

Spring2

是关于Spring1的延伸，

现在还是到**@Autowired**

- 可以使用@Autowired注解那些众所周知的解析依赖性接口，比如：BeanFactory, ApplicationContext, Environment, ResourceLoader, ApplicationEventPublisher, and MessageSource

```
public class MovieRecommender {  
  
    @Autowired  
    private ApplicationContext context;  
  
    public MovieRecommender() {  
    }  
  
    // ...  
}
```

什么意思呢，就像上面一样，进行注解，这样就能引用了

- 可以通过添加注解给需要该类型的数组的字段或方法，以提供ApplicationContext中的所有特定类型的bean

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

- 可以用于装配key为String的Map

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

- 如果希望数组有序，可以让bean实现org.springframework.core.Ordered接口或使用的@Order注解

- @Autowired是由Spring BeanPostProcessor处理的，所以不能在自己的BeanPostProcessor或BeanFactoryPostProcessor类型应用这些注解，这些类型必须通过XML或者Spring的@Bean注解加载

```
@Autowired  
  
private List<BeanInterface> list;  
  
public void say() {  
  
    if (null != list) {  
        for (BeanInterface beanInterface : list) {  
            System.out.println(beanInterface.getClass().getName());  
        }  
    }  
    else {  
    }  
}
```

```

        System.out.println("List<BeanInterface> list 是空的!!!!");
    }

}

```

只需要这样，就可以了，进行关于
关于Order是对List有效，但是关于Map是无效的（这里有点问题，我这个list也无效了）

1.Map.Entry说明

Map是java中的接口，Map.Entry是Map的一个内部接口。

Map提供了一些常用方法，如keySet()、entrySet()等方法，keySet()方法返回值是Map中key值的集合；entrySet()的返回值也是返回一个Set集合，此集合的类型为Map.Entry。

Map.Entry是Map声明的一个内部接口，此接口为泛型，定义为Entry<K,V>。它表示Map中的一个实体（一个key-value对）。接口中有getKey(),getValue方法。

使用Map.Entry类，你可以得到在同一时间得到所有的信息

从Map中取得关键字之后，我们必须每次重复返回到Map中取得相对的值，这是很繁琐和费时的。

@Qualifier

@Qualifier

- 按类型自动装配可能多个bean实例的情况，可以使用Spring的@Qualifier注解缩小范围（或指定唯一），也可以用于指定单独的构造器参数或方法参数
- 可用于注解集合类型变量

@Qualifier

```

public class MovieRecommender {

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

```

```

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;
    }

    // ...
}

```

@Qualifier

```
<?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/>

    <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>
```

@Qualifier

- 如果通过名字进行注解注入，主要使用的不是@Autowired(即使在技术上能够通过@Qualifier指定bean的名字)，替代方式是使用JSR-250@Resource注解，它是通过其独特的名称来定义来识别特定的目标（这是一个与所声明的类型是无关的匹配过程）
- 因语义差异，集合或Map类型的bean无法通过@Autowired来注入，因为没有类型匹配到这样的bean，为这些bean使用@Resource注解，通过唯一名称引用集合或Map的bean

@Qualifier

@Autowired适用于fields, constructors, multi-argument methods这些允许在参数级别使用@Qualifier注解缩小范围的情况

@Resource适用于成员变量、只有一个参数的setter方法，所以在目标是构造器或一个多参数方法时，最好的方式是使用qualifiers

@Qualifier

• 定义自己的qualifier注解并使用

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

public class MovieRecommender {

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

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

    // ...
}
```

上面时候自定义@Qualifier呢，就是自定义注解的时候 @Qualifier就是缩小使用范围，使用我们自己定义的Bean
代码如下

```
@Autowired

@Qualifier("beanImplOne")

private BeanInterface beanInterface;
```

```
if(null!=beanInterface) {

    System.out.println();

    System.out.println(beanInterface.getClass().getName());

}else

{

    System.out.println();

}
```

基于Java的容器注解，

@Bean标识一个用于配置和初始化有一个Spring容器管理

的新对象方法，；类型XML配置文件的<bean/>

可以在spring中@Component注解的类中使用@Bean注解任何方法（仅仅是可以）

上一节中，通常使用的都是@Configuration

上一点，通常使用的是@Component

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

这样的上面配置的@Configuration相当于下面的xml配置方式

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

相当于

只不过上面的是用java注解

@Bean

• 自定义Bean name

```
@Configuration
public class AppConfig {

    @Bean(name = "myFoo")
    public Foo foo() {
        return new Foo();
    }

}
```

- init-method
- destroy-method

```
public class Foo {
    public void init() {
        // initialization logic
    }
}

public class Bar {
    public void cleanup() {
        // destruction logic
    }
}

@Configuration
public class AppConfig {

    @Bean(initMethod = "init")
    public Foo foo() {
        return new Foo();
    }

    @Bean(destroyMethod = "cleanup")
    public Bar bar() {
        return new Bar();
    }

}
```

使用bean的时候，如下：

```
@Bean

public Store getStringStore() {

return new StringStore();

}
```

这个时候没有指定名称的话，默认Bean是方法的名称

这里的代码贴在下面，有点问题，就是这个讲师用的是@Configuration这个注解，我用的是@Component这个注解，而讲师的使用的@Configuration注解，是需要Cglib的包才能够使用 而且同时应该配合@Bean注解一起使用 我在这里用的是这个，

```
@Component

public class StoreConfig {

    @Bean(name = "store",initMethod = "init",destroyMethod = "destroy")
    public Store stringStoreTest() {

return new StringStore();

