第一步：创建web工程作为客户端，引入cxf的jar包

#### 3.2.2 第二步：生成客户端代码

执行命令：wsdl2java -p 包名 －d . 服务说明书地址

示例：

wsdl2java -p com.ws.cxf.client －d . <http://localhost/CXFSpring/ws/weather?wsdl>

#### 3.2.3 第三步：配置spring的配置文件

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns="http://www.springframework.org/schema/beans" xmlns:cxf="http://cxf.apache.org/cxf"  xmlns:jaxrs="http://cxf.apache.org/jaxrs" xmlns:jaxws="http://cxf.apache.org/jaxws"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.3.xsd http://cxf.apache.org/cxf  http://cxf.apache.org/schemas/core.xsd http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd  http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd ">  **<!--jaxws:client实现客户端配置,对JaxWsProxyFactoryBean类封装-->**  **<jaxws:client id="weatherClient" address="http://localhost/CXFSpring/ws/weather"**  **serviceClass="com.ws.cxf.client.WeatherInterface"></jaxws:client>**  </beans> |

#### 3.2.4 第四步：初始化spring上下文，获取接口实现类，调用查询方法

|  |
| --- |
| import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.ws.cxf.client.WeatherInterface;  public class WeatherClient {  public static void main(String[] args) {  ApplicationContext context = new ClassPathXmlApplicationContext("classpath:applicationContext.xml");  WeatherInterface weatherInterface = (WeatherInterface) context.getBean("weatherClient");  System.out.println(weatherInterface.queryWeather("北京"));  }  } |

4.CXF发布REST服务

### 4.1 什么是REST?

**定义：REST就是一种编程风格，它可以精确定位网上资源（服务接口、方法、参数）**

**REST支持数据格式：XML、JSON**

**REST支持发送方式：GET，POST**

### 4.2 需求

* 第一个：查询单个学生
* 第二个：查询多个学生

### 4.3 实现

#### 4.3.1 服务端

开发步骤：

###### 1.导入jar

###### 2.创建学生类，需要加入@ XmlRootElement

@XmlRootElement(name="student")//@XmlRootElement可以实现对象和XML数据之间的转换

public class Student {

private long id;

private String name;

private Date birthday;

.......

set/get.....

}

###### 3.创建SEI接口

package com.sky.ws.rest.server;

import java.util.List;

import javax.jws.WebService;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.PathParam;

import javax.ws.rs.Produces;

import javax.ws.rs.core.MediaType;

import com.sky.pojo.Student;

@WebService

@Path("/student")//@Path("/student")就是将请求路径中的“/student”映射到接口上

public interface StudentInterface {

  /\*\*

   \* 根据id查询单个学生

   \* @param id

   \* @return

   \*/

   @GET  //指定请求方式，如果服务端发布的时候指定的是GET（POST），那么客户端访问时必须使用GET（POST）

   @Produces(MediaType.APPLICATION\_XML)//指定服务数据类型

   @Path("/query/{id}")//@Path("/query/{id}")就是将"/query"映射到方法上，"{id}"映射到参数上，多个参数，以“/”隔开，放到“{}”中

public Student queryStuById(@PathParam("id")long id);

/\*\*

\* 根据查询多个学生

\* @param name

\* @return

\*/

   @GET

   @Produces({MediaType.APPLICATION\_XML,"application/json;charset=utf-8"})

   @Path("/querylist/{name}")

public List<Student> queryStudentList(@PathParam("name") String name);

}

###### 4.创建SEI接口实现类

package com.sky.ws.rest.server;

import java.util.ArrayList;

import java.util.Date;

import java.util.List;

import javax.jws.WebService;

import javax.ws.rs.Path;

import com.sky.pojo.Student;

public class StudentInterfaceImpl implements StudentInterface {

@Override

public Student queryStuById(long id) {

Student st = new Student();

st.setId(id);

st.setName("张三");

st.setBirthday(new Date());

return st;

}

@Override

public List<Student> queryStudentList(String name) {

Student st = new Student();

st.setId(110);

st.setName("张三");

st.setBirthday(new Date());

Student st2 = new Student();

st2.setId(120);

st2.setName("李四");

st2.setBirthday(new Date());

List<Student> list = new ArrayList<Student>();

list.add(st);

list.add(st2);

return list;

}

}

###### 5.发布服务

package com.sky.ws.rest.server;

