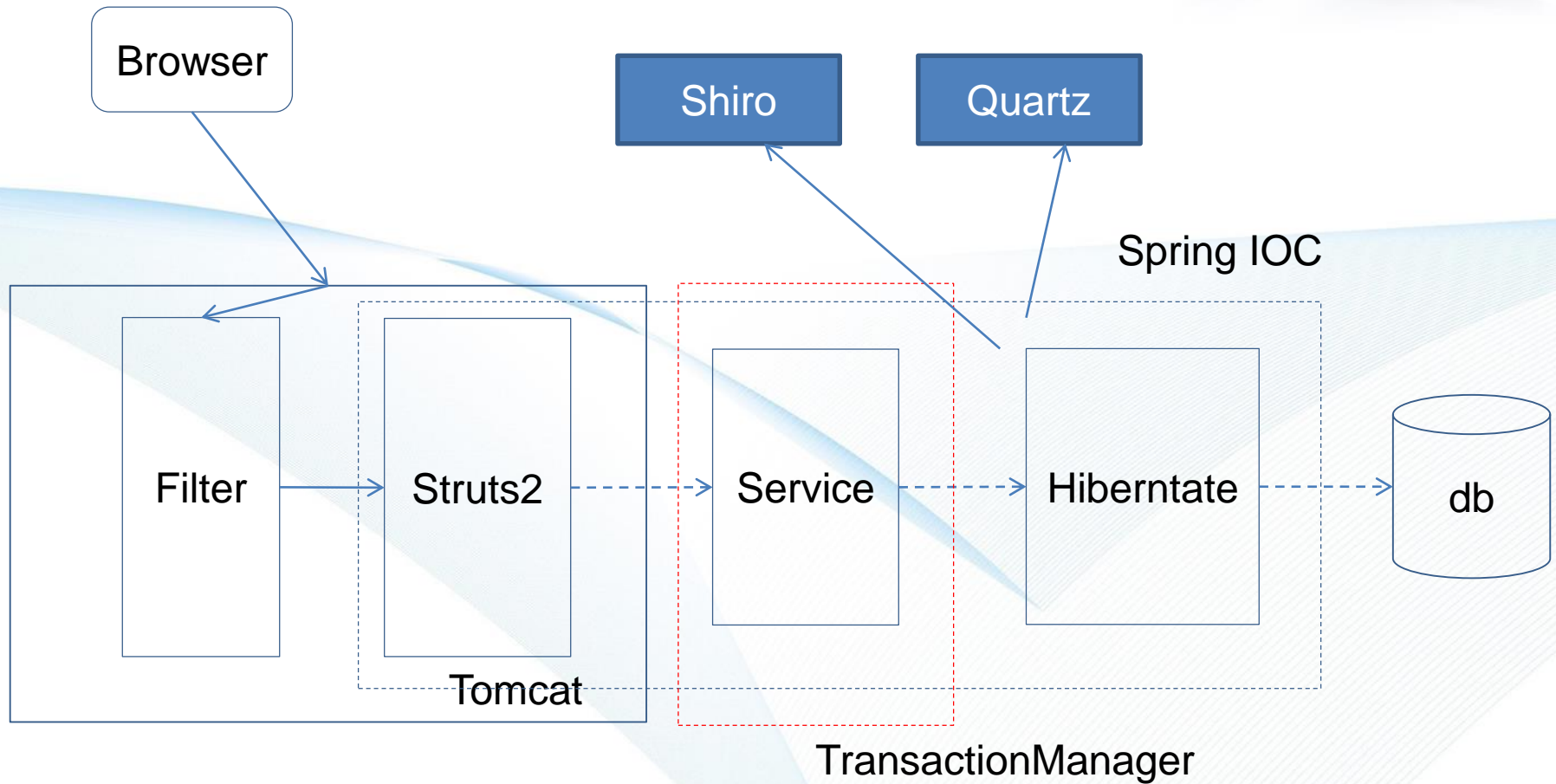




# B\S





Yellow header bar with a faint image of a hand typing on a keyboard.

# Hello World



Large blue wavy graphic element on the left side of the slide.

---

# Spring 是什么 (1)

- Spring 是一个开源框架.
- Spring 为简化企业级应用开发而生. 使用 Spring 可以使简单的 JavaBean 实现以前只有 EJB 才能实现的功能.
- Spring 是一个 IOC (DI) 和 AOP 容器框架.

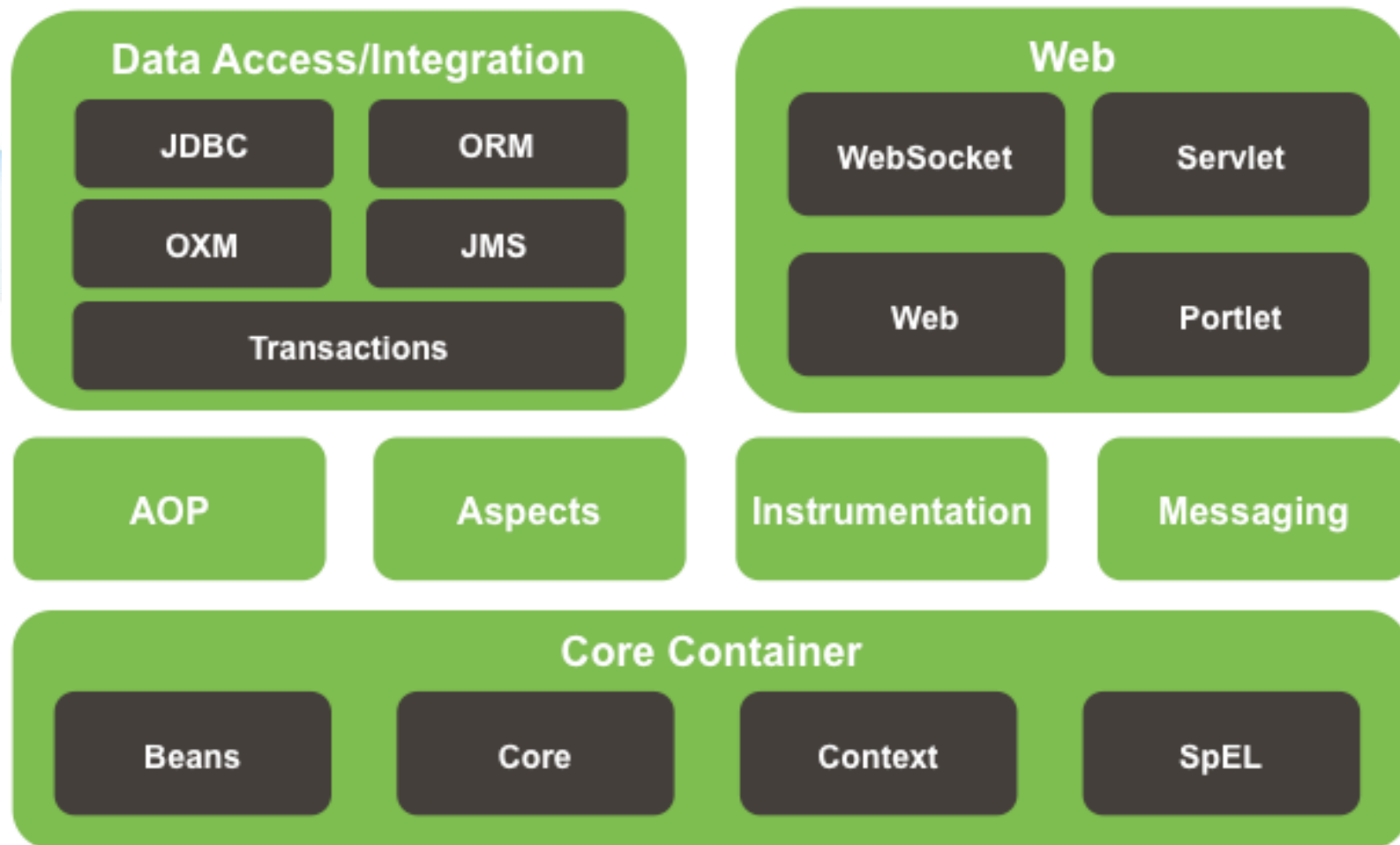


# Spring 是什么 (2)

## ■ 具体描述 Spring:

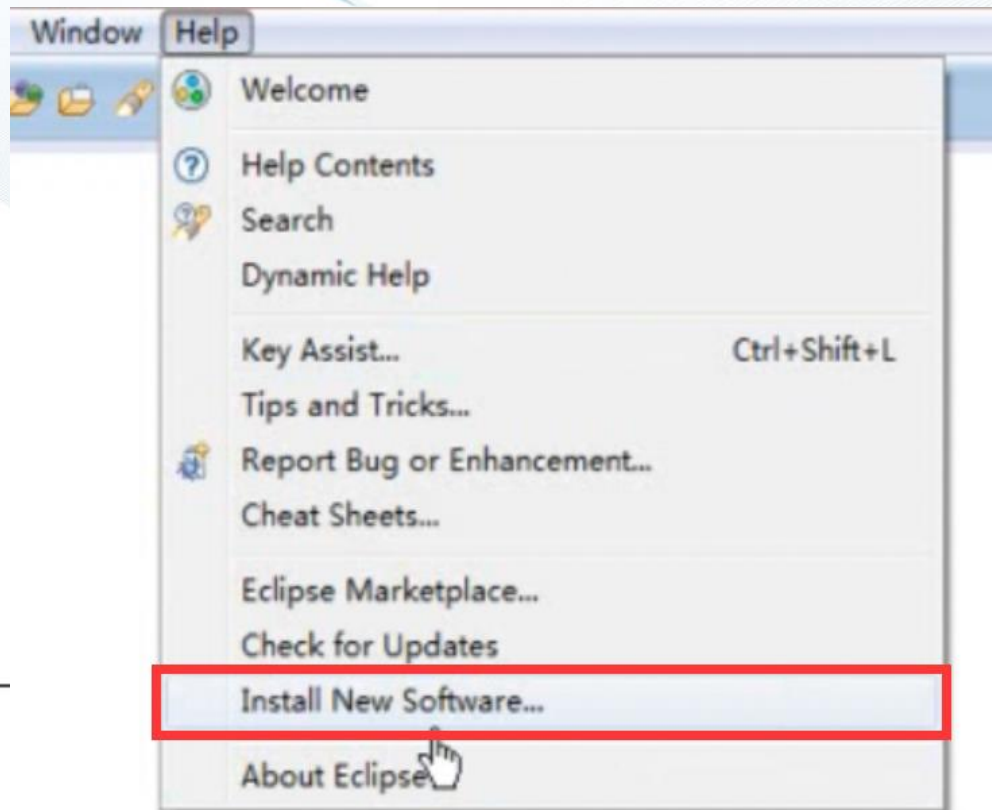
- + **轻量级**: Spring 是非侵入性的 - 基于 Spring 开发的应用中的对象可以不依赖于 Spring 的 API
- + **依赖注入** (DI --- dependency injection、IOC)
- + **面向切面编程** (AOP --- aspect oriented programming)
- + **容器**: Spring 是一个容器, 因为它包含并且管理应用对象的生命周期
- + **框架**: Spring 实现了使用简单的组件配置组合成一个复杂的应用. 在 Spring 中可以使用 XML 和 Java 注解组合这些对象
- + **一站式**: 在 IOC 和 AOP 的基础上可以整合各种企业应用的开源框架和优秀的第三方类库 (实际上 Spring 自身也提供了展现层的 SpringMVC 和 持久层的 Spring JDBC)

# Spring 模块



# 安装 SPRING TOOL SUITE

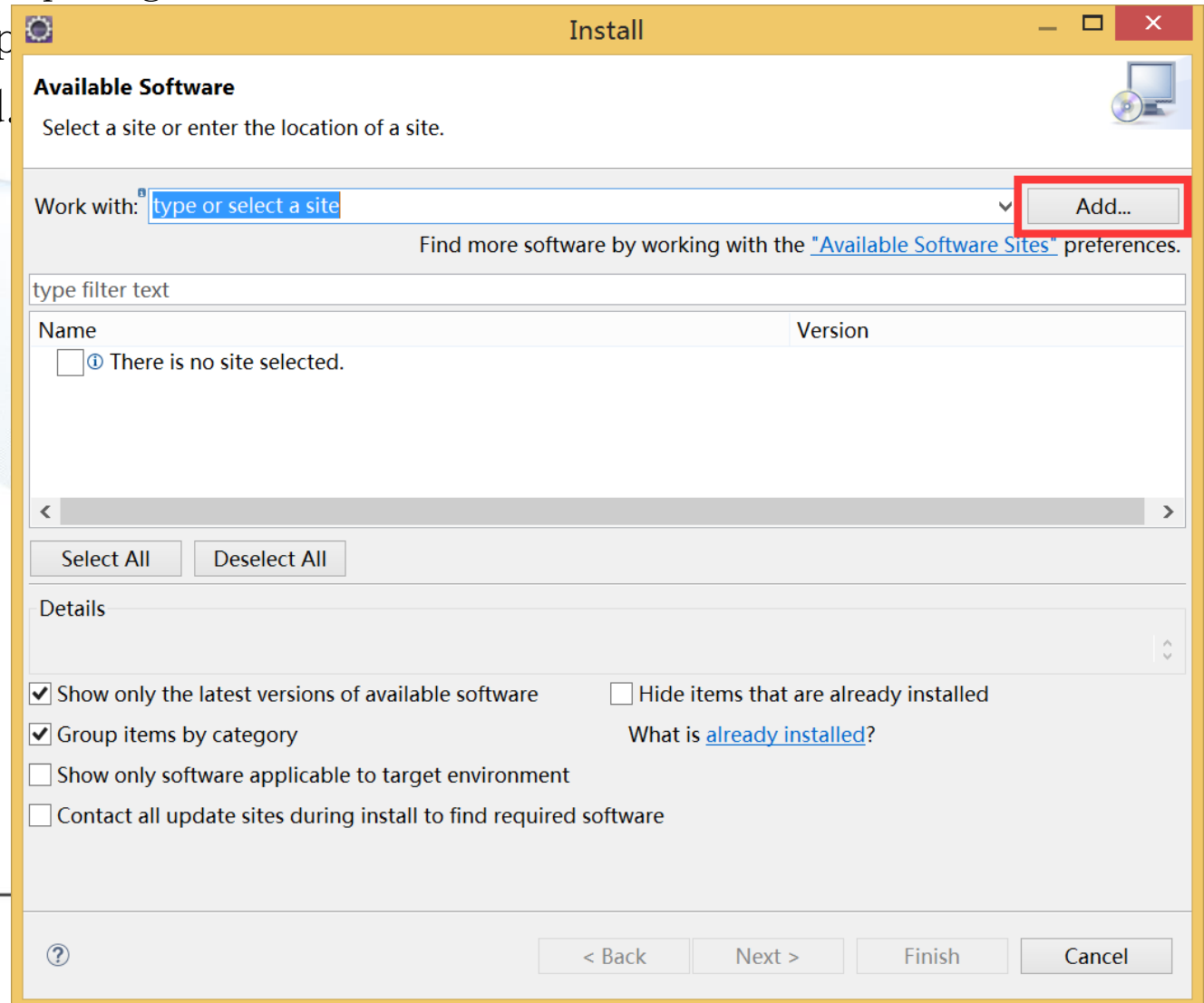
- SPRING TOOL SUITE 是一个 Eclipse 插件，利用该插件可以更方便的在 Eclipse 平台上开发基于 Spring 的应用。
- 安装方法说明（springsource-tool-suite-3.4.0.RELEASE-e4.3.1-updatesite.zip）：
  - + **Help** --> **Install New Software...**



# 安装 SPRING TOOL SUITE

- 安装方法说明 (springsource-tool-suite-3.4.0.RELEASE-e4.3.1-updatesite.zip)

+ Click Add...

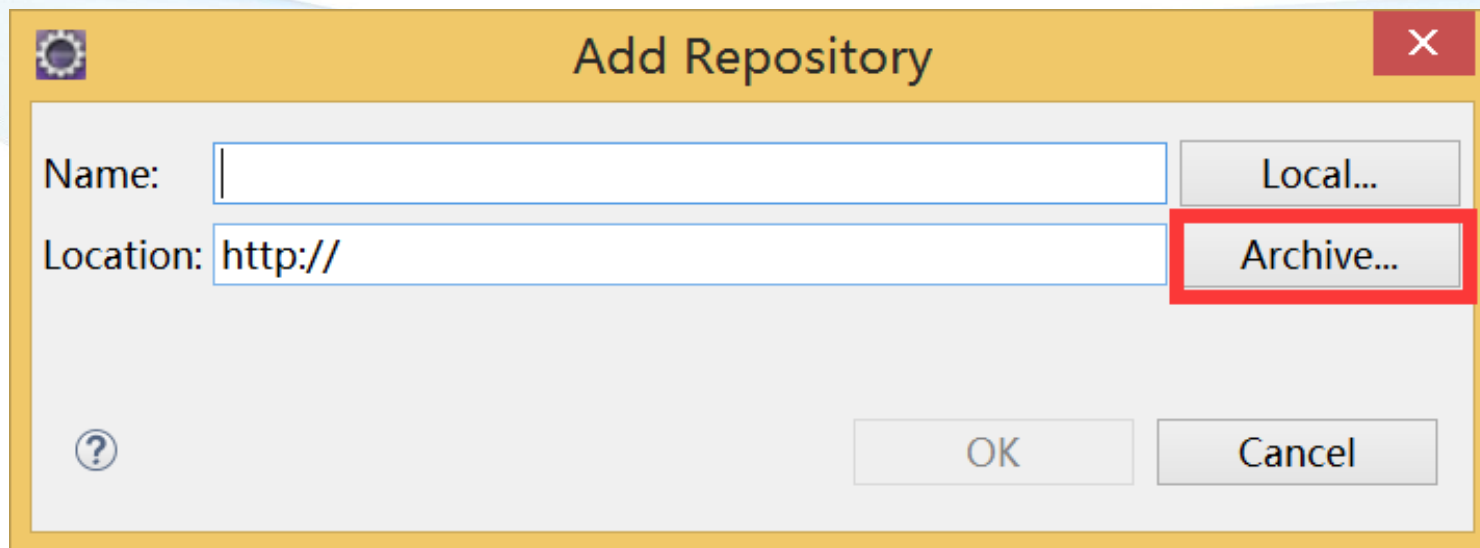




# 安装 SPRING TOOL SUITE

- 安装方法说明（springsource-tool-suite-3.4.0.RELEASE-e4.3.1-updatesite.zip）：

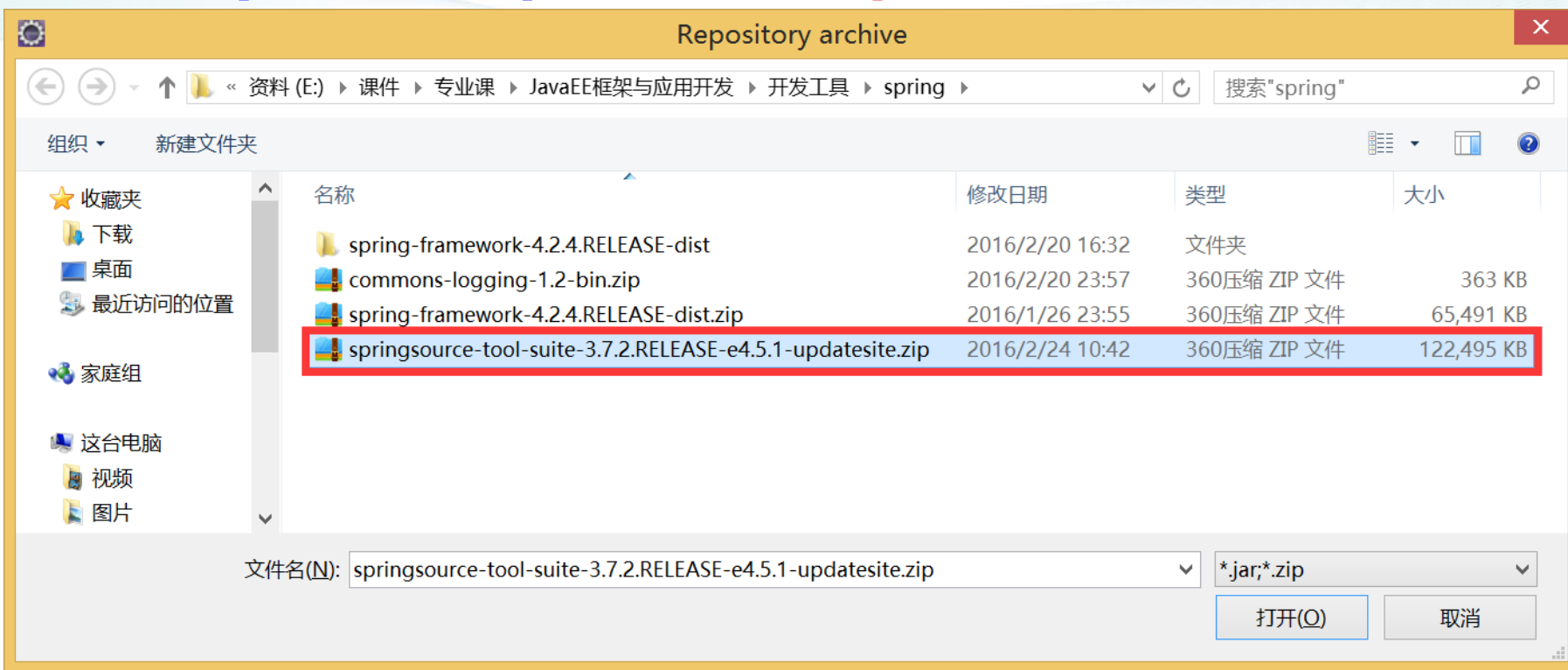
+ In dialog Add Site dialog, click **Archive...**



# 安装 SPRING TOOL SUITE

- 安装方法说明 (springsource-tool-suite-3.4.0.RELEASE-e4.3.1-updatesite.zip) :

+ Navigate to [springsource-tool-suite-3.4.0.RELEASE-e4.3.1-updatesite.zip](#) and click **Open**



# 安装 SPRING TOOL SUITE

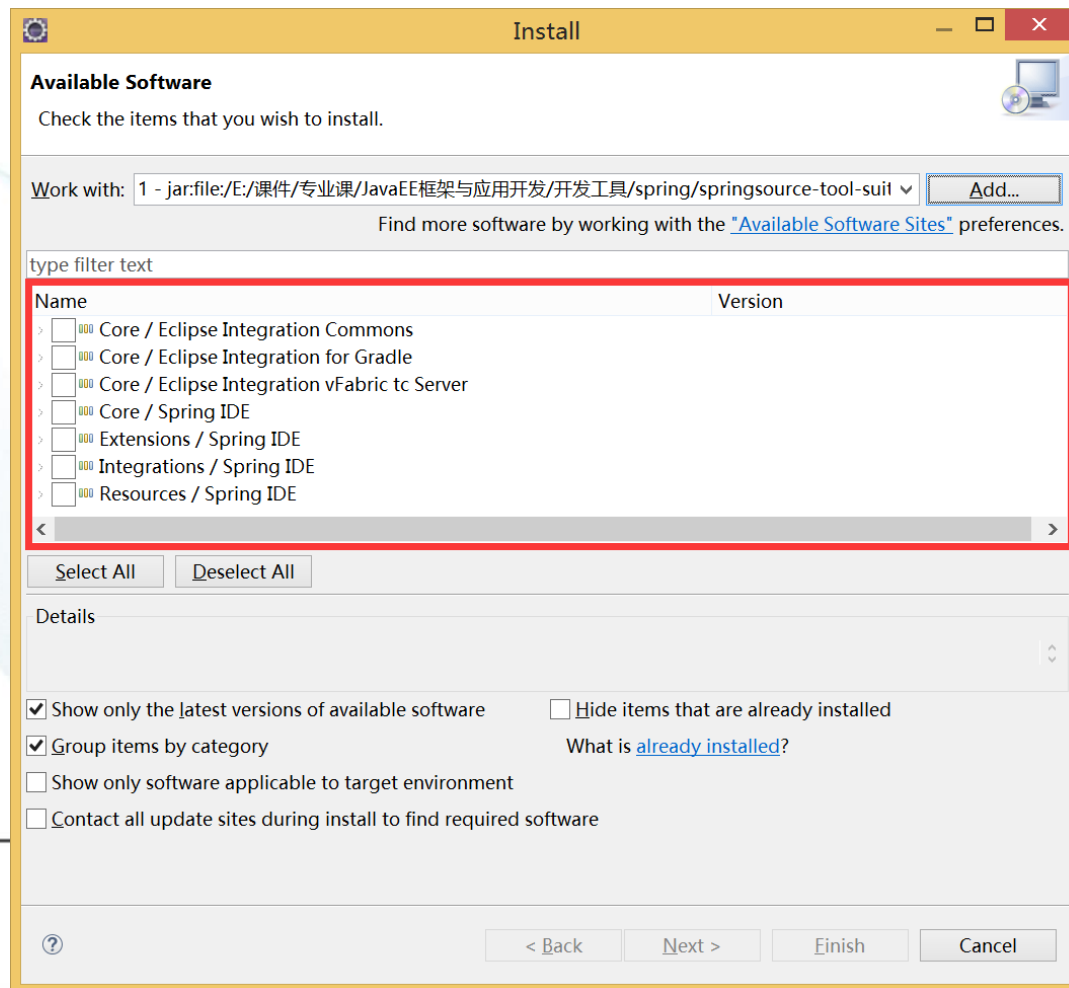
- 安装方法说明（springsource-tool-suite-3.7.0.RELEASE-e4.3.1-updatesite.zip）：
  - + Clicking **OK** in the Add Site dialog will bring you back to the dialog 'Install'



# 安装 SPRING TOOL SUITE

- 安装方法说明（springsource-tool-suite-3.7.0.RELEASE-e4.3.1-updatesite.zip）：

+ Clicking **OK** in the Add Site dialog will bring you back to the dialog 'Install'



# 安装 SPRING TOOL SUITE

- 安装方法说明（springsource-tool-suite-3.7.0.RELEASE-e4.3.1-updatesite.zip）：

+ Select the **xxx/Spring IDE** that has appeared

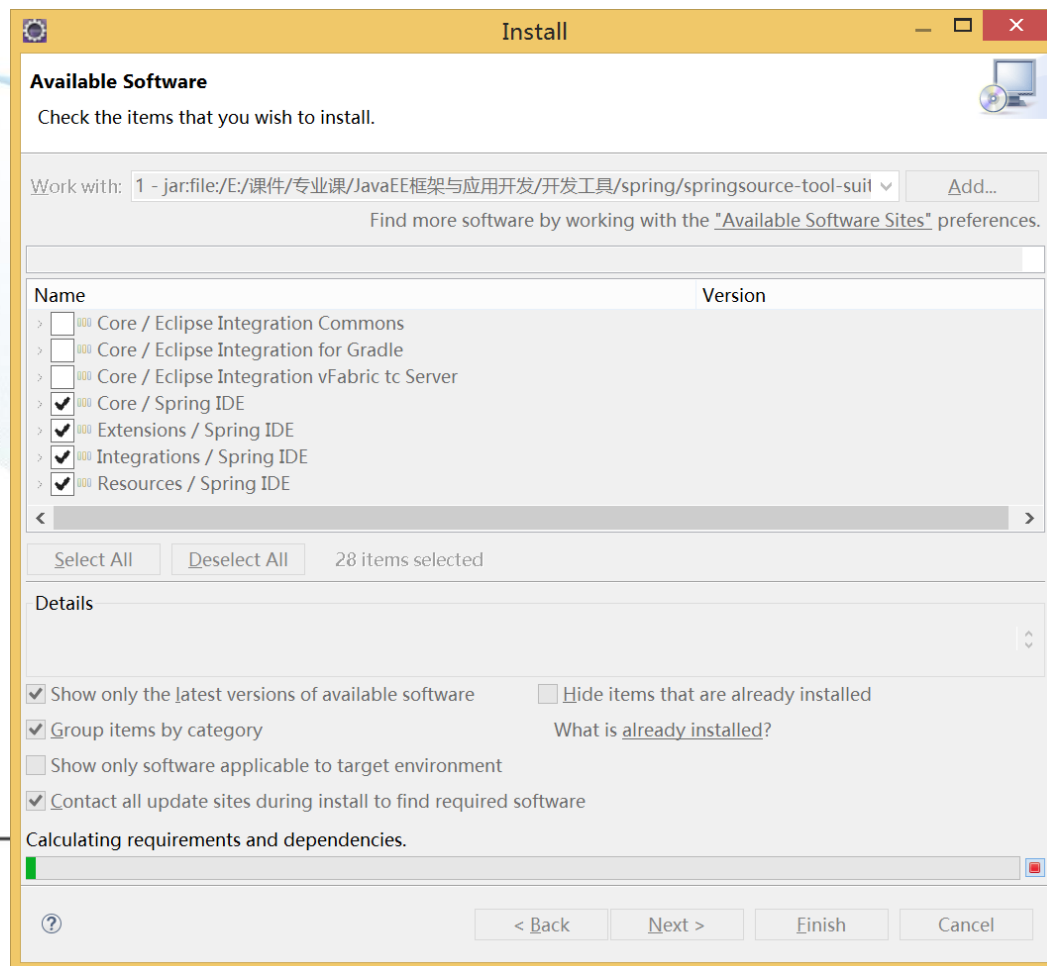
Name	Version
> <input type="checkbox"/> Core / Eclipse Integration Commons	
> <input type="checkbox"/> Core / Eclipse Integration for Gradle	
> <input type="checkbox"/> Core / Eclipse Integration vFabric tc Server	
> <input checked="" type="checkbox"/> Core / Spring IDE	
> <input checked="" type="checkbox"/> Extensions / Spring IDE	
> <input checked="" type="checkbox"/> Integrations / Spring IDE	
> <input checked="" type="checkbox"/> Resources / Spring IDE	



# 安装 SPRING TOOL SUITE

- 安装方法说明（springsource-tool-suite-3.7.0.RELEASE-e4.3.1-updatesite.zip）：

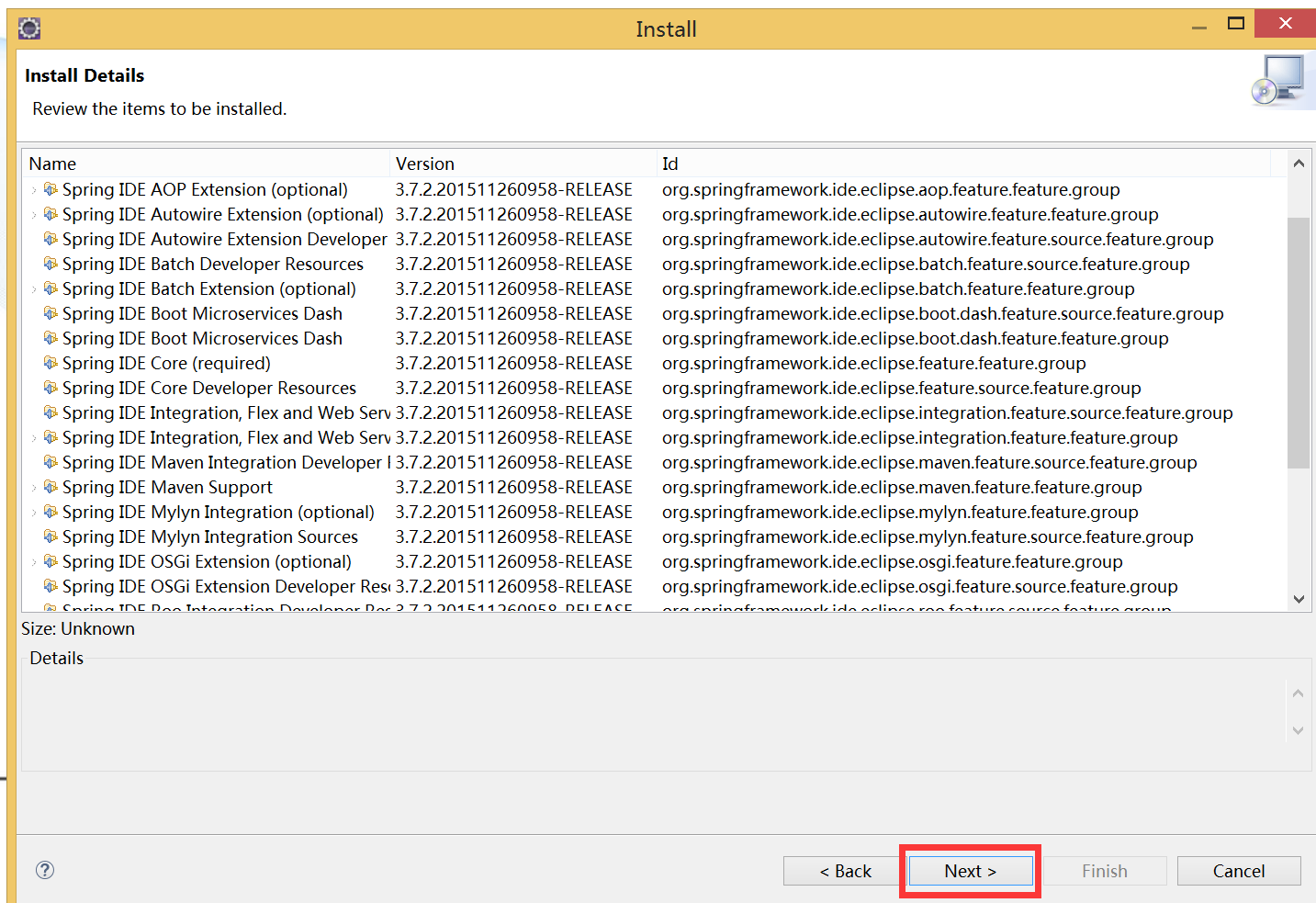
+ Click **Next** and then **Finish**



# 安装 SPRING TOOL SUITE

- 安装方法说明（springsource-tool-suite-3.7.0.RELEASE-e4.3.1-updatesite.zip）：

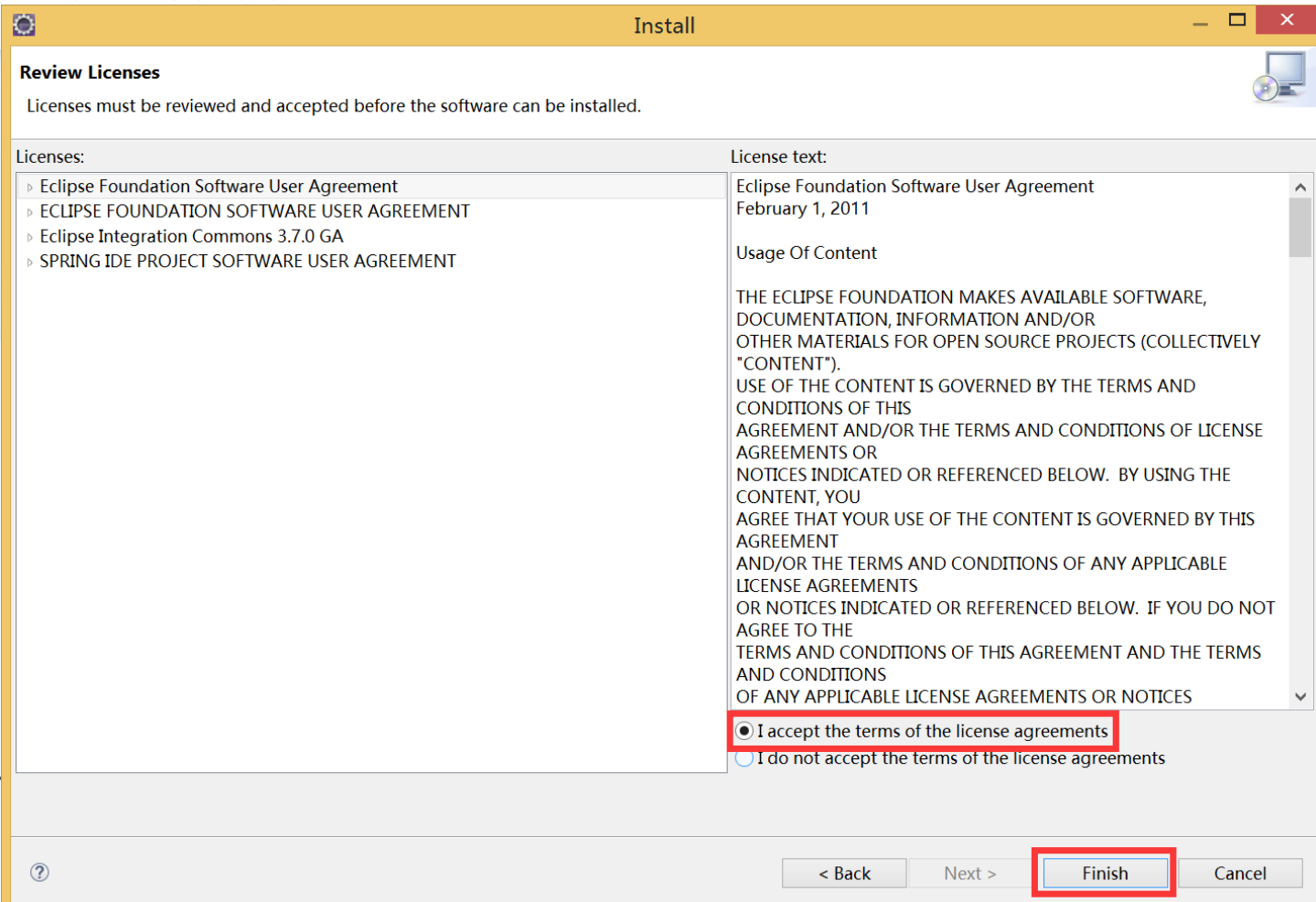
+ Approve the license



# 安装 SPRING TOOL SUITE

- 安装方法说明 (springsource-tool-suite-3.7.0.RELEASE-e4.3.1-updatesite.zip) :

