SPR企业级应用开发

1

第01章：Spring概述

CSG
CAMPUS SOLUTION GROUP

本章内容

- 第1节 Spring框架简介
- 第2节 Spring体系结构
- 第3节 Spring快速入门

本章目标

- 能了解Spring框架的用途和优势。
- 能掌握Spring框架的体系及文档结构。
- 能了解Spring框架的子项目。
- 能够对于IOC和AOP有初步的认知并能够实现一个基本应用。

第1节 Spring框架简介

知识点预览

#	知识点	重要度			重点	难点
		必学	选学	拔高		
1	Spring框架概述	●				●
2	Spring优势	●			●	●
3	Spring应用	●			●	●

Spring框架概述-1

Spring概述

- 要谈Spring的历史，就要先谈J2EE。
  - J2EE应用程序的广泛实现是在1999年和2000年开始的，它的出现带来了诸如事务管理之类的核心**中间层概念的标准化**，但是开发效率，开发难度和实际的性能都令人失望。
  - Spring的形成，最初来自Rod Jahnson所著的一本很有影响力的书籍《Expert One-on-One J2EE Design and Development》，就是在这本书中第一次出现了Spring的一些核心思想，该书出版于2002年。
  - Spring的一个最大的目的就是使J2EE开发更加容易。同时，Spring与Struts、Hibernate等单层框架不同，Spring致力于提供统一的、高效的方式构造整个应用，并且和其它框架揉和在一起建立一个连贯的体系。
- Spring是一个提供了更完善开发环境的一个框架，可以为POJO(Plain Old Java Object)对象提供企业级的服务。
- Spring框架是由于软件开发的复杂性而创建的，从简单性、可测试性和松耦合性的角度而言，绝大部分Java应用都可以从Spring中受益。

Spring框架概述-2

- **Spring初衷**
  - J2EE开发应该更加简单。
  - 使用接口而不是使用类，是更好的编程习惯。Spring将使用接口的复杂度几乎降低到了零。
  - 为JavaBean提供了一个更好的应用配置框架。
  - 更多地强调面向对象的设计，而不是现行的技术如J2EE。
  - 尽量减少不必要的异常捕捉。
  - 使应用程序更加容易测试。
- **Spring的目标**
  - 可以令人方便愉快的使用Spring。
  - 应用程序代码并不依赖于Spring APIs。
  - Spring不和现有的解决方案竞争，而是致力于将它们融合在一起。

关于SpringSource

- Spring的创始人Rod JohnsonSpring 不但技术出众，也是商业奇才，当Spring1.0发布时，Rod就和他的团队成立了SpringSource公司，并以商业化的方式对开源的Spring进行运作。
- SpringSource公司以Spring为依托，开展了很多代表不同技术领域的子项目，将触角伸向Web Service、安全、客户端等技术领域，通过子项目的探索来丰富Spring框架内涵。
- SpringSource公司还关注市场的推广和宣传，进行培训、咨询以及认证等商业盈利模式。2007年吸引风险投资，2008年收购G2One等公司，从而成为开发框架、应用服务器及应用服务监控服务商。
- SpringSource的最终目标是在“云”服务的应用市场拥有话语权，在2009年其和VMWare合并，从而使应用客户数据中心和集成化平台服务成为可能。

Spring优势-1

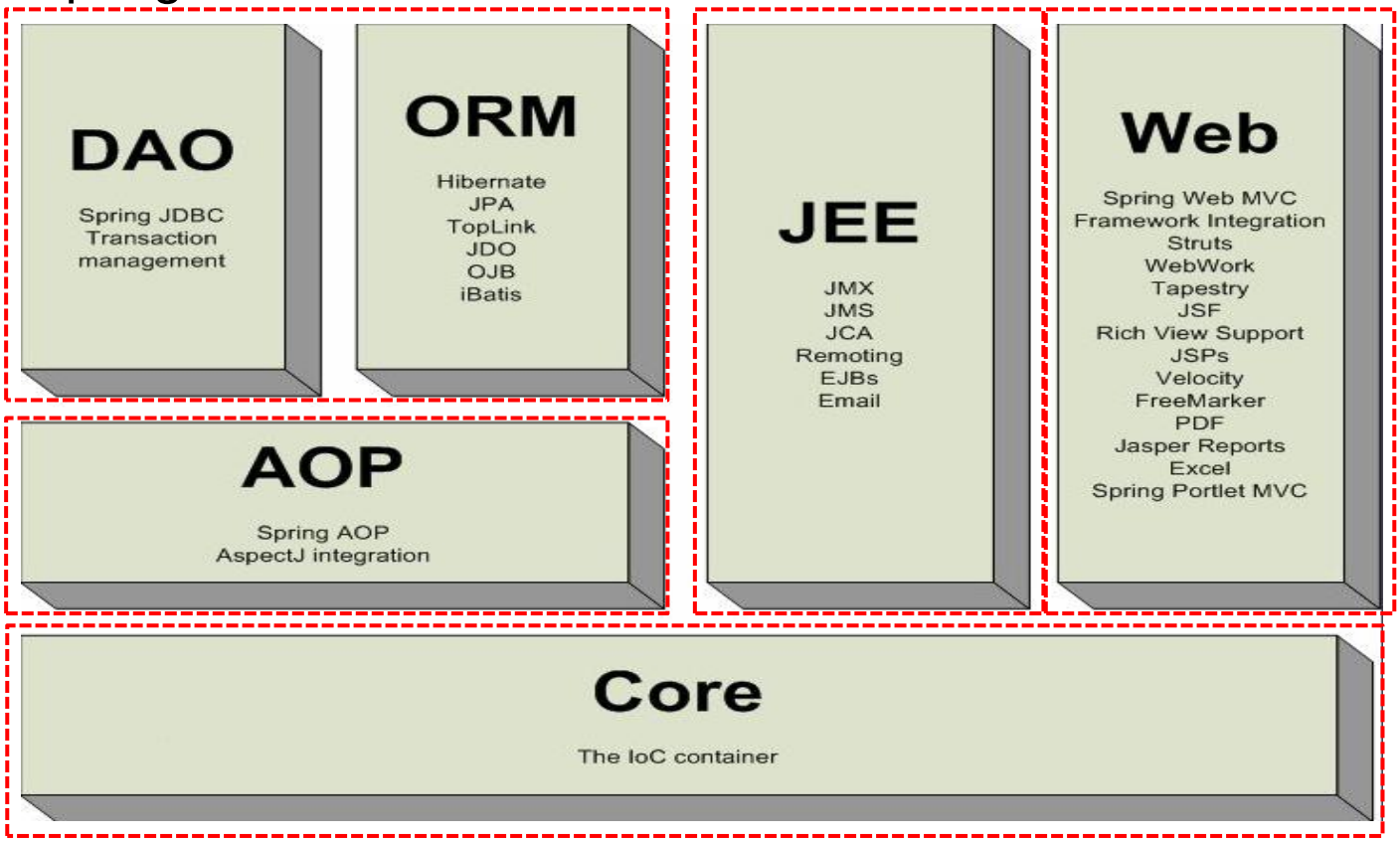
- **Spring的理念**
  - Spring认为JavaEE的开发应该更容易，更简单，在实现这一目标时，Spring一直贯彻并遵守“好的设计优于具体实现，代码应易于测试”这一理念。
    - 在实现时努力提供服务的同时尽量简化开发。
    - 在对于较为复杂的实现时，例如需要在传统JavaEE容器上运行的Web应用，难于测试的实现，而Spring可以基于Junit等应用实现单例测试。
- **Spring的优点**
  - 方便解耦，简化开发：
    - 通过Spring提供的IoC（Inversion of Control）容器，将对象间的依赖关系交由Spring容器进行控制，避免硬编码造成的过度程序耦合。
    - 用户不必再为单实例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。
  - 使复杂的实现变简单：
    - 通过Spring的AOP功能，方便进行面向切面的编程，许多不容易用OOP实现的功能都可以通过AOP轻松应对。

