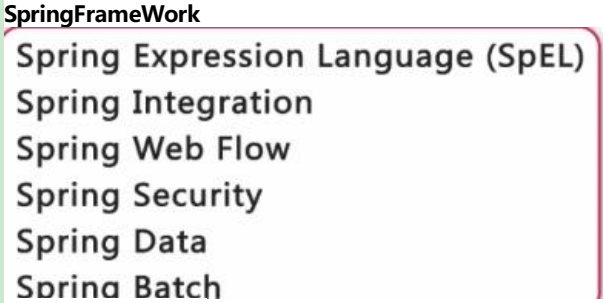
# Spring1

Spring是什么呢？
Spring简介
IOC(配置，注解)
Bean（配置，注解）
AOP（配置，注解，AspectJ，API）

**SpringFrameWork**



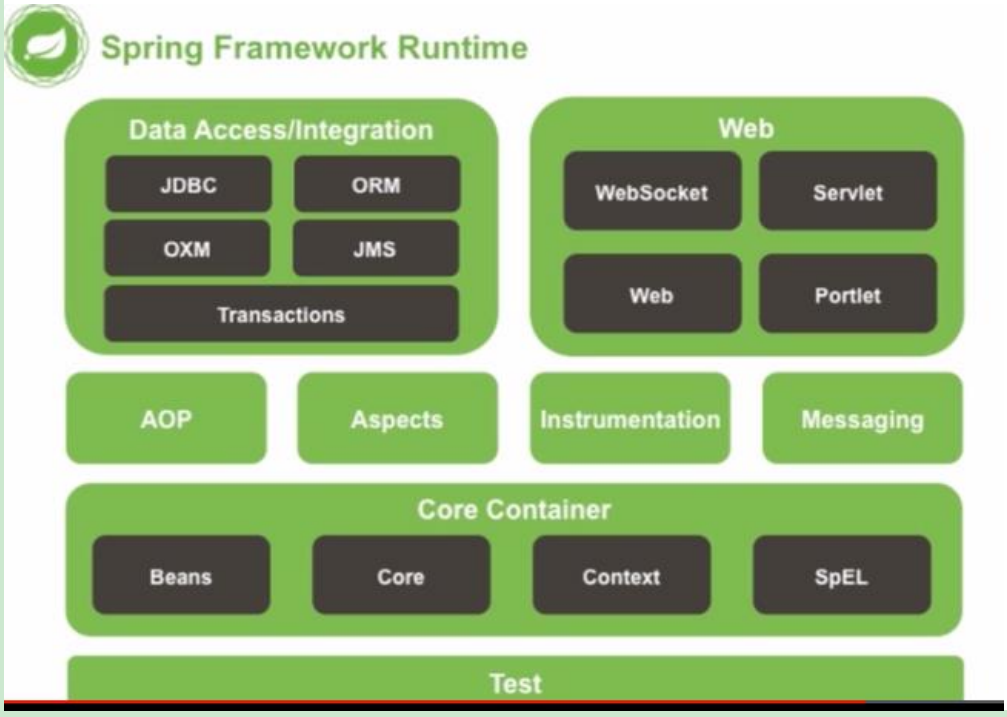

## 如何学习Spring

掌握用法
深入研究
不断实践
反复总结
再次深入理解与实践

Spring是一个开源框架，为了解决企业应用开发的复杂性而创建的，但现在已经不止应用于企业应用，
Spring，是java界，可以说最火的框架，是一个**轻量级的控制反转（IoC）和面向切面（AoP）**的容器框架，从大小，与开销俩方面而言来说，Spring都是轻量的
通过控制反转（IoC）的技术达到松耦合的目的
提供了面向切面编程的丰富支持，允许通过分离企业应用的业务逻辑与系统级的服务进行内聚性的开发
包括并管理应用对象的配置和生命周期，这个意义上是一种容器
将简单的配置应用，组合为复杂的应用，这个意义上是框架

在Spring上开发应用简单，
在Spring上开发应用方便，
在Spring上开发应用快捷，

所有的对象都在spring叫bean，而且由于它这个面向接口编程的应用，使得我们开发中是非常快捷的，
Spring带来了**复杂**的JAVAEE开发的春天



**容器**
提供了对多种技术的支持，
--JMS
--MQ支持
--UnitTest
。。。。。
AOP（事务管理，日志等）
提供了众多方便应用辅助的类（JDBC Templace等）
对主流应用框架（Hibernate等）提供了良好的支持

**适用范围：**
构建企业应用（SpringMvc+Spring+Hibernate/ibatis）
单独的使用Bean容器（Bean管理）
单独使用AOP进行切面
其他的Spring功能，如：对消息的支持等，
在互联网的应用。、。。。。。

