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

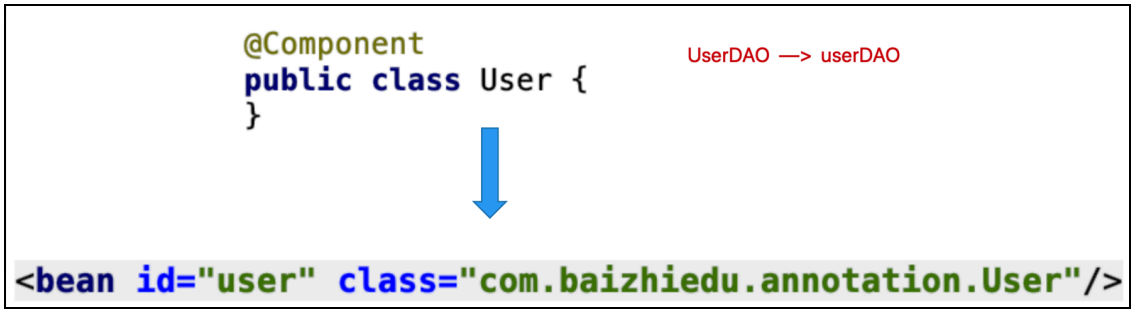
```
指的是在类或者方法上加入特定的注解（@XXX），完成特定功能的开发。

  @Component
public class XXX{}
```

2. 为什么要讲解注解编程

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

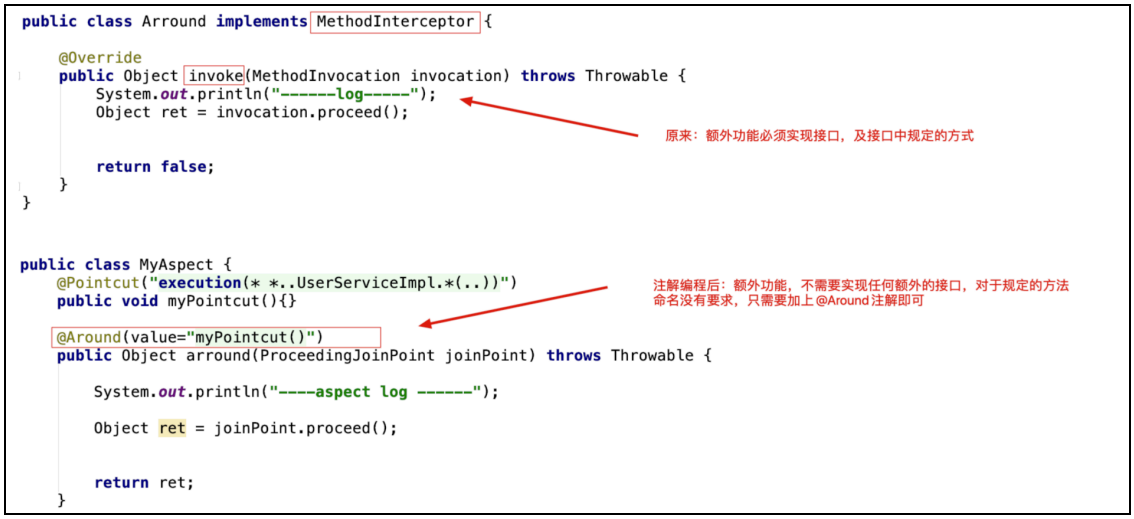
3. 注解的作用

- 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

```
通过注解的方式，在功能调用者和功能提供者之间达成约定，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
```

