

# 百知教育 --- Spring系列课程 --- 注解编程

## 第一章、注解基础概念

### 1. 什么是注解编程

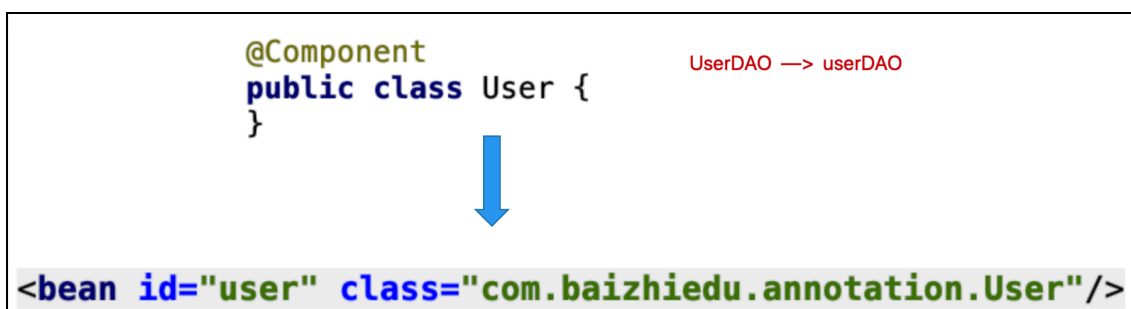
```
1  指的是在类或者方法上加入特定的注解 (@XXX), 完成特定功能的开发。
2
3  @Component
4  public class XXX{}
```

### 2. 为什么要讲解注解编程

```
1  1. 注解开发方便
2      代码简洁 开发速度大大提高
3  2. Spring开发潮流
4      Spring2.x引入注解  Spring3.x完善注解  SpringBoot普及 推广注解编程
```

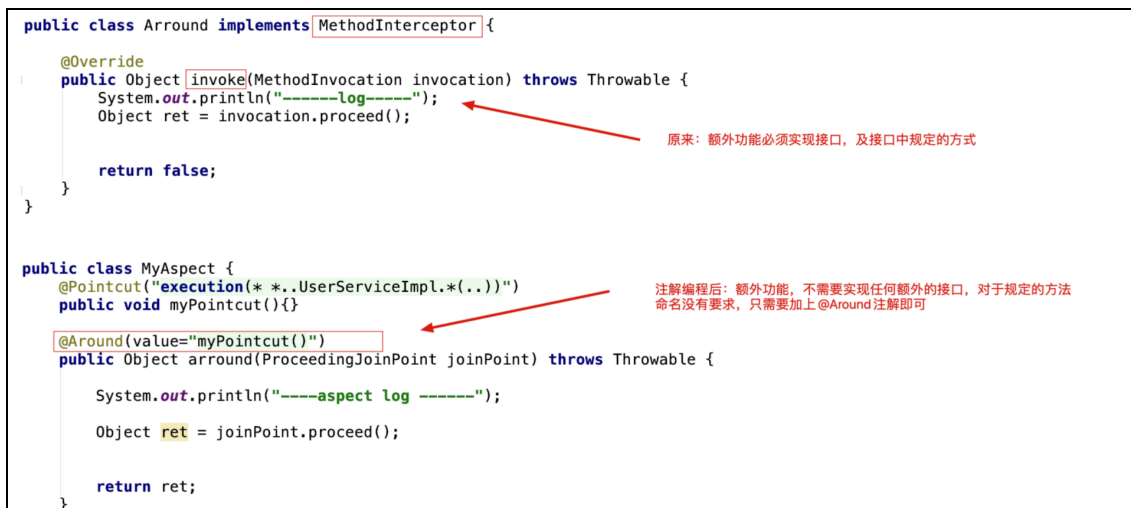
### 3. 注解的作用

- 替换XML这种配置形式，简化配置



- 替换接口，实现调用双方的契约性

1 通过注解的方式，在功能调用者和功能提供者之间达成协议，进而进行功能的调用。因为注解应用更为方便灵活，所以在现在的开发中，更推荐通过注解的形式，完成



## 4. Spring注解的发展历程

1. Spring2.x开始支持注解编程 @Component @Service @Scope..  
目的: 提供的这些注解只是为了在某些情况下简化XML的配置, 作为XML开发的有益补充。
2. Spring3.x @Configuration @Bean..  
目的: 彻底替换XML, 基于纯注解编程
3. Spring4.x SpringBoot  
提倡使用注解常见开发

## 5. Spring注解开发的一个问题

- Spring基于注解进行配置后, 还能否解耦合呢?
- 
- 在Spring框架应用注解时, 如果对注解配置的内容不满意, 可以通过Spring配置文件进行覆盖的。