**软件框架**（Software framework），通常指的是为了实现某个业界标准或完成特定基本任务的软件组件规范，也指为了实现某个软件组件规范时，提供规范所要求之基础功能的软件产品。

框架的功能类似于基础设施，与具体的软件应用无关，但是提供并实现最为基础的软件架构和体系。软件开发者通常依据特定的框架实现更为复杂的商业运用和业务逻辑。这样的软件应用可以在支持同一种框架的软件系统中运行。

简而言之，框架就是制定一套规范或者规则（思想），大家（程序员）在该规范或者规则（思想）下工作。或者说使用别人搭好的舞台来做编剧和表演。

**框架的特点：**

半成品，

封装了特定的处理流程和控制逻辑，

成熟的，不断升级改进的软件

**框架与类库的区别**

框架一般是封装了逻辑的，高内聚的，类库是松散的工具组合，

框架专注某一个领域，类库是更通用的。

**框架一般配有完整的使用说明与使用文档，**

软件系统日趋复杂

重用度高，开发效率高

软件设计人员要专注对领域的了解，使需求分析更充分

易于上手，快速解决问题，更容易去思考其他部分的问题

# IOC

**接口：**

- 用于沟通的中介物的抽象化
- 实体把自己提供给外界的一种抽象化说明，用以由内部操作分离出外部沟通方法，使其能被修改内部而不影响外界其他实体与其交互的方式
- 对应Java接口即声明，声明了哪些方法是对外公开提供的
- 在Java8中，接口可以拥有方法体

**面向接口编程在IOC中获得广泛的应用**

面向接口编程：

# 面向接口编程

- 结构设计中，分清层次及调用关系，每层只向外（上层）提供一组功能接口，各层间仅依赖接口而非实现类
- 接口实现的变动不影响各层间的调用，这一点在公共服务中尤为重要
- “面向接口编程”中的“接口”是用于隐藏具体实现和实现多态性的组件
- *例子*

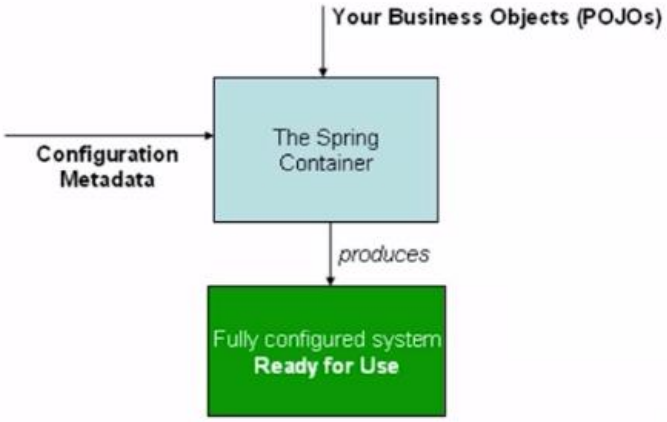
**IOC是什么呢**

**控制反转：**控制权的转移，应用程序本身不负责依赖对象的创建与维护，而是由外部容器负责创建与维护，

**DI（依赖注入）是一种实现方式**

**目的：**创建对象并且组装对象之间的关系　　**这是IOC容器的说明**

Figure 4.1. The Spring IoC container




- 2004年，Martin Fowler探讨了同一个问题，既然IOC是控制反转，那么到底是“哪些方面的控制被反转了呢？”，经过详细地分析和论证后，他得出了答案：“获得依赖对 象的过程被反转了”。控制被反转之后，获得依赖对象的过程由自身管理变为了由IOC容器主动注入。于是，他给“控制反转”取了一个更合适的名字叫做“依赖 注入（Dependency Injection）”。他的这个答案，实 际上给出了实现IOC的方法：注入。所谓依赖注入，就是由IOC容器在运行期间，动态地将某种依赖关系注入到对 象之中。