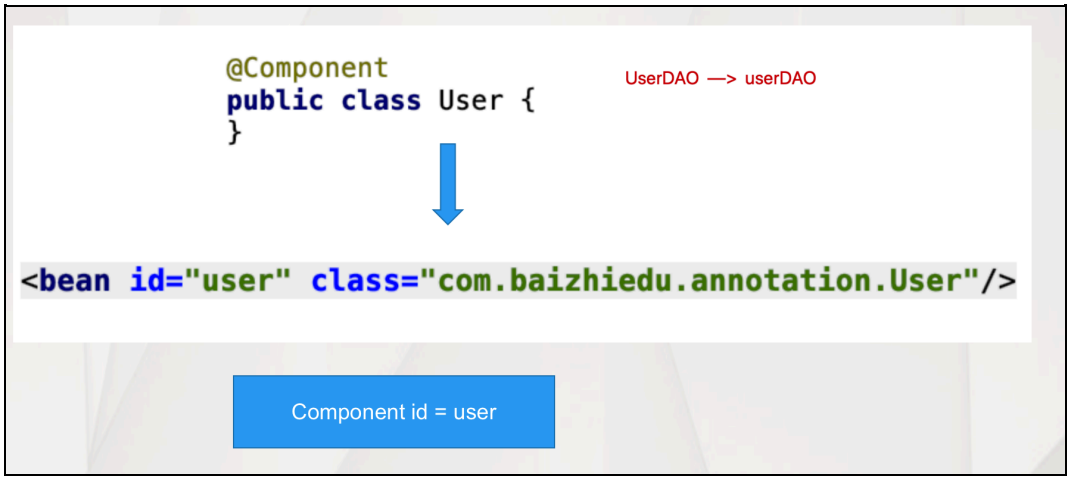
1. 对象创建相关注解

- 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

```
applicationContext.xml

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

id值 class的值 要和 注解中的设置保持一值
```

  - @Component的衍生注解

```
@Repository  --->  XXXDAO
  @Repository
  public class UserDAO{

  }
@Service
  @Service
  public class UserService{

  }
@Controller
  @Controller
  public class RegAction{

  }
注意：本质上这些衍生注解就是@Component
      作用 <bean
      细节 @Service("s")

目的：更加准确的表达一个类型的作用

注意：Spring整合Mybatis开发过程中 不使用@Repository @Component

```

- @Scope注解

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

- @Lazy注解

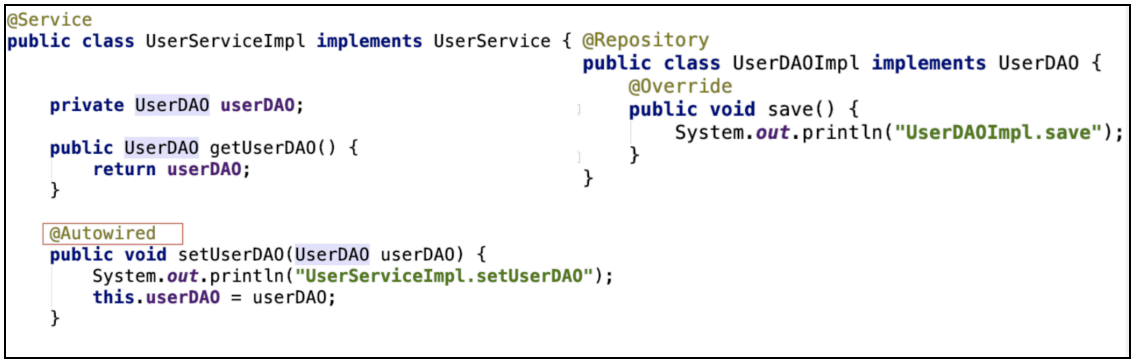
```
作用：延迟创建单实例对象
注意：一旦使用了@Lazy注解后，Spring会在使用这个对象时候，进行这个对象的创建
<bean id="" class="" lazy="false"/>
```

- 生命周期方法相关注解

```
1. 初始化相关方法 @PostConstruct
   InitializingBean
   <bean init-method=""/>
2. 销毁方法 @PreDestroy
   DisposableBean
   <bean destory-method=""/>
注意：1. 上述的2个注解并不是Spring提供的，JSR(JavaEE规范)520
     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 {

    private UserDAO userDAO;

    public UserDAO getUserDAO() {
        return userDAO;
    }

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

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

```
@Autowired细节
1. Autowired注解基于类型进行注入 [推荐]
   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）

2. Autowired Qualifier 基于名字进行注入 [了解]
   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

3. Autowired注解放置位置
    a) 放置在对应成员变量的set方法上
    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]

4. JavaEE规范中类似功能的注解
    JSR250 @Resouce(name="userDAOImpl") 基于名字进行注入
           @Autowired()
           @Qualifier("userDAOImpl")
           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。
    JSR330 @Inject 作用 @Autowired完全一致 基于类型进行注入 --->EJB3.0
          <dependency>
            <groupId>javax.inject</groupId>
            <artifactId>javax.inject</artifactId>
            <version>1</version>
          </dependency>
```

- JDK类型

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

  - @PropertySource

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

    ```
    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

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

3. 注解扫描详解

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

1. 排除方式

