第一章 Spring 框架（一）

课程目标

1、 Spring 概述

2、 开发第一个 Spring 程序-环境搭建

3、 Spring 思想-IOC 与 DI

4、 Spring 配置详解

5、 Spring 属性注入

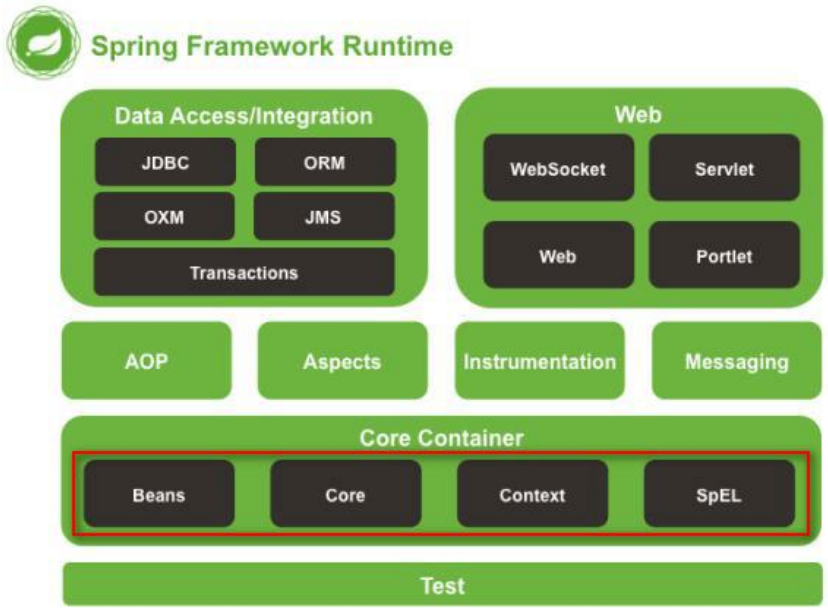
6、 Spring 复杂类型注入

7、 Oracle-jdbc 访问数据

课程内容

1、 Spring 概述

Spring 框架核心模块：



1.1）Spring 是什么？

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

EE 开发分成三层结构:

* WEB 层:Spring MVC.
* 业务层:Bean 管理:(IOC)
* 持久层:Spring 的 JDBC 模板.ORM 模板用于整合其他的持久层框架.