## 第二章、Spring的基础注解 (Spring2.x)

- 这个阶段的注解, 仅仅是简化XML的配置, 并不能完全替代XML

### 1. 对象创建相关注解

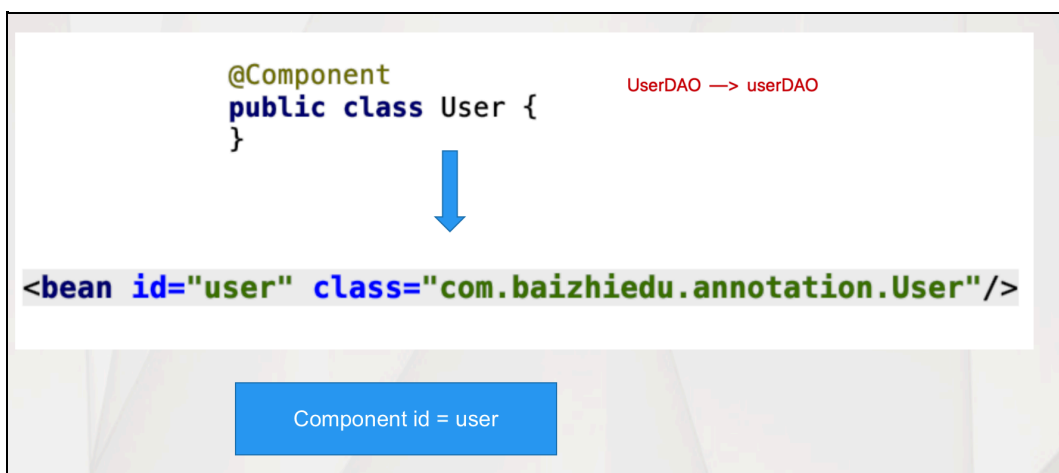
- 搭建开发环境

- `<context:component-scan base-package="com.baizhiedu"/>`
- 
- 作用: 让Spring框架在设置包及其子包中扫描对应的注解, 使其生效。

- 对象创建相关注解

- @Component

- 作用: 替换原有spring配置文件中的<bean标签
- 注意:
- id属性 component注解 提供了默认的设置方式 首单词首字母小写
- class属性 通过反射获得class内容



- @Component 细节

- 如何显示指定工厂创建对象的id值

- `@Component("u")`

- Spring配置文件覆盖注解配置内容

```
1 applicationContext.xml
2
3 <bean id="u" class="com.baizhiedu.bean.User"/>
4
5 id值 class的值 要和 注解中的设置保持一值
```

- @Component的衍生注解

```
1 @Repository ---> XXXDAO
2 @Repository
3 public class UserDao{
4
5 }
6 @Service
7 @Service
8 public class UserService{
9
10 }
11 @Controller
12 @Controller
13 public class RegAction{
14
15 }
16 注意：本质上这些衍生注解就是@Component
17 作用 <bean
18 细节 @Service("s")
19
20 目的：更加准确的表达一个类型的作用
21
22 注意：Spring整合Mybatis开发过程中 不使用@Repository @Component
23
```

- @Scope注解

```
1 作用：控制简单对象创建次数
2 注意：不添加@Scope Spring提供默认值 singleton
3 <bean id="" class="" scope="singleton|prototype"/>
```

- @Lazy注解

```
1 作用：延迟创建单实例对象
2 注意：一旦使用了@Lazy注解后，Spring会在使用这个对象时候，进行这个对象的创建
3 <bean id="" class="" lazy="false"/>
```

- 生命周期方法相关注解

```
1 1. 初始化相关方法 @PostConstruct
2 InitializingBean
3 <bean init-method="" />
4 2. 销毁方法 @PreDestroy
5 DisposableBean
6 <bean destroy-method="" />
7 注意：1. 上述的2个注解并不是Spring提供的，JSR(JavaEE规范)520
8 2. 再一次的验证，通过注解实现了接口的契约性
```

## 2. 注入相关注解

- 用户自定义类型 @Autowired

```
@Service
public class UserServiceImpl implements UserService { @Repository
    public class UserDAOImpl implements UserDAO {
        private UserDAO userDAO;
        @Override
        public void save() {
            System.out.println("UserDAOImpl.save");
        }
    }

    public UserDAO getUserDAO() {
        return userDAO;
    }

    @Autowired
    public void setUserDAO(UserDAO userDAO) {
        System.out.println("UserServiceImpl.setUserDAO");
        this.userDAO = userDAO;
    }
}
```

```
1  @Autowired细节
2  1. Autowired注解基于类型进行注入 [推荐]
3      基于类型的注入：注入对象的类型，必须与目标成员变量类型相同或者是其子类
      (实现类)
4
5  2. Autowired Qualifier 基于名字进行注入 [了解]
6      基于名字的注入：注入对象的id值，必须与Qualifier注解中设置的名字相同
7
8  3. Autowired注解放置位置
9      a) 放置在对应成员变量的set方法上
10     b) 直接把这个注解放置在成员变量之上，Spring通过反射直接对成员变量进行
      注入（赋值）[推荐]
11
12  4. JavaEE规范中类似功能的注解
13      JSR250 @Resource(name="userDAOImpl") 基于名字进行注入
14              @Autowired()
15              @Qualifier("userDAOImpl")
16              注意：如果在应用Resource注解时，名字没有配对成功，那么他会继续
      按照类型进行注入。
17      JSR330 @Inject 作用 @Autowired完全一致 基于类型进行注入 ---》
      EJB3.0
18          <dependency>
19              <groupId>javax.inject</groupId>
20              <artifactId>javax.inject</artifactId>
21              <version>1</version>
22          </dependency>
```

- JDK类型

```
1  @Value注解完成
2  1. 设置xxx.properties
3      id = 10
4      name = suns
5  2. Spring的工厂读取这个配置文件
6      <context:property-placeholder location=""/>
7  3. 代码
8      属性 @Value("${key}")
```

- @PropertySource

```

1  1. 作用：用于替换Spring配置文件中的<context:property-placeholder
    location="" />标签
2  2. 开发步骤
3      1. 设置xxx.properties
4          id = 10
5          name = suns
6      2. 应用@PropertySource
7      3. 代码
8          属性 @Value()

