

Spring_day02

今日目标

- 掌握IOC/DI配置管理第三方bean
- 掌握IOC/DI的注解开发
- 掌握IOC/DI注解管理第三方bean
- 完成Spring与Mybatis及JUnit的整合开发

1, IOC/DI配置管理第三方bean

前面所讲的知识点都是基于我们自己写的类，现在如果有需求让我们去管理第三方jar包中的类，该如何管理？

1.1 案例：数据源对象管理

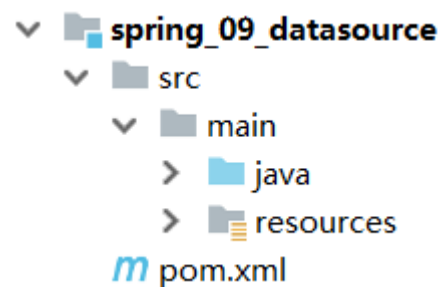
在这一节中，我们将通过一个案例来学习下对于第三方bean该如何进行配置管理。

以后我们会用到很多第三方的bean，本次案例将使用咱们前面提到过的数据源Druid(德鲁伊)和C3P0来配置学习下。

1.1.1 环境准备

学习之前，先来准备下案例环境：

- 创建一个Maven项目



- pom.xml添加依赖

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework</groupId>
4         <artifactId>spring-context</artifactId>
5         <version>5.2.10.RELEASE</version>
6     </dependency>
7 </dependencies>
```

- resources下添加spring的配置文件applicationContext.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="
5           http://www.springframework.org/schema/beans
6           http://www.springframework.org/schema/beans/spring-beans.xsd">
7
8 </beans>
```

- 编写一个运行类App

```
1 public class App {
2     public static void main(String[] args) {
3         ApplicationContext ctx = new
4         ClassPathXmlApplicationContext("applicationContext.xml");
5     }
6 }
```

1.1.2 思路分析

在上述环境下，我们来对数据源进行配置管理，先来分析下思路：

需求：使用Spring的IOC容器来管理Druid连接池对象

1. 使用第三方的技术，需要在pom.xml添加依赖
2. 在配置文件中将【第三方的类】制作成一个bean，让IOC容器进行管理
3. 数据库连接需要基础的四要素 驱动、连接、用户名和密码，【如何注入】到对应的bean中
4. 从IOC容器中获取对应的bean对象，将其打印到控制台查看结果

思考：

- 第三方的类指的是什么？
- 如何注入数据库连接四要素？

1.1.3 实现Druid管理

带着这两个问题，把下面的案例实现下：

步骤1：导入druid的依赖

pom.xml中添加依赖

```
1 <dependency>
2     <groupId>com.alibaba</groupId>
3     <artifactId>druid</artifactId>
4     <version>1.1.16</version>
5 </dependency>
```

步骤2: 配置第三方bean

在applicationContext.xml配置文件中添加DruidDataSource的配置

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="
5           http://www.springframework.org/schema/beans
6           http://www.springframework.org/schema/beans/spring-beans.xsd">
7     <!--管理DruidDataSource对象-->
8     <bean class="com.alibaba.druid.pool.DruidDataSource">
9       <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
10      <property name="url" value="jdbc:mysql://localhost:3306/spring_db"/>
11      <property name="username" value="root"/>
12      <property name="password" value="root"/>
13    </bean>
14 </beans>
```

说明:

- driverClassName: 数据库驱动
- url: 数据库连接地址
- username: 数据库连接用户名
- password: 数据库连接密码
- 数据库连接的四要素要和自己使用的数据库信息一致。

步骤3: 从IOC容器中获取对应的bean对象

```
1 public class App {
2     public static void main(String[] args) {
3         ApplicationContext ctx = new
4         ClassPathXmlApplicationContext("applicationContext.xml");
5         DataSource dataSource = (DataSource) ctx.getBean("dataSource");
6         System.out.println(dataSource);
7     }
8 }
```

步骤4: 运行程序

打印如下结果: 说明第三方bean对象已经被spring的IOC容器进行管理

```
Run: App x
D:\soft\jdk1.8.0_172\bin\java.exe ...
{
    CreateTime:"2021-04-13 18:12:05",
    ActiveCount:0,
    PoolingCount:0,
    CreateCount:0,
    DestroyCount:0,
    CloseCount:0,
    ConnectCount:0,
    Connections:[
    ]
}
Process finished with exit code 0
```

做完案例后，我们可以将刚才思考的两个问题答案说下：

- 第三方的类指的是什么？

```
1 DruidDataSource
```

- 如何注入数据库连接四要素？

```
1 setter注入
```

1.1.4 实现C3P0管理

完成了DruidDataSource的管理，接下来我们再来加深下练习，这次我们来管理C3P0数据源，具体的实现步骤是什么呢？

需求：使用Spring的IOC容器来管理C3P0连接池对象

实现方案和上面基本一致，重点要关注管理的是哪个bean对象？

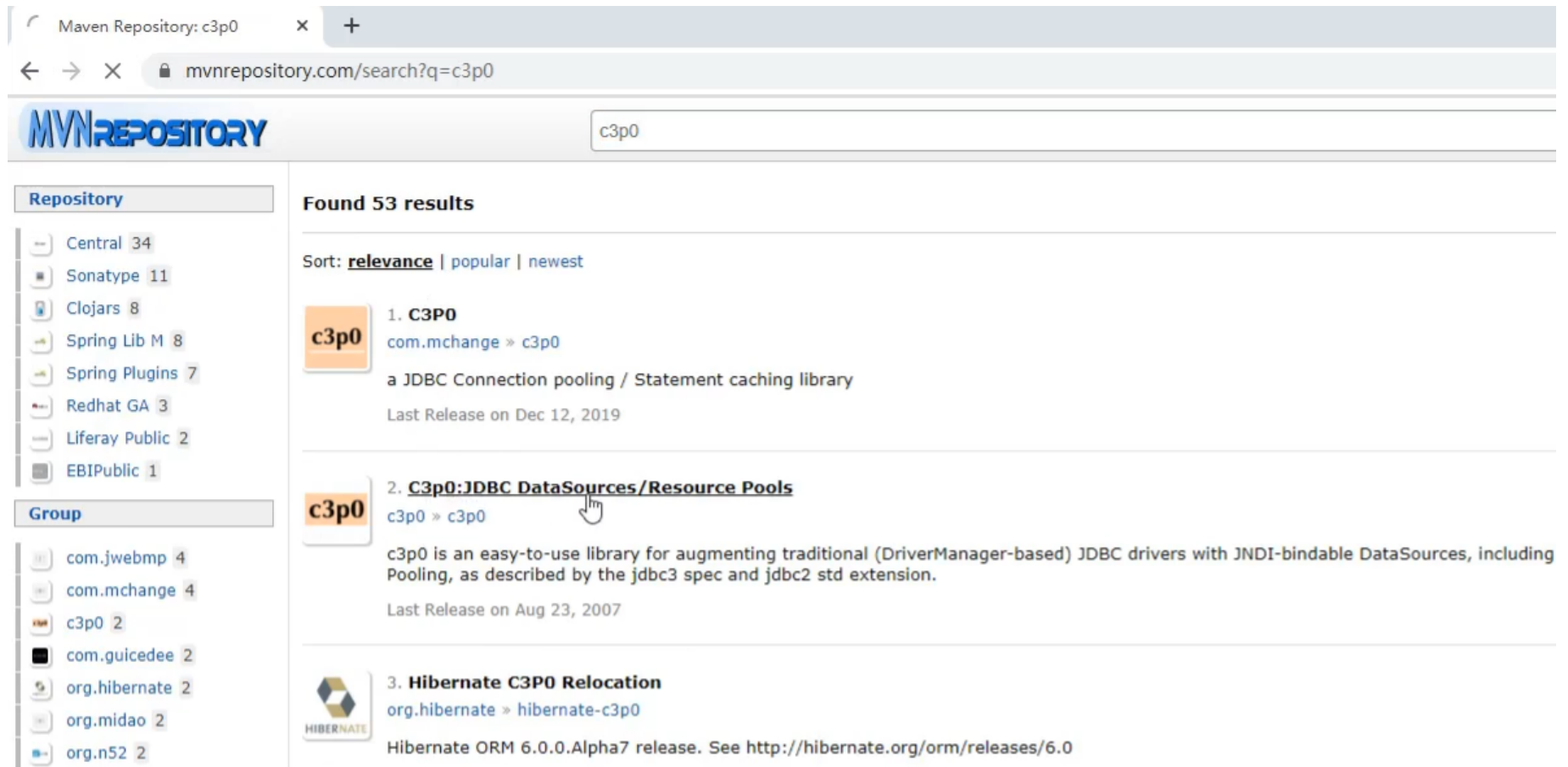
步骤1：导入C3P0的依赖

pom.xml中添加依赖

```
1 <dependency>
2     <groupId>c3p0</groupId>
3     <artifactId>c3p0</artifactId>
4     <version>0.9.1.2</version>
5 </dependency>
```

对于新的技术，不知道具体的坐标该如何查找？

- 直接百度搜索
- 从mvn的仓库<https://mvnrepository.com/>中进行搜索



步骤2:配置第三方bean

在applicationContext.xml配置文件中添加配置

```
1 <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
2     <property name="driverClass" value="com.mysql.jdbc.Driver"/>
3     <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/spring_db"/>
4     <property name="user" value="root"/>
5     <property name="password" value="root"/>
6     <property name="maxPoolSize" value="1000"/>
7 </bean>
```

注意:

- ComboPooledDataSource的属性是通过setter方式进行注入
- 想注入属性就需要在ComboPooledDataSource类或其上层类中有提供属性对应的setter方法
- C3P0的四个属性和Druid的四个属性是不一样的

步骤3:运行程序

程序会报错, 错误如下

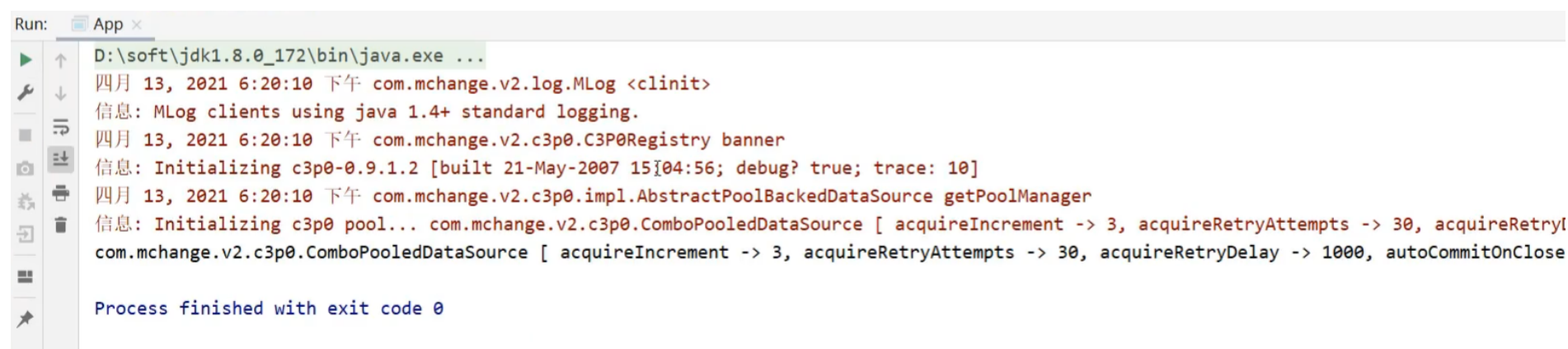


报的错为`ClassNotFoundException`，翻译出来是类没有发现的异常，具体的类为`com.mysql.jdbc.Driver`。错误的原因是缺少mysql的驱动包。

分析出错误的原因，具体的解决方案就比较简单，只需要在`pom.xml`把驱动包引入即可。

```
1 <dependency>
2   <groupId>mysql</groupId>
3   <artifactId>mysql-connector-java</artifactId>
4   <version>5.1.47</version>
5 </dependency>
```

添加完mysql的驱动包以后，再次运行App，就可以打印出结果：



```
Run: App x
D:\soft\jdk1.8.0_172\bin\java.exe ...
四月 13, 2021 6:20:10 下午 com.mchange.v2.log.MLog <clinit>
信息: MLog clients using java 1.4+ standard logging.
四月 13, 2021 6:20:10 下午 com.mchange.v2.c3p0.C3P0Registry banner
信息: Initializing c3p0-0.9.1.2 [built 21-May-2007 15:04:56; debug? true; trace: 10]
四月 13, 2021 6:20:10 下午 com.mchange.v2.c3p0.impl.AbstractPoolBackedDataSource getPoolManager
信息: Initializing c3p0 pool... com.mchange.v2.c3p0.ComboPooledDataSource [ acquireIncrement -> 3, acquireRetryAttempts -> 30, acquireRetryDelay -> 1000, autoCommitOnClose
Process finished with exit code 0
```

注意：

- 数据连接池在配置属性的时候，除了可以注入数据库连接四要素外还可以配置很多其他的属性，具体都有哪些属性用到的时候再去查，一般配置基础的四个，其他都有自己的默认值
- Druid和C3P0在没有导入mysql驱动包的前提下，一个没报错一个报错，说明Druid在初始化的时候没有去加载驱动，而C3P0刚好相反
- Druid程序运行虽然没有报错，但是当调用DruidDataSource的`getConnection()`方法获取连接的时候，也会报找不到驱动类的错误

1.2 加载properties文件

上节中我们已经完成两个数据源druid和C3P0的配置，但是其中包含了一些问题，我们来分析下：

- 这两个数据源中都使用到了一些固定的常量如数据库连接四要素，把这些值写在Spring的配置文件中不利于后期维护
- 需要将这些值提取到一个外部的properties配置文件中
- Spring框架如何从配置文件中读取属性值来配置就是接下来要解决的问题。

问题提出来后，具体该如何实现？

1.2.1 第三方bean属性优化

1.2.1.1 实现思路

需求：将数据库连接四要素提取到properties配置文件，spring来加载配置信息并使用这些信息来完成属性注入。

- 1.在resources下创建一个jdbc.properties(文件的名称可以任意)

2. 将数据库连接四要素配置到配置文件中
3. 在Spring的配置文件中加载properties文件
4. 使用加载到的值实现属性注入

其中第3, 4步骤是需要大家重点关注, 具体是如何实现。

1.2.1.2 实现步骤

步骤1: 准备properties配置文件

resources下创建一个jdbc.properties文件, 并添加对应的属性键值对