**IOC使用的步骤**

1 找IOC容器

2 容器返回对象

3 使用对象

在IOC容器中，它把所有的对象都称之为bean

**Spring对整个Spring的使用的方法有俩种**，1 基于XML的配置，2是通过注解

**不必自己创建对象了，**

IOC机制提供这种创建对象的功能，

面向接口编程了，具体的实现也被IOC隐藏了

不用手动的管理对象，被IOC帮忙管理了

- 不必自己创建对象了
- IOC机制就提供了
- 面向接口编程了
- IOC藏实现了
- 不管对象了
- IOC管了
- 变好了
- IOC
- 爽

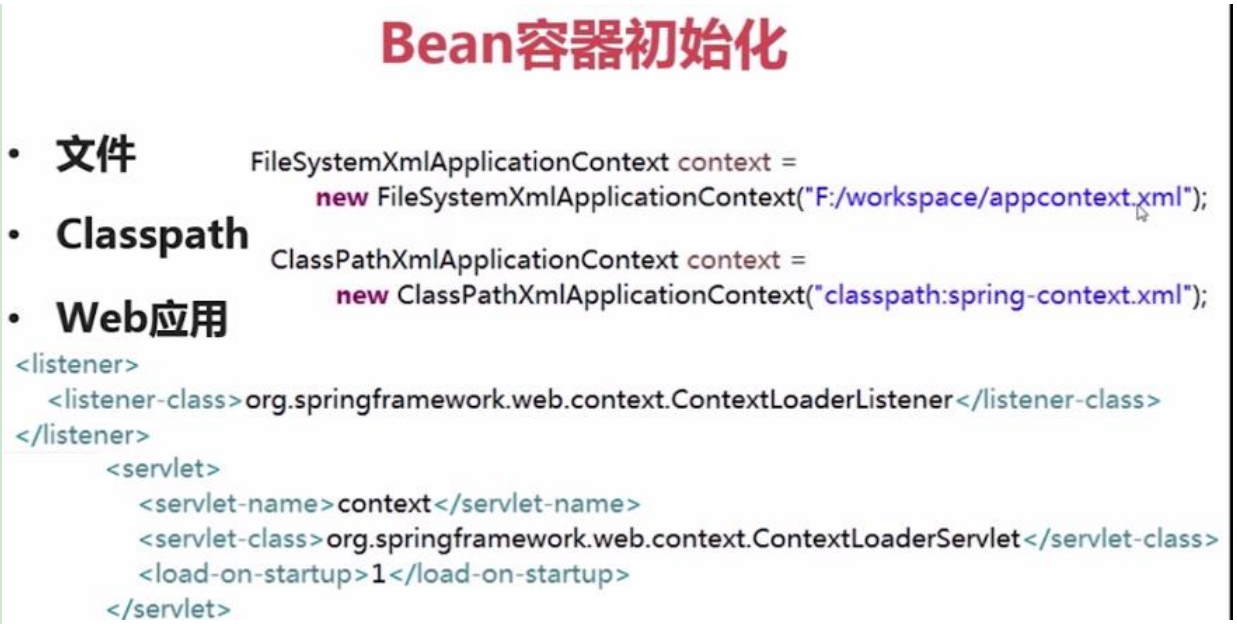
这样不用手动的new对象，

- 下载junit-*.jar并引入工程
- 创建UnitTestBase类，完成对Spring配置文件的加载、销毁
- 所有的单元测试类都继承自UnitTestBase，通过它的getBean方法获取想要得到的对象
- 子类（具体执行单元测试的类）加注解：@RunWith(BlockJUnit4ClassRunner.class)
- 单元测试方法加注解：@Test
- 右键选择要执行的单元测试方法执行或者执行一个类的全部单元测试方法

## Bean容器初始化

- 基础：两个包
  - org.springframework.beans
  - org.springframework.context
  - BeanFactory提供配置结构和基本功能，加载并初始化Bean
  - ApplicationContext保存了Bean对象并在Spring中被广泛使用
- 方式，ApplicationContext
  - 本地文件
  - Classpath
  - Web应用中依赖servlet或Listener