​

import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;

public class StudentServer {

​

public static void main(String[] args) {

//JAXRSServerFactoryBean发布REST的服务

JAXRSServerFactoryBean factoryBean =new JAXRSServerFactoryBean();

//设置服务实现类

factoryBean.setServiceBean(new StudentInterfaceImpl());

//设置资源类,如果有多个资源类，可以以“,”隔开。

factoryBean.setResourceClasses(StudentInterfaceImpl.class);

//设置服务地址

factoryBean.setAddress("http://localhost:8801/user");

//发布服务

factoryBean.create();

}

}

###### 6.测试服务

6.1 测试查询单个学生：http://localhost:8801/user/student/query/1002

结果如下：

   <student>

         <birthday>2017-11-19T21:39:18.449+08:00</birthday>

         <id>1002</id>

         <name>张三</name>

   </student>

6.2 测试查询多个学生：http://localhost:8801/user/student/querylist/1002

GET请求默认是xml类型：

结果如下：

   <students>

         <student>

               <birthday>2017-11-19T21:40:16.609+08:00</birthday>

               <id>110</id>

               <name>张三</name>

         </student>

         <student>

               <birthday>2017-11-19T21:40:16.609+08:00</birthday>

               <id>120</id>

               <name>李四</name>

         </student>

   </students>

6.3 多个类型选择：http://localhost:8801/user/student/querylist/1002?\_type=json

  结果如下：

{"student":[

                    {"birthday":"2017-11-19T21:41:28.204+08:00","id":110,"name":"张三"},   {"birthday":"2017-11-19T21:41:28.204+08:00","id":120,"name":"李四"}

                  ]

        }

6.4 http://localhost:8801/user/student/querylist/1002?\_type=xml

备注：

    如果服务端发布时指定请求方式是GET（POST），客户端必须使用GET（POST）访问服务端，否则会报异常

    如果在同一方法上同时指定XML和JSON媒体类型，默认返回XML

#### 4.3.2 服务端

5 CXF+Spring整合发布REST服务

#### 5.1 服务端

开发步骤：

##### 第一步：创建web项目（引入jar包）

##### 第二步：创建POJO类(同上)

##### 第三步：创建SEI接口(同上)

##### 第四步：创建SEI实现类(同上)

##### 第五步：配置Spring配置文件,applicationContext.xml

使用 jaxrs:server发布服务：设置1.服务地址；2.服务实现类

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

<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

xmlns:cxf="http://cxf.apache.org/cxf"

xmlns:jaxrs="http://cxf.apache.org/jaxrs"

xmlns:jaxws="http://cxf.apache.org/jaxws"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.3.xsd

http://cxf.apache.org/cxf

http://cxf.apache.org/schemas/core.xsd

http://cxf.apache.org/jaxrs

http://cxf.apache.org/schemas/jaxrs.xsd

http://cxf.apache.org/jaxws

http://cxf.apache.org/schemas/jaxws.xsd ">

     <!-- 发布REST的服务,对JAXRSServerFactoryBean类封装 -->

   <jaxrs:server address="/user">

<jaxrs:serviceBeans>

<ref bean="studentServiceImpl"/>

</jaxrs:serviceBeans>

</jaxrs:server>

<!-- 配置服务类 -->

<bean name="studentServiceImpl" class="com.qf.service.impl.StudentServiceImpl"></bean>

</beans>

##### 第六步：配置web.xml

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

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://xmlns.jcp.org/xml/ns/javaee"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"

version="3.1">

<display-name>CXF\_WS\_REST\_Spring</display-name>

<!-- 配置Spring环境，web服务器一启动就加载Spring的配置文件 applicationContext.xml-->

<context-param>

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

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

</context-param>

<!-- Bootstraps the root web application context before servlet initialization -->

<listener>

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

</listener>

<!--配置CXF的Servlet -->

  <servlet>

     <servlet-name>CXFServlet</servlet-name>

     <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>

  </servlet>

  <servlet-mapping>

    <servlet-name>CXFServlet</servlet-name>

    <url-pattern>/ws/\*</url-pattern>

  </servlet-mapping>

</web-app>

##### 第七步：部署到tomcat下，启动tomcat

##### 第八步：测试服务

###### REST服务的使用说明书地址：

http://localhost/CXF\_WS\_REST\_Spring/ws/user?\_wadl

​

**页面ajax去访问rest风格的webservice**

13.分布式项目的整合