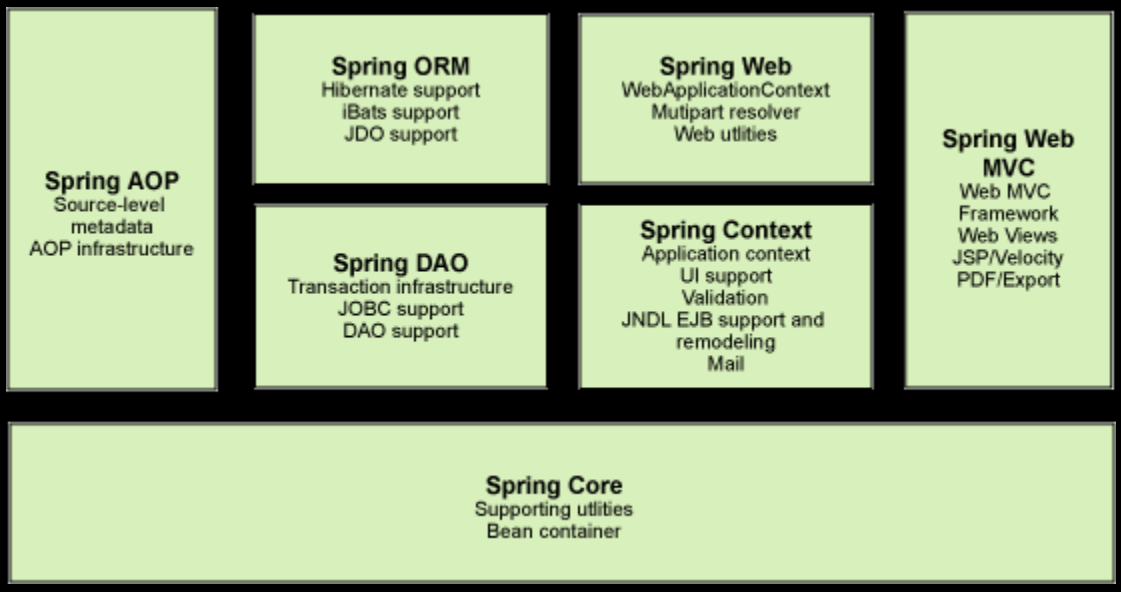
Spring 学习笔记

Spring 框架介绍

Spring 是一个开源框架，是为了解决企业应用程序开发复杂性而创。
主要优势之一就是其分层架构，分层架构允许您选择使用哪一个组件，同时为 J2EE 应用程序开发提供集成的框架。
由 7 个定义良好的模块组成。Spring 模块构建在核心容器之上，核心容器定义了创建、配置和管理 bean 的方式



每个模块（或组件）都可以单独存在，或者与其他一个或多个模块联合实现

功能：

1. 核心容器：核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转 （IOC） 模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。
2. Spring 上下文：Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。
3. Spring AOP：通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring 框架中。所以，可以很容易地使 Spring 框架管理的任何对象支持 AOP。Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖 EJB 组件，就可以将声明性事务管理集成到应用程序中。
4. Spring DAO：JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和关闭连接）。Spring DAO

的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。

5. Spring ORM: Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具,其中包括 JDO、Hibernate 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。
6. Spring Web 模块: Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。
7. Spring MVC 框架: MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。

IOC:

控制反转模式（也称作依赖性介入）的基本概念是：不创建对象，但是描述创建它们的方式。在代码中不直接与对象和服务连接，但在配置文件中描述哪一个组件需要哪一项服务。容器 （在 Spring 框架中是 IOC 容器） 负责将这些联系在一起。

AOP

面向方面的编程，即 AOP，是一种编程技术，它允许程序员对横切关注点或横切典型的职责分界线的行为（例如日志和事务管理）进行模块化。AOP 的核心构造是方面，它将那些影响多个类的行为封装到可重用的模块中。

AOP 的功能完全集成到了 Spring 事务管理、日志和其他各种特性的上下文中。

IOC 容器

Spring 设计的核心是 org.springframework.beans 包，它的设计目标是与 JavaBean 组件一起使用。这个包通常不是由用户直接使用，而是由服务器将其用作其他多数功能的底层中介。下一个最高级抽象是 BeanFactory 接口，它是工厂设计模式的实现，允许通过名称创建和检索对象。BeanFactory 也可以管理对象之间的关系。BeanFactory 支持两个对象模型。

1. 单态 模型提供了具有特定名称的对象的共享实例，可以在查询时对其进行检索。Singleton 是默认的也是最常用的对象模型。对于无状态服务对象很理想。
2. 原型 模型确保每次检索都会创建单独的对象。在每个用户都需要自己的对象时，原型模型最适合。

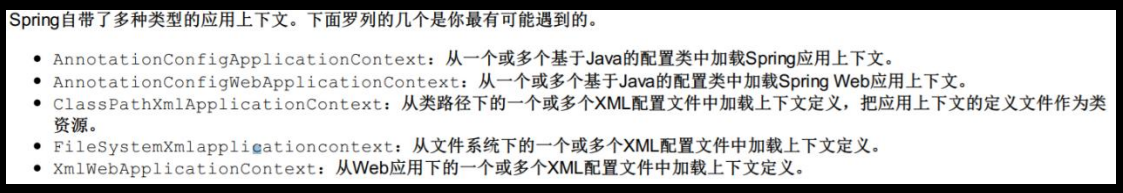
bean 工厂的概念是 Spring 作为 IOC 容器的基础。IOC 将处理事情的责任从应用程序代码转移到框架。正如我将在下一个示例中演示的那样，Spring 框架使用 JavaBean 属性和配置数据来指出必须设置的依赖关系。

https://www.ibm.com/developerworks/cn/java/wa-spring2/index.html?ca=drs-

基本概念

Edwin Xu

- J2EE: Java 2 Enterprise Edition
- DI: Dependency Injection 依赖注入
- AOP: Aspect-Oriented Programing 面向方面编程
- EJB : Enterprise JavaBean
- DAO(Data Access Object) 数据访问对象
- JDO: java Data Object
- POJO:Plain Old Java Object 简单老式 Java 对象
- 为了降低 Java 开发复杂性，Spring 采用策略：
    1. 基于 POJO 的轻量级和最小侵入性编程 ： 在 Spring 或非 Spring 类中都适用
    2. 基于依赖注入和面向接口实现**松耦合**
       （耦合度具有两面性：
       a.紧密耦合的代码难以测试、复用、理解，打地鼠式 bug
       b.耦合度太低也不行）
    3. 基于切片和惯例进行声明式编程：把遍布应用各处的功能分离出来形成可重用的组件
    4. 通过切片和模版减少样板式编程
- 装配 wiring:创建应用之间协作的行为。Spring 有多种装配方式，如 **XML 式**、Java 描述配置
- 容器 container 是核心，应用对象生存于 Spring 容器
- 应用上下文：