Spring优势-2

- **Spring的优点（续）**
  - 声明式事务的支持：
    - 在Spring中，从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提高开发效率和质量。
  - 方便程序的测试：
    - 用非容器依赖的编程方式进行几乎所有的测试工作，在Spring里，测试不是昂贵的事情，而是随手可做的事情。
  - 方便集成各种优秀框架：
    - Spring不排斥各种优秀的开源框架，相反，Spring可以降低各种框架的使用难度，Spring提供了对各种优秀框架（如Struts、Hibernate、Quartz）的直接支持。
  - 降低JavaEE API的使用难度：
    - Spring对很多难用的JavaEE API提供了一个薄薄的封装层，通过Spring的简易封装，这些JavaEE API的使用难度大大降低。

Spring体系结构

- Spring框架按其所属功能划分为5个模块，如图：



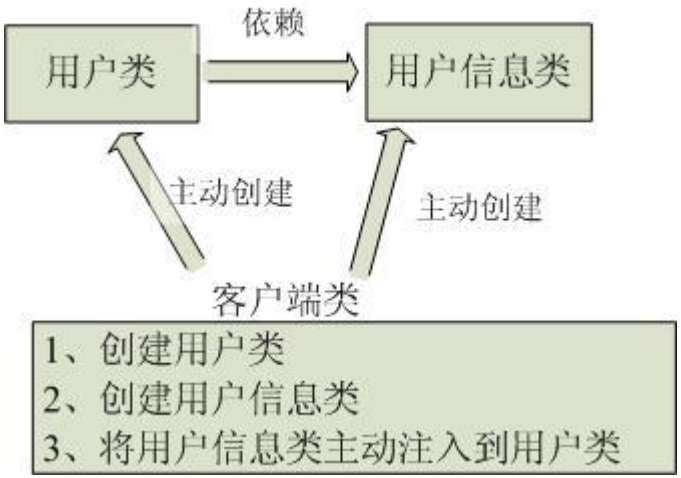
- 以上5个模块为企业提供所需的一切，从持久层、业务层到展现层都拥有相应的支持。

IoC是什么

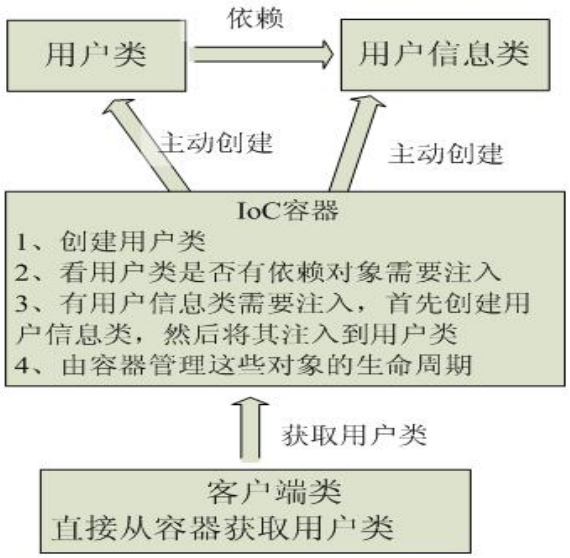
- Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。在Java开发中，Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。

传统程序设计

- 传统Java SE程序设计，我们直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；



- 在Java开发中，Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。



IoC能做什么

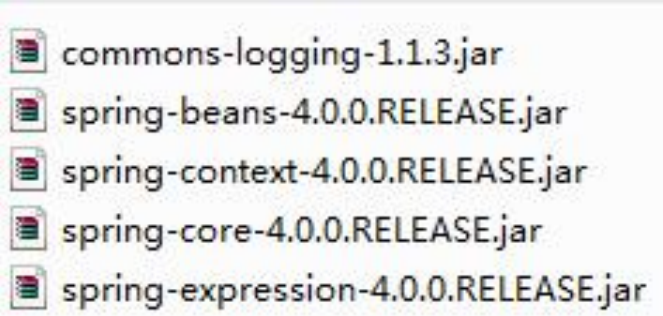
- IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。
- 有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

- IoC对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在IoC/DI思想中，应用程序就变成被动的了，被动的等待IoC容器来创建并注入它所需要的资源了。

- IoC很好的体现了面向对象设计法则之一——— 好莱坞法则：“别找我们，我们找你”；即由IoC容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。

搭建 Spring 开发环境

- 把以下 jar 包加入到工程的 classpath 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.

```
package com.tutorialspoint;
public class HelloWorld {
   private String message;
   public void setMessage(String message){
      this.message  = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " +
message);
   }
}
```

MainApp.java

```
package com.tutorialspoint;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
            new ClassPathXmlApplicationContext("Beans.xml");
      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
      obj.getMessage();
   }
}
```

第一步是我们使用框架 API ClassPathXmlApplicationContext() 来创建应用程序的上下文。这个 API 加载 beans 的配置文件并最终基于所提供的 API，它处理创建并初始化所有的对象，即在配置文件中提到的 beans。