```

◦ @Value注解使用细节

- @Value注解不能应用在静态成员变量上

```

1  如果应用，赋值（注入）失败

```

- @Value注解+Properties这种方式，不能注入集合类型

```

1  Spring提供新的配置形式 YAML YML (SpringBoot)

```

### 3. 注解扫描详解

```

1  <context:component-scan base-package="com.baizhiedu" />
2  当前包 及其 子包

```

#### 1. 排除方式

```

1  <context:component-scan base-package="com.baizhiedu">
2      <context:exclude-filter type="" expression="" />
3      type:assignable:排除特定的类型 不进行扫描
4          annotation:排除特定的注解 不进行扫描
5          aspectj:切入点表达式
6              包切入点: com.baizhiedu.bean.*
7              类切入点: *..User
8          regex:正则表达式
9          custom: 自定义排除策略框架底层开发
10 </context:component-scan>
11
12  排除策略可以叠加使用
13 <context:component-scan base-package="com.baizhiedu">
14     <context:exclude-filter type="assignable"
15     expression="com.baizhiedu.bean.User" />
16     <context:exclude-filter type="aspectj"
17     expression="com.baizhiedu.injection.*" />
18 </context:component-scan>

```

#### 2. 包含方式

```

1  <context:component-scan base-package="com.baizhiedu" use-default-
    filters="false">
2      <context:include-filter type="" expression="" />
3  </context:component-scan>
4
5  1. use-default-filters="false"

```

```

6      作用：让Spring默认的注解扫描方式 失效。
7      2. <context:include-filter type="" expression="" />
8      作用：指定扫描那些注解
9      type:assignable:排除特定的类型 不进行扫描
10     annotation:排除特定的注解 不进行扫描
11     aspectj:切入点表达式
12         包切入点： com.baizhiedu.bean.*
13         类切入点： *..User
14     regex:正则表达式
15     custom: 自定义排除策略框架底层开发
16
17     包含的方式支持叠加
18     <context:component-scan base-package="com.baizhiedu" use-default-
19     filters="false">
20         <context:include-filter type="annotation"
21     expression="org.springframework.stereotype.Repository"/>
22         <context:include-filter type="annotation"
23     expression="org.springframework.stereotype.Service"/>
24     </context:component-scan>

```

## 4. 对于注解开发的思考

- 配置互通

```

1      Spring注解配置 配置文件的配置 互通
2
3      @Repository
4      public class UserDaoImpl{
5
6
7      }
8
9      public class UserServiceImpl{
10         private UserDao userDao;
11         set get
12     }
13
14     <bean id="userService" class="com.baizhiedu.UserServiceImpl">
15         <property name="userDao" ref="userDaoImpl"/>
16     </bean>

```

- 什么情况下使用注解 什么情况下使用配置文件

```

1      @Component 替换 <bean>
2
3      基础注解 (@Component @Autowired @Value) 程序员开发类型的配置
4
5      1. 在程序员开发的类型上 可以加入对应注解 进行对象的创建
6          User  UserService  UserDao  UserAction
7
8      2. 应用其他非程序员开发的类型时，还是需要使用<bean 进行配置的
9          SqlSessionFactoryBean  MapperScannerConfigure

```

## 5. SSM整合开发（半注解开发）

- 搭建开发环境
  - 引入相关jar 【SSM POM】
  - 引入相关配置文件
    - applicationContext.xml
    - struts.xml
    - log4.properties
    - XXXMapper.xml
  - 初始化配置
    - Web.xml Spring (ContextLoaderListener)
    - Web.xml Struts Filter
- 编码

```
1 <context:component-scan base-package="" />
```

- DAO (Spring+Mybatis)

```
1 1. 配置文件的配置
2 1. DataSource
3 2. SqlSessionFactory ----> SqlSessionFactoryBean
4 1. dataSource
5 2. typeAliasesPackage
6 3. mapperLocations
7 3. MapperScannerConfigurer ----> DAO接口实现类
8 2. 编码
9 1. entity
10 2. table
11 3. DAO接口
12 4. 实现Mapper文件
```

- Service

```
1 1. 原始对象 ---》 注入DAO
2 @Service ----> @Autowired
3
4 2. 额外功能 ---》 DataSourceTransactionManager ----> dataSource
5 3. 切入点 + 事务属性
6 @Transactional(propagation,readOnly...)
7 4. 组装切面
8 <tx:annotation-driven
```

- Controller (Spring+Struts2)

```
1 1. @Controller
2 @Scope("prototype")
3 public class RegAction implements Action{
4     @Autowired
5     private UserService userServiceImpl;
6
7 }
8 2. struts.xml
9 <action class="spring配置文件中action对应的id值"/>
```

### 第三章、Spring的高级注解（Spring3.x 及以上）

## 1. 配置Bean

1 Spring在3.x提供的新的注解，用于替换XML配置文件。

```
2
3     @Configuration
4     public class AppConfig{
5
6     }
```

1. 配置Bean在应用的过程中 替换了XML具体什么内容呢？



2. AnnotationConfigApplicationContext

```
1     1. 创建工厂代码
2         ApplicationContext ctx = new
AnnotationConfigApplicationContext();
3     2. 指定配置文件
4         1. 指定配置bean的Class
5             ApplicationContext ctx = new
AnnotationConfigApplicationContext(AppConfig.class);
6         2. 指定配置bean所在的路径
7             ApplicationContext ctx = new
AnnotationConfigApplicationContext("com.baizhiedu");
```