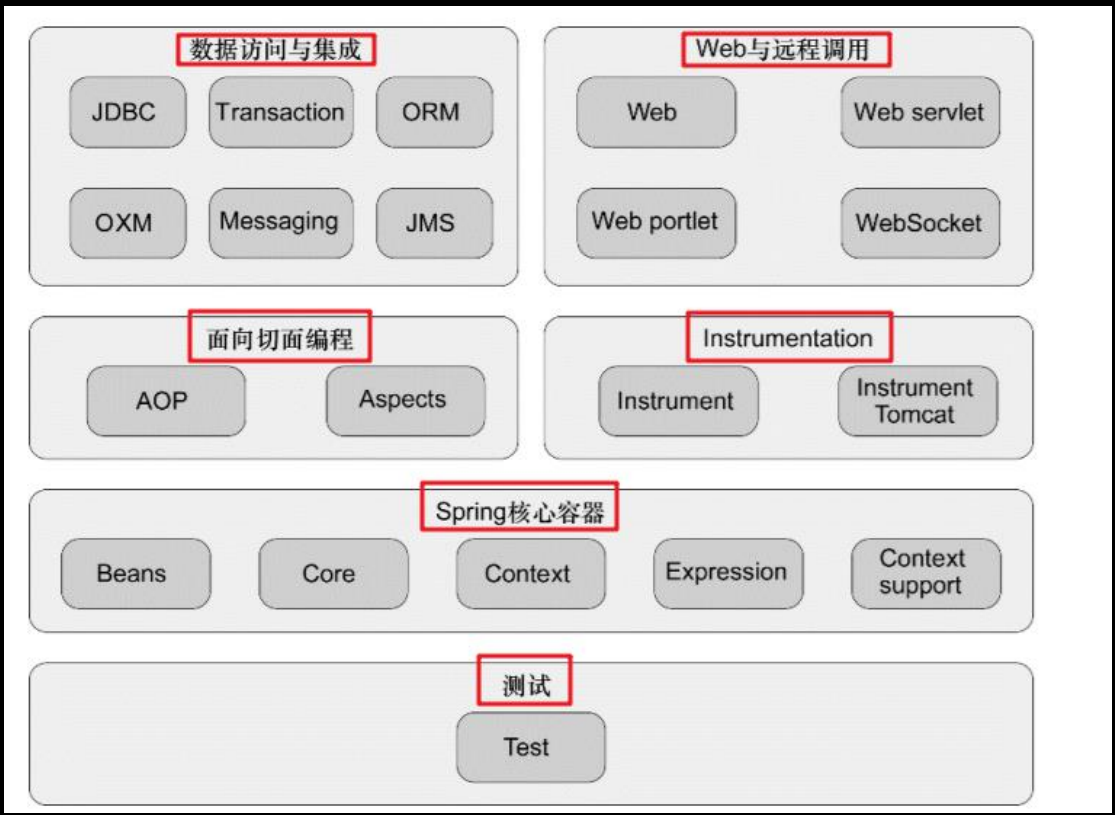


- MVC 模式使应用程序的输入逻辑、业务逻辑、UI 逻辑分离，同时提供这些元素之间的松散耦合。
    1. 模型 Model：封装了应用程序数据，通常他们由 POJO 组成。
    2. 视图 View：渲染模型数据，生成浏览器可以解释的 HTML 输出。
    3. 控制器 Controller：负责处理用户请求并构建适当的模型，将其传递给视图进行渲染。

Spring 模版

Spring4 中框架共有 20 个不同的模版，每个模版有三个 jar 文件（二进制类库、码源 jar、javadoc 的 jar）

可以分为六类：



Edwin Xu

- Spring 核心容器：由 **spring-core，spring-beans，spring-context，spring-context-support 和 spring-expression**（SpEL，Spring 表达式语言，Spring Expression Language）等模块组成。这些容器管理者 Spring 应用中 bean 的创建、配置、管理。该模块包含 Spring bean，它为 Spring 提供了 DI 的功能。除此外，还有 E-mail、JNDI 访问、EJB 集成与调度
- AOP 模块：
- 数据访问与集成
    1. JDBC：提供 jdbc 抽象层，消除了冗长的 jdbc 编码和对数据库供应商特定错误代码的解析。
    2. ORM：Object Relational Mapping 提供了对流行的对象关系映射 API 的集成，包含 JPA、JDO、Hibernate
    3. OXM:Object XML Mapping:提供了对 OXM 实现的支持，比如 JAXB、Castor、XML Beans
    4. JMS
- Web 与远程调用：MVC（Model-View-Controller）可以将界面逻辑与应用逻辑分离，Apache 的 Struts、JSF、WebWork、Tapestry 都是可选的流行 MVC 框架
- Instrumentation：为 JVM 添加代理，即为 Tomcat 提供了一个织入代理。
- 测试：JNDI、Servlet

Spring Portfolio

Protfolio 为每一个领域的 Java 开发提供了框架：

1. Spring Web Flow：基于流程的会话式 Web
2. Spring Web Service:
3. Spring Security
4. Spring Integration
5. Spring Batch
6. Spring Data
7. Spring Social
8. Spring Mobile
9. Spring for Android
10. Spring Boot：进一步简化 Spring 自身

C2 • 装配 Bean

装配 Wiring：创建应用对象之间协作关系的行为

Spring 配置有三种方案：

1. 在 XML 中进行显示配置

2. 在 Java 中进行显示配置

3. 隐式的 bean 发现机制和自动装配
效果一样，可以相互搭配
建议尽可能自动配置，显示配置越少越好

基于 Java 的配置

1. 理解 @Configuration 和 @Bean 注释:
   @Configuration 注释位于类的顶端。它告知 Spring 容器这个类是一个拥有 bean 定义和依赖项的配置类。@Bean 注释用于定义 bean。方法名称与 bean id 或默认名称相同。该方法的返回类型是向 Spring 应用程序上下文注册的 bean。您可使用 bean 的 setter 方法来设置依赖项，容器将调用它们来连接相关项。基于 Java 的配置也被视为基于注释的配置。
2. AnnotationConfigApplicationContext 注册配置类
   在传统 XML 方法中，您可使用 ClassPathXmlApplicationContext 类来加载外部 XML 上下文文件。但在使用基于 Java 的配置时，有一个 AnnotationConfigApplicationContext 类。
   是 ApplicationContext 接口的一个实现，使您能够注册所注释的配置类。此处的配置类是使用 @Configuration 注释声明的 AppContext。
3. 配置 Web 应用程序
   XML 方式下利用 XmlWebApplicationContext 上下文来配置 Spring Web 应用程序，即在 Web 部署描述符文件 web.xml 中指定外部 XML 上下文文件的路径。XMLWebApplicationContext 是 Web 应用程序使用的默认上下文类。

清单 5. 使用外部 XML 上下文文件的 web.xml

```
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name
        <param-value>/WEB-INF/applicationContext.xml<
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLo
        </listener-class>
    </listener>
    <servlet>
    <servlet-name>sampleServlet</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherSer
    </servlet-class>
    </servlet>

...
</web-app>
```

4.
5.

