spring注解开发总结

1.搭建开发环境用到的注解

1.@ComponentScan--相关注解扫描

```
@Configuration
@ComponentScan(basePackages = "com.baizhiedu.scan")
public class AppConfig2 {
}
<context:component-scan base-package=""/>
```

- 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
```

2.@Configuration---替换spring配置文件 <beans

- ```
  Spring在3.x提供的新的注解，用于替换XML配置文件。
  此注解会告诉spring这是配置类
  此步骤相当于创建了一个applicationContext.xml文件
   @Configuration
  public class AppConfig{

  }
  ```

- AnnotationConfigApplicationContext
  ```

```
1. 创建工厂代码
 ApplicationContext ctx = new
AnnotationConfigApplicationContext();
2. 指定配置文件（两种方式）
 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能替换以前xml文件的什么呢?

```
1.创建对象
2.注入
3.定义注解的扫描
4.引入其他配置文件
```

- 注意的细节

```
不能集成 log4j
可以集成logback
logback：1.导入相关依赖 2.引入logback配置文件
```

- @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.对象创建相关注解

1.@Component

1. **搭建开发环境**

   - [ ] 想让spring框架在这时包及其自保重扫描对应的注解，使其生效需要在spring配置文件中写入：

   ```
   <context:component-scan base-package="要扫描的包"/>
   ```

   - [x] @ComponentScan

2. **@Component** 相当于以前spring配置文件中的<bean 标签

   ```
   <bean  id=""  class=""  />
   id 属性 @Component提供默认id---> 首单词字母小写（UserDao--->userDao）
   class 属性 @Component写在哪个class上会通过反射获得
   ```

3. **@Component("指定id")** 可以通过此种方式指定id
   ```

4. **@Component的衍生注解**

```
@Repository   用于Dao层（spring与mybatis整合后中不使用注解创建对象）
@Service      用于service层
@Controller   用于controller层
```

5. **@Scope注解** ----- 控制简单对象的创建次数

   默认为 singleton 单例模式

6. **@Lazy注解** ----- 延迟创建单例对象

   一旦使用了@Lazy注解后，spring会在使用这个对象的时候，进行对象的创建

7. **生命周期**相关注解

```
1.初始化相关方法@PostConstruct
2.销毁方法       @DisposableBean
这两个方法并不是spring提供的，是JSR520提供
```

2.@Bean注解

- 相当于<bean标签
- 基本使用：

```
@Bean
Public User user(){       //User--创建对象类型   user 方法名==id属性
    return new User();  //把对象的创建代码写在方法体中
}
// 大前提得在配置bean里（有@Configuration）
// 自定义id值 ---  @Bean("id")
// 控制创建次数 --- @Bean("singleton|prototype") 默认单例
```

- 提出疑问：class怎么对应，如果不同包下同类名怎么办？？

```
idea会让你选是在哪个包下
return new com.it.Dog();
```

3.@Import 了解即可 （spring底层使用）

4.这三种什么时候使用?

```
1.@Component @Autowired  用于程序员自己开发的类；
        例如：UserService UserDao 等
2.@Bean 多用于复杂对象，框架提供的类型，别的程序员开发的类型（没有源码）
        例如：SqlSessionFactoryBean
3.<bean id="" class="" /> 一般不使用，多用于遗留系统的整合
```

5.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
```

6.基于注解配置的耦合问题

```
原有的开发方式：
@Configuration
public class AppConfig4 {
 @Bean
 public UserDAO userDAO() {
 return new UserDAOImpl();
 }
}
解决办法一：在这个配置bean上添加（有耦合）@ImportResource("applicationContext.xml")

解决办法二：新建一个配置bean
    @Configuration
@ImportResource("applicationContext.xml")
public class AppConfig5{

}
 ApplicationContext ctx = new
AnnotationConfigApplicationContext(AppConfig.class，新建的配置bean);
```

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

3.注入相关注解

1.1用户自定义类型@Autowired

```
@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
```

1.2.JDK类型@value

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

注意：

- @value不能应用在静态成员变量上
  @value+properties这种方式，不能注入集合类型（下面会引入YAML）

2.1 @Bean 用户自定义类型注入

- 把userDao 注入到 userService中

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

2.2@Bean jdk类型注入

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

  注意：如果直接用set方法，会存在耦合问题
  ```

- 配合配置文件使用

```
@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;
 }
}
```

4.整合多个配置信息

按照功能会分成多个配置bean（模块化编程的思想，面向对象各司其职）

- 如何使多种配置信息 汇总成一个整体？？
- 如何实现跨配置的注入？？？

1.配置bean与配置bean整合

- 曾经xml文件的整合方式

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("application-
*.xml");
```

- 现在整合配置bean的方式1---基于包扫描