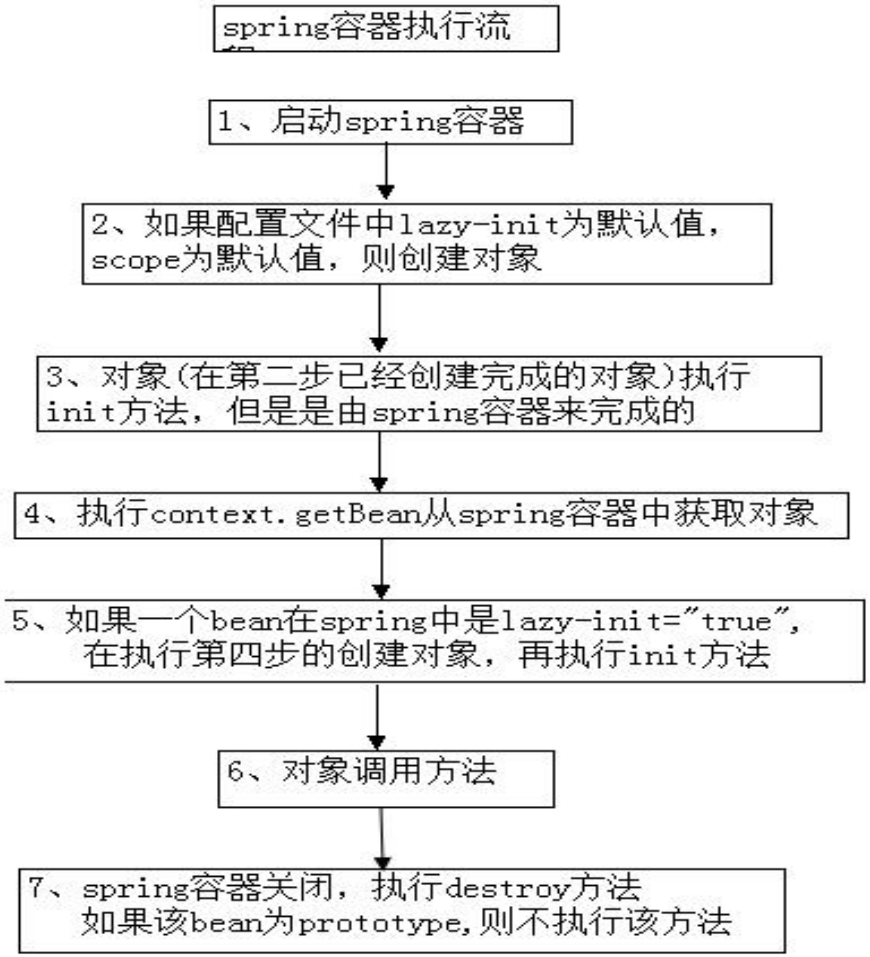
第二步是使用已创建的上下文的 getBean() 方法来获得所需的 bean。这个方法使用 bean 的 ID 返回一个最终可以转换为实际对象的通用对象。一旦有了对象，你就可以使用这个对象调用任何类的方法

创建 bean 的配置文件Beans.xml

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd">

   <bean id="helloWorld" class="com.tutorialspoint.HelloWorld">
       <property name="message" value="Hello World!"/>
   </bean>

</beans>
```

IoC 容器

- spring 容器是 Spring 框架的核心。容器将创建对象，把它们连接在一起，配置它们，并管理他们的整个生命周期从创建到销毁。Spring 容器使用依赖注入（DI）来管理组成一个应用程序的组件。这些对象被称为 Spring Beans。bean 是一个被实例化，组装，并通过 Spring IoC 容器所管理的对象。这些 bean 是由用容器提供的配置元数据创建的，



Spring入门示例-2

Spring入门示例2。该例中使用面向接口编程技术。

1、创建一个接口

```
package com.dao;
public interface UserDao {
    public void save(String uname,String pwd);
}
```

2、创建一个实现类将用户信息保存到MySQL数据库中

```
package com.dao;
public class UserDaoMysqlImpl implements UserDao {
        public void save(String uname, String pwd) {
                System.out.println("---UserDaoMysqlImpl---");
            }
}
```

Spring入门示例

3、创建一个实现类将用户信息保存到Oracle数据库中。

```
package com.dao;
public class UserDaoOracleImpl implements UserDao {
        public void save(String uname, String pwd) {
            System.out.println("---UserDaoOracleImpl---");
        }
}
```

Spring入门示例

4、创建一个管理类，将接口对象作为其属性

```
package com.manager;
import  com.dao.*;
public class UserManager {
    private  UserDao dao; // 将接口对象作为其属性
    public void save(String uname,String upwd){
          dao.save(uname, upwd);
    }
            public UserDao getDao() {
                    return dao;
          }
          public void setDao(UserDao dao) {
                    this.dao = dao;
          }
}
```

Spring入门示例

5、在Spring配置文件applicationContext.xml。将JavaBean由Spring容器来管理。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>   < !--配置Bean使Bean可以由Spring容器管理-->
  <bean id="oracleimpl" class="com.dao.UserDaoOracleImpl"></bean>
  <bean id="mysqlimpl" class="com.dao.UserDaoMysqlImpl"></bean>
  < !-- manager的dao这个属性值依赖Spring来注入，可以在程序中无需代码改变，就可以注入不同实例，本例中向oracle保存数据就注入oracleimpl，如果后来向MySQL中保存数据只需要修改注入实例，即注入mysqlimpl即可，代码并没有改变。-->
  <bean id="manager" class="com.manager.UserManager">
    <property name="dao" ref="oracleimpl"></property>
  </bean>
</beans>
```

Spring入门示例

6、编写测试类

```
public class Test {
        public static void main(String[] args) {
          /*读取Spring配置文件，创建一个Bean工厂*/
BeanFactory factory= new
ClassPathXmlApplicationContext("applicationContext.xml");
     /*读取Spring容器一个称为hello的bean,Spring容器自动创建对象。*/
   UserManager
manager=(UserManager)factory.getBean("manager");
   manager. save ("admin","1234");
 /*因为注入的是oracleimpl所以将信息保存到Oracle数据库中*/
        }
}
```

1 Spring 中的 Bean 配置

内容提要

- IOC & DI 概述
- 配置 bean
  - 配置形式：基于 XML 文件的方式；基于注解的方式
  - 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
  - 自动转配
  - bean 之间的关系：继承；依赖
  - bean 的作用域：singleton；prototype；WEB 环境作用域
  - 使用外部属性文件

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)接受来自如容器的资源注入. 相对于 IOC 而言，这种表述更直接

内容提要

- IOC & DI 概述
- 配置 bean
  - 配置形式：基于 XML 文件的方式；基于注解的方式
  - 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
  - 自动转配
  - bean 之间的关系：继承；依赖
  - bean 的作用域：singleton；prototype；WEB 环境作用域
  - 使用外部属性文件
  - spEL
  - IOC 容器中 Bean 的生命周期
  - Spring 4.x 新特性：泛型依赖注入