自动化装配 bean

最为便利
Spring 从两个角度来实现自动化装配:
1. **组件扫描** component scanning: Spring 会自动发现应用上下文中所创建的 bean
2. **自动装配** autowiring: Spring 自动满足 bean 之间的依赖

类型自动装配：如果一个 bean 的数据类型与其他 bean 属性的数据类型相同，将自动兼容装配它。
Spring 支持 5 种装配模式:

- no – 缺省情况下，自动配置是通过“ref”属性手动设定
- byName – 根据属性名称自动装配。如果一个 bean 的名称和其他 bean 属性的名称是一样的，将会自装配它。
- byType – 按数据类型自动装配。如果一个 bean 的数据类型是用其它 bean 属性的数据类型，兼容并自动装配它。
- constructor – 在构造函数参数的 byType 方式。
- autodetect – 如果找到默认的构造函数，使用“自动装配用构造”；否则，使用“按类型自动装配”。

- @Component: 注解的类会作为一个组件

但是组件扫描默认不启用，需要显示配置 Spring 来使其主动扫描注解类:

- @ComponentScan: 在 Spring 中启用组件扫描（注意，若无其他配置，会扫描本包即所有子包的@Component）
  如果要指明需要扫描的基础包，添加指定 value 即可:
  @Configuration(“包名”)
  或者:
  @Configuration(basePackages = {“包 1”,”包 2”…})
- @Autowired: 注解，声明自动装配，可以用于构造器、setter、变量等上

注意：在装配时如果没有匹配到 bean，则会抛出异常，为了避免抛出异常，可以将@AutoWired 的 required 属性设置为 false:

@AutoWired（required = false）

- @Inject ：和@AutoWired 差别不大

自动装配与注解

一、属性自动装配。

由于使用手动配置 XML 时，常常会发生字母漏缺和大小写错误等，且无法对其进行检查，使开发效率降低。

采用自动装配将避免这些错误，且使配置简单化

当一个 bean 带有 autowire byName 属性时：

1.将查找类中所有的 set 方法名，如 setName,并获取去掉 set 后的首字母小写的字符串，如 name

2.去 Spring 容器中寻找是否有此字符串名称 id 的对象。

3.如果有，取出注入；若没有，报 null 异常

注意：除开 byName 可以自动装配的值外，如果有些值没有被初始化，则自动默认为 null

使用 atuowire = "byType"属性：

类型匹配则装配，否则不装配

按类型装配，条件比较宽松，不会有异常，不影响其他结果，甚至将 id 去掉也可以

全局 autowire

在 XML 中的：xsi:schemaLocation=""中添加全局 autowire：

default-autowire = "byName" / "byType"

然后在 bean 节点中使用 autowire = "default"

二、属性注解

jdk1.5 开始支持注解，Spring2.5 全面支持注解

利用注解的方式注入属性：

1.Spring 引入 Context 文件头

2.开启属性注解

@autoWired

@Autowired 是按类型自动装配的，不支持 id 匹配

<context:annotation-config/>:用于激活那些已经在 Spring 容器里注册过的

bean

<context:component-scan>除了具有<context:annotation-config /> 的功能之外，还**具有自动将带有@component,@service,@Repository 等注解的对象注册到 spring 容器中的功能。**

<context:annotation-config /> 和 <context:component-scan>同时存在的时候，**前者会被忽略**

<context:annotation-config/>还是不太懂

@Qualifier

@Autowired 是根据类型自动装配的，加上@Qualifier 则可以根据 byName 的方式来自动装配，

注意@Qualifier 不能单独使用

可以指定 id: @Qualifier（value="beanid"）

@Resource

@resource 如有指定的 Name 属性，先按该属性进行 byName 查找装配。其次在进行默认的 byName 装配。如果以上都不行，则使用 ByType，否则报错

@Autowired 与@Resource 异同:

1° @Autowired 与@Resource 都可以用来装配 bean。都可以写在字段上，或写在 setter 方法上。

2° @Autowired 默认按类型装配（属于 spring 规范），默认情况下必须要求依赖对象必须存在，如果要允许 null 值，可以设置它的 required 属性为 false，如: @Autowired(required=false) ，如果我们想使用名称装配可以结合@Qualifier 注解进行使用。

3° @Resource（属于 J2EE 复返），默认按照名称进行装配，名称可以通过 name 属性进行指定。如果没有指定 name 属性，当注解写在字段上时，默认取字段名进行按照名称查找，如果注解写在 setter 方法上默认取属性名进行装配。 当找不到与名称匹配的 bean 时才按照类型进行装配。但是需要注意的是，如果 name 属性一旦指定，就只会按照名称进行装配。

它们的作用相同都是用注解方式注入对象，但执行顺序不同。@Autowired 先 byType，@Resource 先 byName。

三.类自动装配

包扫描:

1.@Controller: 只能用在控制器上

2.@Service: 只能用在业务类上
3.@Repository: 只能用在 dao 类上
4.@Component: 无法使用上三类时使用该类

C3 • 高级装配

C4 • AOP

AOP 作用:

把横切关注点与业务逻辑相分离。在 oop 中，正是这种分散在各处且与对象核心功能无关的代码（横切代码）的存在，使得模块复用难度增加。AOP 则将封装好的对象剖开，找出其中对多个对象产生影响的公共行为，并将其封装为一个可重用的模块，这个模块被命名为切面 aspect。切面将那些与业务无关却被业务模块共同调用的逻辑提取并封装起来，减少系统中的重复代码，降低模块间的耦合度，同时提高系统的可维护性。

横切关注点可以被模块化为特殊的类，这些类被称为切面 aspect

AOP 与 OOP 相辅相成， 提供了与 OOP 不同的抽象软件结构的视角． **在 OOP 中，我们以类(class)作为我们的基本单元，** 而 AOP 中的基本单元是 **Aspect(切面)**

好处:

1. 每个关注点都集中于一个地方，而不是分散到多出代码
2. 服务模块更简洁。只包含主要关注点的代码

术语

通知: advice

即切面的工作，由 aspect 添加到特定 joinPoint 的一段代码

Spring 切面可以应用五种类型的通知:

- 前置通知 Before: 在目标方法被调用之前调用通知功能
- 后置通知 After: 在目标方法完成之后调用通知，不关心输出
- 返回通知 After-returning: 在目标方法成功执行之后调用通知
- 异常通知 After-throwing: 在目标方法抛出异常之后调用通知