```
1 jdbc.driver=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://127.0.0.1:3306/spring_db
3 jdbc.username=root
4 jdbc.password=root
```

步骤2: 开启context命名空间

在applicationContext.xml中开context命名空间

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xsi:schemaLocation="
6           http://www.springframework.org/schema/beans
7           http://www.springframework.org/schema/beans/spring-beans.xsd
8           http://www.springframework.org/schema/context
9           http://www.springframework.org/schema/context/spring-
10      context.xsd">
```

步骤3: 加载properties配置文件

在配置文件中使用context命名空间下的标签来加载properties配置文件

```
1 <context:property-placeholder location="jdbc.properties"/>
```

步骤4: 完成属性注入

使用\${key}来读取properties配置文件中的内容并完成属性注入

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xsi:schemaLocation="
```

```

6      http://www.springframework.org/schema/beans
7      http://www.springframework.org/schema/beans/spring-beans.xsd
8      http://www.springframework.org/schema/context
9      http://www.springframework.org/schema/context/spring-
context.xsd">
10
11     <context:property-placeholder location="jdbc.properties"/>
12     <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
13         <property name="driverClassName" value="${jdbc.driver}"/>
14         <property name="url" value="${jdbc.url}"/>
15         <property name="username" value="${jdbc.username}"/>
16         <property name="password" value="${jdbc.password}"/>
17     </bean>
18 </beans>

```

至此，读取外部properties配置文件中的内容就已经完成。

1.2.2 读取单个属性

1.2.2.1 实现思路

对于上面的案例，效果不是很明显，我们可以换个案例来演示下：

需求：从properties配置文件中读取key为name的值，并将其注入到BookDao中并在save方法中进行打印。

1. 在项目中添加BookDao和BookDaoImpl类
2. 为BookDaoImpl添加一个name属性并提供setter方法
3. 在jdbc.properties中添加数据注入到bookDao中打印方便查询结果
4. 在applicationContext.xml添加配置完成配置文件加载、属性注入（\${key}）

1.2.2.2 实现步骤

步骤1：在项目中添对应的类

BookDao和BookDaoImpl类，并在BookDaoImpl类中添加name属性与setter方法

```

1 public interface BookDao {
2     public void save();
3 }
4
5 public class BookDaoImpl implements BookDao {
6     private String name;
7
8     public void setName(String name) {
9         this.name = name;
10    }

```



```

11
12     public void save() {
13         System.out.println("book dao save ..." + name);
14     }
15 }

```

步骤2: 完成配置文件的读取与注入

在applicationContext.xml添加配置, bean的配置管理、读取外部properties、依赖注入:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:context="http://www.springframework.org/schema/context"
5     xsi:schemaLocation="
6         http://www.springframework.org/schema/beans
7         http://www.springframework.org/schema/beans/spring-beans.xsd
8         http://www.springframework.org/schema/context
9         http://www.springframework.org/schema/context/spring-
context.xsd">
10
11     <context:property-placeholder location="jdbc.properties"/>
12
13     <bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl">
14         <property name="name" value="${jdbc.driver}"/>
15     </bean>
16 </beans>

```

步骤3: 运行程序

在App类中, 从IOC容器中获取bookDao对象, 调用方法, 查看值是否已经被获取到并打印控制台

```

1 public class App {
2     public static void main(String[] args) throws Exception{
3         ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
4         BookDao bookDao = (BookDao) ctx.getBean("bookDao");
5         bookDao.save();
6
7     }
8 }

```

```

Run: App x
D:\soft\jdk1.8.0_172\bin\java.exe ...
book dao save ...com.mysql.jdbc.Driver
Process finished with exit code 0

```

1.2.2.3 注意事项

至此，读取properties配置文件中的内容就已经完成，但是在使用的时候，有些注意事项：

- 问题一：键值对的key为username引发的问题

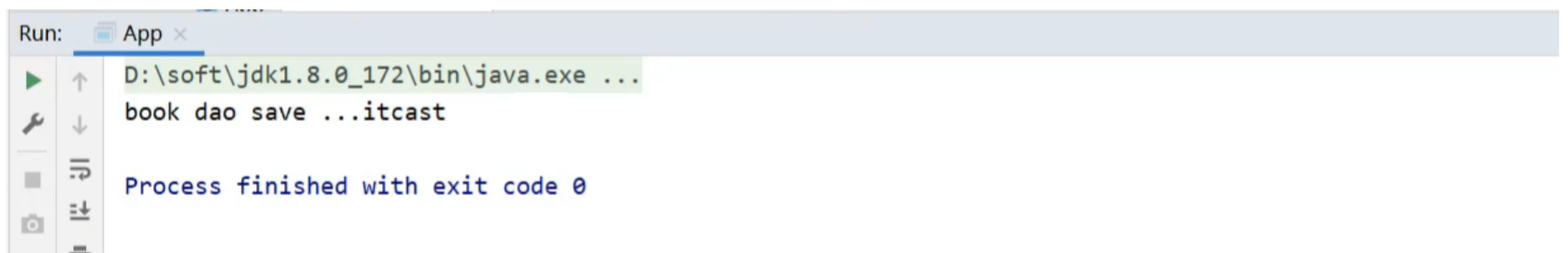
1. 在properties中配置键值对的时候，如果key设置为username

```
1 username=root666
```

2. 在applicationContext.xml注入该属性

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xsi:schemaLocation="
6           http://www.springframework.org/schema/beans
7           http://www.springframework.org/schema/beans/spring-beans.xsd
8           http://www.springframework.org/schema/context
9           http://www.springframework.org/schema/context/spring-
context.xsd">
10
11     <context:property-placeholder location="jdbc.properties"/>
12
13     <bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl">
14       <property name="name" value="${username}"/>
15     </bean>
16 </beans>
```

3. 运行后，在控制台打印的却不是root666，而是自己电脑的用户名



4. 出现问题的原因是<context:property-placeholder/>标签会加载系统的环境变量，而且环境变量的值会被优先加载，如何查看系统的环境变量？

```
1 public static void main(String[] args) throws Exception{
2     Map<String, String> env = System.getenv();
3     System.out.println(env);
4 }
```

大家可以自行运行，在打印出来的结果中会有一个USERNAME=XXX [自己电脑的用户名称]

5. 解决方案

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:context="http://www.springframework.org/schema/context"
5     xsi:schemaLocation="
6         http://www.springframework.org/schema/beans
7         http://www.springframework.org/schema/beans/spring-beans.xsd
8         http://www.springframework.org/schema/context
9         http://www.springframework.org/schema/context/spring-
10    context.xsd">
11     <context:property-placeholder location="jdbc.properties" system-
12    properties-mode="NEVER"/>
13 </beans>

```

system-properties-mode: 设置为NEVER, 表示不加载系统属性, 就可以解决上述问题。

当然还有一个解决方案就是避免使用username作为属性的key。

- 问题二: 当有多个properties配置文件需要被加载, 该如何配置?

1. 调整下配置文件的内容, 在resources下添加jdbc.properties, jdbc2.properties, 内容如下:

jdbc.properties

```

1 jdbc.driver=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://127.0.0.1:3306/spring_db
3 jdbc.username=root
4 jdbc.password=root

```

jdbc2.properties

```

1 username=root666

```

2. 修改applicationContext.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:context="http://www.springframework.org/schema/context"
5     xsi:schemaLocation="
6         http://www.springframework.org/schema/beans
7         http://www.springframework.org/schema/beans/spring-beans.xsd
8         http://www.springframework.org/schema/context
9         http://www.springframework.org/schema/context/spring-
10    context.xsd">
11     <!--方式一 -->

```

```

11     <context:property-placeholder
      location="jdbc.properties,jdbc2.properties" system-properties-
      mode="NEVER"/>
12     <!--方式二-->
13     <context:property-placeholder location="*.properties" system-
      properties-mode="NEVER"/>
14     <!--方式三 -->
15     <context:property-placeholder location="classpath:*.properties"
      system-properties-mode="NEVER"/>
16     <!--方式四-->
17     <context:property-placeholder location="classpath*:*.properties"
      system-properties-mode="NEVER"/>
18 </beans>

```

说明:

- 方式一:可以实现,如果配置文件多的话,每个都需要配置
- 方式二: `*.properties` 代表所有以 `properties` 结尾的文件都会被加载,可以解决方式一的问题,但是不标准
- 方式三:标准的写法, `classpath:` 代表的是从根路径下开始查找,但是只能查询当前项目的根路径
- 方式四:不仅可以加载当前项目还可以加载当前项目所依赖的所有项目的根路径下的 `properties` 配置文件

1.2.3 加载properties文件小结

本节主要讲解的是 `properties` 配置文件的加载,需要掌握的内容有:

- 如何开启 `context` 命名空间

```

<?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">
</beans>

```

- 如何加载 `properties` 配置文件

```
1 <context:property-placeholder location="" system-properties-mode="NEVER"/>
```

- 如何在 `applicationContext.xml` 引入 `properties` 配置文件中的值

```
1 ${key}
```

2, 核心容器

前面已经完成bean与依赖注入的相关知识学习，接下来我们主要学习的是IOC容器中的**核心容器**。

这里所说的核心容器，大家可以把它简单的理解为ApplicationContext，前面虽然已经用到过，但是并没有系统的学习，接下来咱们从以下几个问题入手来学习下容器的相关知识：

- 如何创建容器？
- 创建好容器后，如何从容器中获取bean对象？
- 容器类的层次结构是什么？
- BeanFactory是什么？

2.1 环境准备

在学习和解决上述问题之前，先来准备下案例环境：

- 创建一个Maven项目
- pom.xml添加Spring的依赖

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework</groupId>
4         <artifactId>spring-context</artifactId>
5         <version>5.2.10.RELEASE</version>
6     </dependency>
7 </dependencies>
```

- resources下添加applicationContext.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="
5         http://www.springframework.org/schema/beans
6         http://www.springframework.org/schema/beans/spring-beans.xsd">
7     <bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl"/>
8 </beans>
```

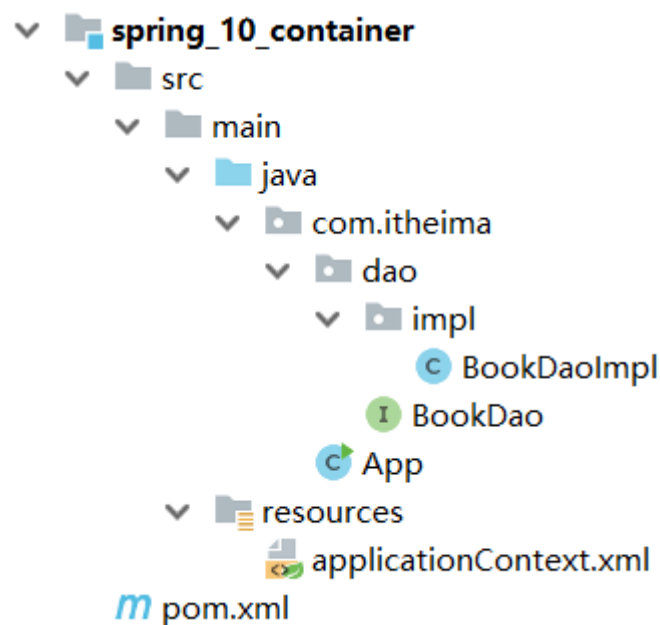
- 添加BookDao和BookDaoImpl类

```
1 public interface BookDao {
2     public void save();
3 }
4 public class BookDaoImpl implements BookDao {
5     public void save() {
6         System.out.println("book dao save ..." );
7     }
8 }
```

- 创建运行类App

```
1 public class App {
2     public static void main(String[] args) {
3         ApplicationContext ctx = new
4         ClassPathXmlApplicationContext("applicationContext.xml");
5         BookDao bookDao = (BookDao) ctx.getBean("bookDao");
6         bookDao.save();
7     }
8 }
```

最终创建好的项目结构如下：



2.2 容器

2.2.1 容器的创建方式

案例中创建 `ApplicationContext` 的方式为：

```
1 ApplicationContext ctx = new
2   ClassPathXmlApplicationContext("applicationContext.xml");
```

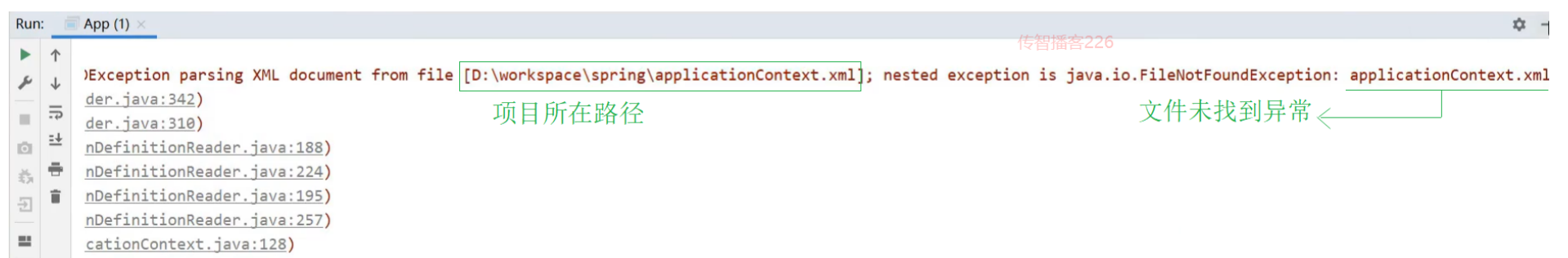
这种方式翻译为：**类路径下的XML配置文件**

除了上面这种方式，Spring还提供了另外一种创建方式为：