```
<context:component-scan base-package="com.baizhiedu">
   <context:exclude-filter type="" expression=""/>
   type:assignable:排除特定的类型 不进行扫描
        annotation:排除特定的注解 不进行扫描
        aspectj:切入点表达式
                包切入点： com.baizhiedu.bean..*
                类切入点： *..User
        regex:正则表达式
        custom: 自定义排除策略框架底层开发
</context:component-scan>

排除策略可以叠加使用
<context:component-scan base-package="com.baizhiedu">
  <context:exclude-filter type="assignable" expression="com.baizhiedu.bean.User"/>

  <context:exclude-filter type="aspectj" expression="com.baizhiedu.injection..*"/>
</context:component-scan>
```

2. 包含方式

```
<context:component-scan base-package="com.baizhiedu" use-default-filters="false">
   <context:include-filter type="" expression=""/>
</context:component-scan>

1. use-default-filters="false"
```

```
   作用：让Spring默认的注解扫描方式 失效。
2. <context:include-filter type="" expression=""/>
   作用：指定扫描那些注解
   type:assignable:排除特定的类型 不进行扫描
        annotation:排除特定的注解 不进行扫描
        aspectj:切入点表达式
                包切入点：  com.baizhiedu.bean..*
                类切入点：  *..User
        regex:正则表达式
        custom：自定义排除策略框架底层开发

包含的方式支持叠加
 <context:component-scan base-package="com.baizhiedu" use-default-filters="false">
        <context:include-filter type="annotation" expression="org.springframework.stereotype.Repository"/>
        <context:include-filter type="annotation" expression="org.springframework.stereotype.Service"/>
 </context:component-scan>
```

4. 对于注解开发的思考

- 配置互通

```
Spring注解配置 配置文件的配置 互通

@Repository
public class UserDAOImpl{


}

public class UserServiceImpl{
   private UserDAO userDAO;
   set get
}

<bean id="userService" class="com.baizhiedu.UserServiceImpl">
   <property name="userDAO" ref="userDAOImpl"/>
</bean>
```

- 什么情况下使用注解 什么情况下使用配置文件

```
@Component 替换 <bean

基础注解（@Component @Autowired @Value）程序员开发类型的配置

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
   User  UserService  UserDAO  UserAction

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
   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

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

  - DAO (Spring+Mybatis)

```
1. 配置文件的配置
   1. DataSource
   2. SqlSessionFactory ----> SqlSessionFactoryBean
      1. dataSource
      2. typeAliasesPackage
      3. mapperLocations
   3. MapperScannerConfigur ---> DAO接口实现类
2. 编码
   1. entity
   2. table
   3. DAO接口
   4. 实现Mapper文件
```

  - Service

```
1. 原始对象 ---》 注入DAO
   @Service ---> @Autowired

2. 额外功能 ---》 DataSourceTransactionManager ---> dataSource
3. 切入点 + 事务属性
   @Transactional(propagation,readOnly...)
4. 组装切面
   <tx:annotation-driven
```

  - Controller (Spring+Struts2)

```
1. @Controller
   @Scope("prototype")
   public class RegAction implements Action{
      @Autowired
      private UserService userServiceImpl;

   }
2. struts.xml
    <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

```
Spring在3.x提供的新的注解，用于替换XML配置文件。

  @Configuration
public class AppConfig{

}
```

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. 创建工厂代码
    ApplicationContext ctx = new AnnotationConfigApplicationContext();
 2. 指定配置文件
    1. 指定配置bean的Class
        ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    2. 指定配置bean所在的路径
        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

```
不能集成Log4j
集成logback
```

    - 引入相关jar

```
<dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.25</version>
    </dependency>

    <dependency>
      <groupId>org.slf4j</groupId>
```

```
        <artifactId>jcl-over-slf4j</artifactId>
        <version>1.7.25</version>
      </dependency>

      <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
      </dependency>

      <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-core</artifactId>
        <version>1.2.3</version>
      </dependency>