- 环绕通知 Around：包含被通知的方法，在被通知的方法调用之前和之后执行自定义的行为

连接点 Join Point：

连接点是在应用执行过程中能够插入切面的一个点

在 Spring AOP 中，join Point 总是方法的执行点。

切点 pointcut：

切点定义了“何处”，切点会匹配通知所要植入的一个或多个点

Advice 是在 join Point 上执行的，而 Point cut 规定了那些 join Point 可以执行那些 advice

切面 aspect：

切面是通知和切点的结合，通知和切面定义了切面的全部内容。

Aspect 由 pointcut 和 advice 组成，既包含横切逻辑的定义，也包含连接点的定义，SpringAOP 就是负责实施切面的框架，将横切逻辑织入到连接点中。

AOP 的工作在于如何将 advice 织入目标对象的连接点上，包含两个具体工作：

1.如何通过 pointcut 和 advice 定位到特定的 joinPoint 上

2.如何在 advice 中编写切面代码

可以简单认为，使用@aspect 注解的类就是切面

引入 introduction：

引入允许向现有的类添加新方法或属性。

目标对象 Target

织入 advice 的目标对象，advised Object；

因为 Spring AOP 使用运行时代理的方式来实现 aspect，因此 advised Object 总是一个代理对象 proxied Object

注意：advisedObject 不是指原来的类，而是织入 advice 后所产生的类。

织入 Weaving：

把切面应用到目标对象并创建新的代理对象的过程。切面在指定的连接点被织入到目标对象中。在目标对象生命周期中有多个点进行织入：

- 编译期：需要特殊编译器
- 类加载期：切面在目标类被加载到 JVM 时被织入。需要特殊类加载器，在加载之前增强字节码
- 运行期：切面在应用运行时被织入。一般情况下，在织入切面时 AOP 容器会为目标对象动态地建立一个代理对象。这是 Spring AOP 采用的方式

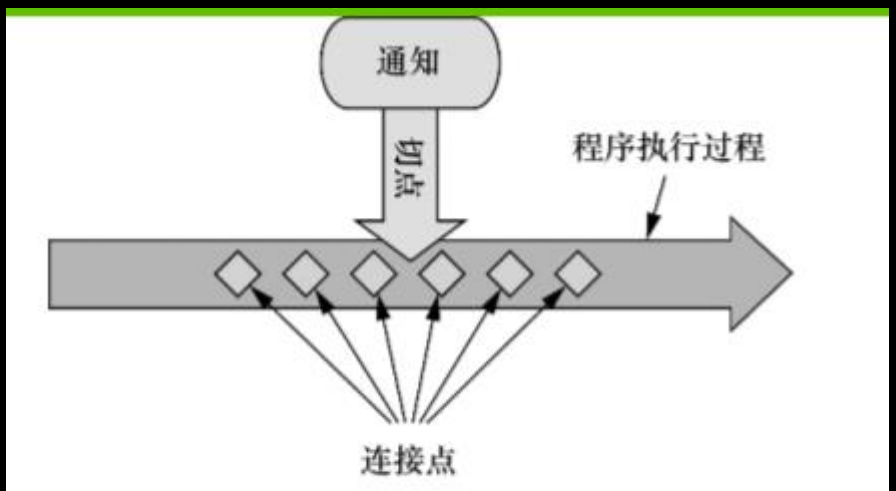


图4.2　在一个或多个连接点上，
可以把切面的功能（通知）织入到程序的执行过程中

AOP 代理：

Spring 中的 AOP 代理可以是 JDK 动态代理。基于接口；也可以是 CGLIB 代理，基于子类。

Spring 模式使用 JDK 动态代理，在需要代理类而不是代理接口的边缘。

@AspectJ 支持

@Aspectj 是一种使用 java 注解来实现 AOP 的编码风格

Aspectj 比 Spring AOP 更加成熟复杂

可以使用 XML 或注解，但都需要导入包：aspectjweaver.jar

使用 XML：　<aop:aspectj-autoproxy/>

表4.1　Spring借助AspectJ的切点表达式语言来定义Spring切面

AspectJ指示器	描　述
arg()	限制连接点匹配参数为指定类型的执行方法
@args()	限制连接点匹配参数由指定注解标注的执行方法
execution()	用于匹配是连接点的执行方法
this()	限制连接点匹配AOP代理的bean引用为指定类型的类
target	限制连接点匹配目标对象为指定类型的类
@target()	限制连接点匹配特定的执行对象，这些对象对应的类要具有指定类型的注解
within()	限制连接点匹配指定的类型
@within()	限制连接点匹配指定注解所标注的类型（当使用Spring AOP时，方法定义在由指定的注解所标注的类里）
@annotation	限定匹配带有指定注解的连接点

使用指示器可以限定范围：

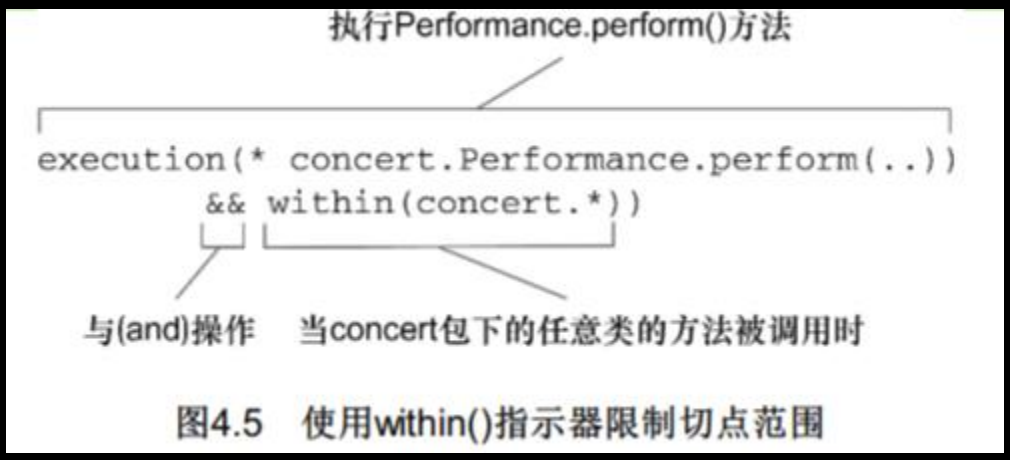


图4.5 使用within()指示器限制切点范围

图4.4 使用AspectJ切点表达式来选择Performance的perform()方法

使用 execution（）指示器选择类的方法，方法表达式以*开头，表示我们不关心返回值类型，然后指定全限定类名和方法名，使用..表明切点不关心参数

表4.2 Spring使用AspectJ注解来声明通知方法

注　解	通　知
@After	通知方法会在目标方法返回或抛出异常后调用
@AfterReturning	通知方法会在目标方法返回后调用
@AfterThrowing	通知方法会在目标方法抛出异常后调用
@Around	通知方法会将目标方法封装起来
@Before	通知方法会在目标方法调用之前执行