• 配置Bean开发的细节分析

◦ 基于注解开发使用日志

```
1     不能集成Log4j
2     集成logback
```

▪ 引入相关jar

```
1     <dependency>
2         <groupId>org.slf4j</groupId>
3         <artifactId>slf4j-api</artifactId>
4         <version>1.7.25</version>
5     </dependency>
6
7     <dependency>
8         <groupId>org.slf4j</groupId>
```



```

9      <artifactId>jcl-over-slf4j</artifactId>
10     <version>1.7.25</version>
11 </dependency>
12
13 <dependency>
14     <groupId>ch.qos.logback</groupId>
15     <artifactId>logback-classic</artifactId>
16     <version>1.2.3</version>
17 </dependency>
18
19 <dependency>
20     <groupId>ch.qos.logback</groupId>
21     <artifactId>logback-core</artifactId>
22     <version>1.2.3</version>
23 </dependency>
24
25 <dependency>
26     <groupId>org.logback-extensions</groupId>
27     <artifactId>logback-ext-spring</artifactId>
28     <version>0.1.4</version>
29 </dependency>

```

- 引入logback配置文件 (logback.xml)

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <configuration>
3      <!-- 控制台输出 -->
4      <appender name="STDOUT"
5          class="ch.qos.logback.core.ConsoleAppender">
6          <encoder>
7              <!-- 格式化输出：%d表示日期，%thread表示线程名，
              %-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符-->
              <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS}
8              [%thread] %-5level %logger{50} - %msg%n</pattern>
9          </encoder>
10         </appender>
11
12         <root level="DEBUG">
13             <appender-ref ref="STDOUT" />
14         </root>
15 </configuration>

```

- @Configuration注解的本质

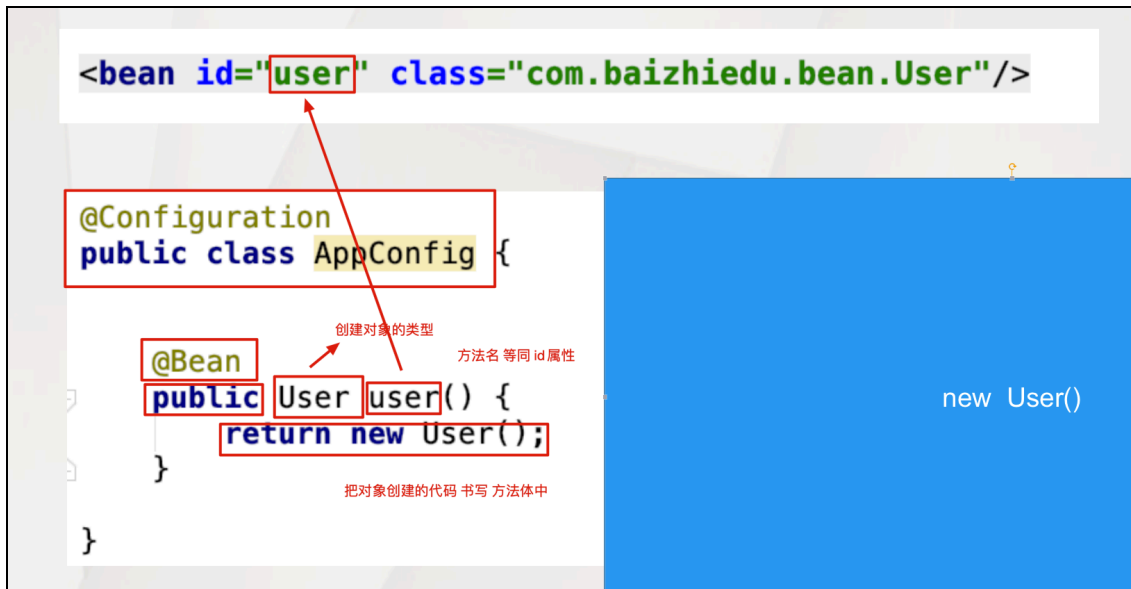
- 1 本质：也是@Component注解的衍生注解
- 2
- 3 可以应用<context:component-scan进行扫描

## 2. @Bean注解

- 1 @Bean注解在配置bean中进行使用，等同于XML配置文件中的<bean标签

### 1. @Bean注解的基本使用

- 对象的创建



1. 简单对象  
直接能够通过new方式创建的对象  
User UserService UserDao
2. 复杂对象  
不能通过new的方式直接创建的对象  
Connection SqlSessionFactory

- @Bean注解创建复杂对象的注意事项

```

1  遗留系统整合
2  @Bean
3  public Connection conn1() {
4      Connection conn = null;
5      try {
6          ConnectionFactoryBean factoryBean = new
7          ConnectionFactoryBean();
8          conn = factoryBean.getObject();
9      } catch (Exception e) {
10         e.printStackTrace();
11     }
12     return conn;
13 }

```

- 自定义id值

```
1 @Bean("id")
```

- 控制对象创建次数

```

1 @Bean
2 @Scope("singleton|prototype") 默认值 singleton

```

## 2. @Bean注解的注入

- 用户自定义类型

```
1 @Bean
```

```

2  public UserDao userDao() {
3      return new UserDaoImpl();
4  }
5
6  @Bean
7  public UserService userService(UserDao userDao) {
8      UserServiceImpl userService = new UserServiceImpl();
9      userService.setUserDAO(userDao);
10     return userService;
11 }
12
13 //简化写法
14 @Bean
15 public UserService userService() {
16     UserServiceImpl userService = new UserServiceImpl();
17     userService.setUserDAO(userDao());
18     return userService;
19 }

```

- JDK类型的注入