在 Spring 的 IOC 容器里配置 Bean

- 在 xml 文件中通过 bean 节点来配置 bean

```
<!-- 通过全类名的方式来配置 bean -->
<bean id="helloWorld"
    class="com.atguigu.spring.helloworld.HelloWorld">
</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.

Spring 的 BeanFactory 容器

- 主要的功能是为依赖注入 （DI） 提供支持，这个容器接口在 org.springframework.beans.factory.BeanFactory 中被定义。

- 在 Spring 中，有大量对 BeanFactory 接口的实现。其中，最常被使用的是 XmlBeanFactory 类。这个容器从一个 XML 文件中读取配置元数据，由这些元数据来生成一个被配置化的系统或者应用。

创建一个 Spring 应用程序

步骤	描述
1	创建一个名为 *SpringExample* 的工程并在 **src** 文件夹下新建一个名为 *com.tutorialspoint* 文件夹。
2	点击右键，选择 *Add External JARs* 选项，导入 Spring 的库文件，正如我们在 *Spring Hello World Example* 章节中提到的导入方式。
3	在 *com.tutorialspoint* 文件夹下创建 *HelloWorld.java* 和 *MainApp.java* 两个类文件。
4	在 **src** 文件夹下创建 Bean 的配置文件 *Beans.xml*
5	最后的步骤是创建所有 Java 文件和 Bean 的配置文件的内容，按照如下所示步骤运行应用程序。

HelloWorld.java

```
package com.tutorialspoint;
public class HelloWorld {
   private String message;
   public void setMessage(String message){
    this.message  = message;
   }
   public void getMessage(){
    System.out.println("Your Message : " +
message);
   }
}
```

MainApp.java

```
package com.tutorialspoint;
import org.springframework.beans.factory.InitializingBean;
import
org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
public class MainApp {
   public static void main(String[] args) {
      XmlBeanFactory factory = new XmlBeanFactory
                                  (new
ClassPathResource("Beanstt.xml"));
      HelloWorld obj = (HelloWorld)
factory.getBean("helloWorld");
      obj.getMessage();
   }
}
```

Beanstt.xml

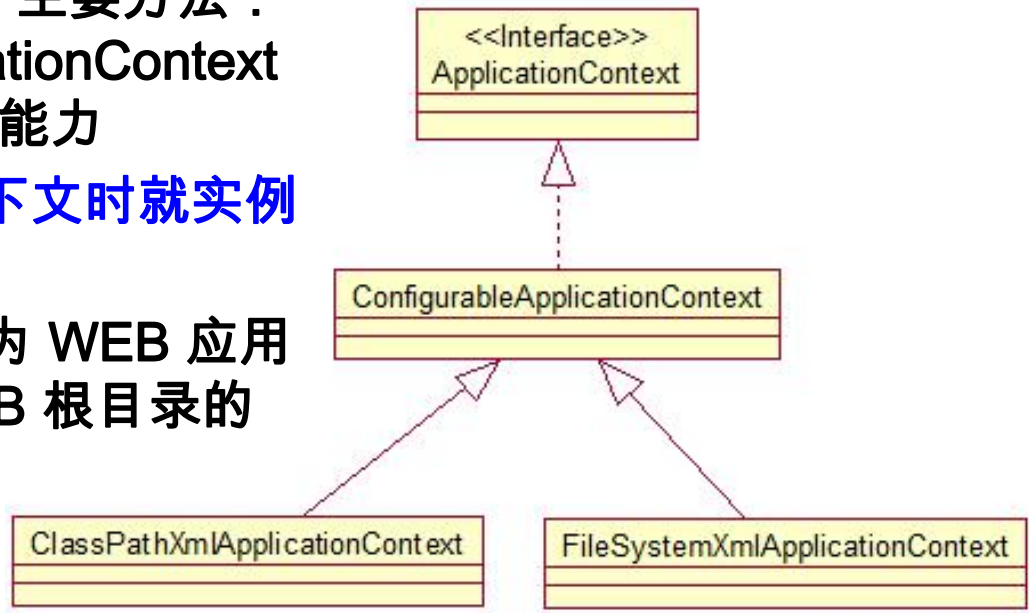
```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema
/beans
    http://www.springframework.org/schema/beans/spring-
beans-3.0.xsd">

   <bean id="helloWorld"
class="com.tutorialspoint.HelloWorld">
       <property name="message" value="Hello World!"/>
   </bean>

</beans>
```

Spring ApplicationContext 容器

- Application Context 是 spring 中较高级的容器。和 BeanFactory 类似，它可以加载配置文件中定义的 bean，将所有的 bean 集中在一起，当有请求的时候分配 bean。另外，它增加了企业所需要的功能，比如，从属性文件从解析文本信息和将事件传递给所指定的监听器。这个容器在 org.springframework.context.ApplicationContext interface 接口中定义。

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



最常被使用的 ApplicationContext 接口实现

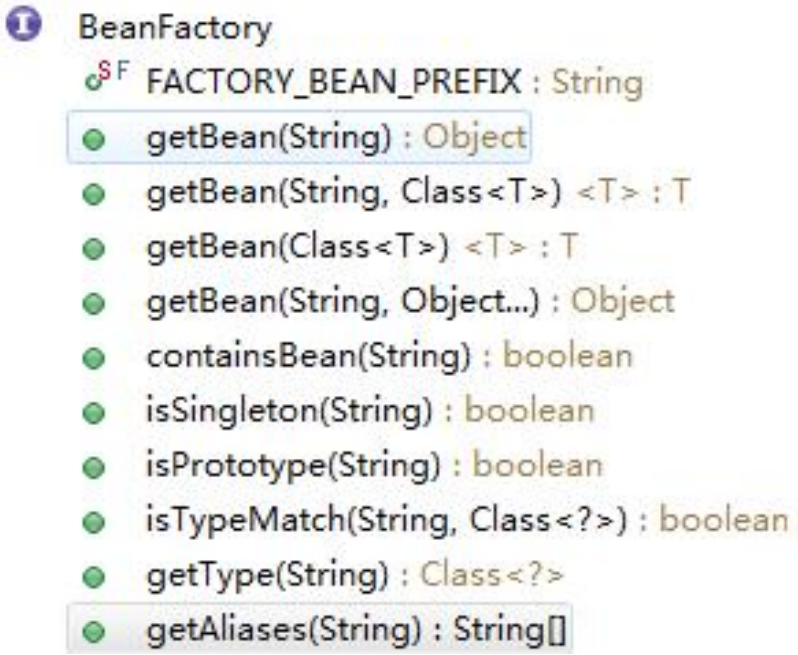
- FileSystemXmlApplicationContext：该容器从 XML 文件中加载已被定义的 bean。在这里，你需要提供给构造器 XML文件的完整路径

- ClassPathXmlApplicationContext：该容器从 XML 文件中加载已被定义的 bean。在这里，你不需要提供 XML 文件的完整路径，只需正确配置 CLASSPATH 环境变量即可，因为，容器会从 CLASSPATH 中搜索 bean 配置文件。

- WebXmlApplicationContext：该容器会在一个 web 应用程序的范围内加载在 XML 文件中已被定义的 bean。

从 IOC 容器中获取 Bean

- 调用 ApplicationContext 的 getBean() 方法



第二个例子