//使用 Aspectj 的方式，必须导入 com.springsource.org.aop 等鸡公煲
//使用@Aspectj 表示这是一个切点

切点表达式的重用

通常一个@Aspectj 注解的很多切点表达式是重复的，为了避免讨厌的重复：
@Pointcut 注解能够在一个@Aspectj 内定义可重用的切点
原来每一个@Aspectj 注解就需要一个切点表达式

```
@Before("execution(**edwin.aop.Performance.perform(..))")
public void silenceCellPhone() {
    System.out.println("Silencing cell phone");
}
```

注意注意：**后面有一个空格

现在，使用@pointcut 定义一个新方法，

```
//使用@pointcut来减少切点表达式的重复：
@Pointcut("execution(**edwin.aop.Performance.perform(..))")
public void p() {
    //do nothing
}
```

后面的@pointcut 直接使用该方法即可

```
@Before("p()")
public void takeSeats() {
    System.out.println("Taking seats");
}
```

搭配完@Aspectj，但是并没有启动，需要使用前面的设置代理方法来启动这些注解，可以使用 JavaConfig，也可以使用 XML

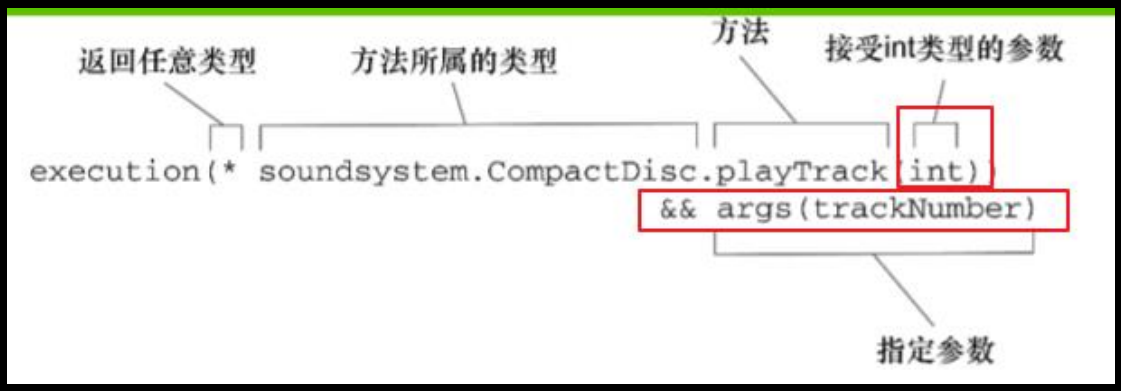
在 Xml 中，通过 Spring 的 AOP 命名空间启用 Aspectj 自动代理：

环绕通知

最为强大的通知，可以使用之来代替前置通知和后置通知

```
/*
 * 环绕通知
 */
@Pointcut("execution(** edwin.aop.Performance.perform(..))")
public void p() {}
@Around("p()")
public void watchPerformance(ProceedingJoinPoint pjp) {//必须要使用ProceedingJoinPoint对象
    try {
        System.out.println("Silencing cell phone");
        System.out.println("Taking seats");
        pjp.proceed();//接入点，即该方法执行，用于划分前置与后置
        System.out.println("好！好！好！鼓掌！");
    }catch (Throwable e) {
        System.out.println("Demanding a refund");
    }
```

切点表达式的参数



通过注解引入新功能

Java 是静态语言，一旦编译就不能添加新的方法，但是通过 Spring 中的引入 AOP，切面可以为 Spring bean 添加新方法

```
package concert;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
public class EncoreableIntroducer {

  @DeclareParents(value="concert.Performance+",
                  defaultImpl=DefaultEncoreable.class)
  public static Encoreable encoreable;

}
```

可以看到，EncoreableIntroducer是一个切面。但是，它与我们之前所创建的切面不同，它并没有提供前置、后置或环绕通知，而是通过@DeclareParents注解，将Encoreable接口引入到Performance bean中。

@DeclareParents注解由三部分组成：

- value属性指定了哪种类型的bean要引入该接口。在本例中，也就是所有实现Performance的类型。（标记符后面的加号表示是Performance的所有子类型，而不是Performance本身。）
- defaultImpl属性指定了为引入功能提供实现的类。在这里，我们指定的是DefaultEncoreable提供实现。
- @DeclareParents注解所标注的静态属性指明了要引入了接口。在这里，我们所引入的是Encoreable接口。

和其他的切面一样，我们需要在Spring应用中将EncoreableIntroducer声明为一个bean：

```
<bean class="concert.EncoreableIntroducer" />
```

在 XML 中声明切面

注解配置 优于 Java 配置 优于 XML 配置

表4.3　Spring的AOP配置元素能够以非侵入性的方式声明切面

AOP配置元素	用　途
<aop:advisor>	定义AOP通知器
<aop:after>	定义AOP后置通知（不管被通知的方法是否执行成功）
<aop:after-returning>	定义AOP返回通知
<aop:after-throwing>	定义AOP异常通知
<aop:around>	定义AOP环绕通知
<aop:aspect>	定义一个切面
<aop:aspectj-autoproxy>	启用@AspectJ注解驱动的切面
<aop:before>	定义一个AOP前置通知
<aop:config>	顶层的AOP配置元素。大多数的<aop:*>元素必须包含在<aop:config>元素内
<aop:declare-parents>	以透明的方式为被通知的对象引入额外的接口
<aop:pointcut>	定义一个切点

程序清单4.9　通过XML将无注解的Audience声明为切面

```
<aop:config>
  <aop:aspect ref="audience">          <-- 引用 audience Bean

    <aop:before
      pointcut="execution(** concert.Performance.perform(..))"
      method="silenceCellPhones"/>
                                                              表演之前
    <aop:before
      pointcut="execution(** concert.Performance.perform(..))"
      method="takeSeats"/>

    <aop:after-returning                          <-- 表演之后
      pointcut="execution(** concert.Performance.perform(..))"
      method="applause"/>
                                                  表演失败之后
    <aop:after-throwing                           <--
      pointcut="execution(** concert.Performance.perform(..))"
      method="demandRefund"/>

  </aop:aspect>
</aop:config>
```

这里使用 XML 的切点实现太落后，使用注解更好，就不深入学习了

注入 Aspectj 切面

Spring AOP 相对于 AspectJ 是一个弱的 AOP 解决方案

对于大部分功能来讲，AspectJ 切面与 Spring 是相互独立的。但是有些切面可能会依

赖于其他类，如果在执行通知时，切面依赖于一个或多个类，我们可以在切面内部实例化这些协作对象，但更好的方式是，我们借助于 Spring 的 DI 把 bean 装配到 AspectJ 中

```
package concert;
public aspect CriticAspect {
  public CriticAspect() {}

  pointcut performance() : execution(* perform(..));

  afterReturning() : performance() {
    System.out.println(criticismEngine.getCriticism());
  }

  private CriticismEngine criticismEngine;

  public void setCriticismEngine(CriticismEngine criticismEngine) {
    this.criticismEngine = criticismEngine;
  }
```