classpath是个相对路径，



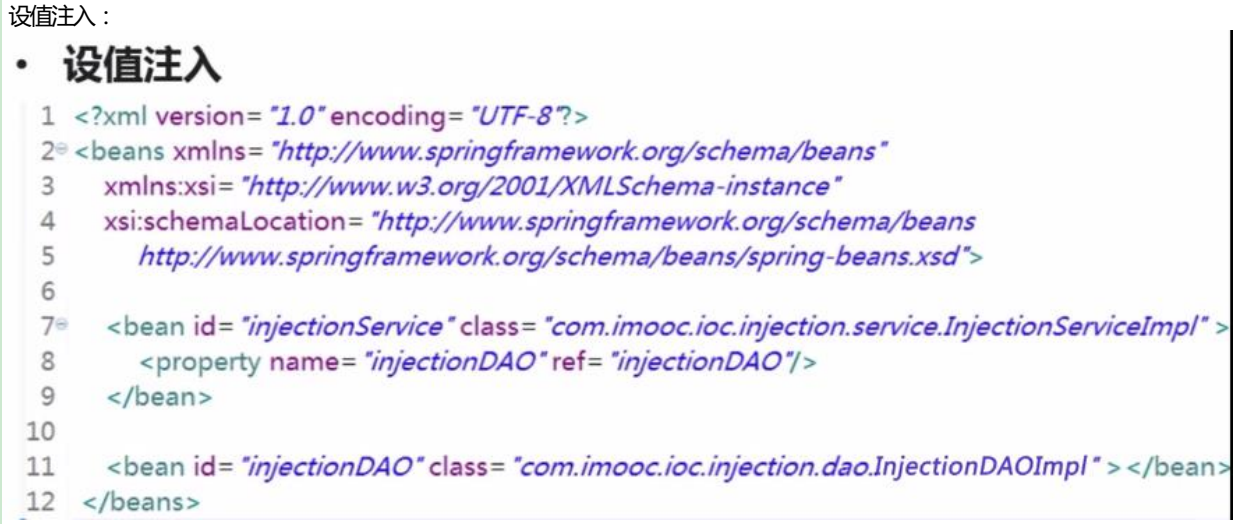

Spring注入是指在启动Spring容器加载bean配置的时候，完成对变量的赋值行为

## 常用的俩种注入方式

## ----设值注入

## ----构造注入

设值注入：




构造注入：

- 构造注入

```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.xsd">

    <bean id="injectionService" class="com.imooc.ioc.injection.service.InjectionServiceImpl" >
        <constructor-arg name="injectionDAO" ref="injectionDAO"/>
    </bean>

    <bean id="injectionDAO" class="com.imooc.ioc.injection.dao.InjectionDAOImpl" ></bean>
</beans>
```

#spring——Spring注入方式—设值注入#<br>

Spring是指在启动Spring容器加载bean配置的时候，完成对变量的赋值行为

常用注入方式：设值注入，构造注入

注意：参数的名称必须保持一致！！

无论是设置注入，还是构造注入，都是注入，都有这个DAO层的 Bean ，然后通过service层的Bean加载

```xml
<bean id="injectionDAO" class="com.imooc.ioc.injection.dao.InjectionDAOImpl">
</bean>
```

设值注入（不需要显示地调用set方法，会根据xml的相关配置自动进行调用。）
**在service方法中有一个方法是set方法，我们是通过xml文件的配置，内部自动就完成了**

利用属性或成员变量的set方法进行注入

**其中property里面的name是需要注入参数的成员变量的名称，ref是注入参数引入bean的名称**

构造注入：

**用constructor-arg标签添加注入参数，其中name是构造函数中要传入的参数名，ref是参数的引用类型。**


**Bean配置项**
**Bean的作用越**
**Bean的生命周期**
**Bean的自动装配**
**Resources&ReasourceLoder**

- singleton：单例，指一个Bean容器中只存在一份
- prototype：每次请求（每次使用）创建新的实例，destroy方式不生效
- request：每次http请求创建一个实例且仅在当前request内有效
- session：同上，每次http请求创建，当前session内有效
- global session：基于portlet的web中有效（portlet定义了global session），如果是在web中，同session

**官网上的解释**

| Scope | Description |
|---|---|
| singleton | (Default) Scopes a single bean definition to a single object instance per Spring IoC container. |
| prototype | Scopes a single bean definition to any number of object instances. |
| request | Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext. |
| session | Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext. |
| global session | Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet |

**singleton是单例模式，上面具体有说，但是prototype，是每次只要请求，就会生成不一样的实例**

**单例模式spring中默认的格式**
**这个生命周期其实非常的复杂 但是这里只是可能为了理解将其说成了这样**
**生命周期**
定义
初始化
使用
销毁

初始化一般有俩种方式，第一种，是实现

- 实现org.springframework.beans.factory.InitializingBean接口，覆盖afterPropertiesSet方法

```
public class ExampleInitializingBean implements InitializingBean {

    @Override
    public void afterPropertiesSet() throws Exception {
        //do something
    }

}
```

第二种是配置属性的是时候 init-method

```xml
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

```
public class ExampleBean {

    public void init() {
        // do some initialization work
    }

}
```

Bean的销毁

第一种

- 实现org.springframework.beans.factory.DisposableBean接口，覆盖destroy方法

```
public class ExampleDisposableBean implements DisposableBean {

  @Override
  public void destroy() throws Exception {
    //do something
  }

}
```

第二种

在bean管理时候配置属性destory-menthod

```xml
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

```
public class ExampleBean {

    public void cleanup() {
        // do some destruction work (like releasing pooled connections)
    }

}
```

另外还有一种全局的配置方式

- 配置全局默认初始化、销毁方法

```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.xsd"
        default-init-method="init" default-destroy-method="destroy">

</beans>
```

在这个执行顺序中， 下面这是实现的顺序

bean start() 接口实现开始方法.....

bean start() 自定义开始方法.....

接口实现的顺序 大于我们在bean中实现的顺序， 而默认的初始化方法是不实现的，因为spring会检测我们是否在xml或接口中，配置，如果有的话，这些默认的就自动不会生效，当然你要强制生效也可以的，

bean中是可以没有我们默认定义的这俩个方法的

```xml
default-init-method="defautInit" default-destroy-method="defaultDestroy">
```

如果我们没有实现初始化，与销毁方法，也没有关系，这是一个可选的方法

如果我们没有实现

```xml
init-method="start" destroy-method="stop"
```

这个定义的方法的话，强行使用，只会出错

# Aware

- Spring中提供了一些以Aware结尾的接口，实现了Aware接口的bean在被初始化之后，可以获取相应资源
- 通过Aware接口，可以对Spring相应资源进行操作（一定要慎重）
- 为对Spring进行简单的扩展提供了方便的入口