      <dependency>
        <groupId>org.logback-extensions</groupId>
        <artifactId>logback-ext-spring</artifactId>
        <version>0.1.4</version>
      </dependency>
```

- 引入logback配置文件 (logback.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!-- 控制台输出 -->
    <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <!--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}
[%thread] %-5level %logger{50} - %msg%n</pattern>
        </encoder>
    </appender>

    <root level="DEBUG">
        <appender-ref ref="STDOUT" />
    </root>

</configuration>
```

  - @Configuration注解的本质

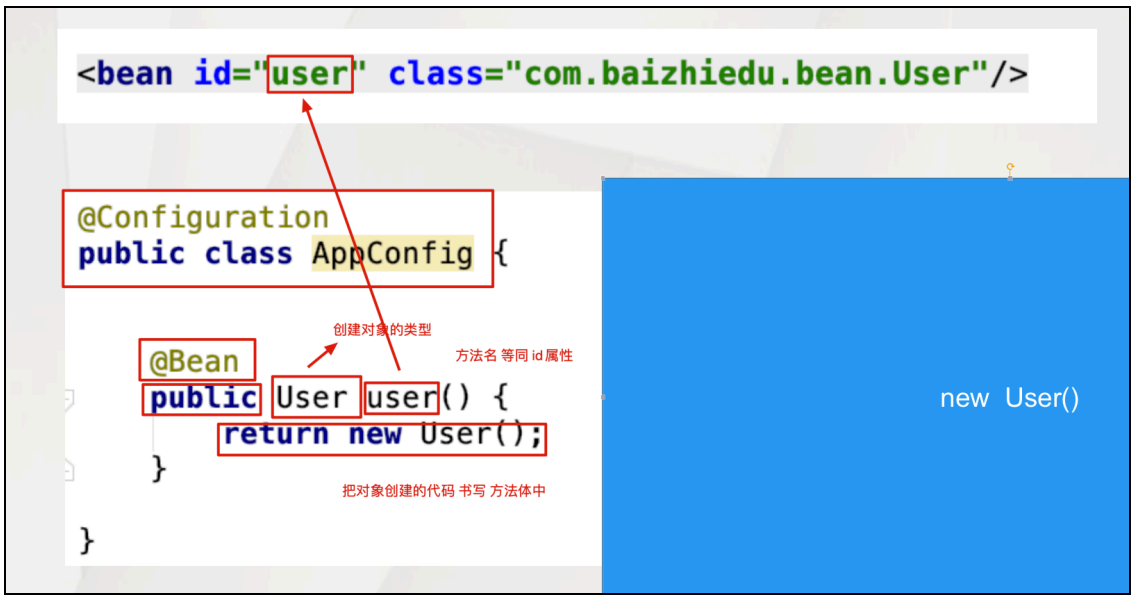
```
本质：也是@Component注解的衍生注解

可以应用<context:component-scan进行扫描
```

2. @Bean注解

```
@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

```
遗留系统整合
@Bean
public Connection conn1() {
  Connection conn = null;
  try {
    ConnectionFactoryBean factoryBean = new
ConnectionFactoryBean();
    conn = factoryBean.getObject();
  } catch (Exception e) {
    e.printStackTrace();
  }
  return conn;
}
```

- 自定义id值

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

- 控制对象创建次数

```
@Bean
@Scope("singleton|prototype") 默认值 singleton
```

2. @Bean注解的注入

- 用户自定义类型

```
@Bean
```

```
  public UserDAO userDAO() {
    return new UserDAOImpl();
  }

  @Bean
  public UserService userService(UserDAO userDAO) {
    UserServiceImpl userService = new UserServiceImpl();
    userService.setUserDAO(userDAO);
    return userService;
  }

  //简化写法
  @Bean
  public UserService userService() {
    UserServiceImpl userService = new UserServiceImpl();
    userService.setUserDAO(userDAO());
    return userService;
  }
```

- JDK类型的注入

```
@Bean
public Customer customer() {
  Customer customer = new Customer();
  customer.setId(1);
  customer.setName("xiaohei");

  return customer;
}
```

  - JDK类型注入的细节分析

```
如果直接在代码中进行set方法的调用，会存在耦合的问题

@Configuration
@PropertySource("classpath:/init.properties")
public class AppConfig1 {

    @Value("${id}")
    private Integer id;
    @Value("${name}")
    private String name;