注入 CriticismEngine

Spring MVC

SpringMVC 框架中，Spring 将请求在调度 Servlet，处理器映射 handler Mapping、控制器、视图解析器 view resolver 之间移动。

SpringMVC 请求

用户在 WEB 浏览器中点击连接或提交表单的时候，请求开始工作，请求从浏览器到获取响应返回，会历经许多站，每站都会留下一些信息同时带上一些信息

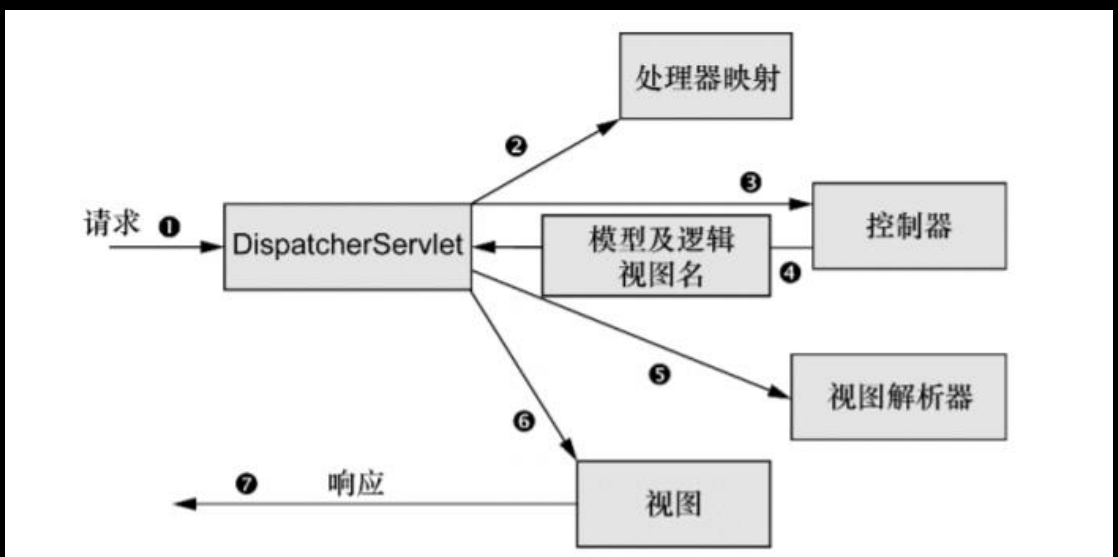


图5.1 一路上请求会将信息带到很多站点，并生产期望的结果

所有请求发出后都会通过一个前端控制器（front controller）Servlet，前端控制器是常用的 WEB 应用程序模式，在这里一个单实例的 Servlet 将请求委托给应用程序的其他组件来执行实际的处理。SpringMVC 中 DispatcherServlet 就是前段控制器，它是 SpringMVC 中的核心。

DispatcherServlet 将请求发送给 SpringMVC 控制器 controller，控制器是处

理请求的组件，可以多个，通过查询处理器映射来确定下一站

控制器在完成逻辑处理后，产生一些需要返回到浏览器上显示（以 HTML）的信息（即模型 Model），所以，信息要发送给一个试图 view，通常会是 jsp。

控制器最后将模型数据打包，标出用于渲染输出的视图名，然后将请求连同模型和视图名发送给 4 会 DispatcherServlet，这样控制器就不会与特定的视图耦合。

Spring 实战这本书后面 MVC 看不懂了，算了，不看了
现在基本了解了一下 Spring

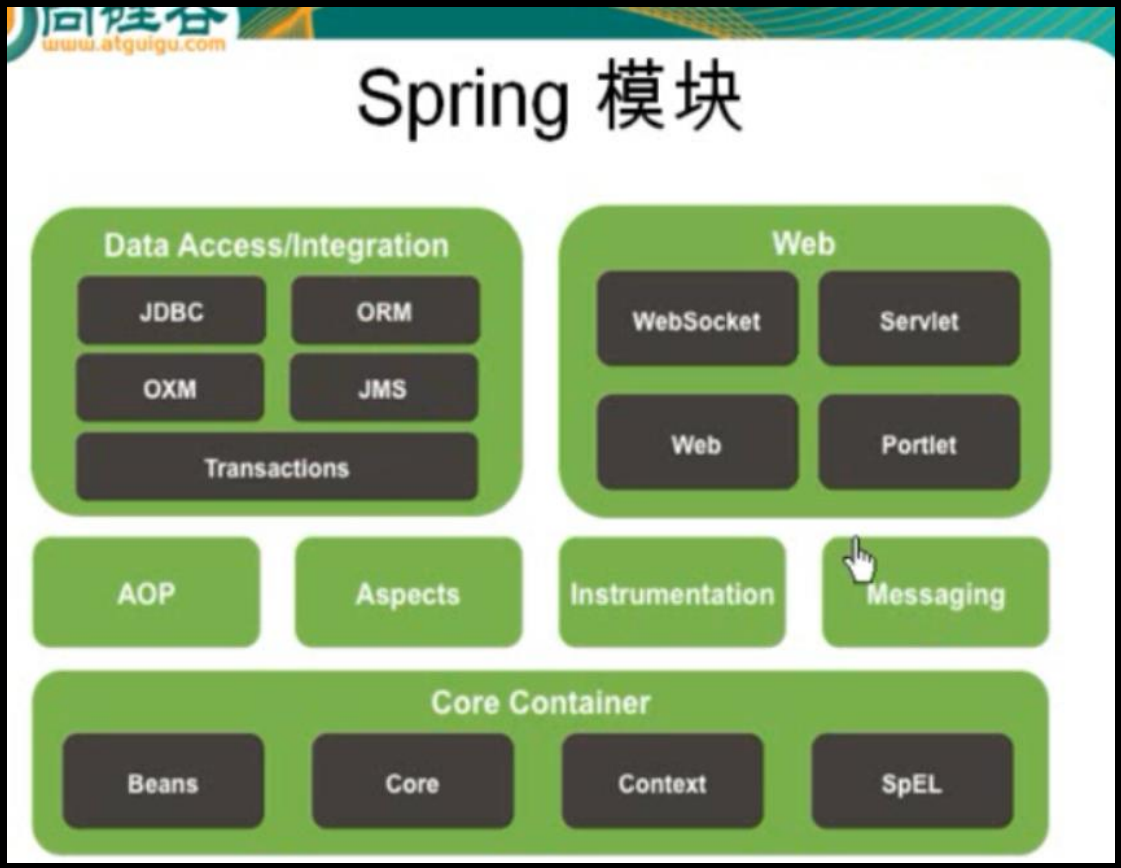
Spring 视频学习

Spring 是什么

- Spring 起于 2003
- 开源框架
- 为简化企业级应用而生，Spring 可以使简单的 javabean 实现以前只有 EJB 才能实现的功能。

Edwin Xu

- Spring 是一个 IOC（DI）和 AOP 容器框架
- 轻量级：非侵入型的
- Spring 是一个容器，因为它管理、包含应用对象的生命周期
- 框架：使用简单的组件配置组合成复杂应用，可以使用 XML、Java 注解组合这些对象。
- 一站式：可以整合各种企业级开源框架和优秀第三方库。