```
1 ApplicationContext ctx = new
2   FileSystemXmlApplicationContext("applicationContext.xml");
```

这种方式翻译为：**文件系统下的XML配置文件**

使用这种方式，运行，会出现如下错误：



从错误信息中能发现，这种方式是从项目路径下开始查找 applicationContext.xml 配置文件的，所以需要将其修改为：

```
1 ApplicationContext ctx = new
  FileSystemXmlApplicationContext("D:\\workspace\\spring\\spring_10_container\\s
  rc\\main\\resources\\applicationContext.xml");
```

说明：大家练习的时候，写自己的具体路径。

这种方式虽能实现，但是当项目的位置发生变化后，代码也需要跟着改，耦合度较高，不推荐使用。

2.2.2 Bean的三种获取方式

方式一，就是目前案例中获取的方式：

```
1 BookDao bookDao = (BookDao) ctx.getBean("bookDao");
```

这种方式存在的问题是每次获取的时候都需要进行类型转换，有没有更简单的方式呢？

方式二：

```
1 BookDao bookDao = ctx.getBean("bookDao", BookDao.class);
```

这种方式可以解决类型强转问题，但是参数又多加了一个，相对来说没有简化多少。

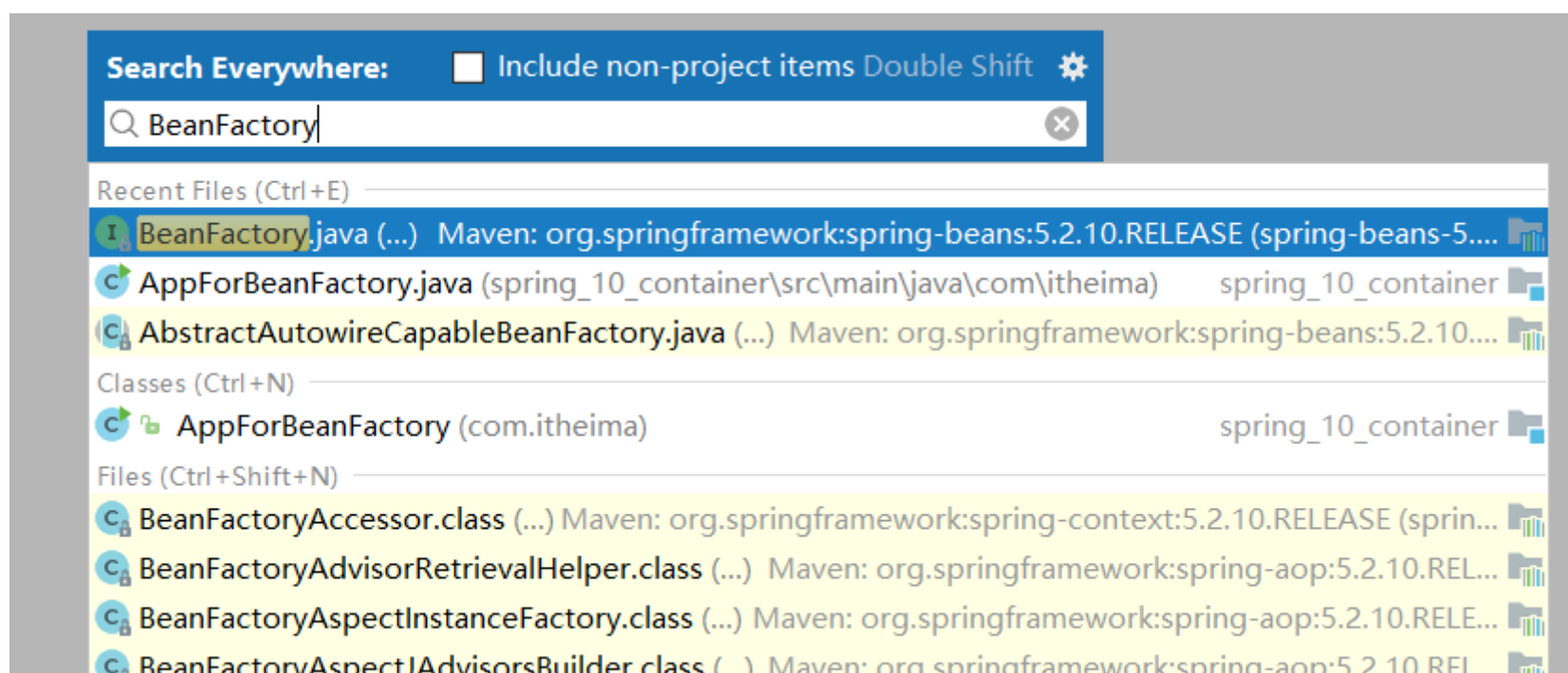
方式三：

```
1 BookDao bookDao = ctx.getBean(BookDao.class);
```

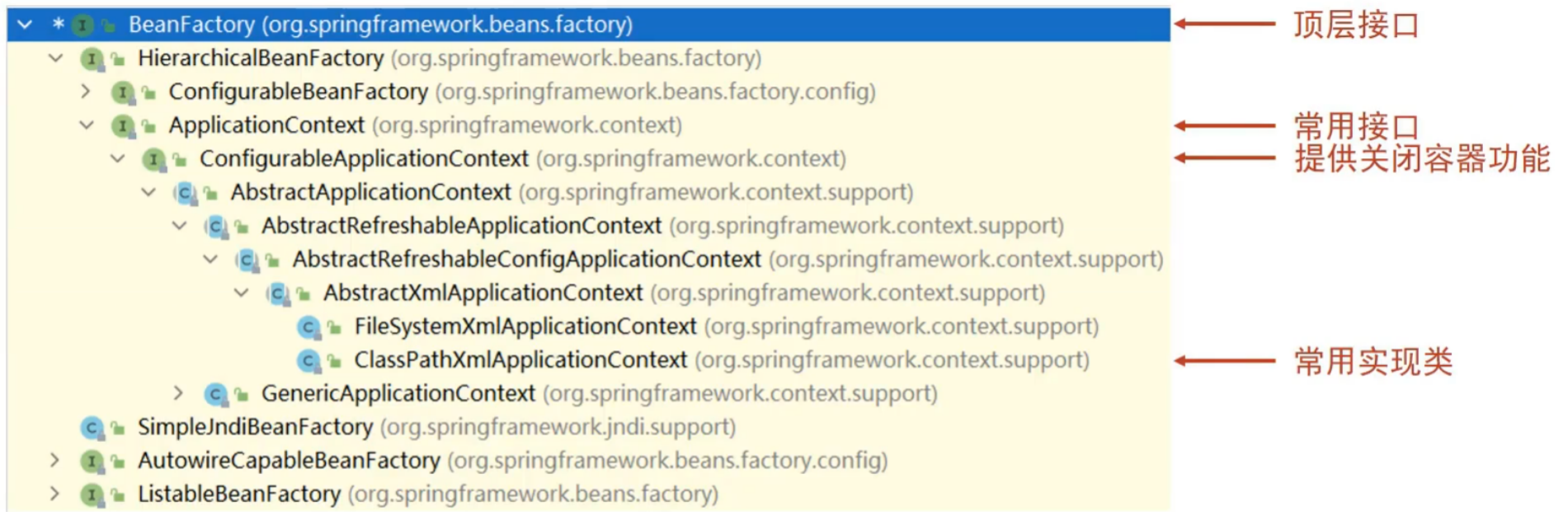
这种方式就类似我们之前所学习依赖注入中的按类型注入。必须要确保IOC容器中该类型对应的bean对象只能有一个。

2.2.3 容器类层次结构

(1) 在IDEA中双击 `shift`，输入 `BeanFactory`



(2) 点击进入 `BeanFactory` 类，`ctrl+h`，就能查看到如下结构的层次关系



从图中可以看出，容器类也是从无到有根据需要一层层叠加上来的，大家重点了解下这种设计思想。

2.2.4 BeanFactory的使用

使用BeanFactory来创建IOC容器的具体实现方式为：

```

1 public class AppForBeanFactory {
2     public static void main(String[] args) {
3         Resource resources = new ClassPathResource("applicationContext.xml");
4         BeanFactory bf = new XmlBeanFactory(resources);
5         BookDao bookDao = bf.getBean(BookDao.class);
6         bookDao.save();
7     }
8 }

```

为了更好的看出BeanFactory和ApplicationContext之间的区别，在BookDaoImpl添加如下构造函数：

```

1 public class BookDaoImpl implements BookDao {
2     public BookDaoImpl() {
3         System.out.println("constructor");
4     }
5     public void save() {
6         System.out.println("book dao save ..." );
7     }
8 }

```

如果不去获取bean对象，打印会发现：

- BeanFactory是延迟加载，只有在获取bean对象的时候才会去创建
- ApplicationContext是立即加载，容器加载的时候就会创建bean对象
- ApplicationContext要想成为延迟加载，只需要按照如下方式进行配置


```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="
5           http://www.springframework.org/schema/beans
6           http://www.springframework.org/schema/beans/spring-beans.xsd">
7   <bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl" lazy-
  init="true"/>
8 </beans>
```

小结

这一节中所讲的知识点包括：

- 容器创建的两种方式
 - `ClassPathXmlApplicationContext` [掌握]
 - `FileSystemXmlApplicationContext` [知道即可]
- 获取Bean的三种方式
 - `getBean("名称")` : 需要类型转换
 - `getBean("名称", 类型.class)` : 多了一个参数
 - `getBean(类型.class)` : 容器中不能有多个该类的bean对象

上述三种方式，各有各的优缺点，用哪个都可以。

- 容器类层次结构
 - 只需要知晓容器的最上级的父接口为 `BeanFactory`即可
- `BeanFactory`
 - 使用`BeanFactory`创建的容器是延迟加载
 - 使用`ApplicationContext`创建的容器是立即加载
 - 具体`BeanFactory`如何创建只需要了解即可。

2.2 核心容器总结

这节中没有新的知识点，只是对前面知识的一个大总结，共包含如下内容：

2.2.1 容器相关

- `BeanFactory`是IoC容器的顶层接口，初始化`BeanFactory`对象时，加载的bean延迟加载
- `ApplicationContext`接口是Spring容器的核心接口，初始化时bean立即加载
- `ApplicationContext`接口提供基础的bean操作相关方法，通过其他接口扩展其功能
- `ApplicationContext`接口常用初始化类
 - **`ClassPathXmlApplicationContext` (常用)**
 - `FileSystemXmlApplicationContext`

2.2.2 bean相关

<code><bean</code>	
<code>id="bookDao"</code>	bean的Id
<code>name="dao bookDaoImpl daoImpl"</code>	bean别名
<code>class="com.itheima.dao.impl.BookDaoImpl"</code>	bean类型, 静态工厂类, FactoryBean类
<code>scope="singleton"</code>	控制bean的实例数量
<code>init-method="init"</code>	生命周期初始化方法
<code>destroy-method="destory"</code>	生命周期销毁方法
<code>autowire="byType"</code>	自动装配类型
<code>factory-method="getInstance"</code>	bean工厂方法, 应用于静态工厂或实例工厂
<code>factory-bean="com.itheima.factory.BookDaoFactory"</code>	实例工厂bean
<code>lazy-init="true"</code>	控制bean延迟加载
<code></></code>	

其实整个配置中最常用的就两个属性id和class。

把scope、init-method、destroy-method框起来的原因是, 后面注解在讲解的时候还会用到, 所以大家对这三个属性关注下。

2.2.3 依赖注入相关

	<code><bean id="bookService" class="com.itheima.service.impl.BookServiceImpl"></code>	
构造器注入 一般用在第三方技术整合	<code><constructor-arg name="bookDao" ref="bookDao"/></code>	构造器注入引用类型
	<code><constructor-arg name="userDao" ref="userDao"/></code>	
	<code><constructor-arg name="msg" value="WARN"/></code>	构造器注入简单类型
	<code><constructor-arg type="java.lang.String" index="3" value="WARN"/></code>	类型匹配与索引匹配
setter注入 对于我们自己写的类, 这种方式用的相对比较多	<code><property name="bookDao" ref="bookDao"/></code>	setter注入引用类型
	<code><property name="userDao" ref="userDao"/></code>	
	<code><property name="msg" value="WARN"/></code>	setter注入简单类型
	<code><property name="names"></code>	setter注入集合类型
	<code><list></code>	list集合
	<code><value>itcast</value></code>	集合注入简单类型
	<code><ref bean="dataSource"/></code>	集合注入引用类型
	<code></list></code>	
	<code></property></code>	
	<code></bean></code>	

3, IOC/DI注解开发

Spring的IOC/DI对应的配置开发就已经讲解完成, 但是使用起来相对来说还是比较复杂的, 复杂的地方在**配置文件**。

前面咱们聊Spring的时候说过, Spring可以简化代码的开发, 到现在并没有体会到。

所以Spring到底是如何简化代码开发的呢?

要想真正简化开发, 就需要用到Spring的注解开发, Spring对注解支持的版本历程:

- 2.0版开始支持注解
- 2.5版注解功能趋于完善

- 3.0版支持纯注解开发

关于注解开发，我们会讲解两块内容 注解开发定义bean和纯注解开发。

注解开发定义bean用的是2.5版提供的注解，纯注解开发用的是3.0版提供的注解。

3.1 环境准备

在学习注解开发之前，先来准备下案例环境：

- 创建一个Maven项目
- pom.xml添加Spring的依赖

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework</groupId>
4         <artifactId>spring-context</artifactId>
5         <version>5.2.10.RELEASE</version>
6     </dependency>
7 </dependencies>
```