```

1  @Bean
2  public Customer customer() {
3      Customer customer = new Customer();
4      customer.setId(1);
5      customer.setName("xiaohai");
6
7      return customer;
8  }

```

- JDK类型注入的细节分析

```

1  如果直接在代码中进行set方法的调用，会存在耦合的问题
2
3  @Configuration
4  @PropertySource("classpath:/init.properties")
5  public class AppConfig1 {
6
7      @Value("${id}")
8      private Integer id;
9      @Value("${name}")
10     private String name;
11
12     @Bean
13     public Customer customer() {
14         Customer customer = new Customer();
15         customer.setId(id);
16         customer.setName(name);
17
18         return customer;
19     }
20 }

```

### 3. @ComponentScan注解

```
1  @ComponentScan注解在配置bean中进行使用，等同于XML配置文件中的
   <context:component-scan>标签
2
3  目的：进行相关注解的扫描（@Component @Value ...@Autowired）
```

## 1. 基本使用

```
1  @Configuration
2  @ComponentScan(basePackages = "com.baizhiedu.scan")
3  public class AppConfig2 {
4
5  }
6
7  <context:component-scan base-package="" />
```

## 2. 排除、包含的使用

- 排除

```
1  <context:component-scan base-package="com.baizhiedu">
2    <context:exclude-filter type="assignable"
3      expression="com.baizhiedu.bean.User" />
4  </context:component-scan>
5
6  @ComponentScan(basePackages = "com.baizhiedu.scan",
7    excludeFilters = {@ComponentScan.Filter(type=
8      FilterType.ANNOTATION,value={Service.class}),
9      @ComponentScan.Filter(type=
10       FilterType.ASPECTJ,pattern = "*..User1")})
11
12  type = FilterType.ANNOTATION      value
13      .ASSIGNABLE_TYPE              value
14      .ASPECTJ                      pattern
15      .REGEX                        pattern
16      .CUSTOM                       value
```

- 包含

```
1  <context:component-scan base-package="com.baizhiedu" use-default-
2    filters="false">
3    <context:include-filter type="" expression="" />
4  </context:component-scan>
5
6  @ComponentScan(basePackages = "com.baizhiedu.scan",
7    useDefaultFilters = false,
8    includeFilters = {@ComponentScan.Filter(type=
9      FilterType.ANNOTATION,value={Service.class})})
10
11  type = FilterType.ANNOTATION      value
12      .ASSIGNABLE_TYPE              value
13      .ASPECTJ                      pattern
14      .REGEX                        pattern
15      .CUSTOM                       value
```

## 4. Spring工厂创建对象的多重配置方式

### 1. 多种配置方式的应用场景

```
@Component
public class User {
}
```

```
@Bean
public User user() {
    return new User();
}
```

@Component 衍生 @Autowired  
程序员自己开发的类型上:  
UserService UserDao Controller..

@Bean  
框架提供的类型, 别的程序员开发的  
类型 (没有源码)  
SqlSessionFactoryBean  
MapperScannerConfigure

```
<bean id="user" class="com.baizhiedu.bean.User"/>
```

纯注解的开发过程中, 基本不用  
遗留系统的整合

@Import

1. Spring框架的底层使用  
2. 多配置bean整合

### 2. 配置优先级

```
1  @Component及其衍生注解 < @Bean < 配置文件bean标签
2  优先级高的配置 覆盖优先级低配置
3
4  @Component
5  public class User{
6
7  }
8
9  @Bean
10 public User user(){
11     return new User();
12 }
13
14 <bean id="user" class="xxx.User"/>
15
16 配置覆盖: id值 保持一致
```

- 解决基于注解进行配置的耦合问题

```
1  @Configuration
2  //@ImportResource("applicationContext.xml")
3  public class AppConfig4 {
4
5      @Bean
6      public UserDao userDao() {
7          return new UserDaoImpl();
8      }
9  }
10
11 @Configuration
12 @ImportResource("applicationContext.xml")
13 public class AppConfig5{
14
15 }
```

```

16
17 applicationContext.xml
18 <bean id="userDAO"
    class="com.baizhiedu.injection.UserDAOImplNew" />

```

## 5. 整合多个配置信息

- 为什么会有多个配置信息

1 拆分多个配置bean的开发，是一种模块化开发的形式，也体现了面向对象各司其职的设计思想

- 多配置信息整合的方式
  - 多个配置Bean的整合
  - 配置Bean与@Component相关注解的整合
  - 配置Bean与SpringXML配置文件的整合
- 整合多种配置需要关注那些要点
  - 如何使多配置的信息 汇总成一个整体
  - 如何实现跨配置的注入

### 1. 多个配置Bean的整合

- 多配置的信息汇总
  - base-package进行多个配置Bean的整合

The screenshot shows an IDE interface. On the left, a package explorer shows a package named 'com.baizhiedu' containing two classes: 'AppConfig1' and 'AppConfig2'. On the right, the code editor displays two Java classes. The first class, 'AppConfig1', is annotated with '@Configuration' and '@Bean', and contains a 'user()' method that returns a new 'User()' object. The second class, 'AppConfig2', is also annotated with '@Configuration' and '@Bean', and contains a 'product()' method that returns a new 'Product()' object. Below the code editor, a line of Java code is shown: 'ApplicationContext ctx = new AnnotationConfigApplicationContext(...basePackages: "com.baizhiedu");'.

- @Import

1. 可以创建对象
2. 多配置bean的整合

The screenshot shows an IDE interface. On the left, the code editor displays a Java class 'AppConfig1' annotated with '@Configuration' and '@Import(AppConfig2.class)'. It contains a 'user()' method that returns a new 'User()' object. On the right, the code editor displays another Java class 'AppConfig2' annotated with '@Configuration' and '@Bean', containing a 'product()' method that returns a new 'Product()' object. Below the code editor, a line of Java code is shown: 'ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig1.class);'. At the bottom of the IDE, a blue bar contains the XML snippet '<import resource=""/>'.

- 在工厂创建时，指定多个配置Bean的Class对象 【了解】