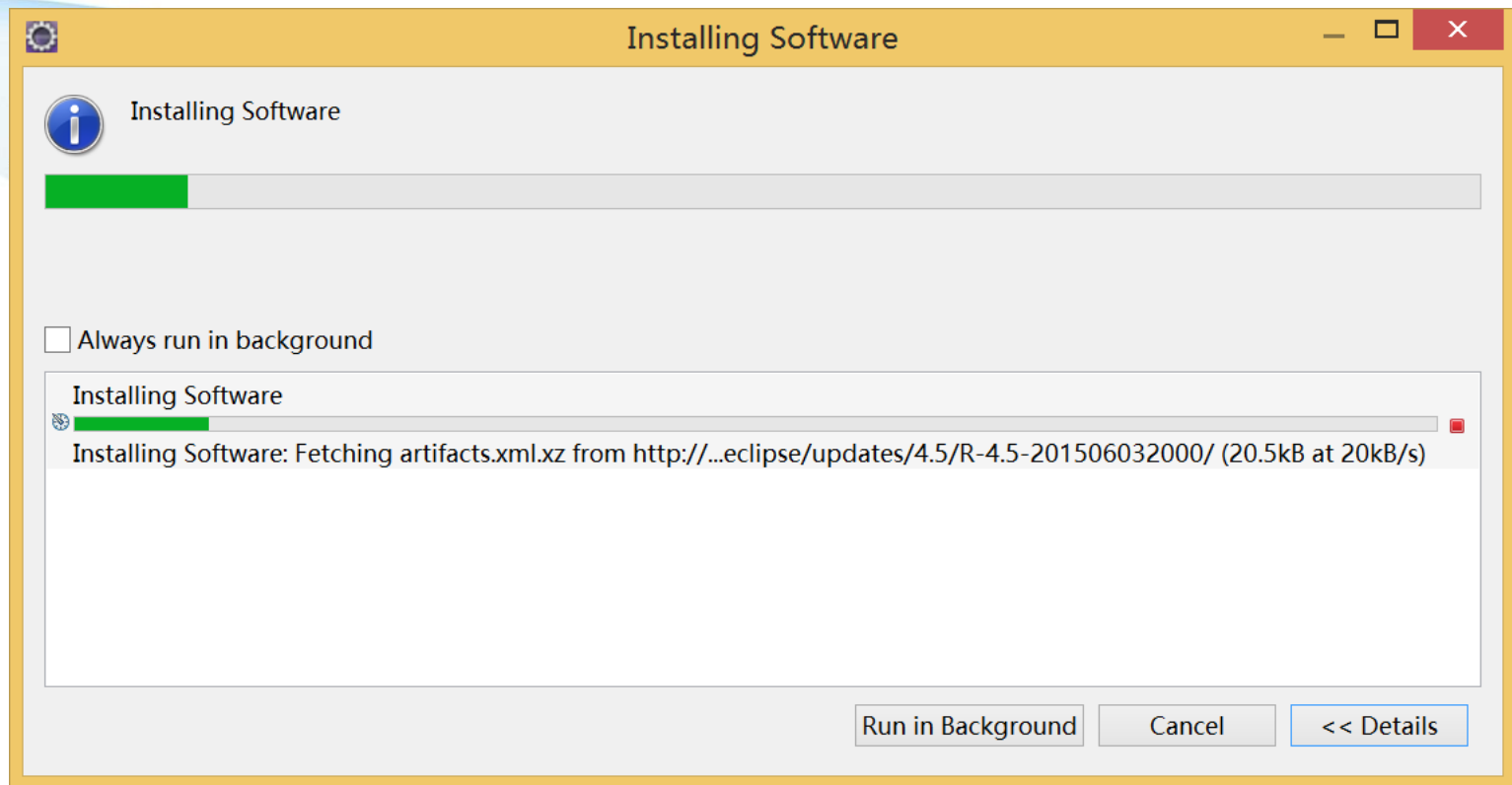
+ Approve the license



# 安装 SPRING TOOL SUITE

- 安装方法说明（springsource-tool-suite-3.7.0.RELEASE-e4.3.1-updatesite.zip）：

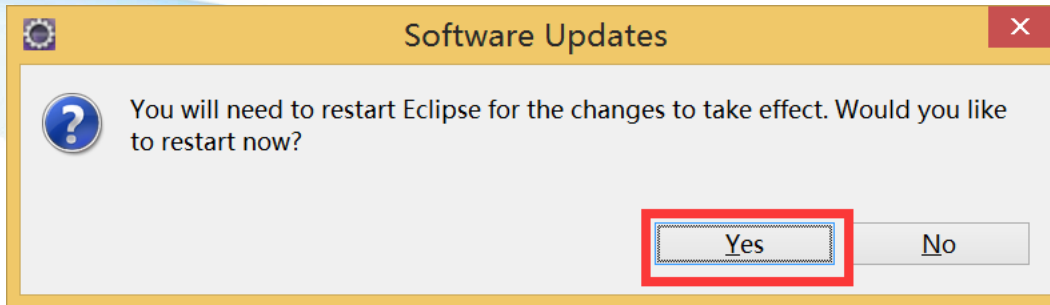
+ Restart eclipse when that is asked



# 安装 SPRING TOOL SUITE

- 安装方法说明（springsource-tool-suite-3.7.0.RELEASE-e4.3.1-updatesite.zip）：

+ Restart eclipse when that is asked

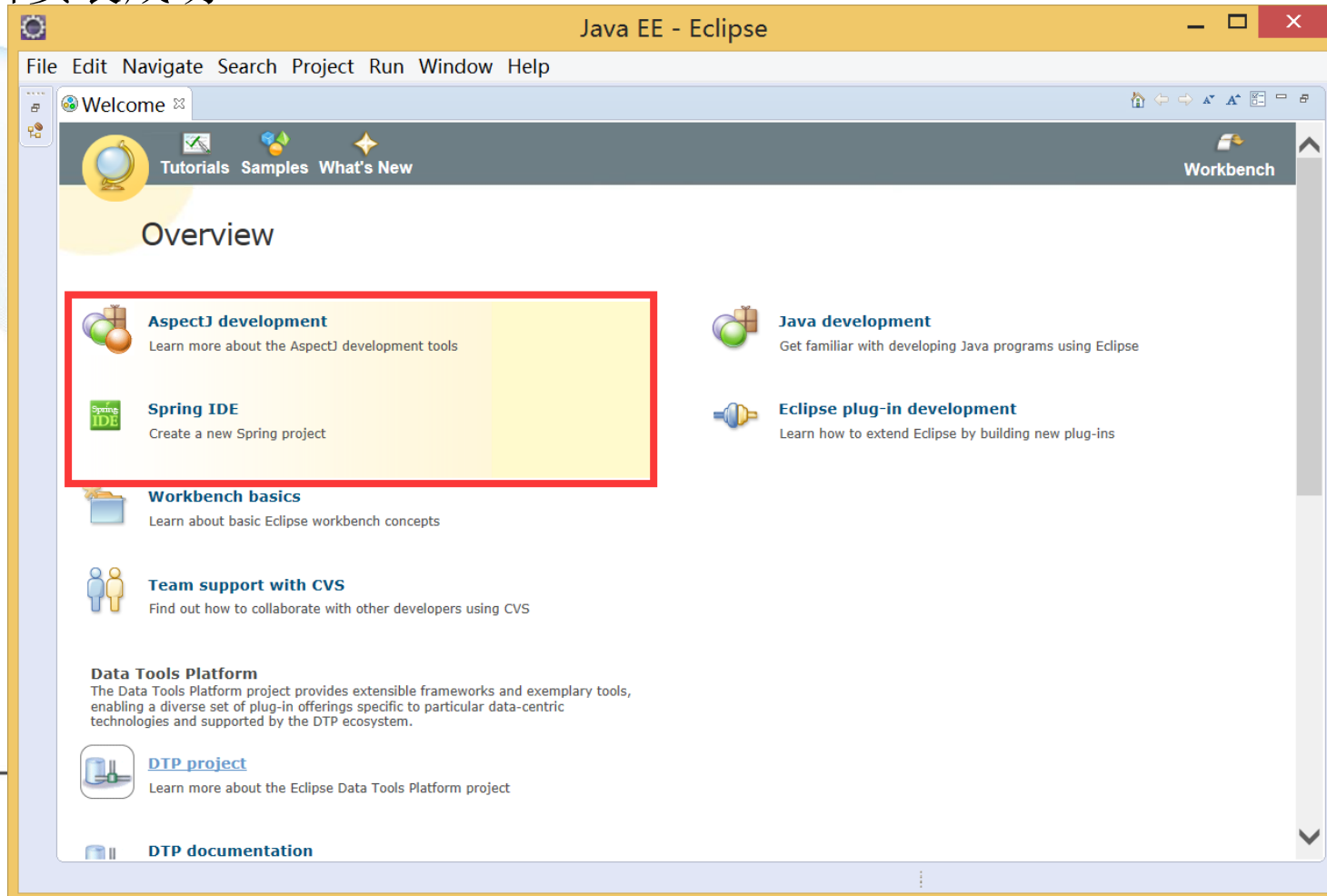




# 安装 SPRING TOOL SUITE

- 安装方法说明（springsource-tool-suite-3.7.0.RELEASE-e4.3.1-updatesite.zip）：

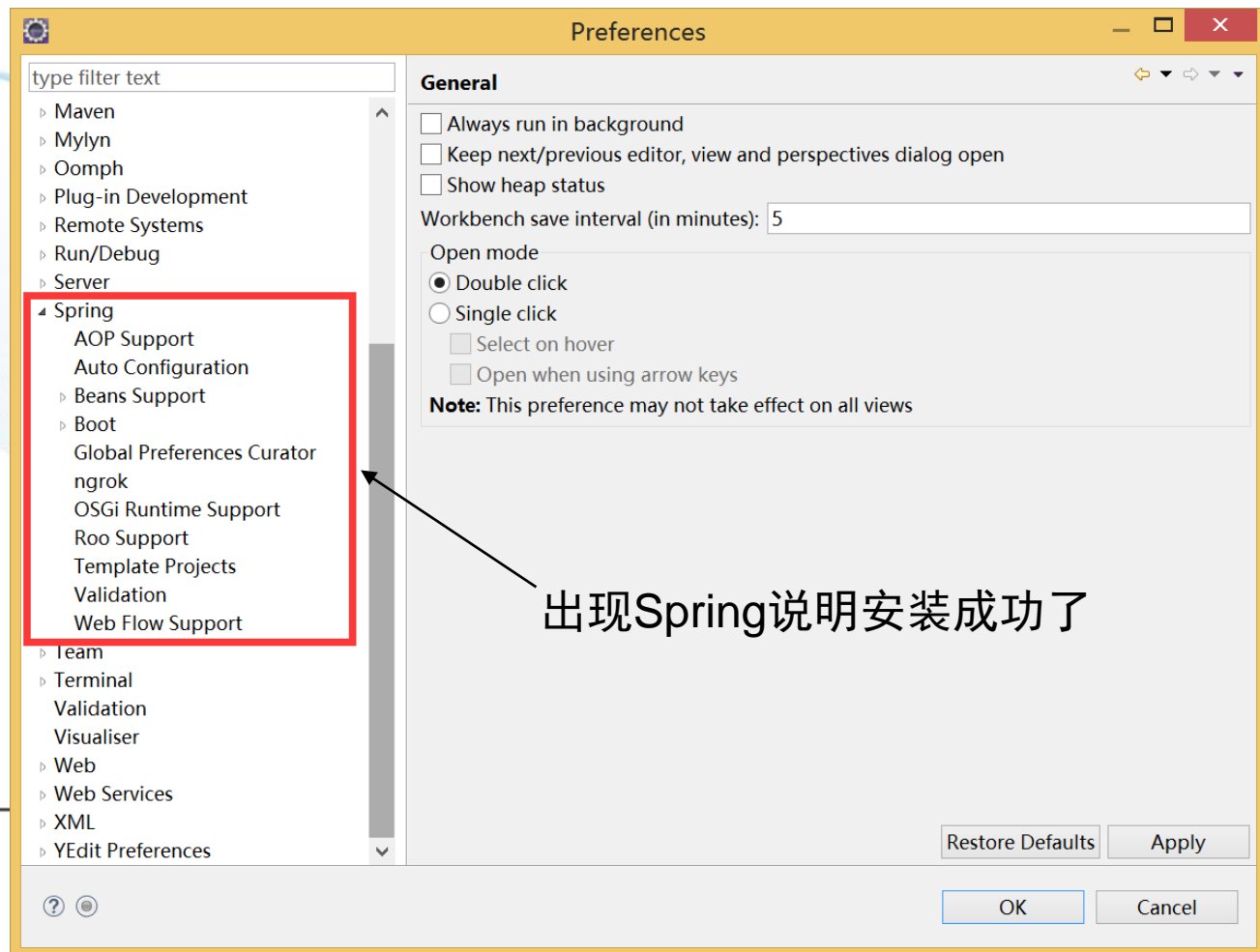
+ 检测是否安装成功



# 安装 SPRING TOOL SUITE






- 安装方法说明（springsource-tool-suite-3.7.0.RELEASE-e4.3.1-updatesite.zip）：

+ 检测是否安装成功



# 搭建 Spring 开发环境

- 把以下 jar 包加入到工程的 classpath 下：

-  commons-logging-1.2.jar
-  spring-beans-4.2.4.RELEASE.jar
-  spring-context-4.2.4.RELEASE.jar
-  spring-core-4.2.4.RELEASE.jar
-  spring-expression-4.2.4.RELEASE.jar

注：

1. commons-logging 的 jar 包需单独下载
2. Spring 的四个 Jar 包在 lib 目录下

- Spring 的配置文件：一个典型的 Spring 项目需要创建一个或多个 Bean 配置文件，这些配置文件用于在 Spring IOC 容器里配置 Bean. Bean 的配置文件可以放在 classpath 下，也可以放在其它目录下.

# 建立 Spring 项目

```
package nuc.sw.spring.beans;  
public class HelloWorld {
```

HelloWorld.java

```
    public HelloWorld() {  
        System.out.println("HelloWorld's constructor...");  
    }
```

```
    private String name;
```

```
    public void setName(String name) {  
        System.out.println("set name = " + name);  
        this.name = name;  
    }
```

```
    public void hello(){  
        System.out.println("Hello:"+name);  
    }  
}
```

<!-- 通过全类名方式配置Bean -->

```
<bean id="helloWorld" class="nuc.sw.spring.beans.HelloWorld">  
    <property name="name" value="中北大学软件学院"/>  
</bean>
```

applicationContext.xml

# 建立 Spring 项目

```
public static void main(String[] args) {  
  
    //1. 创建 Spring 的 IOC 容器  
    ApplicationContext ctx =  
        new ClassPathXmlApplicationContext("applicationContext.xml");  
  
    //2. 从容器中获取 Bean  
    HelloWorld helloWorld = (HelloWorld) ctx.getBean("helloWorld");  
    System.out.println(helloWorld);  
  
    //3. 调用方法  
    helloWorld.hello();  
}
```





# Spring 中的 Bean 配置

# 内容提要

## ■ IOC & DI 概述

# IOC 和 DI

- IOC(Inversion of Control): 其思想是**反转资源获取的方向**. 传统的资源查找方式要求组件向容器发起请求查找资源. 作为回应, 容器适时的返回资源. 而应用了 IOC 之后, 则是**容器主动地将资源推送给它所管理的组件, 组件所要做的仅是选择一种合适的方式来接受资源**. 这种行为也被称为查找的被动形式
- DI(Dependency Injection) — IOC 的另一种表述方式: 即**组件以一些预先定义好的方式(例如: setter 方法)接受来自容器的资源注入**. 相对于 IOC 而言, 这种表述更直接

```
class A{
```

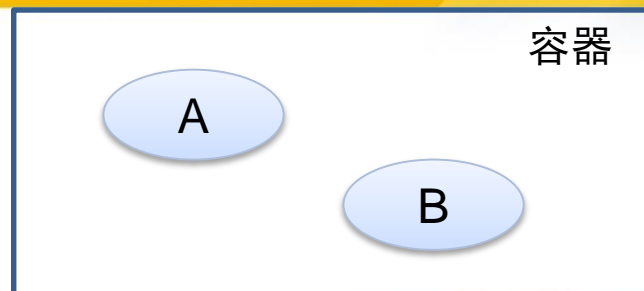
```
class B{
```

```
    private A a;  
    public void setA(A a){  
        this.a = a;  
    }
```

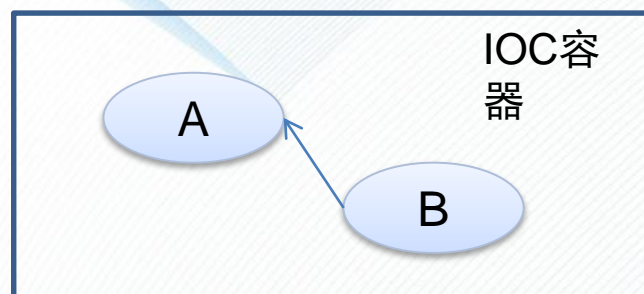
```
}
```



需求：从容器中获取 B 对象，  
并使 B 对象的 a 属性被赋  
值为容器中 A 对象的引用



```
A a = getA();  
B b = getB();  
b.setA(a);
```

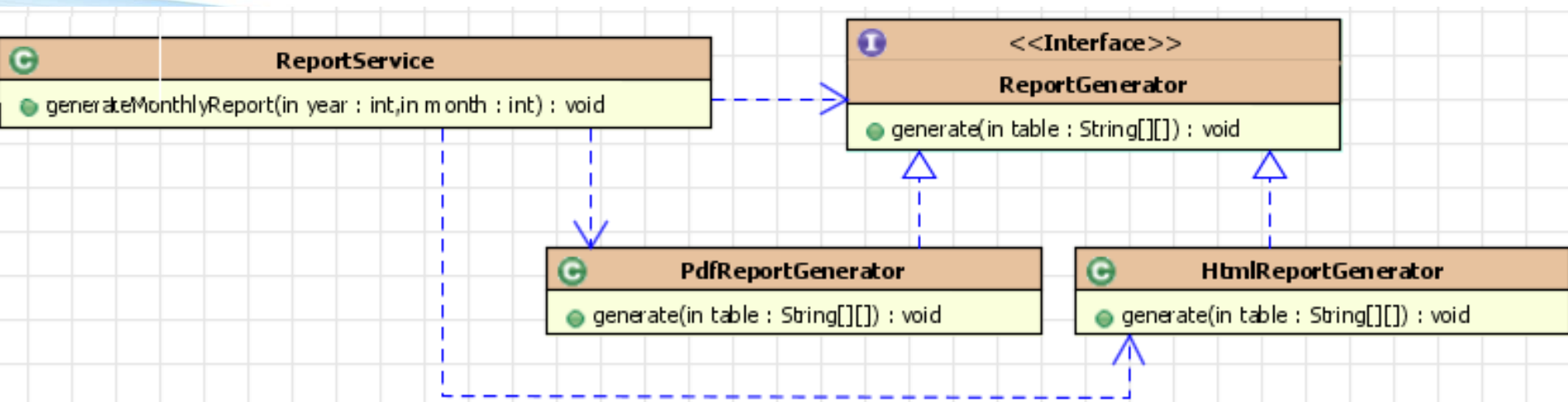


```
B b = getB();
```



# IOC 前身 ---- 分离接口与实现

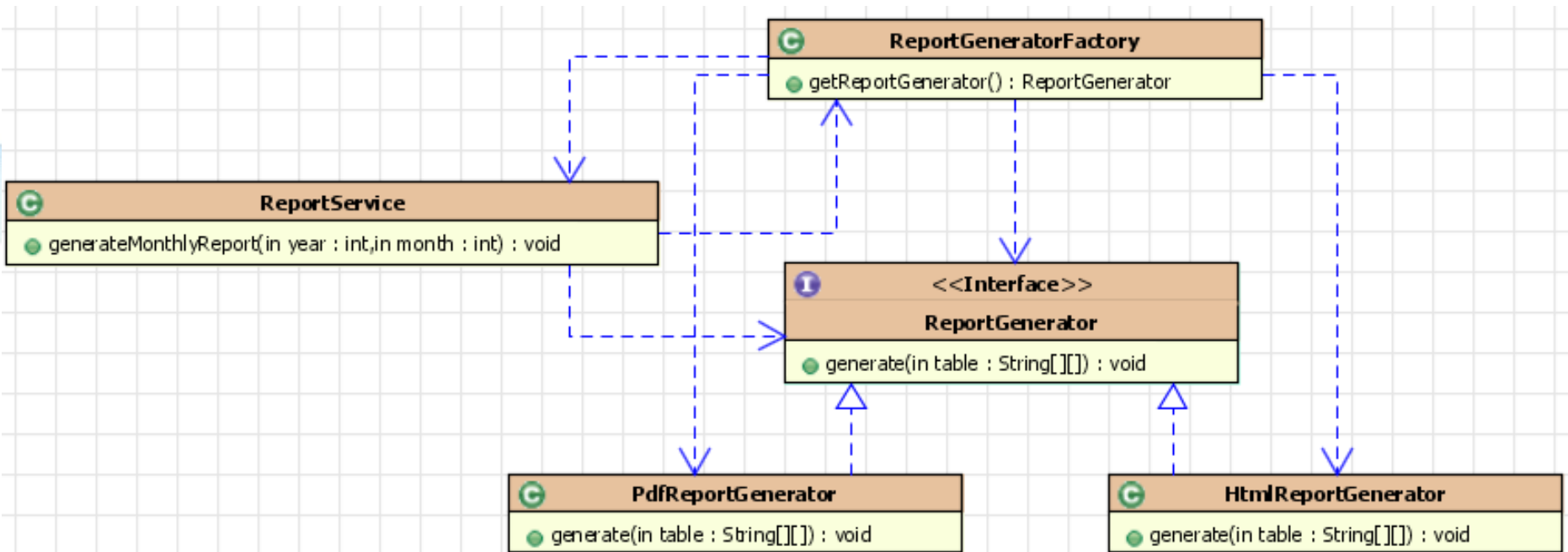
- 需求：生成 HTML 或 PDF 格式的不同类型的报表。



这种方式的耦合度较高，ReportService类不仅要知道要创建的具体类是谁，还要知道创建的细节。

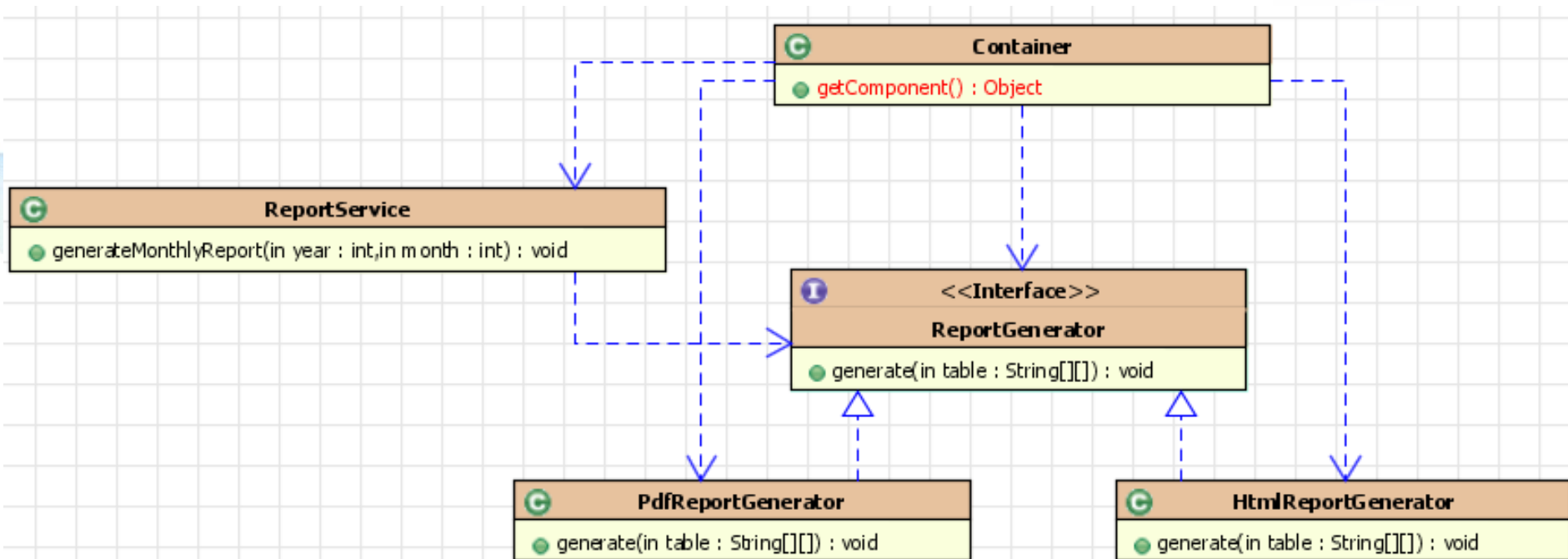


# IOC 前生 --- 采用工厂设计模式



改进：由工厂负责创建具体的实现类，ReportService类无需知道实现类的创建细节以及实现类是谁就能拿到实现类，虽然耦合度一定程度上降低了（此时ReportService类仍需要自己去取得实现类的实例），但是代码的复杂程度却增加了。

# IOC --- 采用反转控制



采用IoC方式： ReportService类无需自己去获取实现类的实例，这个实例由容器负责注入。

# 内容提要

## ■ 配置 bean

- + 配置形式：基于 XML 文件的方式；
- + Bean 的配置方式：通过全类名（反射）、通过工厂方法（静态工厂方法 & 实例工厂方法）、FactoryBean
- + IOC 容器 BeanFactory & ApplicationContext 概述
- + 依赖注入的方式：属性注入；构造器注入
- + 注入属性值细节

# 在 Spring 的 IOC 容器里配置 Bean

- 在 xml 文件中通过 bean 节点来配置 bean

<!-- 通过全类名方式配置Bean -->

```
<bean id="helloWorld" class="nuc.sw.spring.beans.HelloWorld">  
    <property name="name" value="中北大学软件学院"/>  
</bean>
```

- id: Bean 的名称。

- + 在 IOC 容器中必须是唯一的
- + 若 id 没有指定, Spring 自动将权限定性类名作为 Bean 的名字
- + id 可以指定多个名字, 名字之间可用逗号、分号、或空格分隔

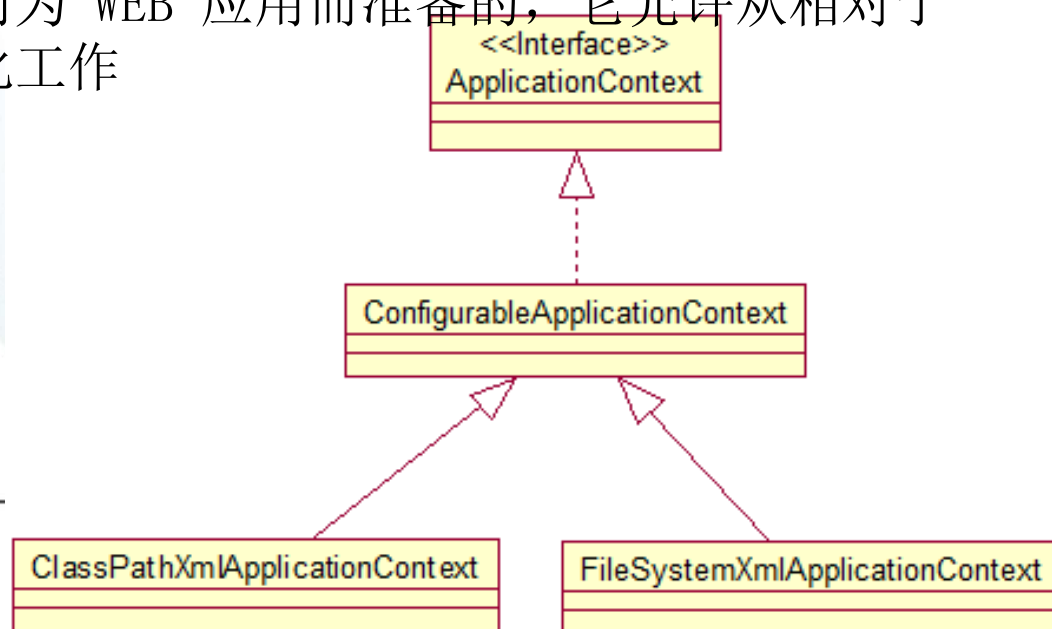


# Spring 容器

- 在 **Spring IOC 容器** 读取 Bean 配置创建 Bean 实例之前，必须对它进行实例化。只有在容器实例化后，才可以从 IOC 容器里获取 Bean 实例并使用。
- Spring 提供了两种类型的 IOC 容器实现。
  - + **BeanFactory**: IOC 容器的基本实现。
  - + **ApplicationContext**: 提供了更多的高级特性。是 BeanFactory 的子接口。
  - + BeanFactory 是 Spring 框架的基础设施，面向 Spring 本身；ApplicationContext 面向使用 Spring 框架的开发者，**几乎所有的应用场合都直接使用 ApplicationContext 而非底层的 BeanFactory**
  - + 无论使用何种方式，配置文件时相同的。

# ApplicationContext

- ApplicationContext 的主要实现类：
  - + **ClassPathXmlApplicationContext**: 从 **类路径下**加载配置文件
  - + **FileSystemXmlApplicationContext**: 从文件系统中加载配置文件
- ConfigurableApplicationContext 扩展于 ApplicationContext, 新增加两个主要方法: refresh() 和 **close()**, 让 ApplicationContext 具有启动、刷新和关闭上下文的能力
- **ApplicationContext** 在初始化上下文时就实例化所有单例的 Bean。
- WebApplicationContext 是专门为 WEB 应用而准备的, 它允许从相对于 WEB 根目录的路径中完成初始化工作





# 从 IOC 容器中获取 Bean

## ■ 调用 `ApplicationContext` 的 `getBean()` 方法

### ▲ `BeanFactory`

- `FACTORY_BEAN_PREFIX` : `String`
- `getBean(String)` : `Object`
- `getBean(String, Class<T>)` `<T> : T`
- `getBean(Class<T>)` `<T> : T`
- `getBean(String, Object...)` : `Object`
- `getBean(Class<T>, Object...)` `<T> : T`
- `containsBean(String)` : `boolean`
- `isSingleton(String)` : `boolean`
- `isPrototype(String)` : `boolean`
- `isTypeMatch(String, ResolvableType)` : `boolean`
- `isTypeMatch(String, Class<?>)` : `boolean`
- `getType(String)` : `Class<?>`
- `getAliases(String)` : `String[]`

如果只指定要返回的Bean的类型就想从IoC容器中取得Bean的前提是该类型的Bean在IoC容器中只有一个。

# 依赖注入的方式

■ Spring 支持 3 种依赖注入的方式

+ 属性注入

+ 构造器注入

+ 工厂方法注入（很少使用，不推荐）

# 属性注入

- 属性注入即通过 **setter 方法**注入Bean 的属性值或依赖的对象
- 属性注入使用 `<property>` 元素, 使用 `name` 属性指定 Bean 的属性名称, `value` 属性或 `<value>` 子节点指定属性值
- 属性注入是实际应用中最常用的注入方式

```
<!-- 通过全类名方式配置Bean -->  
<bean id="helloWorld" class="nuc.sw.spring.beans.HelloWorld">  
    <property name="name" value="中北大学软件学院"/>  
</bean>
```

# 构造方法注入

- 通过构造方法注入Bean 的属性值或依赖的对象，它保证了Bean 实例在实例化后就可以使用。
- 构造器注入在 `<constructor-arg>` 元素里声明属性，`<constructor-arg>` 中没有 `name` 属性

`<!-- 通过构造方法来配置bean的属性 -->`

```
<bean id="car" class="nuc.sw.spring.beans.Car">  
    <constructor-arg value="Audi"/>  
    <constructor-arg value="shanghai"/>  
    <constructor-arg value="300000"/>  
</bean>
```

```
public Car(String brand, String corp, double price) {  
    super();  
    this.brand = brand;  
    this.corp = corp;  
    this.price = price;  
}
```



# 构造方法注入

## ■ 按索引匹配入参:

```
public Car(String brand, String corp, double price) {  
    super();  
    this.brand = brand;  
    this.corp = corp;  
    this.price = price;  
}
```

<!-- 按索引匹配入参 -->

```
<bean id="car" class="nuc.sw.spring.beans.Car">  
    <constructor-arg value="Audi" index="0"/>  
    <constructor-arg value="shanghai" index="1"/>  
    <constructor-arg value="300000" index="2"/>  
</bean>
```

Car [brand=Audi, corp=shanghai, price=300000, maxSpeed=0]

# 构造方法注入

## ■ 按类型匹配入参:

```
public Car(String brand, String corp, int maxSpeed) {  
    super();  
    this.brand = brand;  
    this.corp = corp;  
    this.maxSpeed = maxSpeed;  
}
```

<!-- 按类型匹配入参 -->

```
<bean id="car" class="nuc.sw.spring.beans.Car">  
    <constructor-arg value="Audi" type="java.lang.String"/>  
    <constructor-arg value="shanghai" type="java.lang.String"/>  
    <constructor-arg value="240" type="int"/>  
</bean>
```

Car [brand=Audi, corp=shanghai, price=0.0, maxSpeed=240]



# 字面值

- 字面值：可用字符串表示的值，可以通过 `<value>` 元素标签或 `value` 属性进行注入。
- 基本数据类型及其封装类、`String` 等类型都可以采取字面值注入的方式
- 若字面值中包含特殊字符，可以使用 `<![CDATA[]]>` 把字面值包裹起来。

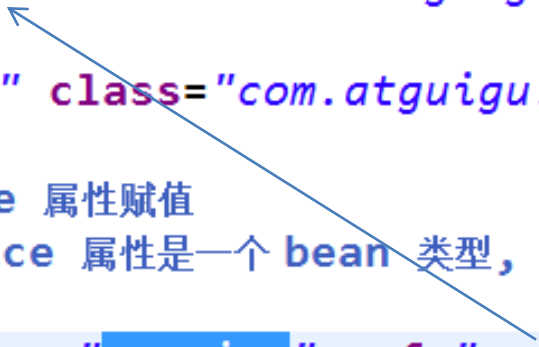
```
<bean id="car2" class="cn.edu.nuc.spring.beans.Car">
    <constructor-arg value="Baoma" type="java.lang.String"/>
    <constructor-arg type="java.lang.String">
        <!-- 如果包含特殊字符要使用 <![CDATA[字面值]]>将字面值包裹起来-->
        <value><![CDATA[shanghai]]></value>
    </constructor-arg>
    <constructor-arg type="int">
        <value>250</value>
    </constructor-arg>
</bean>
```

# 引用其它 Bean

- 组成应用程序的 Bean 经常需要相互协作以完成应用程序的功能. 要使 Bean 能够相互访问, 就必须在 Bean 配置文件中指定对 Bean 的引用
- 在 Bean 的配置文件中, 可以通过 `<ref>` 元素或 `ref` 属性为 Bean 的属性或构造器参数指定对 Bean 的引用.
- 也可以在属性或构造器里包含 Bean 的声明, 这样的 Bean 称为内部 Bean

```
<bean id="service" class="com.atguigu.spring.ioc.ref.Service"></bean>

<bean id="action" class="com.atguigu.spring.ioc.ref.Action">
  <!--
    为 service 属性赋值
    因为 service 属性是一个 bean 类型, 可以使用 ref 指向 ioc 容器中的其他的 bean
  -->
  <property name="service" ref="service"></property>
</bean>
```

A blue arrow originates from the text 'ref="service"' in the `<property>` tag of the 'action' bean and points to the 'service' attribute in the 'service' bean definition above it.