1.2）为什么学习 Spring？

① 方便解耦，简化开发

Spring 就是一个大工厂，可以将所有对象创建和依赖关系维护，交给 Spring 管理

② AOP 编程的支持

Spring 提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能

③ 声明式事务的支持

只需要通过配置就可以完成对事务的管理，而无需手动编程

④ 方便程序的测试

Spring 对 Junit4 支持，可以通过注解方便的测试 Spring 程序

⑤ 方便集成各种优秀框架

Spring 不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz 等）的直接支持

⑥ 降低 JavaEE API 的使用难度

Spring 对 JavaEE 开发中非常难用的一些 API（JDBC、JavaMail、远程调用等），都提供了封装，使这些 API 应用难度大大降低

1.3）Spring 版本？

目前有 Spring3.x,Spring4.x,Spring5 三种版本。

2、 开发第一个 Spring 程序-环境搭建

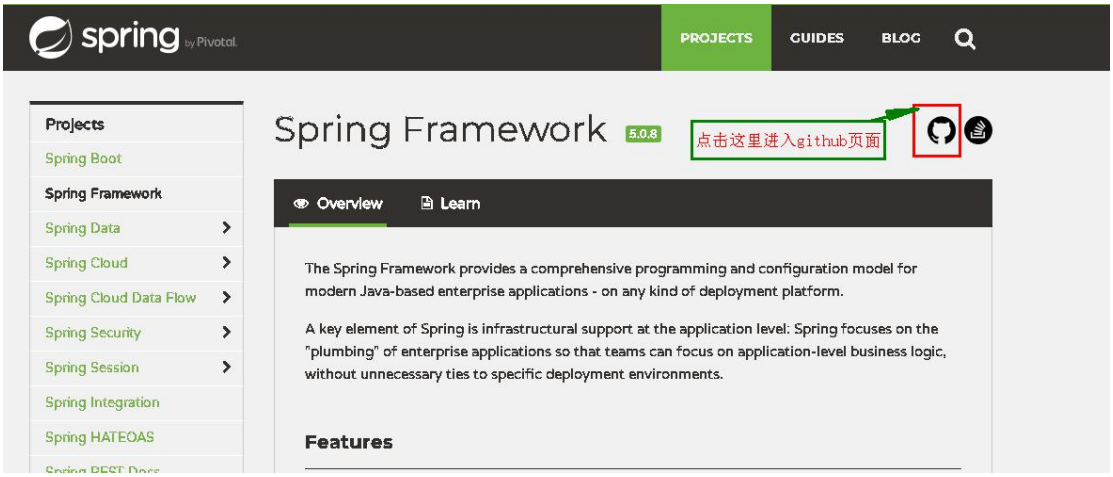
2.0）补充知识：

2.0.1）如何下载 Spring？

第一步，进入 spring 官网，

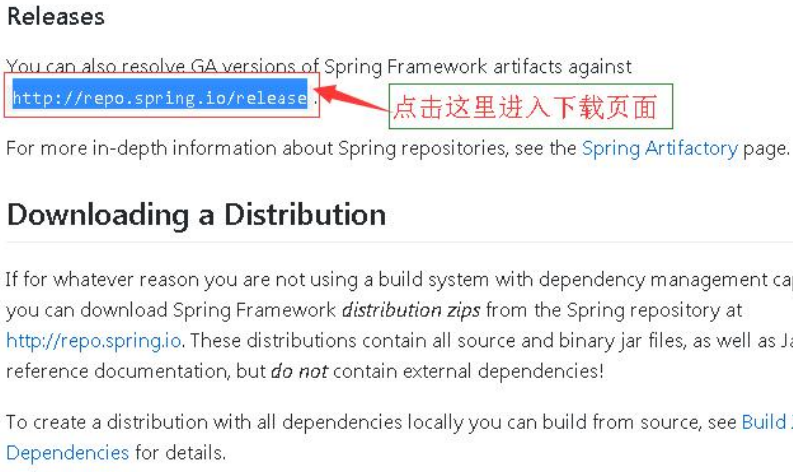
http://spring.io

第二步：选择 **project** 菜单，点击后，找到 **spring framework** 项，点击进入：



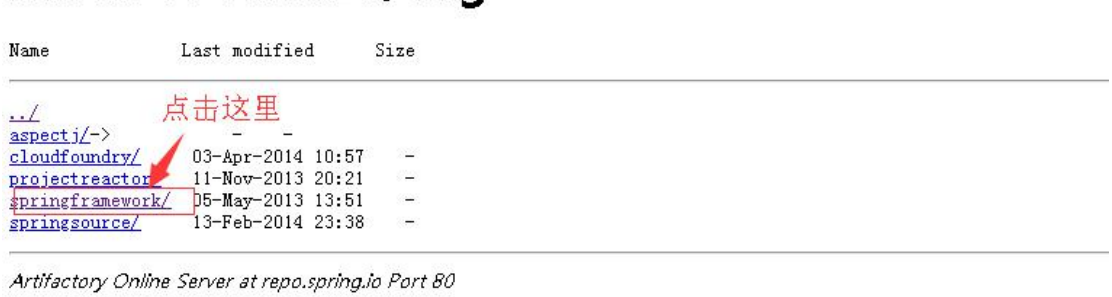
第三步：进行 **github** 的二进制下载页面：

第四步：进入上面页面，再复制下面的链接到新页面进行下载：

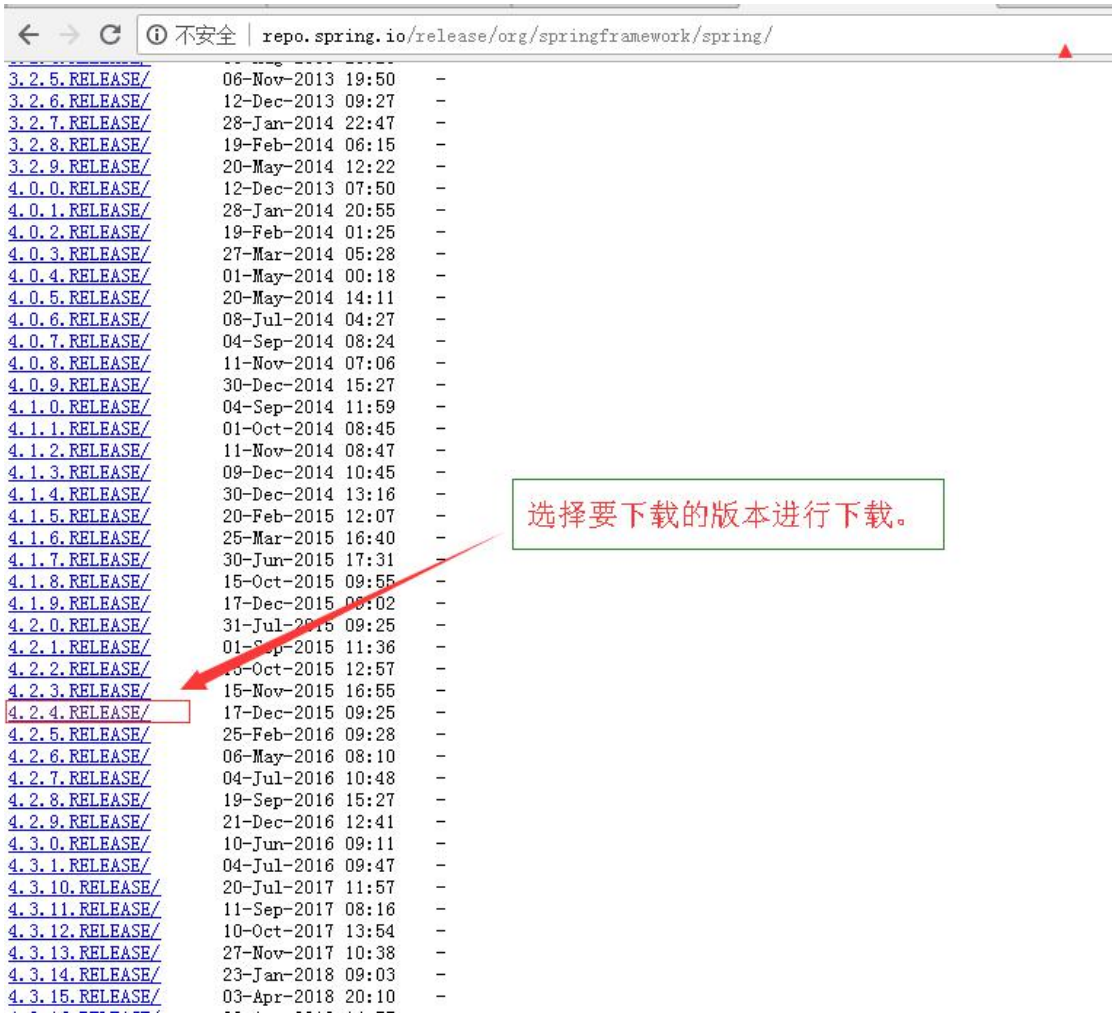


第五步：进入下载页面：

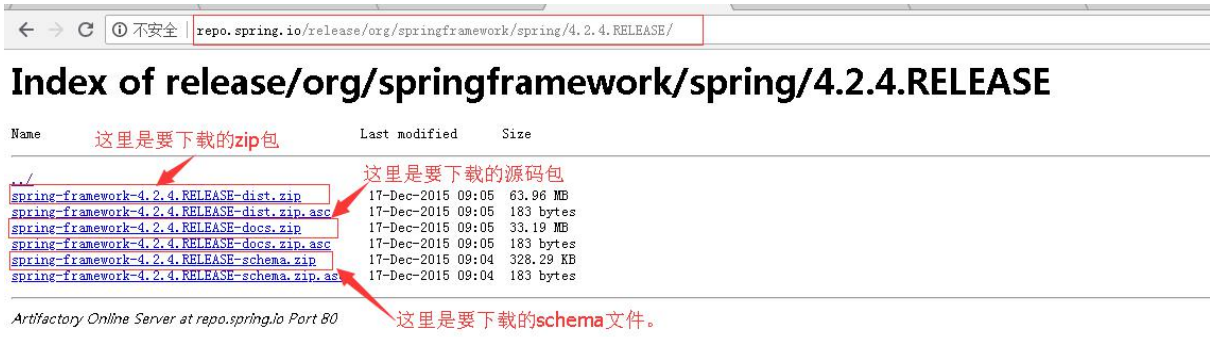