    }

}
```

```
public class StringStore implements Store<String> {

public void init() {

    System.out.println("This is init.");

}

public void destroy() {

    System.out.println("This is destroy.");

}
```

```

}

}

```

```

public interface Store<T> {

}

```

这是用xml配置的方式，下面的图：

```

@Configuration
@ImportResource("classpath:/com/acme/jdbc.properties")
public class AppConfig {

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;

    @Bean
    public DataSource dataSource() {

```

jdbc ResourceBundle

Keys	
jdbc.password	root
jdbc.url	jdbc:mysql://127.0.0.1:3306/database
jdbc.username	root

```

<beans>
<!-- enable processing of annotations such as @Autowired and @Configuration -->
<context:annotation-config/>
<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

<bean class="com.acme.AppConfig"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

```

这样的通过properties加载，通过表达式加载，这样的话，在JDBC中经常使用到

使用@ImportResource和@Value注解进行资源文件读取
注意

@importResource注解是为了
这个是兼容传统xml配置的，毕竟JavaConfig还不是万能的 而不是加载properties文件的 只是用来加载xml的

下面的例子是加载资源文件以及读取资源文件的两种方式

方式一：通过配置文件的方式：

<context:property-placeholder location="" ></context:property> 作用是加载资源文件

那么读取文件可以通过\${key}方式获取配置文件中的值，key表示配置文件中的key

示例中的例子表示将配置文件中jdbc的url,username,password的值分别获取并装配到DriverManagerDataSource类中的url, username, password属性中去。

方式二：通过注解方式获取

将一个用于装载资源文件的类用@Configuration注解该类并通过注解@ImportResource引入配置文件；获取配置文件中的值是通过@Value (“\${key}”) 的方式获取值，@Value要标注到属性上，才能自动将配置文件的值装配到属性上。

用@Value获取配置文件值的时候要注意，如果配置文件中key为“username”，此时spring会读取登录当前操作系统的用户名，所以此种情况要注意，不要用username做key，我们应该用jdbc.username这样的作key

```

package com.Spring.Test;

```

```

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * Created by han on 2016/8/31.
 */

@Configuration
@ImportResource("classpath:Spring/applicationcontext2.xml")
public class Test06 {

    @Value("${Test06.id}")
    private int id;

    @Value("${Test06.name}")
    private String name;

    @Value("${Test06.age}")
    private int age;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

}

```

xml中的配置

```

<context:property-placeholder location="classpath:Test06.properties" file-encoding="UTF-8"/>

```

@bean and @Scope

Bean是单例的

```
@Configuration
public class MyConfiguration {

    @Bean
    @Scope("prototype")
    public Encryptor encryptor() {
        // ...
    }
}
```

```
@Bean
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public UserPreferences userPreferences() {
    return new UserPreferences();
}

@Bean
public Service userService() {
    UserService service = new SimpleUserService();
    // a reference to the proxied userPreferences bean
    service.setUserPreferences(userPreferences());
    return service;
}
```

默认@Bean是单例的

```
@Scope(value="prototype",proxyMode = ScopedProxyMode.DEFAULT)
```

Demo如下

```
@Configuration
public class Test05 {

    @Bean
    @Scope(value = "prototype",proxyMode = ScopedProxyMode.DEFAULT)
    public Test05 test05() {
        return new Test05();
    }

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("Spring/applicationcontext2.xml");
        context.getBean("test05");
    }
}
```

还可以这样搞，通过后面的这个属性来声明

CustomAutowireConfigurer

- CustomAutowireConfigurer是BeanFactoryPostProcessor的子类，通过它可以注册自己的qualifier注解类型(即使没有使用Spring的@Qualifier 注解)

```
<bean id="customAutowireConfigurer"
      class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">
    <property name="customQualifierTypes">
        <set>
            <value>example.CustomQualifier</value>
        </set>
    </property>
</bean>
```

- 该AutowireCandidateResolver决定自动装配的候选者：
 - 每个bean定义的autowire-candidate值
 - 任何<bean/>中的default-autowire-candidates

@Qualifier注解及使用CustomAutowireConfigurer的自定义类型

- @Qualifier注解及使用CustomAutowireConfigurer的自定义类型

关于这个自定义的@Qualifier注解 可以在xml实现的方法 有一个网站有相应的Demo 我就不做了
http://www.concretepage.com/spring/example_customautowireconfigurer_spring

关于Bean容器的基本说完了，主要是这几个部分，

Bean的配置项
Bean的作用域
Bean的生命周期
Bean的自动装配
Resources&ResourceLoader
Bean的注解方式

然后我们由通过注解来实现了一编

这样注解的实现了比较方便，容易实现，那如果xml配置的呢，阅读起来非常的方便，也集中好管理，使用注解的呢，是比较容易读代码的，因为xml的话，这样的话还是要结合xml看代码才行，所以注解比较容易读代码，，，这个讲师的话，更多的是用注解来实现项目工程。在servlet3.0的标准中呢，web.xml这个配置文件就是可以不存在的，这个以前在其他的笔记中有说，现在的话，越少的配置越好，？、但是在改变的时候，这样的改起来可能没有xml的方便。但是，我们在维护比较老的情况下，必须要根据他的工程来根据具体情况来维护，来使用xml或者注解。所以说，这样的配置方法，无论好坏，只有适合的

现在还有一个流行的框架 就是Spring boot 采用的就是全注解的方式