- resources下添加applicationContext.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="
5           http://www.springframework.org/schema/beans
6           http://www.springframework.org/schema/beans/spring-beans.xsd">
7     <bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl"/>
8 </beans>
```

- 添加BookDao、BookDaoImpl、BookService、BookServiceImpl类

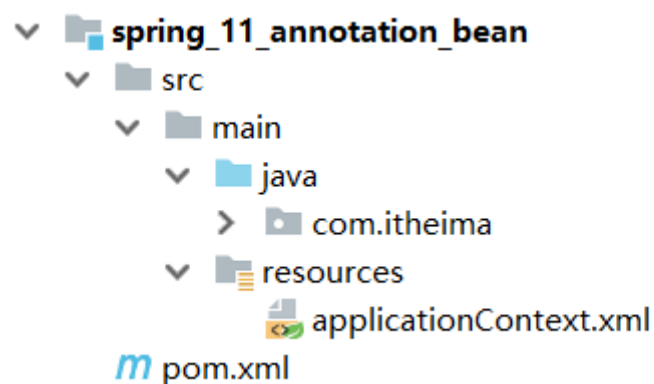
```
1 public interface BookDao {
2     public void save();
3 }
4 public class BookDaoImpl implements BookDao {
5     public void save() {
6         System.out.println("book dao save ..." );
7     }
8 }
9 public interface BookService {
10    public void save();
11 }
12
13 public class BookServiceImpl implements BookService {
14    public void save() {
15        System.out.println("book service save ...");
16    }
17 }
```

```
16     }
17 }
18
```

- 创建运行类App

```
1 public class App {
2     public static void main(String[] args) {
3         ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
4         BookDao bookDao = (BookDao) ctx.getBean("bookDao");
5         bookDao.save();
6     }
7 }
```

最终创建好的项目结构如下：



3.2 注解开发定义bean

在上述环境的基础上，我们来学一学Spring是如何通过注解实现bean的定义开发？

步骤1：删除原XML配置

将配置文件中的<bean> 标签删除掉

```
1 <bean id="bookDao" class="com.itheima.dao.impl.BookDaoImpl"/>
```

步骤2：Dao上添加注解

在BookDaoImpl类上添加@Component 注解

```
1 @Component("bookDao")
2 public class BookDaoImpl implements BookDao {
3     public void save() {
4         System.out.println("book dao save ..." );
5     }
6 }
```

注意：@Component注解不可以添加在接口上，因为接口是无法创建对象的。

XML与注解配置的对应关系：



步骤3: 配置Spring的注解包扫描

为了让Spring框架能够扫描到写在类上的注解, 需要在配置文件上进行包扫描

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="
5         http://www.springframework.org/schema/beans
6         http://www.springframework.org/schema/beans/spring-beans.xsd">
7     <context:component-scan base-package="com.itheima"/>
8 </beans>
```

说明:

component-scan

- component: 组件, Spring将管理的bean视作自己的一个组件
- scan: 扫描

base-package指定Spring框架扫描的包路径, 它会扫描指定包及其子包中的所有类上的注解。

- 包路径越多 [如: com.itheima.dao.impl], 扫描的范围越小速度越快
- 包路径越少 [如: com.itheima], 扫描的范围越大速度越慢
- 一般扫描到项目的组织名称即Maven的groupId下 [如: com.itheima] 即可。

步骤4: 运行程序

运行App类查看打印结果

```
Run: App x
D:\soft\jdk1.8.0_172\bin\java.exe ...
com.itheima.dao.impl.BookDaoImpl@73d4cc9e
Process finished with exit code 0
```

步骤5: Service上添加注解

在BookServiceImpl类上也添加@Component交给Spring框架管理

```
1 @Component
2 public class BookServiceImpl implements BookService {
3     private BookDao bookDao;
4
5     public void setBookDao(BookDao bookDao) {
6         this.bookDao = bookDao;
7     }
8
9     public void save() {
10        System.out.println("book service save ...");
11        bookDao.save();
12    }
13 }
```

步骤6: 运行程序

在App类中, 从IOC容器中获取BookServiceImpl对应的bean对象, 打印

```
1 public class App {
2     public static void main(String[] args) {
3         ApplicationContext ctx = new
4         ClassPathXmlApplicationContext("applicationContext.xml");
5         BookDao bookDao = (BookDao) ctx.getBean("bookDao");
6         System.out.println(bookDao);
7         //按类型获取bean
8         BookService bookService = ctx.getBean(BookService.class);
9         System.out.println(bookService);
10    }
11 }
```

打印观察结果, 两个bean对象都已经打印到控制台

```

Run: App x
D:\soft\jdk1.8.0_172\bin\java.exe ...
com.itheima.dao.impl.BookDaoImpl@57f23557
com.itheima.service.impl.BookServiceImpl@43d7741f
Process finished with exit code 0

```

说明:

- BookServiceImpl类没有起名称,所以在App中是按照类型来获取bean对象
- @Component注解如果不起名称,会有一个默认值就是当前类名首字母小写,所以也可以按照名称获取,如

```

1 BookService bookService = (BookService)ctx.getBean("bookServiceImpl");
2 System.out.println(bookService);

```

对于@Component注解,还衍生出了其他三个注解@Controller、@Service、@Repository

通过查看源码会发现:

```

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

```

这三个注解和@Component注解的作用是一样的,为什么要衍生出这三个呢?

方便我们后期在编写类的时候能很好的区分出这个类是属于表现层、业务层还是数据层的类。

知识点1: @Component等

名称	@Component/@Controller/@Service/@Repository
类型	类注解
位置	类定义上方
作用	设置该类为spring管理的bean
属性	value (默认): 定义bean的id

3.2 纯注解开发模式

上面已经可以使用注解来配置bean,但是依然有用到配置文件,在配置文件中对包进行了扫描, Spring在3.0版已经支持纯注解开发

- Spring3.0开启了纯注解开发模式,使用Java类替代配置文件,开启了Spring快速开发赛道

具体如何实现?

3.2.1 思路分析

实现思路为：

- 将配置文件applicationContext.xml删除掉，使用类来替换。

3.2.2 实现步骤

步骤1：创建配置类

创建一个配置类SpringConfig

```
1 public class SpringConfig {
2 }
3
```

步骤2：标识该类为配置类

在配置类上添加@Configuration注解，将其标识为一个配置类，替换applicationContext.xml

```
1 @Configuration
2 public class SpringConfig {
3 }
```

步骤3：用注解替换包扫描配置

在配置类上添加包扫描注解@ComponentScan替换<context:component-scan base-package=""/>

```
1 @Configuration
2 @ComponentScan("com.itheima")
3 public class SpringConfig {
4 }
```

步骤4：创建运行类并执行

创建一个新的运行类AppForAnnotation

```
1 public class AppForAnnotation {
2
3     public static void main(String[] args) {
4         ApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
5         BookDao bookDao = (BookDao) ctx.getBean("bookDao");
6         System.out.println(bookDao);
7         BookService bookService = ctx.getBean(BookService.class);
8         System.out.println(bookService);
9     }
10 }
```

运行AppForAnnotation，可以看到两个对象依然被获取成功

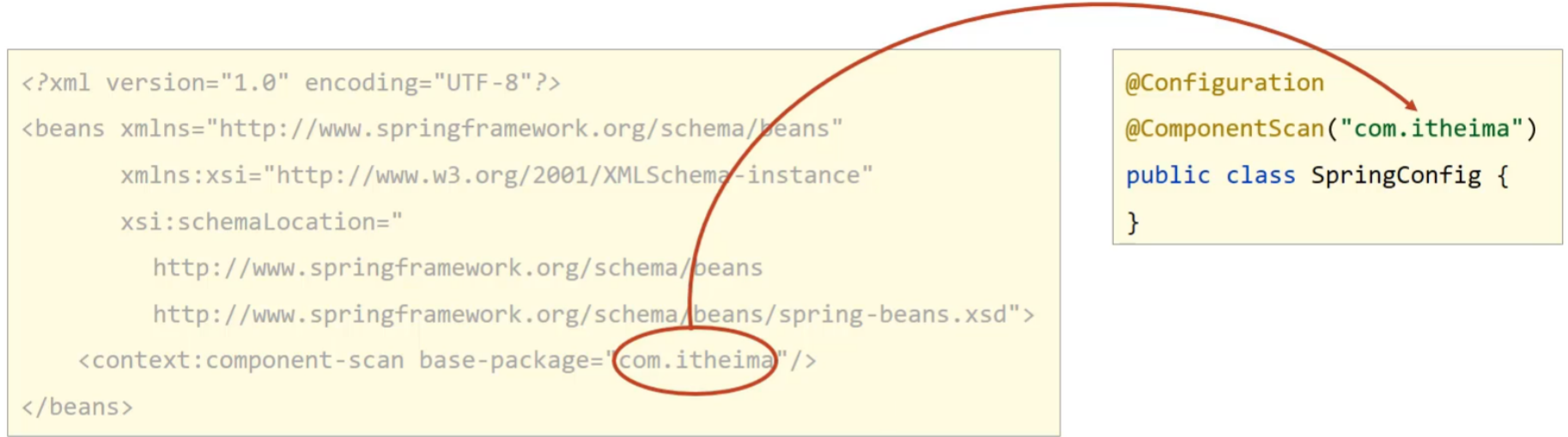

```

Run: AppForAnnotation x
D:\soft\jdk1.8.0_172\bin\java.exe ...
com.itheima.dao.impl.BookDaoImpl@51931956
com.itheima.service.impl.BookServiceImpl@62e136d3
Process finished with exit code 0

```

至此，纯注解开发的方式就已经完成了，主要内容包括：

- Java类替换Spring核心配置文件



- @Configuration注解用于设定当前类为配置类
- @ComponentScan注解用于设定扫描路径，此注解只能添加一次，多个数据请用数组格式

```

1 @ComponentScan({com.itheima.service", "com.itheima.dao"})

```

- 读取Spring核心配置文件初始化容器对象切换为读取Java配置类初始化容器对象

```

1 //加载配置文件初始化容器
2 ApplicationContext ctx = new
  ClassPathXmlApplicationContext("applicationContext.xml");
3 //加载配置类初始化容器
4 ApplicationContext ctx = new
  AnnotationConfigApplicationContext(SpringConfig.class);

```

知识点1: @Configuration

名称	@Configuration
类型	类注解
位置	类定义上方
作用	设置该类为spring配置类
属性	value (默认) : 定义bean的id

知识点2: @ComponentScan

名称	@ComponentScan
类型	类注解
位置	类定义上方
作用	设置spring配置类扫描路径，用于加载使用注解格式定义的bean
属性	value（默认）：扫描路径，此路径可以逐层向下扫描

小结：

这一节重点掌握的是使用注解完成Spring的bean管理，需要掌握的内容为：

- 记住@Component、@Controller、@Service、@Repository这四个注解
- applicationContext.xml中<context:component-scan/>的作用是指定扫描包路径，注解为@ComponentScan
- @Configuration标识该类为配置类，使用类替换applicationContext.xml文件
- ClassPathXmlApplicationContext是加载XML配置文件
- AnnotationConfigApplicationContext是加载配置类

3.3 注解开发bean作用范围与生命周期管理

使用注解已经完成了bean的管理，接下来按照前面所学习的内容，将通过配置实现的内容都换成对应的注解实现，包含两部分内容：bean作用范围和bean生命周期。

3.3.1 环境准备

老规矩，学习之前先来准备环境：

- 创建一个Maven项目
- pom.xml添加Spring的依赖

```

1 <dependencies>
2     <dependency>
3         <groupId>org.springframework</groupId>
4         <artifactId>spring-context</artifactId>
5         <version>5.2.10.RELEASE</version>
6     </dependency>
7 </dependencies>

```

- 添加一个配置类 SpringConfig

```

1 @Configuration
2 @ComponentScan("com.itheima")
3 public class SpringConfig {
4 }

```

- 添加BookDao、BookDaoImpl类

```

1 public interface BookDao {
2     public void save();
3 }
4 @Repository
5 public class BookDaoImpl implements BookDao {
6     public void save() {
7         System.out.println("book dao save ..." );
8     }
9 }

```

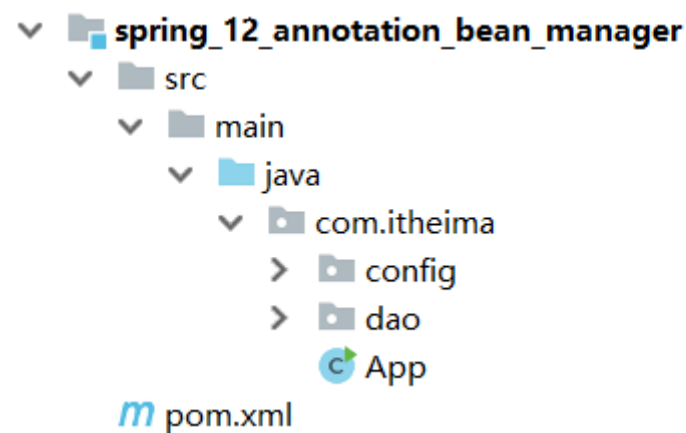
- 创建运行类App

```

1 public class App {
2     public static void main(String[] args) {
3         AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
4         BookDao bookDao1 = ctx.getBean(BookDao.class);
5         BookDao bookDao2 = ctx.getBean(BookDao.class);
6         System.out.println(bookDao1);
7         System.out.println(bookDao2);
8     }
9 }

```

最终创建好的项目结构如下：



3.3.2 Bean的作用范围

- (1) 先运行App类,在控制台打印两个一模一样的地址,说明默认情况下bean是单例

```

Run: App (1) x
D:\soft\jdk1.8.0_172\bin\java.exe ...
com.itheima.dao.impl.BookDaoImpl@491666ad
com.itheima.dao.impl.BookDaoImpl@491666ad
Process finished with exit code 0

```

- (2) 要想将BookDaoImpl变成非单例,只需要在其类上添加@scope注解

```

1 @Repository
2 // @Scope设置bean的作用范围
3 @Scope("prototype")
4 public class BookDaoImpl implements BookDao {
5
6     public void save() {
7         System.out.println("book dao save ...");
8     }
9 }

```

再次执行App类，打印结果：

```

Run: App (1) x
D:\soft\jdk1.8.0_172\bin\java.exe ...
com.itheima.dao.impl.BookDaoImpl@57175e74
com.itheima.dao.impl.BookDaoImpl@7bb58ca3
Process finished with exit code 0

```

知识点1: @Scope

名称	@Scope
类型	类注解
位置	类定义上方
作用	设置该类创建对象的作用范围 可用于设置创建出的bean是否为单例对象
属性	value (默认) : 定义bean作用范围, 默认值singleton (单例) , 可选值prototype (非单例)

3.3.3 Bean的生命周期

(1) 在BookDaoImpl中添加两个方法, `init`和`destroy`, 方法名可以任意

```

1 @Repository
2 public class BookDaoImpl implements BookDao {
3     public void save() {
4         System.out.println("book dao save ...");
5     }
6     public void init() {
7         System.out.println("init ...");
8     }
9     public void destroy() {
10        System.out.println("destroy ...");
11    }
12 }
13

```

(2) 如何对方法进行标识, 哪个是初始化方法, 哪个是销毁方法?

