> 作者：千锋-索尔
>
> 版本：QF1.0
>
> 版权：千锋Java教研院

一、类路径扫描和管理组件类路径扫描和管理组件

注解是否比XML配置更好?

在引入基于注解的配置之后,便出现了注解是否比 XML 配置更好的问题. 答案是,得看情况,每种方法都有其优缺点,通常由开发人员决定使用那种方式. 首先看看两种定义方式,注解在它们的声明中提供了很多上下文信息,使得配置变得更短、更简洁; 但是,XML 擅长于在不接触源码或者无需反编译的情况下装配组件,一些开发人员更喜欢在源码上使用注解配置. 而另一些人认为注解类不再是 POJO,同时认为注解配置会很分散,最终难以控制.

无论选择如何,Spring 都可以兼顾两种风格,甚至可以将它们混合在一起. Spring 通过其 JavaConfig选项,允许注解以无侵入的方式使用,即无需接触目标组件源代码.

- Spring2.5支持XML+注解的方式
- Spring3.0支持JavaConfig代替XML配置定义外部Bean
- Spring4.0之后Springboot完全采用JavaConfig开发方式

本章中的大多数示例会使用 XML 配置指定在 Spring 容器中生成每个 `BeanDefinition` 的元数据,上一节(基于注解的容器配置)演示了如何通过源代码注解提供大量的元数据配置. 然而,即使在这些示例中,注解也仅仅用于驱动依赖注入. "base" bean依然会显式地在XML文件中定义. 本节介绍通过扫描类路径隐式检测候选组件的选项. 候选者组件是 class 类, 这些类经过过滤匹配,由 Spring 容器注册的 bean 定义会成为 Spring bean. 这消除了使用 XML 执行 bean 注册的需要(也就是没有 XML 什么事儿了),可以使用注解(例如 `@Component`), AspectJ 类型表达式或开发者自定义过滤条件来选择哪些类将在容器中注册 bean 定义.

```
从 Spring 3.0 开始,Spring JavaConfig 项目提供的许多功能都是核心 Spring 框架的一部分. 这允许开发者使用 Java 而不是使用传统的 XML 文件来定义 bean. 有关如何使用这些新功能的示例,请查看 `@Configuration`, `@Bean`,`@Import`, 和 `@DependsOn` 注解.
```

1. `@Component` 注解和更多模板注解

`@Repository` 注解用于满足存储库(也称为数据访问对象或DAO)的情况,这个注解的用途是自动转换异常.

Spring 提供了更多的构造型注解: `@Component`, `@Service`, 和 `@Controller`. `@Component` 可用于管理任何 Spring 的组件. `@Repository`, `@Service`, 或 `@Controller` 是 `@Component` 的特殊化. 用于更具体的用例(分别在持久性,服务和表示层中) . 因此,可以使用 `@Component` 注解组件类,但是,通过使用 `@Repository`, `@Service`, 和 `@Controller` 注解它们,能够让你的类更易于被合适的工具处理或与相应的切面关联. 例如,这些注解可以使目标组件变成切入点. 在 Spring 框架的未来版本中, `@Repository`, `@Service`, 和 `@Controller` 也可能带有附加的语义. 因此,如果在使用 `@Component` 或 `@Service` 来选择服务层时, `@Service` 显然是更好的选择. 同理,在持久化层要选择 `@Repository`,它能自动转换异常.

- @Controller: 注册bean并标识为控制层组件
- @Service：注册bean并标识为业务层组件
- @Repository：注册bean并标识为持久化层组件
- @Component：注册bean，但bean不属于上述三层

2. 使用元注解和组合注解

Spring 提供的许多注解都可以在自己的代码中用作元注解. 元注解是可以应用于另一个注解的注解. 例如, 前面提到的 `@Service` 注解是使用 `@Component` 进行元注解的,如下例所示:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Service {

    // ...
}
```

`@Component` 使 `@Service` 以与 `@Component` 相同的方式处理.

元注解也可以进行组合,进而创建组合注解. 例如,来自 Spring MVC 的 `@RestController` 注解是由 `@Controller` 和 `@ResponseBody` 组成的

此外,组合注解也可以重新定义来自元注解的属性. 这在只想暴露元注解的部分属性时非常有用. 例如,Spring 的 `@SessionScope` 注解将它的作用域硬编码为 `session` ,但仍允许自定义 `proxyMode` . 以下清单显示了 `SessionScope` 注解的定义:

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Scope(WebApplicationContext.SCOPE_SESSION)
public @interface SessionScope {

    /**
     * Alias for {@link Scope#proxyMode}.
     * <p>Defaults to {@link ScopedProxyMode#TARGET_CLASS}.
     */
    @AliasFor(annotation = Scope.class)
    ScopedProxyMode proxyMode() default ScopedProxyMode.TARGET_CLASS;

}
```

然后,可以使用 `@SessionScope` 而不声明 `proxyMode` ,如下所示:

```
@Service
@SessionScope
public class SessionScopedService {
    // ...
}
```