第六步：选择 **springframework:**



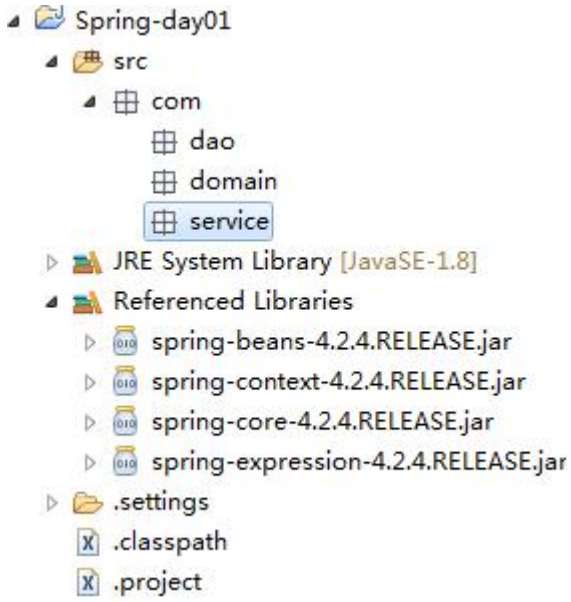
第七步：选择 spring，再选择要下载的版本：



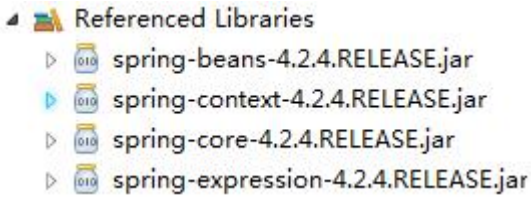
第八步：根据需要选择各自的内容：



2.1）在 Eclipse 中创建工程（可以是任意工程）：



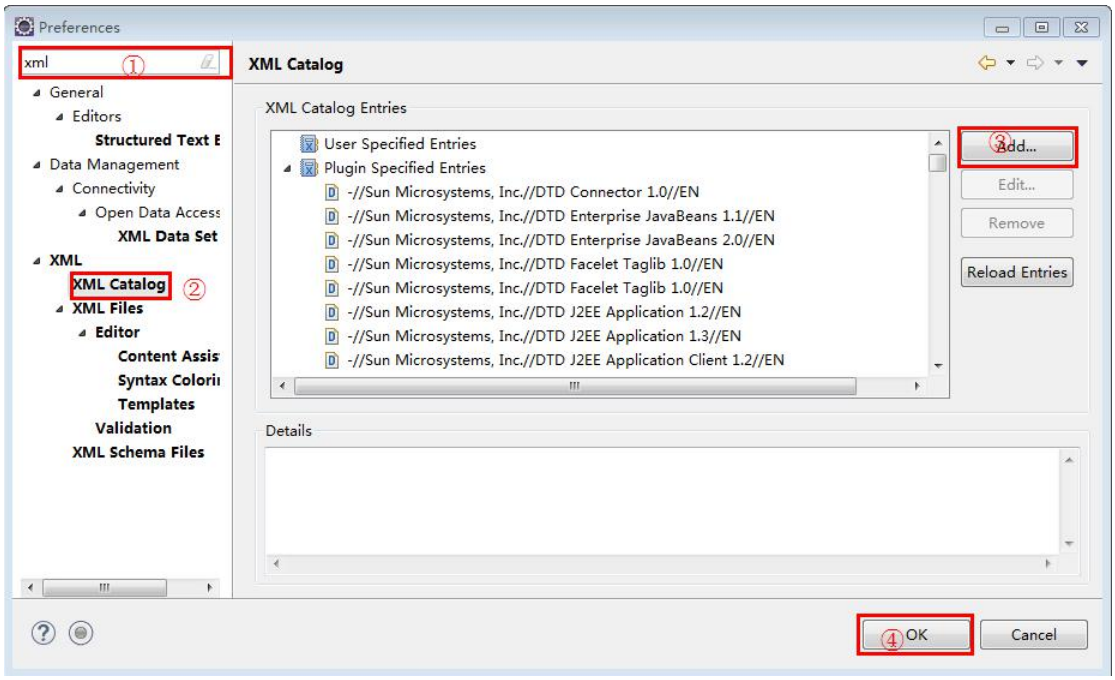
2.2）导入 spring 开发的基础 jar 包（4 个）：

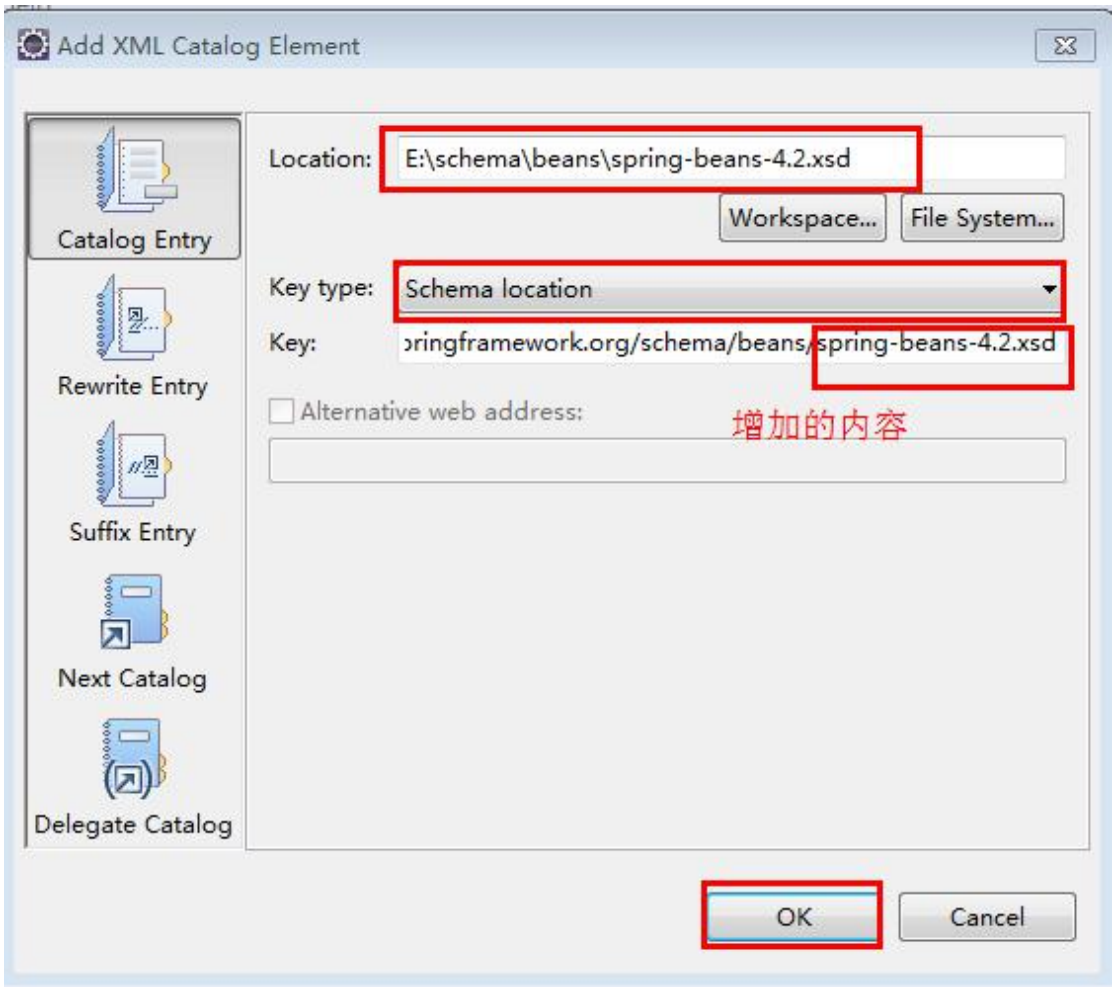


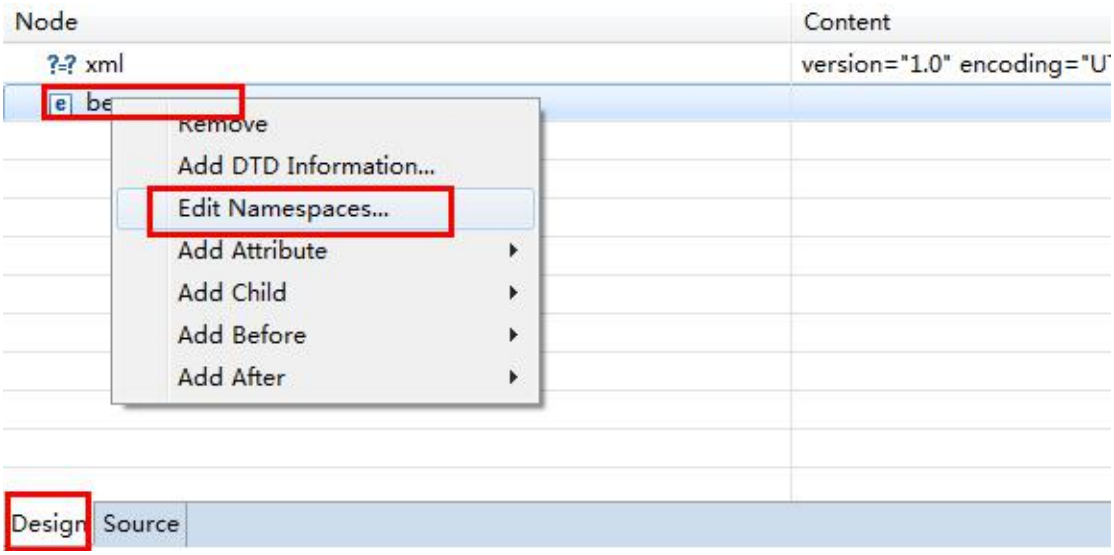
2.3）定义 spring 的配置文件：

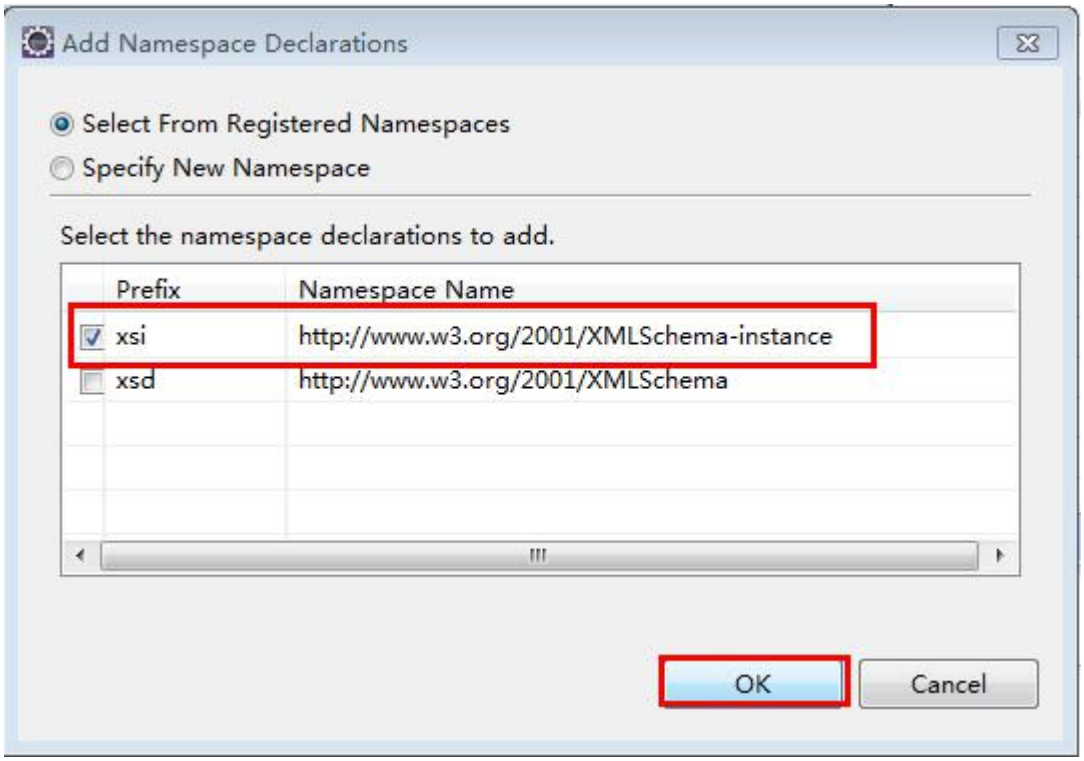
Spring 的配置文件一般默认为：applicationContext.xml 文件，当然，其实，我们可以随意定义。