```
1 ApplicationContext ctx = new
  AnnotationConfigApplicationContext(AppConfig1.class, AppConfig2
    .class);
```

- 跨配置进行注入

```
1  在应用配置Bean的过程中，不管使用哪种方式进行配置信息的汇总，其操作方式都是通过
  成员变量加入@Autowired注解完成。
2  @Configuration
3  @Import(AppConfig2.class)
4  public class AppConfig1 {
5
6      @Autowired
7      private UserDao userDao;
8
9      @Bean
10     public UserService userService() {
11         UserServiceImpl userService = new UserServiceImpl();
12         userService.setUserDao(userDao);
13         return userService;
14     }
15 }
16
17 @Configuration
18 public class AppConfig2 {
19
20     @Bean
21     public UserDao userDao() {
22         return new UserDaoImpl();
23     }
24 }
```

## 2. 配置Bean与@Component相关注解的整合

```
1  @Component(@Repository)
2  public class UserDaoImpl implements UserDao{
3
4  }
5
6  @Configuration
7  @ComponentScan("")
8  public class AppConfig3 {
9
10     @Autowired
11     private UserDao userDao;
12
13     @Bean
14     public UserService userService() {
15         UserServiceImpl userService = new UserServiceImpl();
16         userService.setUserDao(userDao);
17         return userService;
18     }
19 }
20
```

```
21 ApplicationContext ctx = new
    AnnotationConfigApplicationContext(AppConfig3.class);
```

### 3. 配置Bean与配置文件整合

```
1  1. 遗留系统的整合 2. 配置覆盖
2
3  public class UserDAOImpl implements UserDAO{
4
5  }
6  <bean id="userDAO" class="com.baizhiedu.injection.UserDAOImpl"/>
7
8  @Configuration
9  @ImportResource("applicationContext.xml")
10 public class AppConfig4 {
11
12     @Autowired
13     private UserDAO userDAO;
14
15     @Bean
16     public UserService userService() {
17         UserServiceImpl userService = new UserServiceImpl();
18         userService.setUserDAO(userDAO);
19         return userService;
20     }
21 }
22
23 ApplicationContext ctx = new
    AnnotationConfigApplicationContext(AppConfig4.class);
```

### 6. 配置Bean底层实现原理

1 Spring在配置Bean中加入了@Configuration注解后，底层就会通过Cglib的代理方式，来进行对象相关的配置、处理

```
▼ this = {AppConfig$$EnhancerBySpringCGLIB$$8d7755fe@2509}
  f CGLIB$BOUND = true
  ▶ f CGLIB$CALLBACK_0 = {ConfigurationClassEnhancer$BeanMethodInterceptor@2512}
  ▶ f CGLIB$CALLBACK_1 = {ConfigurationClassEnhancer$BeanFactoryAwareMethodInterceptor@2518}
  ▶ f CGLIB$CALLBACK_2 = {NoOp$1@2519}
  ▶ f $$beanFactory = {DefaultListableBeanFactory@2520} "org.springframework.beans.factory.support."
```

### 7. 四维一体的开发思想

#### 1. 什么是四维一体

```
1 Spring开发一个功能的4种形式，虽然开发方式不同，但是最终效果是一样的。
2 1. 基于schema
3 2. 基于特定功能注解
4 3. 基于原始<bean
5 4. 基于@Bean注解
```

#### 2. 四维一体的开发案例



```
1 1. <context:property-placeholder
2 2. @PropertySource    【推荐】
3 3. <bean id="" class="PropertySourcePlaceholderConfigure" />
4 4. @Bean              【推荐】
```

## 8. 纯注解版AOP编程

### 1. 搭建环境

```
1 1. 应用配置Bean
2 2. 注解扫描
```

### 2. 开发步骤

```
1 1. 原始对象
2   @Service(@Component)
3   public class UserServiceImpl implements UserService{
4
5   }
6 2. 创建切面类 （额外功能 切入点 组装切面）
7   @Aspect
8   @Component
9   public class MyAspect {
10
11       @Around("execution(* login(..))")
12       public Object around(ProceedingJoinPoint joinPoint) throws
13       Throwable {
14
15           System.out.println("----aspect log -----");
16
17           Object ret = joinPoint.proceed();
18
19           return ret;
20       }
21   }
22 3. Spring的配置文件中
23   <aop:aspectj-autoproxy />
24   @EnableAspectjAutoProxy ----> 配置Bean
```

### 3. 注解AOP细节分析

```
1 1. 代理创建方式的切换 JDK Cglib
2   <aop:aspectj-autoproxy proxy-target-class=true|false />
3   @EnableAspectjAutoProxy(proxyTargetClass)
4 2. SpringBoot AOP的开发方式
5   @EnableAspectjAutoProxy 已经设置好了
6
7   1. 原始对象
8       @Service(@Component)
9       public class UserServiceImpl implements UserService{
10
11       }
12 2. 创建切面类 （额外功能 切入点 组装切面）
```