```
 ApplicationContext ctx = new
AnnotationConfigApplicationContext("com.config");
```

- 方式2 --- @Import ---

```
@Configuration
@Import(AppConfig2.class)
public class AppConfig1{
    @Bean
    public User user(){
        return new User();
    }
}

ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig1);
```

- 方式3 ---- 创建工厂时指定多个bean的class对象【不推荐使用】

```
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整合

只需要加上注解的扫描!

```
@Component或@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spring配置文件整合

- 配置覆盖
- 遗留系统的整合

```
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);
```

5.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
```

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
```

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）
     @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;
 }
 }
 Spring AOP 代理默认实现 JDK SpringBOOT AOP 代理默认实现 Cglib
```

纯注解版spring+mybatis整合

基础配置（配置Bean)

```
1. 连接池
 <!--连接池-->
 <bean id="dataSource"
class="com.alibaba.druid.pool.DruidDataSource">
 <property name="driverClassName"
value="com.mysql.jdbc.Driver"></property>
 <property name="url" value="jdbc:mysql://localhost:3306/suns?
useSSL=false"></property>
 <property name="username" value="root"></property>
 <property name="password" value="123456"></property>
 </bean>
 ------------------------------------------------
 @Bean
```

```
 public DataSource dataSource(){
 DruidDataSource dataSource = new DruidDataSource();
 dataSource.setDriverClassName("");
 dataSource.setUrl();
 ...
 return dataSource;
 }
--------------------------------------------------
--------------------------------------------------
2. SqlSessionFactoryBean
 <!--创建SqlSessionFactory SqlSessionFactoryBean-->
 <bean id="sqlSessionFactoryBean"
class="org.mybatis.spring.SqlSessionFactoryBean">
 <property name="dataSource" ref="dataSource"></property>
 <property name="typeAliasesPackage"
value="com.baizhiedu.entity"></property>
 <property name="mapperLocations">
 <list>
<value>classpath:com.baizhiedu.mapper/*Mapper.xml</value>
 </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

- 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)
```

配置Bean数据耦合问题

配置文件:

```
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
```

使用配置文件

```
@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"));
try {
 ResourcePatternResolver resolver = new
PathMatchingResourcePatternResolver();
 Resource[] resources =
resolver.getResources(mybatisProperties.getMapperLocations());
 sqlSessionFactoryBean.setMapperLocations(resources);
 } catch (IOException e) {
 e.printStackTrace();
 }
 return sqlSessionFactoryBean;
 }
}
```

纯注解版事务编程

```
1. 原始对象 XXXService
 <bean id="userService"
class="com.baizhiedu.service.UserServiceImpl">
 <property name="userDAO" ref="userDAO"/>
 </bean>
-----------------------------------------------
@Service
 public class UserServiceImpl implements UserService{
 @Autowired
 private UserDAO userDAO;
 }
-----------------------------------------------
-----------------------------------------------
2. 额外功能
 <!--DataSourceTransactionManager-->
 <bean id="dataSourceTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManage
r">
 <property name="dataSource" ref="dataSource"/>
 </bean>
-----------------------------------------------
@Bean
```

```
 public DataSourceTransactionManager
dataSourceTransactionManager(DataSource dataSource){
 DataSourceTransactionManager dstm = new
DataSourceTransactionManager();
 dstm.setDataSource(dataSource);
 return dstm;
 }
-----------------------------------------------------
-----------------------------------------------------

3. 事务属性
@Transactional
 @Service
 public class UserServiceImpl implements UserService {
 @Autowired
 private UserDAO userDAO;
-----------------------------------------------------
-----------------------------------------------------
4. 基于Schema的事务配置
<tx:annotation-driven transaction-manager="dataSourceTransactionManager"/>
-------------------------------------------------
 @EnableTransactionManager ---> 配置Bean
```

细节:

```
1. ApplicationContext ctx = new
AnnotationConfigApplicationContext("com.baizhiedu.mybatis");
 SpringBoot 实现思想
2. 注解版MVC整合（目前无法实现）
 SpringMyBatis --->DAO 事务基于注解 --> Service Controller
 org.springframework.web.context.ContextLoaderListener ---> XML工厂 无
法提供 new AnnotationConfigApplicationContext
```

YML

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进行配置的问题

```
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继承思路的分析

```
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
  ```

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
```