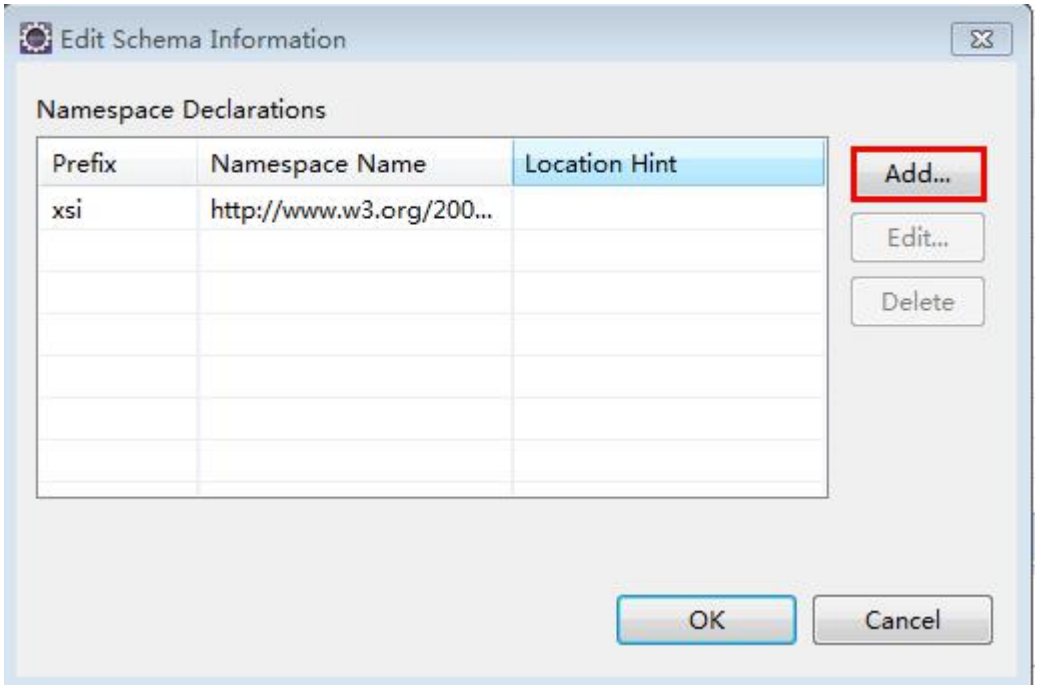
2.3.1）配置 xsd 文件，方便 xml 出现提示：

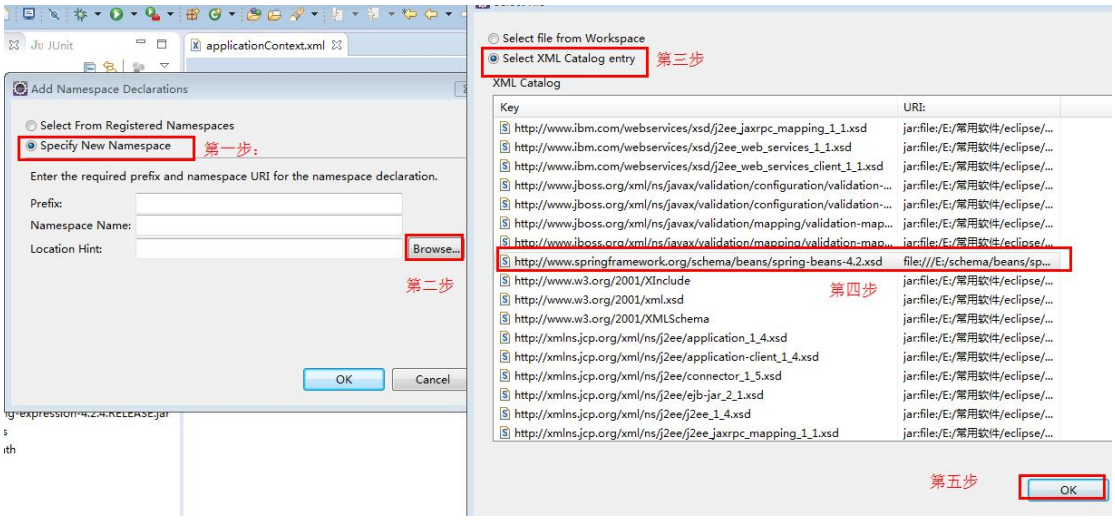


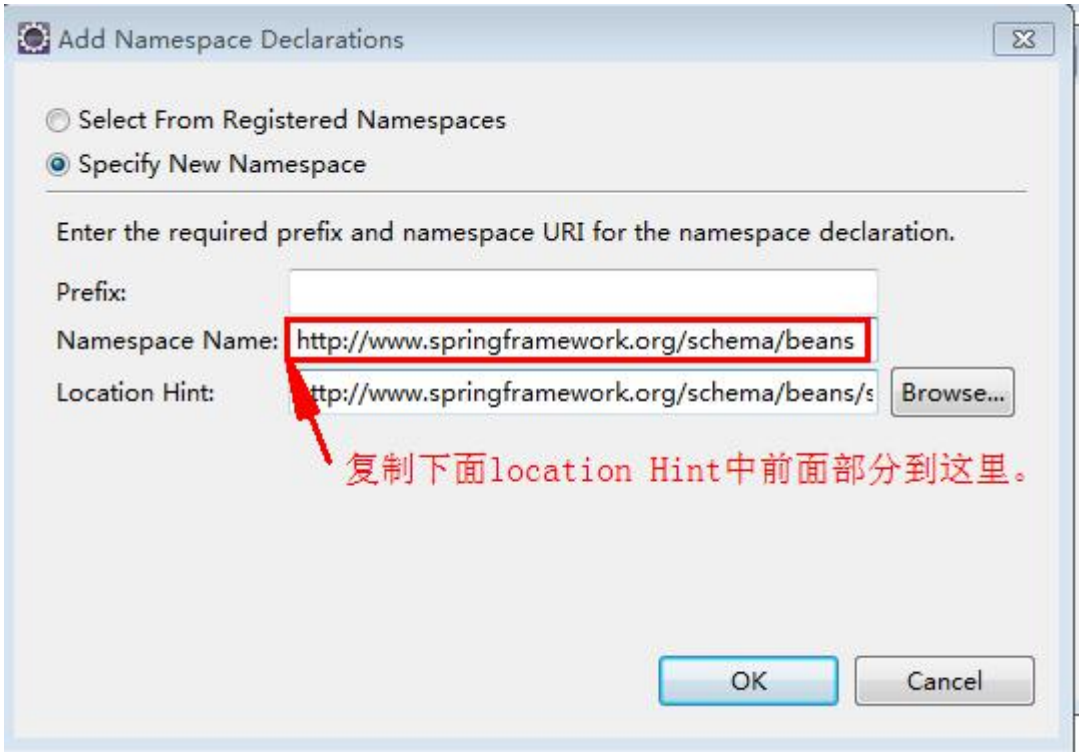



Node
Content
xml
version="1.0" encoding="U
be
Remove
Add DTD Information...
Edit Namespaces...
Add Attribute
Add Child
Add Before
Add After
Design
Source


Add Namespace Declarations
Select From Registered Namespaces
Specify New Namespace
Select the namespace declarations to add.
Prefix
Namespace Name
xsi
http://www.w3.org/2001/XMLSchema-instance
xsd
http://www.w3.org/2001/XMLSchema
OK
Cancel

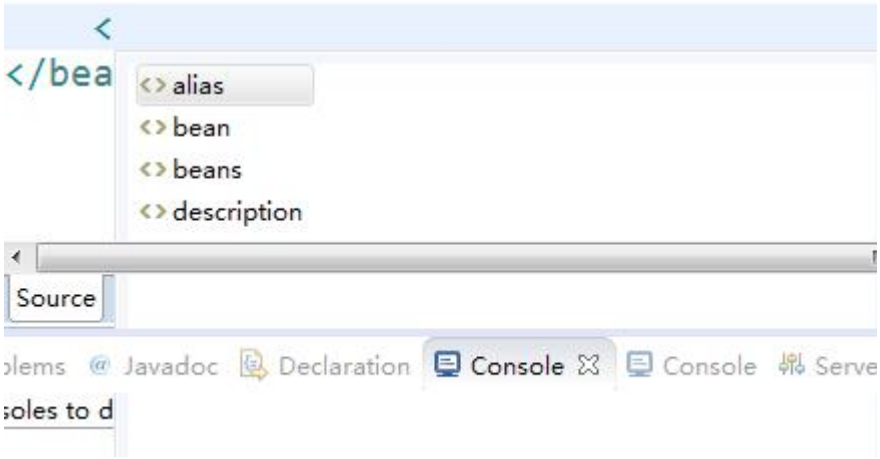

Edit Schema Information
Namespace Declarations
Prefix
Namespace Name
Location Hint
xsi
http://www.w3.org/200...
Add...
Edit...
Delete
OK
Cancel


Add Namespace Declarations
Select From Registered Namespaces
Specify New Namespace
第一步：
Enter the required prefix and namespace URI for the namespace declaration.
Prefix:
Namespace Name:
Location Hint:
Browse...
第二步
OK
Cancel
Select file from Workspace
Select XML Catalog entry
第三步
XML Catalog
Key
URI:
http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
file:///E:/schema/beans/sp...
第四步
第五步
OK



最终页面在没有网络的情况下，也可以显示配置文件中的提示内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001
    xmlns="http://www.springframework.or
    xsi:schemaLocation="http://www.sprir
                        http://www.sprir
```



2.3.2）定义 javabean，通过 junit 进行测试

```
public class Emp {

    private int empno;
    private String ename;
    private String job;
    private int mgr;
    private String hiredate;
    private int sal;
    private int comm;
    private int deptno;
```

如果使用 junit 进行测试，需要添加关于日志的两个包:

测试代码如下：

```
@Test
public void test01(){
    //1.得到一个ApplicationContext对象
    ApplicationContext ac =
            new ClassPathXmlApplicationContext("applicationContext.xml");
    System.out.println(ac);
    //2.通过上面的对象取得javaBean对象
    Emp emp = (Emp) ac.getBean("emp01");
    System.out.println(emp);
}
```

BeanFactory 与 ApplicationContext 对象使用区别:

1、BeanFactory 加载对象时机：是在使用某个对象时才加载，节省内存。

2、ApplicationContext 对象是 Spring 容器一启动，就加载所有的配置文件中的<bean>标签作为 javabean 对象，现在使用较多，系统一般不缺资源 。

3、web 开发中,使用 applicationContext. 在资源匮乏的环境可以使用 BeanFactory.

3、Spring 思想-IOC 与 DI

3.1）IOC-控制反转（Inverse Of Control）

以前程序中的 javabean 对象是由我们程序员创建完成，现在由 spring 容器帮我们创建完成，并且以前 javabean 的依赖关系也是我们程序员自己定义，现在也由 spring 容器帮我们创建这种依赖关系，所以，这种控制权以前由我们完成现在改由 spring 容器完成，这样，控制权发生了反转。

3.2) DI-依赖注入（Dependency-Injection）

就是我们可以在 spring 的配置文件中定义 javabean 时，可以为其指定依赖关系，这样，我们在程序中的这种依赖就不需要我们程序员去干预。

4、Spring 配置详解

4.1）bean 标签-name,class,id,scope 详解：

```
<!--
        bean 标签的一些常用属性说明：
        ① name：代表为自定义的 bean 起一个名字，可以存放特殊字符。
        ② id：代表为自定义的 bean 指定一个 id，不能存放特殊字符。
        ③ class：代表自定义 bean 的完整类名，包括包名。
        ④ abstract:代表自定义 bean 是否是抽象类，true:是抽象类，false:不是抽象类
        ⑤ init-method:代表自定义 bean 的初始化方法.
        ⑥ destroy-method:代表自定义 bean 的销毁方法。
        ⑦ scope:代表自定义 bean 的作用范围。可取值如下：
          singleton:代表此对象为单例。【默认值】（常用）
          prototype:代表此对象为多例。（常用）
          request:代表此对象在一次请求有效。
          session:代表此对象在浏览器不关闭的情况下有效。
    -->
```

测试代码：

```
@Test
    public void test() {
        ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("myBeans.xml");
        Student student = (Student) ac.getBean("student");
        System.out.println(student);
        Student student2 = (Student) ac.getBean("student");
        System.out.println(student2);
        ac.close();
    }
```

测试效果：

```
log4j:WARN No appenders could be found for logger (org.springframework.core.e
log4j:WARN Please initialize the log4j system properly.
这里是student类的实始化方法.
com.zelin.pojo.Student@4671e53b
这里是student类的实始化方法.
com.zelin.pojo.Student@2db7a79b
```

这里使用scope="prototype"代表生成的对象为多例。

4.2）bean 标签生命周期两个方法：

```
    init-method:代表javabean生命周期开始时的方法。
    destroy-method:代表javabean生命周期结束时的方法.
-->
<bean name="emp01" class="com.domain.Emp" scope="singleton"
      init-method="init"
      destroy-method="destory"></bean>
```

测试代码：

```
@Test
public void test01(){
    //1.得到一个ApplicationContext对象
    ClassPathXmlApplicationContext ac =
            new ClassPathXmlApplicationContext("applicationContext.xml");
    System.out.println(ac);
    //2.通过上面的对象取得javaBean对象
    Emp emp01 = (Emp)ac.getBean("emp01");
    Emp emp02 = (Emp)ac.getBean("emp01");
    System.out.println(emp01);
    System.out.println(emp01 == emp02);
    //关闭ac对象时会触发javabean所关联的destory-method属性指定的方法：
    ac.close();
}
```

测试结果：