# 内部 Bean

- 当 Bean 实例**仅仅**给一个特定的属性使用时，可以将其声明为内部 Bean. 内部 Bean 声明直接包含在 <property> 或 <constructor-arg> 元素里，不需要设置任何 id 或 name 属性
- 内部 Bean 不能使用在任何其他地方

<!-- 内部bean，只在内部使用，不能在外部被引用 -->

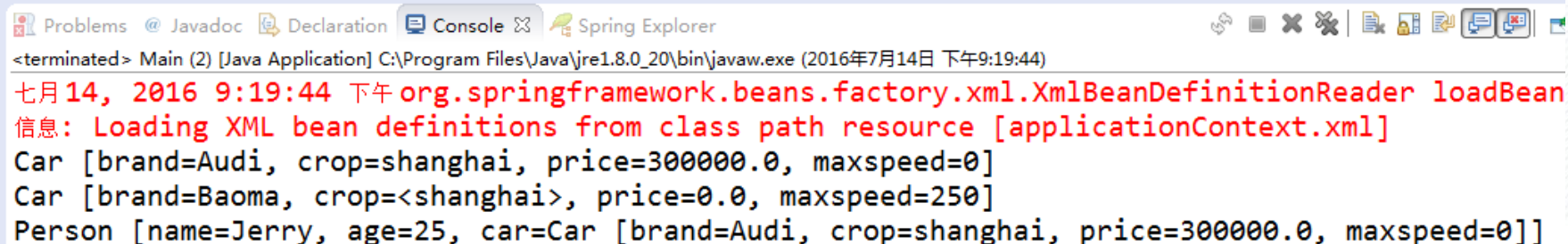
```
<property name="car">
    <bean id="car" class="cn.edu.nuc.spring.beans.Car">
        <constructor-arg value="Ford"/>
        <constructor-arg value="Changan"/>
        <constructor-arg value="200000" type="double"/>
    </bean>
</property>
```

```
Car [brand=Audi, crop=shanghai, price=300000.0, maxspeed=0]
Car [brand=Baoma, crop=<shanghai>, price=0.0, maxspeed=250]
Person [name=Tom, age=24, car=Car [brand=Ford, crop=Changan, price=200000.0, maxspeed=0]]
```

# 内部 Bean

- 还有另外一种方式，在构造器中添加也是可以的

```
<bean id="person2" class="cn.edu.nuc.spring.beans.Person">  
    <constructor-arg value="Jerry"/>  
    <constructor-arg value="25"/>  
    <constructor-arg ref="car"/>  
</bean>
```



The screenshot shows a Java IDE with a console window. The console output displays the following information:

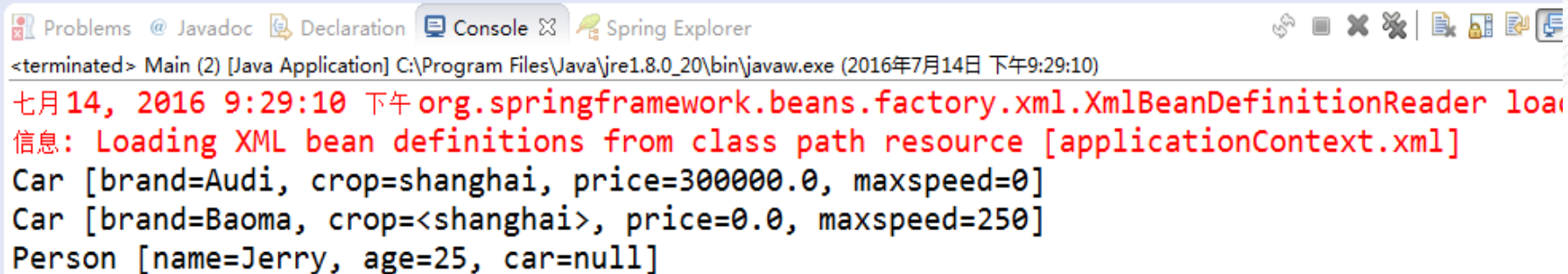
```
<terminated> Main (2) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年7月14日 下午9:19:44)  
七月 14, 2016 9:19:44 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBean  
信息: Loading XML bean definitions from class path resource [applicationContext.xml]  
Car [brand=Audi, crop=shanghai, price=300000.0, maxspeed=0]  
Car [brand=Baoma, crop=<shanghai>, price=0.0, maxspeed=250]  
Person [name=Jerry, age=25, car=Car [brand=Audi, crop=shanghai, price=300000.0, maxspeed=0]]
```



# 注入参数详解：null 值

- 可以使用专用的 `<null/>` 元素标签为 Bean 的字符串或其它对象类型的属性注入 null 值

```
<bean id="person2" class="cn.edu.nuc.spring.beans.Person">
    <constructor-arg value="Jerry"/>
    <constructor-arg value="25"/>
    <!-- null值测试 -->
    <constructor-arg><null/></constructor-arg>
</bean>
```



The screenshot shows an IDE console window with the following content:

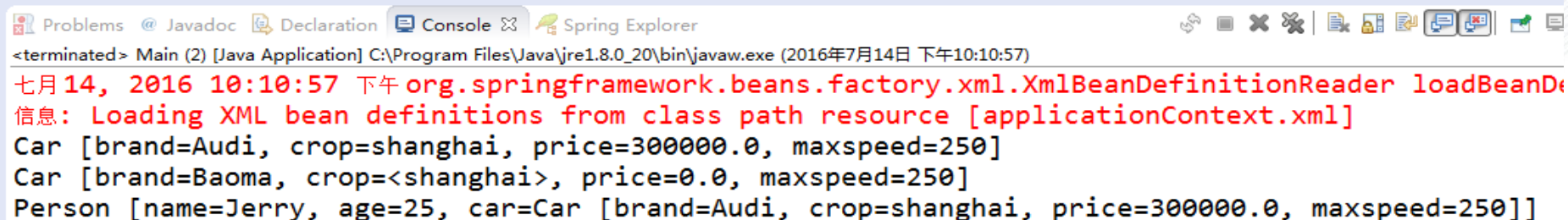
```
Problems @ Javadoc Declaration Console Spring Explorer
<terminated> Main (2) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年7月14日 下午9:29:10)
七月14, 2016 9:29:10 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader load
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
Car [brand=Audi, crop=shanghai, price=300000.0, maxspeed=0]
Car [brand=Baoma, crop=<shanghai>, price=0.0, maxspeed=250]
Person [name=Jerry, age=25, car=null]
```



# 注入参数详解：级联属性

- 和 Struts、Hiberante 等框架一样，Spring 支持级联属性的配置。

```
<bean id="person2" class="cn.edu.nuc.spring.beans.Person">
    <constructor-arg value="Jerry"/>
    <constructor-arg value="25"/>
    <constructor-arg ref="car"></constructor-arg>
    <property name="car.maxspeed" value="250"></property>
</bean>
```



The screenshot shows an IDE console window with the following content:

```
Problems @ Javadoc Declaration Console Spring Explorer
<terminated> Main (2) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年7月14日 下午10:10:57)
七月 14, 2016 10:10:57 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
Car [brand=Audi, crop=shanghai, price=300000.0, maxspeed=250]
Car [brand=Baoma, crop=<shanghai>, price=0.0, maxspeed=250]
Person [name=Jerry, age=25, car=Car [brand=Audi, crop=shanghai, price=300000.0, maxspeed=250]]
```

# 注入参数详解：级联属性

- 要注意，Spring和Struts2不同之处是，为级联属性赋值之前必须先初始化对象，否则会出现异常。而Struts 2不需要，Struts 2会自动创建对象并完成赋值操作。

```
<!--  
<property name="car">  
    <bean id="car" class="cn.edu.nuc.spring.beans.Car">  
        <constructor-arg value="Ford"/>  
        <constructor-arg value="Changan"/>  
        <constructor-arg value="200000" type="double"/>  
    </bean>  
</property>  
-->
```

```
<property name="car.maxspeed" value="260"/>
```



The screenshot shows an IDE window with a console tab. The console output indicates a runtime error: `NullPointerException: Invalid property 'car' of bean class [cn.edu.nuc.spring.beans.Person]:`. The error stack trace includes the following lines:

- `.AbstractNestablePropertyAccessor.getNestedPropertyAccessor(AbstractNestablePropertyAccessor.java:842)`
- `.AbstractNestablePropertyAccessor.getPropertyAccessorForPropertyPath(AbstractNestablePropertyAccessor.java:842)`
- `.AbstractNestablePropertyAccessor.setPropertyValue(AbstractNestablePropertyAccessor.java:270)`
- `.AbstractPropertyAccessor.setPropertyValues(AbstractPropertyAccessor.java:95)`
- `.AbstractPropertyAccessor.setPropertyValues(AbstractPropertyAccessor.java:75)`
- `.factory.support.AbstractAutowireCapableBeanFactory.applyPropertyValues(AbstractAutowireCapableBeanFactory.java:106)`

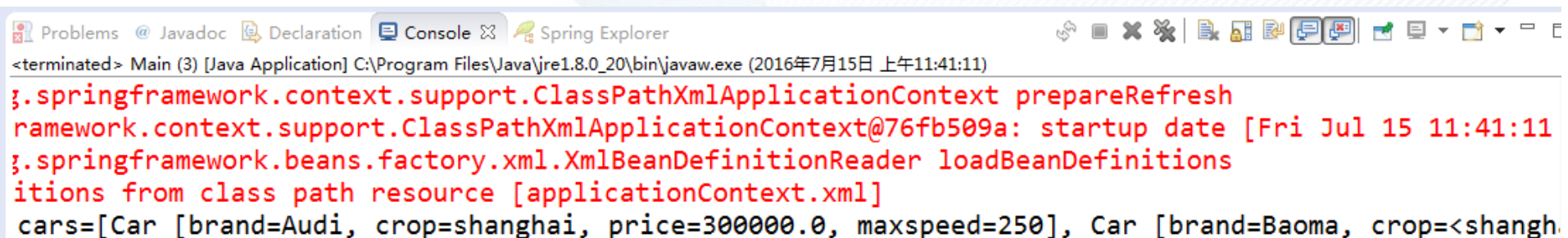
# 集合属性 - 数组、List、Set

- 在 Spring 中可以通过一组内置的 xml 标签 (例如: `<list>`, `<set>` 或 `<map>`) 来配置集合属性.
- 配置 `java.util.List` 类型的属性, 需要指定 `<list>` 标签, 在标签里包含一些元素. 这些标签可以通过 `<value>` 指定简单的常量值, 通过 `<ref>` 指定对其他 Bean 的引用. 通过 `<bean>` 指定内置 Bean 定义. 通过 `<null/>` 指定空元素. 甚至可以内嵌其他集合.
- 数组的定义和 List 一样, 都使用 `<list>`
- 配置 `java.util.Set` 需要使用 `<set>` 标签, 定义元素的方法与 List 一样.



# 集合属性 - List集合范例

```
<!-- 测试配置集合属性 -->
<bean id="person3"
class="cn.edu.nuc.spring.beans.collections.Person">
    <property name="name" value="Mike"/>
    <property name="age" value="27"/>
    <property name="cars">
        <list>
            <ref bean="car"/>
            <ref bean="car2"/>
        </list>
    </property>
</bean>
```

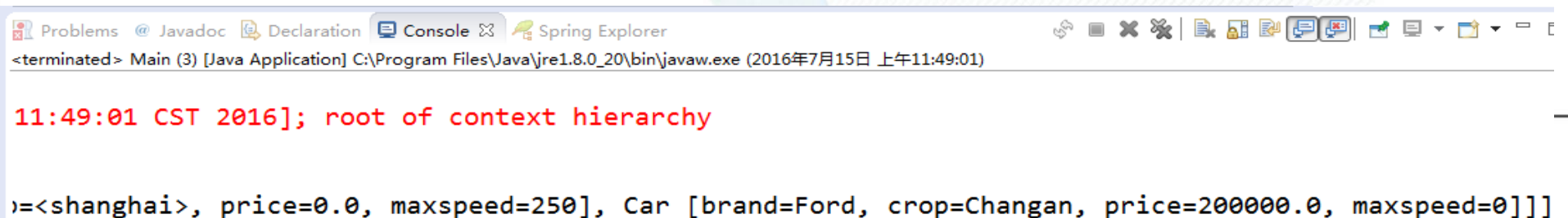


The screenshot shows an IDE interface with a console window. The console displays the following log messages:

```
<terminated> Main (3) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年7月15日 上午11:41:11)
org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
org.springframework.context.support.ClassPathXmlApplicationContext@76fb509a: startup date [Fri Jul 15 11:41:11
org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
Definitions from class path resource [applicationContext.xml]
cars=[Car [brand=Audi, crop=shanghai, price=300000.0, maxspeed=250], Car [brand=Baoma, crop=<shangh
```

# 集合属性 - List集合

```
<!-- 直接添加内部bean-->
<bean id="person3" class="cn.edu.nuc.spring.beans.collections.Person">
    <property name="name" value="Mike"/>
    <property name="age" value="27"/>
    <property name="cars">
        <list>
            <ref bean="car"/>
            <ref bean="car2"/>
            <bean id="car" class="cn.edu.nuc.spring.beans.Car">
                <constructor-arg value="Ford"/>
                <constructor-arg value="Changan"/>
                <constructor-arg value="200000" type="double"/>
            </bean>
        </list>
    </property>
</bean>
```



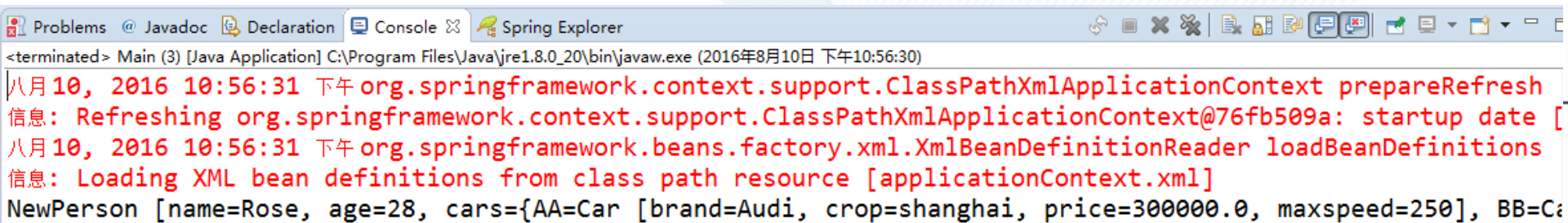


# 集合属性 - Map集合

- `java.util.Map` 通过 `<map>` 标签定义, `<map>` 标签里可以使用多个 `<entry>` 作为子标签. 每个条目包含一个键和一个值.
- 必须在 `<key>` 标签里定义键
- 因为键和值的类型没有限制, 所以可以自由地为它们指定 `<value>`, `<ref>`, `<bean>` 或 `<null>` 元素.
- 可以将 `Map` 的键和值作为 `<entry>` 的属性定义: 简单常量使用 `key` 和 `value` 来定义; `Bean` 引用通过 `key-ref` 和 `value-ref` 属性定义

# 集合属性—Map集合范例

```
<!-- 测试配置Map集合属性 -->
<bean id="newPerson"
class="cn.edu.nuc.spring.beans.collections.NewPerson">
    <property name="name" value="Rose"/>
    <property name="age" value="28"/>
    <property name="cars">
        <!-- 使用map节点和map的entry子节点
            配置Map类型的成员变量 -->
        <map>
            <entry key="AA" value-ref="car"/>
            <entry key="BB" value-ref="car2"/>
        </map>
    </property>
</bean>
```



# 集合属性 - Properties集合

- 使用 `<props>` 定义 `java.util.Properties`, 该标签使用多个 `<prop>` 作为子标签. 每个 `<prop>` 标签必须定义 `key` 属性.

```
compact1, compact2, compact3  
java.util
```

## Class Properties

```
java.lang.Object  
    java.util.Dictionary<K,V>  
        java.util.Hashtable<Object,Object>  
            java.util.Properties
```

### All Implemented Interfaces:

```
Serializable, Cloneable, Map<Object,Object>
```

### Direct Known Subclasses:

```
Provider
```

```
compact1, compact2, compact3  
java.util
```

## Class Hashtable<K,V>

```
java.lang.Object  
    java.util.Dictionary<K,V>  
        java.util.Hashtable<K,V>
```

### All Implemented Interfaces:

```
Serializable, Cloneable, Map<K,V>
```

### Direct Known Subclasses:

```
Properties, UIDefaults
```

# 集合属性 - Properties集合范例

```
<!-- 配置Properties属性值 -->
```

```
<bean id="dataSource"
```

```
    class="cn.edu.nuc.spring.beans.collections.DataSource">
```

```
    <property name="properties">
```

```
        <props>
```

```
            <prop key="dbuser">root</prop>
```

```
            <prop key="dbpass">123456</prop>
```

```
            <prop key="dburl">
```

```
                jdbc:mysql://localhost:3306/test
```

```
            </prop>
```

```
            <prop key="dbdriver">
```

```
                org.gjt.mm.mysql.Driver
```

```
            </prop>
```

```
        </props>
```

```
    </property>
```

```
</bean>
```

cn.edu.nuc.spring.beans.collections

DataSource

properties : Properties

getProperties() : Properties

setProperties(Properties) : void

toString() : String

Problems Javadoc Declaration Console Spring Explorer

<terminated> Main (3) [Java Application] C:\Program Files\Java\jre1.8.0\_20\bin\javaw.exe (2016年8月11日 上午12:28:18)

八月 11, 2016 12:28:18 上午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh  
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@76fb509a: startup date  
八月 11, 2016 12:28:18 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions  
信息: Loading XML bean definitions from class path resource [applicationContext.xml]  
{dburl=jdbc:mysql://localhost:3306/test, dbpass=123456, dbdriver=org.gjt.mm.mysql.Driver, dbuser=root}



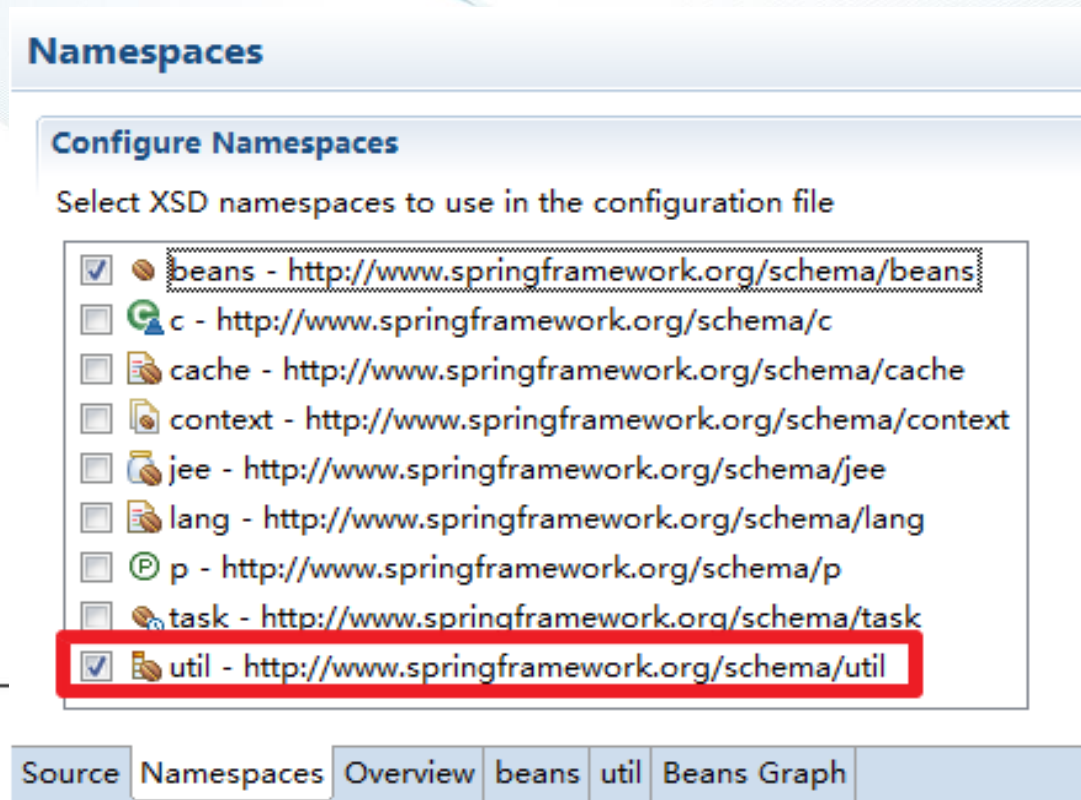
# 使用 `utility` `schema` 定义集合

- 使用基本的集合标签定义集合时，**不能将集合作为独立的 Bean 定义，导致其他 Bean 无法引用该集合，所以无法在不同 Bean 之间共享集合。**
- 可以使用 `util` `schema` 里的集合标签定义独立的集合 Bean. 需要注意的是，必须在 `<beans>` 根元素里添加 `util` `schema` 定义



# 使用 utility scheme 定义集合

- 如果想用util schema来定义集合，需要先导入util命名空间，具体方法是在配置文件编辑窗口找到Namespaces选项卡，然后加选util命令空间（打上勾表示选中）就可以了

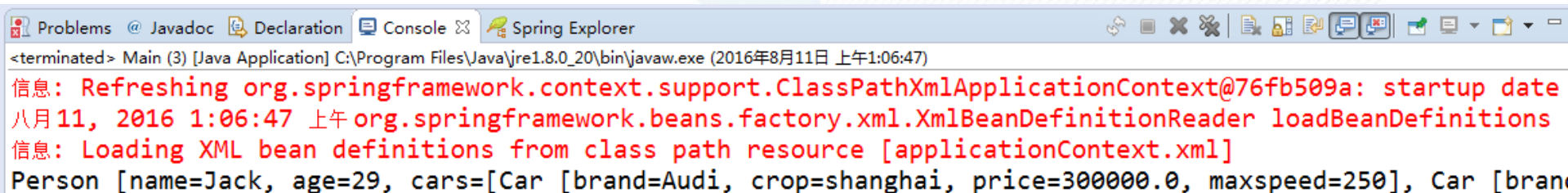


# 使用 utility scheme 定义集合

<!-- 配置单例集合bean，以供多个bean进行引用，需要导入util命名空间 -->

```
<util:list id="cars">
    <ref bean="car"/>
    <ref bean="car2"/>
</util:list>
```

```
<bean id="person4"
      class="cn.edu.nuc.spring.beans.collections.Person">
    <property name="name" value="Jack"/>
    <property name="age" value="29"/>
    <property name="cars" ref="cars"></property>
</bean>
```



The screenshot shows the bottom portion of an IDE window with several tabs: Problems, Javadoc, Declaration, Console, and Spring Explorer. The Console tab is active, displaying the following text:

```
<terminated> Main (3) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月11日 上午1:06:47)
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@76fb509a: startup date
八月 11, 2016 1:06:47 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
Person [name=Jack, age=29, cars=[Car [brand=Audi, crop=shanghai, price=300000.0, maxspeed=250], Car [brand=
```

# 使用 p 命名空间

- 为了简化 XML 文件的配置，越来越多的 XML 文件采用属性而非子元素配置信息。
- Spring 从 2.5 版本开始引入了一个新的 p 命名空间，可以通过 <bean> 元素属性的方式配置 Bean 的属性。
- 使用 p 命名空间后，基于 XML 的配置方式将进一步简化
- 使用此种方式配置之前需要先导入 p 命名空间。方法与之前导入 util 命名空间类似：

## Namespaces

### Configure Namespaces

Select XSD namespaces to use in the configuration file

- ☒ beans - <http://www.springframework.org/schema/beans>
- ☐ c - <http://www.springframework.org/schema/c>
- ☐ cache - <http://www.springframework.org/schema/cache>
- ☐ context - <http://www.springframework.org/schema/context>
- ☐ jee - <http://www.springframework.org/schema/jee>
- ☐ lang - <http://www.springframework.org/schema/lang>
- ☒ p - <http://www.springframework.org/schema/p>
- ☐ task - <http://www.springframework.org/schema/task>
- ☒ util - <http://www.springframework.org/schema/util>

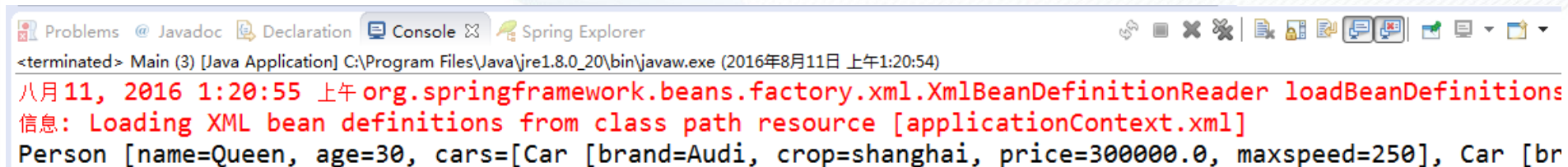


# 使用 p 命名空间范例

<!-- 通过p命名空间为bean的属性赋值，需要先导入p命名空间，  
相对于传统方式更加简洁-->

```
<bean id="person5"  
      class="cn.edu.nuc.spring.beans.collections.Person"  
      p:name="Queen" p:age="30" p:cars-ref="cars"/>
```

```
<util:list id="cars">  
    <ref bean="car"/>  
    <ref bean="car2"/>  
</util:list>
```



# 内容提要

- 配置 bean
  - + 自动装配

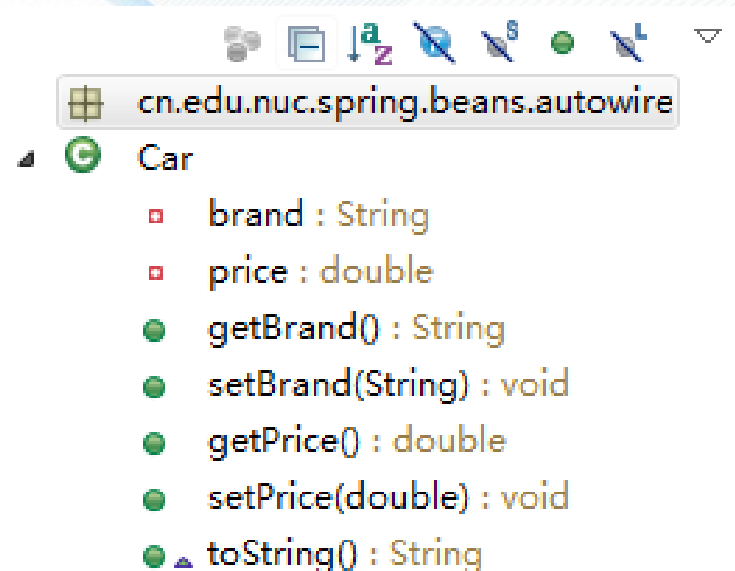
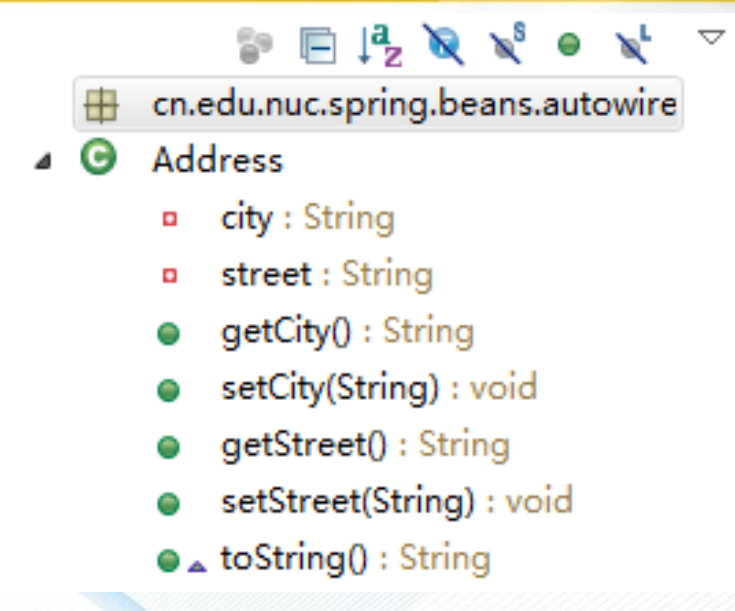
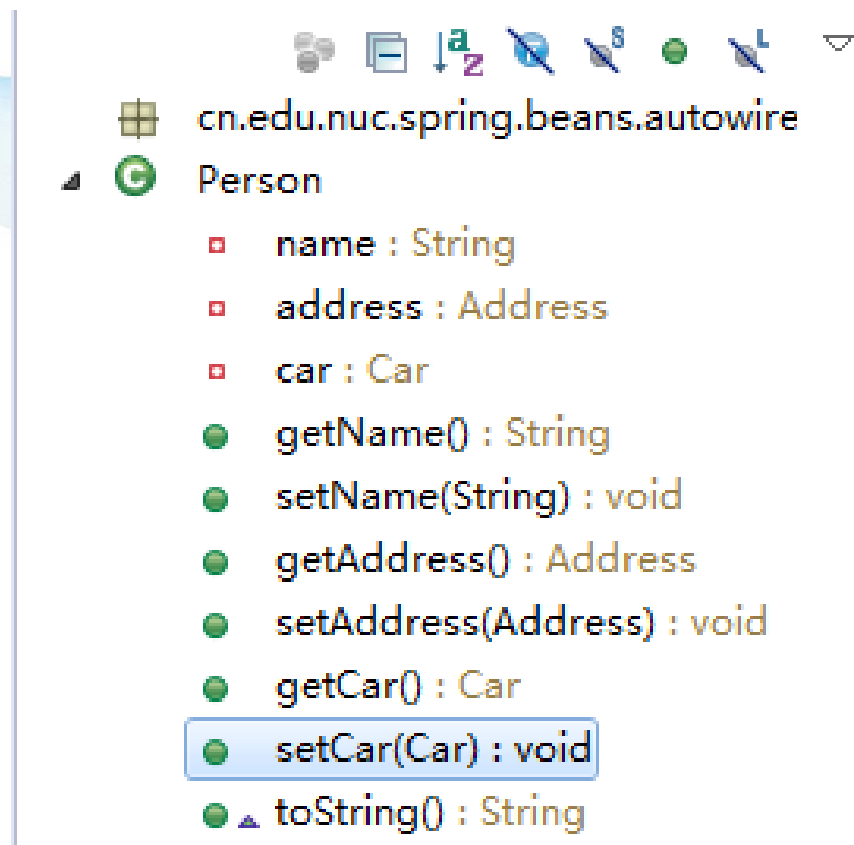


# XML 配置里的 Bean 自动装配

- Spring IOC 容器可以自动装配 Bean. 需要做的仅仅是在 `<bean>` 的 `autowire` 属性里指定自动装配的模式
- `byType` (根据类型自动装配): 若 IOC 容器中有多个与目标 Bean 类型一致的 Bean. 在这种情况下, Spring 将无法判定哪个 Bean 最合适该属性, 所以不能执行自动装配.
- `byName` (根据名称自动装配): 必须将目标 Bean 的名称和属性名设置的完全相同.
- `constructor` (通过构造器自动装配): 当 Bean 中存在多个构造器时, 此种自动装配方式将会很复杂. **不推荐使用**

# XML 配置里的 Bean 自动装配范例

## ■ 新建三个类，类的结构如下：



# XML 配置里的 Bean 自动装配范例

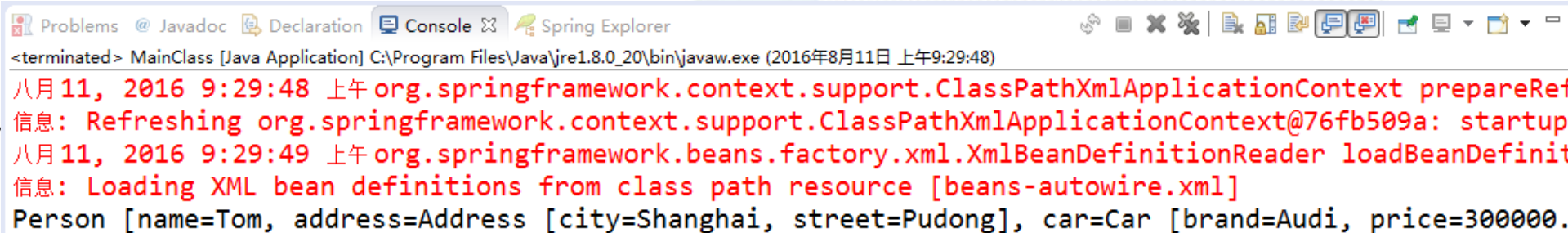
按名称装配bean范例:

```
<bean id="address"  
      class="cn.edu.nuc.spring.beans.autowire.Address"  
      p:city="Shanghai" p:street="Pudong"/>
```

```
<bean id="car"  
      class="cn.edu.nuc.spring.beans.autowire.Car"  
      p:brand="Audi" p:price="300000"/>
```

<!-- 可以使用autowire属性指定自动装配的方式, byname根据bean的名字和当前bean的setter风格属性名进行自动装配 -->

```
<bean id="person"  
      class="cn.edu.nuc.spring.beans.autowire.Person"  
      p:name="Tom" autowire="byName"/>
```



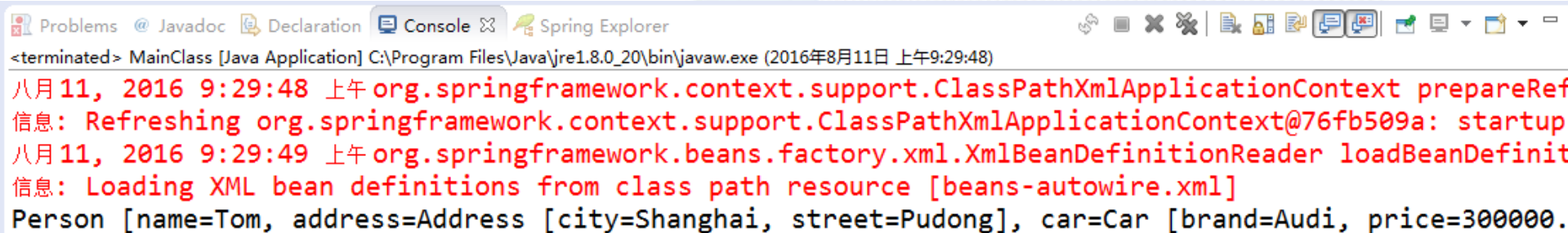
The screenshot shows the bottom portion of an IDE window with several tabs: Problems, Javadoc, Declaration, Console, and Spring Explorer. The Console tab is active, displaying the following log output:

```
<terminated> MainClass [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月11日 上午9:29:48)  
八月 11, 2016 9:29:48 上午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRef  
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@76fb509a: startup  
八月 11, 2016 9:29:49 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinit  
信息: Loading XML bean definitions from class path resource [beans-autowire.xml]  
Person [name=Tom, address=Address [city=Shanghai, street=Pudong], car=Car [brand=Audi, price=300000.
```

# XML 配置里的 Bean 自动装配范例

按类型装配bean范例：

```
<bean id="address1"
      class="cn.edu.nuc.spring.beans.autowire.Address"
      p:city="Shanghai" p:street="Pudong"/>
<bean id="car1"
      class="cn.edu.nuc.spring.beans.autowire.Car"
      p:brand="Audi" p:price="300000"/>
<!-- 可以使用autowire属性指定自动装配的方式，
      byType根据bean的类型和当前bean的属性的类型进行自动装配。-->
<bean id="person"
      class="cn.edu.nuc.spring.beans.autowire.Person"
      p:name="Tom" autowire="byType"/>
```



The screenshot shows the bottom portion of an IDE window with a console tab active. The title bar includes 'Problems', 'Javadoc', 'Declaration', 'Console', and 'Spring Explorer'. The console output shows the following sequence of events:

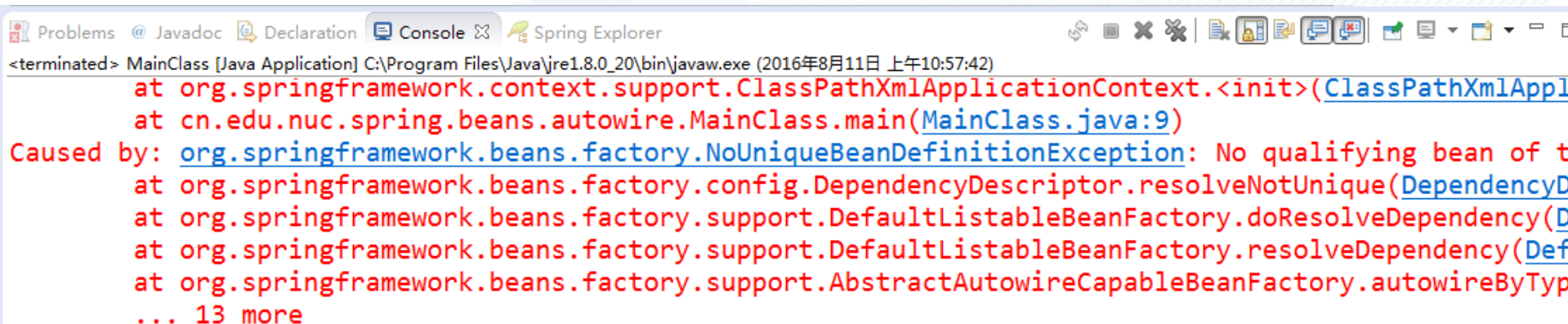
- <terminated> MainClass [Java Application] C:\Program Files\Java\jre1.8.0\_20\bin\javaw.exe (2016年8月11日 上午9:29:48)
- 八月 11, 2016 9:29:48 上午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRef
- 信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@76fb509a: startup
- 八月 11, 2016 9:29:49 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinit
- 信息: Loading XML bean definitions from class path resource [beans-autowire.xml]
- Person [name=Tom, address=Address [city=Shanghai, street=Pudong], car=Car [brand=Audi, price=300000.



