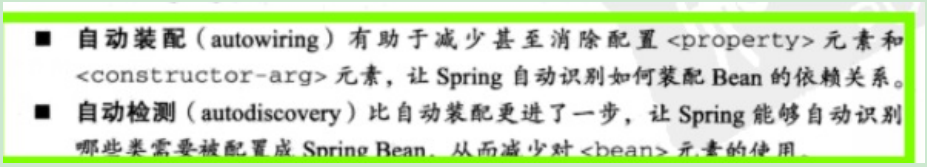
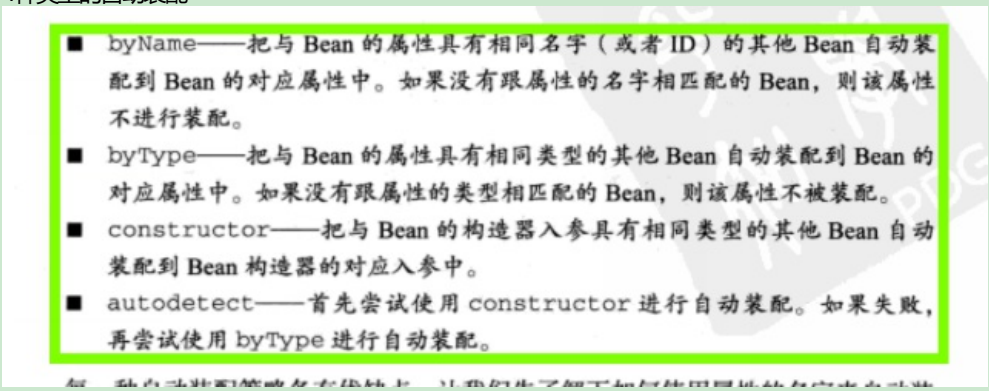
阅读笔记2

第三章
自动装配Bean属性

有俩种技巧 来帮助我们减少XML的配置数量



4种类型的自动装配




```
<bean id="kenny2" class="com.spring.bean.Instrumentalist" autowire="byName"><!--自动装配  autowire-->
<property name="song" > <null/></property>
<!-- <property name="instrument" >
        <bean  class="com.spring.bean.Saxophone"/>
    </property>-->
</bean>

<bean id="kenny2" class="com.spring.bean.Saxophone" ></bean>
```

这是关于Spring 的一个自动装配 用的是byname方式 是通过Bean的名字来进行匹配的
使用byName自动装配的缺点是需要假设Bean的名字与其他Bean的属性名字一样

```
<bean id="instrument" class="com.spring.bean.Saxophone" ></bean>
```

还有另外一种配置方式
就是**bytype**
byType自动装配存在一个局限性 如果Spring寻到多个Bean 它们的类型与需要自动装配的属性都相匹配 在这个情况下 Spring不会去猜测哪个更合适 而是抛出异常 所以 应用只允许存在一个Bean与需要装配的属性类型相匹配

为了解决这个问题 可以为自动装配标识一个首选Bean 或者可以取消某个Bean自动装配的候选资格
这里的是用的**primary的属性** 这个属性是标识首选bean的 如果只有一个自动装配的候选Bean的primary属性设置为true 那么该Bean将比其他候选Bean优先被选择 但是primary属性有个很怪异的一点 它默认设置为true 这意味着所有的候选Bean都将变为首选Bean
所以为了使用primary属性 我们不得不将所有非首选Bean的primary属性设置为false

```
<bean id="saxophone" class="com.spring.bean.Saxophone" primary="false"></bean>
```

primary属性仅对首选Bean有意义 如果在自动装配时 我们希望排除某些Bean 那可以设置这些Bean的autowire-candidate属性为false

这里的autowire-candidate属性是排除bean的

```
<bean id="saxophone" class="com.spring.bean.Saxophone" autowire-candidate="false"></bean>
```

通过构造器注入
这个就是构造器注入的方法 上述声明告诉Spring去审视Juggler的构造器 并尝试在Spring配置中寻找匹配Juggler某一个构造器所有入参的Bean 当找到匹配的时候 Spring使用这个构造器 就将这个Bean作为入参传入

```
<bean id="duke"
```

```
class="com.springinaction.springidol.Juggler" autowire="constructor" />
```

构造器注入与buType自动装配有相同的局限性 当发现多个Bean匹配某个构造器的入参时 Spring不会猜测哪一个Bean更适合自动装配 此外 如果发现一个类有多个构造器 它们都满足自动装配的条件时 Spring也不会尝试猜测哪一个构造器更适合使用

最佳自动装配

```
<bean id="duke"
class="com.springinaction.springidol.Juggler" autowire="autodetect" />
```

这个就是先用构造器注入 然后使用bytype

还有一种的默认自动装配方式 就是在根目录下声明

```
default-autowire="byType"
```

这样的话 所有的xml下的所有bean都默认配置这个 这个属性默认是none
可以用自己bean中定义的autowire属性来覆盖它
注意 default-autowire应用指定Spring配置文件中的所有Bean 但是你可以在一个Spring应用上下文中定义多个配置文件 每一个配置文件都有自己的默认自动装配策略

```
<?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:util="http://www.springframework.org/schema/util"
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd"
default-autowire="byType">
```