通过Aware获得的资源，我们一定要慎重，因为我们获取很可能是核心的内容，当我们队它进行了一些操作，比如destory，就很麻烦了

ApplicationContextAware这个接是为了获取上下文信息

这个接口的作用就是获取到我们获取到的bean

BeanNameAware

这个接口的作用就是获取Bean的Name

```
public class MoocApplicationContext  implements  ApplicationContextAware   {



public void setApplicationContext(ApplicationContext applicationContext) throws Bean
sException {

      System.out.println("MoocApplicationContext 是 :"+applicationContext.getBean("m
oocApplicationContext"));

}

}
```

| Name | Injected Dependency |
| --- | --- |
| ApplicationContextAware | Declaring ApplicationContext |
| ApplicationEventPublisherAware | Event publisher of the enclosing ApplicationContext |
| BeanClassLoaderAware | Class loader used to load the bean classes. |
| BeanFactoryAware | Declaring BeanFactory |
| BeanNameAware | Name of the declaring bean |
| BootstrapContextAware | Resource adapter BootstrapContext the container runs in. Typically available only in JCA aware ApplicationContexts |
| LoadTimeWeaverAware | Defined *weaver* for processing class |

| Name | Injected Dependency |
| --- | --- |
| MessageSourceAware | Configured strategy for resolving messages (with support for parametrization and internationalization) |
| NotificationPublisherAware | Spring JMX notification publisher |
| PortletConfigAware | Current PortletConfig the container runs in. Valid only in a web-aware Spring ApplicationContext |
| PortletContextAware | Current PortletContext the container runs in. Valid only in a web-aware Spring ApplicationContext |
| ResourceLoaderAware | Configured loader for low-level access to |

| Name | Injected Dependency |
| --- | --- |
| ServletConfigAware | Current ServletConfig the container runs in. Valid only in a web-aware Spring ApplicationContext |
| ServletContextAware | Current ServletContext the container runs in. Valid only in a web-aware Spring ApplicationContext |

Aware 接口

当 bean 实现 Sping 中以 Aware 结尾的接口后，初始化可以获取相应资源

Bean 实现 ApplicationContextAware 接口

public void setApplicationContext(ApplicationContext applicationContext)

通过 applicationContext.getBean("xxx") 获取 bean

Bean 实现 BeanNameAware 接口

public void setBeanName(String name)

通过 this.beanName = name 获取 bean

# Bean的自动装配（Atuowiring）

# Bean的自动装配（Autowiring）

- **No:**不做任何操作
- **byname:**根据属性名自动装配。此选项将检查容器并根据名字查找与属性完全一致的bean，并将其与属性自动装配
- **byType**:如果容器中存在一个与指定属性类型相同的bean，那么将与该属性自动装配；如果存在多个该类型bean，那么抛出异常，并指出不能使用byType方式进行自动装配；如果没有找到相匹配的bean，则什么事都不发生
- **Constructor**:与byType方式类似，不同之处在于它应用于构造器参数。如果容器中没有找到与构造器参数类型一致的bean，那么抛出异常

这个是官方文档

| Mode | Explanation |
| --- | --- |
| no | (Default) No autowiring. Bean references must be defined via a ref element. Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity. To some extent, it documents the structure of a system. |
| byName | Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired. For example, if a bean definition is set to autowire by name, and it contains a *master* property (that is, it has a *setMaster(..)* method), Spring looks for a bean definition named master, and uses it to set the property. |
| byType | Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use *byType* autowiring for that bean. If there are no matching beans, nothing happens; the property is not set. |
| constructor | Analogous to *byType*, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised. |

Bean 的自动装配（Autowiring）

default-autowire="no/byName/byType/constructor"

byName byType 通过设置注入自动加载参数指定的bean；

byname方法即根据bean的id查找bean。id必须唯一，否则bean容器在加载时遇到有id重复会报错。

constructor 通过构造注入自动加载构造函数参数类型相同的bean;

这个构造器自动装配的时候，是通过类型来匹配的，id是否有好像无所谓
这就是自动装配，就是说依赖注入的自动装来装去

# Resources

针对于资源文件的统一接口

- **Resources**
  - UrlResource：URL对应的资源，根据一个URL地址即可构建
  - ClassPathResource：获取类路径下的资源文件
  - FileSystemResource：获取文件系统里面的资源
  - ServletContextResource：ServletContext封装的资源，用于访问ServletContext环境下的资源
  - InputStreamResource：针对于输入流封装的资源
  - ByteArrayResource:针对于字节数组封装的资源

## ResourceLoader

| Prefix | Example | Explanation |
| --- | --- | --- |
| classpath: | classpath:com/myapp/config.xml | Loaded from the classpath. |
| file: | file:/data/config.xml | Loaded as a URL, from the filesystem. [1] |
| http: | http://myserver/logo.png | Loaded as a URL. |
| (none) | /data/config.xml | Depends on the underlying ApplicationContext. |