```
package com.atguigu.spring.helloworld;
public class HelloWorld {
	private String user;
	public HelloWorld() {
		System.out.println("HelloWorld's constructor...");
	}
	public void setUser(String user) {
		System.out.println("setUser:" + user);
		this.user = user;
	}
	public HelloWorld(String user) {
		this.user = user;
	}
	public void hello(){
		System.out.println("Hello: " + user);
	}
}
```

第二个例子建立 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();
}
```

创建 bean 的配置文件Beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
<!-- 配置一个 bean -->
<bean id="helloWorld" class="com.atguigu.spring.helloworld.HelloWorld">
            <!-- 为属性赋值 -->
            <property name="user" value="Jerry"></property>
</bean>
</beans>
```

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

```
<!-- 通过全类名的方式来配置 bean -->
<bean id="helloWorld"
    class="com.atguigu.spring.helloworld.HelloWorld">
    <property name="userName" value="atguigu"></property>
</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

构造方法注入

- 按索引匹配入参：

```
<bean id="car" class="com.atguigu.spring.helloworld.Car">
    <constructor-arg value="奥迪" index="0"></constructor-arg>
    <constructor-arg value="长春一汽" index="1"></constructor-arg>
    <constructor-arg value="500000" index="2"></constructor-arg>
</bean>
```

- 按类型匹配入参：

```
<bean id="car" class="com.atguigu.spring.helloworld.Car">
    <constructor-arg value="奥迪" type="java.lang.String"/>
    <constructor-arg value="长春一汽" type="java.lang.String"/>
    <constructor-arg value="500000" type="double"/>
</bean>
```

```
public class Car {
	private String company;
	private String brand;
	private int maxSpeed;
	private float price;
	public Car(String company, String brand, float price) {
		super();
		this.company = company;
		this.brand = brand;
		this.price = price;
	}
	public Car(String company, String brand, int maxSpeed) {
		super();
		this.company = company;
		this.brand = brand;
		this.maxSpeed = maxSpeed;
	}
	public Car(String company, String brand, int maxSpeed, float price) {
		super();
		this.company = company;
		this.brand = brand;
		this.maxSpeed = maxSpeed;
		this.price = price;
	}
}
```

```
Car car = (Car) ctx.getBean("car");
System.out.println(car);
```

内容提要

- IOC & DI 概述
- 配置 bean
  - 配置形式：基于 XML 文件的方式；基于注解的方式
  - 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
  - 自动转配
  - bean 之间的关系：继承；依赖
  - bean 的作用域：singleton；prototype；WEB 环境作用域
  - 使用外部属性文件
  - spEL
  - IOC 容器中 Bean 的生命周期
  - Spring 4.x 新特性：泛型依赖注入

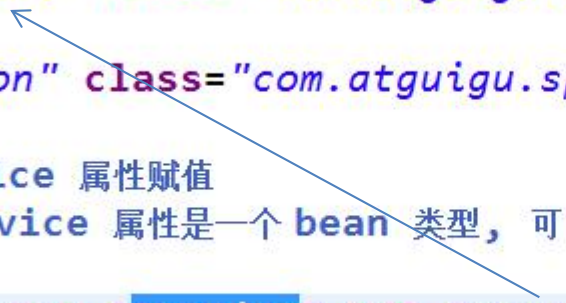
字面值

- 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。

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>
```



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

集合属性

- 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

- 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
- 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.

JavaCollection.java

```
package com.tutorialspoint;
import java.util.*;
public class JavaCollection {
   List addressList;
   Set  addressSet;
   Map  addressMap;
   Properties addressProp;
   // a setter method to set List
   public void setAddressList(List addressList) {
      this.addressList = addressList;
   }
   // prints and returns all the elements of the list.
   public List getAddressList() {
      System.out.println("List Elements :"  + addressList);
      return addressList;
   }
   // a setter method to set Set
   public void setAddressSet(Set addressSet) {
      this.addressSet = addressSet;
   }
   // prints and returns all the elements of the Set.
   public Set getAddressSet() {
      System.out.println("Set Elements :"  + addressSet);
      return addressSet;
   }
   // a setter method to set Map
   public void setAddressMap(Map addressMap) {
      this.addressMap = addressMap;
   }
   // prints and returns all the elements of the Map.
   public Map getAddressMap() {
      System.out.println("Map Elements :"  + addressMap);
      return addressMap;
   }
   // a setter method to set Property
   public void setAddressProp(Properties addressProp) {
      this.addressProp = addressProp;
   }
   // prints and returns all the elements of the Property.
   public Properties getAddressProp() {
      System.out.println("Property Elements :"  + addressProp);
      return addressProp;
   }
}
```

MainApp.java

```
package com.tutorialspoint;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationC
ontext;
public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
            new ClassPathXmlApplicationContext("Beans.xml");
      JavaCollection
jc=(JavaCollection)context.getBean("javaCollection");
      jc.getAddressList();
      jc.getAddressSet();
      jc.getAddressMap();
      jc.getAddressProp();
   }
}
```

Beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <!-- Definition for javaCollection -->
   <bean id="javaCollection" class="com.tutorialspoint.JavaCollection">

      <!-- results in a setAddressList(java.util.List) call -->
      <property name="addressList">
         <list>
            <value>INDIA</value>
            <value>Pakistan</value>
            <value>USA</value>
            <value>USA</value>
         </list>
      </property>

      <!-- results in a setAddressSet(java.util.Set) call -->
      <property name="addressSet">
         <set>
            <value>INDIA</value>
            <value>Pakistan</value>
            <value>USA</value>
            <value>USA</value>
        </set>
      </property>

      <!-- results in a setAddressMap(java.util.Map) call -->
      <property name="addressMap">
         <map>
            <entry key="1" value="INDIA"/>
            <entry key="2" value="Pakistan"/>
            <entry key="3" value="USA"/>
            <entry key="4" value="USA"/>
         </map>
      </property>

      <!-- results in a setAddressProp(java.util.Properties) call -->
      <property name="addressProp">
         <props>
            <prop key="one">INDIA</prop>
            <prop key="two">Pakistan</prop>
            <prop key="three">USA</prop>
            <prop key="four">USA</prop>
         </props>
      </property>

   </bean>

</beans>
```

List Elements :[INDIA, Pakistan, USA, USA]
Set Elements :[INDIA, Pakistan, USA]
Map Elements :{1=INDIA, 2=Pakistan, 3=USA, 4=USA}
Property Elements :{two=Pakistan, one=INDIA, three=USA, four=USA}

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 getBean() 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 WebApplicationContext 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 WebApplicationContext 环境

singleton 作用域：

- 如果作用域设置为 singleton，那么 Spring IoC 容器刚好创建一个由该 bean 定义的对象的实例。该单一实例将存储在这种单例 bean 的高速缓存中，以及针对该 bean 的所有后续的请求和引用都返回缓存对象。