```

13     @Aspect
14     @Component
15     public class MyAspect {
16
17         @Around("execution(* login(..))")
18         public Object around(ProceedingJoinPoint joinPoint) throws
Throwable {
19
20             System.out.println("----aspect log -----");
21
22             Object ret = joinPoint.proceed();
23
24
25             return ret;
26         }
27     }
28     Spring AOP 代理默认实现 JDK   SpringBOOT AOP 代理默认实现 Cglib

```

## 9. 纯注解版Spring+MyBatis整合

- 基础配置 （配置Bean）

```

1  1. 连接池
2      <!--连接池-->
3      <bean id="dataSource"
4          class="com.alibaba.druid.pool.DruidDataSource">
5          <property name="driverClassName"
6              value="com.mysql.jdbc.Driver"></property>
7          <property name="url" value="jdbc:mysql://localhost:3306/suns?
8              useSSL=false"></property>
9          <property name="username" value="root"></property>
10         <property name="password" value="123456"></property>
11     </bean>
12
13     @Bean
14     public DataSource dataSource(){
15         DruidDataSource dataSource = new DruidDataSource();
16         dataSource.setDriverClassName("");
17         dataSource.setUrl();
18         ...
19         return dataSource;
20     }
21
22 2. SqlSessionFactoryBean
23     <!--创建SqlSessionFactory SqlSessionFactoryBean-->
24     <bean id="sqlSessionFactoryBean"
25         class="org.mybatis.spring.SqlSessionFactoryBean">
26         <property name="dataSource" ref="dataSource"></property>
27         <property name="typeAliasesPackage"
28             value="com.baizhiedu.entity"></property>
29         <property name="mapperLocations">
30             <list>
31
32                 <value>classpath:com.baizhiedu.mapper/*Mapper.xml</value>
33
34             </list>
35         </property>
36     </bean>

```

```

28         </property>
29     </bean>
30
31     @Bean
32     public SqlSessionFactoryBean sqlSessionFactoryBean(DataSource
dataSource){
33         SqlSessionFactoryBean sqlSessionFactoryBean = new
SqlSessionFactoryBean();
34         sqlSessionFactoryBean.setDataSource(dataSource);
35         sqlSessionFactoryBean.setTypeAliasesPackage("");
36         ...
37         return sqlSessionFactoryBean;
38     }
39
40 3. MapperScannerConfigure
41     <!--创建DAO对象 MapperScannerConfigure-->
42     <bean id="scanner"
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
43         <property name="sqlSessionFactoryBeanName"
value="sqlSessionFactoryBean"></property>
44         <property name="basePackage" value="com.baizhiedu.dao">
</property>
45     </bean>
46
47     @MapperScan(basePackages={"com.baizhiedu.dao"}) ---> 配置bean完成
48

```

- 编码

```

1  1. 实体
2  2. 表
3  3. DAO接口
4  4. Mapper文件

```

## 1. MapperLocations编码时通配的写法

```

1  //设置Mapper文件的路径
2  sqlSessionFactoryBean.setMapperLocations(Resource..);
3  Resource resouce = new ClassPathResouce("UserDAOMapper.xml")
4
5  sqlSessionFactoryBean.setMapperLocations(new
ClassPathResource("UserDAOMapper.xml"));
6
7  <property name="mapperLocations">
8      <list>
9          <value>classpath:com.baizhiedu.mapper/*Mapper.xml</value>
10     </list>
11 </property>
12 一组Mapper文件
13
14 ResourcePatternResolver resolver = new
PathMatchingResourcePatternResolver();
15 Resource[] resources =
resolver.getResources("com.baizhi.mapper/*Mapper.xml");
16 sqlSessionFactoryBean.setMapperLocations(resources)

```

## 2. 配置Bean数据耦合的问题

```
1  mybatis.driverClassName = com.mysql.jdbc.Driver
2  mybatis.url = jdbc:mysql://localhost:3306/suns?useSSL=false
3  mybatis.username = root
4  mybatis.password = 123456
5  mybatis.typeAliasesPackages = com.baizhiedu.mybatis
6  mybatis.mapperLocations = com.baizhiedu.mapper/*Mapper.xml
7
8  @Component
9  @PropertySource("classpath:mybatis.properties")
10 public class MybatisProperties {
11     @Value("${mybatis.driverClassName}")
12     private String driverClassName;
13     @Value("${mybatis.url}")
14     private String url;
15     @Value("${mybatis.username}")
16     private String username;
17     @Value("${mybatis.password}")
18     private String password;
19     @Value("${mybatis.typeAliasesPackages}")
20     private String typeAliasesPackages;
21     @Value("${mybatis.mapperLocations}")
22     private String mapperLocations;
23 }
24
25 public class MyBatisAutoConfiguration {
26
27     @Autowired
28     private MybatisProperties mybatisProperties;
29
30     @Bean
31     public DataSource dataSource() {
32         DruidDataSource dataSource = new DruidDataSource();
33
34         dataSource.setDriverClassName(mybatisProperties.getDriverClassName());
35         dataSource.setUrl(mybatisProperties.getUrl());
36         dataSource.setUsername(mybatisProperties.getUsername());
37         dataSource.setPassword(mybatisProperties.getPassword());
38         return dataSource;
39     }
40
41     @Bean
42     public SqlSessionFactoryBean sqlSessionFactoryBean(DataSource dataSource) {
43         SqlSessionFactoryBean sqlSessionFactoryBean = new
44         SqlSessionFactoryBean();
45         sqlSessionFactoryBean.setDataSource(dataSource);
46
47         sqlSessionFactoryBean.setTypeAliasesPackage(mybatisProperties.ge
48         tTypeAliasesPackages());
49         //sqlSessionFactoryBean.setMapperLocations(new
50         ClassPathResource("UserDAOMapper.xml"));
51     }
52 }
```