# XML 配置里的 Bean 自动装配范例

范例：需要注意，使用按类型装配bean方式配置的时候配置文件中该类型的bean只能有一个，否则就会出现問題

```
<bean id="address"  
      class="cn.edu.nuc.spring.beans.autowire.Address"  
      p:city="Shanghai" p:street="Pudong"/>  
<bean id="address2"  
      class="cn.edu.nuc.spring.beans.autowire.Address"  
      p:city="Dalian" p:street="Zhongshan"/>  
<bean id="person"  
      class="cn.edu.nuc.spring.beans.autowire.Person"  
      p:name="Tom" autowire="byType"/>
```



The screenshot shows an IDE window with a console tab. The console displays the following error message:

```
<terminated> MainClass [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月11日 上午10:57:42)  
at org.springframework.context.support.ClassPathXmlApplicationContext.<init>(ClassPathXmlApp  
at cn.edu.nuc.spring.beans.autowire.MainClass.main(MainClass.java:9)  
Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of t  
at org.springframework.beans.factory.config.DependencyDescriptor.resolveNotUnique(DependencyD  
at org.springframework.beans.factory.support.DefaultListableBeanFactory.doResolveDependency(D  
at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveDependency(Def  
at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.autowireByTyp  
... 13 more
```



# XML 配置里的 Bean 自动装配的缺点

- 在 Bean 配置文件里设置 `autowire` 属性进行自动装配将会装配 Bean 的所有属性。然而，若只希望装配个别属性时，`autowire` 属性就不够灵活了。
- `autowire` 属性要么根据类型自动装配，要么根据名称自动装配，不能两者兼而有之。
- 一般情况下，在实际的项目中很少使用自动装配功能，因为和自动装配功能所带来的好处比起来，明确清晰的配置文档更有说服力一些

# 内容提要

## ■ 配置 bean

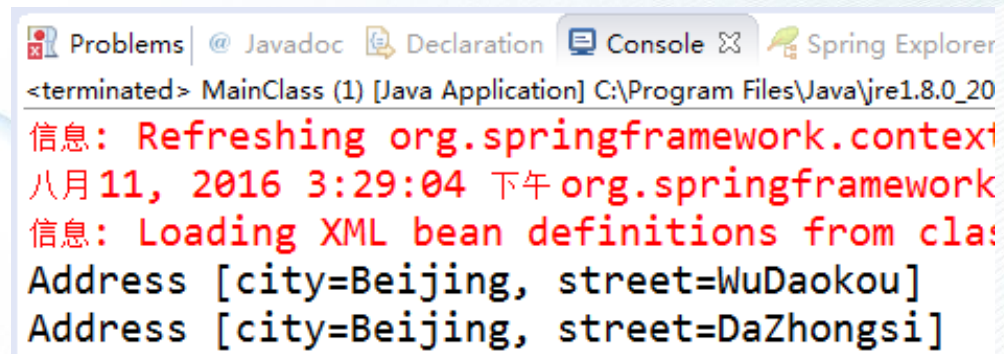
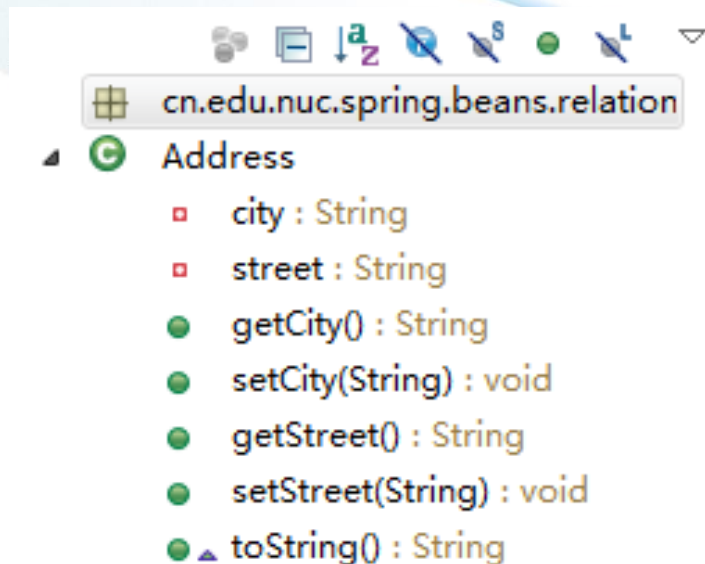
+ bean 之间的关系：继承；依赖

# 继承 Bean 配置

- Spring 允许继承 bean 的配置，被继承的 bean 称为父 bean。继承这个父 Bean 的 Bean 称为子 Bean
- 继承仅仅是指配置上的继承，并不意味着这两个bean之间存在继承关系。继承bean配置仅仅是为了复用其配置信息。
- 子 Bean 从父 Bean 中继承配置，包括 Bean 的属性配置
- 子 Bean 也可以覆盖从父 Bean 继承过来的配置
- 父 Bean 可以作为配置模板，也可以作为 Bean 实例。若只想把父 Bean 作为模板，可以设置 <bean> 的abstract 属性为 true，这样 Spring 将不会实例化这个 Bean
- 并不是 <bean> 元素里的所有属性都会被继承。比如：autowire, abstract 等。
- 也可以忽略父 Bean 的 class 属性，让子 Bean 指定自己的类，而共享相同的属性配置。但此时 abstract 必须设为 true

# 继承 Bean 配置范例 (1)

```
<bean id="address"  
      class="cn.edu.nuc.spring.beans.relation.Address"  
      p:city="Beijing" p:street="WuDaokou"/>  
<bean id="address2" p:street="DaZhongsi" parent="address"/>
```





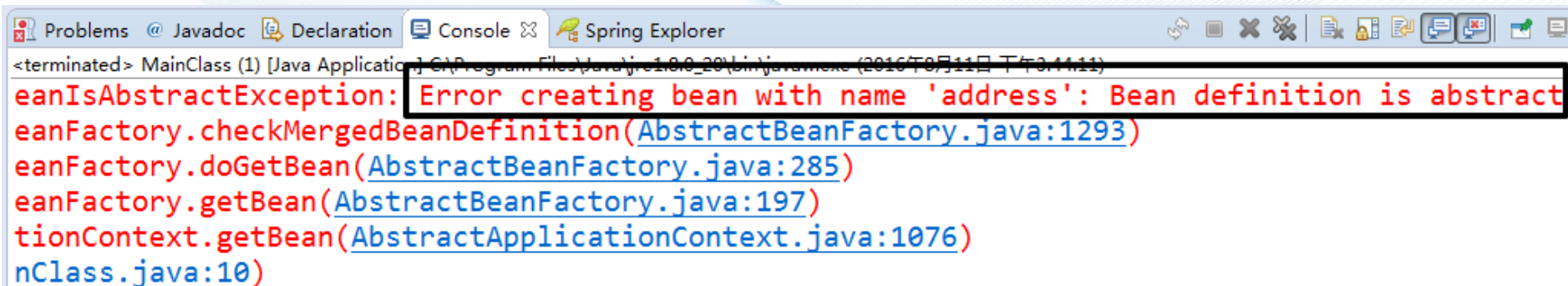
# 继承 Bean 配置范例 (2)

<!-- 抽象bean: bean的abstract属性为true的bean。这样的bean不能被ioc容器实例化, 只能被用来继承配置。若某一个bean的class属性没有指定, 则该bean必须是一个抽象bean -->

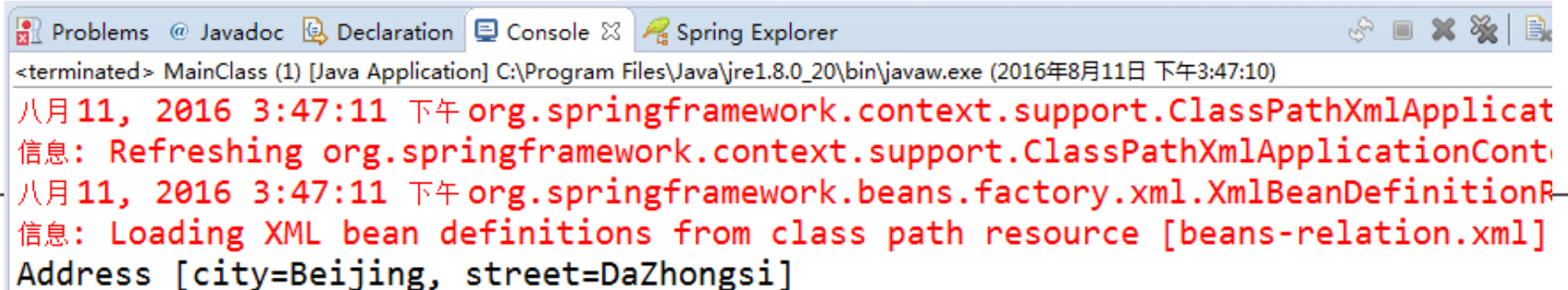
```
<bean id="address" p:city="Beijing" p:street="WuDaokou"
      abstract="true"/>
```

<!-- bean配置的继承: 使用bean的parent属性指定继承那个bean -->

```
<bean id="address2" p:street="DaZhongsi" parent="address"/>
```

A screenshot of an IDE's console window. The title bar shows 'Problems', 'Javadoc', 'Declaration', 'Console', and 'Spring Explorer'. The console text shows a stack trace for a 'BeanDefinitionException' with the message 'Error creating bean with name 'address': Bean definition is abstract'. The stack trace includes 'AbstractBeanFactory.java:1293', 'AbstractBeanFactory.java:285', 'AbstractBeanFactory.java:197', 'AbstractApplicationContext.java:1076', and 'AbstractApplicationContext.java:10'.

```
<terminated> MainClass (1) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月11日 下午3:44:11)
BeanDefinitionException: Error creating bean with name 'address': Bean definition is abstract
    at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:1293)
    at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:285)
    at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:197)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1076)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:10)
```

A screenshot of an IDE's console window. The title bar shows 'Problems', 'Javadoc', 'Declaration', 'Console', and 'Spring Explorer'. The console text shows log messages from the Spring framework, including 'Refreshing org.springframework.context.support.ClassPathXmlApplicationContext', 'Loading XML bean definitions from class path resource [beans-relation.xml]', and 'Address [city=Beijing, street=DaZhongsi]'.

```
<terminated> MainClass (1) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月11日 下午3:47:10)
八月 11, 2016 3:47:11 下午 org.springframework.context.support.ClassPathXmlApplicationContext: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext
八月 11, 2016 3:47:11 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader: Loading XML bean definitions from class path resource [beans-relation.xml]
Address [city=Beijing, street=DaZhongsi]
```



# 依赖 Bean 配置

- Spring 允许用户通过 `depends-on` 属性设定 Bean 前置依赖的Bean，前置依赖的 Bean 会在本 Bean 实例化之前创建好
- 如果前置依赖于多个 Bean，则可以通过逗号，空格或的方式配置 Bean 的名称

# 依赖 Bean 配置范例

```
<bean id="address" p:city="Beijing" p:street="WuDaokou"
      abstract="true"/>
<bean id="address2"
      class="cn.edu.nuc.spring.beans.relation.Address"
      p:street="DaZhongsi" parent="address"/>
<bean id="car" class="cn.edu.nuc.spring.beans.relation.Car"
      p:brand="Audi" p:price="300000"/>
<bean id="person" class="cn.edu.nuc.spring.beans.relation.Person"
      p:name="Tom" p:address-ref="address2" depends-on="car"/>>
```

Problems @ Javadoc Declaration Console Spring Explorer

<terminated> MainClass (1) [Java Application] C:\Program Files\Java\jre1.8.0\_20\bin\javaw.exe (2016年8月11日 下午7:32:35)

八月 11, 2016 7:32:35 下午 org.springframework.context.support.ClassPathXmlApplicationContext p  
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@76fb509a  
八月 11, 2016 7:32:35 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBe  
信息: Loading XML bean definitions from class path resource [beans-relation.xml]  
Person [name=Tom, address=Address [city=Beijing, street=DaZhongsi], car=null]

# 内容提要

## ■ 配置 bean

+ bean 的作用域:

singleton;

prototype;

WEB 环境作用域

# Bean 的作用域

- 在 Spring 中，可以在 <bean> 元素的 **scope** 属性里设置 Bean 的作用域。
- 默认情况下，Spring 只为每个在 IOC 容器里声明的 Bean 创建唯一的一个实例，整个 IOC 容器范围内都能共享该实例：所有后续的 `getBean()` 调用和 Bean 引用都将返回这个唯一的 Bean 实例。该作用域被称为 **singleton**，它是所有 Bean 的默认作用域。

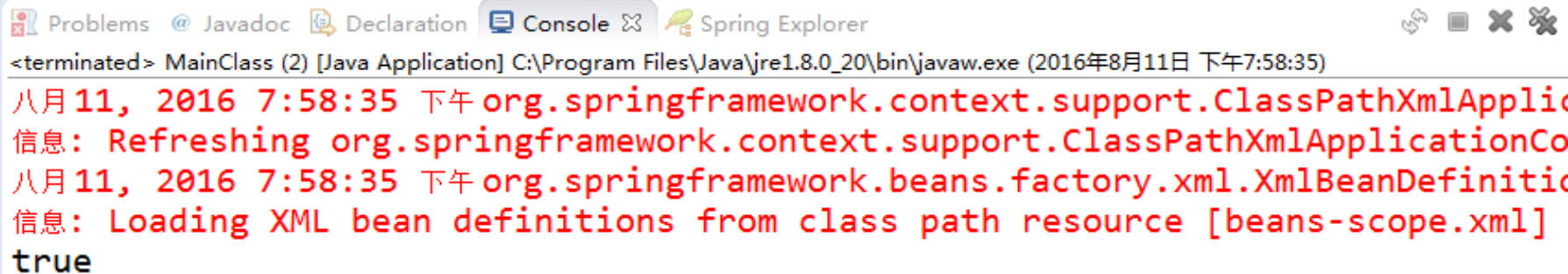
类别	说明
singleton	在 SpringIOC 容器中仅存在一个 Bean 实例，Bean 以单实例的方式存在
prototype	每次调用 <code>getBean()</code> 时都会返回一个新的实例
request	每次 HTTP 请求都会创建一个新的 Bean，该作用域仅适用于 <code>WebApplicationContext</code> 环境
session	同一个 HTTP Session 共享一个 Bean，不同的 HTTP Session 使用不同的 Bean。该作用域仅适用于 <code>WebApplicationContext</code> 环境



# 单例bean范例

```
<bean id="car" class="cn.edu.nuc.spring.beans.scope.Car">
    <property name="brand" value="Audi"/>
    <property name="price" value="300000"/>
</bean>
```

```
public class MainClass {
    public static void main(String[] args) {
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext(
                "beans-scope.xml");
        Car car1 = (Car) ctx.getBean("car");
        Car car2 = (Car) ctx.getBean("car");
        System.out.println(car1 == car2);
    }
}
```



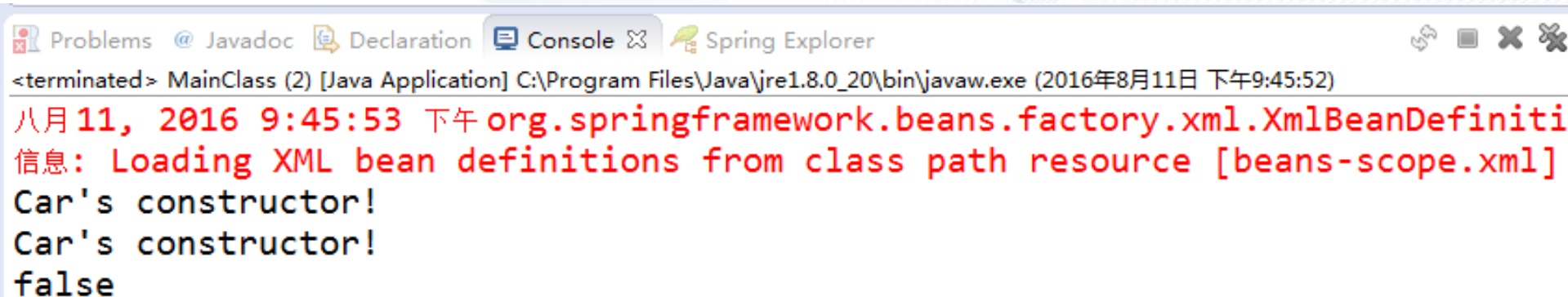
```
<terminated> MainClass (2) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月11日 下午7:58:35)
八月 11, 2016 7:58:35 下午 org.springframework.context.support.ClassPathXmlApplic
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationCo
八月 11, 2016 7:58:35 下午 org.springframework.beans.factory.xml.XmlBeanDefinitio
信息: Loading XML bean definitions from class path resource [beans-scope.xml]
true
```

# 原型bean范例

<!-- 使用bean的scope属性来配置bean的作用域  
singleton:默认值。容器初始化时创建bean实例，整个容器的生命周期内只创建这一个bean。单例的  
prototype:原型的。容器初始化时不创建bean实例，而在每次请求时都创建一个新的bean实例，并返回。

-->

```
<bean id="car" class="cn.edu.nuc.spring.beans.scope.Car"
      scope="prototype">
    <property name="brand" value="Audi"/>
    <property name="price" value="300000"/>
</bean>
```



The screenshot shows an IDE console window with the following tabs: Problems, Javadoc, Declaration, Console, and Spring Explorer. The Console tab is active, displaying the following output:

```
<terminated> MainClass (2) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月11日 下午9:45:52)
八月11, 2016 9:45:53 下午 org.springframework.beans.factory.xml.XmlBeanDefinition
信息: Loading XML bean definitions from class path resource [beans-scope.xml]
Car's constructor!
Car's constructor!
false
```

# 内容提要

## ■ 配置 bean

+ 使用外部属性文件



# 使用外部属性文件

- 在配置文件里配置 Bean 时，有时需要在 Bean 的配置里混入**系统部署的细节信息**（例如：文件路径，数据源配置信息等）。而这些部署细节实际上需要和 Bean 配置相分离
- Spring 提供了一个 PropertyPlaceholderConfigurer 的 **BeanFactory 后置处理器**，这个处理器允许用户将 Bean 配置的部分内容外移到**属性文件**中。可以在 Bean 配置文件里使用形式为 **`${var}`** 的变量，PropertyPlaceholderConfigurer 从属性文件里加载属性，并使用这些属性来替换变量。
- Spring 还允许在属性文件中使用 **`${propName}`**，以实现属性之间的相互引用。



# 注册 PropertyPlaceholderConfigurer

## ■ Spring 2.0:

```
<bean  
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
    <property name="location" value="classpath:jdbc.properties"></property>  
</bean>
```

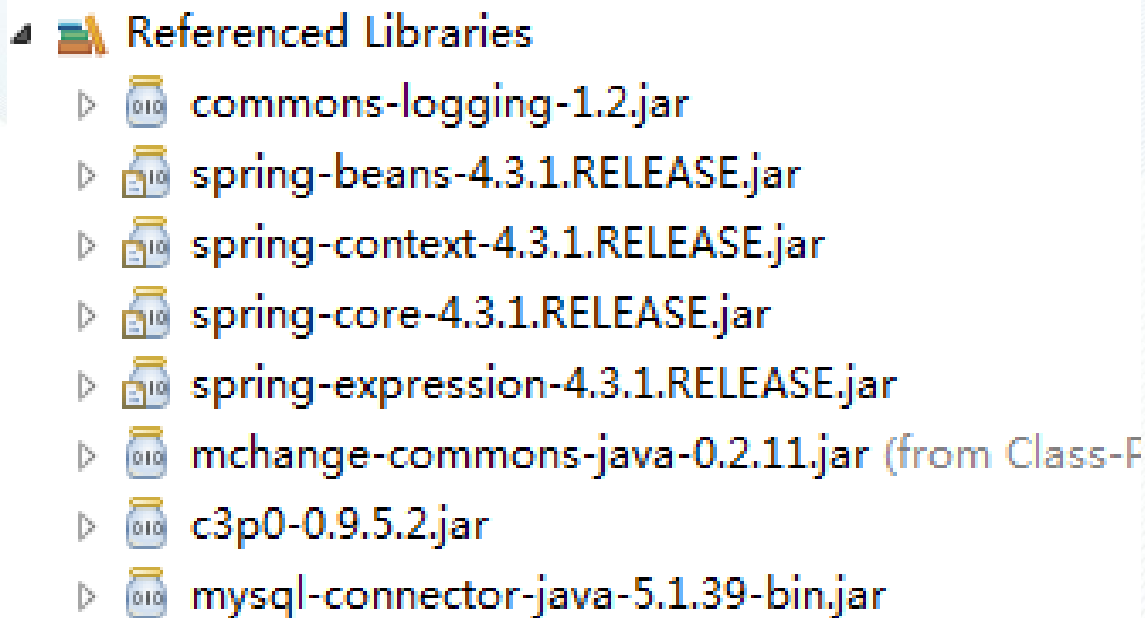
## ■ Spring 2.5 之后：可通过 <context:property-placeholder> 元素简化：

- + <beans> 中添加 context Schema 定义
- + 在配置文件中加入如下配置：

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

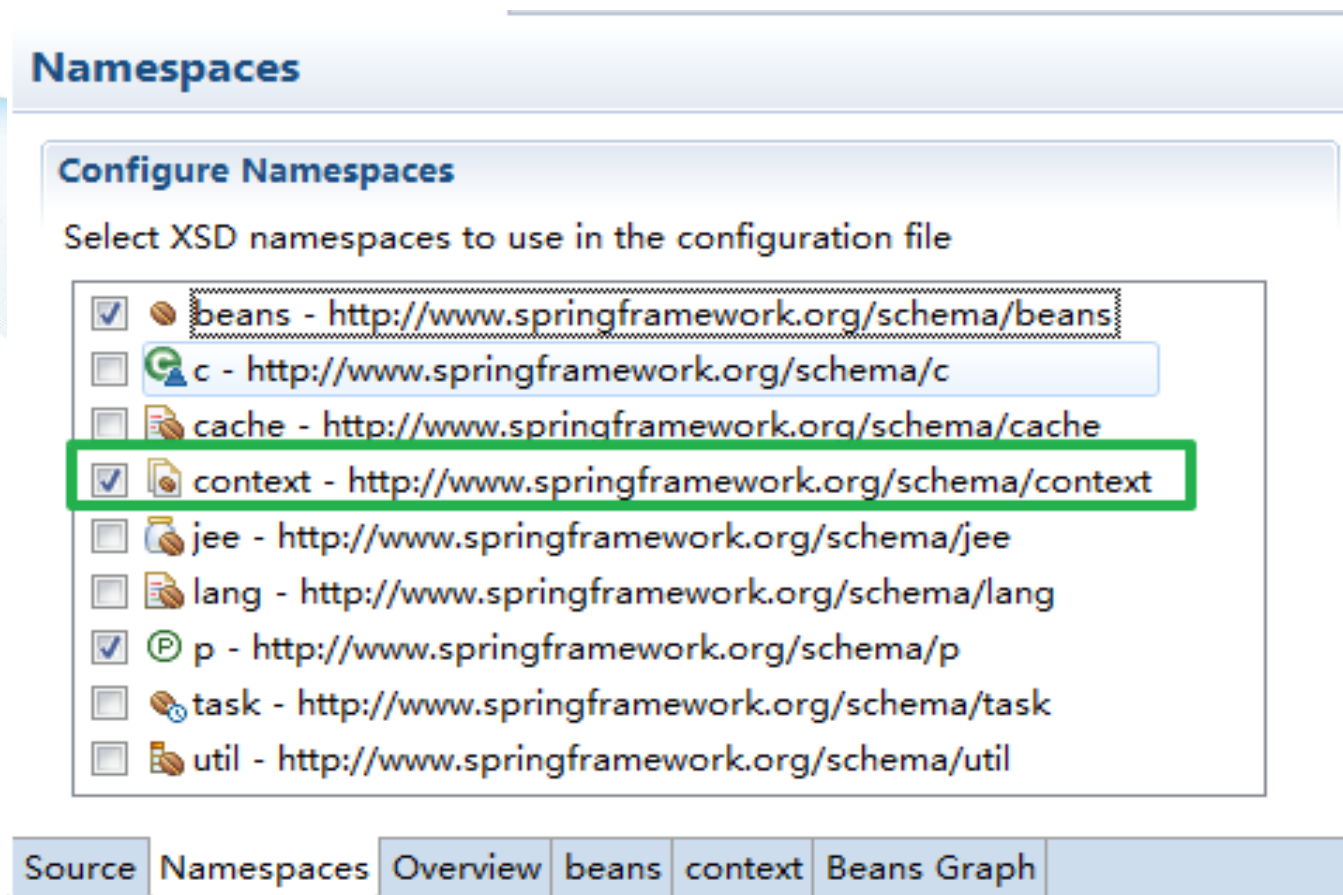
# 使用外部属性文件配置数据源范例（1）

首先向项目中加入三个jar包，一个是c3p0的jar包，另一个是mchange-commons-java-0.2.11.jar，还有一个是mysql的jar包，如下图所示：



# 使用外部属性文件配置数据源范例（2）

导入context命名空间：



# 使用外部属性文件配置数据源范例（3）

```
<!-- 导入属性文件 -->
```

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

```
<!-- 使用外部化属性文件的属性 -->
```

beans-properties.xml

```
<bean id="datasource"
```

```
class="com.mchange.v2.c3p0.ComboPooledDataSource">
```

```
  <property name="user" value="${user}"/>
```

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

```
  <property name="driverClass" value="${driverClass}"/>
```

```
  <property name="jdbcUrl" value="${jdbcUrl}"/>
```

```
</bean>
```

```
user=root
```

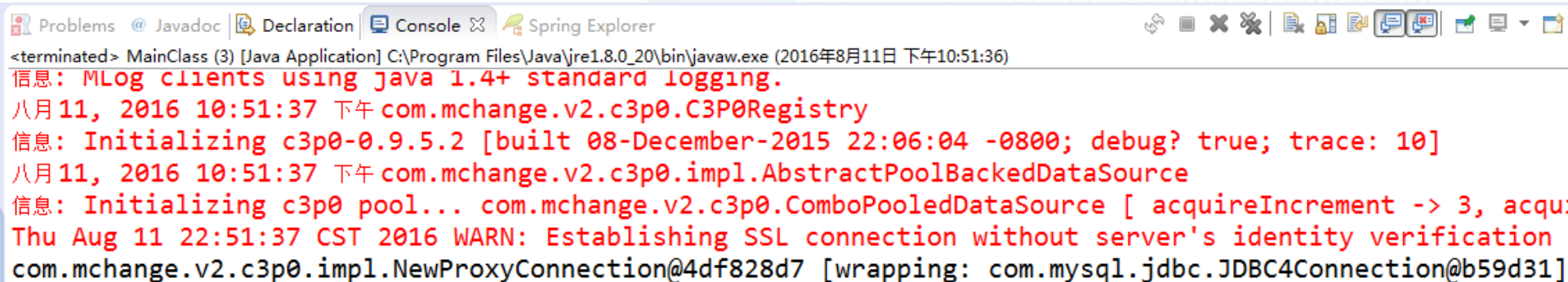
```
password=123456
```

```
driverClass=org.gjt.mm.mysql.Driver
```

```
jdbcUrl=jdbc:mysql://172.16.72.154:3306/test
```

db.properties

这样做的好处在于，一旦需要更换底层数据库或者需要修改数据库的信息时，修改资源文件的成本要比修改Spring配置文件的成本低得多。





# 内容提要

- 配置 bean
  - + SpEL

# Spring表达式语言：SpEL

- Spring 表达式语言（简称SpEL）：是一个支持运行时查询和操作对象图的强大的表达式语言。
- 语法类似于 EL：SpEL 使用 `#{...}` 作为定界符，所有在大括号中的字符都将被认为是 SpEL
- SpEL 为 bean 的属性进行动态赋值提供了便利
- 通过 SpEL 可以实现：
  - + 通过 bean 的 id 对 bean 进行引用
  - + 调用方法以及引用对象中的属性
  - + 计算表达式的值
  - + 正则表达式的匹配

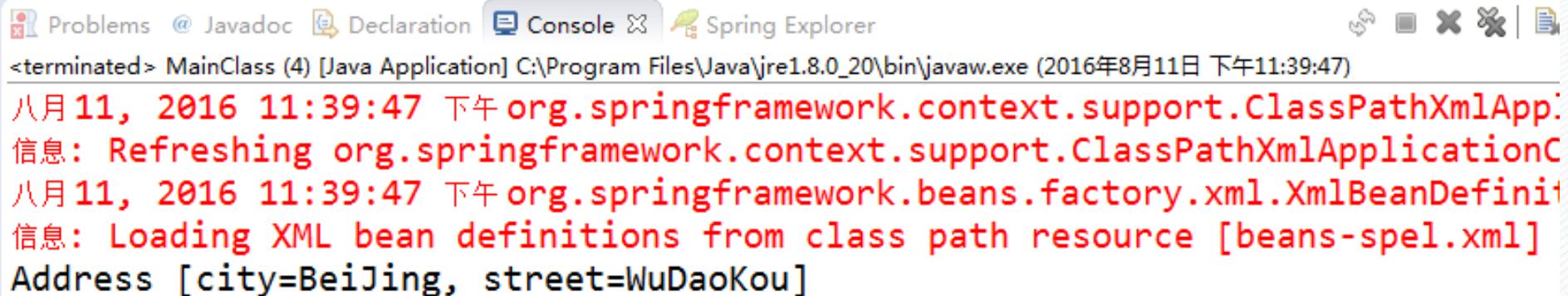
# SpEL: 字面量（了解即可，不常用）

## ■ 字面量的表示:

- + 整数: `<property name="count" value="#{5}"/>`
- + 小数: `<property name="frequency" value="#{89.7}"/>`
- + 科学计数法: `<property name="capacity" value="#{1e4}"/>`
- + **String**可以使用单引号或者双引号作为字符串的定界符号:  
`<property name="name" value="#{'Chuck'}/>` 或 `<property name='name' value='#{"Chuck"}'/>`
- + Boolean: `<property name="enabled" value="#{false}"/>`

# SpEL: 字面量范例

```
<bean id="address" class="cn.edu.nuc.spring.spel.Address">  
    <property name="city" value="#{'BeiJing'}"/>  
    <property name="street" value="WuDaoKou"/>  
</bean>
```



The screenshot shows an IDE console window with the following tabs: Problems, Javadoc, Declaration, Console, and Spring Explorer. The console output is as follows:

```
<terminated> MainClass (4) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月11日 下午11:39:47)  
八月 11, 2016 11:39:47 下午 org.springframework.context.support.ClassPathXmlApp:  
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationC  
八月 11, 2016 11:39:47 下午 org.springframework.beans.factory.xml.XmlBeanDefiniti  
信息: Loading XML bean definitions from class path resource [beans-spel.xml]  
Address [city=BeiJing, street=WuDaoKou]
```



# SpEL: 引用 Bean、属性和方法 (1)

## ■ 引用其他对象:

<!-- 通过 value 属性和 SpEL 配置 Bean 之间的应用关系 -->

```
<property name="prefix" value="#{prefixGenerator}"></property>
```

## ■ 引用其他对象的属性

<!-- 通过 value 属性和 SpEL 配置 suffix 属性值为另一个 Bean 的 suffix 属性值 -->

```
<property name="suffix" value="#{sequenceGenerator2.suffix}"/>
```

## ■ 调用其他方法, 还可以链式操作

<!-- 通过 value 属性和 SpEL 配置 suffix 属性值为另一个 Bean 的方法的返回值 -->

```
<property name="suffix" value="#{sequenceGenerator2.toString()}">
```

<!-- 方法的连缀 -->

```
<property name="suffix"
```

```
    value="#{sequenceGenerator2.toString().toUpperCase()}">
```

# SpEL支持的运算符符号（1）

- 算数运算符：+，-，\*，/，%，^：

```
<property name="adjustedAmount" value="#{counter.total + 42}"/>
<property name="adjustedAmount" value="#{counter.total - 20}"/>
<property name="circumference" value="#{2 * T(java.lang.Math).PI * circle.radius}"/>
<property name="average" value="#{counter.total / counter.count}"/>
<property name="remainder" value="#{counter.total % counter.count}"/>
<property name="area" value="#{T(java.lang.Math).PI * circle.radius ^ 2}"/>
```

- 加号还可以用作字符串连接：

```
<constructor-arg
    value="#{performer.firstName + ' ' + performer.lastName}"/>
```

- 比较运算符：<，>，==，<=，>=，lt，gt，eq，le，ge

```
<property name="equal" value="#{counter.total == 100}"/>
<property name="hasCapacity" value="#{counter.total le 100000}"/>
```