我们是可以混用自动装配与显示装配的
这里是有一个是可以装配null值的

```
<property name="instrument" ref="saxophone"><null/></property>
```

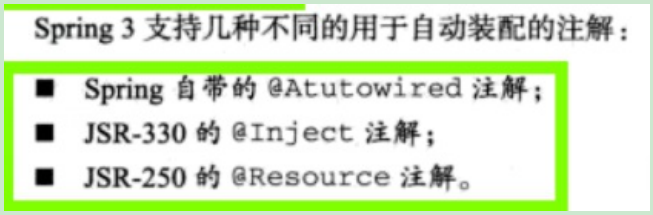
我们还可以进行使用注解进行自动装配
这样子与在xml中进行的装配是并没有太大的区别 但是可以标志在某个类上选择性的标志某个属性来进行自动装配
Spring容器是默认不能是注解装配的 我们必须要在xml中进行打开它

有几种方式 一种是打开相应注解的Bean 但是这种已经不推荐了 还有一种就是使用 打开注解配置

```
<context:annotation-config></context:annotation-config>
```

注意 如果使用扫描注解来注册bean的方式 注解配置是默认打开的

```
<context:component-scan base-package="com.Spring.Aop.Test"/>
```



使用@Atuowired注解来进行自动装配

```
@Autowired
public void setInstrument(Instrument instrument) {
this.instrument = instrument;
}
}
```

还可以自动装配的Bean引用的任意方法

```
@Autowired
public void perform() throws PerformanceException {
```

```
  System.out.print("Playing " + song + " : ");
instrument.play();
}
```

还可以进行自动装配构造器 表示当创建Bean时 即使在XML文件中没有<constructor-arg>元素配置标志属性 这个构造器也需要进行自动装配

```
@Autowired
public Instrumentalist() {
}
```

但是这个@Autowired有时候就容易出问题 就是如果没有bean或者多个bean进行匹配的时候 就出错了
这个时候我们就可以使用一种可选的自动装配

```
@Autowired(required = false)
private Instrument instrument;
```

如果没有查询到intrument属性的bean 就不会出现任何问题 自动将instrument属性设为null
注意在构造器中装配时 只有一个构造器可以将@Autowired的requeied的属性设置为true 其他使用的只能设为false
此外 当使用@Atupwired标注多个构造器时，Spring就会从所有满足条件的构造器中选择入参最多的那个构造器

这个是进行限定歧义性的依赖
指定了这个只能装配这个名为quitar的bean

```
@Autowired
@Qualifier("quitar")
public void setInstrument(Instrument instrument) {
this.instrument = instrument;
```

我们还可以在xml上进行直接使用qualifier来缩小范围 ，当然还直接可以在类上直接使用注解来标识 这俩个是一样的

```
<bean class="com.springinaction.springidol.Piano">
    <qualifier value="stringed"></qualifier>
</bean>
```

```
@Qualifier("stringed")
public class Piano implements Instrument {
```

这是一个自定义的限定器（Qualifier） 当spring尝试装配instrument属性时，spring会把所有的可选择的乐器Bean缩小到只有被@StringedInsTrument属性

```
@Target({ElementType.FIELD ,ElementType.PARAMETER,ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface StringedInstrument {



}
```

如果用这个@StringedInsTrument进行注解的属性有多个 我们需要进一步的限定来缩小范围
在我们需要装配的类上加上这个注解 另外在我们需要装配的地方上也加上 这样就可以达到了缩小范围的目的

我们可以定义另一个限定器注解来进一步限定 例如 克里里琴就加这俩个 而吉他就只有@StringedInsTrument

借助@Inject实现自动装配
这个@Inject注解是可以注入Provider的
Provuder接口可以实现Bean的引用的延迟注入以及注入Bean的多个实例等功能
注意 @Inject没有required属性 因此 @Inject注解所标注的依赖关系必须存在 如果不存在 就会抛出异常

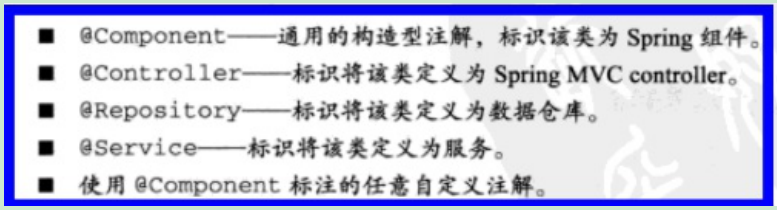
@Name
就是跟@Autowire属性中的@Qualifier是一样的作用 不过是用于@Inject不过 @Name是通过Bean的ID来标识可选择的Bean
@Qualifier注解帮助我们缩小所匹配的Bean的选择范围 而@Named通过Bean的ID来标识可选择的Bean