还可以覆盖 `proxyMode` 的值,如以下示例所示:

```
@Service
@SessionScope(proxyMode = ScopedProxyMode.INTERFACES)
public class SessionScopedUserService implements UserService {
    // ...
}
```

3. 自动探测类并注册 bean 定义

Spring 可以自动检测各代码层中被注解的类,并使用 `ApplicationContext` 内注册相应的 `BeanDefinition`. 例如,以下两个类就可以被自动探测:

```
@Service
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

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

想要自动检测这些类并注册相应的 bean,需要在 `@Configuration` 配置中添加 `@ComponentScan` 注解,其中 `basePackages` 属性是两个类的父包路径. (或者,可以指定以逗号或分号或空格分隔的列表,其中包含每个类的父包) .

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
    // ...
}
```

为简洁起见,前面的示例可能使用了注解的 `value` 属性(即 `@ComponentScan("org.example")`).或者使用 XML 配置代替扫描:

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

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

</beans>
```

使用 `<context:component-scan>` 隐式启用 `<context:annotation-config>` 的功能. 使用 `<context:component-scan>` 时,通常不需要包含 `<context:annotation-config>` 元素.类路径扫描的包必须保证这些包出现在类路径中.

在使用 `component-scan` 元素时, `AutowiredAnnotationBeanPostProcessor` 和 `CommonAnnotationBeanPostProcessor` 都会隐式包含. 意味着这两个组件也是自动探测和注入的. 所有这些都无需 XML 配置.

4. 在自定义扫描中使用过滤器

默认情况下,使用 `@Component`, `@Repository`, `@Service`, `@Controller` `@Configuration` 注解的类或者注解为 `@Component` 的自定义注解类才能被检测为候选组件. 但是,开发者可以通过应用自定义过滤器来修改和扩展此行为. 将它们添加为 `@ComponentScan` 注解的 `includeFilters` 或 `excludeFilters` 参数(或作为 `component-scan` 元素. 元素的 `include-filter` 或 `exclude-filter` 子元素. 每个 filter 元素都需要包含 `type` 和 `expression` 属性. 下表介绍了筛选选项:

过滤类型	表达式例子	描述
annotation (default)	`org.example.SomeAnnotation`	要在目标组件中的类级别出现的注解.
assignable	`org.example.SomeClass`	目标组件可分配给(继承或实现) 的类(或接口) .
aspectj	`org.example..*Service+`	要由目标组件匹配的 AspectJ 类型表达式.
regex	`org\.example\.Default.*`	要由目标组件类名匹配的正则表达式.
custom	`org.example.MyTypeFilter`	`org.springframework.core.type.TypeFilter` 接口的自定义实现.

以下示例显示忽略所有 `@Repository` 注解并使用 “stub” 存储库的配置:

```
@Configuration
@ComponentScan(basePackages = "org.example",
        includeFilters = @Filter(type = FilterType.REGEX, pattern =
".*Stub.*Repository"),
        excludeFilters = @Filter(Repository.class))
public class AppConfig {
    // ...
}
```

以下清单显示了等效的 XML:

```
<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>
```

还可以通过在注解上设置 `useDefaultFilters=false` 或通过 `use-default-filters="false"` 作为 `<<component-scan/>` 元素的属性来禁用默认过滤器. 这样将不会自动检测带有 `@Component`, `@Repository`, `@Service`, `@Controller`, 或 `@Configuration`.

5. 在组件中定义bean的元数据

Spring 组件也可以向容器提供 bean 定义元数据. 在 `@Configuration` 注解的类中使用 `@Bean` 注解定义 bean 元数据(也就是 Spring bean),以下示例显示了如何执行此操作:

```
@Component
public class FactoryMethodComponent {

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }

    public void doWork() {
        // Component method implementation omitted
    }
}
```

这个类是一个 Spring 组件,它有个 `doWork()` 方法. 然而,它还有一个工厂方法 `publicInstance()` 用于产生 bean 定义. `@Bean` 注解了工厂方法, 还设置了其他 bean 定义的属性,例如通过 `@Qualifier` 注解的 `qualifier` 值. 可以指定的其他方法级别的注解是 `@Scope`, `@Lazy` 以及自定义的 qualifier 注解.

除了用于组件初始化的角色之外,`@Lazy` 注解也可以在 `@Autowired` 或者 `@Inject` 注解上,在这种情况下,该注入将会变成延迟注入代理 `lazy-resolution` proxy(也就是懒加载) 对于复杂的惰性交互，尤其是组合对于可选依赖项，我们推荐使用 `ObjectProvider<MyTargetBean>`。

自动注入的字段和方法也可以像前面讨论的一样被支持,也支持 `@Bean` 方法的自动注入. 以下示例显示了如何执行此操作:

```
@Component
public class FactoryMethodComponent {

    private static int i;