只需要在对应的方法上添加 `@PostConstruct` 和 `@PreDestroy` 注解即可。

```

1 @Repository
2 public class BookDaoImpl implements BookDao {
3     public void save() {
4         System.out.println("book dao save ...");
5     }
6     @PostConstruct //在构造方法之后执行, 替换 init-method
7     public void init() {
8         System.out.println("init ...");
9     }
10    @PreDestroy //在销毁方法之前执行, 替换 destroy-method
11    public void destroy() {
12        System.out.println("destroy ...");
13    }
14 }
15

```

(3) 要想看到两个方法执行, 需要注意的是 `destroy` 只有在容器关闭的时候, 才会执行, 所以需要修改 `App` 的类

```

1 public class App {
2     public static void main(String[] args) {
3         AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
4         BookDao bookDao1 = ctx.getBean(BookDao.class);
5         BookDao bookDao2 = ctx.getBean(BookDao.class);
6         System.out.println(bookDao1);
7         System.out.println(bookDao2);
8         ctx.close(); //关闭容器
9     }
10 }

```

(4) 运行App, 类查看打印结果, 证明init和destroy方法都被执行了。

```
Run: App (1) x
D:\soft\jdk1.8.0_172\bin\java.exe ...
init ...
com.itheima.dao.impl.BookDaoImpl@1a38c59b
com.itheima.dao.impl.BookDaoImpl@1a38c59b
destroy ...
Process finished with exit code 0
```

注意: @PostConstruct和@PreDestroy注解如果找不到, 需要导入下面的jar包

```
1 <dependency>
2   <groupId>javax.annotation</groupId>
3   <artifactId>javax.annotation-api</artifactId>
4   <version>1.3.2</version>
5 </dependency>
```

找不到的原因是, 从JDK9以后jdk中的javax.annotation包被移除了, 这两个注解刚好就在这个包中。

知识点1: @PostConstruct

名称	@PostConstruct
类型	方法注解
位置	方法上
作用	设置该方法为初始化方法
属性	无

知识点2: @PreDestroy

名称	@PreDestroy
类型	方法注解
位置	方法上
作用	设置该方法为销毁方法
属性	无

小结

```
<bean id="" class="" scope="" init-method="" destroy-method="" />
```

↓ ↓ ↓ ↓

@Component("") @Scope() @PostConstruct @PreDestroy

@Controller("")

@Service("")

@Repository("")

3.4 注解开发依赖注入

Spring为了使用注解简化开发，并没有提供构造函数注入、setter注入对应的注解，只提供了自动装配的注解实现。

3.4.1 环境准备

在学习之前，把案例环境介绍下：

- 创建一个Maven项目
- pom.xml添加Spring的依赖

```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework</groupId>
4     <artifactId>spring-context</artifactId>
5     <version>5.2.10.RELEASE</version>
6   </dependency>
7 </dependencies>
```

- 添加一个配置类 SpringConfig

```
1 @Configuration
2 @ComponentScan("com.itheima")
3 public class SpringConfig {
4 }
```

- 添加BookDao、BookDaoImpl、BookService、BookServiceImpl类

```
1 public interface BookDao {
2   public void save();
3 }
4 @Repository
5 public class BookDaoImpl implements BookDao {
6   public void save() {
7     System.out.println("book dao save ..." );
8   }
9 }
```

```

10 public interface BookService {
11     public void save();
12 }
13 @Service
14 public class BookServiceImpl implements BookService {
15     private BookDao bookDao;
16     public void setBookDao(BookDao bookDao) {
17         this.bookDao = bookDao;
18     }
19     public void save() {
20         System.out.println("book service save ...");
21         bookDao.save();
22     }
23 }

```

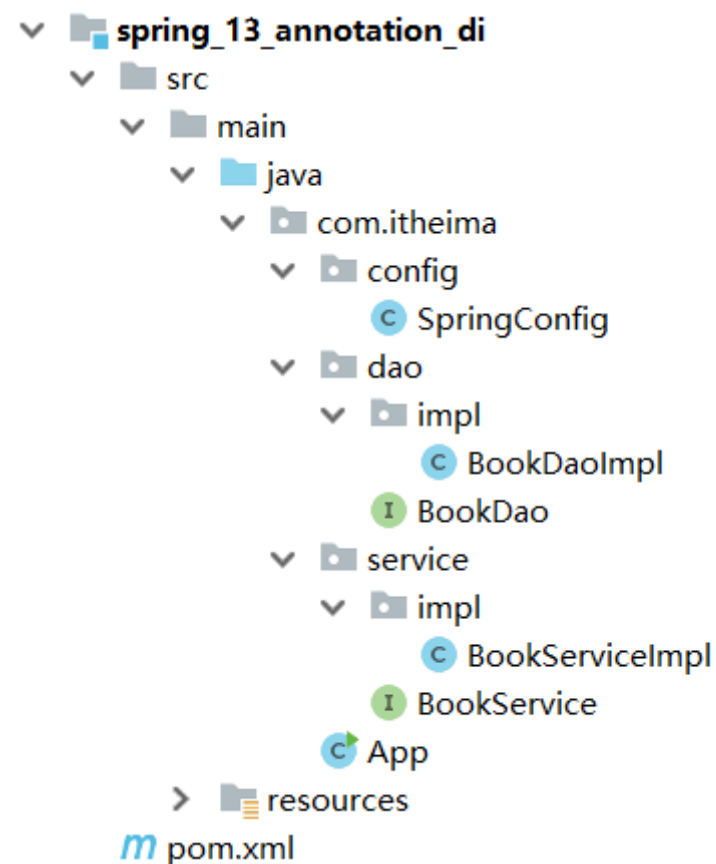
- 创建运行类App

```

1 public class App {
2     public static void main(String[] args) {
3         AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
4         BookService bookService = ctx.getBean(BookService.class);
5         bookService.save();
6     }
7 }

```

最终创建好的项目结构如下：



环境准备好后，运行后会发现有问題


```
Run: App (2) x
D:\soft\jdk1.8.0_172\bin\java.exe ...
book service save ...
Exception in thread "main" java.lang.NullPointerException Create breakpoint
    at com.itheima.service.impl.BookServiceImpl.save(BookServiceImpl.java:20)
    at com.itheima.App.main(App.java:14)
Process finished with exit code 1
```

出现问题的原因是，在BookServiceImpl类中添加了BookDao的属性，并提供了setter方法，但是目前是没有提供配置注入BookDao的，所以bookDao对象为Null，调用其save方法就会报空指针异常。

3.4.2 注解实现按照类型注入

对于这个问题使用注解该如何解决？

- (1) 在BookServiceImpl类的bookDao属性上添加@Autowired注解

```
1 @Service
2 public class BookServiceImpl implements BookService {
3     @Autowired
4     private BookDao bookDao;
5
6     // public void setBookDao(BookDao bookDao) {
7     //     this.bookDao = bookDao;
8     // }
9     public void save() {
10        System.out.println("book service save ...");
11        bookDao.save();
12    }
13 }
```

注意：

- @Autowired可以写在属性上，也可也写在setter方法上，最简单的处理方式是写在属性上并将setter方法删除掉
- 为什么setter方法可以删除呢？
 - 自动装配基于反射设计创建对象并通过暴力反射为私有属性进行设值
 - 普通反射只能获取public修饰的内容
 - 暴力反射除了获取public修饰的内容还可以获取private修改的内容
 - 所以此处无需提供setter方法

- (2) @Autowired是按照类型注入，那么对应BookDao接口如果有多个实现类，比如添加BookDaoImpl2

```

1 @Repository
2 public class BookDaoImpl2 implements BookDao {
3     public void save() {
4         System.out.println("book dao save ...2");
5     }
6 }

```

这个时候再次运行App, 就会报错

此时, 按照类型注入就无法区分到底注入哪个对象, 解决方案: 按照名称注入

- 先给两个Dao类分别起个名称

```

1 @Repository("bookDao")
2 public class BookDaoImpl implements BookDao {
3     public void save() {
4         System.out.println("book dao save ..." );
5     }
6 }
7 @Repository("bookDao2")
8 public class BookDaoImpl2 implements BookDao {
9     public void save() {
10        System.out.println("book dao save ...2" );
11    }
12 }

```

此时就可以注入成功, 但是得思考个问题:

- @Autowired是按照类型注入的, 给BookDao的两个实现起了名称, 它还是有两个bean对象, 为什么不报错?
- @Autowired默认按照类型自动装配, 如果IOC容器中同类的Bean找到多个, 就按照变量名和Bean的名称匹配。因为变量名叫bookDao而容器中也有一个bookDao, 所以可以成功注入。
- 分析下面这种情况是否能完成注入呢?

```

@Service
public class BookServiceImpl implements BookService {

    @Autowired
    private BookDao bookDao;
}

```

```

@Repository("bookDao1")
public class BookDaoImpl implements BookDao {
    public void save() {
        System.out.println("book dao save ...");
    }
}

```

```

@Repository("bookDao2")
public class BookDaoImpl2 implements BookDao {
    public void save() {
        System.out.println("book dao save ...2");
    }
}

```

- 。不行，因为按照类型会找到多个bean对象，此时会按照bookDao名称去找，因为IOC容器只有名称叫bookDao1和bookDao2，所以找不到，会报NoUniqueBeanDefinitionException

3.4.3 注解实现按照名称注入

当根据类型在容器中找到多个bean，注入参数的属性名又和容器中bean的名称不一致，这个时候该如何解决，就需要使用到@Qualifier来指定注入哪个名称的bean对象。

```
1 @Service
2 public class BookServiceImpl implements BookService {
3     @Autowired
4     @Qualifier("bookDao1")
5     private BookDao bookDao;
6
7     public void save() {
8         System.out.println("book service save ...");
9         bookDao.save();
10    }
11 }
```

@Qualifier注解后的值就是需要注入的bean的名称。

注意:@Qualifier不能独立使用，必须和@Autowired一起使用

3.4.4 简单数据类型注入

引用类型看完，简单类型注入就比较容易懂了。简单类型注入的是基本数据类型或者字符串类型，下面在BookDaoImpl类中添加一个name属性，用其进行简单类型注入

```
1 @Repository("bookDao")
2 public class BookDaoImpl implements BookDao {
3     private String name;
4     public void save() {
5         System.out.println("book dao save ..." + name);
6     }
7 }
```

数据类型换了，对应的注解也要跟着换，这次使用@Value注解，将值写入注解的参数中就行了

```
1 @Repository("bookDao")
2 public class BookDaoImpl implements BookDao {
3     @Value("itheima")
4     private String name;
5     public void save() {
6         System.out.println("book dao save ..." + name);
7     }
8 }
```

注意数据格式要匹配，如将"abc"注入给int值，这样程序就会报错。

介绍完后，会有一种感觉就是这个注解好像没什么用，跟直接赋值是一个效果，还没有直接赋值简单，所以这个注解存在的意义是什么？

3.4.5 注解读取properties配置文件

`@value`一般会被用在从`properties`配置文件中读取内容进行使用，具体如何实现？

步骤1: resource下准备properties文件

`jdbc.properties`

```
1 name=itheima888
```

步骤2: 使用注解加载properties配置文件

在配置类上添加`@PropertySource`注解

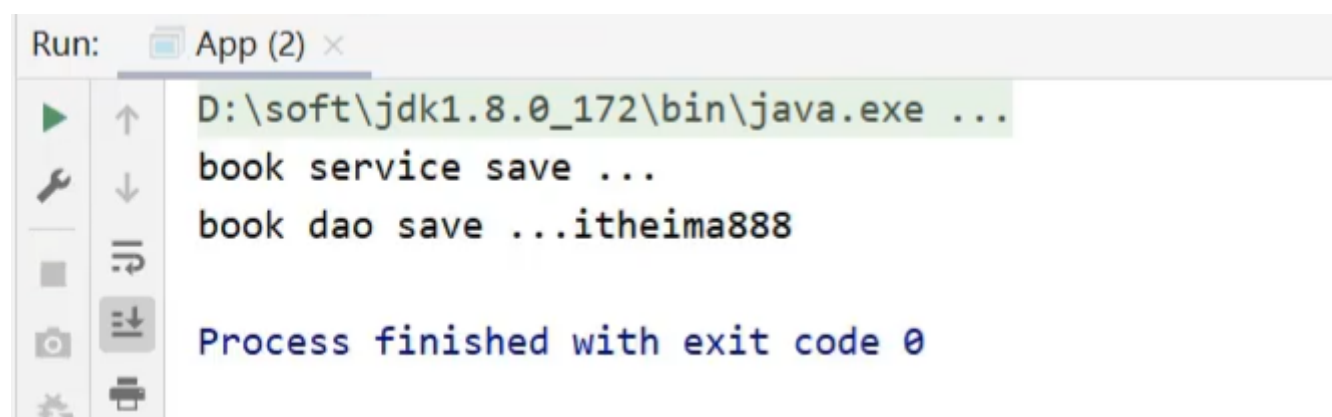
```
1 @Configuration
2 @ComponentScan("com.itheima")
3 @PropertySource("jdbc.properties")
4 public class SpringConfig {
5 }
6
```