- 默认作用域是始终是 singleton，但是当仅仅需要 bean 的一个实例时，你可以在 bean 的配置文件中设置作用域的属性为 singleton，

HelloWorld.java

```
package com.tutorialspoint.single;
public class HelloWorld {
   private String message;
   public void setMessage(String message){
      this.message  = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " +
message);
   }
}
```

Beansingle.xml

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd">

   <bean id="helloWorld"
class="com.tutorialspoint.single.HelloWorld"
      scope="singleton">
   </bean>

</beans>
```

MainApp.java

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new
ClassPathXmlApplicationContext("Beans.xml");
      HelloWorld objA = (HelloWorld) context.getBean("helloWorld");
      objA.setMessage("I'm object A");
      objA.getMessage();
      HelloWorld objB = (HelloWorld) context.getBean("helloWorld");
      objB.getMessage();
   }
}
```

```
Your Message : I'm object A
Your Message : I'm object A
```

CSG
CAMPUS SOLUTION GROUP

prototype 作用域

- 如果作用域设置为 prototype，那么每次特定的 bean 发出请求时 Spring IoC 容器就创建对象的新的 Bean 实例。
- HelloWorld.java
- package com.tutorialspoint.prototype;

```
public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }

   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

Beanprototype.xml

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd">

   <bean id="helloWorld" class="com.tutorialspoint.HelloWorld"
      scope="prototype">
   </bean>

</beans>
```

MainApp.java

```
package com.tutorialspoint;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationCont
ext;
public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new
ClassPathXmlApplicationContext("Beans.xml");
      HelloWorld objA = (HelloWorld)
context.getBean("helloWorld");
      objA.setMessage("I'm object A");
      objA.getMessage();
      HelloWorld objB = (HelloWorld)
context.getBean("helloWorld");
      objB.getMessage();
   }
}
```

Your Message : I'm object A

Your Message : null

CSG

CAMPUS SOLUTION GROUP

Bean 的生命周期

- 理解 Spring bean 的生命周期很容易。当一个 bean 被实例化时，它可能需要执行一些初始化使它转换成可用状态。同样，当 bean 不再需要，并且从容器中移除时，可能需要做一些清除工作。

- 尽管还有一些在 Bean 实例化和销毁之间发生的活动，但是本章将只讨论两个重要的生命周期回调方法，它们在 bean 的初始化和销毁的时候是必需的。

- 为了定义安装和拆卸一个 bean，我们只要声明带有 init-method 和/或 destroy-method 参数的 。init-method 属性指定一个方法，在实例化 bean 时，立即调用该方法。同样，destroy-method 指定一个方法，只有从容器中移除 bean 之后，才能调用该方法。

HelloWorld.java

```
package com.tutorialspoint.life;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
   public void init(){
      System.out.println("Bean is going through init.");
   }
   public void destroy(){
      System.out.println("Bean will destroy now.");
   }
}
```

Beanlife.xml

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

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id="helloWorld"
       class="com.tutorialspoint.life.HelloWorld"
       init-method="init" destroy-method="destroy">
       <property name="message" value="Hello World!"/>
   </bean>

</beans>
```

MainApp.java

```
package com.tutorialspoint;
import
org.springframework.context.support.AbstractApplicationConte
xt;
import
org.springframework.context.support.ClassPathXmlApplicationC
ontext;
public class MainApp {
   public static void main(String[] args) {
      AbstractApplicationContext context = new
ClassPathXmlApplicationContext("Beanlife.xml");
      HelloWorld obj = (HelloWorld)
context.getBean("helloWorld");
      obj.getMessage();
      context.registerShutdownHook();
   }
}
```

Bean is going through init.

Your Message : Hello World!

Bean will destroy now.

CSG

CAMPUS SOLUTION GROUP

内容提要

- IOC & DI 概述
- 配置 bean
  - 配置形式：基于 XML 文件的方式；基于注解的方式
  - 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
  - 自动装配
  - bean 之间的关系：继承；依赖
  - bean 的作用域：singleton；prototype；WEB 环境作用域
  - 使用外部属性文件
  - spEL
  - IOC 容器中 Bean 的生命周期
  - Spring 4.x 新特性：泛型依赖注入

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

基于注解的配置

- 从 Spring 2.5 开始就可以使用注解来配置依赖注入。而不是采用 XML 来描述一个 bean 连线，你可以使用相关类，方法或字段声明的注解，将 bean 配置移动到组件类本身。

内容提要

- IOC & DI 概述
- 配置 bean
  - 配置形式：基于 XML 文件的方式；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）
  - 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；工厂方法注入
  - 注入属性值细节
  - 自动转配
  - bean 之间的关系：继承；依赖；
  - bean 的作用域：singleton；prototype；WEB 环境作用域
  - 使用外部属性文件
  - spEL
  - IOC 容器中 Bean 的生命周期
  - Spring 4.x 新特性：泛型依赖注入

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
  - @Respository: 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	com.atguigu.XxxAnnotation	所有标注了 XxxAnnotation 的类。该类型采用目标类是否标注了某个注解进行过滤
assinable	com.atguigu.XxxService	所有继承或扩展 XxxService 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	com.atguigu..*Service+	所有类名以 Service 结束的类及继承或扩展它们的类。该类型采用 AspejctJ 表达式进行过滤
regex	com.\atguigu\.anno\..*	所有 com.atguigu.anno 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	com.atguigu.XxxTypeFilter	采用 XxxTypeFilter 通过代码的方式定义过滤规则。该类必须实现 org.springframework.core.type.TypeFilter 接口

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 @Authwired 注解
  - 默认情况下, 所有使用 @Authwired 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 @Authwired 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 @Qualifiter 已指定注入 Bean 的名称
  - @Authwired 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.
  - @Authwired 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.
  - @Authwired 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 @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 reqired 属性
- 建议使用 @Autowired 注解

Student.java

```
import org.springframework.beans.factory.annotation.Required;
public class Student {
   private Integer age;
   private String name;
   @Required
   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
      return age;
   }
   @Required
   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }
}
```

MainApp.java

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplica
tionContext;
public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new
ClassPathXmlApplicationContext("beanauto.xml");
      Student student = (Student)
context.getBean("student");
      System.out.println("Name : " +
student.getName() );
      System.out.println("Age : " + student.getAge() );
   }
}
```

beanauto.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:annotation-config/>

   <!-- Definition for student bean -->
   <bean id="student" class="com.tutorialspoint.auto.Student">
      <property name="name"  value="Zara" />

      <!-- try without passing age and check the result -->
      <property name="age"  value="11"/>
     <!--  <property name="age"  value="11"/> -->
   </bean>
       <!-- Definition for textEditor bean without constructor-arg  -->
   <bean id="textEditor" class="com.tutorialspoint.auto.TextEditor">
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.tutorialspoint.auto.SpellChecker">
   </bean>