    @Bean
    public Customer customer() {
        Customer customer = new Customer();
        customer.setId(id);
        customer.setName(name);

        return customer;
    }
}
```

3. @ComponentScan注解

```
@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

目的：进行相关注解的扫描  (@Component @Value ...@Autowired)
```

1. 基本使用

```
@Configuration
@ComponentScan(basePackages = "com.baizhiedu.scan")
public class AppConfig2 {

}

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

2. 排除、包含的使用

- 排除

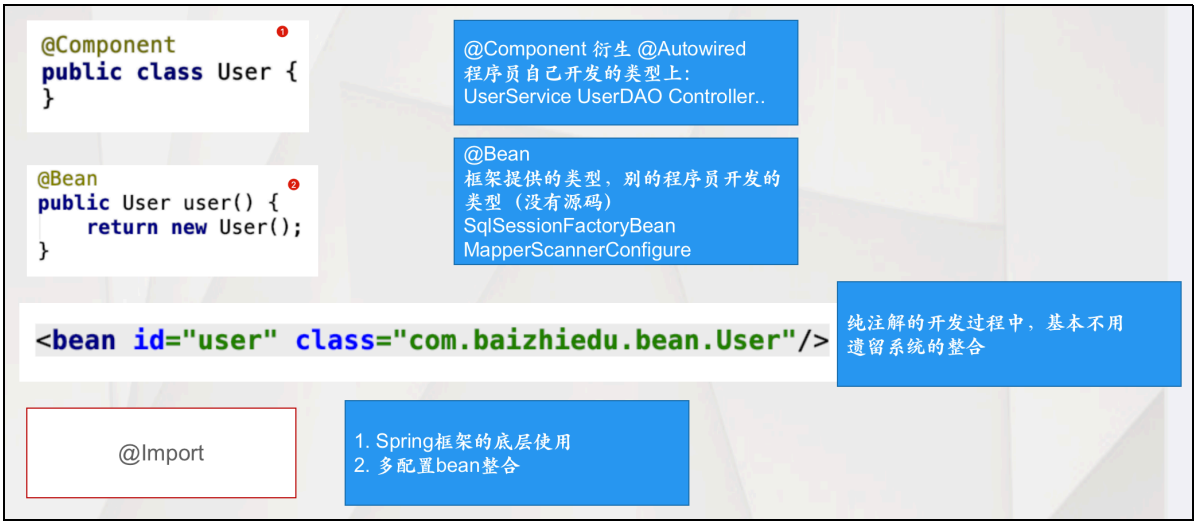
```
<context:component-scan base-package="com.baizhiedu">
  <context:exclude-filter type="assignable"
expression="com.baizhiedu.bean.User"/>
</context:component-scan>

@ComponentScan(basePackages = "com.baizhiedu.scan",
               excludeFilters = {@ComponentScan.Filter(type=
FilterType.ANNOTATION,value={Service.class}),
                                 @ComponentScan.Filter(type=
FilterType.ASPECTJ,pattern = "*..User1")})

type = FilterType.ANNOTATION          value
                 .ASSIGNABLE_TYPE     value
                 .ASPECTJ             pattern
                 .REGEX               pattern
                 .CUSTOM              value
```

- 包含

```
<context:component-scan base-package="com.baizhiedu" use-default-
filters="false">
   <context:include-filter type="" expression=""/>
</context:component-scan>

@ComponentScan(basePackages = "com.baizhiedu.scan",
               useDefaultFilters = false,
               includeFilters = {@ComponentScan.Filter(type=
FilterType.ANNOTATION,value={Service.class})})

type = FilterType.ANNOTATION          value
                 .ASSIGNABLE_TYPE     value
                 .ASPECTJ             pattern
                 .REGEX               pattern
                 .CUSTOM              value
```

4. Spring工厂创建对象的多种配置方式

1. 多种配置方式的应用场景



2. 配置优先级

```
@Component及其衍生注解 < @Bean < 配置文件bean标签
优先级高的配置 覆盖优先级低配置

@Component
public class User{

}

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

<bean id="user" class="xxx.User"/>

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

```
@Configuration
//@ImportResource("applicationContext.xml")
public class AppConfig4 {