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }

    // use of a custom qualifier and autowiring of method parameters
    @Bean
    protected TestBean protectedInstance(
            @Qualifier("public") TestBean spouse,
            @Value("#{privateInstance.age}") String country) {
        TestBean tb = new TestBean("protectedInstance", 1);
        tb.setSpouse(spouse);
        tb.setCountry(country);
        return tb;
    }

    @Bean
    private TestBean privateInstance() {
        return new TestBean("privateInstance", i++);
    }

    @Bean
    @RequestScope
    public TestBean requestScopedInstance() {
        return new TestBean("requestScopedInstance", 3);
    }
}
```

该示例将方法参数为 `String`,名称为 `country` 的 bean 自动装配为另一个名为 `privateInstance` 的 bean 的 age 属性值. Spring 表达式语言元素通过记号 `#{ <expression> }` 来定义属性的值. 对于 `@Value` 注解,表达式解析器在解析表达式后,会查找 bean 的名字并设置 value 值.

从 Spring4.3 开始,还可以声明一个类型为 `InjectionPoint` 的工厂方法参数(或其更具体的子类: `DependencyDescriptor`) 以访问触发创建当前 bean 的请求注入点. 请注意,这仅适用于真实创建的 bean 实例,而不适用于注入现有实例. 因此,这个特性对 prototype scope 的 bean 最有意义. 对于其他作用域,工厂方法将只能看到触发在给定 scope 中创建新 bean 实例的注入点. 例如,触发创建一个延迟单例 bean 的依赖. 在这种情况下,使用

提供的注入点元数据拥有优雅的语义. 以下示例显示了如何使用 `InjectionPoint`:

```
@Component
public class FactoryMethodComponent {

    @Bean @Scope("prototype")
    public TestBean prototypeInstance(InjectionPoint injectionPoint) {
        return new TestBean("prototypeInstance for " + injectionPoint.getMember());
    }
}
```

6. 命名自动注册组件

扫描处理过程,其中一步就是自动探测组件,扫描器使用 `BeanNameGenerator` 对探测到的组件命名. 默认情况下,各代码层注解(`@Component`, `@Repository`, `@Service`, 和 `@Controller`)所包含的 name 值,将会作为相应的 bean 定义的名字.

如果这些注解没有 name 值,或者是其他一些被探测到的组件(比如使用自定义过滤器探测到的),默认会又 bean name 生成器生成,使用小写类名作为 bean 名字. 例如,如果检测到以下组件类,则名称为 `myMovieLister` 和 `movieFinderImpl`:

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

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

如果不想依赖默认的 bean 命名策略,则可以提供自定义 bean 命名策略. 首先,实现 `BeanNameGenerator` 接口,并确保包括一个默认的无参构造函数. 然后,在配置扫描程序时提供完全限定的类名,如以下示例注解和 bean 定义所示:

```
@Configuration
@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)
public class AppConfig {
    // ...
}
<beans>
    <context:component-scan base-package="org.example"
        name-generator="org.example.MyNameGenerator" />
</beans>
```

作为一般规则,考虑在其他组件可能对其进行显式引用时使用注解指定名称. 另一方面,只要容器负责装配时,自动生成的名称就足够了.

7. 为自动检测的组件提供作用域

与 Spring 管理组件一样,自动检测组件的默认和最常见的作用域是 `singleton`. 但是,有时需要一个可由 `@Scope` 注解指定的不同作用域. 可以在注解中提供作用域的名称,如以下示例所示:

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

想要提供自定义作用域的解析策略,而不是依赖于基于注解的方法,那么需要实现 `ScopeMetadataResolver` 接口,并确保包含一个默认的无参数构造函数. 然后,在配置扫描程序时提供完全限定类名. 以下注解和 bean 定义示例显示:

```
@Configuration
@ComponentScan(basePackages = "org.example", scopeResolver = MyScopeResolver.class)
public class AppConfig {
    // ...
}
<beans>
    <context:component-scan base-package="org.example" scope-
resolver="org.example.MyScopeResolver"/>
</beans>
```

当使用某个非单例作用域时,为作用 domain 对象生成代理可能非常必要,原因参看 依赖有 Scope 的 Bean作为依赖的作用域 bean. 为此,组件扫描元素上提供了 scoped-proxy 属性. 三个可能的值是: `no`, `interfaces`, 和 `targetClass`. 例如,以下配置导致标准 JDK 动态代理:

```
@Configuration
@ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode.INTERFACES)
public class AppConfig {
    // ...
}
<beans>
    <context:component-scan base-package="org.example" scoped-proxy="interfaces"/>
</beans>
```

二、基于注解的容器配置

1. @Required

`@Required` 注解适用于 bean 属性 setter 方法,如下例所示:

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}
```

被注解的 bean 属性必须在配置时通过 bean 定义中的显式赋值或自动注入值. 如果受影响的 bean 属性尚未指定值,容器将抛出异常; 这导致及时的、明确的失败,避免在运行后再抛出 `NullPointerException` 或类似的异常. 在这里,建议开发者将断言放入 bean 类本身,例如放入 init 方法. 这样做强制执行那些必需的引用和值,即使是在容器外使用这个类.