步骤3: 使用`@Value`读取配置文件中的内容

```
1 @Repository("bookDao")
2 public class BookDaoImpl implements BookDao {
3     @Value("${name}")
4     private String name;
5     public void save() {
6         System.out.println("book dao save ..." + name);
7     }
8 }
```

步骤4: 运行程序

运行`App`类，查看运行结果，说明配置文件中的内容已经被加载到



```
Run: App (2) x
D:\soft\jdk1.8.0_172\bin\java.exe ...
book service save ...
book dao save ...itheima888
Process finished with exit code 0
```

注意:

- 如果读取的`properties`配置文件有多个，可以使用`@PropertySource`的属性来指定多个

```
1 @PropertySource({"jdbc.properties", "xxx.properties"})
```

- @PropertySource 注解属性中不支持使用通配符 *，运行会报错

```
1 @PropertySource({"*.properties"})
```

- @PropertySource 注解属性中可以把 classpath: 加上，代表从当前项目的根路径找文件

```
1 @PropertySource({"classpath:jdbc.properties"})
```

知识点1: @Autowired

名称	@Autowired
类型	属性注解 或 方法注解 (了解) 或 方法形参注解 (了解)
位置	属性定义上方 或 标准set方法上方 或 类set方法上方 或 方法形参前面
作用	为引用类型属性设置值
属性	required: true/false, 定义该属性是否允许为null

知识点2: @Qualifier

名称	@Qualifier
类型	属性注解 或 方法注解 (了解)
位置	属性定义上方 或 标准set方法上方 或 类set方法上方
作用	为引用类型属性指定注入的beanId
属性	value (默认) : 设置注入的beanId

知识点3: @Value

名称	@Value
类型	属性注解 或 方法注解 (了解)
位置	属性定义上方 或 标准set方法上方 或 类set方法上方
作用	为 基本数据类型 或 字符串类型 属性设置值
属性	value (默认) : 要注入的属性值

知识点4: @PropertySource

名称	@PropertySource
类型	类注解
位置	类定义上方
作用	加载properties文件中的属性值
属性	value (默认) : 设置加载的properties文件对应的文件名或文件名组成的数组

4, IOC/DI注解开发管理第三方bean

前面定义bean的时候都是在自己开发的类上面写个注解就完成了, 但如果是第三方的类, 这些类都是在jar包中, 我们没有办法在类上面添加注解, 这个时候该怎么办?

遇到上述问题, 我们就需要有一种更加灵活的方式来定义bean, 这种方式不能在原始代码上面书写注解, 一样能定义bean, 这就用到了一个全新的注解@Bean。

这个注解该如何使用呢?

咱们把之前使用配置方式管理的数据源使用注解再来一遍, 通过这个案例来学习下@Bean的使用。

4.1 环境准备

学习@Bean注解之前先来准备环境:

- 创建一个Maven项目
- pom.xml添加Spring的依赖

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework</groupId>
4         <artifactId>spring-context</artifactId>
5         <version>5.2.10.RELEASE</version>
6     </dependency>
7 </dependencies>
```

- 添加一个配置类 SpringConfig

```
1 @Configuration
2 public class SpringConfig {
3 }
```

- 添加BookDao、BookDaoImpl类

```

1 public interface BookDao {
2     public void save();
3 }
4 @Repository
5 public class BookDaoImpl implements BookDao {
6     public void save() {
7         System.out.println("book dao save ..." );
8     }
9 }

```

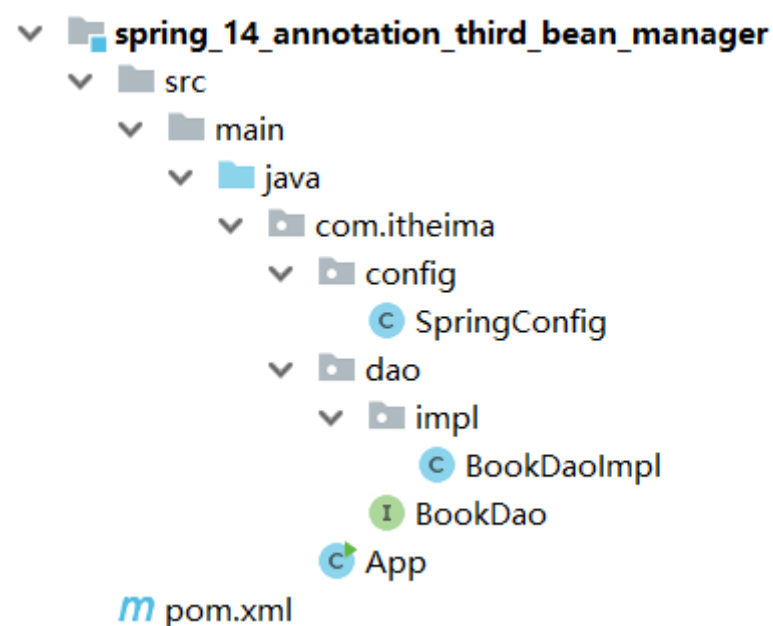
- 创建运行类App

```

1 public class App {
2     public static void main(String[] args) {
3         AnnotationConfigApplicationContext ctx = new
4         AnnotationConfigApplicationContext(SpringConfig.class);
5     }
6 }

```

最终创建好的项目结构如下：



4.2 注解开发管理第三方bean

在上述环境中完成对Druid数据源的管理，具体的实现步骤为：

步骤1：导入对应的jar包

```

1 <dependency>
2     <groupId>com.alibaba</groupId>
3     <artifactId>druid</artifactId>
4     <version>1.1.16</version>
5 </dependency>

```

步骤2：在配置类中添加一个方法

注意该方法的返回值就是要创建的Bean对象类型

```

1 @Configuration
2 public class SpringConfig {
3     public DataSource dataSource(){
4         DruidDataSource ds = new DruidDataSource();
5         ds.setDriverClassName("com.mysql.jdbc.Driver");
6         ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
7         ds.setUsername("root");
8         ds.setPassword("root");
9         return ds;
10    }
11 }

```

步骤3: 在方法上添加@Bean注解

@Bean注解的作用是将方法的返回值制作为Spring管理的一个bean对象

```

1 @Configuration
2 public class SpringConfig {
3     @Bean
4     public DataSource dataSource(){
5         DruidDataSource ds = new DruidDataSource();
6         ds.setDriverClassName("com.mysql.jdbc.Driver");
7         ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
8         ds.setUsername("root");
9         ds.setPassword("root");
10    return ds;
11    }
12 }

```

注意: 不能使用 `DataSource ds = new DruidDataSource()`

因为DataSource接口中没有对应的setter方法来设置属性。

步骤4: 从IOC容器中获取对象并打印

```

1 public class App {
2     public static void main(String[] args) {
3         AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
4         DataSource dataSource = ctx.getBean(DataSource.class);
5         System.out.println(dataSource);
6     }
7 }

```

至此使用@Bean来管理第三方bean的案例就已经完成。

如果有多个bean要被Spring管理，直接在配置类中多些几个方法，方法上添加@Bean注解即可。

4.3 引入外部配置类

如果把所有的第三方bean都配置到Spring的配置类SpringConfig中，虽然可以，但是不利于代码阅读和分类管理，所有我们就想能不能按照类别将这些bean配置到不同的配置类中？

对于数据源的bean，我们新建一个JdbcConfig配置类，并把数据源配置到该类下。

```
1 public class JdbcConfig {
2     @Bean
3     public DataSource dataSource(){
4         DruidDataSource ds = new DruidDataSource();
5         ds.setDriverClassName("com.mysql.jdbc.Driver");
6         ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
7         ds.setUsername("root");
8         ds.setPassword("root");
9         return ds;
10    }
11 }
```

现在的问题是，这个配置类如何能被Spring配置类加载到，并创建DataSource对象在IOC容器中？

针对这个问题，有两个解决方案：

4.3.1 使用包扫描引入

步骤1：在Spring的配置类上添加包扫描

```
1 @Configuration
2 @ComponentScan("com.itheima.config")
3 public class SpringConfig {
4
5 }
```

步骤2：在JdbcConfig上添加配置注解

JdbcConfig类要放入到com.itheima.config包下，需要被Spring的配置类扫描到即可

```
1 @Configuration
2 public class JdbcConfig {
3     @Bean
4     public DataSource dataSource(){
5         DruidDataSource ds = new DruidDataSource();
6         ds.setDriverClassName("com.mysql.jdbc.Driver");
7         ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
8         ds.setUsername("root");
9         ds.setPassword("root");
10        return ds;
11    }
12 }
```

步骤3: 运行程序

依然能获取到bean对象并打印控制台。

这种方式虽然能够扫描到，但是不能很快的知晓都引入了哪些配置类，所有这种方式不推荐使用。

4.3.2 使用@Import引入

方案一实现起来有点小复杂，Spring早就想到了这一点，于是又给我们提供了第二种方案。

这种方案可以不用加@Configuration注解，但是必须在Spring配置类上使用@Import注解手动引入需要加载的配置类

步骤1: 去除JdbcConfig类上的注解

```
1 public class JdbcConfig {
2     @Bean
3     public DataSource dataSource(){
4         DruidDataSource ds = new DruidDataSource();
5         ds.setDriverClassName("com.mysql.jdbc.Driver");
6         ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
7         ds.setUsername("root");
8         ds.setPassword("root");
9         return ds;
10    }
11 }
```

步骤2: 在Spring配置类中引入

```
1 @Configuration
2 //@ComponentScan("com.itheima.config")
3 @Import({JdbcConfig.class})
4 public class SpringConfig {
5
6 }
```

注意:

- 扫描注解可以移除
- @Import参数需要的是一个数组，可以引入多个配置类。
- @Import注解在配置类中只能写一次，下面的方式是**不允许的**

```

1 @Configuration
2 //@ComponentScan("com.itheima.config")
3 @Import(JdbcConfig.class)
4 @Import(Xxx.class)
5 public class SpringConfig {
6
7 }

```

步骤3: 运行程序

依然能获取到bean对象并打印控制台

知识点1: @Bean

名称	@Bean
类型	方法注解
位置	方法定义上方
作用	设置该方法的返回值作为spring管理的bean
属性	value (默认) : 定义bean的id

知识点2: @Import

名称	@Import
类型	类注解
位置	类定义上方
作用	导入配置类
属性	value (默认) : 定义导入的配置类类名, 当配置类有多个时使用数组格式一次性导入多个配置类

4.4 注解开发实现为第三方bean注入资源

在使用@Bean创建bean对象的时候, 如果方法在创建的过程中需要其他资源该怎么办?

这些资源会有两大类, 分别是简单数据类型 和 引用数据类型。

4.4.1 简单数据类型

4.4.1.1 需求分析

对于下面代码关于数据库的四要素不应该写死在代码中, 应该是从properties配置文件中读取。如何来优化下面的代码?

```

1 public class JdbcConfig {
2     @Bean
3     public DataSource dataSource(){
4         DruidDataSource ds = new DruidDataSource();
5         ds.setDriverClassName("com.mysql.jdbc.Driver");
6         ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
7         ds.setUsername("root");
8         ds.setPassword("root");
9         return ds;
10    }
11 }

```

4.4.1.2 注入简单数据类型步骤

步骤1: 类中提供四个属性

```

1 public class JdbcConfig {
2     private String driver;
3     private String url;
4     private String userName;
5     private String password;
6     @Bean
7     public DataSource dataSource(){
8         DruidDataSource ds = new DruidDataSource();
9         ds.setDriverClassName("com.mysql.jdbc.Driver");
10        ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
11        ds.setUsername("root");
12        ds.setPassword("root");
13        return ds;
14    }
15 }

```

步骤2: 使用@value注解引入值

```

1 public class JdbcConfig {
2     @value("com.mysql.jdbc.Driver")
3     private String driver;
4     @value("jdbc:mysql://localhost:3306/spring_db")
5     private String url;
6     @value("root")
7     private String userName;
8     @value("password")
9     private String password;
10    @Bean
11    public DataSource dataSource(){
12        DruidDataSource ds = new DruidDataSource();
13        ds.setDriverClassName(driver);
14        ds.setUrl(url);

```

```
15     ds.setUsername(userName);
16     ds.setPassword(password);
17     return ds;
18 }
19 }
```

扩展

现在的数据库连接四要素还是写在代码中，需要做的是将这些内容提

取到jdbc.properties配置文件，大家思考下该如何实现？

1. resources目录下添加jdbc.properties
2. 配置文件中提供四个键值对分别是数据库的四要素
3. 使用@PropertySource加载jdbc.properties配置文件
4. 修改@Value注解属性的值，将其修改为\${key}，key就是键值对中的键的值

具体的实现就交由大家自行实现下。

4.4.2 引用数据类型

4.4.2.1 需求分析

假设在构建DataSource对象的时候，需要用到BookDao对象，该如何把BookDao对象注入进方法内让其使用呢？