# SpEL支持的运算符符号（2）

- 逻辑运算符: and, or, not, |

```
<property name="largeCircle" value="#{shape.kind == 'circle' and shape.perimeter gt 10000}"/>
<property name="outOfStock" value="#{!product.available}"/>
<property name="outOfStock" value="#{not product.available}"/>
```

- if-else 运算符: ?: (ternary), ?: (Elvis)

```
<constructor-arg
  value="#{songSelector.selectSong()=='Jingle Bells'?piano:' Jingle Bells '}"/>
```

- if-else 的变体

```
<constructor-arg
  value="#{kenny.song ?: 'Greensleeves'}"/>
```

- 正则表达式: matches

```
<constructor-arg
  value="#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}'}"/>
```

# SpEL举例 (1)

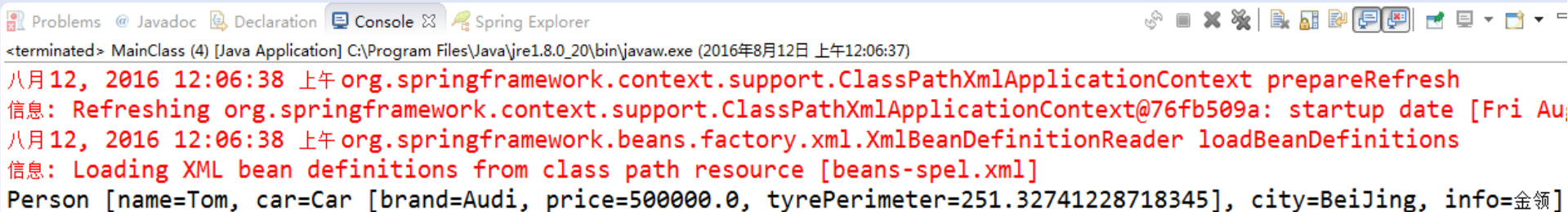
```
<bean id="address" class="cn.edu.nuc.spring.spel.Address">  
    <!-- 使用spel为属性赋一个字面值 -->  
    <property name="city" value="#{'BeiJing'}"/>  
    <property name="street" value="WuDaoKou"/>  
</bean>
```

```
<bean id="car" class="cn.edu.nuc.spring.spel.Car">  
    <property name="brand" value="Audi"/>  
    <property name="price" value="500000"/>  
    <!-- 使用SpEL引用静态属性 -->  
    <property name="tyrePerimeter"  
        value="#{T(java.lang.Math).PI * 80}"/>  
</bean>
```



# SpEL举例 (2)

```
<bean id="person" class="cn.edu.nuc.spring.spel.Person">
    <!-- 使用SpEL引用其他的bean -->
    <property name="car" value="#{car}"/>
    <!-- 使用SpEL引用其他的bean的属性 -->
    <property name="city" value="#{address.city}"/>
    <!-- 在SpEL中使用运算符 -->
    <property name="info"
        value="#{car.price > 300000 ? '金领' : '白领'}/>
    <property name="name" value="Tom"/>
</bean>
```



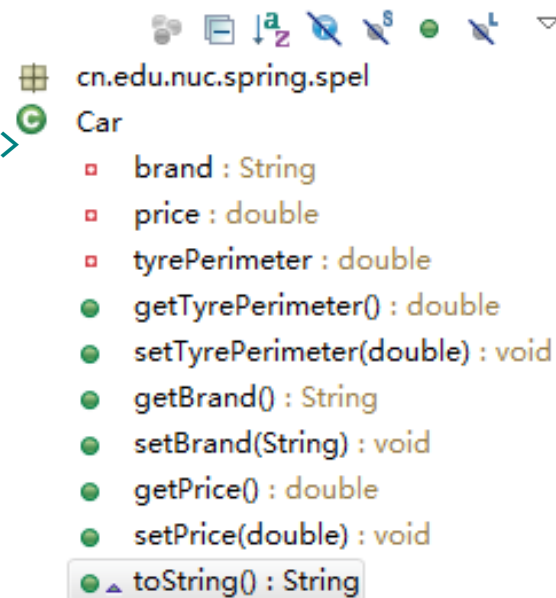
The screenshot shows an IDE interface with a console window. The console displays the following logs:

```
<terminated> MainClass (4) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月12日 上午12:06:37)
八月 12, 2016 12:06:38 上午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@76fb509a: startup date [Fri Aug 12, 2016 12:06:38 上午]
八月 12, 2016 12:06:38 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [beans-spel.xml]
Person [name=Tom, car=Car [brand=Audi, price=500000.0, tyrePerimeter=251.32741228718345], city=BeiJing, info=金领]
```

# SpEL: 调用静态方法或静态属性

- **调用静态方法或静态属性**: 通过 **T()** 调用一个类的静态方法, 它将返回一个 **Class Object**, 然后再调用相应的方法或属性:

```
<bean id="car" class="cn.edu.nuc.spring.spel.Car">
  <property name="brand" value="Audi"/>
  <property name="price" value="500000"/>
  <!-- 使用SpEL引用静态属性 -->
  <property name="tyrePerimeter"
    value="#{T(java.Lang.Math).PI * 80}"/>
</bean>
```



Problems @ Javadoc Declaration Console Spring Explorer

<terminated> MainClass (4) [Java Application] C:\Program Files\Java\jre1.8.0\_20\bin\javaw.exe (2016年8月11日 下午11:50:10)

```
八月 11, 2016 11:50:11 下午 org.springframework.context.support.ClassPathXmlApp
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationC
八月 11, 2016 11:50:11 下午 org.springframework.beans.factory.xml.XmlBeanDefinit
信息: Loading XML bean definitions from class path resource [beans-spel.xml]
Car [brand=Audi, price=500000.0, tyrePerimeter=251.32741228718345]
```

# 内容提要

## ■ 配置 bean

+ IOC 容器中 Bean 的生命周期

# IOC 容器中 Bean 的生命周期方法

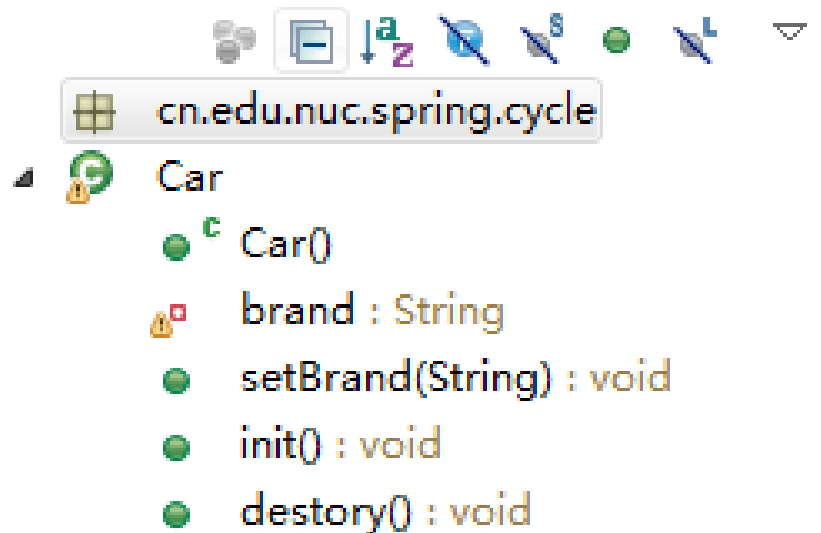
- Spring IOC 容器可以管理 Bean 的生命周期, Spring 允许在 Bean 生命周期的特定点执行定制的任务.
- Spring IOC 容器对 Bean 的生命周期进行管理的过程:
  - + 通过构造器或工厂方法创建 Bean 实例
  - + 为 Bean 的属性设置值和对其他 Bean 的引用
  - + 调用 Bean 的初始化方法
  - + Bean 可以使用了
  - + 当容器关闭时, 调用 Bean 的销毁方法
- 在 Bean 的声明里设置 `init-method` 和 `destroy-method` 属性, 为 Bean 指定初始化和销毁方法.



# IOC 容器中 Bean 的生命周期方法范例 (1)

```
package cn.edu.nuc.spring.cycle;
public class Car {
    public Car() {
        System.out.println("car's constructor...");
    }
    private String brand;
    public void setBrand(String brand) {
        System.out.println("set brand...");
        this.brand = brand;
    }
    public void init() {
        System.out.println("init...");
    }
    public void destroy() {
        System.out.println("destroy...");
    }
}
```

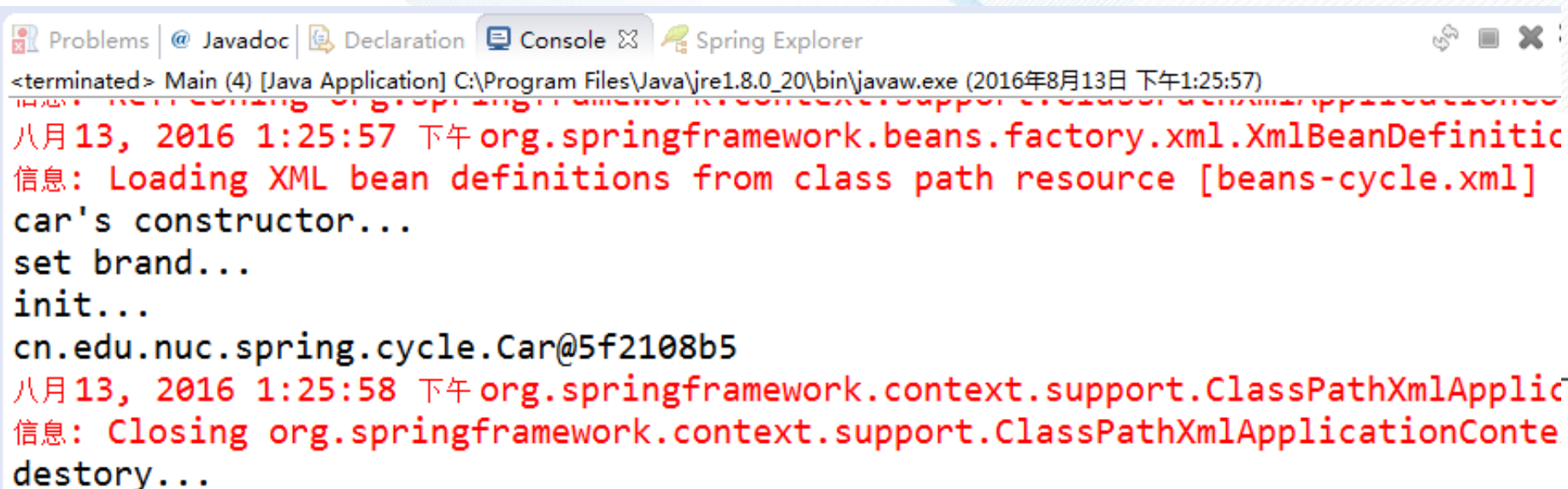
# IOC 容器中 Bean 的生命周期方法范例 (2)



```
<bean id="car" class="cn.edu.nuc.spring.cycle.Car"
      init-method="init" destroy-method="destory">
  <property name="brand" value="Audi"/>
</bean>
```

# IOC 容器中 Bean 的生命周期方法范例 (3)

```
package cn.edu.nuc.spring.cycle;
public class Main {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext("beans-cycle.xml");
        Car car = (Car) ctx.getBean("car");
        System.out.println(car);
        ctx.close(); //关闭Ioc容器
    }
}
```



<terminated> Main (4) [Java Application] C:\Program Files\Java\jre1.8.0\_20\bin\javaw.exe (2016年8月13日 下午1:25:57)

信息: Loading XML bean definitions from class path resource [beans-cycle.xml]

car's constructor...

set brand...

init...

cn.edu.nuc.spring.cycle.Car@5f2108b5

八月13, 2016 1:25:58 下午 org.springframework.context.support.ClassPathXmlApplic

信息: Closing org.springframework.context.support.ClassPathXmlApplicationConte

destory...

# 创建 Bean 后置处理器(1)

- Bean 后置处理器允许在调用初始化方法前后对 Bean 进行额外的处理.
- Bean 后置处理器对 IOC 容器里的所有 Bean 实例逐一处理, 而非单一实例. 其典型应用是: 检查 Bean 属性的正确性 or 根据特定的标准更改 Bean 的属性.



# 创建 Bean 后置处理器 (2)

- 对Bean 后置处理器而言，需要实现 `org.springframework.beans.factory.config` **Interface BeanPostProcessor** 接口。在初始化方法被调用前后，Spring 将把每个 Bean 实例分别传递给上述接口的以下两个方法：

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
Object	<code>postProcessAfterInitialization(Object bean, String beanName)</code> Apply this BeanPostProcessor to the given new bean instance <i>after</i> any bean initialization callbacks (like InitializingBean's <code>afterPropertiesSet</code> or a custom <code>init-method</code> ).	
Object	<code>postProcessBeforeInitialization(Object bean, String beanName)</code> Apply this BeanPostProcessor to the given new bean instance <i>before</i> any bean initialization callbacks (like InitializingBean's <code>afterPropertiesSet</code> or a custom <code>init-method</code> ).	

# 添加 Bean 后置处理器后 Bean 的生命周期

## ■ Spring IOC 容器对 Bean 的生命周期进行管理的过程：

- + 通过构造器或工厂方法创建 Bean 实例
- + 为 Bean 的属性设置值和对其他 Bean 的引用
- + 将 Bean 实例传递给 Bean 后置处理器的 `postProcessBeforeInitialization` 方法
- + 调用 Bean 的初始化方法
- + 将 Bean 实例传递给 Bean 后置处理器的 `postProcessAfterInitialization` 方法
- + Bean 可以使用了
- + 当容器关闭时，调用 Bean 的销毁方法

# 添加 Bean 后置处理器范例（1）

```
public class MyBeanPostProcessor implements BeanPostProcessor{
    @Override
    public Object postProcessBeforeInitialization(
        Object bean, String beanName) throws BeansException {
        System.out.println("PostProcessBeforeInitalization: "
            + bean + "," + beanName);
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(
        Object bean, String beanName) throws BeansException {
        System.out.println("PostProcessBeforeInitalization: "
            + bean + "," + beanName);
        Car car = new Car();
        car.setBrand("Ford");
        return car;
    }
}
```

## 添加 Bean 后置处理器范例（2）

```
<bean id="car" class="cn.edu.nuc.spring.cycle.Car"
      init-method="init" destroy-method="destory">
  <property name="brand" value="Audi"></property>
</bean>
```

<!-- 实现BeanPostProcessor接口，并提供

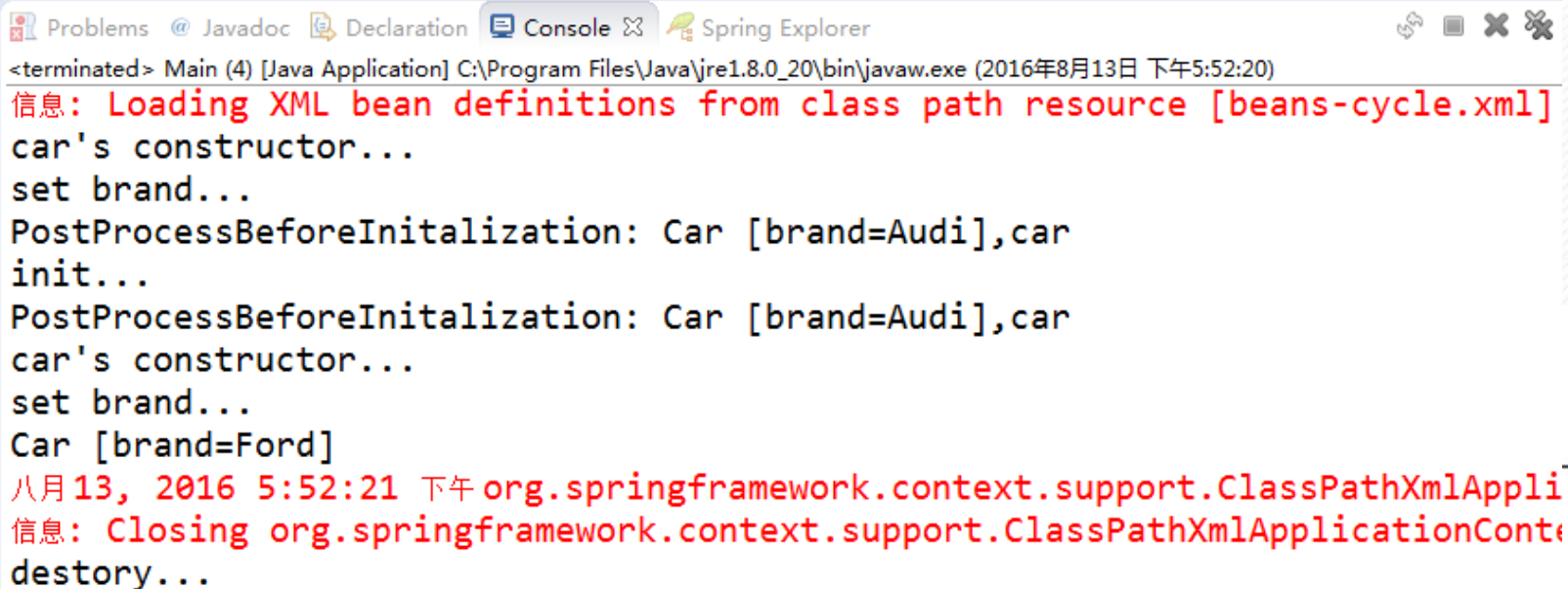
Object postProcessBeforeInitialization(Object bean, String beanName)

Object postProcessAfterInitialization(Object bean, String beanName)

方法的实现-->

<!-- 配置bean的后置处理器 -->

```
<bean class="cn.edu.nuc.spring.cycle.MyBeanPostProcessor"/>
```



```
Problems @ Javadoc Declaration Console Spring Explorer
<terminated> Main (4) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月13日 下午5:52:20)
信息: Loading XML bean definitions from class path resource [beans-cycle.xml]
car's constructor...
set brand...
PostProcessBeforeInitalization: Car [brand=Audi],car
init...
PostProcessBeforeInitalization: Car [brand=Audi],car
car's constructor...
set brand...
Car [brand=Ford]
八月 13, 2016 5:52:21 下午 org.springframework.context.support.ClassPathXmlAppli
信息: Closing org.springframework.context.support.ClassPathXmlApplicationConte
destory...
```



# 内容提要

## ■ 配置 bean

- + Bean 的配置方式：通过工厂方法（静态工厂方法 & 实例工厂方法）、FactoryBean

# 通过调用静态工厂方法创建 Bean

- 调用**静态工厂方法**创建 Bean是将**对象创建的过程封装到静态方法中**. 当客户端需要对象时, 只需要简单地调用静态方法, 而不同关心创建对象的细节.
- 要声明通过静态方法创建的 Bean, 需要在 Bean 的 **class** 属性里指定拥有该工厂的方法的类, 同时在 **factory-method** 属性里指定工厂方法的名称. 最后, 使用 **<constructor-arg>** 元素为该方法传递方法参数.

# 通过调用静态工厂方法创建 Bean 范例 (1)

```
package cn.edu.nuc.spring.factory;

public class Car {
    private String brand;
    private double price;
    public Car() {System.out.println("Car's Constructor...");}
    public String getBrand() {return brand;}
    public void setBrand(String brand) {this.brand = brand;}
    public double getPrice() {return price;}
    public void setPrice(double price) {this.price = price;}
    @Override
    public String toString() {
        return "Car [brand=" + brand + ", price=" + price + "];"
    }
    public Car(String brand, double price) {
        this.brand = brand;
        this.price = price;
    }
}
```

# 通过调用静态工厂方法创建 Bean 范例 (2)

```
package cn.edu.nuc.spring.factory;
```

```
/**
```

```
 * 静态工厂方法：直接调用某一个类的静态方法就可以返回Bean的实例
```

```
 */
```

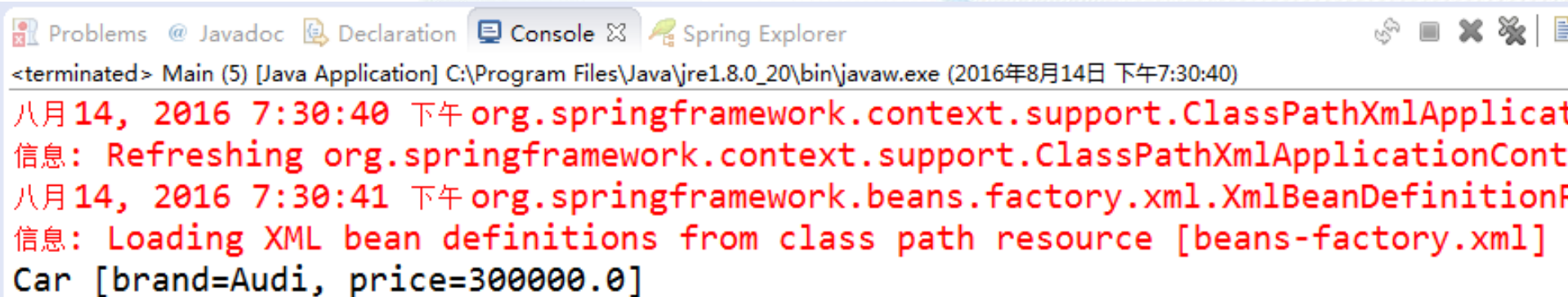
```
public class StaticCarFactory {  
    private static Map<String, Car> cars = new HashMap<>();  
    static {  
        cars.put("Audi", new Car("Audi", 300000));  
        cars.put("Ford", new Car("Ford", 400000));  
    }  
    //静态工厂方法  
    public static Car getCar(String name) {  
        return cars.get(name);  
    }  
}
```



# 通过调用静态工厂方法创建 Bean 范例 (3)

<!-- 通过静态工厂方法来配置bean,注意不是配置静态工厂方法实例,而是配置bean实例 -->

```
<bean id="car1" class="cn.edu.nuc.spring.factory.StaticCarFactory"
      factory-method="getCar">
  <constructor-arg value="Audi"/>
</bean>
```



The screenshot shows an IDE interface with a 'Console' tab selected. The console output displays the following information:

- IDE tabs: Problems, Javadoc, Declaration, Console, Spring Explorer.
- Execution command: `<terminated> Main (5) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月14日 下午7:30:40)`
- Log messages (all in red text):
  - 八月 14, 2016 7:30:40 下午 `org.springframework.context.support.ClassPathXmlApplication`
  - 信息: `Refreshing org.springframework.context.support.ClassPathXmlApplicationCont`
  - 八月 14, 2016 7:30:41 下午 `org.springframework.beans.factory.xml.XmlBeanDefinitionF`
  - 信息: `Loading XML bean definitions from class path resource [beans-factory.xml]`
  - `Car [brand=Audi, price=300000.0]`

# 通过调用实例工厂方法创建 Bean

- **实例工厂方法**：将对象的创建过程封装到另外一个对象实例的方法里。当客户端需要请求对象时，只需要简单的调用该实例方法而不需要关心对象的创建细节。
- 要声明通过实例工厂方法创建的 Bean
  - + 在 bean 的 **factory-bean** 属性里指定拥有该工厂方法的 Bean
  - + 在 **factory-method** 属性里指定该工厂方法的名称
  - + 使用 **constructor-arg** 元素为工厂方法传递方法参数

# 通过调用实例工厂方法创建 Bean范例（1）

```
package cn.edu.nuc.spring.factory;
```

```
/**
```

\* 实例工厂方法:实例工厂的方法，即需要先创建工厂本身，在调用工厂的实例方法来返回bean的实例

```
*
```

```
*/
```

```
public class InstanceCarFactory {  
    private Map<String, Car> cars = null;
```

```
    public InstanceCarFactory() {  
        cars = new HashMap<>();  
        cars.put("Audi", new Car("Audi", 300000));  
        cars.put("Ford", new Car("Ford", 400000));  
    }
```

```
    public Car getCar(String brand) {  
        return cars.get(brand);  
    }
```

```
}
```



# 通过调用实例工厂方法创建 Bean范例（2）

```
<!-- 配置工厂的实例 -->
```

```
<bean id="carFactory"  
      class="cn.edu.nuc.spring.factory.InstanceCarFactory"/>
```

```
<!-- 通过实例工厂方法来配置bean -->
```

```
<!--
```

factory-bean属性：指向静态工厂方法的bean

factory-method：指向静态工厂方法的名字

constructor-arg：如果工厂方法需要传入参数，则用constructor-arg配置参数

```
-->
```

```
<bean id="car2" factory-bean="carFactory" factory-method="getCar">  
    <constructor-arg value="Ford"/>
```

```
</bean>
```

Problems @ Javadoc Declaration Console Spring Explorer

<terminated> Main (5) [Java Application] C:\Program Files\Java\jre1.8.0\_20\bin\javaw.exe (2016年8月14日 下午7:58:27)

八月 14, 2016 7:58:27 下午 org.springframework.context.support.ClassPathXmlApplicationContext

信息： Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@76f

八月 14, 2016 7:58:27 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader

信息： Loading XML bean definitions from class path resource [beans-factory.xml]

Car [brand=Ford, price=400000.0]



# 实现 FactoryBean 接口在 Spring IOC 容器中配置 Bean

- Spring 中有两种类型的 Bean，一种是普通Bean，另一种是工厂Bean，即FactoryBean.
- 工厂 Bean 跟普通Bean不同，其返回的对象不是指定类的一个实例，其返回的是该工厂 Bean 的 getObject 方法所返回的对象

```
public interface FactoryBean {  
    //FactoryBean 返回的实例  
    Object getObject() throws Exception;  
  
    //FactoryBean 返回的类型  
    Class getObjectType();  
  
    //FactoryBean 返回的实例是否为单例  
    boolean isSingleton();  
}
```

# 实现 FactoryBean 接口在 Spring IOC 容器中配置 Bean

org.springframework.beans.factory

## Interface FactoryBean<T>

### Method Summary

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
T	<code>getObject()</code>	Return an instance (possibly shared or independent) of the object managed by this factory.
<code>Class&lt;?&gt;</code>	<code>getObjectType()</code>	Return the type of object that this FactoryBean creates, or null if not known in advance.
boolean	<code>isSingleton()</code>	Is the object managed by this factory a singleton? That is, will <code>getObject()</code> always return the same object (a reference that can be cached)?

FactoryBean<T>

`getObject() : T`

`getObjectType() : Class<?>`

`isSingleton() : boolean`

# 实现 FactoryBean 接口在 IOC 容器中配置 Bean 范例 (1)

```
//自定义的FactoryBean需要实现FactoryBean接口
public class CarFactoryBean implements FactoryBean<Car>{
    private String brand;
    public void setBrand(String brand) {this.brand = brand;}
    //返回Bean的对象
    @Override
    public Car getObject() throws Exception {
        return new Car(brand, 500000);
    }
    //返回Bean的类型
    @Override
    public Class<?> getObjectType() {
        return Car.class;
    }
    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

# 实现 FactoryBean 接口在 IOC 容器中配置 Bean 范例 (2)

<!-- 通过FactoryBean创建Bean的实例

class: 指向FactoryBean的全类名

property: 配置FactoryBean的属性

但实际返回的实例却是FactoryBean的getObject()方法返回的实例!

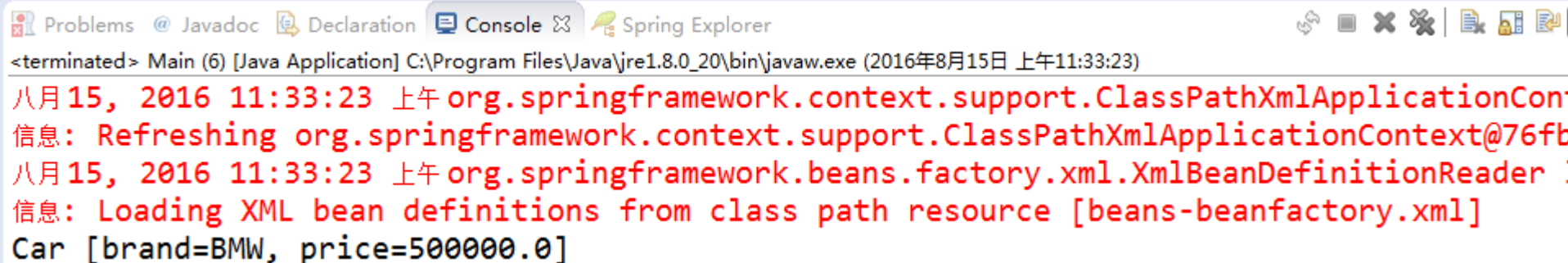
-->

<bean id="car"

class="cn.edu.nuc.spring.factorybean.CarFactoryBean">

<property name="brand" value="BMW"/>

</bean>



The screenshot shows an IDE window with a console tab. The title bar includes 'Problems', 'Javadoc', 'Declaration', 'Console', and 'Spring Explorer'. The console output shows the following logs:

```
<terminated> Main (6) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月15日 上午11:33:23)
八月 15, 2016 11:33:23 上午 org.springframework.context.support.ClassPathXmlApplicationContext:
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@76fb
八月 15, 2016 11:33:23 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader :
信息: Loading XML bean definitions from class path resource [beans-beanfactory.xml]
Car [brand=BMW, price=500000.0]
```



# 内容提要

## ■ 配置 bean

### + 配置形式：基于注解的方式

（基于注解配置 Bean；基于注解来装配 Bean 的属性）

# 在 classpath 中扫描组件

- 组件扫描(component scanning): Spring 能够从 classpath 下自动扫描, 侦测和实例化具有特定注解的组件.
- 特定组件包括:
  - + @Component: 基本注解, 标识了一个受 Spring 管理的组件
  - + @Repository: 标识持久层组件
  - + @Service: 标识服务层(业务层)组件
  - + @Controller: 标识表现层组件
- 对于扫描到的组件, **Spring 有默认的命名策略**: 使用非限定类名, 第一个字母小写. **也可以在注解中通过 value 属性值标识组件的名称**

# 在 classpath 中扫描组件

- 当在组件类上使用了特定的注解之后，还需要在 Spring 的配置文件中声明 `<context:component-scan>`：
  - + `base-package` 属性指定一个需要扫描的基类包，Spring 容器将会扫描这个基类包里及其子包中的所有类。
  - + 当需要扫描多个包时，可以使用逗号分隔。
  - + 如果仅希望扫描特定的类而非基包下的所有类，可使用 `resource-pattern` 属性过滤特定的类，示例：

```
<context:component-scan  
    base-package="com.atguigu.spring.beans"  
    resource-pattern="autowire/*.class"/>
```

- + `<context:include-filter>` 子节点表示要包含的目标类
- + `<context:exclude-filter>` 子节点表示要排除在外的目标类
- + `<context:component-scan>` 下可以拥有若干个 `<context:include-filter>` 和 `<context:exclude-filter>` 子节点

# 在 classpath 中扫描组件

- `<context:include-filter>` 和 `<context:exclude-filter>` 子节点支持多种类型的过滤表达式:

类别	示例	说明
annotation	<code>com.atguigu.XxxAnnotation</code>	所有标注了 <b>XxxAnnotation</b> 的类。该类型采用目标类是否标注了某个注解进行过滤
assinable	<code>com.atguigu.XxxService</code>	所有继承或扩展 <b>XxxService</b> 的类。该类型采用目标类是否继承或扩展某个特定类进行过滤
aspectj	<code>com.atguigu..*Service+</code>	所有类名以 <code>Service</code> 结束的类及继承或扩展它们的类。该类型采用 <code>AspejctJ</code> 表达式进行过滤
regex	<code>com.\atguigu\.anno\..*</code>	所有 <code>com.atguigu.anno</code> 包下的类。该类型采用正则表达式根据类的类名进行过滤
custom	<code>com.atguigu.XxxTypeFilter</code>	采用 <code>XxxTypeFilter</code> 通过代码的方式定义过滤规则。该类必须实现 <code>org.springframework.core.type.TypeFilter</code> 接口



# 在 classpath 中扫描组件范例 (1)

```
package cn.edu.nuc.spring.annotation;
```

```
@Component
```

```
public class TestObject {}
```

```
package cn.edu.nuc.spring.annotation.repository;
```

```
public interface UserRepository {
```

```
    public void save();
```

```
}
```

```
package cn.edu.nuc.spring.annotation.repository;
```

```
import org.springframework.stereotype.Repository;
```

```
@Repository(value="userRepository")
```

```
public class UserRepositoryImpl implements UserRepository {
```

```
    @Override
```

```
    public void save() {
```

```
        System.out.println("UserRepositoryImpl Save...");
```

```
    }
```

```
}
```

# 在 classpath 中扫描组件范例 (2)

```
package cn.edu.nuc.spring.annotation.service;
import org.springframework.stereotype.Service;
@Service
public class UserService {
    public void add() {
        System.out.println("UserService add...");
    }
}

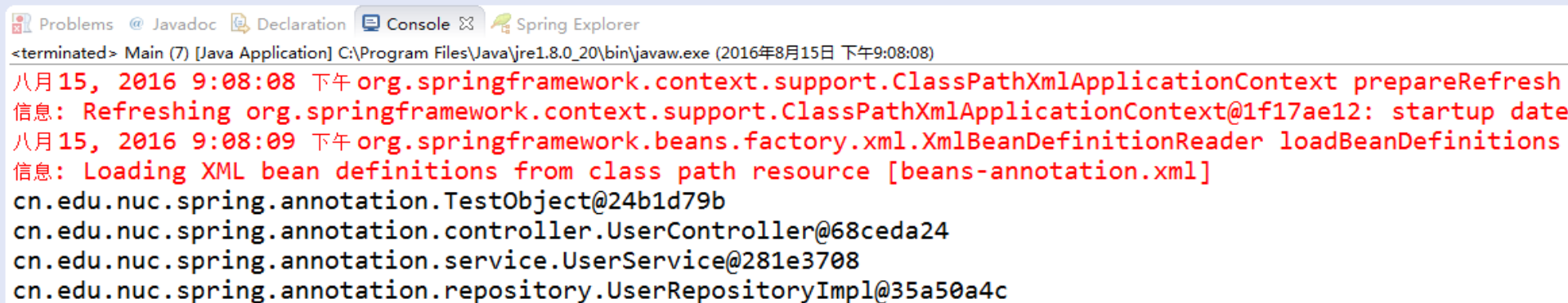
package cn.edu.nuc.spring.annotation.controller;
import org.springframework.stereotype.Controller;
@Controller
public class UserController {
    public void execute() {
        System.out.println("UserController execute...");
    }
}
```

# 在 classpath 中扫描组件范例 (3)

<!-- 指定Spring IoC 容器扫描的包 -->

<context:component-scan

base-package="cn.edu.nuc.spring.annotation"/>

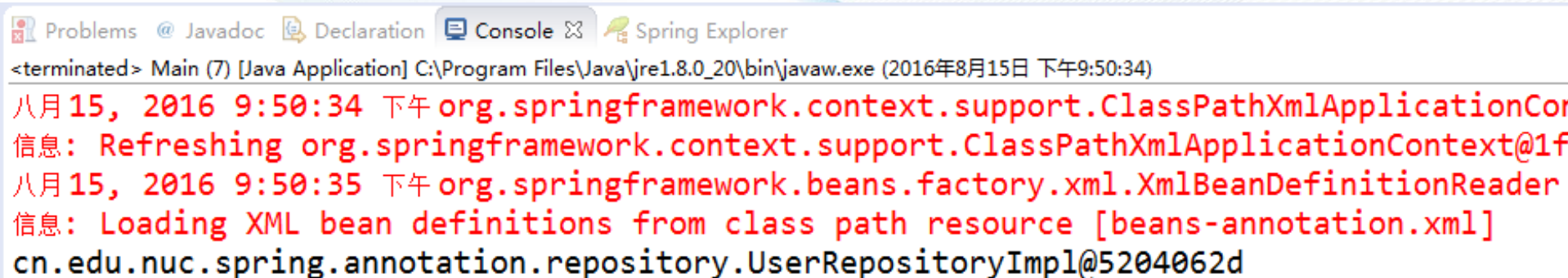


The screenshot shows an IDE console window with the following tabs: Problems, Javadoc, Declaration, Console, and Spring Explorer. The console output is as follows:

```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月15日 下午9:08:08)
八月 15, 2016 9:08:08 下午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1f17ae12: startup date
八月 15, 2016 9:08:09 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [beans-annotation.xml]
cn.edu.nuc.spring.annotation.TestObject@24b1d79b
cn.edu.nuc.spring.annotation.controller.UserController@68ceda24
cn.edu.nuc.spring.annotation.service.UserService@281e3708
cn.edu.nuc.spring.annotation.repository.UserRepositoryImpl@35a50a4c
```

# 在 classpath 中扫描组件范例 (4)

```
<!-- 指定Spring IoC 容器扫描的包 -->
<!-- 可以通过resource-pattern指定扫描的资源 -->
<context:component-scan
    base-package="cn.edu.nuc.spring.annotation"
    resource-pattern="repository/*.class"
/>
```



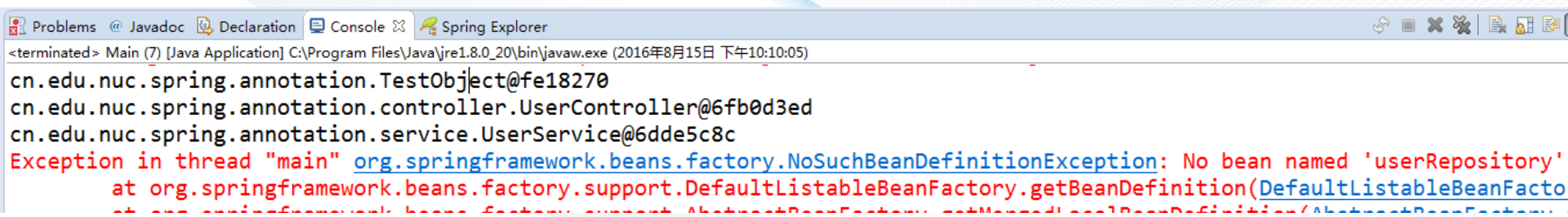
The screenshot shows an IDE interface with a console window. The console title bar includes 'Problems', 'Javadoc', 'Declaration', 'Console', and 'Spring Explorer'. The console output shows the application starting on August 15, 2016, at 9:50:34 PM. It logs the refreshing of the Spring context and the loading of XML bean definitions from the classpath resource 'beans-annotation.xml'. The loaded bean is 'cn.edu.nuc.spring.annotation.repository.UserRepositoryImpl@5204062d'.