2. `@Autowired`

开发者可以在构造器上使用 `@Autowired` 注解:

```
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

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

    // ...
}
```

从 Spring Framework 4.3 开始,如果目标 bean 仅定义一个构造函数,则不再需要 `@Autowired` 构造函数. 如果有多个构造函数可用并且没有 primary/default 构造器,则至少有一个必须注解 `@Autowired` 以让容器知道它使用的是哪个。

还可以将 `@Autowired` 注解应用于 *传统* setter 方法,如以下示例所示:

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}
```

还可以将注解应用于具有任意名称和多个参数的方法,如以下示例所示:

```
public class MovieRecommender {

    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public void prepare(MovieCatalog movieCatalog,
            CustomerPreferenceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

还可以将 `@Autowired` 应用于字段,甚至可以和构造函数混合使用:

```
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    private MovieCatalog movieCatalog;

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

    // ...
}
```

确保的组件(例如,`MovieCatalog` 或 `CustomerPreferenceDao`) 始终按照用于 `@Autowired` 注入的类型声明. 否则,由于在运行时未找到类型匹配,注入可能会失败.对于通过类路径扫描找到的 XML 定义的 bean 或组件类,容器通常预先知道具体类型. 但是,对于 `@Bean` 工厂方法,需要确保其声明的具体返回类型. 对于实现多个接口的组件或可能由其实现类型引用的组件,请考虑在工厂方法上声明最具体的返回类型(至少与引用 bean 的注入点所需的特定类型一致) 。

还可以通过将 `@Autowired` 注解添加到需要该类型数组的字段或方法来指示 Spring 从 `ApplicationContext` 提供特定类型的所有 bean，如以下示例所示：

```
public class MovieRecommender {

    @Autowired
    private MovieCatalog[] movieCatalogs;

    // ...
}
```

也可以应用于集合类型,如以下示例所示:

```
public class MovieRecommender {

    private Set<MovieCatalog> movieCatalogs;

    @Autowired
    public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }

    // ...
}
```

如果想让数组元素或集合元素按特定顺序排列, 可以实现 `org.springframework.core.Ordered` , 或者使用 `@Order` 或标准的 `@Priority` 注解,否则,它们的顺序遵循容器中相应目标 bean 定义的注册顺序.可以在类级别和 `@Bean` 方法上声明 `@Order` 注解,可能是通过单个 bean 定义(在多个定义使用相同 bean 类的情况下) . `@Order` 值可能会影响注入点的优先级,但要注意它们不会影响单例启动顺序,这是由依赖和 `@DependsOn` 声明确定的.请注意,标准的 `javax.annotation.Priority` 注解在 `@Bean` 级别不可用,因为它无法在方法上声明. 它的语义可以通过 `@Order` 值与 `@Primary` 定义每个类型的单个 bean 上.

只要键类型是 String, `Map` 类型就可以自动注入. Map 值将包含所有类型的 bean,并且键将包含相应的 bean 名称. 如以下示例所示:

```
public class MovieRecommender {

    private Map<String, MovieCatalog> movieCatalogs;

    @Autowired
    public void setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }

    // ...
}
```

默认情况下,当没有候选的 bean 可用时,自动注入将会失败; 对于声明的数组,集合或映射,至少应有一个匹配元素.

默认的处理方式是将带有注解的方法、构造函数和字段标明为必须依赖,也可以使用 `@Autowired` 中的 `required=false` 属性. 来标明这种依赖不是必须的,如下:

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

    // ...
}
```

如果不需要的方法(或在多个参数的情况下,其中一个依赖) 不可用,则根本不会调用该方法. 在这种情况下,完全不需要填充非必需字段,而将其默认值保留在适当的位置.

构造函数和工厂方法参数注入是一种特殊情况,因为由于 Spring 的构造函数解析算法可能会处理多个构造函数,因此 `@Autowired` 中的 `required` 属性的含义有所不同. 默认情况下,需要构造函数和工厂方法参数,但是在单构造函数场景中有一些特殊规则, 例如,如果没有可用的匹配 bean,则多元素注入点(数组,集合,映射) 解析为空实例. 这允许一种通用的实现模式,其中所有依赖都可以在唯一的多参数构造函数中声明-例如,声明为没有 `@Autowired` 注解的单个公共构造函数.

每个类仅可以有一个带 `@Autowired` 注解,并将 `required` 属性设置为 `true` 的构造函数,但是可以注解多个构造函数. 在这种情况下,它们都必须声明 `required = false` 才能被 被视为自动装配的候选对象(类似于XML中的 `autowire = constructor`).而 Spring 使用的是最贪婪的构造函数.这个构造函数的依赖可以得到满足,那就是具有最多参数的构造函数.同样,如果一个类 声明了多个构造函数,但是没有一个用 `@Autowired` 注解,然后将使用 primary/default 构造函数(如果存在).如果一个类仅声明一个构造函数,即使没有注解,也将始终使用它.请注意带注解的构造函数不必是 public 的.推荐使用 `@Required` 注解来代替 `@Autowired` 的 `required` 属性,`required` 属性表示该属性不是自动装配必需的,如果该属性不能被自动装配. 则该属性会被忽略. 另一方面, `@Required` 会强制执行通过容器支持的任何方式来设置属性. 如果没有值被注入的话,会引发相应的异常.

或者,可以通过 Java 8 的 `java.util.Optional` 表达特定依赖的非必需特性,如以下示例所示:

```
public class SimpleMovieLister {

    @Autowired
    public void setMovieFinder(Optional<MovieFinder> movieFinder) {
        ...
    }
}
```

从 Spring Framework 5.0 开始,还可以使用 `@Nullable` 注解(任何包中的任何类型,例如,来自 JSR-305 的 `javax.annotation.Nullable`)

```
public class SimpleMovieLister {

    @Autowired
    public void setMovieFinder(@Nullable MovieFinder movieFinder) {
        ...
    }
}
```

也可以使用 `@Autowired` 作为常见的可解析依赖的接口,`BeanFactory`, `ApplicationContext`, `Environment`, `ResourceLoader`, `ApplicationEventPublisher`, 和 `MessageSource` 这些接口及其扩展接口 (例如 `ConfigurableApplicationContext` 或 `ResourcePatternResolver`) 会自动解析,无需特殊设置. 以下示例自动装配 `ApplicationContext` 对象:

```
public class MovieRecommender {

    @Autowired
    private ApplicationContext context;

    public MovieRecommender() {
    }

    // ...
}
```

`@Autowired`, `@Inject`, `@Value`, 和 `@Resource` 注解 由 Spring BeanPostProcessor 实现. 也就是说开发者不能使用自定义的 `BeanPostProcessor` 或者自定义 `BeanFactoryPostProcessor` 来使用这些注解 必须使用 XML 或 Spring `@Bean` 方法显式地"连接"这些类型.

总结:

- @Autowired默认根据类型匹配
- 如果匹配到多个接口的实现类，则需要根据名字匹配
    - 修改成员变量的名字
    - 修改bean的名字：@Service("bean的名字")
    - 使用@Qualifier注解
    - 使用@Primary注解设置主bean

    - 使用范型
- 如果类没有名字则会报错

3. `@Primary`

由于按类型的自动注入可能匹配到多个候选者,所以通常需要对选择过程添加更多的约束. 使用 Spring 的 `@Primary` 注解是实现这个约束的一种方法. 它表示如果存在多个候选者且另一个 bean 只需要一个特定类型的 bean 依赖时,就明确使用标记有 `@Primary` 注解的那个依赖. 如果候选中只有一个"Primary" bean,那么它就是自动注入的值

请考虑以下配置,将 `firstMovieCatalog` 定义为主要 `MovieCatalog` :

```
@Configuration
public class MovieConfiguration {

    @Bean
    @Primary
    public MovieCatalog firstMovieCatalog() { ... }

    @Bean
    public MovieCatalog secondMovieCatalog() { ... }

    // ...
}
```

使用上述配置,以下 `MovieRecommender` 将与 `firstMovieCatalog` 一起自动装配:

```
public class MovieRecommender {

    @Autowired
    private MovieCatalog movieCatalog;

    // ...
}
```

相应的bean定义如下:

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

    <context:annotation-config/>
```

```
    <bean class="example.SimpleMovieCatalog" primary="true">
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

4. @Qualifier

`@Primary` 是一种用于解决自动装配多个值的注入的有效的方法,当需要对选择过程做更多的约束时,可以使用 Spring 的 `@Qualifier` 注解,可以为指定的参数绑定限定的值. 缩小类型匹配集,以便为每个参数选择特定的 bean. 在最简单的情况下,这可以是一个简单的描述性值,如以下示例所示:

```
public class MovieRecommender {

    @Autowired
    @Qualifier("main")
    private MovieCatalog movieCatalog;

    // ...
}
```

带有限定符 `main` 的 bean 会被装配到拥有相同值的构造方法参数上.

还可以在各个构造函数参数或方法参数上指定 `@Qualifier` 注解,如以下示例所示:

```
public class MovieRecommender {

    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public void prepare(@Qualifier("main") MovieCatalog movieCatalog,
            CustomerPreferenceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