```
1 public class JdbcConfig {
2     @Bean
3     public DataSource dataSource(){
4         DruidDataSource ds = new DruidDataSource();
5         ds.setDriverClassName("com.mysql.jdbc.Driver");
6         ds.setUrl("jdbc:mysql://localhost:3306/spring_db");
7         ds.setUsername("root");
8         ds.setPassword("root");
9         return ds;
10    }
11 }
```

4.4.2.2 注入引用数据类型步骤

步骤1: 在SpringConfig中扫描BookDao

扫描的目的是让Spring能管理到BookDao，也就是说要让IOC容器中有一个bookDao对象

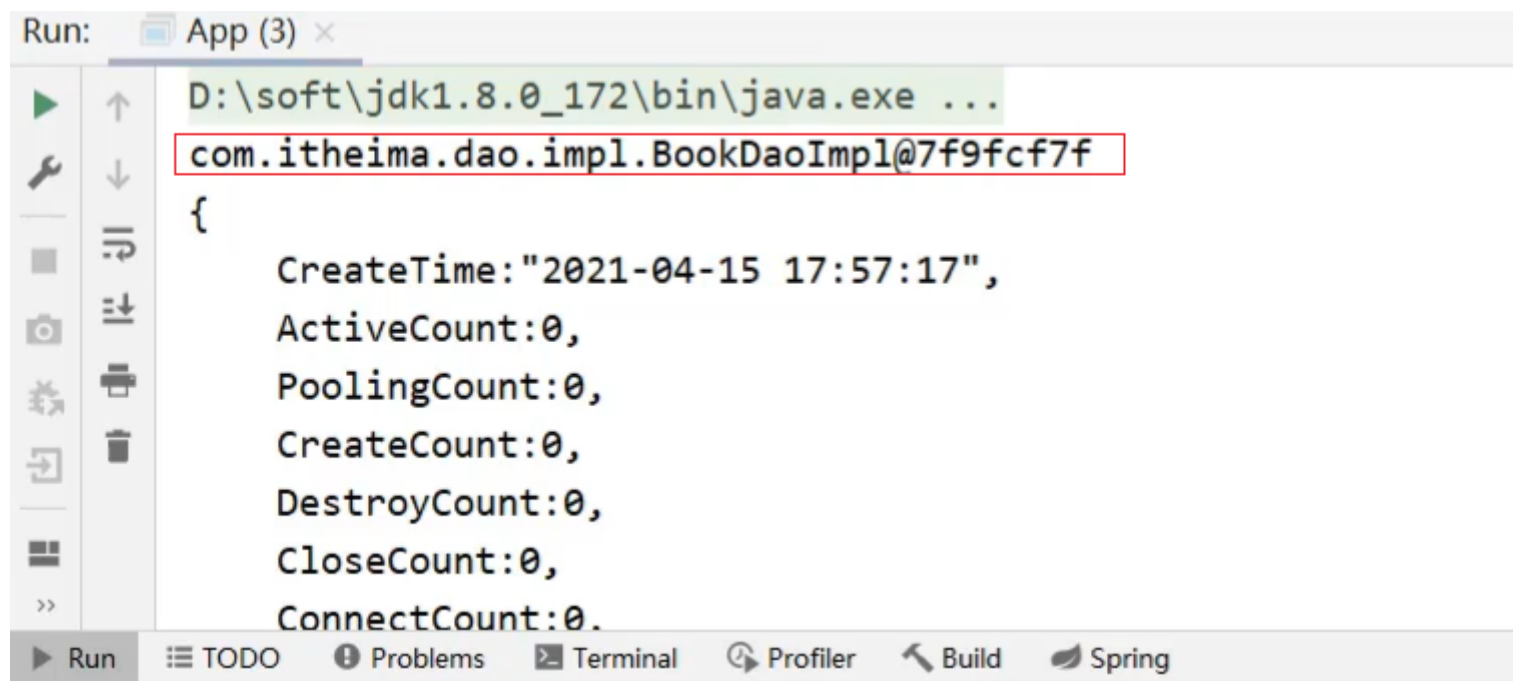
```
1 @Configuration
2 @ComponentScan("com.itheima.dao")
3 @Import({JdbcConfig.class})
4 public class SpringConfig {
5 }
```

步骤2: 在JdbcConfig类的方法上添加参数

```
1 @Bean
2 public DataSource dataSource(BookDao bookDao){
3     System.out.println(bookDao);
4     DruidDataSource ds = new DruidDataSource();
5     ds.setDriverClassName(driver);
6     ds.setUrl(url);
7     ds.setUsername(userName);
8     ds.setPassword(password);
9     return ds;
10 }
```

引用类型注入只需要为bean定义方法设置形参即可，容器会根据类型自动装配对象。

步骤3: 运行程序



```
Run: App (3) x
D:\soft\jdk1.8.0_172\bin\java.exe ...
com.itheima.dao.impl.BookDaoImpl@7f9fcf7f
{
    CreateTime:"2021-04-15 17:57:17",
    ActiveCount:0,
    PoolingCount:0,
    CreateCount:0,
    DestroyCount:0,
    CloseCount:0,
    ConnectCount:0.
```

5, 注解开发总结

前面我们已经完成了XML配置和注解的开发实现，至于两者之间的差异，咱们放在一块去对比回顾下：

功能	XML配置	注解
定义bean	bean标签 <ul style="list-style-type: none"> ● id属性 ● class属性 	@Component <ul style="list-style-type: none"> ● @Controller ● @Service ● @Repository @ComponentScan
设置依赖注入	setter注入(set方法) <ul style="list-style-type: none"> ● 引用/简单 构造器注入(构造方法) <ul style="list-style-type: none"> ● 引用/简单 自动装配	@Autowired <ul style="list-style-type: none"> ● @Qualifier @Value
配置第三方bean	bean标签 静态工厂、实例工厂、FactoryBean	@Bean
作用范围	<ul style="list-style-type: none"> ● scope属性 	@Scope
生命周期	标准接口 <ul style="list-style-type: none"> ● init-method ● destroy-method 	@PostConstructor @PreDestroy

6, Spring整合

课程学习到这里，已经对Spring有一个简单的认识了，Spring有一个容器，叫做IoC容器，里面保存bean。在进行企业级开发的时候，其实除了将自己写的类让Spring管理之外，还有一部分重要的工作就是使用第三方的技术。前面已经讲了如何管理第三方bean了，下面结合IoC和DI，整合2个常用技术，进一步加深对Spring的使用理解。

6.1 Spring整合Mybatis思路分析

6.1.1 环境准备

在准备环境的过程中，我们也来回顾下Mybatis开发的相关内容：

步骤1：准备数据库表

Mybatis是用来操作数据库表，所以先创建一个数据库及表

```

1 create database spring_db character set utf8;
2 use spring_db;
3 create table tbl_account(
4     id int primary key auto_increment,
5     name varchar(35),
6     money double
7 );

```

步骤2：创建项目导入jar包

项目的pom.xml添加相关依赖

```

1 <dependencies>
2     <dependency>
3         <groupId>org.springframework</groupId>
4         <artifactId>spring-context</artifactId>

```

```

5     <version>5.2.10.RELEASE</version>
6 </dependency>
7 <dependency>
8     <groupId>com.alibaba</groupId>
9     <artifactId>druid</artifactId>
10    <version>1.1.16</version>
11 </dependency>
12 <dependency>
13     <groupId>org.mybatis</groupId>
14     <artifactId>mybatis</artifactId>
15     <version>3.5.6</version>
16 </dependency>
17 <dependency>
18     <groupId>mysql</groupId>
19     <artifactId>mysql-connector-java</artifactId>
20     <version>5.1.47</version>
21 </dependency>
22 </dependencies>

```

步骤3: 根据表创建模型类

```

1 public class Account implements Serializable {
2
3     private Integer id;
4     private String name;
5     private Double money;
6     //setter...getter...toString...方法略
7 }

```

步骤4: 创建Dao接口

```

1 public interface AccountDao {
2
3     @Insert("insert into tbl_account(name,money)values(#{name},#{money})")
4     void save(Account account);
5
6     @Delete("delete from tbl_account where id = #{id} ")
7     void delete(Integer id);
8
9     @Update("update tbl_account set name = #{name} , money = #{money} where
10 id = #{id} ")
11     void update(Account account);
12
13     @Select("select * from tbl_account")
14     List<Account> findAll();
15
16     @Select("select * from tbl_account where id = #{id} ")
17     Account findById(Integer id);

```



```
17 }
```

步骤5: 创建Service接口和实现类

```
1 public interface AccountService {
2
3     void save(Account account);
4
5     void delete(Integer id);
6
7     void update(Account account);
8
9     List<Account> findAll();
10
11     Account findById(Integer id);
12
13 }
14
15 @Service
16 public class AccountServiceImpl implements AccountService {
17
18     @Autowired
19     private AccountDao accountDao;
20
21     public void save(Account account) {
22         accountDao.save(account);
23     }
24
25     public void update(Account account){
26         accountDao.update(account);
27     }
28
29     public void delete(Integer id) {
30         accountDao.delete(id);
31     }
32
33     public Account findById(Integer id) {
34         return accountDao.findById(id);
35     }
36
37     public List<Account> findAll() {
38         return accountDao.findAll();
39     }
40 }
```

步骤6: 添加jdbc.properties文件

resources目录下添加, 用于配置数据库连接四要素

```
1 jdbc.driver=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/spring_db?useSSL=false
3 jdbc.username=root
4 jdbc.password=root
```

useSSL:关闭MySQL的SSL连接

步骤7:添加Mybatis核心配置文件

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5 <configuration>
6     <!--读取外部properties配置文件-->
7     <properties resource="jdbc.properties"></properties>
8     <!--别名扫描的包路径-->
9     <typeAliases>
10        <package name="com.itheima.domain"/>
11    </typeAliases>
12    <!--数据源-->
13    <environments default="mysql">
14        <environment id="mysql">
15            <transactionManager type="JDBC"></transactionManager>
16            <dataSource type="POOLED">
17                <property name="driver" value="${jdbc.driver}"></property>
18                <property name="url" value="${jdbc.url}"></property>
19                <property name="username" value="${jdbc.username}">
20                    </property>
21                <property name="password" value="${jdbc.password}">
22                    </property>
23            </dataSource>
24        </environment>
25    </environments>
26    <!--映射文件扫描包路径-->
27    <mappers>
28        <package name="com.itheima.dao"></package>
29    </mappers>
30 </configuration>
```

步骤8:编写应用程序

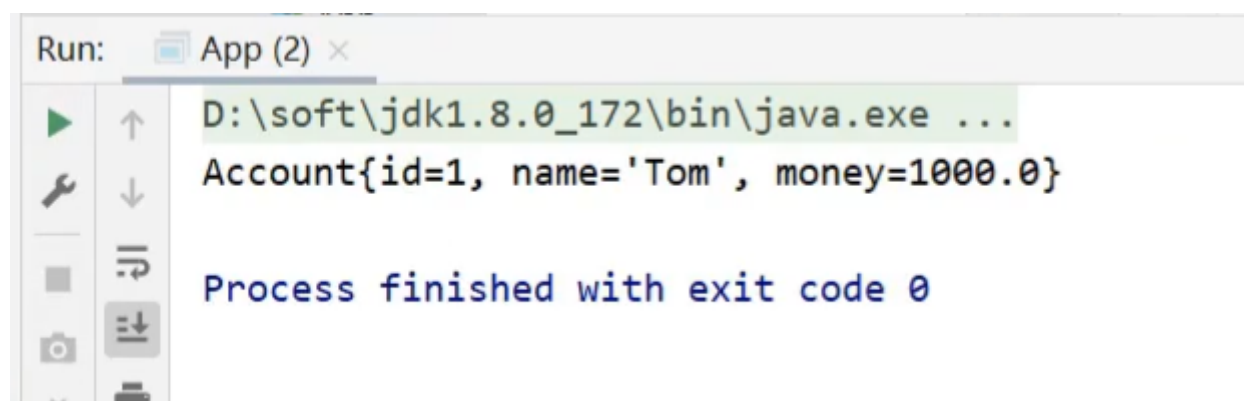
```
1 public class App {
2     public static void main(String[] args) throws IOException {
3         // 1. 创建SqlSessionFactoryBuilder对象
4         SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
5         SqlSessionFactoryBuilder();
6         // 2. 加载SqlMapConfig.xml配置文件
```

```

6      InputStream inputStream =
Resources.getResourceAsStream("SqlMapConfig.xml.bak");
7      // 3. 创建SqlSessionFactory对象
8      SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(inputStream);
9      // 4. 获取SqlSession
10     SqlSession sqlSession = sqlSessionFactory.openSession();
11     // 5. 执行SqlSession对象执行查询, 获取结果User
12     AccountDao accountDao = sqlSession.getMapper(AccountDao.class);
13
14     Account ac = accountDao.findById(1);
15     System.out.println(ac);
16
17     // 6. 释放资源
18     sqlSession.close();
19 }
20 }

```

步骤9: 运行程序



6.1.2 整合思路分析

Mybatis的基础环境我们已经准备好了, 接下来就得分析下在上述的内容中, 哪些对象可以交给Spring来管理?