HelloWorld

```java
package com.edwin.spring.beans;

public class HelloWorld {
    private String name;
    public void setName(String name) {
        this.name = name;
        System.out.println("this is setName");
    }
    public void hello() {
        System.out.println("Hello "+name);
    }

    public HelloWorld() {
        System.out.println("this is a constructor");
    }
}
```

```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/bea

    <!-- 配置bean -->
    <bean id="id" class="com.edwin.spring.beans.HelloWorld">
        <!-- 这个Name和setName方法是对应的 -->
        <property name="name" value="Spring"></property>
    </bean>

</beans>
```

```java
ic class Main {

public static void main(String[] args) {
    //创建对象
    //HelloWorld helloWorld = new HelloWorld();
    //////赋值
    //helloWorld.setName("Edwin");

    //1.创建SpringIOC容器对象
    ApplicationContext c= new ClassPathXmlApplicationContext("applicationContext.xml");

    //2.从IOC容器中获取Bean实例
    HelloWorld helloWorld = (HelloWorld)c.getBean("id");

    //3.调用hello方法

    helloWorld.hello();
```

即使只创建容器：
会调用HelloWorld的构造器，然后初始化XML

中响应的Bean，bean里面的属性等都会被初始化

IOC | DI 概述

- IOC: Inversst of Control 反转资源获取的方向，传统的资源查找方式要求向容器发起请求查找资源，容器会适时返回资源。
  使用 IOC，容器主动将资源推动给它所管理的组件，组件所需要做的就是选择合适

的方式来接收资源（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（如 setter 方法）接受来自如容器的资源注入。
- IOC 前生：
  1. 分离接口与实现
  2. 工厂设计模式
  3. 反转控制

Bean 配置

IOC 中配置 bean

- Class：bean 的全类名，通过反射的方式在 IOC 中创建 bean，所以要求 **bean 中必须有无参数的构造器。**
- Id：标识容器中 的 bean，唯一。
- IOC 容器读取 bean 配置创建 bean 实例之前，必须对它进行实例化。
- Spring 提供两种类型的 IOC 容器实现：
  1. BeanFactory：IOC 容器的基本实现，是 Spring 的基本设施，面向 Spring 本身。几乎不直接使用。
  2. ApplicationContext 更高级，是 BeanFactory 的子接口。面向使用 Spring 框架的开发者。

  注意：两种方式下配置文件相同

ApplicationContext

主要实现类：

- ClassPathXmlApplicationContext：从类路径加载配置文件，是 ApplicationContext 的实现类
- FileSystemXmlApplicationContext：从文件系统中加载文件

ConfigurableApplicationContext 扩展于 ApplicationContext，refresh（）、close（）可以启动关闭上下文

•• 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 应用准备的，允许从 WEB 根目录完成初始化工作

•• getBean()方法的参数也可以是 ClassName.class，但是这样做仅限于 XML 中的 bean 只有一个该 class 类，否则编译时不知道使用那个因而报错。

HelloWorld helloWorld = (HelloWorld)c.getBean("id");利用类型返回 IOC 中的 bean，但是这个 id 必须是唯一的

注入依赖的方式

1. 属性注入：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-value
   - 属性注入是最常用的。
2. 构造器注入：通过构造方法注入 bean 的属性值或依赖的对象，它保证 bean 实例在实例化后就可以使用。
   - 使用**<constructor-arg>**标签声明属性：
     注意：构造器需要几个参数，就需要几个 sconstructor-arg>标签。另外还有属性：index——顺序；type——数值类型（区分重载的构造器：int，double,string…）

```
    <constructor-arg value="Audi" index="0"></constructor-arg>
    <constructor-arg value="ShangHai" index="1"></constructor-arg>
    <constructor-arg value="300000" type="double"></constructor-arg>
</bean>
```

3. 工厂方法注入（很少使用）：

String 属性配置细节

- 上面的属性值都可以使用标签（子节点）表示
- 如果含特殊字符如<,>，则需要使用 ：**<![CDATA[特殊字符]]>**
- **建立 bean 之间的引用关系：ref**
  使用 ref 可以使 bean 之间相互访问，对于属性值是另一个 bean 的情况即可使用 ref 指向。（如人有车）
  有三种写法：
  A. <property name = “” ref = “”>
  B. <property name = “”>
  <ref bean=”car” />
  </property>
  C. 内部 bean：在内部在添加一个 bean 标签（注意：内部 bean 不能被外部引用）

Null 与级联属性

- 可以使用<null/>元素标签为 bean 的字符串或其他对象类型的属性注入 null 值

```
<constructor-arg><null/></constructor-arg>
```

-

```
<!-- 为级联属性赋值 -->
<property name="car.maxSpeed" value="250"></property>
```

用不着

- <sonstructor-arg>属性：构造器装配

集合属性：

- 使用内置的 XML 标签如<list>,<set>,<map>来配置集合属性。

```
<property name="cars">
    <!-- 使用 list 节点为 List 类型的属性赋值 -->
    <list>
        <ref bean="car"/>
        <ref bean="car2"/>
    </list>
</property>
```