```
Emp生命周期开始-------> init()方法被调用！
org.springframework.context.support.ClassPathXmlApplicationCor
Emp [empno=0, ename=null, job=null, mgr=0, hiredate=null, sal=
true
Emp生命周期结束-------> destory()方法被调用！
```

4.3）Spring 创建对象的四种方式：

4.3.1）使用空构造方法创建：

```
<bean name="emp01" class="com.domain.Emp" scope="singleton"
      init-method="init"
      destroy-method="destory"></bean>
```

方式一：使用空构造方法创建

测试见前面的代码：

4.3.2）使用静态工厂方法创建：

```
public class BeanFactory {

    //静态工厂方法
    public static Emp getInstance(){
        System.out.println("使用静态工厂方法创建Emp对象！");
        return new Emp();
    }
}
```

```
<!-- 创建对象方式二：使用静态工厂方式创建对象 -->
<bean name="emp02" class="com.factory.BeanFactory"
      factory-method="getInstance"></bean>
```

测试代码：

```
@Test
//使用静态工厂的方式创建对象
public void test02(){
    //1.得到一个ApplicationContext对象
    ApplicationContext ac =
            new ClassPathXmlApplicationContext("applicationContext.xml");
    //2.通过上面的对象取得javaBean对象
    Emp emp02 = (Emp)ac.getBean("emp02");
    System.out.println(emp02);
}
```

测试结果：

```
Emp生命周期开始------->
使用静态工厂方法创建Emp对象！
Emp [empno=0, ename=null, job=null, mgr=0, hiredate=null, sal=0, comm=0, deptno=0]
```

4.3.3）使用对象工厂方法创建：

```
public class StudentMethodFactory {
    public Student getStudent(){
        return new Student();
    }
}
```

```
<!-- 方法四：使用工厂方法构建对象 -->
<bean name="studentFactory" class="com.zelin.factory.StudentMethodFactory"/>
<bean name="student5" factory-bean="studentFactory"
factory-method="getStudent"/>
```

测试代码：

```
//创建对象的方法四：使用工厂方法构造对象
        Student student5 = (Student) ac.getBean("student5");
        System.out.println("student5:" + student5);
```

测试结果：

```
log4j:WARN No appenders could be found for logger (org.springframework.c
log4j:WARN Please initialize the log4j system properly.
Student [sid=1001, sname=张三, sex=null, age=0, addr=null]
Student [sid=1005, sname=小明, sex=null, age=0, addr=null]
Student [sid=0, sname=null, sex=null, age=0, addr=null]
student5:Student [sid=0, sname=null, sex=null, age=0, addr=null]
```

使用工厂方法创建对象

总结：

从上面的例子可以看出，BeanFactory 是在需要对象时才会从配置文件中加载对象，而 ApplictionContext 在 Spring 容器一启动时就会将配置文件中所有的 bean 加载到容器中，所以效率更高，我们一般在 web 开发中，使用 AppcationContext 对象。

4、Spring 属性注入(非常重要)

4.1）set 方法注入：

4.1.1) Beans.xml 文件中：

```
<bean name="emp01" class="com.domain.Emp">
    <!-- 配置javabean属性的方法 -->
    <!-- 方法一：使用set方法赋值（最常使用）-->
    <property name="empno" value="1001"/>
    <property name="ename" value="张三"/>
    <property name="job" value="工程师"/>
    <property name="mgr" value="1000"/>
    <property name="hiredate" value="2000-10-9"/>
    <property name="sal" value="20000"/>
    <property name="comm" value="10000"/>
    <property name="deptno" value="10"/>
</bean>
```

4.1.2)测试代码：

```
@Test
//为属性赋值方法一：使用set方法赋值
public void test04(){
    ClassPathXmlApplicationContext ac =
            new ClassPathXmlApplicationContext("beans.xml");
    Emp emp = (Emp) ac.getBean("emp01");
    System.out.println(emp);
}
```

4.1.3)测试结果：

```
log4j:WARN No appenders could be found for logger (org.springframework.core.env.StandardEnvironment).
log4j:WARN Please initialize the log4j system properly.
Emp [empno=1001, ename=张三, job=工程师, mgr=1000, hiredate=2000-10-9, sal=20000, comm=10000, deptno=10]
```

4.2）构造方法注入：

4.2.1）beans.xml 文件：

```
<!-- 方法二：使用构造方法赋值（较使用）-->
<bean name="emp02" class="com.domain.Emp">
    <constructor-arg index="0" name="empno" value="1002"/>
    <constructor-arg index="1" name="ename" value="李四"/>
    <constructor-arg index="2" name="job" value="行政人员"/>
    <constructor-arg index="3" name="mgr" value="1001"/>
    <constructor-arg index="4" name="hiredate" value="2010-9-18"/>
    <constructor-arg index="5" name="sal" value="3000"/>
    <constructor-arg index="6" name="comm" value="100"/>
    <constructor-arg index="7" name="deptno" value="20"/>
</bean>
```

4.2.2)测试代码：

```
@Test
//为属性赋值方法二：使用构造方法赋值
public void test05(){
    ClassPathXmlApplicationContext ac =
            new ClassPathXmlApplicationContext("beans.xml");
    Emp emp = (Emp) ac.getBean("emp02");
    System.out.println(emp);
}
```

4.2.3)测试结果：

```
log4j:WARN No appenders could be found for logger (org.springframework.core.env.StandardEnvironment).
log4j:WARN Please initialize the log4j system properly.
Emp [empno=1002, ename=李四, job=行政人员, mgr=1001, hiredate=2010-9-18, sal=3000, comm=100, deptno=20]
```

4.3）p 名称空间注入：

4.3.1) beans.xml 文件：

```
<!-- 方法三：使用p名称空间(开发中不经常使用) -->
<bean name="emp03" class="com.domain.Emp"
      p:empno="1003" p:ename="王五"
      p:job="清洁工" p:mgr="1000"
      p:hiredate="2010-2-19" p:sal="3400"
      p:comm="100" p:deptno="20"/>
```

4.3.2)测试代码：

```
@Test
//为属性赋值方法三：使用p名称空间赋值
public void test06(){
    ClassPathXmlApplicationContext ac =
            new ClassPathXmlApplicationContext("beans.xml");
    Emp emp = (Emp) ac.getBean("emp03");
    System.out.println(emp);
}
```

4.3.3)测试结果：

```
log4j:WARN No appenders could be found for logger (org.springframework.core.env.StandardEnvironment)
log4j:WARN Please initialize the log4j system properly.
Emp [empno=1003, ename=王五, job=清洁工, mgr=1000, hiredate=2010-2-19, sal=3400, comm=100, deptno=20]
```

4.4）spel 注入：

4.4.1）beans.xml 文件：

```
<!-- 方法四：使用springel(Spring Expression Language)(开发中不经常使用) -->
<bean name="emp04" class="com.domain.Emp" >
      <property name="empno" value="#{emp03.empno}"/>
      <property name="ename" value="赵六"/>
      <property name="sal" value="#{emp01.sal}"/>
</bean>
```

4.4.2)测试代码：

```
@Test
//为属性赋值方法四：使用springel赋值
public void test07(){
    ClassPathXmlApplicationContext ac =
            new ClassPathXmlApplicationContext("beans.xml");
    Emp emp = (Emp) ac.getBean("emp04");
    System.out.println(emp);
}
```

4.4.3)测试结果：

```
log4j:WARN No appenders could be found for logger (org.springframework.core.env.StandardEnvironment).
log4j:WARN Please initialize the log4j system properly.
Emp [empno=1003, ename=赵六, job=null, mgr=0, hiredate=null, sal=20000, comm=0, deptno=0]
```

5、Spring 复杂类型注入(非常重要)

5.1）数组的注入：

5.1.1）Beans.xml 文件：