以下示例显示了相应的 bean 定义.

```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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="main"/>

        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="action"/>

        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

bean 的 name 会作为备用的 qualifier 值,因此可以定义 bean 的 `id` 为 `main` 替代内嵌的 qualifier 元素.这种匹配方式同样有效. 但是,虽然可以使用这个约定来按名称引用特定的 bean, 但是 `@Autowired` 默认是由带限定符的类型驱动注入的. 这就意味着 qualifier 值,甚至是 bean 的 name 作为备选项,只是为了缩小类型匹配的范围. 它们在语义上不表示对唯一 bean `id` 的引用. 良好的限定符值是像 `main` 或 `EMEA` 或 `persistent` 这样的,能表示与 bean `id` 无关的特定组件的特征,在匿名 bean 定义的情况下可以自动生成.

Qualifiers 也可以用于集合类型,如上所述,例如 `Set<MovieCatalog>`. 在这种情况下,根据声明的限定符,所有匹配的 bean 都作为集合注入. 这意味着限定符不必是唯一的. 相反,它们构成过滤标准. 例如,可以使用相同的限定符值 “action” 定义多个 `MovieCatalog` bean,所有这些 bean 都注入到使用 `@Qualifier("action")` 注解的 `Set<MovieCatalog>` 中.

在类型匹配候选项中,根据目标 bean 名称选择限定符值,在注入点不需要 `@Qualifier` 注解. 如果没有其他解析指示符(例如限定符或主标记) , 则对于非唯一依赖性情况,Spring 会将注入点名称(即字段名称或参数名称) 与目标 bean 名称进行匹配,然后选择同名的候选者,如果有的话.

如果打算 `by name` 来驱动注解注入,那么就不要使用 `@Autowired`(多数情况) ,即使在技术上能够通过 `@Qualifier` 值引用 bean 名字. 相反,应该使用 JSR-250 `@Resource` 注解,该注解在语义上定义为通过其唯一名称标识特定目标组件, 其中声明的类型与匹配进程无关. `@Autowired` 具有多种不同的语义,在 `by type` 选择候选 bean 之后,指定的 String 限定的值只会考虑这些被选择的候选者. 例如将 `account` 限定符与标有相同限定符标签的 bean 相匹配.

对于自身定义为 `collection`, `Map`, 或者 `array` 类型的 bean, `@Resource` 是一个很好的解决方案,通过唯一名称引用特定的集合或数组 bean. 也就是说,从 Spring4.3 开始, 只要元素类型信息保存在 `@Bean` 返回类型签名或集合(或其子类) 中,就可以通过 Spring 的 `@Autowired` 类型匹配算法匹配 Map 和数组类型. 在这种情况下,可以使用限定的值来选择相同类型的集合,如上一段所述.

从 Spring4.3 开始,`@Autowired` 也考虑了注入的自引用,即引用当前注入的 bean. 自引用只是一种后备选项,还是优先使用正常的依赖注入操作其它 bean. 在这个意义上,自引用不参与到正常的候选者选择中,并且总是次要的, 相反,它们总是拥有最低的优先级. 在实践中,自引用通常被用作最后的手段. 例如,通过 bean 的事务代理在同一实例上调用其他方法 在这种情况下,考虑将受影响的方法分解为单独委托的 bean,或者使用 `@Resource`,它可以通过其唯一名称获取代理返回到当前的 bean 上.

尝试将 `@Bean` 方法的结果注入相同的配置类也实际上是一种自引用方案. 要么在实际需要的方法签名中惰性解析此类引用(与配置类中的自动装配字段相对) , 要么将受影响的 `@Bean` 方法声明为静态,将其与包含的配置类实例及其生命周期脱钩. 否则,仅在回退阶段考虑此类 Bean,而将其他配置类上的匹配 Bean 选作主要候选对象(如果可用) .

`@Autowired` 可以应用在字段、构造函数和多参数方法上,允许在参数上使用 qualifier 限定符注解缩小取值范围. 相比之下,`@Resource` 仅支持具有单个参数的字段和 bean 属性 setter 方法. 因此,如果注入目标是构造函数或多参数方法,请使用 qualifiers 限定符.

开发者也可以创建自定义的限定符注解,只需定义一个注解,在其上提供了 `@Qualifier` 注解即可. 如以下示例所示:

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

    String value();
}
```

然后,可以在自动装配的字段和参数上提供自定义限定符,如以下示例所示:

```
public class MovieRecommender {

    @Autowired
    @Genre("Action")
    private MovieCatalog actionCatalog;

    private MovieCatalog comedyCatalog;

    @Autowired
    public void setComedyCatalog(@Genre("Comedy") MovieCatalog comedyCatalog) {
        this.comedyCatalog = comedyCatalog;
    }

    // ...
}
```

接下来,提供候选 bean 定义的信息. 开发者可以添加 `<qualifier/>` 标签作为 `<bean/>` 标签的子元素,然后指定 `type` 类型和 `value` 值来匹配自定义的 qualifier 注解. `type` 是自定义注解的权限定类名(包路径+类名) . 如果没有重名的注解,那么可以使用类名(不含包路径) . 以下示例演示了两种方法:

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

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier type="Genre" value="Action"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier type="example.Genre" value="Comedy"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

在类路径扫描和组件管理,将展示一个基于注解的替代方法,可以在 XML 中提供 qualifier 元数据。

在某些情况下,使用没有值的注解可能就足够了. 当注解用于更通用的目的并且可以应用在多种不同类型的依赖上时,这是很有用的. 例如,可以提供可在没有 Internet 连接时搜索的 Offline 目录. 首先,定义简单注解,如以下示例所示:

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

}
```

然后将注解添加到需要自动注入的字段或属性中:

```
public class MovieRecommender {

    @Autowired
    @Offline
    private MovieCatalog offlineCatalog;

    // ...
}
```

现在 bean 定义只需要一个限定符类型,如下例所示:

```
<bean class="example.SimpleMovieCatalog">
    <qualifier type="Offline"/>
    <!-- inject any dependencies required by this bean -->
</bean>
```

开发者还可以为自定义限定名 qualifier 注解增加属性,用于替代简单的 `value` 属性. 如果在要自动注入的字段或参数上指定了多个属性值,则 bean 的定义必须全部匹配这些属性值才能被视为自动注入候选者. 例如,请考虑以下注解定义:

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

    String genre();

    Format format();
}
```

在这种情况下, `Format` 是一个枚举类型,定义如下:

```
public enum Format {
    VHS, DVD, BLURAY
}
```

要自动装配的字段使用自定义限定符进行注解,并包含两个属性的值: `genre` 和 `format`,如以下示例所示:

```
public class MovieRecommender {

    @Autowired
    @MovieQualifier(format=Format.VHS, genre="Action")
    private MovieCatalog actionVhsCatalog;

    @Autowired
    @MovieQualifier(format=Format.VHS, genre="Comedy")
```

```
    private MovieCatalog comedyVhsCatalog;

    @Autowired
    @MovieQualifier(format=Format.DVD, genre="Action")
    private MovieCatalog actionDvdCatalog;

    @Autowired
    @MovieQualifier(format=Format.BLURAY, genre="Comedy")
    private MovieCatalog comedyBluRayCatalog;

    // ...
}
```

最后,bean 定义应包含匹配的限定符值. 此示例还演示了可以使用 bean meta 属性而不是使用 `<qualifier/>` 子元素. 如果可行, `<qualifier/>` 元素及其属性优先, 但如果不存在此类限定符,那么自动注入机制会使用 `<meta/>` 标签中提供的值,如以下示例中的最后两个 bean 定义:

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

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier type="MovieQualifier">
            <attribute key="format" value="VHS"/>
            <attribute key="genre" value="Action"/>
        </qualifier>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier type="MovieQualifier">
            <attribute key="format" value="VHS"/>
            <attribute key="genre" value="Comedy"/>
        </qualifier>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <meta key="format" value="DVD"/>
        <meta key="genre" value="Action"/>
        <!-- inject any dependencies required by this bean -->
```

```
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <meta key="format" value="BLURAY"/>
        <meta key="genre" value="Comedy"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

</beans>
```

5. 使用泛型作为自动装配限定符

除了 `@Qualifier` 注解之外,还可以使用 Java 泛型类型作为隐式的限定形式. 例如,假设具有以下配置:

```
@Configuration
public class MyConfiguration {

    @Bean
    public StringStore stringStore() {
        return new StringStore();
    }

    @Bean
    public IntegerStore integerStore() {
        return new IntegerStore();
    }
}
```

假设上面的 bean 都实现了泛型接口,即 `Store<String>` 和 `Store<Integer>` ,那么可以用 `@Autowire` 来注解 `Store` 接口, 并将泛型用作限定符,如下例所示:

```
@Autowired
private Store<String> s1; // <String> qualifier, injects the stringStore bean

@Autowired
private Store<Integer> s2; // <Integer> qualifier, injects the integerStore bean
```

通用限定符也适用于自动装配列表,`Map` 实例和数组. 以下示例自动装配通用 `List` :

```
// Inject all Store beans as long as they have an <Integer> generic
// Store<String> beans will not appear in this list
@Autowired
private List<Store<Integer>> s;
```

6. `@Resource`

Spring 还通过在字段或 bean 属性 setter 方法上使用 JSR-250 `@Resource`(`javax.annotation.Resource`) 注解来支持注入. 这是 Java EE 中的常见模式(例如,JSF-managed beans 和 JAX-WS 端点中) . Spring 也为 Spring 管理对象提供这种模式.

`@Resource` 接受一个 `name` 属性.. 默认情况下,Spring 将该值解释为要注入的 bean 名称. 换句话说,它遵循按名称的语义,如以下示例所示:

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

如果未明确指定名称,则默认名称是从字段名称或 `setter` 方法生成的. 如果是字段,则采用字段名称. 在 `setter` 方法的情况下,它采用 bean 属性名称. 下面的例子将把名为 `movieFinde` r的 bean 注入其 setter 方法:

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

`ApplicationContext` 若使用了 `CommonAnnotationBeanPostProcessor` ,注解提供的name名字将被解析为 bean 的 name 名字. 如果配置了 Spring 的 `SimpleJndiBeanFactory`, 这些 name 名称就可以通过 JNDI 解析. 但是,推荐使用默认的配置,简单地使用 Spring 的 JNDI,这样可以保持逻辑引用. 而不是直接引用.

`@Resource` 在没有明确指定 name 时,其行为类似于 `@Autowired`,对于特定bean(Spring API内的 bean) , `@Resource` 找到主要类型匹配而不是特定的命名 bean, 并解析众所周知的可解析依赖: `ApplicationContext`, `ResourceLoader`, `ApplicationEventPublisher`, 和 `MessageSource` 接口.

因此,在以下示例中,`customerPreferenceDao` 字段首先查找名为 `customerPreferenceDao` 的 bean,如果未找到,则会使用类型匹配 `CustomerPreferenceDao` 类的实例:

```
public class MovieRecommender {

    @Resource
    private CustomerPreferenceDao customerPreferenceDao;

    @Resource
    private ApplicationContext context;

    public MovieRecommender() {
    }

    // ...
}
```

8. `@Value`

`@Value` 通常用于注入外部属性:

```
@Component
public class MovieRecommender {

    private final String catalog;

    public MovieRecommender(@Value("${catalog.name}") String catalog) {
        this.catalog = catalog;
    }
}
```

使用以下配置:

```
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig { }
```

以及以下 `application.properties` 文件:

```
catalog.name=MovieCatalog
```

在这种情况下,`catalog` 参数和字段将等于 `MovieCatalog` 值.

Spring 提供了一个默认的内嵌值解析器. 它将尝试解析属性值,如果无法解析,则将属性名称(例如 `${catalog.name}`) 作为值注入. 如果要严格控制不存在的值,则应声明一个 `PropertySourcesPlaceholderConfigurer` bean,如以下示例所示:

```
@Configuration
public class AppConfig {

    @Bean
    public static PropertySourcesPlaceholderConfigurer propertyPlaceholderConfigurer()
{
        return new PropertySourcesPlaceholderConfigurer();
    }
}
```

如果无法解析任何 `${}` 占位符,则使用上述配置 Spring 初始化会失败. 也可以使用 `setPlaceholderPrefix`, `setPlaceholderSuffix` 或 `setValueSeparator` 之类的方法来自定义占位符.

Spring 提供的内置转换器支持允许自动处理简单的类型转换(例如,转换为 `Integer` 或 `int`). 多个逗号分隔的值可以自动转换为 `String` 数组,而无需付出额外的努力.

可以提供如下默认值:

```
@Component
public class MovieRecommender {

    private final String catalog;

    public MovieRecommender(@Value("${catalog.name:defaultCatalog}") String catalog) {
        this.catalog = catalog;
    }
}
```

Spring `BeanPostProcessor` 在后台使用 `ConversionService` 处理将 `@Value` 中的 `String` 值转换为目标类型的过程. 如果要为自己的自定义类型提供转换支持,则可以提供自己的 `ConversionService` bean 实例,如以下示例所示:

```
@Configuration
public class AppConfig {

    @Bean
    public ConversionService conversionService() {
        DefaultFormattingConversionService conversionService = new
DefaultFormattingConversionService();
        conversionService.addConverter(new MyCustomConverter());
        return conversionService;
    }
}
```

当 `@Value` 包含 `SpEL` 表达式 时,该值将在运行时动态计算,如以下示例所示:

```
@Component
public class MovieRecommender {

    private final String catalog;

    public MovieRecommender(@Value("#{systemProperties['user.catalog'] + 'Catalog' }")
String catalog) {
        this.catalog = catalog;
    }
}
```

SpEL 还可以使用更复杂的数据结构:

```
@Component
public class MovieRecommender {

    private final Map<String, Integer> countOfMoviesPerCatalog;

    public MovieRecommender(
            @Value("#{{'Thriller': 100, 'Comedy': 300}}") Map<String, Integer>
countOfMoviesPerCatalog) {
        this.countOfMoviesPerCatalog = countOfMoviesPerCatalog;
    }
}
```

9. `@PostConstruct` 和 `@PreDestroy`

`CommonAnnotationBeanPostProcessor` 不仅仅识别 `@Resource` 注解,还识别 JSR-250 生命周期注解 `javax.annotation.PostConstruct` 和 `javax.annotation.PreDestroy`,在 Spring 2.5 中引入了这些注解, 它们提供了另一个代初始化回调 和销毁回调. 如果 `CommonAnnotationBeanPostProcessor` 在Spring `ApplicationContext` 中注册,它会在相应的 Spring bean 生命周期中调用相应的方法,就像是 Spring 生命周期接口方法,或者是明确声明的回调函数那样. 在以下示例中,缓存在初始化时预先填充并在销毁时清除:

```
public class CachingMovieLister {

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

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

和 `@Resource` 一样,`@PostConstruct` 和 `@PreDestroy` 注解也是 JDK6-8 的标准 java 库中的一部分,但是,在 JDK 9 中,整个 `javax.annotation` 和 java 核心模块分离,最终在 java 11 中移除. 如果你需要引用,则通过 maven 获取 `javax.annotation-api` artifact. 就像其他任何库一样,只需添加到应用程序的类路径中即可.