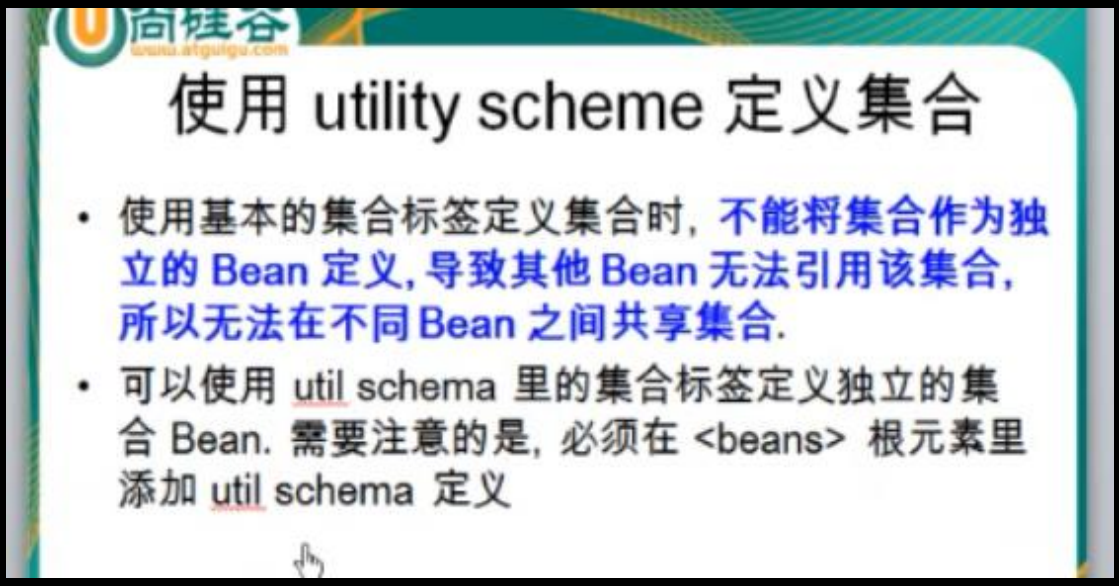
```
<property name="name" value="Mike"></property>
<property name="age" value="27"></property>
<property name="cars">
    <!-- 使用 list 节点为 List 类型的属性赋值 -->
    <list>
        <ref bean="car"/>
        <ref bean="car2"/>
        <bean class="com.atguigu.spring.beans.Car">
            <constructor-arg value="Ford"></constructor-arg>
            <constructor-arg value="Changan"></constructor-arg>
            <constructor-arg value="200000" type="double"></constructo
        </bean>
    </list>
```

- Map: 使用 Map 标签时, 标签里可以含多个<entry>子标签标签

- 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.

```
<property name="cars">
    <map>
        <entry key="AA" value-ref="car"></entry>
        <entry key="BB" value-ref="car2"></entry>
    </map>
</property>
```

- Utility Scheme 定义集合



使用 p 命名空间:

```
p:city="BeiJing" p:street="WuDaoKou"></bean>
```

更加简单

自动装配

- XML 自动装配:
  1.byName():可以使用 autowire 进行自动装配，byName 是根据 bean 及 setter 的名字来装配。byType 则根据类型自动装配。

继承

可以使用 parent 属性指向父类

使用 abstract 属性： abstract=“true” 定义抽象 bean，不能被实例化，只能被继承

依赖关系

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

<!-- 要求再配置 Person 时, 必须有一个关联的 car! 换句话说 person 这个 bean 依赖于 Car 这个 bean -->
<bean id="person" class="com.atguigu.spring.beans.autowire.Person"
    p:name="Tom" p:address-ref="address2" depends-on="car"></bean>
```

使用 depend-on 指定依赖的 bean

Bean 作用域

```
使用 bean 的 scope 属性来配置 bean 的作用域
singleton: 默认值. 容器初始时创建 bean 实例. 在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 的实例. 而在每次请求时都创建一个新的 Bean 实例, 并返回.
```

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。

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，所有在大框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

- 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}"/>

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}"/>
```

- 逻辑运算符号：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：引用 Bean、属性和方法（2）

- **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：

```
<property name="initValue"
    value="#{T(java.lang.Math).PI}"></property>
```

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

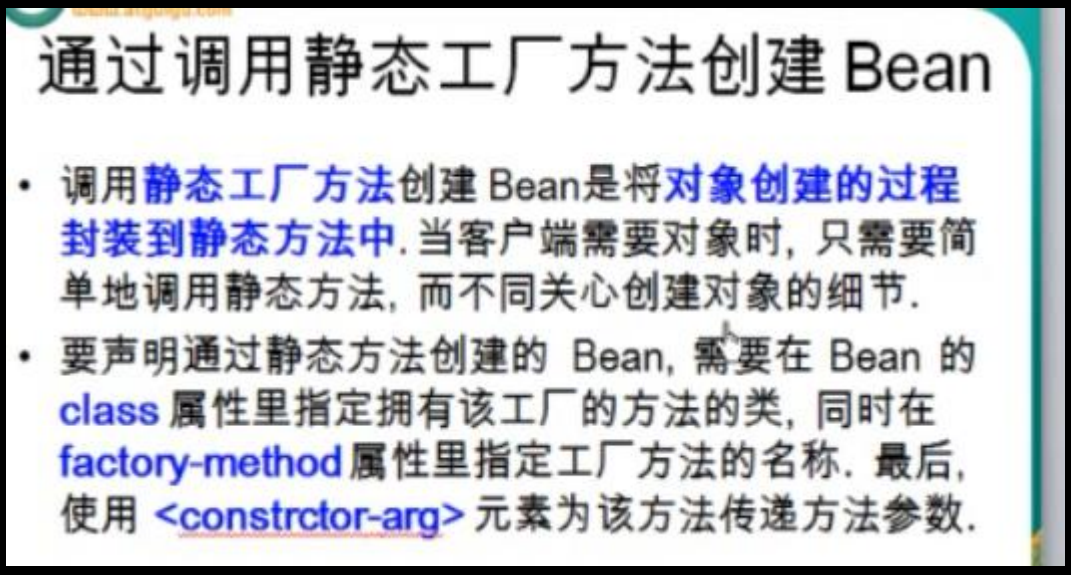
```
<bean id="person" class="com.atguigu.spring.beans.spel.Person">
    <!-- 使用 SpEL 来应用其他的 Bean -->
    <property name="car" value="#{car}"></property>
    <!-- 使用 SpEL 来应用其他的 Bean 的属性 -->
    <property name="city" value="#{address.city}"></property>
    <!-- 在 SpEL 中使用运算符 -->
    <property name="info"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

生命周期管理

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.

工厂方法


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 <constrctor-arg> 元素为该方法传递方法参数.


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 construtor-arg 元素为工厂方法传递方法参数

注解配置

有时间搞搞 Spring 源码

错误日志

XML 第一行

XML 第一行必须是<?xml version="1.0" encoding="UTF-8"?>
不能是空行或其他
不然报错：不允许有匹配 "[xX][mM][lL]" 的处理指令目标。

元素未绑定

XML 的配置时，若出现：
元素 "context:component-scan" 的前缀 "context" 未绑定
之类的问题，则是 XML 的绑定问题：
需要添加：

```
xmlns:context="http://www.springframework.org/schema/context"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context.xsd
   ">
```

小结：
好像需要两处：

1. xmlns:aop=http://www.springframework.org/schema/aop
2. http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop.xsd

log4j 报错

```
log4j:WARN No appenders could be found for logger
log4j:WARN Please initialize the log4j system properly.
```

解决：
在 src 下建立：log4j.properties：

```
# Configure logging for testing: optionally with log file
log4j.rootLogger=WARN, stdout
# log4j.rootLogger=WARN, stdout, logfile
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
```

```
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n

log4j.appender.logfile=org.apache.log4j.FileAppender
log4j.appender.logfile.File=target/spring.log
log4j.appender.logfile.layout=org.apache.log4j.PatternLayout
log4j.appender.logfile.layout.ConversionPattern=%d %p [%c] - %m%n
```