    @Bean
    public UserDAO userDAO() {
        return new UserDAOImpl();
    }
}

@Configuration
@ImportResource("applicationContext.xml")
public class AppConfig5{

}
```

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

5. 整合多个配置信息

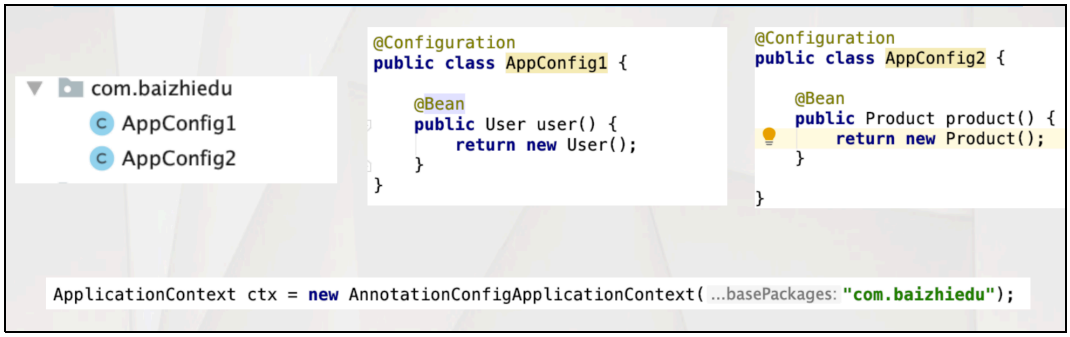
- 为什么会有多个配置信息

```
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

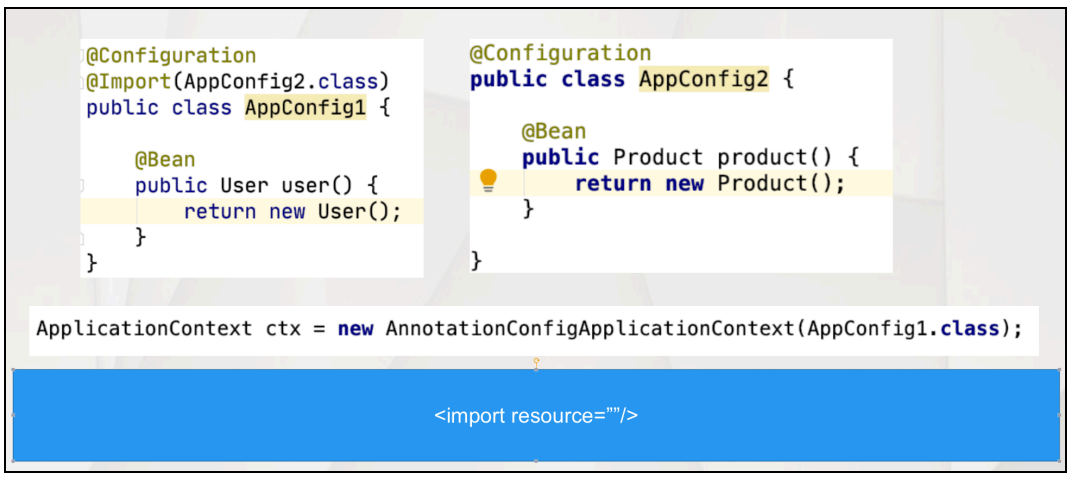
1. 多个配置Bean的整合

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



  - @Import

```
1   1. 可以创建对象
2   2. 多配置bean的整合
```



  - 在工厂创建时，指定多个配置Bean的Class对象 【了解】

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

- 跨配置进行注入

```
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。
@Configuration
@Import(AppConfig2.class)
public class AppConfig1 {

    @Autowired
    private UserDAO userDAO;

    @Bean
    public UserService userService() {
        UserServiceImpl userService = new UserServiceImpl();
        userService.setUserDAO(userDAO);
        return userService;
    }
}

@Configuration
public class AppConfig2 {

    @Bean
    public UserDAO userDAO() {
        return new UserDAOImpl();
    }
}
```

2. 配置Bean与@Component相关注解的整合

```
@Component(@Repository)
public class UserDAOImpl implements UserDAO{

}

@Configuration
@ComponentScan("")
public class AppConfig3 {

    @Autowired
    private UserDAO userDAO;

    @Bean
    public UserService userService() {
        UserServiceImpl userService = new UserServiceImpl();
        userService.setUserDAO(userDAO);
        return userService;
    }
}