```
<bean name="collectionBean" class="com.domain.CollectionBean">
    <!-- 1、注入数组类型 -->
    <property name="names">
        <array>
            <value>张三</value>
            <value>李四</value>
            <value>王五</value>
            <value>赵六</value>
        </array>
    </property>
</bean>
```

为数组类型赋值。

5.1.2）测试代码：

```
@Test
//复杂属性赋值一：为数组赋值
public void test08(){
    ClassPathXmlApplicationContext ac =
            new ClassPathXmlApplicationContext("beans.xml");
    CollectionBean cb = (CollectionBean) ac.getBean("collectionBean");
    System.out.println(cb);
}
```

5.1.3）测试效果：

```
log4j:WARN No appenders could be found for logger (org.springframework.core.env.StandardEnvi
log4j:WARN Please initialize the log4j system properly.
CollectionBean [names=[张三, 李四, 王五, 赵六]]
```

5.2）list 集合的注入：

5.2.1)beans.xml 文件：

```
<!-- 定义几本图书 -->
    <bean name="book1" class="com.domain.Book"
          p:bid="100" p:bname="《三国演义》"
```

```
        p:author="罗贯中" p:price="100"
        p:publisher="中国青年出版社"/>
    <bean name="book2" class="com.domain.Book"
        p:bid="101" p:bname="《西游记》"
        p:author="吴承恩" p:price="90"
        p:publisher="中国邮电出版社"/>
    <bean name="book3" class="com.domain.Book"
        p:bid="102" p:bname="《红楼梦》"
        p:author="曹雪芹" p:price="110"
        p:publisher="人民出版社"/>
    <!-- 定义 com.domain.CollectionBean 对象 -->
    <bean name="collectionBean" class="com.domain.CollectionBean">
        <!-- 1、注入数组类型 -->
        <property name="names">
            <array>
                <value>张三</value>
                <value>李四</value>
                <value>王五</value>
                <value>赵六</value>
            </array>
        </property>
        <!-- 2、注入 list 集合 -->
        <property name="books">
            <list>
                <ref bean="book1"/>
                <ref bean="book2"/>
                <ref bean="book3"/>
            </list>
        </property>
    </bean>
```

5.2.2)测试代码：

```
@Test
    //复杂属性赋值二：为 list 集合赋值
    public void test09(){
        ClassPathXmlApplicationContext ac =
                new ClassPathXmlApplicationContext("beans.xml");
        CollectionBean cb = (CollectionBean) ac.getBean("collectionBean");
        System.out.println(cb);
    }
```

5.2.3)测试效果：

```
CollectionBean [names=[张三, 李四, 王五, 赵六], books=[Book [bid=100, bname=
《三国演义》, author=罗贯中, price=100.0, publisher=中国青年出版社], Book [bid=101,
bname=《西游记》, author=吴承恩, price=90.0, publisher=中国邮电出版社], Book
```

[bid=102, bname=《红楼梦》, author=曹雪芹, price=110.0, publisher=人民出版社]]]

5.3）Map 集合的注入

5.3.1）beans.xml 文件：

```
<!-- 3、注入 map 集合,注意下面使用 springel 可以取出对象中的属性值作为 map 的 key -->
        <property name="maps">
            <map>
                <entry key="#{book1.bid}" value-ref="book1"/>
                <entry key="#{book2.bid}" value-ref="book2"/>
                <entry key="#{book3.bid}" value-ref="book3"/>
            </map>
        </property>
```

5.3.2)测试代码：

```
@Test
    //复杂属性赋值三：为 map 集合赋值
    public void test10(){
        ClassPathXmlApplicationContext ac =
                new ClassPathXmlApplicationContext("beans.xml");
        CollectionBean cb = (CollectionBean) ac.getBean("collectionBean");
        System.out.println(cb);
    }
```

5.3.3)测试效果：（打印三种对象）

CollectionBean [names=[张三, 李四, 王五, 赵六], books=[Book [bid=100, bname=《三国演义》, author=罗贯中, price=100.0, publisher=中国青年出版社], Book [bid=101, bname=《西游记》, author=吴承恩, price=90.0, publisher=中国邮电出版社], Book [bid=102, bname=《红楼梦》, author=曹雪芹, price=110.0, publisher=人民出版社]], maps={100=Book [bid=100, bname=《三国演义》, author=罗贯中, price=100.0, publisher=中国青年出版社], 101=Book [bid=101, bname=《西游记》, author=吴承恩, price=90.0, publisher=中国邮电出版社], 102=Book [bid=102, bname=《红楼梦》, author=曹雪芹, price=110.0, publisher=人民出版社]}]

5.4）properties 集合的注入：

5.4.1) beans.xml 文件：

```
        <!-- 4、注入 properties 对象 -->
        <property name="properties">
            <props>
                <prop key="jdbcDriver">com.mysql.jdbc.Driver</prop>
                <prop key="jdbcUrl">jdbc:mysql:///java1301</prop>
```

```
            <prop key="username">root</prop>
            <prop key="password">123</prop>
        </props>
    </property>
```

5.4.2) 测试代码：

```
@Test
    //复杂属性赋值四：为 properties 集合赋值
    public void test11(){
        ClassPathXmlApplicationContext ac =
                new ClassPathXmlApplicationContext("beans.xml");
        CollectionBean cb = (CollectionBean) ac.getBean("collectionBean");
        System.out.println(cb);
    }
```

5.4.3) 测试效果：

CollectionBean [names=[张三, 李四, 王五, 赵六], books=[Book [bid=100, bname=《三国演义》, author=罗贯中, price=100.0, publisher=中国青年出版社], Book [bid=101, bname=《西游记》, author=吴承恩, price=90.0, publisher=中国邮电出版社], Book [bid=102, bname=《红楼梦》, author=曹雪芹, price=110.0, publisher=人民出版社]], maps={100=Book [bid=100, bname=《三国演义》, author=罗贯中, price=100.0, publisher=中国青年出版社], 101=Book [bid=101, bname=《西游记》, author=吴承恩, price=90.0, publisher=中国邮电出版社], 102=Book [bid=102, bname=《红楼梦》, author=曹雪芹, price=110.0, publisher=人民出版社]}, properties={jdbcDriver=com.ocacle.jdbc.Driver,password=tiger, jdbcUrl=jdbc:oracle:thin:1521://orcl, username=scott}]

JavaBean 实体类 CollectionBean.java 文件定义如下：

```
public class CollectionBean {

    //1.数组类型
    private String[] names;
    //2.集合类型
    private List<Book> books;
    //3.map集合
    private Map<String, Book> maps;
    //4.properties对象
    private Properties properties;
```

6、MySQL-jdbc 访问数据库

6.1）访问普通的 jdbc 的方法：（使用属性文件）

6.1.1）在类路径下定义 db.properties 文件（本例定义在 config/目录下）

```
db.driver=com.mysql.jdbc.Driver
db.url=jdbc:mysql:///java1301?useUnicode=true&amp;characterEncoding=utf-8
db.user=root
db.password=123
```

6.1.2）定义 jdbcUtils 工具类：

```
public class JdbcUtils {
    private static Properties db;
    private static String driver;
    private static String url;
    private static String user;
    private static String password;
    public void setDb(Properties db) {
        this.db = db;
    }
    public static Connection getConn() {
        try {
            // 1.加载资源
            driver = db.getProperty("driver");
            url = db.getProperty("url");
            user = db.getProperty("user");
            password = db.getProperty("password");
            // 2.加载驱动
            Class.forName(driver);
            // 3.得到连接
            return DriverManager.getConnection(url, user, password);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }
    // 关闭资源
    public static void closeAll(Statement ps, Connection conn, ResultSet rs)
{
```

```
        try {
            if (rs != null)
                rs.close();
            if (ps != null)
                ps.close();
            if (conn != null)
                conn.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