在这里我在下面进行一系列的演示吧，下面有代码清单

```
public class MoocResource implements ApplicationContextAware  {

private ApplicationContext applicationContext;


   public void setApplicationContext(ApplicationContext applicationContext)
throws BeansException {
this.applicationContext = applicationContext;
}

public void resource() throws IOException {
      Resource resource= applicationContext.getResource("classpath:config.rxr");
System.out.println(resource.getFilename());
System.out.println(resource.contentLength());
}

}
```

在上面chlasspath为什么能得到，因为我们在把这个文件，放在这个工程的目录中，也就是放在这个工程的classpath中
在下面还有

```
Resource resource= applicationContext.getResource("file:D:\\新建文件夹\\static.txt");
```

这个是在电脑上本地加载，注意**格式**

```
Resource resource= applicationContext.getResource("url:http://wiki.jikexueyuan.com/p
```

```
roject/intellij-idea-tutorial/about-this-tutorial.html");
```

**这个是可以在网络上获取**

```
Resource resource= applicationContext.getResource("config.rxr");
```

**这个是不写直接加载 如果上面都不写，我们当前的applicationContext是通过什么前缀加载的，就是用什么方式加载这个，这个默认是classpath**

# 注解

**Bean的定义，以及作用域的注解实现**

## Bean管理的注解实现及例子

- Classpath扫描与组件管理
- 类的自动检测与注册Bean
- <context:annotation-config/>
- @Component, @Repository, @Service, @Controller
- @Required
- @Autowired
- @Qualifier
- @Resource

## Classpath扫描与组件管理

- 从Spring3.0开始， Spring JavaConfig项目提供了很多特性，包括使用java而不是XML定义bean，比如@Configuration, @Bean, @Import, @DependsOn
- @Component是一个通用注解，可用于任何bean
- @Repository, @Service, @Controller是更有针对性的注解
  - @Repository通常用于注解DAO类，即持久层
  - @Service通常用于注解Service类，即服务层
  - @Controller通常用于Controller类，即控制层（MVC）

**可以用注解来给spring配置，**

许多spring提供的注解可以作为自己的代码，即“元数据注解”，元注解是一个简单的注解，可以应用到另一个注解

# 元注解(Meta-annotations)

- 许多Spring提供的注解可以作为自己的代码，即“元数据注解”，元注解是一个简单的注解，可以应用到另一个注解

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component // Spring will see this and treat @Service in the same way as @Component
public @interface Service {

    // ....
```

- 除了value()，元注解还可以有其它的属性，允许定制

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Scope("session")
public @interface SessionScope {
```

# 类的自动检测及Bean的注册

- Spring可以自动检测类并注册Bean到ApplicationContext中

```
Service
ublic class SimpleMovieLister {

  private MovieFinder movieFinder;

  @Autowired
  public SimpleMovieLister(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }
```

```
@Repository
public class JpaMovieFinder implements MovieFinder {
    // implementation elided for clarity
}
```

**spring可以自动检测类，并注册Bean到ApplicationContext**

# <context:annotation-config/>

- 通过在基于XML的Spring配置如下标签（请注意包含上下文命名空间）
- <context:annotation-config/>仅会查找在同一个applicationContext中的bean注解

```
<?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.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>
```

以前我们用xml配置的时候用的是上面的部分，比如schemLocation 又叫命名空间
用context开头的时候，就可以用下面的

```
    <context:component-scan base-package="org.example"/>
</beans>
```

- <context:component-scan>包含<context:annotation-config>，通常在使用前者后，不用再使用后者
- AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor也会被包含进来

**<context：component-scan>是扫描基于类的注解**
**<context:annotation-config>（这个使用的少）,是扫描bean的方法或成员变量的注解**

# 使用过滤器进行自定义扫描

- 默认情况下，类被自动发现并注册bean的条件是：使用@Component, @Repository, @Service, @Controller注解或者使用@Component的自定义注解
- 可以通过过滤器修改上面的行为，如：下面例子的XML配置忽略所有的@Repository注解并用“Stub”代替

```
<beans>
    <context:component-scan base-package="org.example">
        <context:include-filter type="regex"
                expression=".*Stub.*Repository"/>
        <context:exclude-filter type="annotation"
                expression="org.springframework.stereotype.Repository"/>
    </context:component-scan>
</beans>
```

- 还可使用use-default-filters=“false”禁用自动发现与注册

(1)注解的name属性相当于xml配置bean的id属性；

(2)当注解时未设置name时，则调用BeanName生成器生成name值，默认是使用被注解的类名进行首字母小写得到。

# Using filters to customize scanning

| Filter Type | Example Expression | Description |
|---|---|---|
| annotation | org.example.SomeAnnotation | An annotation to be present at the type level in target components. |
| assignable | org.example.SomeClass | A class (or interface) that the target components are assignable to (extend/implement). |
| aspectj | org.example..*Service+ | An AspectJ type expression to be matched by the target components. |
| regex | org\.example\.Default.* | A regex expression to be matched by the target components class names. |
| custom | org.example.MyTypeFilter | A custom implementation of the org.springframework.core.type |