```

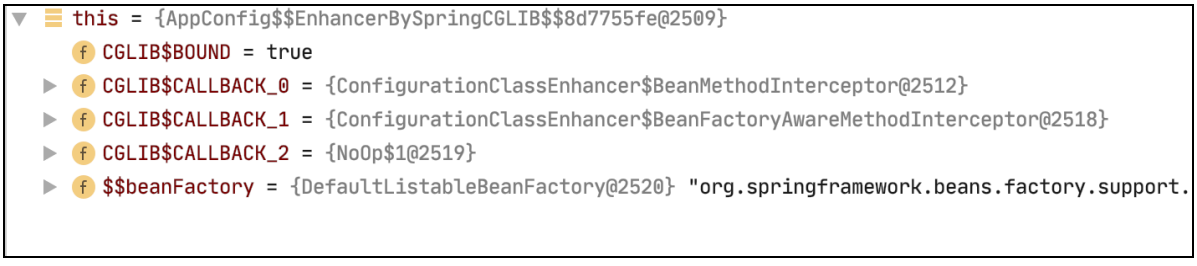
```
ApplicationContext ctx = new
AnnotationConfigApplicationContext(AppConfig3.class);
```

3. 配置Bean与配置文件整合

```
1. 遗留系统的整合 2. 配置覆盖

public class UserDAOImpl implements UserDAO{

}
<bean id="userDAO" class="com.baizhiedu.injection.UserDAOImpl"/>

@Configuration
@ImportResource("applicationContext.xml")
public class AppConfig4 {

    @Autowired
    private UserDAO userDAO;

    @Bean
    public UserService userService() {
        UserServiceImpl userService = new UserServiceImpl();
        userService.setUserDAO(userDAO);
        return userService;
    }
}

ApplicationContext ctx = new
AnnotationConfigApplicationContext(AppConfig4.class);
```

6. 配置Bean底层实现原理

```
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
```



7. 四维一体的开发思想

1. 什么是四维一体

```
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。
1. 基于schema
2. 基于特定功能注解
3. 基于原始<bean
4. 基于@Bean注解
```

2. 四维一体的开发案例

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

8. 纯注解版AOP编程

1. 搭建环境

```
1. 应用配置Bean
2. 注解扫描
```

2. 开发步骤

```
1. 原始对象
   @Service(@Component)
   public class UserServiceImpl implements UserService{

   }
2. 创建切面类  (额外功能 切入点 组装切面)
    @Aspect
    @Component
    public class MyAspect {

        @Around("execution(* login(..))")
        public Object arround(ProceedingJoinPoint joinPoint) throws
Throwable {

            System.out.println("----aspect log ------");

            Object ret = joinPoint.proceed();


            return ret;
        }
    }
3. Spring的配置文件中
   <aop:aspectj-autoproxy />
   @EnableAspectjAutoProxy ---> 配置Bean
```

3. 注解AOP细节分析

```
1. 代理创建方式的切换 JDK Cglib
   <aop:aspectj-autoproxy proxy-target-class=true|false />
   @EnableAspectjAutoProxy(proxyTargetClass)
2. SpringBoot AOP的开发方式
     @EnableAspectjAutoProxy 已经设置好了

    1. 原始对象
     @Service(@Component)
     public class UserServiceImpl implements UserService{

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

```
      </property>
    </bean>

    @Bean
    public SqlSessionFactoryBean sqlSessionFactoryBean(DataSource dataSource){
        SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();
        sqlSessionFactoryBean.setDataSource(dataSource);
        sqlSessionFactoryBean.setTypeAliasesPackage("");
        ...
        return sqlSessionFactoryBean;
    }

3. MapperScannerConfigure
   <!--创建DAO对象 MapperScannerConfigure-->
  <bean id="scanner" class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactoryBean"></property>
    <property name="basePackage" value="com.baizhiedu.dao"></property>
  </bean>

  @MapperScan(basePackages={"com.baizhiedu.dao"}) ---> 配置bean完成

```

- 编码

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

1. MapperLocations编码时通配的写法