</beans>
```

CSG
CAMPUS SOLUTION GROUP

- <context:annotation-config>:注解扫描是针对已经在Spring容器里注册过的Bean
- <context:component-scan>:不仅具备<context:annotation-config>的所有功能，还可以在指定的package下面扫描对应的bean

XML注册Bean方式

- package com.test;
- pubic class B
- { public B()
  - { System.out.println("B类"); }
- }

```
package com.test;
public class A {
    private B bClass;

    public void setBClass(B bClass){
        this.bClass = bClass;
        System.out.println("通过set的方式注入B类");
    }

    public A(){
        System.out.println("A类");
    }
}
```

Xml配置

- <bean id="bBean"class="com.test.B"/>
- <bean id="aBean"class="com.test.A">
- <property name="bClass" ref="bBean"/>
- </bean>

类B
类A
通过set的方式注入B类

annotation配置注解开启方式

- package com.test; pubic class B{ public B(){ System.out.println(″B类″); } }

```
package com.test;
public class A {
        private B bClass;
        @Autowired
        public void setBClass(B bClass){
                this.bClass = bClass;
                System.out.println("通过set的方式注入B类");
        }

        public A(){
                System.out.println("A类");
        }
}
```

- <bean id="bBean"class="com.test.B"/>
- <bean id="aBean"class="com.test.A"/>
- 或者（仅仅开启扫描，不注册Bean）
- <context:annotation-config/>

类B
类A

- <context:annotation-config/>
- <bean id="bBean"class="com.test.B"/>
- <bean id="aBean"class="com.test.A"/>

类B
类A
通过set的方式注入B类

归纳：<context:annotation-config>:**注解扫描是针对已经在Spring容器里注册过的Bean**

component配置注解开启方式

```
package com.test;
pubic class B{
 public B(){ System.out.println("B类"); }
 }
```

```
package com.test;
@Component
public class A {
    private B bClass;

    @Autowired
    public void setBClass(B bClass){
        this.bClass = bClass;
        System.out.println("通过set的方式注入B类");
    }

    public A(){
        System.out.println("A类");
    }
}
```

开启annotaion-config扫描

- <context:annotation-config />

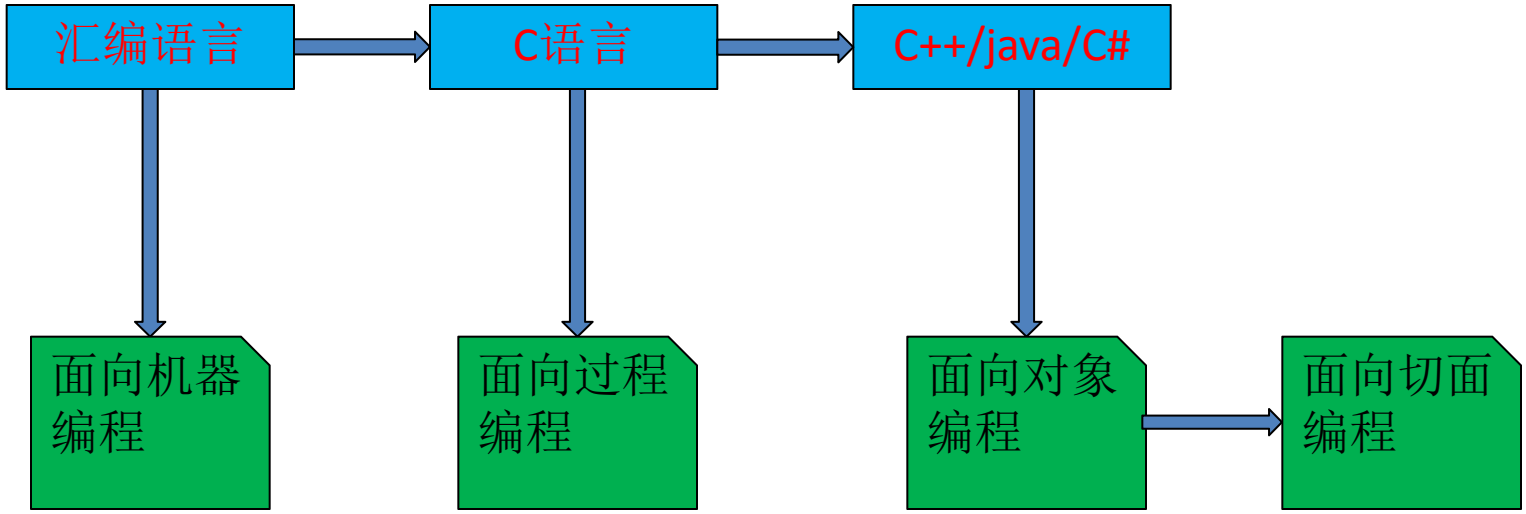
类B
类A

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

输出:
类B
类A
通过set的方式注入B类

1Spring AOP介绍

语言的发展历史



Spring AOP介绍

- AOP：Aspect Oriented Programming，面向切面编程，可以说是OOP（Object Oriented Programming，面向对象编程）的补充和完善。

- OOP引入封装、继承、多态等概念来建立一种对象层次结构
- AOP则利用一种称为“横切”的技术，剖开对象内部，并将影响了多个类的的公共行为封装到可重用模块，从而减少重复代码，降低耦合。
- 如果说 IoC 是 Spring 的核心，那么面向切面编程就是 Spring 最为重要的功能之一了，在数据库事务中切面编程被广泛使用.

SPRING AOP

- AOP基本概念
  - 面向方面编程将程序分解成各个层次的对象
  - 面向切面编程将程序运行过程分解成各个切面
  - AOP把日志记录、性能统计、安全控制、事务处理、异常处理等等这些行为代码从业务逻辑中划分出来，使得改变这些行为的时候不会影响到业务逻辑的代码
  - Spring框架提供了丰富的AOP支持。应用对象只需要实现它们应该完成的业务逻辑，并不负责其他系统级业务，例如日志记录、事务支持等等

Spring AOP编程(1)

面向切面编程：Aspect Oriented Programming，可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。可以说是OOP（Object-Oriented Programing，面向对象编程）的补充和完善。

在OOP设计中有可能导致代码的重复,不利于模块的重用性，例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能关系不大。但是在OOP中这些业务要和核心业务代码在代码这一级集成。还有些如安全性、事务等也是如此。

能不能把这些与核心业务无关但系统中需要使用的业务（称为切面）单独编写成一个模块，在主要核心业务代码中不调用，而是在配置文件中做些配置，配置核心业务需要使用到的切面部分，在系统编译时才织入到业务模块中。

以下是几个AOP基本概念

切面（Aspect）：对象操作过程中的截面。简单的理解就是那些与核心业务无关的代码形成的平行四边形拦截了程序流程。我们把它提取出来，进行封装成一个或几个模块用来处理那些附加的功能代码。（如日志，事务，安全验证）我们把这个模块的作用理解为一个切面，其实切面就是我们写一个类，这个类中的代码原来是在业务模块中完成的，现在单独成一个或几个类。在业务模块需要的时候才织入。

连接点（Joinpoint）：对象操作过程中的某个阶段点。在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候。 在Spring AOP中，一个连接点总是代表一个方法的执行。通过声明一个JoinPoint类型的参数可以使通知（Advice）的主体部分获得连接点信息。

切入点（Pointcut）：连接点的集合。本质上是一个捕获连接点的结构。在AOP中，可以定义一个pointcut，来捕获相关方法的调用

以下是几个AOP基本概念

- 切入点（Pointcut）：连接点的集合。切面与程序流程的“交叉点”就是程序的切入点，即它是“切面注入”到程序中的位置，“切面”是通过切入点被“注入”的。程序中有很多个切入点。

以下是几个AOP基本概念

织入（Weaving）：是将切面功能应用到目标对象的过程。（三种织入方式）这些可以在**编译时**（例如使用AspectJ编译器），**类加载时**和**运行时**完成。**Spring和其它纯Java AOP框架一样，在运行时完成织入。**

通知（Advice）：通知是某个切入点被横切后所采取的处理逻辑，也就是说在“切入点”处拦截程序后通过通知来执行切面。故通知是在切面的某个特定的连接点（Joinpoint）上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。 通知的类型将在后面部分进行讨论。许多AOP框架，包括Spring，都是以拦截器做通知模型，并维护一个以连接点为中心的拦截器链。

以下是几个AOP基本概念

通知的类型：

前置通知（Before advice）：在某连接点（join point）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。

返回后通知（After returning advice）：在某连接点（join point）正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。

抛出异常后通知（After throwing advice）：在方法抛出异常退出时执行的通知。

后置通知（After （finally） advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。

环绕通知（Around Advice）：包围一个连接点（join point）的通知，如方法调用。这是最强大的一种通知类型。 环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。

Aspect对AOP的支持

- Aspect即Spring中所说的切面，它是对象操作过程中的截面，在AOP中是一个非常重要的概念。
- Aspect是对系统中的对象操作过程中截面逻辑进行模块化封装的AOP概念实体，通常下可以包含多个切入点和通知。AspectJ是Spring框架2.0之后增加的新特性，Spring使用了AspectJ提供的一个库来完成切入点的解析和匹配。但是AOP在运行时仍旧是纯粹的Spring AOP，它并不依赖于AspectJ的编译器或者织入器，在底层中使用的仍然是Spring2.0之前的实现体系。
- 最初在Spring中没有完全明确的Aspect概念，只是在Spring中的Aspect的实现和特性有所特殊而已，而Spring中的Aspect就是Advisor。Advisor就是切入点的配置器，它能将Advice（通知）注入程序中的切入点的位置，并可以直接编程实现Advisor，也可以通过XML来配置切入点和Advisor。下一节我们介绍如何运用Aspect通过这两种方式实现AOP。

Spring AOP编程(5)

例11.8 AOP示例1。**(通过配置文件实现AOP**）
第1步： 编写一个类封装用户的常见操作。 UserDao.java