```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月15日 下午9:50:34)
八月 15, 2016 9:50:34 下午 org.springframework.context.support.ClassPathXmlApplicationCor
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1f
八月 15, 2016 9:50:35 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader
信息: Loading XML bean definitions from class path resource [beans-annotation.xml]
cn.edu.nuc.spring.annotation.repository.UserRepositoryImpl@5204062d
```



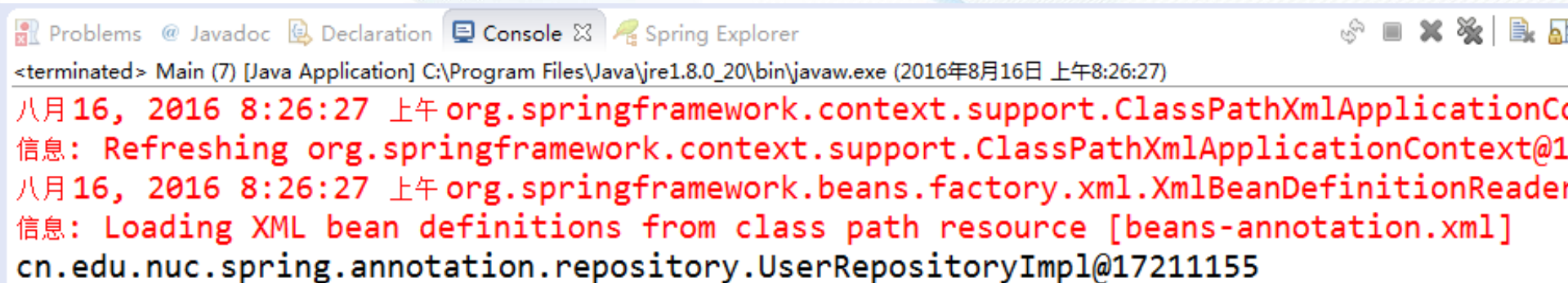
## 在 classpath 中扫描组件范例 (5)

```
<!-- context:exclude-filter子节点指定排除哪些指定表达式的组件 -->
<context:component-scan base-package="cn.edu.nuc.spring.annotation">
    <!-- annotation方式 -->
    <context:exclude-filter type="annotation"
        expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```



# 在 classpath 中扫描组件范例 (6)

```
<!-- context:include-filter子节点指定包含哪些表达式的组件，  
    该子节点需要use-default-filters配合使用 -->  
<context:component-scan base-package="cn.edu.nuc.spring.annotation"  
    use-default-filters="false">  
    <!-- annotation方式 -->  
    <context:include-filter type="annotation"  
        expression="org.springframework.stereotype.Repository"/>  
</context:component-scan>
```

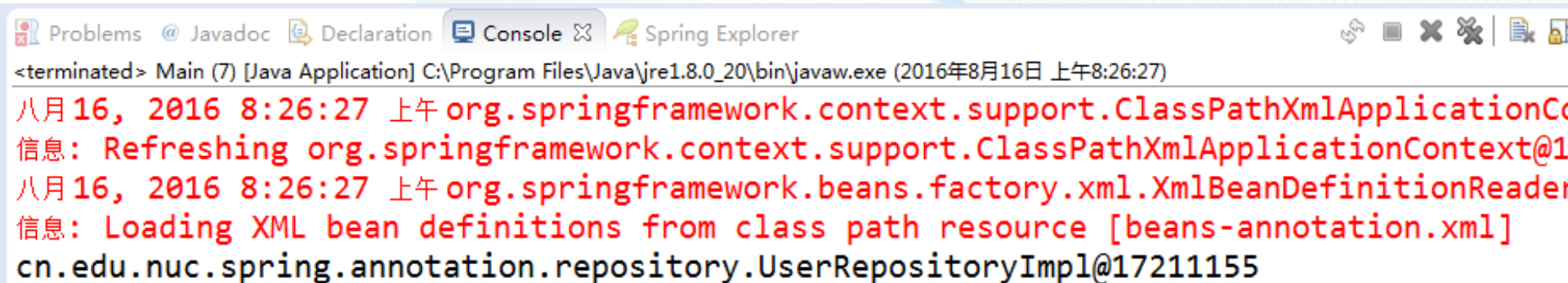


The screenshot shows an IDE interface with a console window. The console displays the following logs:

```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月16日 上午8:26:27)  
八月 16, 2016 8:26:27 上午 org.springframework.context.support.ClassPathXmlApplicationContext:  
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1  
八月 16, 2016 8:26:27 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader:  
信息: Loading XML bean definitions from class path resource [beans-annotation.xml]  
cn.edu.nuc.spring.annotation.repository.UserRepositoryImpl@17211155
```

# 在 classpath 中扫描组件范例 (7)

```
<!-- context:include-filter子节点指定包含哪些表达式的组件，
    该子节点需要use-default-filters配合使用 -->
<context:component-scan base-package="cn.edu.nuc.spring.annotation"
    use-default-filters="false">
    <!-- annotation方式 -->
    <context:include-filter type="annotation"
        expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

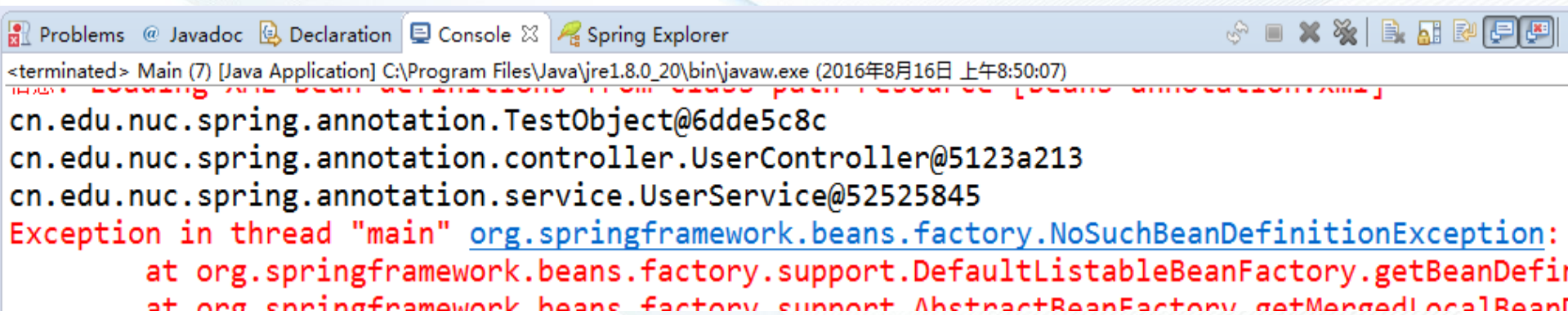


The screenshot shows an IDE interface with a console window. The console displays the following logs:

```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月16日 上午8:26:27)
八月 16, 2016 8:26:27 上午 org.springframework.context.support.ClassPathXmlApplicationContext
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1
八月 16, 2016 8:26:27 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader
信息: Loading XML bean definitions from class path resource [beans-annotation.xml]
cn.edu.nuc.spring.annotation.repository.UserRepositoryImpl@17211155
```

# 在 classpath 中扫描组件范例 (8)

```
<!-- context:exclude-filter子节点指定排除哪些指定表达式的组件 -->
<context:component-scan base-package="cn.edu.nuc.spring.annotation">
    <context:exclude-filter type="assignable"
expression="cn.edu.nuc.spring.annotation.repository.UserRepository"/>
</context:component-scan>
```



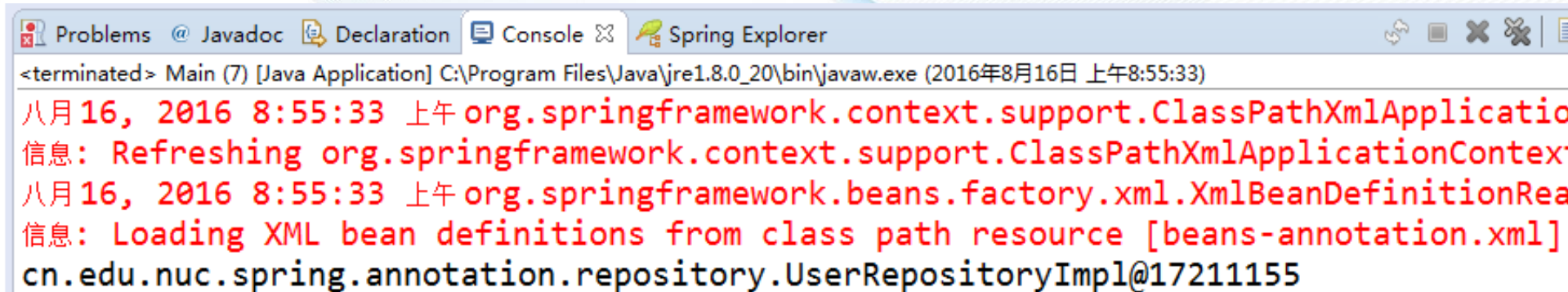
The screenshot shows an IDE window with tabs for Problems, Javadoc, Declaration, Console, and Spring Explorer. The Console tab is active, displaying the following output:

```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月16日 上午8:50:07)
cn.edu.nuc.spring.annotation.TestObject@6dde5c8c
cn.edu.nuc.spring.annotation.controller.UserController@5123a213
cn.edu.nuc.spring.annotation.service.UserService@52525845
Exception in thread "main" org.springframework.beans.factory.NoSuchBeanDefinitionException:
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanDefinition
    at org.springframework.beans.factory.support.AbstractBeanFactory.getMergedLocalBeanDefinition
```



# 在 classpath 中扫描组件范例 (9)

```
<!-- context:include-filter子节点指定包含哪些表达式的组件，  
    该子节点需要use-default-filters配合使用 -->  
<context:component-scan base-package="cn.edu.nuc.spring.annotation"  
    use-default-filters="false">  
    <context:include-filter type="assignable"  
expression="cn.edu.nuc.spring.annotation.repository.UserRepository"/>  
</context:component-scan>
```



The screenshot shows an IDE console window with the following tabs: Problems, Javadoc, Declaration, Console, and Spring Explorer. The console output is as follows:

```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月16日 上午8:55:33)  
八月 16, 2016 8:55:33 上午 org.springframework.context.support.ClassPathXmlApplication  
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext  
八月 16, 2016 8:55:33 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionRea  
信息: Loading XML bean definitions from class path resource [beans-annotation.xml]  
cn.edu.nuc.spring.annotation.repository.UserRepositoryImpl@17211155
```

# 组件装配

- `<context:component-scan>` 元素还会自动注册 `AutowiredAnnotationBeanPostProcessor` 实例，该实例可以自动装配具有 `@Autowired` 和 `@Resource`、`@Inject` 注解的属性。

# 使用 @Autowired 自动装配 Bean

- @Autowired 注解自动装配具有兼容类型的单个 Bean 属性
  - + 构造器, 普通字段(即使是非 public), 一切具有参数的方法都可以应用 @Autowired 注解
  - + 默认情况下, 所有使用 @Autowired 注解的属性都需要被设置. 当 Spring 找不到匹配的 Bean 装配属性时, 会抛出异常, 若某一属性允许不被设置, 可以设置 @Autowired 注解的 required 属性为 false
  - + 默认情况下, 当 IOC 容器里存在多个类型兼容的 Bean 时, 通过类型的自动装配将无法工作. 此时可以在 @Qualifier 注解里提供 Bean 的名称. Spring 允许对方法的入参标注 @Qualifier 已指定注入 Bean 的名称
  - + @Autowired 注解也可以应用在数组类型的属性上, 此时 Spring 将会把所有匹配的 Bean 进行自动装配.
  - + @Autowired 注解也可以应用在集合属性上, 此时 Spring 读取该集合的类型信息, 然后自动装配所有与之兼容的 Bean.
  - + @Autowired 注解用在 java.util.Map 上时, 若该 Map 的键值为 String, 那么 Spring 将自动装配与之 Map 值类型兼容的 Bean, 此时 Bean 的名称作为键值



# 使用 @Autowired 自动装配 Bean 范例 (1)

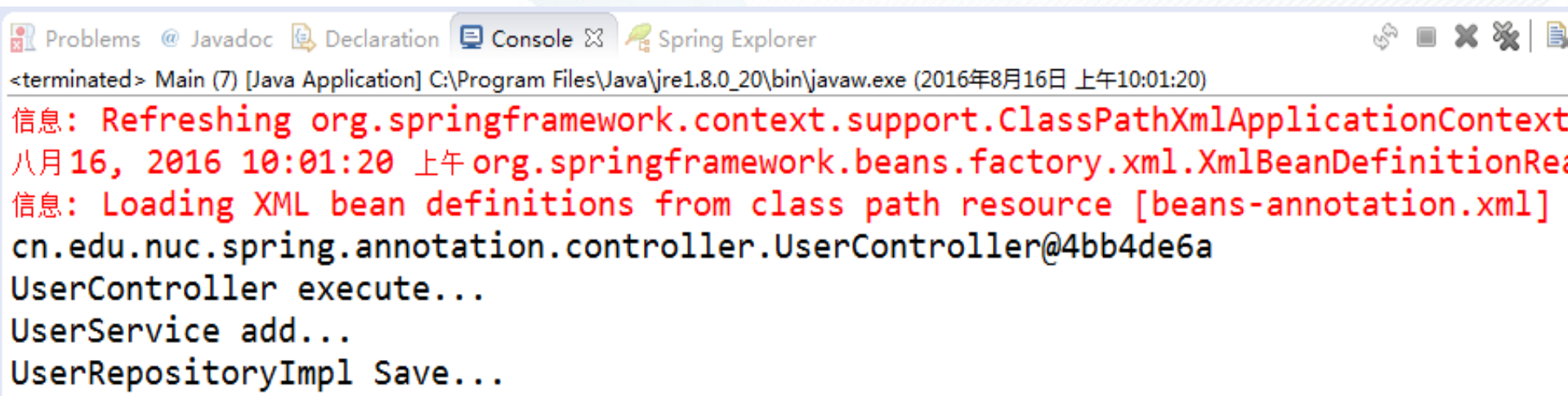
```
package cn.edu.nuc.spring.annotation.controller;  
@Controller  
public class UserController {  
    @Autowired  
    private UserService userService;  
    public void execute() {  
        System.out.println("UserController execute...");  
        userService.add();  
    }  
}
```



# 使用 @Autowired 自动装配 Bean 范例 (2)

```
package cn.edu.nuc.spring.annotation.service;  
@Service  
public class UserService {  
    @Autowired  
    private UserRepository userRepository;  
    public void add() {  
        System.out.println("UserService add...");  
        userRepository.save();  
    }  
}
```

```
<context:component-scan base-package="cn.edu.nuc.spring.annotation"/>
```



The screenshot shows the bottom portion of an IDE window with several tabs: Problems, Javadoc, Declaration, Console, and Spring Explorer. The Console tab is active, displaying the following log output in red and black text:

```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (2016年8月16日 上午10:01:20)  
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext  
八月 16, 2016 10:01:20 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader  
信息: Loading XML bean definitions from class path resource [beans-annotation.xml]  
cn.edu.nuc.spring.annotation.controller.UserController@4bb4de6a  
UserController execute...  
UserService add...  
UserRepositoryImpl Save...
```

# 使用 @Resource 或 @Inject 自动装配 Bean

- Spring 还支持 @Resource 和 @Inject 注解，这两个注解和 @Autowired 注解的功用类似
- @Resource 注解要求提供一个 Bean 名称的属性，若该属性为空，则自动采用标注处的变量或方法名作为 Bean 的名称
- @Inject 和 @Autowired 注解一样也是按类型匹配注入的 Bean，但没有 required 属性
- 建议使用 @Autowired 注解

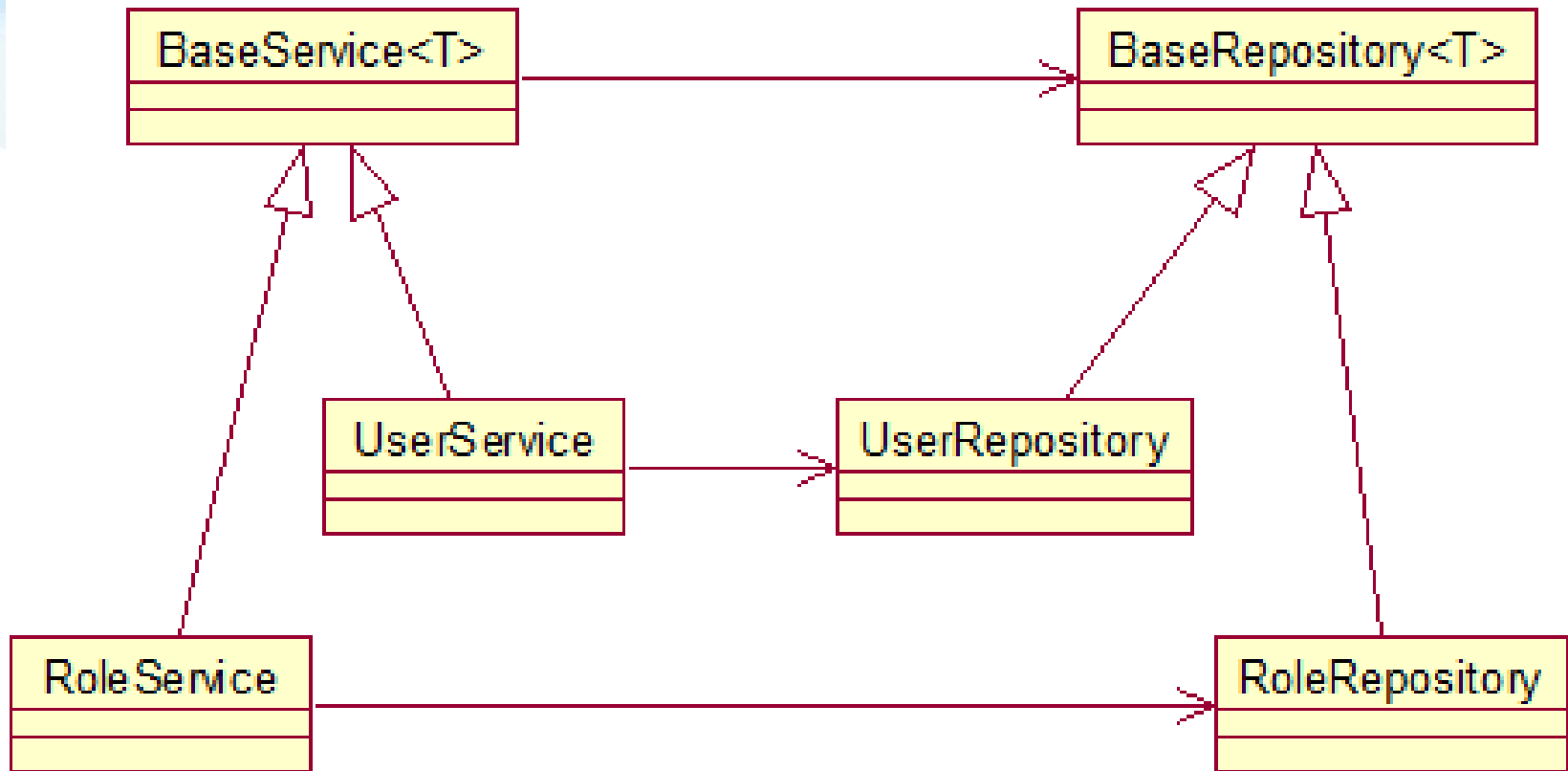
# 内容提要

- 配置 bean

- + Spring 4.x 新特性: 泛型依赖注入

# 泛型依赖注入

- Spring 4.x 中可以为子类注入子类对应的泛型类型的成员变量的引用





# 整合多个配置文件

- Spring 允许通过 `<import>` 将多个配置文件引入到一个文件中，进行配置文件的集成。这样在启动 Spring 容器时，仅需要指定这个合并好的配置文件就可以。
- `import` 元素的 `resource` 属性支持 Spring 的标准的路径资源

地址前缀	示例	对应资源类型
classpath:	classpath:spring-mvc.xml	从类路径下加载资源，classpath: 和 classpath:/ 是等价的
file:	file:/conf/security/spring-shiro.xml	从文件系统目录中装载资源，可采用绝对或相对路径
http://	http://www.atguigu.com/resource/beans.xml	从 WEB 服务器中加载资源
ftp://	ftp://www.atguigu.com/resource/beans.xml	从 FTP 服务器中加载资源



# Spring AOP



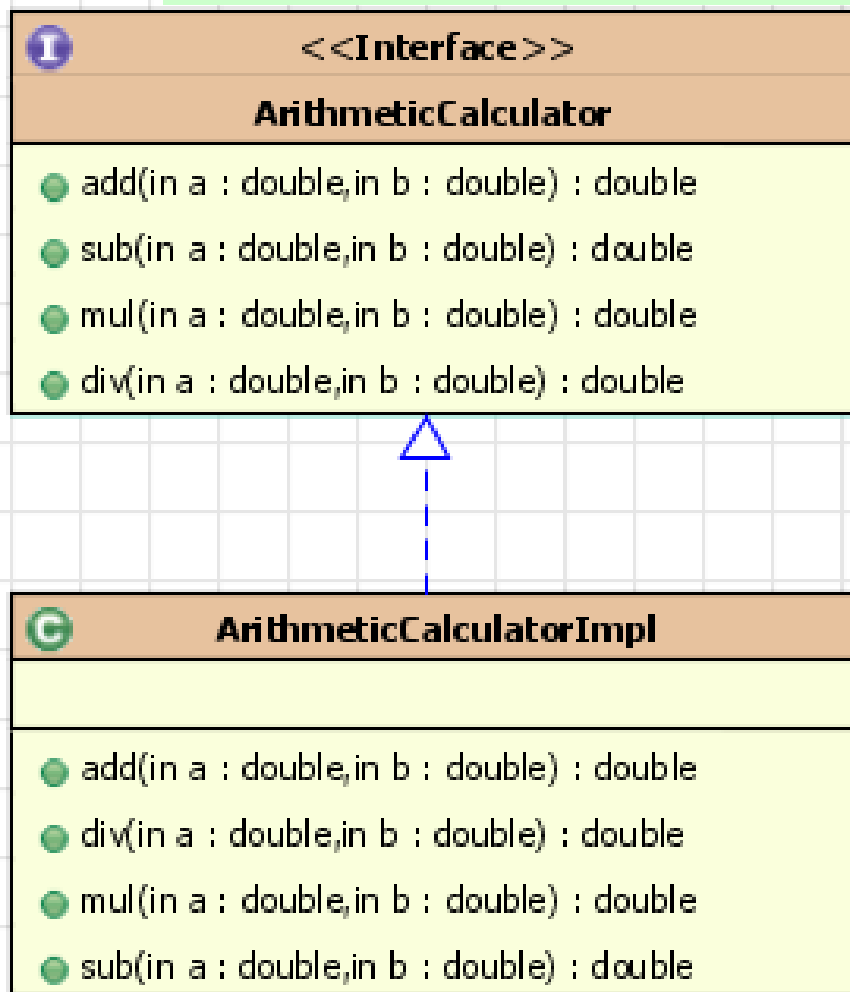
---

# AOP 前奏

- WHY AOP ?

需求1-日志：在程序执行期间追踪正在发生的活动

需求2-验证：希望计算器只能处理正数的运算



# 代码实现片段

```
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {
```

```
@Override
```

```
public void add(int i, int j) {
```

```
    System.out.println("日志:The method add begins with [" +  
        i + ", " + j + "]" );
```

```
    int result = i + j;
```

```
    System.out.println("result: " + result);
```

```
    System.out.println("日志:The method add ends with " + result);
```

```
}
```

```
@Override
```

```
public void sub(int i, int j) {
```

```
    System.out.println("日志:The method sub begins with [" +  
        i + ", " + j + "]" );
```

```
    int result = i - j;
```

```
    System.out.println("result: " + result);
```

```
    System.out.println("日志:The method sub ends with " + result);
```

```
}
```

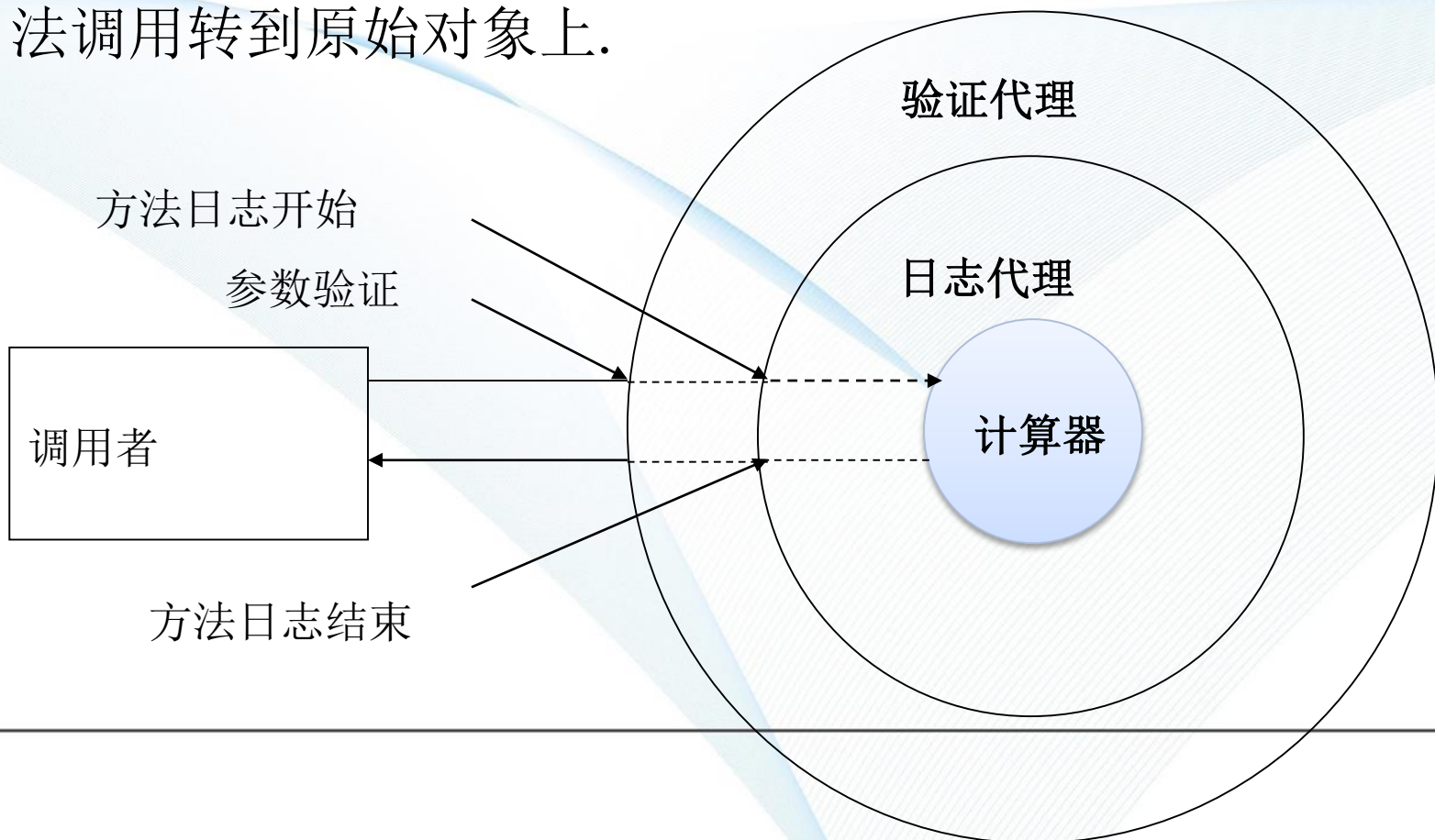


# 问题

- 代码混乱：越来越多的非业务需求（日志和验证等）加入后，原有的业务方法急剧膨胀。每个方法在处理核心逻辑的同时还必须兼顾其他多个关注点。
- 代码分散：以日志需求为例，只是为了满足这个单一需求，就不得不在多个模块（方法）里多次重复相同的日志代码。如果日志需求发生变化，必须修改所有模块。

# 使用动态代理解决上述问题

- 代理设计模式的原理：使用一个代理将对象包装起来，然后用该代理对象取代原始对象。任何对原始对象的调用都要通过代理。代理对象决定是否以及何时将方法调用转到原始对象上。



# CalculatorLoggingHandler

```
public class CalculatorLoggingHandler implements InvocationHandler {

    private Log log = LogFactory.getLog(this.getClass());

    private Object target;

    public CalculatorLoggingHandler(Object target) {
        super();
        this.target = target;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        log.info("The method " + method.getName() + "() begins with " + Arrays.toString(args));
        Object result = method.invoke(target, args);
        log.info("The method " + method.getName() + "() ends with " + result);
        return result;
    }

    public static Object createProxy(Object target){
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
            target.getClass().getInterfaces(),
            new CalculatorLoggingHandler(target));
    }
}
```

# CalculatorValidationHandler

```
public class CalculatorValidationHandler implements InvocationHandler {  
    private Object target;  
  
    public CalculatorValidationHandler(Object target) {  
        this.target = target;  
    }  
  
    public Object invoke(Object proxy, Method method, Object[] args)  
        throws Throwable {  
        for(Object arg : args){  
            validate((Double) arg);  
        }  
        Object result = method.invoke(target, args);  
        return result;  
    }  
  
    public static Object createProxy(Object target){  
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),  
            target.getClass().getInterfaces(),  
            new CalculatorValidationHandler(target));  
    }  
  
    private void validate(double a){  
        if(a < 0)  
            throw new IllegalArgumentException("Positive numbers only");  
    }  
}
```



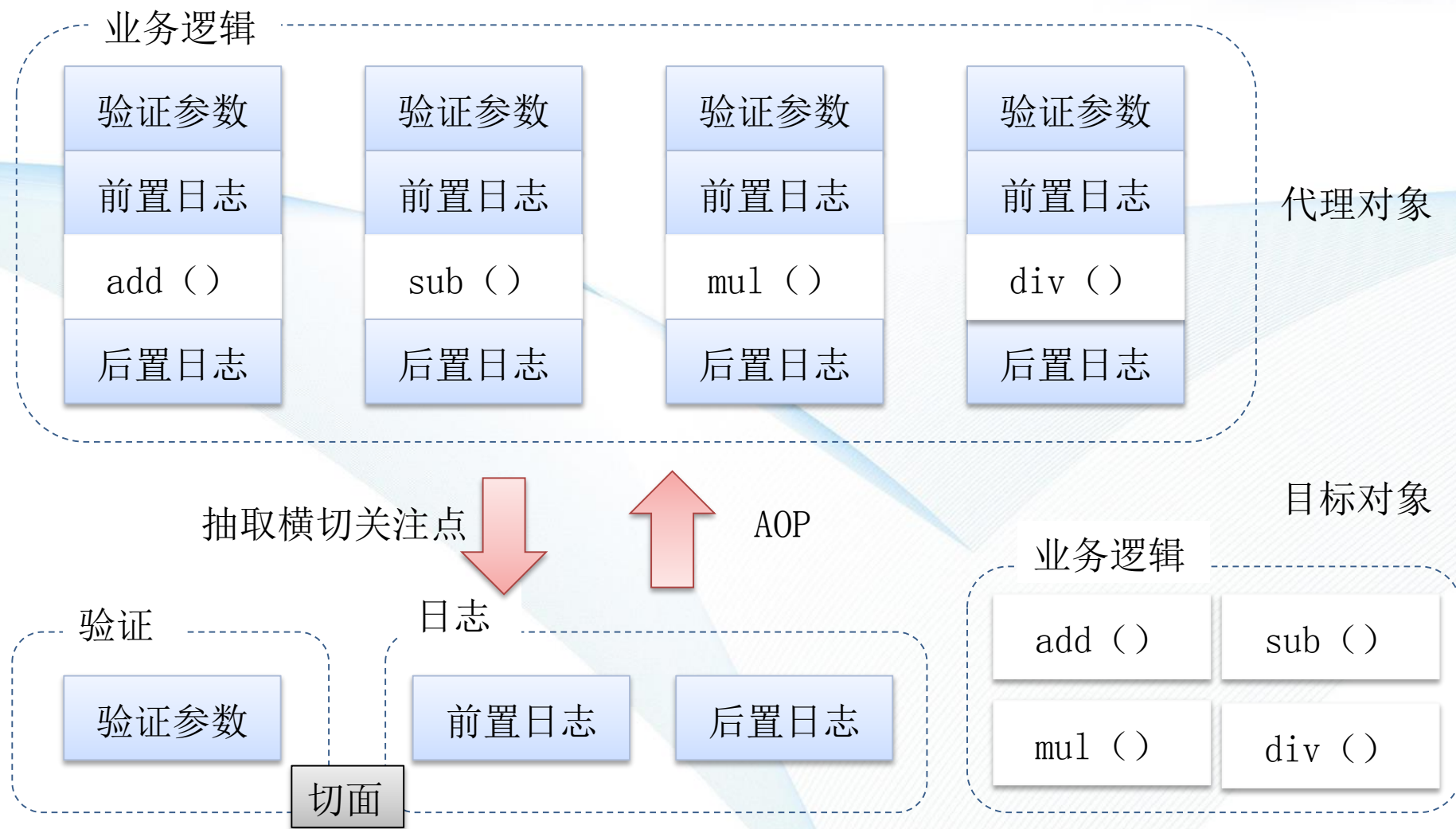
# 测试代码

```
public class Main {  
    public static void main(String[] args) {  
        ArithmeticCalculator arithmeticCalculatorImpl =  
            new ArithmeticCalculatorImpl();  
  
        ArithmeticCalculator arithmeticCalculator  
            = (ArithmeticCalculator) CalculatorValidationHandler  
                .createProxy(CalculatorLoggingHandler  
                    .createProxy(arithmeticCalculatorImpl));  
        System.out.println(arithmeticCalculator.add(-12, 13));  
    }  
}
```

# AOP 简介

- AOP (Aspect-Oriented Programming, **面向切面编程**): 是一种新的方法论, 是对传统 OOP (Object-Oriented Programming, 面向对象编程) 的补充.
- AOP 的主要编程对象是**切面** (aspect), 而**切面模块化横切关注点**.
- 在应用 AOP 编程时, 仍然需要**定义公共功能**, 但可以明确的定义这个功能在哪里, 以什么方式应用, **并且不必修改受影响的类**. 这样一来**横切关注点就被模块化到特殊的对象(切面)**里.
- AOP 的好处:
  - + 每个事物逻辑位于一个位置, 代码不分散, 便于维护和升级
  - + 业务模块更简洁, 只包含核心业务代码.

# AOP



# AOP 术语

- 切面 (Aspect): 横切关注点 (跨越应用程序多个模块的功能) 被模块化的特殊对象
- 通知 (Advice): 切面必须要完成的工作
- 目标 (Target): 被通知的对象
- 代理 (Proxy): 向目标对象应用通知之后创建的对象
- 连接点 (Joinpoint): 程序执行的某个特定位置: 如类某个方法调用前、调用后、方法抛出异常后等。连接点由两个信息确定: 方法表示的程序执行点; 相对点表示的方位。例如 `ArithmeticCalculator#add()` 方法执行前的连接点, 执行点为 `ArithmeticCalculator#add()`; 方位为该方法执行前的位置
- 切点 (pointcut): 每个类都拥有多个连接点: 例如 `ArithmeticCalculator` 的所有方法实际上都是连接点, 即连接点是程序类中客观存在的事务。AOP 通过切点定位到特定的连接点。类比: 连接点相当于数据库中的记录, 切点相当于查询条件。切点和连接点不是一对一的关系, 一个切点匹配多个连接点, 切点通过 `org.springframework.aop.Pointcut` 接口进行描述, 它使用类和方法作为连接点的查询条件。



# Spring AOP

- **AspectJ**: Java 社区里最完整最流行的 AOP 框架.
- 在 Spring2.0 以上版本中, 可以使用基于 AspectJ 注解或基于 XML 配置的 AOP

# 在 Spring 中启用 AspectJ 注解支持

- 要在 Spring 应用中使用 AspectJ 注解, **必须在 classpath 下包含 AspectJ 类库**: aopalliance.jar、aspectj.weaver.jar 和 spring-aspects.jar
- **将 aop Schema 添加到 <beans> 根元素中.**
- 要在 Spring IOC 容器中启用 AspectJ 注解支持, 只要在 Bean 配置文件中定义一个空的 XML 元素 **<aop:aspectj-autoproxy>**
- 当 Spring IOC 容器侦测到 Bean 配置文件中的 <aop:aspectj-autoproxy> 元素时, 会自动为与 AspectJ 切面匹配的 Bean 创建代理.

# 用 AspectJ 注解声明切面

- 要在 Spring 中声明 AspectJ 切面，只需要在 IOC 容器中将切面声明为 Bean 实例。当在 Spring IOC 容器中初始化 AspectJ 切面之后，Spring IOC 容器就会为那些与 AspectJ 切面相匹配的 Bean 创建代理。
- 在 AspectJ 注解中，切面只是一个带有 @Aspect 注解的 Java 类。
- 通知是标注有某种注解的简单的 Java 方法。
- AspectJ 支持 5 种类型的通知注解：
  - + @Before: 前置通知，在方法执行之前执行
  - + @After: 后置通知，在方法执行之后执行
  - + @AfterRunning: 返回通知，在方法返回结果之后执行
  - + @AfterThrowing: 异常通知，在方法抛出异常之后
  - + @Around: 环绕通知，围绕着方法执行

# 前置通知

- 前置通知:在方法执行之前执行的通知
- 前置通知使用 @Before 注解, 并将切入点表达式的值作为注解值.