```

47         try {
48             ResourcePatternResolver resolver = new
PathMatchingResourcePatternResolver();
49             Resource[] resources =
resolver.getResources(mybatisProperties.getMapperLocations());
50             sqlSessionFactoryBean.setMapperLocations(resources);
51         } catch (IOException e) {
52             e.printStackTrace();
53         }
54
55         return sqlSessionFactoryBean;
56     }
57 }
58

```

## 10. 纯注解版事务编程

```

1  1. 原始对象 XXXService
2      <bean id="userService"
class="com.baizhiedu.service.UserServiceImpl">
3      <property name="userDAO" ref="userDAO"/>
4      </bean>
5
6      @Service
7      public class UserServiceImpl implements UserService{
8          @Autowired
9          private UserDAO userDAO;
10     }
11
12  2. 额外功能
13      <!--DataSourceTransactionManager-->
14      <bean id="dataSourceTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManage
r">
15          <property name="dataSource" ref="dataSource"/>
16      </bean>
17
18      @Bean
19      public DataSourceTransactionManager
dataSourceTransactionManager(DataSource dataSource){
20          DataSourceTransactionManager dstm = new
DataSourceTransactionManager();
21          dstm.setDataSource(dataSource);
22          return dstm
23      }
24
25  3. 事务属性
26      @Transactional
27      @Service
28      public class UserServiceImpl implements UserService {
29          @Autowired
30          private UserDAO userDAO;
31
32  4. 基于Schema的事务配置

```

```
33     <tx:annotation-driven transaction-  
manager="dataSourceTransactionManager"/>  
34     @EnableTransactionManager ---> 配置Bean
```

```
1  1. ApplicationContext ctx = new  
AnnotationConfigApplicationContext("com.baizhiedu.mybatis");  
2      SpringBoot 实现思想  
3  2. 注解版MVC整合, SpringMVC中进行详细讲解  
4      SpringMyBatis --->DAO 事务基于注解 --> Service Controller  
5      org.springframework.web.context.ContextLoaderListener ---> XML工厂 无  
法提供 new AnnotationConfigApplicationContext
```

## 11. Spring框架中YML的使用

### 1. 什么是YML

```
1  YML(YAML)是一种新形式的配置文件, 比XML更简单, 比Properties更强大。  
2  
3  YAML is a nice human-readable format for configuration, and it has some  
useful hierarchical properties. It's more or less a superset of JSON,  
so it has a lot of similar features.
```

### 2. Properties进行配置问题

```
1  1. Properties表达过于繁琐, 无法表达数据的内在联系。  
2  2. Properties无法表达对象 集合类型
```

### 3. YML语法简介

```
1  1. 定义yaml文件  
2      xxx.yml xxx.yaml  
3  2. 语法  
4      1. 基本语法  
5          name: suns  
6          password: 123456  
7      2. 对象概念  
8          account:  
9              id: 1  
10             password: 123456  
11      3. 定义集合  
12          service:  
13              - 11111  
14              - 22222
```

### 4. Spring与YML集成思路的分析

```

1  1. 准备yaml配置文件
2      init.yml
3      name: suns
4      password: 123456
5  2. 读取yaml 转换成 Properties
6      YamlPropertiesFactoryBean.setResources( yaml配置文件的路径 ) new
        ClassPathResource();
7      YamlPropertiesFactoryBean.getObject() ----> Properties
8  3. 应用PropertySourcesPlaceholderConfigurer
9      PropertySourcesPlaceholderConfigurer.setProperties();
10 4. 类中 @Value注解 注入

```

## 5. Spring与YML集成编码

- 环境搭建

```

1  <dependency>
2      <groupId>org.yaml</groupId>
3      <artifactId>snakeyaml</artifactId>
4      <version>1.23</version>
5  </dependency>
6  最低版本 1.18

```

- 编码

```

1  1. 准备yaml配置文件
2  2. 配置Bean中操作 完成YAML读取 与 PropertySourcePlaceholderConfigure
    的创建
3      @Bean
4      public PropertySourcesPlaceholderConfigurer configurator() {
5          YamlPropertiesFactoryBean yamlPropertiesFactoryBean = new
        YamlPropertiesFactoryBean();
6          yamlPropertiesFactoryBean.setResources(new
        ClassPathResource("init.yml"));
7          Properties properties =
        yamlPropertiesFactoryBean.getObject();
8
9          PropertySourcesPlaceholderConfigurer configurator = new
        PropertySourcesPlaceholderConfigurer();
10         configurator.setProperties(properties);
11         return configurator;
12     }
13 3. 类 加入 @Value注解

```

## 6. Spring与YML集成的问题

```

1  1. 集合处理的问题
2      SpringEL表达式解决
3      @Value("#{${list}'.split(',')}")
4  2. 对象类型的YAML进行配置时 过于繁琐
5      @Value("${account.name}")
6
7  SpringBoot @ConfigurationProperties

```