```
package com.aop;
public class UserDao {
   public void save(String name){
            System.out.println("----save user----");
   }
   public void delete(){
             System.out.println("----delete user----");
   }
   public void update(){
          System.out.println("----update user----");
   }
 }
```

Spring AOP编程(6)

第2步： 编写一个检查用户是否合法的类。

```
package com.aop;
public class CheckSecurity {
        public void check(){
           System.out.println("-----check admin----");
    }
}
```

Spring AOP编程(7)

（3）改写Spring配置文件AOP

```
<bean id="checkbean"
class="com.aop.CheckSecurity"></bean>
 <bean id="userDao" class="com.aop.UserDao"/>
 <aop:config>
    <!--创建一个切面Aspect 引用了bean "checkbean"-->
    <aop:aspect id="security" ref="checkbean">
    <!--声明一个切入点 当调用com.aop.UserDao.*(..)中的所有
函数时-->
<aop:pointcut id="allAddMethod"
expression="execution(* com.aop.UserDao.*(..))"/>
     <!--当执行到切入点时，会在执行切入点之前调用bean
"checkbean"中"check" 函数--><aop:before
method="check" pointcut-ref="allAddMethod"/>
        </aop:aspect>
</aop:config>
```

CSG
CAMPUS SOLUTION GROUP

Spring AOP编程(8)

第4步：编写测试类 Test.java。

```
package com.aop;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.*;
public class Test {
        public static void main(String[] args) {
                /* 读取Spring配置文件，创建一个Bean工厂 */
BeanFactory factory = new ClassPathXmlApplicationContext(

"applicationContext.xml");
    UserDao dao = (UserDao) factory.getBean("userDao");
                dao.save("zhou");
        }
}
```

例子2

- Page148 吃饭

```
package edu.hdu.spring.aop;
public interface DinningInterface {
public void eat();
}
package edu.hdu.spring.aop;
public class DinningInterfaceImpl implements
DinningInterface {
public void eat() {
System.out.println("开吃啦！");
}
}
```

```
/*Before.java 提前通知*/
package edu.hdu.spring.aop;
import java.lang.reflect.Method;
import
org.springframework.aop.MethodBeforeAdvice;
public class Before implements
MethodBeforeAdvice {
public void before(Method arg0, Object[] arg1,
Object arg2) throws Throwable {
System.out.println("点好菜！ ");
}
}
```

```
package edu.hdu.spring.aop;
import java.lang.reflect.Method;
import
org.springframework.aop.AfterReturningAdvice;
public class After implements
AfterReturningAdvice{
public void afterReturning(Object returnValue,
Method method, Object[] args,
Object target) throws Throwable {
        System.out.println("结账了！");
}
}
```

```
<bean id="targetbean" class="edu.hdu.spring.aop.DinningInterfaceImpl"></bean>
<!-- Advisor配置部分-->
<bean id="beforeadvisor"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
<property name="advice">
<ref local="beforeadvice" />
</property>
<property name="pattern">
<value>edu.hdu.spring.aop.DinningInterface.eat</value>
</property>
</bean>
<bean id="afteradvisor"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
<property name="advice">
<ref local="afteradvice" />
</property>
<property name="pattern">
<value>edu.hdu.spring.aop.DinningInterface.eat</value>
</property>
</bean>
<!-- Advice配置部分-->
<bean id="beforeadvice" class="edu.hdu.spring.aop.Before"></bean>
<bean id="afteradvice" class="edu.hdu.spring.aop.After"></bean>
```

```
package edu.hdu.spring.aop;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.FileSystemXmlApplic
ationContext;
public class MainTest {
public static void main(String[] args) {
        ApplicationContext ac = new
FileSystemXmlApplicationContext("src/spring-aop.xml");
DinningInterface din =
(DinningInterface)ac.getBean("proxybean");
        din.eat();
}
}
```