- Mybatis程序核心对象分析

<code>// 1. 创建SqlSessionFactoryBuilder对象</code>	初始化 SqlSessionFactory
<code>SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();</code>	
<code>// 2. 加载SqlMapConfig.xml配置文件</code>	
<code>InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");</code>	
<code>// 3. 创建SqlSessionFactory对象</code>	
SqlSessionFactory <code>sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);</code>	
<code>// 4. 获取SqlSession</code>	获取连接, 获取实现
<code>SqlSession sqlSession = sqlSessionFactory.openSession();</code>	
<code>// 5. 执行SqlSession对象执行查询, 获取结果User</code>	
<code>AccountDao accountDao = sqlSession.getMapper(AccountDao.class);</code>	
<code>Account ac = accountDao.findById(2);</code>	获取数据层接口
<code>System.out.println(ac);</code>	
<code>// 6. 释放资源</code>	关闭连接
<code>sqlSession.close();</code>	

从图中可以获取到, 真正需要交给Spring管理的是**SqlSessionFactory**

- 整合Mybatis, 就是将Mybatis用到的内容交给Spring管理, 分析下配置文件

```

<configuration>
  <properties resource="jdbc.properties"></properties>
  <typeAliases>
    <package name="com.itheima.domain"/>
  </typeAliases>
  <environments default="mysql">
    <environment id="mysql">
      <transactionManager type="JDBC"></transactionManager>
      <dataSource type="POOLED">
        <property name="driver" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <package name="com.itheima.dao"></package>
  </mappers>
</configuration>

```

初始化属性数据

初始化类型别名

初始化dataSource

初始化映射配置

说明:

- 第一行读取外部properties配置文件, Spring有提供具体的解决方案@PropertySource, 需要交给Spring
- 第二行起别名包扫描, 为SqlSessionFactory服务的, 需要交给Spring
- 第三行主要用于做连接池, Spring之前我们已经整合了Druid连接池, 这块也需要交给Spring
- 前面三行一起都是为了创建SqlSession对象用的, 那么用Spring管理SqlSession对象吗? 回忆下SqlSession是由SqlSessionFactory创建出来的, 所以只需要将SqlSessionFactory交给Spring管理即可。
- 第四行是Mapper接口和映射文件[如果使用注解就没有该映射文件], 这个是在获取到SqlSession以后执行具体操作的时候用, 所以它和SqlSessionFactory创建的时机都不在同一个时间, 可能需要单独管理。

6.2 Spring整合Mybatis

前面我们已经分析了Spring与Mybatis的整合, 大体需要做两件事,

第一件事是: Spring要管理MyBatis中的SqlSessionFactory

第二件事是: Spring要管理Mapper接口的扫描

具体该如何实现, 具体的步骤为:

步骤1: 项目中导入整合需要的jar包

```

1 <dependency>
2   <!--Spring操作数据库需要该jar包-->
3   <groupId>org.springframework</groupId>
4   <artifactId>spring-jdbc</artifactId>
5   <version>5.2.10.RELEASE</version>

```

```

6 </dependency>
7 <dependency>
8     <!--
9         Spring与Mybatis整合的jar包
10        这个jar包mybatis在前面, 是Mybatis提供的
11    -->
12    <groupId>org.mybatis</groupId>
13    <artifactId>mybatis-spring</artifactId>
14    <version>1.3.0</version>
15 </dependency>

```

步骤2: 创建Spring的主配置类

```

1 //配置类注解
2 @Configuration
3 //包扫描, 主要扫描的是项目中的AccountServiceImpl类
4 @ComponentScan("com.itheima")
5 public class SpringConfig {
6 }
7

```

步骤3: 创建数据源的配置类

在配置类中完成数据源的创建

```

1 public class JdbcConfig {
2     @Value("${jdbc.driver}")
3     private String driver;
4     @Value("${jdbc.url}")
5     private String url;
6     @Value("${jdbc.username}")
7     private String userName;
8     @Value("${jdbc.password}")
9     private String password;
10
11     @Bean
12     public DataSource dataSource(){
13         DruidDataSource ds = new DruidDataSource();
14         ds.setDriverClassName(driver);
15         ds.setUrl(url);
16         ds.setUsername(userName);
17         ds.setPassword(password);
18         return ds;
19     }
20 }

```

步骤4: 主配置类中读properties并引入数据源配置类

```

1 @Configuration
2 @ComponentScan("com.itheima")
3 @PropertySource("classpath:jdbc.properties")
4 @Import(JdbcConfig.class)
5 public class SpringConfig {
6 }
7

```

步骤5: 创建Mybatis配置类并配置SqlSessionFactory

```

1 public class MybatisConfig {
2     //定义bean, SqlSessionFactoryBean, 用于产生SqlSessionFactory对象
3     @Bean
4     public SqlSessionFactoryBean sqlSessionFactory(DataSource dataSource){
5         SqlSessionFactoryBean ssfb = new SqlSessionFactoryBean();
6         //设置模型类的别名扫描
7         ssfb.setTypeAliasesPackage("com.itheima.domain");
8         //设置数据源
9         ssfb.setDataSource(dataSource);
10        return ssfb;
11    }
12    //定义bean, 返回MapperScannerConfigurer对象
13    @Bean
14    public MapperScannerConfigurer mapperScannerConfigurer(){
15        MapperScannerConfigurer msc = new MapperScannerConfigurer();
16        msc.setBasePackage("com.itheima.dao");
17        return msc;
18    }
19 }

```

说明:

- 使用SqlSessionFactoryBean封装SqlSessionFactory需要的环境信息

```

<configuration>
  <properties resource="jdbc.properties"></properties>
  <typeAliases>
    <package name="com.itheima.domain"/>
  </typeAliases>
  <environments default="mysql">
    <environment id="mysql">
      <transactionManager type="JDBC"></transactionManager>
      <dataSource type="POOLED">
        <property name="driver" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <package name="com.itheima.dao"></package>
  </mappers>
</configuration>

```

```

@Bean
public SqlSessionFactoryBean sqlSessionFactory(DataSource
dataSource){
    SqlSessionFactoryBean ssfb = new SqlSessionFactoryBean();
    ssfb.setTypeAliasesPackage("com.itheima.domain");
    ssfb.setDataSource(dataSource);
    return ssfb;
}

```

- `SqlSessionFactoryBean`是前面我们讲解`FactoryBean`的一个子类，在该类中将`SqlSessionFactory`的创建进行了封装，简化对象的创建，我们只需要将其需要的内容设置即可。
- 方法中有一个参数为`dataSource`，当前Spring容器中已经创建了Druid数据源，类型刚好是`DataSource`类型，此时在初始化`SqlSessionFactoryBean`这个对象的时候，发现需要使用`DataSource`对象，而容器中刚好有这么一个对象，就自动加载了`DruidDataSource`对象。
- 使用`MapperScannerConfigurer`加载Dao接口，创建代理对象保存到IOC容器中

```

<configuration>
  <properties resource="jdbc.properties"></properties>
  <typeAliases>
    <package name="com.itheima.domain"/>
  </typeAliases>
  <environments default="mysql">
    <environment id="mysql">
      <transactionManager type="J
      <dataSource type="POOLED">
        <property name="driver"
        <property name="url" va
        <property name="usernam
        <property name="passwor
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <package name="com.itheima.dao"></package>
  </mappers>
</configuration>

```

```

@Bean
public MapperScannerConfigurer mapperScannerConfigurer(){
    MapperScannerConfigurer msc = new
    MapperScannerConfigurer();
    msc.setBasePackage("com.itheima.dao");
    return msc;
}

```

- 这个`MapperScannerConfigurer`对象也是MyBatis提供的专用于整合的jar包中的类，用来处理原始配置文件中的`mappers`相关配置，加载数据层的`Mapper`接口类
- `MapperScannerConfigurer`有一个核心属性`basePackage`，就是用来设置所扫描的包路径

步骤6：主配置类中引入Mybatis配置类

```

1 @Configuration
2 @ComponentScan("com.itheima")
3 @PropertySource("classpath:jdbc.properties")
4 @Import({JdbcConfig.class,MybatisConfig.class})
5 public class SpringConfig {
6 }

```

步骤7：编写运行类

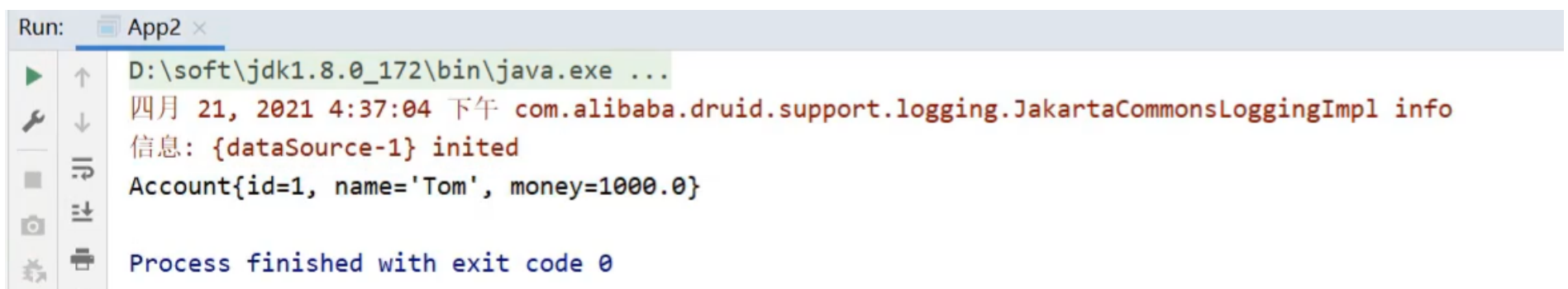
在运行类中，从IOC容器中获取`Service`对象，调用方法获取结果

```

1 public class App2 {
2     public static void main(String[] args) {
3         ApplicationContext ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
4
5         AccountService accountService = ctx.getBean(AccountService.class);
6
7         Account ac = accountService.findById(1);
8         System.out.println(ac);
9     }
10 }
11

```

步骤8:运行程序



```

Run: App2 x
D:\soft\jdk1.8.0_172\bin\java.exe ...
四月 21, 2021 4:37:04 下午 com.alibaba.druid.support.logging.JakartaCommonsLoggingImpl info
信息: {dataSource-1} inited
Account{id=1, name='Tom', money=1000.0}
Process finished with exit code 0

```

支持Spring与Mybatis的整合就已经完成了，其中主要用到的两个类分别是：

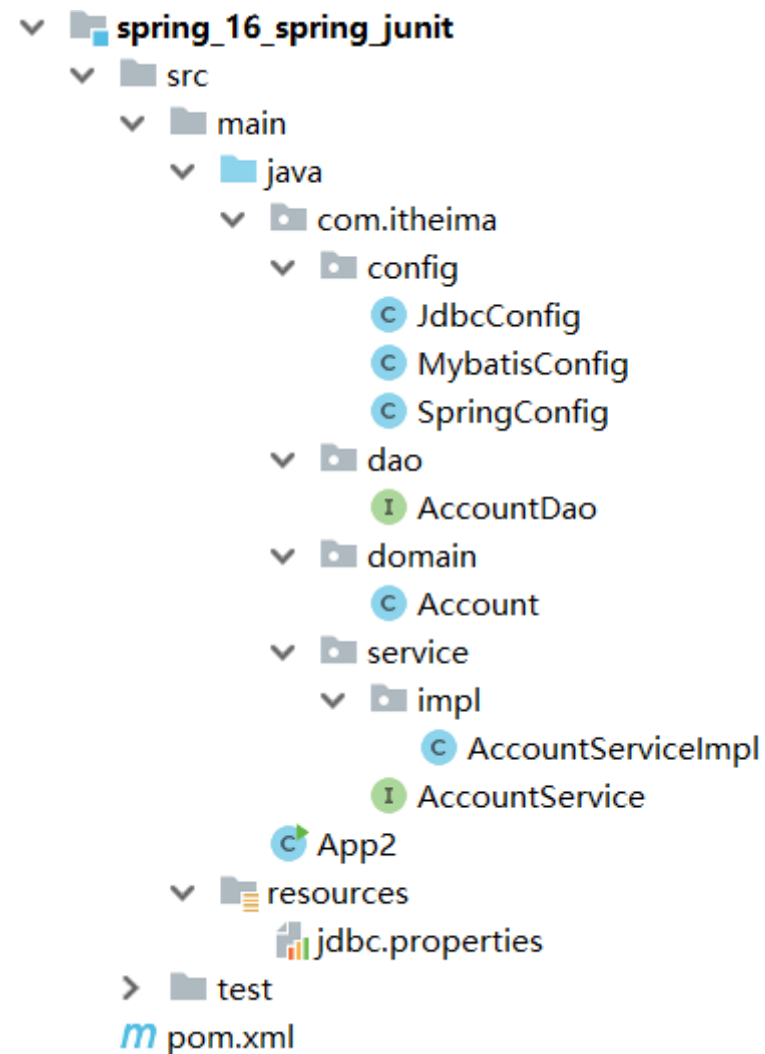
- **SqlSessionFactoryBean**
- **MapperScannerConfigurer**

6.3 Spring整合JUnit

整合JUnit与整合Druid和MyBatis差异比较大，为什么呢？JUnit是一个搞单元测试用的工具，它不是我们程序的主体，也不会参加最终程序的运行，从作用上来说就和之前的东西不一样，它不是做功能的，看做是一个辅助工具就可以了。

6.3.1 环境准备

这块环境，大家可以直接使用Spring与Mybatis整合的环境即可。当然也可以重新创建一个，因为内容是一模一样，所以我们直接来看下项目结构即可：



6.3.2 整合Junit步骤

在上述环境的基础上，我们来对Junit进行整合。

步骤1：引入依赖

pom.xml

```
1 <dependency>
2     <groupId>junit</groupId>
3     <artifactId>junit</artifactId>
4     <version>4.12</version>
5     <scope>test</scope>
6 </dependency>
7
8 <dependency>
9     <groupId>org.springframework</groupId>
10    <artifactId>spring-test</artifactId>
11    <version>5.2.10.RELEASE</version>
12 </dependency>
```

步骤2：编写测试类

在test\java下创建一个AccountServiceTest,这个名字任意

```
1 //设置类运行器
2 @RunWith(SpringJUnit4ClassRunner.class)
3 //设置Spring环境对应的配置类
4 @ContextConfiguration(classes = {SpringConfiguration.class}) //加载配置类
```

```

5 // @ContextConfiguration(locations={"classpath:applicationContext.xml"}) // 加载
  配置文件
6 public class AccountServiceTest {
7     // 支持自动装配注入bean
8     @Autowired
9     private AccountService accountService;
10    @Test
11    public void testFindById(){
12        System.out.println(accountService.findById(1));
13    }
14    }
15    @Test
16    public void testFindAll(){
17        System.out.println(accountService.findAll());
18    }
19 }

```

注意：

- 单元测试，如果测试的是注解配置类，则使用 `@ContextConfiguration(classes = 配置类.class)`
- 单元测试，如果测试的是配置文件，则使用 `@ContextConfiguration(locations={配置文件名,...})`
- JUnit运行后是基于Spring环境运行的，所以Spring提供了一个专用的类运行器，这个务必要设置，这个类运行器就在Spring的测试专用包中提供的，导入的坐标就是这个东西 `SpringJUnit4ClassRunner`
- 上面两个配置都是固定格式，当需要测试哪个bean时，使用自动装配加载对应的对象，下面的工作就和以前做JUnit单元测试完全一样了

知识点1：@RunWith

名称	@RunWith
类型	测试类注解
位置	测试类定义上方
作用	设置JUnit运行器
属性	value (默认)：运行所使用的运行期

知识点2：@ContextConfiguration

名称	@ContextConfiguration
类型	测试类注解
位置	测试类定义上方
作用	设置JUnit加载的Spring核心配置
属性	classes: 核心配置类, 可以使用数组的格式设定加载多个配置类 locations: 配置文件, 可以使用数组的格式设定加载多个配置文件名称