```
@Aspect  
public class CalculatorLoggingAspect {  
    private Log log = LogFactory.getLog(this.getClass());  
  
    @Before("execution(* ArithmeticCalculator.add(..))")  
    public void logBefore(){  
        log.info("The method add() begins");  
    }  
}
```

标识这个类是一个切面

标识这个方法是个前置通知, 切点表达式表示执行 ArithmeticCalculator 接口的 add() 方法. \* 代表匹配任意修饰符及任意返回值, 参数列表中的 .. 匹配任意数量的参数

标识这个方法是个前置通知, 切点表达式表示执行 ArithmeticCalculator 接口的 add() 方法. \* 代表匹配任意修饰符及任意返回值, 参数列表中的 .. 匹配任意数量的参数



# 利用方法签名编写 AspectJ 切入点表达式

## ■ 最典型的切入点表达式时根据方法的签名来匹配各种方法：

- + execution \* com.atguigu.spring.ArithmeticCalculator.\*(..): 匹配 ArithmeticCalculator 中声明的所有方法, 第一个 \* 代表任意修饰符及任意返回值. 第二个 \* 代表任意方法. .. 匹配任意数量的参数. 若目标类与接口与该切面在同一个包中, 可以省略包名.
- + execution public \* ArithmeticCalculator.\*(..): 匹配 ArithmeticCalculator 接口的所有公有方法.
- + execution public double ArithmeticCalculator.\*(..): 匹配 ArithmeticCalculator 中返回 double 类型数值的方法
- + execution public double ArithmeticCalculator.\*(double, ..): 匹配第一个参数为 double 类型的方法, .. 匹配任意数量任意类型的参数
- + execution public double ArithmeticCalculator.\*(double, double): 匹配参数类型为 double, double 类型的方法.

# 合并切入点表达式

- 在 AspectJ 中，切入点表达式可以通过操作符 &&, ||, ! 结合起来.

```
@Pointcut("execution(* *.add(int, ..)) || execution(* *.sub(int, ..))")
private void loggingOperation(){}

@Before("loggingOperation()")
public void logBefore(JoinPoint joinPoint){
    log.info("The method " + joinPoint.getSignature().getName()
        + "() begins with " + Arrays.toString(joinPoint.getArgs()));
}
```

# 让通知访问当前连接点的细节

- 可以在通知方法中声明一个类型为 `JoinPoint` 的参数. 然后就能访问链接细节. 如方法名称和参数值.

```
@Aspect
public class CalculatorLoggingAspect {
    private Log log = LoggerFactory.getLog(this.getClass());

    @Before("execution(* *.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));
    }
}
```

标识这个方法是个前置通知, 切点表达式表示执行任意类的任意方法. 第一个 \* 代表匹配任意修饰符及任意返回值, 第二个 \* 代表任意类的对象, 第三个 \* 代表任意方法, 参数列表中的 .. 匹配任意数量的参数

# 后置通知

- 后置通知是在连接点完成之后执行的，即连接点返回结果或者抛出异常的时候，下面的后置通知记录了方法的终止。
- 一个切面可以包括一个或者多个通知。

```
@Aspect
public class CalculatorLoggingAspect {
    private Log log = LogFactory.getLog(this.getClass());

    @Before("execution(* *.*(..))")
    public void logBefore(JoinPoint joinPoint){
        log.info("The method " + joinPoint.getSignature().getName()
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));
    }

    @After("execution(* *.*(..))")
    public void logAfter(JoinPoint joinPoint){
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends");
    }
}
```



# 返回通知

- 无论连接点是正常返回还是抛出异常，后置通知都会执行。如果只想在连接点返回的时候记录日志，应使用返回通知代替后置通知。

```
@Aspect
public class CalculatorLoggingAspect {
    private Log log = LogFactory.getLog(this.getClass());

    @Before("execution(* *.*(..))")
    public void logBefore(JoinPoint joinPoint){
        log.info("The method " + joinPoint.getSignature().getName()
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));
    }

    @AfterReturning("execution(* *.*(..))")
    public void logAfterReturning(JoinPoint joinPoint){
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends");
    }
}
```

# 在返回通知中访问连接点的返回值

- 在返回通知中，只要将 **returning** 属性添加到 `@AfterReturning` 注解中，就可以访问连接点的返回值。该属性的值即为用来传入返回值的参数名称。
- 必须在通知方法的签名中添加一个**同名参数**。在运行时，Spring AOP 会通过这个参数传递返回值。
- **原始的切点表达式需要出现在 `pointcut` 属性中**

```
@AfterReturning(pointcut="execution(* *.*(..))", returning="result")
public void logAfterReturning(JoinPoint joinPoint, Object result){
    log.info("The method " + joinPoint.getSignature().getName()
        + "() ends with " + result);
}
```

# 异常通知

- 只在连接点抛出异常时才执行异常通知
- 将 **throwing** 属性添加到 **@AfterThrowing** 注解中，也可以访问连接点抛出的异常。Throwable 是所有错误和异常类的超类。所以在异常通知方法可以捕获到任何错误和异常。
- 如果只对某种特殊的异常类型感兴趣，可以将参数声明为其他异常的参数类型。然后通知就只在抛出这个类型及其子类的异常时才被执行。

```
@AfterThrowing(pointcut="execution(* *.*(..))", throwing="e")  
public void logAfterThrowing(JoinPoint joinPoint, ArithmeticException e){  
    log.info("An exception " + e + " has been throwing in "  
            + joinPoint.getSignature().getName() + "()");  
}
```

# 环绕通知

- 环绕通知是所有通知类型中功能最为强大的，能够全面地控制连接点。甚至可以控制是否执行连接点。
- 对于环绕通知来说，连接点的参数类型必须是 `ProceedingJoinPoint`。它是 `JoinPoint` 的子接口，允许控制何时执行，是否执行连接点。
- 在环绕通知中需要明确调用 `ProceedingJoinPoint` 的 `proceed()` 方法来执行被代理的方法。如果忘记这样做就会导致通知被执行了，但目标方法没有被执行。
- 注意：环绕通知的方法需要返回目标方法执行之后的结果，即调用 `joinPoint.proceed()`；的返回值，否则会出现空指针异常



# 环绕通知示例代码

```
@Around("execution(* *.*(..))")
public void logAround(ProceedingJoinPoint joinPoint) throws Throwable{
    log.info("The method " + joinPoint.getSignature().getName()
        + "() begins with " + Arrays.toString(joinPoint.getArgs()));

    try {
        joinPoint.proceed();
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends");
    } catch (Throwable e) {
        log.info("An exception " + e + " has been throwing in "
            + joinPoint.getSignature().getName() + "()");
        throw e;
    }
}
```

# 指定切面的优先级

- 在同一个连接点上应用不止一个切面时，除非明确指定，否则它们的优先级是不确定的。
- 切面的优先级可以通过实现 `Ordered` 接口或利用 `@Order` 注解指定。
- 实现 `Ordered` 接口，`getOrder()` 方法的返回值越小，优先级越高。
- 若使用 `@Order` 注解，序号出现在注解中

```
@Aspect
@Order(0)
public class CalculatorValidationAspect {

@Aspect
@Order(1)
public class CalculatorLoggingAspect {
```

# 重用切入点定义

- 在编写 AspectJ 切面时，可以直接在通知注解中书写切入点表达式。但同一个切入点表达式可能会在多个通知中重复出现。
- 在 AspectJ 切面中，可以通过 `@Pointcut` 注解将一个切入点声明成简单的方法。切入点的方法体通常是空的，因为将切入点定义与应用程序逻辑混在一起是不合理的。
- 切入点方法的访问控制符同时也控制着这个切入点的可见性。如果切入点要在多个切面中共用，最好将它们集中在一个公共的类中。在这种情况下，它们必须被声明为 `public`。在引入这个切入点时，必须将类名也包括在内。如果类没有与这个切面放在同一个包中，还必须包含包名。
- 其他通知可以通过方法名称引入该切入点。

# 重用切入点定义示例代码

```
@Pointcut("execution(* *.*(..))")
private void loggingOperation(){}

@Before("loggingOperation()")
public void logBefore(JoinPoint joinPoint){
    log.info("The method " + joinPoint.getSignature().getName()
        + "() begins with " + Arrays.toString(joinPoint.getArgs()));
}

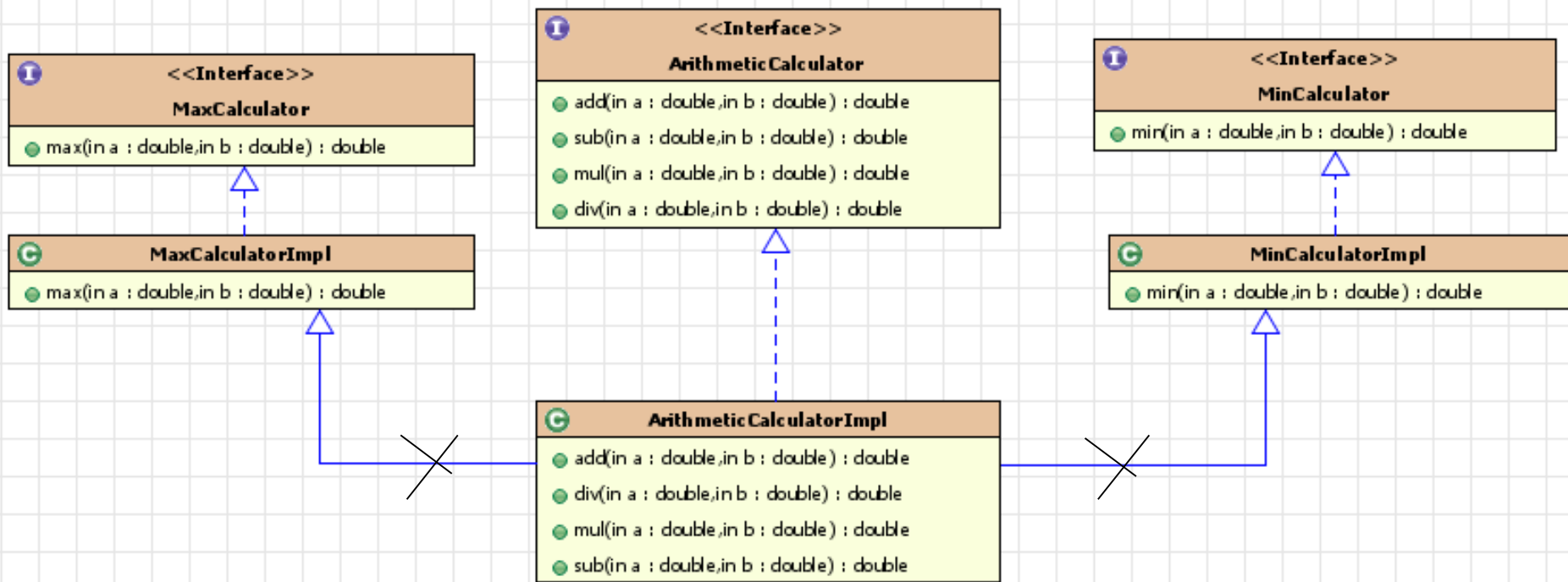
@AfterReturning(pointcut="loggingOperation()", returning="result")
public void logAfterReturning(JoinPoint joinPoint, Object result){
    log.info("The method " + joinPoint.getSignature().getName()
        + "() ends with " + result);
}

@AfterThrowing(pointcut="loggingOperation()", throwing="e")
public void logAfterThrowing(JoinPoint joinPoint, ArithmeticException e){
    log.info("An exception " + e + " has been throwing in "
        + joinPoint.getSignature().getName() + "()");
}
```



# 引入通知

- 引入通知是一种特殊的通知类型。它通过为接口提供实现类，允许对象动态地实现接口，就像对象已经在运行时扩展了实现类一样。



# 引入通知

- 引入通知可以使用两个实现类 `MaxCalculatorImpl` 和 `MinCalculatorImpl`, 让 `ArithmeticCalculatorImpl` 动态地实现 `MaxCalculator` 和 `MinCalculator` 接口. 而这与从 `MaxCalculatorImpl` 和 `MinCalculatorImpl` 中实现多继承的效果相同. 但却不需要修改 `ArithmeticCalculatorImpl` 的源代码
- 引入通知也必须在切面中声明
- 在切面中, 通过为任意字段添加 `@DeclareParents` 注解来引入声明.
- 注解类型的 `value` 属性表示哪些类是当前引入通知的目标. `value` 属性值也可以是一个 AspectJ 类型的表达式, 以将一个即可引入到多个类中. `defaultImpl` 属性中指定这个接口使用的实现类

# 引入通知示例代码

```
@Aspect
public class CalculatorLoggingAspect implements Ordered{
    private Log log = LogFactory.getLog(this.getClass());

    @DeclareParents(value="* *.Arithmetic*", defaultImpl=MaxCalculatorImpl.class)
    private MaxCalculator maxCalculator;

    @DeclareParents(value="* *.Arithmetic*", defaultImpl=MinCalculatorImpl.class)
    private MinCalculator minCalculator;
```

```
MinCalculator minCalculator = (MinCalculator)
    ctx.getBean("airthmeticCalculator");
minCalculator.min(1, 2);
```

# 用基于 XML 的配置声明切面

- 除了使用 AspectJ 注解声明切面，Spring 也支持在 Bean 配置文件中声明切面。这种声明是通过 aop schema 中的 XML 元素完成的。
- 正常情况下，**基于注解的声明要优先于基于 XML 的声明。**通过 AspectJ 注解，切面可以与 AspectJ 兼容，而基于 XML 的配置则是 Spring 专有的。由于 AspectJ 得到越来越多的 AOP 框架支持，所以以注解风格编写的切面将会有更多重用的机会。



# 基于 XML ----- 声明切面

- 当使用 XML 声明切面时，需要在 `<beans>` 根元素中导入 aop Schema
- 在 Bean 配置文件中，所有的 Spring AOP 配置都必须定义在 `<aop:config>` 元素内部。对于每个切面而言，都要创建一个 `<aop:aspect>` 元素来为具体的切面实现引用后端 Bean 实例。
- 切面 Bean 必须有一个标示符，供 `<aop:aspect>` 元素引用

# 声明切面的实例代码

```
<bean id="calculatorLoggingAspect"  
      class="org.simpleit.CalculatorLoggingAspect"></bean>  
  
<bean id="calculatorValidationAspect"  
      class="org.simpleit.CalculatorValidationAspect"></bean>  
  
<aop:config>  
  <aop:aspect id="loggingAspect"  
    ref="calculatorLoggingAspect"></aop:aspect>  
  
  <aop:aspect id="validationAspect"  
    ref="calculatorValidationAspect"></aop:aspect>  
</aop:config>
```



# 基于 XML ----- 声明切入点

- 切入点使用 `<aop:pointcut>` 元素声明
- 切入点必须定义在 `<aop:aspect>` 元素下，或者直接定义在 `<aop:config>` 元素下。
  - + 定义在 `<aop:aspect>` 元素下：只对当前切面有效
  - + 定义在 `<aop:config>` 元素下：对所有切面都有效
- 基于 XML 的 AOP 配置不允许在切入点表达式中用名称引用其他切入点。

# 声明切入点的示例代码

```
<aop:config>  
  <aop:pointcut id="testOperation"  
    expression="execution(* org.simpleit.bean.Arithmetic*.*(..))"/>  
  
  <aop:aspect id="loggingAspect"  
    ref="calculatorLoggingAspect">  
  </aop:aspect>  
  
  <aop:aspect id="validationAspect"  
    ref="calculatorValidationAspect">  
  </aop:aspect>  
</aop:config>
```



# 基于 XML ----- 声明通知

- 在 aop Schema 中，每种通知类型都对应一个特定的 XML 元素.
- 通知元素需要使用 `<pointcut-ref>` 来引用切入点，或用 `<pointcut>` 直接嵌入切入点表达式. `method` 属性指定切面类中通知方法的名称.

# 声明通知示例代码

```
<aop:config>
  <aop:pointcut id="testOperation"
    expression="execution(* org.simpleit.bean.Arithmetic*.*(..))"/>

  <aop:aspect id="loggingAspect"
    ref="calculatorLoggingAspect">
    <aop:after method="logBefore"
      pointcut-ref="testOperation"/>
  </aop:aspect>

  <aop:aspect id="validationAspect"
    ref="calculatorValidationAspect">
    <aop:before method="validateBefore"
      pointcut-ref="testOperation"/>
  </aop:aspect>
</aop:config>
```

# 声明引入

- 可以利用 `<aop:declare-parents>` 元素在切面内部声明引入

```
<aop:aspect id="loggingAspect"  
  ref="calculatorLoggingAspect">  
  <aop:after method="logBefore"  
    pointcut-ref="testOperation"/>  
  <aop:declare-parents
```

```
    types-matching="org.simpleit.bean.Arithmetic*"  
    implement-interface="org.simpleit.bean.MinCalculator"  
    default-impl="org.simpleit.bean.MinCalculatorImpl"/>  
</aop:declare-parents>  
</aop:aspect>
```



# Spring 对 JDBC 的支持





# JdbcTemplate 简介

- 为了使 JDBC 更加易于使用，Spring 在 JDBC API 上定义了一个抽象层，以此建立一个 JDBC 存取框架。
- 作为 Spring JDBC 框架的核心，**JDBC 模板**的设计目的是为不同类型的 JDBC 操作提供**模板方法**。每个模板方法都能控制整个过程，并允许覆盖过程中的特定任务。通过这种方式，可以在尽可能保留灵活性的情况下，将数据库存取的工作量降到最低。

# 书库

- update

## ■ 批量更新数据库：

```
public int[] batchUpdate(String sql,  
                        List<Object[]> batchArgs)
```

# 使用 JdbcTemplate 查询数据库

## ■ 查询单行:

### **queryForObject**

```
public <T> T queryForObject(String sql,  
                           ParameterizedRowMapper<T> rm,  
                           Object... args)  
    throws DataAccessException
```

## ■ 便利的 BeanPropertyRowMapper 实现

[org.springframework.jdbc.core.simple](#)

**Class [ParameterizedBeanPropertyRowMapper](#)<T>**

[java.lang.Object](#)

└ [org.springframework.jdbc.core.BeanPropertyRowMapper](#)

└ [org.springframework.jdbc.core.simple.ParameterizedBeanPropertyRowMapper](#)<T>

# 使用 JdbcTemplate 查询数据库

## ■ 查询多行:

### **query**

```
public <T> List<T> query(String sql,  
                           ParameterizedRowMapper<T> rm,  
                           Object... args)  
    throws DataAccessException
```

## ■ 单值查询:

### **queryForObject**

```
public <T> T queryForObject(String sql,  
                             Class<T> requiredType,  
                             Object... args)  
    throws DataAccessException
```



# 简化 JDBC 模板查询

- 每次使用都创建一个 `JdbcTemplate` 的新实例，这种做法效率很低下.
- `JdbcTemplate` 类被设计成为线程安全的，所以可以再 IOC 容器中声明它的单个实例，并将这个实例注入到所有的 DAO 实例中.
- `JdbcTemplate` 也利用了 Java 1.5 的特定(自动装箱，泛型，可变长度等)来简化开发
- Spring JDBC 框架还提供了一个 `JdbcDaoSupport` 类来简化 DAO 实现. 该类声明了 `jdbcTemplate` 属性，它可以从 IOC 容器中注入，或者自动从数据源中创建.

# 注入 JDBC 模板示例代码

```
<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="personDAO"
      class="org.simpleit.jdbc.PersonDAO">
  <property name="jdbcTemplate" ref="jdbcTemplate"></property>
</bean>
```

# 扩展 JdbcDaoSupport 示例代码

```
public class PersonDAO extends JdbcDaoSupport
```

```
<bean id="personDAO"  
      class="org.simpleit.jdbc.PersonDAO">  
  <property name="dataSource" ref="dataSource" />  
</bean>
```

# 在 JDBC 模板中使用具名参数

- 在经典的 JDBC 用法中, SQL 参数是用占位符 ? 表示, 并且受到位置的限制. 定位参数的问题在于, 一旦参数的顺序发生变化, 就必须改变参数绑定.
- 在 Spring JDBC 框架中, 绑定 SQL 参数的另一种选择是使用具名参数(named parameter).
- 具名参数: SQL 按名称(以冒号开头)而不是按位置进行指定. 具名参数更易于维护, 也提升了可读性. 具名参数由框架类在运行时用占位符取代
- 具名参数只在 `NamedParameterJdbcTemplate` 中得到支持



# 在 JDBC 模板中使用具名参数

- 在 SQL 语句中使用具名参数时，可以在一个 Map 中提供参数值，参数名为键
- 也可以使用 SqlParameterSource 参数
- 批量更新时可以提供 Map 或 SqlParameterSource 的数组

## update

```
public int update(String sql,  
                  Map args)  
    throws DataAccessException
```

## batchUpdate

```
public int[] batchUpdate(String sql,  
                          Map[] batchValues)
```

## update

```
public int update(String sql,  
                  SqlParameterSource args)  
    throws DataAccessException
```

## batchUpdate

```
public int[] batchUpdate(String sql,  
                          SqlParameterSource[] batchArgs)
```



Spring

中的事务管理

# 事务简介

- 事务管理是企业级应用程序开发中必不可少的技术，**用来确保数据的完整性和一致性。**
- 事务就是一系列的动作，它们被当做一个单独的工作单元。这些动作要么全部完成，要么全部不起作用
- 事务的四个关键属性(ACID)
  - + **原子性(atomicity)**：事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成要么完全不起作用。
  - + **一致性(consistency)**：一旦所有事务动作完成，事务就被提交。数据和资源就处于一种满足业务规则的一致性状态中。
  - + **隔离性(isolation)**：可能有许多事务会同时处理相同的数据，因此每个事物都应该与其他事务隔离开来，防止数据损坏。
  - + **持久性(durability)**：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响。通常情况下，事务的结果被写到持久化存储器中。

# 事务管理的问题

## ■ 问题:

- + 必须为不同的方法重写类似的样板代码
- + 这段代码是特定于 JDBC 的, 一旦选择类其它数据库存取技术, 代码需要作出相应的修改

```
public void purchase(String isbn, String username){
    Connection conn = null;

    try {
        conn = dataSource.getConnection();
        conn.setAutoCommit(false);

        //...

        conn.commit();
    } catch (SQLException e) {
        e.printStackTrace();
        if(conn != null){
            try {
                conn.rollback();
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
        }
        throw new RuntimeException(e);
    } finally{
        if(conn != null){
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```



# Spring 中的事务管理

- 作为企业级应用程序框架，Spring 在不同的事务管理 API 之上定义了一个抽象层。而应用程序开发人员不必了解底层的事务管理 API，就可以使用 Spring 的事务管理机制。
- Spring 既支持编程式事务管理，也支持声明式的事务管理。
- **编程式事务管理**：将事务管理代码嵌入到业务方法中来控制事务的提交和回滚。在编程式管理事务时，必须在每个事务操作中包含额外的事务管理代码。
- **声明式事务管理**：大多数情况下比编程式事务管理更好用。它将事务管理代码从业务方法中分离出来，以声明的方式来实现事务管理。事务管理作为一种横切关注点，可以通过 AOP 方法模块化。Spring 通过 Spring AOP 框架支持声明式事务管理。

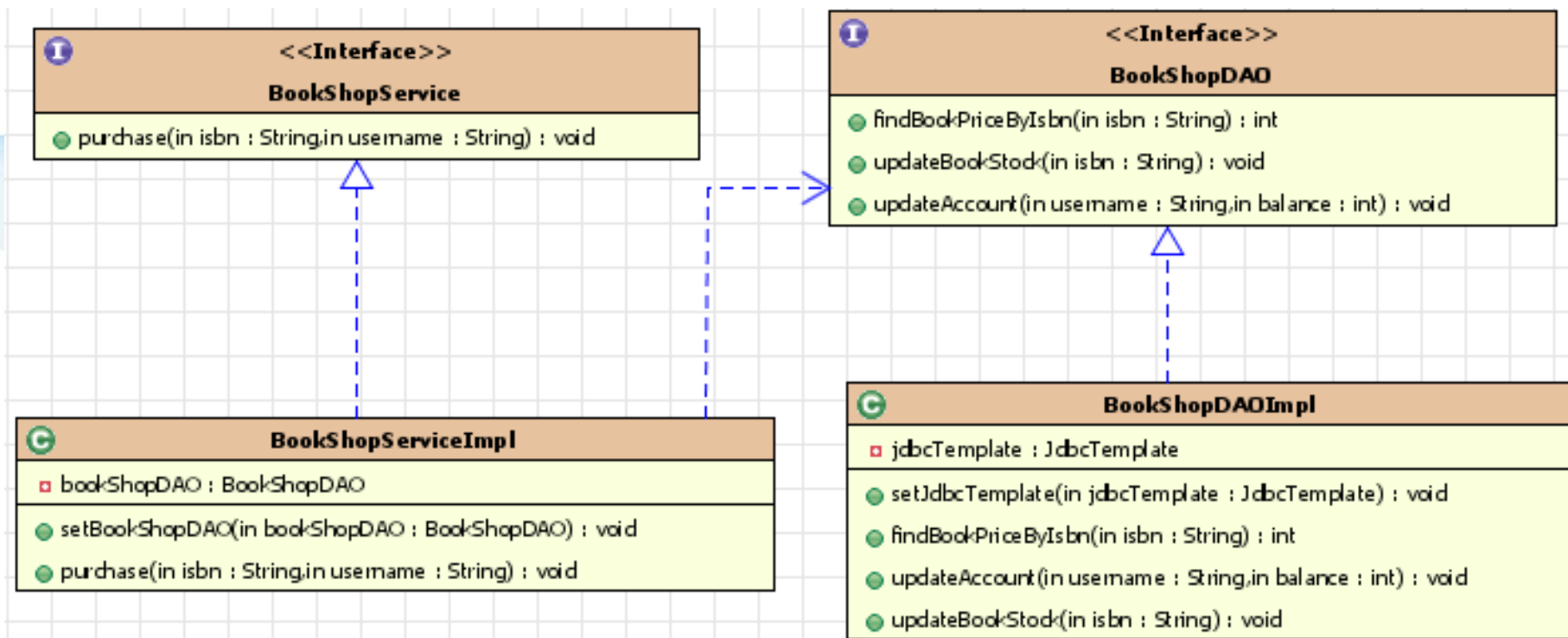
# Spring 中的事务管理器

- Spring 从不同的事务管理 API 中抽象了一整套的事务机制. 开发人员不必了解底层的事务 API, 就可以利用这些事务机制. **有了这些事务机制, 事务管理代码就能独立于特定的事务技术了.**
- Spring 的核心事务管理抽象是 org.springframework.transaction **Interface PlatformTransactionManager** 它为事务管理封装了一组独立于技术的方法. 无论使用 Spring 的哪种事务管理策略(编程式或声明式), 事务管理器都是必须的.