```
//设置Mapper文件的路径
sqlSessionFactoryBean.setMapperLocations(Resource..);
Resource resouce = new ClassPathResouce("UserDAOMapper.xml")

sqlSessionFactoryBean.setMapperLocations(new ClassPathResource("UserDAOMapper.xml"));

<property name="mapperLocations">
   <list>
     <value>classpath:com.baizhiedu.mapper/*Mapper.xml</value>
   </list>
</property>
一组Mapper文件

ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
Resource[] resources = resolver.getResources("com.baizhi.mapper/*Mapper.xml");
sqlSessionFactoryBean.setMapperLocations(resources)
```

2. 配置Bean数据耦合的问题

```
mybatis.driverClassName = com.mysql.jdbc.Driver
mybatis.url = jdbc:mysql://localhost:3306/suns?useSSL=false
mybatis.username = root
mybatis.password = 123456
mybatis.typeAliasesPackages = com.baizhiedu.mybatis
mybatis.mapperLocations = com.baizhiedu.mapper/*Mapper.xml

@Component
@PropertySource("classpath:mybatis.properties")
public class MybatisProperties {
    @Value("${mybatis.driverClassName}")
    private String driverClassName;
    @Value("${mybatis.url}")
    private String url;
    @Value("${mybatis.username}")
    private String username;
    @Value("${mybatis.password}")
    private String password;
    @Value("${mybatis.typeAliasesPackages}")
    private String typeAliasesPackages;
    @Value("${mybatis.mapperLocations}")
    private String mapperLocations;
}

public class MyBatisAutoConfiguration {

    @Autowired
    private MybatisProperties mybatisProperties;

    @Bean
    public DataSource dataSource() {
        DruidDataSource dataSource = new DruidDataSource();

 dataSource.setDriverClassName(mybatisProperties.getDriverClassNa
me());
        dataSource.setUrl(mybatisProperties.getUrl());
        dataSource.setUsername(mybatisProperties.getUsername());
        dataSource.setPassword(mybatisProperties.getPassword());
        return dataSource;
    }

    @Bean
    public SqlSessionFactoryBean sqlSessionFactoryBean(DataSource
dataSource) {
        SqlSessionFactoryBean sqlSessionFactoryBean = new
SqlSessionFactoryBean();
        sqlSessionFactoryBean.setDataSource(dataSource);

 sqlSessionFactoryBean.setTypeAliasesPackage(mybatisProperties.ge
tTypeAliasesPackages());
        //sqlSessionFactoryBean.setMapperLocations(new
ClassPathResource("UserDAOMapper.xml"));

```

```
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
```

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

```
1. ApplicationContext ctx = new AnnotationConfigApplicationContext("com.baizhiedu.mybatis");
   SpringBoot 实现思想
2. 注解版MVC整合，SpringMVC中进行详细讲解
   SpringMyBatis --->DAO  事务基于注解 --> Service   Controller
   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

```
YML(YAML)是一种新形式的配置文件，比XML更简单，比Properties更强大。

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

3. YML语法简介

```
1. 定义yml文件
   xxx.yml xxx.yaml
2. 语法
   1. 基本语法
      name: suns
      password: 123456
   2. 对象概念
      account:
         id: 1
         password: 123456
   3. 定义集合
      service:
         - 11111
         - 22222
```

4. Spring与YML集成思路的分析

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

5. Spring与YML集成编码

- 环境搭建

```
<dependency>
  <groupId>org.yaml</groupId>
  <artifactId>snakeyaml</artifactId>
  <version>1.23</version>
</dependency>
最低版本 1.18
```

- 编码

```
1. 准备yml配置文件
2. 配置Bean中操作 完成YAML读取 与 PropertySourcePlaceholderConfigure的创建
    @Bean
    public PropertySourcesPlaceholderConfigurer configurer() {
        YamlPropertiesFactoryBean yamlPropertiesFactoryBean = new YamlPropertiesFactoryBean();
        yamlPropertiesFactoryBean.setResources(new ClassPathResource("init.yml"));
        Properties properties = yamlPropertiesFactoryBean.getObject();

        PropertySourcesPlaceholderConfigurer configurer = new PropertySourcesPlaceholderConfigurer();
        configurer.setProperties(properties);
        return configurer;
    }
3. 类 加入 @Value注解
```

6. Spring与YML集成的问题

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

SpringBoot  @ConfigurationProperties
```