@Value就是新的一种装配注解
可以让我们使用注解装配String类型的值和基本类型的值 列如int.boolean
@Value直接标注属性 方法 或者方法参数 并传入一个String类型的表达式来装配属性

@Value ("Eruption")
private String song;
@Value最重要的作用是进行装配表达式 来装配硬编码的值并没有太大的意义

```
@Value("#{systemProperties.myFavoriteSong}");
private  String song;
```

自动检测Bean



这是一种过滤组件的扫描
<context:exclude-filter 的type和experssion属性一起协作来定义组件扫描策略
在这个进行自定义组件扫描的过程中 其中注解的方式最多的

```
<context:component-scan base-package="com.springinaction.springidol">
       <context:include-filter type="assignable" expression="com.springinaction.springidol.In
strument"/>
</context:component-scan>
```

这里的我们还可以用到另外一种配置进行排除

```
<context:component-scan base-package="com.springinaction.springidol">
       <context:include-filter type="assignable"   expression="com.springinaction.springidol.
Instrument"/>
       <context:exclude-filter type="assignable" expression="com.springinaction.springidol.Po
em"/>

</context:component-scan>
```

使用Spring基于java的配置

我们可以使用@Configuration来使用注解javabean 这个注解的作用于beans的作用是一样的

关于定义一个配置类

使用@Configuration 和@Bean注解

@Configuration就是相当于xml中的<beans> @Bean相当于<bean>

在这里XML与注解的一个方式的区别

注解的@Bean属性有一个优点 在xml配置的时候 我们的类型与ID都是由String类型来确定的 这个是无法进行编译检查的如果我在xml中进行了修改 而在类中忘记了修改 这就会导致出错 而注解不会出现这种情况
在Spring的基于java的配置中 并没有String属性 Bean的ID和类型都是被视为方法签名的一部分 Bean的实际创建对象是在方法体中定义的 因为它们全都是java代码 所以我们可以进行编译器检查来确保Bean的类型是合法类型 并且Bean的ID是唯一的

用Bean装配另一个Bean引用也是十分简单的

```
public class HelloworldTest{
}
```

```
public class HelloWorld {

private HelloworldTest HelloworldTest;


public    HelloWorld(){

    }
public HelloWorld(HelloworldTest HelloworldTest) {
this.HelloworldTest = HelloworldTest;
    }

}
```

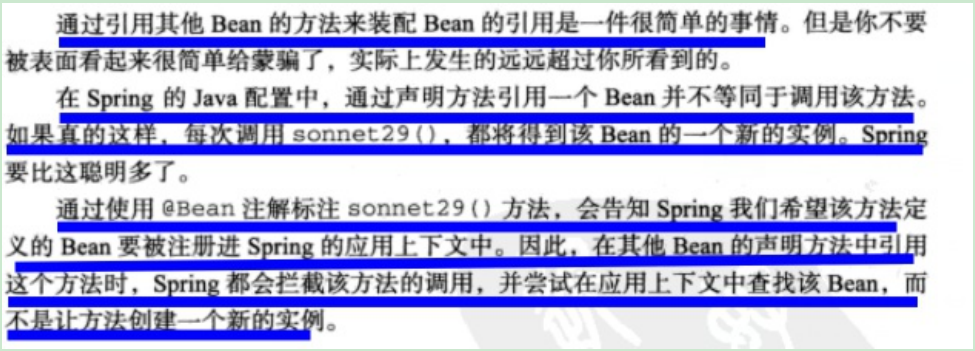
```java
@Configuration
public class HelloWorldConfig {
@Bean(name = "hello")
public static HelloWorld helloWorld(){
return new HelloWorld();
   }


@Bean(name = "helloworldTest")
public  static  HelloworldTest helloworldTest() {
return new HelloworldTest();
 }


@Bean(name = "hello2")
public static HelloWorld helloWorld2(){
return new HelloWorld(helloworldTest());
   }


public static void main(String[] args) {
ApplicationContext context = new ClassPathXmlApplicationContext("Spring/applicationcontext2.xml");
HelloWorld helloWorld = (HelloWorld) context.getBean("hello2");

   }
}
```



通过引用其他 Bean 的方法来装配 Bean 的引用是一件很简单的事情。但是你不要被表面看起来很简单给蒙骗了，实际上发生的远远超过你所看到的。

在 Spring 的 Java 配置中，通过声明方法引用一个 Bean 并不等同于调用该方法。如果真的这样，每次调用 sonnet29()，都将得到该 Bean 的一个新的实例。Spring 要比这聪明多了。

通过使用 @Bean 注解标注 sonnet29() 方法，会告知 Spring 我们希望该方法定义的 Bean 要被注册进 Spring 的应用上下文中。因此，在其他 Bean 的声明方法中引用这个方法时，Spring 都会拦截该方法的调用，并尝试在应用上下文中查找该 Bean，而不是让方法创建一个新的实例。