# Spring 中的事务管理器的不同实现

- org.springframework.jdbc.datasource  
**Class DataSourceTransactionManager** : 在应用程序中只需要处理一个数据源, 而且通过 JDBC 存取
- org.springframework.transaction.jta  
**Class JtaTransactionManager** : 在 JavaEE 应用服务器上  
用 JTA (Java Transaction API) 进行事务管理
- org.springframework.orm.hibernate3  
**Class HibernateTransactionManager** : 用 Hibernate 框架存取数据库
- .....
- 事务管理器以普通的 Bean 形式声明在 Spring  
IOC 容器中

# 需求





# 数据表中的数据

Account 表

username ▲	balance
Tom	30

Book 表

isbn ▲	book_name	price
0001	Java	50

Book\_STOCK 表

isbn ▲	stock
0001	10

# 用事务通知声明式地管理事务

- 事务管理是一种横切关注点
- 为了在 Spring 2.x 中启用声明式事务管理，可以通过 tx Schema 中定义的 **<tx:advice> 元素声明事务通知**，为此必须事先将这个 Schema 定义添加到 <beans> 根元素中去。
- 声明了事务通知后，就需要将它与切入点关联起来。由于事务通知是在 <aop:config> 元素外部声明的，所以它无法直接与切入点产生关联。所以必须在 **<aop:config> 元素中声明一个增强器通知与切入点关联起来**。
- 由于 Spring AOP 是基于代理的方法，所以只能增强公共方法。因此，**只有公有方法才能通过 Spring AOP 进行事务管理**。

# 用事务通知声明式地管理事务示例代码

```
<bean id="bookShopService"
      class="org.simpleit.transaction.BookShopServiceImpl">
  <property name="bookShopDAO" ref="bookDAO"/>
</bean>
```

声明事务管理器

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"></property>
</bean>
```

声明事务通知

```
<tx:advice id="bookShopTxAdvice"
           transaction-manager="transactionManager">
</tx:advice>
```

声明 事务通知需要通知方法(即需要进行事务管理的方法)

```
<aop:config>
  <aop:pointcut expression="execution(* *.BookShopService.*(..))"
                id="bookShopOperation"/>
  <aop:advisor advice-ref="bookShopTxAdvice"
               pointcut-ref="bookShopOperation"/>
</aop:config>
```

# 用 @Transactional 注解声明式地管理事务

- 除了在带有切入点，通知和增强器的 Bean 配置文件中声明事务外，Spring 还允许简单地用 @Transactional 注解来**标注事务方法**.
- **为了将方法定义为支持事务处理的，可以为方法添加 @Transactional 注解**. 根据 Spring AOP 基于代理机制，**只能标注公有方法**.
- 可以在方法或者**类级别上**添加 @Transactional 注解. 当把这个注解应用到类上时，这个类中的所有公共方法都会被定义成支持事务处理的.
- 在 Bean 配置文件中只需要启用 `<tx:annotation-driven>` 元素，并为之指定事务管理器就可以了.
- 如果事务处理器的名称是 `transactionManager`，就可以在 `<tx:annotation-driven>` 元素中省略 `transaction-manager` 属性. 这个元素会自动检测该名称的事务处理器.



# 用 @Transactional 注解声明式地管理事务配置文件示例代码

```
<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"></property>
</bean>

<context:component-scan base-package="org.simpleit.transaction_1"/>

<tx:annotation-driven/>
```

# 事务传播属性

- 当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。
- 事务的传播行为可以由传播属性指定。Spring 定义了 7 种类传播行为。

# Spring 支持的事务传播行为

传播属性	描述
REQUIRED	如果有事务在运行，当前的方法就在这个事务内运行，否则，就启动一个新的事务，并在自己的事务内运行
REQUIRED_NEW	当前的方法必须启动新事务，并在它自己的事务内运行。如果有事务正在运行，应该将它挂起
SUPPORTS	如果有事务在运行，当前的方法就在这个事务内运行。否则它可以不运行在事务中。
NOT_SUPPORTED	当前的方法不应该运行在事务中。如果有运行的事务，将它挂起
MANDATORY	当前的方法必须运行在事务内部，如果没有正在运行的事务，就抛出异常
NEVER	当前的方法不应该运行在事务中。如果有运行的事务，就抛出异常
NESTED	如果有事务在运行，当前的方法就应该在这个事务的嵌套事务内运行。否则，就启动一个新的事务，并在它自己的事务内运行。

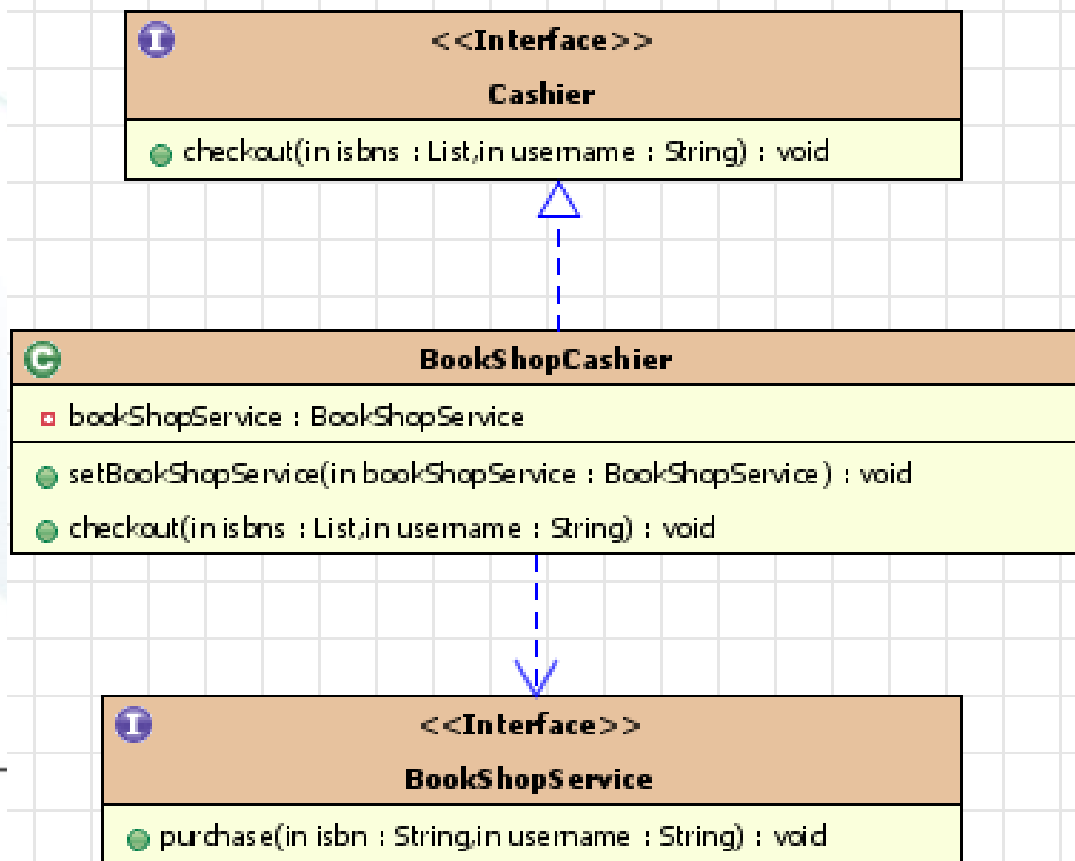
# 需求

- 新定义 Cashier 接口：表示客户的结账操作
- 修改数据表信息如下，目的是用户 Tom 在结账时，余额只能支付第一本书，不够支付第二本书：

username ▲	balance
Tom	60

isbn ▲	book_name	price
0002	Oracle	80
0001	Java	50

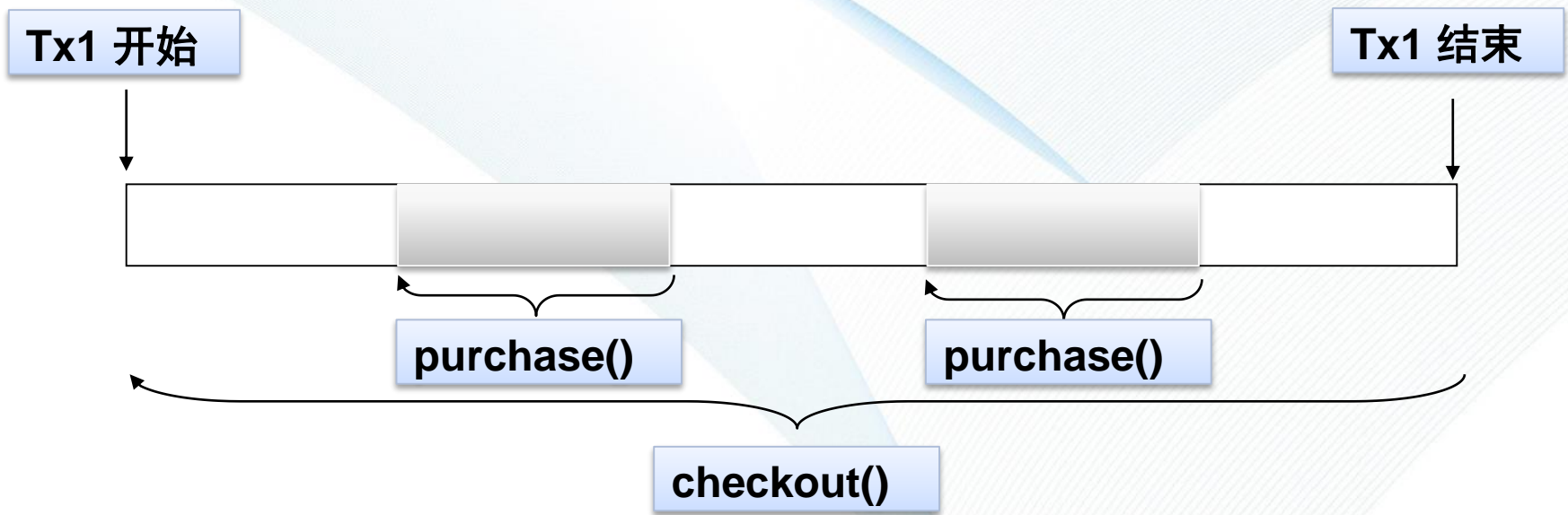
isbn ▲	stock
0001	10
0002	10





# REQUIRED 传播行为

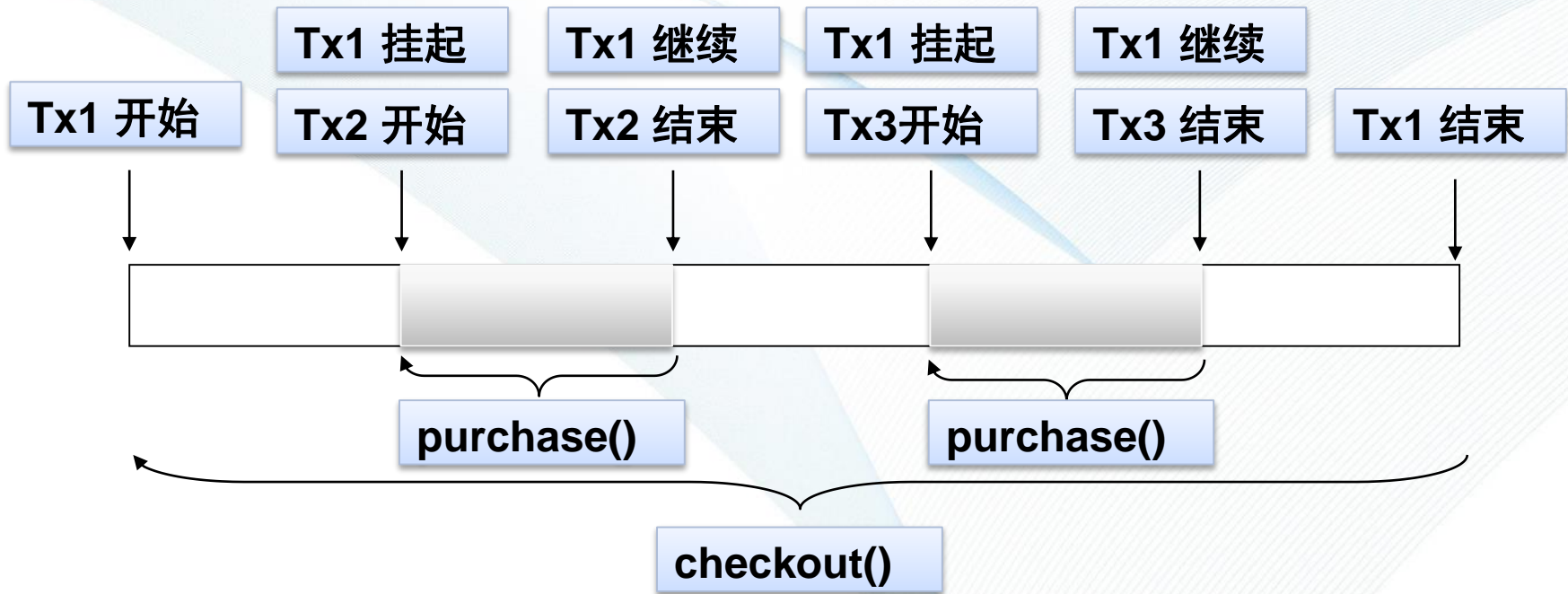
- 当 bookService 的 purchase() 方法被另一个事务方法 checkout() 调用时，它默认会在现有的事务内运行。这个默认的传播行为就是 REQUIRED。因此在 checkout() 方法的开始和终止边界内只有一个事务。这个事务只在 checkout() 方法结束的时候被提交，结果用户一本书都买不了
- 事务传播属性可以在 @Transactional 注解的 propagation 属性中定义



# REQUIRES\_NEW 传播行为

- 另一种常见的传播行为是 REQUIRES\_NEW. 它表示该方法必须启动一个新事务，并在自己的事务内运行. 如果有事务在运行，就应该先挂起它.

```
@Transactional(propagation=Propagation.REQUIRES_NEW)  
public void purchase(String isbn, String username) {
```



# 在 Spring 2.x 事务通知中配置传播属性

- 在 Spring 2.x 事务通知中，可以像下面这样在 `<tx:method>` 元素中设定传播事务属性

```
<tx:advice id="bookShopTxAdvice"  
  transaction-manager="transactionManager">  
  <tx:attributes>  
    <tx:method name="purchase" propagation="REQUIRES_NEW"/>  
  </tx:attributes>  
</tx:advice>
```

# 并发事务所导致的问题

- 当同一个应用程序或者不同应用程序中的多个事务在同一个数据集上并发执行时，可能会出现许多意外的问题
- 并发事务所导致的问题可以分为下面三种类型：
  - + 脏读：对于两个事物 T1, T2, T1 读取了已经被 T2 更新但还没有被提交的字段。之后，若 T2 回滚，T1读取的内容就是临时且无效的。
  - + 不可重复读：对于两个事物 T1, T2, T1 读取了一个字段，然后 T2 更新了该字段。之后，T1再次读取同一个字段，值就不同了。
  - + 幻读：对于两个事物 T1, T2, T1 从一个表中读取了一个字段，然后 T2 在该表中插入了一些新的行。之后，如果 T1 再次读取同一个表，就会多出几行。



# 事务的隔离级别

- 从理论上来说，事务应该彼此完全隔离，以避免并发事务所导致的问题。然而，那样会对性能产生极大的影响，因为事务必须按顺序运行。
- 在实际开发中，为了提升性能，事务会以较低的隔离级别运行。
- 事务的隔离级别可以通过隔离事务属性指定

# Spring 支持的事务隔离级别

隔离级别	描述
DEFAULT	使用底层数据库的默认隔离级别。对于大多数数据库来说，默认隔离级别都是 READ_COMMITTED
READ_UNCOMMITTED	允许事务读取未被其他事物提交的变更。脏读，不可重复读和幻读的问题都会出现
READ_COMMITTED	只允许事务读取已经被其它事务提交的变更。可以避免脏读，但不可重复读和幻读问题仍然可能出现
REPEATABLE_READ	确保事务可以多次从一个字段中读取相同的值。在这个事务持续期间，禁止其他事物对这个字段进行更新。可以避免脏读和不可重复读，但幻读的问题仍然存在。
SERIALIZABLE	确保事务可以从一个表中读取相同的行。在这个事务持续期间，禁止其他事务对该表执行插入，更新和删除操作。所有并发问题都可以避免，但性能十分低下。

- 事务的隔离级别要得到底层数据库引擎的支持，而不是应用程序或者框架的支持。
- Oracle 支持的 2 种事务隔离级别：READ\_COMMITTED ， SERIALIZABLE
- Mysql 支持 4 中事务隔离级别。

# 设置隔离事务属性

- 用 `@Transactional` 注解声明式地管理事务时可以在 `@Transactional` 的 `isolation` 属性中设置隔离级别。

```
@Transactional(propagation=Propagation.REQUIRES_NEW,  
                isolation=Isolation.READ_COMMITTED)  
public void purchase(String isbn, String username) {
```

- 在 Spring 2.x 事务通知中，可以在 `<tx:method>` 元素中指定隔离级别

```
<tx:advice id="bookShopTxAdvice"  
  transaction-manager="transactionManager">  
  <tx:attributes>  
    <tx:method name="purchase"  
      propagation="REQUIRES_NEW"  
      isolation="READ_COMMITTED"/>  
  </tx:attributes>  
</tx:advice>
```

# 设置回滚事务属性

- **默认情况下只有未检查异常** (RuntimeException和Error类型的异常) **会导致事务回滚**. 而受检查异常不会.
- 事务的回滚规则可以通过 @Transactional 注解的 rollbackFor 和 noRollbackFor 属性来定义. 这两个属性被声明为 Class[] 类型的, 因此可以为这两个属性指定多个异常类.
  - + rollbackFor: 遇到时必须进行回滚
  - + noRollbackFor: 一组异常类, 遇到时必须不回滚

```
@Transactional(propagation=Propagation.REQUIRES_NEW,  
                isolation=Isolation.READ_COMMITTED,  
                rollbackFor={ IOException.class, SQLException.class },  
                noRollbackFor=ArithmeticException.class)  
public void purchase(String isbn, String username) {
```



# 设置回滚事务属性

- 在 Spring 2.x 事务通知中，可以在 <tx:method> 元素中指定回滚规则。如果有不止一种异常，用逗号分隔。

```
<tx:advice id="bookShopTxAdvice"
  transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="purchase"
      propagation="REQUIRES_NEW"
      isolation="READ_COMMITTED"
      rollback-for="java.io.IOException, java.sql.SQLException"
      no-rollback-for="java.lang.ArithmeticException"/>
  </tx:attributes>
</tx:advice>
```

# 超时和只读属性

- 由于事务可以在行和表上获得锁， 因此长事务会占用资源， 并对整体性能产生影响.
- 如果一个事物只读取数据但不做修改， 数据库引擎可以对这个事务进行优化.
- **超时事务属性**： 事务在强制回滚之前可以保持多久. 这样可以防止长期运行的事务占用资源.
- **只读事务属性**： 表示这个事务只读取数据但不更新数据， 这样可以帮助数据库引擎优化事务.

# 设置超时和只读事务属性

- 超时和只读属性可以在 `@Transactional` 注解中定义. 超时属性以秒为单位来计算.

```
@Transactional(propagation=Propagation.REQUIRES_NEW,  
    isolation=Isolation.READ_COMMITTED,  
    rollbackFor={IOException.class, SQLException.class},  
    noRollbackFor=ArithmeticException.class,  
    readOnly=true,  
    timeout=30)  
public void purchase(String isbn, String username) {
```

- 在 Spring 2.x 事务通知中, 超时和只读属性可以在 `<tx:method>` 元素中进行指定.

```
<tx:advice id="bookShopTxAdvice"  
    transaction-manager="transactionManager">  
    <tx:attributes>  
        <tx:method name="purchase"  
            propagation="REQUIRES_NEW"  
            isolation="READ_COMMITTED"  
            rollback-for="java.io.IOException, java.sql.SQLException"  
            no-rollback-for="java.lang.ArithmeticException"  
            timeout="30"  
            read-only="true"/>  
    </tx:attributes>  
</tx:advice>
```



# Spring 整合 Hibernate





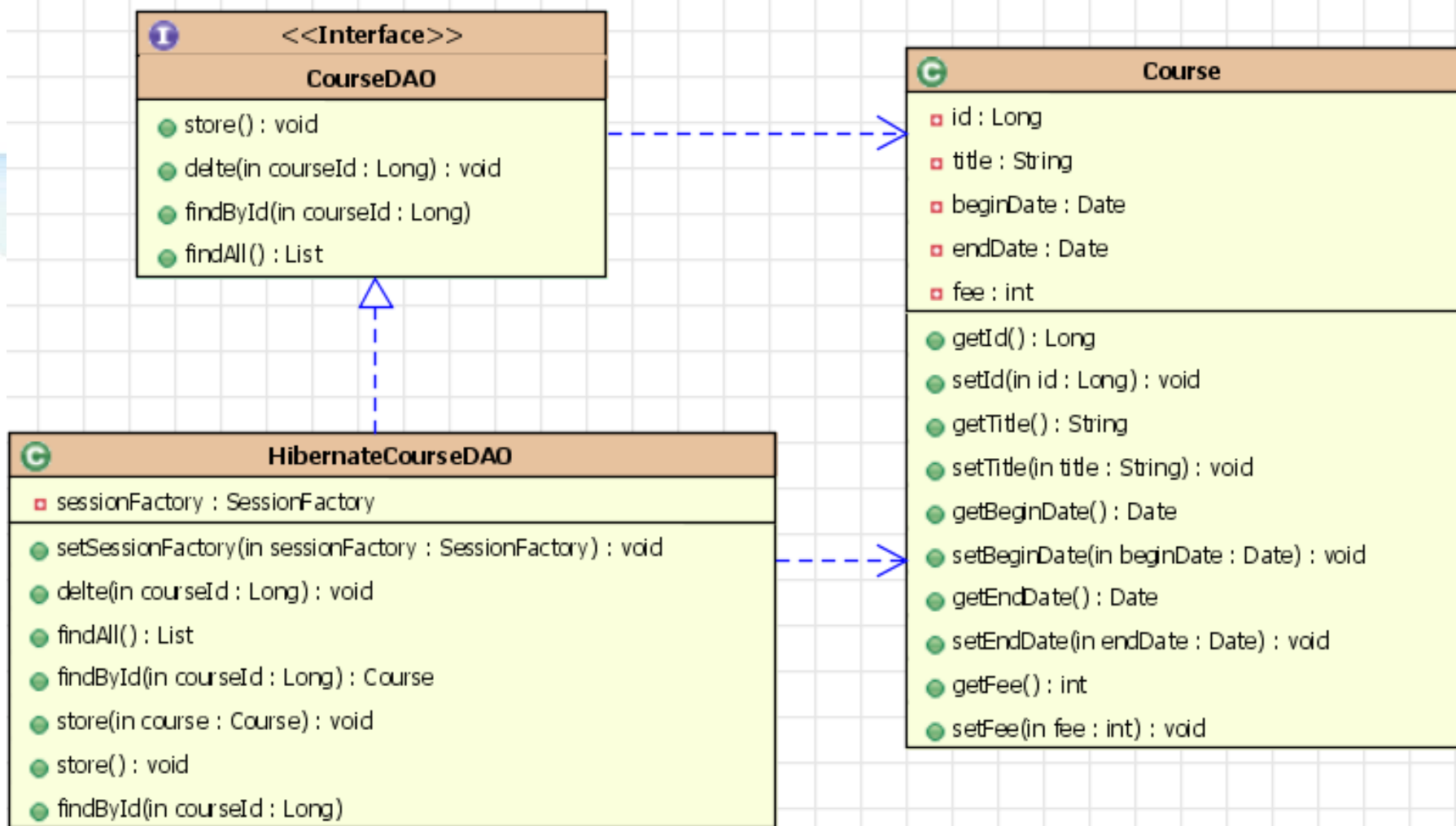
# Spring 整合 Hibernate

- Spring 支持大多数流行的 ORM 框架，包括 Hibernate JDO, TopLink, Ibatis 和 JPA。
- **Spring 对这些 ORM 框架的支持是一致的**，因此可以把和 Hibernate 整合技术应用到其他 ORM 框架上。
- Spring 2.0 同时支持 Hibernate 2.x 和 3.x. 但 Spring 2.5 只支持 Hibernate 3.1 或更高版本

# 在 Spring 中配置 SessionFactory

- 对于 Hibernate 而言，必须从原生的 Hibernate API 中构建 SessionFactory. 此外，应用程序也无法利用 Spring 提供的数据存储机制(例如：Spring 的事务管理机制)
- Spring 提供了对应的工厂 Bean，可以用单实例的形式在 IOC 容器中创建 SessionFactory 实例.

# 需求



# 在 Spring 中配置 SessionFactory (1)

- 可以利用 **LocalSessionFactoryBean** 工厂 Bean, 声明一个使用 XML 映射文件的 SessionFactory 实例.
- 需要为该工厂 Bean 指定 configLocation 属性来加载 Hibernate 配置文件.

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="configLocation" value="hibernate.cfg.xml"/>
</bean>
```

```
<bean id="courseDAO"
      class="org.simpleit.HibernateCourseDAO">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```



# 在 Spring 中配置 SessionFactory (2)

- 如果在 Spring IOC 容器中配置数据源. 可以将该数据源注入到 LocalSessionFactoryBean 的 dataSource 属性中. 该属性可以指定的数据源会覆盖掉 Hibernate 配置文件里的数据库配置.

# 在 Spring 中配置 SessionFactory (2)

```
<context:property-placeholder location="C3P0_config.properties"/>

<bean id="dataSource"
      class="com.mchange.v2.c3p0.ComboPooledDataSource">
  <property name="user" value="${user}"/>
  <property name="password" value="${password}"/>
  <property name="jdbcUrl" value="${jdbcUrl}"/>
  <property name="driverClass" value="${driverClass}"/>

  <property name="checkoutTimeout" value="${checkoutTimeout}"/>
  <property name="idleConnectionTestPeriod" value="${idleConnectionTestPeriod}"/>
  <property name="initialPoolSize" value="${initialPoolSize}"/>
  <property name="maxIdleTime" value="${maxIdleTime}"/>

  <property name="maxPoolSize" value="${checkoutTimeout}"></property>
  <property name="minPoolSize" value="${minPoolSize}"></property>
  <property name="maxStatements" value="${maxStatements}"></property>
</bean>

<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="configLocation" value="hibernate.cfg.xml"/>
  <property name="dataSource" ref="dataSource"></property>
</bean>
```

# 在 Spring 中配置 SessionFactory (3)

- 可以将所有配置合并到 LocalSessionFactoryBean 中, 从而忽略 Hibernate 配置文件.
- 可以在 LocalSessionFactoryBean 的 **mappingResources** 属性中指定 XML 映射文件的位置. 该属性为 String[] 类型. 因此可以指定一组映射文件.
- 在 hibernateProperties 属性中指定数据库方言等.

# 在 Spring 中配置 SessionFactory (3)

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource"></property>

  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
      <prop key="hibernate.show_sql">true</prop>
      <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
  </property>

  <property name="mappingResources">
    <list>
      <value>org/simpleit/Course.hbm.xml</value>
    </list>
  </property>
</bean>
```



# 用 Spring 的 ORM 模板持久化对象

- 在单独使用 ORM 框架时，必须为每个 DAO 操作重复某些常规任务。例如：打开关闭 Session 对象；启动，提交，回滚事务等。
- 同 JDBC 一样，Spring 采取了相同的方法 ----- 定义 **模板类和 DAO 支持类**来简化 ORM 框架的使用。而且 Spring 在不同的事务管理 API 之上定义了一个事务抽象层。对于不同的 ORM 框架，只需要选择相应的事务管理器实现。

# Spring 对不同数据存储策略的支持类

支持类	JDBC	Hibernate
模板类	JdbcTemplate	HibernateTemplate
DAO 支持类	JdbcDaoSupport	HibernateDaoSupport
事务管理类	DataSourceTransactionManager	HibernateTransactionManager

- HibernateTemplate 确保了 Hibernate 会话能够正确地打开和关闭.
- HibernateTemplate 也会让原生的 Hibernate 事务参与到 Spring 的事务管理体系中来. 从而利用 Spring 的声明式事务管理事务.

# 使用 Hibernate 模板

- `HibernateTemplate` 中的模板方法管理会话和事务. 如果在一个支持事务的 DAO 方法中有多个 Hibernate 操作, 模板方法可以确保它们会在同一个会话和事务中运行. 因此没有必要为了会话和事务管理去和 Hibernate API 打交道.
- 通过为 DAO 方法添加 `@Transactional` 注解将其声明为受事务管理的.
- `HibernateTemplate` 类是线程安全的, 因此可以在 Bean 配置文件中只声明一个实例, 并将该实例注入到所有的 Hibernate DAO 中.

# 使用 Hibernate 模板示例代码

```
private HibernateTemplate hibernateTemplate;

public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
    this.hibernateTemplate = hibernateTemplate;
}

@Override
@Transactional
public void delte(Long courseId) {
    Course course =
        (Course) hibernateTemplate.get(Course.class, courseId);
    hibernateTemplate.delete(course);
}

@Override
@Transactional(readOnly=true)
public List<Course> findAll() {
    return hibernateTemplate.find("FROM Count");
}

@Override
@Transactional(readOnly=true)
public Course findById(Long courseId) {
    return (Course) hibernateTemplate.get(Course.class, courseId);
}

@Override
@Transactional
public void store(Course course) {
    hibernateTemplate.saveOrUpdate(course);
}
```



# 使用 Hibernate 模板示例代码

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"></property>
</bean>

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

<bean id="hibernateTemplate"
      class="org.springframework.orm.hibernate3.HibernateTemplate">
  <property name="sessionFactory" ref="sessionFactory"></property>
</bean>

<bean id="courseDAO_"
      class="org.simpleit.HibernateCourseDAO_">
  <property name="hibernateTemplate" ref="hibernateTemplate"/>
</bean>
```

# 在 HibernateTemplate 中访问 Hibernate 底层 Session

```
hibernateTemplate.execute(new HibernateCallback() {  
    @Override  
    public Object doInHibernate(Session arg0) throws HibernateException,  
        SQLException {  
        //...  
        return null;  
    }  
});
```

# 继承 Hibernate 的 DAO 支持类

- Hibernate DAO 可以通过继承 `HibernateDaoSupport` 来继承 `setSessionFactory()` 和 `setHibernateTemplate()` 方法. 然后, 只要在 DAO 方法中调用 `getHibernateTemplate()` 方法就可以获取到模板实例.
- 如果为 `HibernateDaoSupport` 实现类注入了 `SessionFactory` 实例, 就不需要在为之注入 `HibernateTemplate` 实例了, 因为 `HibernateDaoSupport` 会根据传入的 `SessionFactory` 在其构造器内创建 `HibernateTemplate` 的实例, 并赋给 `hibernateTemplate` 属性

# 用 Hibernate 的上下文 Session 持久化对象

- Spring 的 HibernateTemplate 可以管理会话和事务，简化 DAO 实现. 但使用 HibernateTemplate 意味着DAO 必须依赖于 Spring 的 API
- 代替 HibernateTemplate 的另一种办法是使用 Hibernate 的上下文 Session 对象.
- **Hibernate 上下文 Session 对象和 Spring 的事务管理合作的很好，但此时需保证所有的DAO 方法都支持事务**
- 注意此时不需在 beans.xml 文件中配置，因为 Spring 此时已经开始事务，所以已经在 ThreadLocal 对象中绑定了 Session 对象

```
<prop key="hibernate.current_session_context_class">thread</prop>
```



# 用 Hibernate 的上下文 Session 持久化对象

- 在 Hibernate 会话中调用原生的方法时，抛出的异常依旧是原生的 `HibernateException`.
- 为了保持一致的异常处理方法，即把 Hibernate 异常转换为 Spring 的 `DataAccessException` 异常，那么必须为需要异常转换的 DAO 类添加 `@Repository` 注解.
- 然后在注册一个 `org.springframework.dao.annotation` **`Class PersistenceExceptionTranslationPostProcessor`** 实例，将原生的 Hibernate 异常转换为 Spring 的 `DataAccessException` 层次结构中的数据存取异常. 这个 Bean 后置处理器只为添加了 `@Repository` 注解的 Bean 转换异常.

# Hibernate 上下文相关的 Session(1)

- 从 Hibernate 3 开始, SessionFactory 新增加了 `getCurrentSession()` 方法, 该方法可直接获取“上下文”相关的 Session.
- Hibernate 通过 `CurrentSessionContext` 接口的实现类和配置参数 `hibernate.current_session_context_class` 定义“上下文”
  - + `JTASessionContext`: 根据 JTA 来跟踪和界定 Session 对象.
  - + `ThreadLocalSessionContext`: 通过当前正在执行的线程来跟踪和界定 Session 对象
  - + `ManagedSessionContext`: 通过正在当前执行来跟踪和界定 Session 对象. 但程序需要调用该类的静态方法来绑定 Session 对象, 取消绑定, flush 或者关闭 Session 对象.

# Hibernate 上下文相关的 Session(2)

- 如果使用 ThreadLocalSessionContext 策略, Hibernate 的 Session 会随着 getCurrentSession() 方法自动打开, 随着事务提交自动关闭.
- 若当前应用是基于 JTA 的分布式事务, 通常采用第一种方式; 而对于独立的 Hibernate 应用则使用第二种应用.
- 配置:

- + 根据 JTA 来跟踪和界定 Session 对象:

```
<property name="hibernate.current_session_context_class">thread</property>
```

- + 通过当前正在执行的线程来跟踪和界定 Session 对象:

```
<property name="hibernate.current_session_context_class">jta</property>
```



# 整合 Struts2



---



# 在通用的 web 应用中访问 Spring

- 通过注册 Servlet 监听器 `ContextLoaderListener`, Web 应用程序可以加载 Spring 的 `ApplicationContext` 对象. 这个监听器会将加载好的 `ApplicationContext` 对象保存到 Web 应用程序的 `ServletContext` 中. 随后, Servlet 或可以访问 `ServletContext` 的任意对象就能通过一个辅助方法来访问 Spring 的应用程序上下文了.

# 在通用的 web 应用中访问 Spring 具体实现

- 在 web.xml 文件中注册 Spring 提供的 Servlet 监听器 org.springframework.web.context **Class ContextLoaderListener**，它会在当前 web 应用被加载时将 Spring 的 ApplicationContext 保存到 ServletContext 对象中。
- org.springframework.web.context **Class ContextLoaderListener** 监听器通过查找 web 应用初始化参数 contextConfigLocation 来获取 Bean 配置文件的位置。如果有多个 Bean 配置文件，可以通过逗号或空格进行分隔。contextConfigLocation 的默认值为 /WEB-INF/applicationContext.xml。若实际的文件和默认值一致则可以省略这个 web 应用的初始化参数

# web.xml 文件示例代码

```
<context-param>
    <param-name>contextConfiguration</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>

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

# 在 web 应用程序中访问 Spring 的 ApplicationContext 对象

- 可以通过  
的静态方法

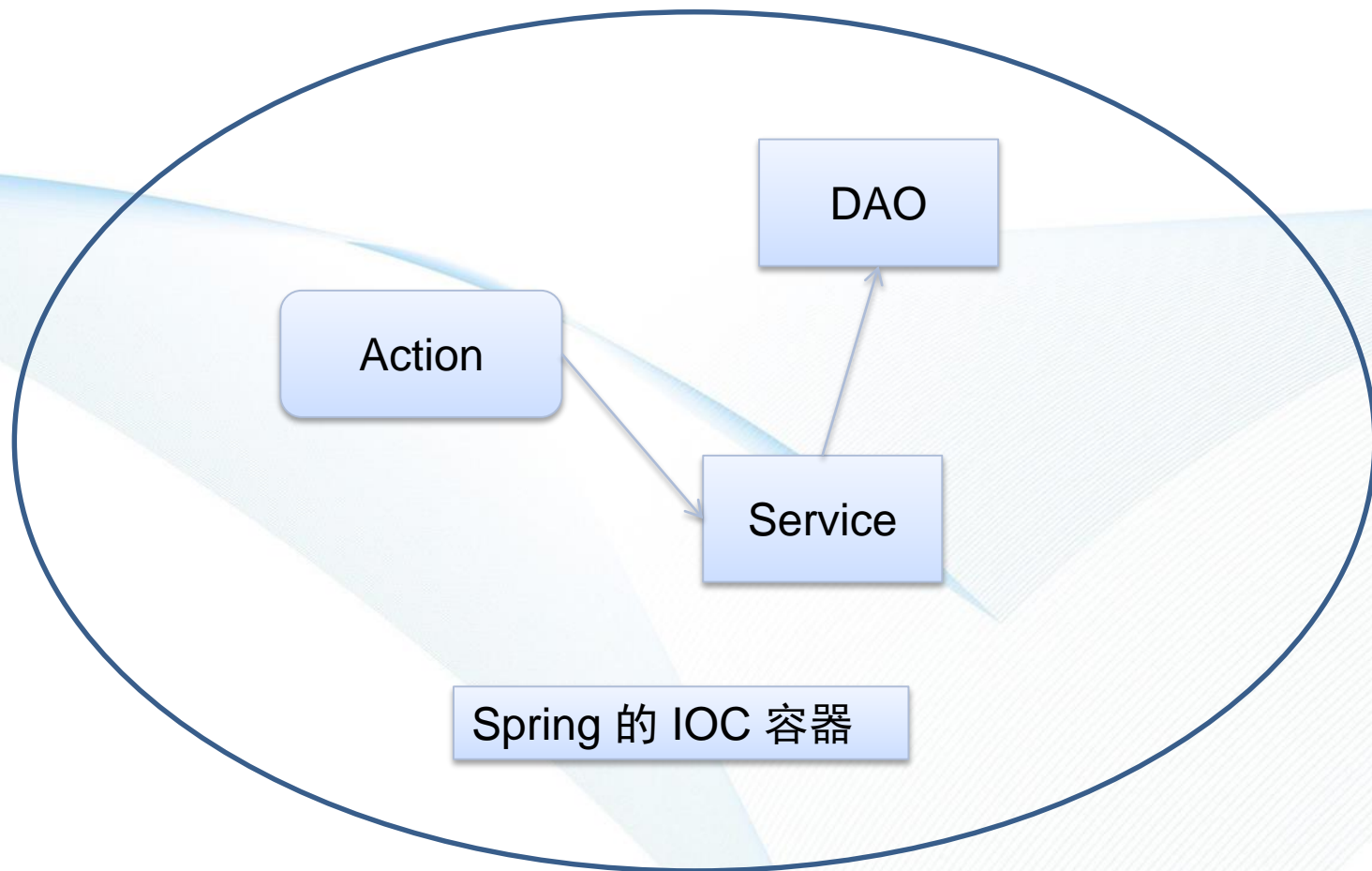
org.springframework.web.context.support  
**Class WebApplicationContextUtils**

```
public static WebApplicationContext getRequiredWebApplicationContext(ServletContext sc)  
throws IllegalStateException
```

来获取 Spring 的 ApplicationContext 对象

```
WebApplicationContext applicationContext =  
    WebApplicationContextUtils.getRequiredWebApplicationContext(this.getServletContext());  
Test test = (Test) applicationContext.getBean("test");  
test.hello();
```





# 整合 Struts2

- Struts2 通过插件实现和 Spring 的整合.
- Struts2 提供了两种和 Spring整合基本的策略：
  - + 将 Action 实例交给 Spring 容器来负责生成，管理，通过这种方式，可以充分利用 Spring 容器的 IOC 特性，提供最好的解耦
  - + 利用 Spring 插件的自动装配功能，当 Spring 插件创建 Action 实例后，立即将 Spring 容器中对应的业务逻辑组件注入 Action 实例.

# 让 Spring 管理控制器

- 将 Action 实例交给 Spring 容器来负责生成, 管理, 通过这种方式, 可以充分利用 Spring 容器的 IOC 特性, 提供最好的解耦
- 整合流程:
  - + 安装 Spring 插件: 把 struts2-spring-plugin-2.2.1.jar 复制到当前 WEB 应用的 WEB-INF/lib 目录下
  - + 在 Spring 的配置文件中配置 Struts2 的 Action 实例
  - + 在 Struts 配置文件中配置 action, 但其 class 属性不再指向该 Action 的实现类, 而是指向 Spring 容器中 Action 实例的 ID



# 自动装配

- 利用 Spring 插件的自动装配功能, 当 Spring 插件创建 Action 实例后, 立即将 Spring 容器中对应的业务逻辑组件注入 Action 实例.
- 配置自动装配策略: Spring 插件的自动装配可以通过 `struts.objectFactory.spring.autoWire` 常量指定, 该常量可以接受如下值:
  - + `name`: 根据属性名自动装配.
  - + `type`: 根据类型自动装配. 若有多个 `type` 相同的 Bean, 就抛出一个致命异常; 若没有匹配的 Bean, 则什么都不会发生, 属性不会被设置
  - + `auto`: Spring 插件会自动检测需要使用哪种方式自动装配方式
  - + `constructor`: 同 `type` 类似, 区别是 `constructor` 使用构造器来构造注入所需的参数
- 整合流程:
  - + 安装 Spring 插件
  - + 正常编写 `struts` 配置文件
  - + 编写 `spring` 配置文件, 在该配置文件中不需要配置 Action 实例