# 定义Bean

## 扫描过程中组件被自动检测，那么Bean名称是由BeanNameGenerator生成的(@Component, @Repository,@Service,@Controller都会有个name属性用于显式设置Bean Name)

```
@Service("myMovieLister")
public class SimpleMovieLister {
    // ...
}
```

```
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

## 可自定义bean命名策略，实现BeanNameGenerator接口，并一定要包含一个无参数构造函器

```
<beans>
    <context:component-scan base-package="org.example"
        name-generator="org.example.MyNameGenerator" />
```

```
<beans>
    <context:component-scan base-package="org.example"
        name-generator="org.example.MyNameGenerator" />
</beans>
```

如果我们在上面通过@service声明了一个，就会自动扫描，相当于ID,如果没有，像第二个列子，@Repository，**那么就根据BeanNameGenerator生成他的ID，通常的声明规则是以类名为基础，并把类名的第一个基础小写，把这个字符串作为ID，同样 ，我们可以自定义，实现接口，与上面一样，我们怎么使用我们自己定义的命名的策略呢，就是一定要在<context:componen>去指定我们的命名策略的实现**

# 作用域(Scope)

## 通常情况下自动查找的Spring组件，其scope是singleton，Spring2.5提供了一个标识scope的注解@Scope

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

## 也可以自定义scope策略，实现ScopeMetadataResolver接口并提供一个无参构造器

```
<beans>
    <context:component-scan base-package="org.example"
            scope-resolver="org.example.MyScopeResolver" />
</beans>
```

# 代理方式

- 可以使用*scoped-proxy*属性指定代理，有三个值可选：no, interfaces, targetClass

```
<beans>
    <context:component-scan base-package="org.example"
        scoped-proxy="interfaces" />
</beans>
```

**@Required**

**@Required** 注释应用于 bean 属性的 setter 方法，它表明受影响的 bean 属性在配置时必须放在 XML 配置文件中，否则容器就会抛

出一个 BeanInitializationException 异常。下面显示的是一个使用 @Required 注释的示例。

- @Required注解适用于bean属性的setter方法
- 这个注解仅仅表示，受影响的bean属性必须在配置时被填充通过在bean定义或通过自动装配一个明确的属性值

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Required
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...

}
```

**bean的属性必须在配置的时候就被赋值**

这个注解不怎么常用，知道就好

## @Autowired

@Autowired

- 可以将@Autowired注解为“传统”的setter方法

```
private MovieFinder movieFinder;

@Autowired
public void setMovieFinder(MovieFinder movieFinder) {
    this.movieFinder = movieFinder;
}
```

- 可用于构造器或成员变量

```
@Autowired
private MovieCatalog movieCatalog;

private CustomerPreferenceDao customerPreferenceDao;

@Autowired
public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
    this.customerPreferenceDao = customerPreferenceDao;
}
```

相比于前面的来**@Required来说还可以用到构造器上**

默认情况下，如果因找不到合适的bean将会导致autowiring失败抛出异常，可以通过下面的方式避免

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired(required=false)
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...

}
```

每个类只能有一个构造器被标记为required=true

@Autowired的必要属性，建议使用@Required注解

**就是通过声明required=false**

**下面贴代码，关键部分的**

```
@Service
```

```
public class InjectionServiceImpl implements InjectionService {
```

```
@Autowired

public InjectionServiceImpl(InjectionDAO injectionDAO) {

this.injectionDAO = injectionDAO;

}
```

```
@Autowired

private InjectionDAO injectionDAO;
```

**还有上面这种方式**

```
private InjectionDAO injectionDAO;

@Autowired

public void setInjectionDAO(InjectionDAO injectionDAO) {

this.injectionDAO = injectionDAO;

}
```

**还有上面这种方式 有三种方式， 1 是直接在变量上声明，2 是在构造器上声明，3 是在set方法上声明**

```
@Repository

public class InjectionDAOImplimplements InjectionDAO {
```

**就是 相当于我们以前的构造注入与设值注入 不过这是直接放在成员变量上 与set注入是比较类似的**

http://blog.csdn.net/heyutao007/article/details/5981555

@Autowired 注释，它可以对类成员变量、方法及构造函数进行标注，完成自动装配的工作。 通过 @Autowired的使用来消除 set ，get方法。

**基于泛型的自动装配**

```
public interface Store<T> {
}
```

```
@Configuration
public class IntegerStoreimplements Store<Integer> {


}
```

```
@Configuration
public class StringStoreimplements Store<String> {
}
```

```
public interface Testi {
public StringStore stringStore();
public IntegerStore integerStore();
public Store stringStoreTest();
}
```