6.1.3）定义 myBeans5.xml 配置文件：（最核心）

```
<?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-4.2.xsd
                http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-4.2.xsd">
<!-- 1、引入 db.properties 属性文件 -->
    <context:property-placeholder location="classpath:db.properties"/>
    <!-- 2、对 JdbcUtils 中的 properties 属性注入值 -->
    <bean id="jdbcUtils" class="com.zelin.utils.JdbcUtils">
        <property name="db">
            <props>
                <prop key="driver">${db.driver}</prop>
                <prop key="url">${db.url}</prop>
                <prop key="user">${db.user}</prop>
                <prop key="password">${db.password}</prop>
            </props>
        </property>
    </bean>
</beans>
```

6.1.4）测试运行结果：

```
log4j:WARN No appenders could be found for logger (org.spr
log4j:WARN Please initialize the log4j system properly.
com.mysql.jdbc.JDBC4Connection@442675e1
```

6.2) 访问 C3p0 数据库连接池方法：

6.2.1）在类路径下定义 db.properties 文件：（同前面）

6.2.2）定义 JdbcC3p0PooledUtils 访问 c3p0 的工具类：

```
public class JdbcC3p0PooledUtils {
    private static ComboPooledDataSource cpd ;
    public static ComboPooledDataSource getCpd() {
        return cpd;
    }
    public static void setCpd(ComboPooledDataSource cpd) {
        JdbcC3p0PooledUtils.cpd = cpd;
    }
    //得到数据源
    public static DataSource getDataSource(){
        return cpd;
    }
    //得到连接对象
    public  Connection getConnection(){
        try {
            return cpd.getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

6.2.3）定义 myBeans6.xml 配置文件(重要):

```
<?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-4.2.xsd
        http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.2.xsd">
    <!-- 1、引入 db.properties 属性文件 -->
    <context:property-placeholder location="classpath:db.properties"/>
    <!-- 2、对 JdbcUtils 中的 properties 属性注入值 -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${db.driver}"/>
```

```
        <property name="jdbcUrl" value="${db.url}"/>
        <property name="user" value="${db.user}"/>
        <property name="password" value="${db.password}"/>
    </bean>
    <bean id="c3p0Pool" class="com.zelin.utils.JdbcC3p0PooledUtils">
        <property name="cpd" ref="dataSource"/>
    </bean>
</beans>
```

6.2.4）测试 c3p0 连接池代码：

```
public class TestJdbcUtils {
    @Test
    public void testGetConnection(){
        ApplicationContext ac  = new
ClassPathXmlApplicationContext("myBeans5.xml");
        JdbcUtils jdbcUtils = (JdbcUtils) ac.getBean("jdbcUtils");
        Connection conn = JdbcUtils.getConn();
        System.out.println(conn);
    }
}
```

6.2.5）运行效果如下：

```
log4j:WARN No appenders could be found for logger (org.springframework.
log4j:WARN Please initialize the log4j system properly.
com.mchange.v2.c3p0.impl.NewProxyConnection@6f3b5d16
```

6.3) 在 web 程序中使用 spring 框架：

6.3.1)在/WEB-INF/定义 applicationContext.xml 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: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-4.2.xsd">

    <!-- 1.从 db.properties 文件中读取数据库访问的属性信息 -->
    <!-- 1.1)读取属性文件的方法一 -->
    <!-- <context:property-placeholder location="classpath:db.properties"/> -->
    <!-- 1.2)读取属性文件的方法二 -->
    <bean
```

```
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location" value="classpath:db.properties"/>
    </bean>
    <!-- 2.定义 c3p0 连接池（数据源） -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${db.driver}"/>
        <property name="jdbcUrl" value="${db.url}"/>
        <property name="user" value="${db.user}"/>
        <property name="password" value="${db.password}"/>
    </bean>
    <!-- 定义一个 queryQunner 对象 -->
    <bean id="qr" class="org.apache.commons.dbutils.QueryRunner">
        <constructor-arg name="ds" ref="dataSource"/>
    </bean>
    <!-- 定义一个 StudentDaoImpl -->
    <bean id="studentDao" class="com.zelin.dao.impl.StudentDaoImpl">
        <property name="qr" ref="qr"/>
    </bean>
</beans>
```

6.3.2)定义 StudentDaoImpl 类:

```
public class StudentDaoImpl implements StudentDao {
    private QueryRunner qr;
    public void setQr(QueryRunner qr) {
        this.qr = qr;
    }
    public QueryRunner getQr() {
        return qr;
    }
    @Override
    public List<Student> findAll() throws Exception {
        return qr.query("select * from student", new
BeanListHandler<>(Student.class));
    }
}
```

6.3.3)定义 StudentServlet 类:

```
public class StudentServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private WebApplicationContext wac;
    private StudentDao studentDao;
    @Override
    public void init() throws ServletException {
        ServletContext sc = this.getServletContext();
        //1.得到 spring 的 web 上下文对象(得到 spring 容器)
```

```
        wac =
WebApplicationContextUtils.getWebApplicationContext(getServletContext());
        //2.从 spring 容器取取出 studentDao 对象
        studentDao = (StudentDao) wac.getBean("studentDao");
    }
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        resp.setContentType("text/html;charset=utf-8");
        //1.得到请求参数
        String method = req.getParameter("method");
        //2.根据参数调用不同的方法
        if("list".equals(method)){
            this.list(req,resp);        //列表所有的学生
        }
    }
    //列表所有的学生
    private void list(HttpServletRequest req, HttpServletResponse resp) {
        try {
            //1.得到所有的学生
            List<Student> students = studentDao.findAll();
            //2.转换上面的集合为 json 串
            String json = JSON.toJSONString(students);
            //3.输出 json 到客户端
            resp.getWriter().print(json);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

6.3.4)定义 index.html:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
  <link rel="stylesheet" href="bootstrap-3.3.7/css/bootstrap.min.css">
  <script src="bootstrap-3.3.7/js/jquery.min.js"></script>
  <script src="bootstrap-3.3.7/js/bootstrap.min.js"></script>
  <script src="bootstrap-3.3.7/js/docs.min.js"></script>
</head>
<body>
```

```
<div class="container">
    <div class="page-header">
        <h3>学生列表</h3>
    </div>
    <div class="panel panel-primary">
        <div class="panel-heading">
            <h3>学生列表</h3>
        </div>
        <table class="table table-hover">
            <tr>
                <td>学号</td>
                <td>姓名</td>
                <td>性别</td>
                <td>年龄</td>
                <td>住址</td>
                <td>操作</td>
            </tr>
            <tbody>

            </tbody>
        </table>
    </div>
</div>
<script>
    $(function(){
        $.post(
            "student?method=list",
            function(data){
                var info = "";
                $.each(data,function(i,v){
                    info += "<tr>";
                    info += "<td>" + v.sid + "</td>"
                    info += "<td>" + v.sname + "</td>"
                    info += "<td>" + v.sex + "</td>"
                    info += "<td>" + v.age + "</td>"
                    info += "<td>" + v.addr + "</td>"
                    info += "<td>修改  删除</td>"
                    info += "</tr>";
                })
                $('table tbody').html(info);
            },'json');
    })

</script>
```

```
</body>
</html>
```

6.3.5)web.xml 文件中定义监听器：

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

6.3.6)运行效果如下：

学生列表

学生列表

1	张三	男	20	上海	修改 删除
2	小五	男	28	广州	修改 删除
3	小红	女	19	杭州	修改 删除
4	王二小	男	12	岳阳	修改 删除
5	小2添	女	18	襄阳	修改 删除
6	黄家驹	男	54	香港	修改 删除
7	罗成	男	22	邵阳	修改 删除
8	魏征	男	28	洛ab阳	修改 删除
9	徐达	男	29	江苏无锡	修改 删除
12	小虎	男	21	上海	修改 删除
13	孙宾	男	35	上海浦东	修改 删除
15	赵本山	男	65	东北大街	修改 删除