```
@Configuration
public class Test07  implements  Testi{

@Autowired
      @Qualifier("stringStore")
private Store<String> s1;
```

```
@Autowired
      @Qualifier("integerStore")
private Store<Integer> s2;


@Bean

public StringStore stringStore() {

return new StringStore();

    }


@Bean

public IntegerStore integerStore() {

return new IntegerStore();

    }


@Bean(name = "stringStoreTest")

public Store stringStoreTest() {
System.out.println("s1 : " + s1.getClass().getName());

System.out.println("s2 : " + s2.getClass().getName());

return new StringStore();

    }

public static void main(String[] args) {
ApplicationContext context=new ClassPathXmlApplicationContext("Spring/applicationcontext2.xml");
context.getBean("stringStoreTest");
    }
}
```

## @Configuration 和 @Bean 注解

带有 **@Configuration** 的注解类表示这个类可以使用 Spring IoC 容器作为 bean 定义的来源。**@Bean** 注解告诉 Spring，一个带有 @Bean 的注解方法将返回一个对象，该对象应该被注册为在 Spring 应用程序上下文中的 bean

## @Resource

# @Resource

Spring还支持使用JSR-250@Resource注解的变量或setter方法，这是一种在Java EE 5和6的通用模式，Spring管理的对象也支持这种模式

@ Resource有一个name属性，并且默认Spring解释该值作为被注入bean的名称

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Resource(name="myMovieFinder")
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
```

- 如果没有显式地指定@Resource的name，默认的名称是从属性名或者setter方法得出

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Resource
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

}
```

- 注解提供的名字被解析为一个bean的名称，这是由ApplicationContext的中的CommonAnnotationBeanPostProcessor发现并处理的

## @PostConstruct and @PreDestroy

- CommonAnnotationBeanPostProcessor不仅能识别JSR-250中的生命周期注解@Resource，在Spring2.5中引入支持初始化回调和销毁回调，前提是CommonAnnotationBeanPostProcessor是Spring的ApplicationContext中注册的

```
public class CachingMovieLister {

    @PostConstruct
    public void populateMovieCache() {
        // populates the movie cache upon initialization...
    }

    @PreDestroy
    public void clearMovieCache() {
        // clears the movie cache upon destruction...
    }
```

这个上面的**@PostConstruct与@PreDestory**
**这方法与以前的上面的inti-method与destory-method是相同的，用注解的时候，我们通常是这样的 ，同理，下面的spring2的笔记中，是有我们记录的，关于@Bean注解也是可以实现的 ，**

**其实在日常使用中，@Service，@Resource尤其是@Autoawried使用时相当高的**
像这种@PostConstruct与@PreDestory在特殊情况下也是经常使用的
在自动set的情况下，使用@Resource与@Autowir的使用都是可以的

下面是代码:

```java
@Service
public class JsrServie {

// @Resource
private JsrDAO jsrDAO;
@Resource
public void setJsrDAO(JsrDAO jsrDAO) {
this.jsrDAO = jsrDAO;
}
@PostConstruct
public  void init(){
        System.out.println("PostConstruct inti...");
}
@PreDestroy
public  void destory(){
      System.out.println("PreDestroy destory...");
}

public void save() {
jsrDAO.save();
}

}
```

- 从Spring3.0开始支持JSR330标准注解(依赖注入注解)，其扫描方式与Spring注解一致
- 使用JSR330需要依赖javax.inject包
- 使用Maven引入方式

```xml
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

**这个@Inject，是等效于@Autowired，可以用于类，属性，方法，构造器。**
**@Named**
如果想使用特定名称进行依赖注入，使用@Named
**@Named与@Component是等效的**

- 如果想使用特定名称进行依赖注入，使用@Named
- @Named与@Component是等效的

```
import javax.inject.Inject;
import javax.inject.Named;

public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(@Named("main") MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
```

一种是注解在类上， 如上面这个列子，一种是指定某一个名称的Bean

下面贴代码：

```
@Named

public class JsrServie {

//    @Inject

private JsrDAO jsrDAO;

// @Resource

@Inject

public void setJsrDAO(@Named("jsrDAO") JsrDAO jsrDAO) {

this.jsrDAO = jsrDAO;

}

@PostConstruct

public  void init(){

        System.out.println("PostConstruct inti...");

}

@PreDestroy

public  void destory(){

      System.out.println("PreDestroy destory...");

}


public void save() {

jsrDAO.save();

}


}
```

其中的

```
public void setJsrDAO(@Named("jsrDAO") JsrDAO jsrDAO) {
```

**@Named ( "jsrDAO" )** 为什么要加这个呢，是因为，这个JsrDAO有很多（有可能）重名的。又比如，这是个接口，这个接口有俩个实

现类，这个我们就可以用@Named来引用注明是用哪个，