《Spring IoC与注解》Spring_Day01

一、教学课题

1、能力养成

阅读过Spring的源码，了解Spring IoC、AOP、MVC的原理以及对 Spring 循环依赖、生命周期、事务有一定的研究。

2、Spring 课程安排

参见：《Spring知识大纲.xmind》

二、教学目标

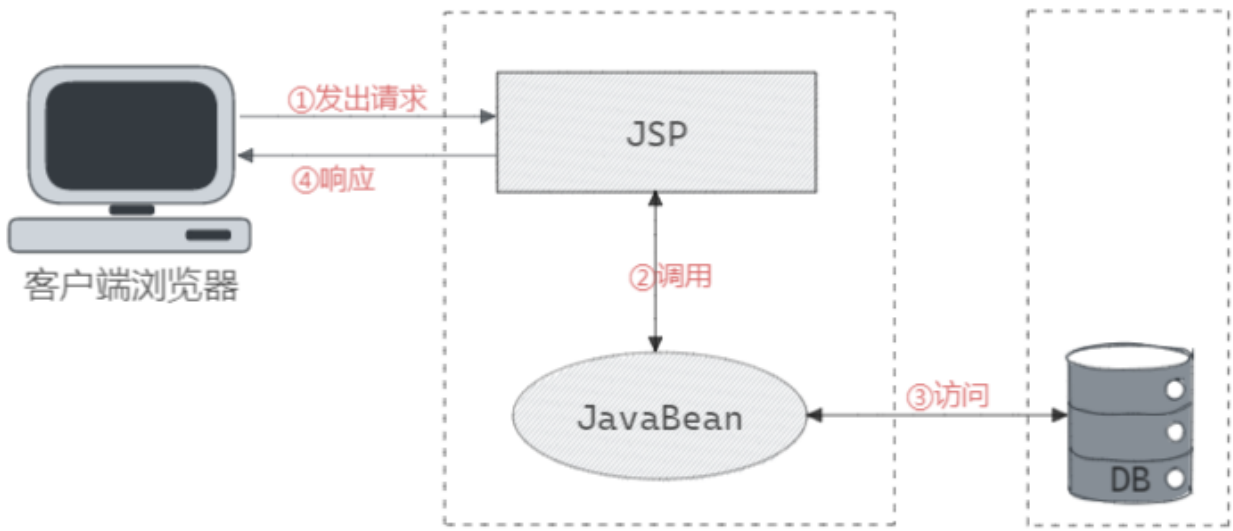
1. 了解 Spring 基本概念
2. 理解 IoC 思想
3. 理解 Spring IoC 工作原理
4. 掌握 Spring 常用注解
5. 掌握 Spring 自定义注解

三、教学过程

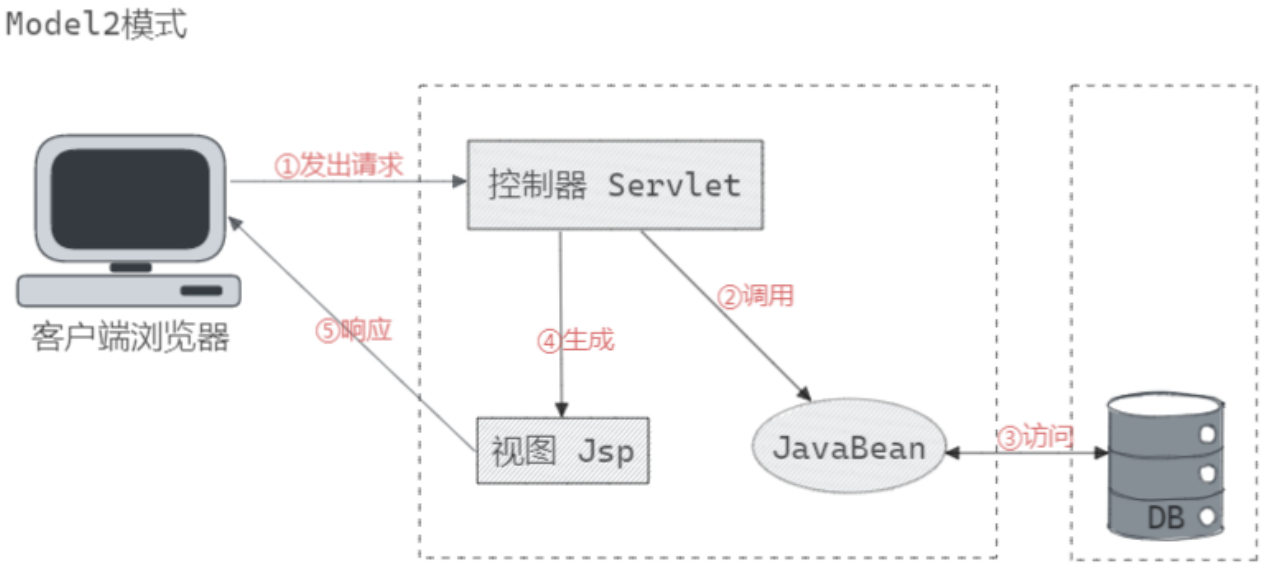
1、Java web开发技术的演变

- Modle1（JSP）

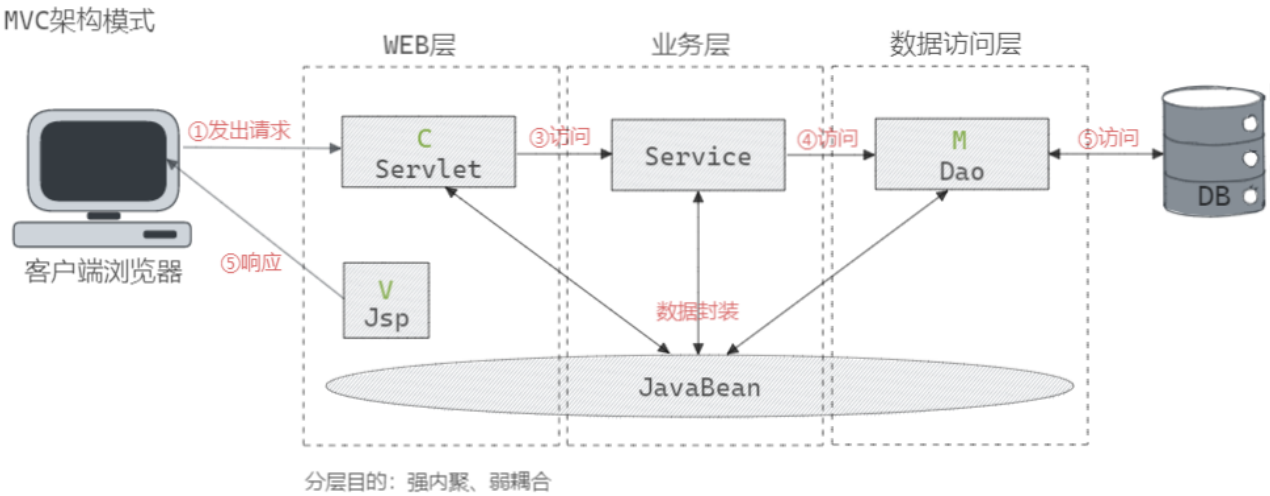
Model1模式



- Modle2（JSP+Servlet）



- MVC



- 思考和改进MVC
  - 会产生大量的对象
  - 单例模型
  - 池化技术（数据库连接池、线程池）
  - 创建一个池
    - 需要考虑的问题：哪些对象进去？怎样放进去？如何管理？对象的生命周期如何设定？

2、Spring是什么

Spring是一个轻量级的IoC和AOP的开源容器框架

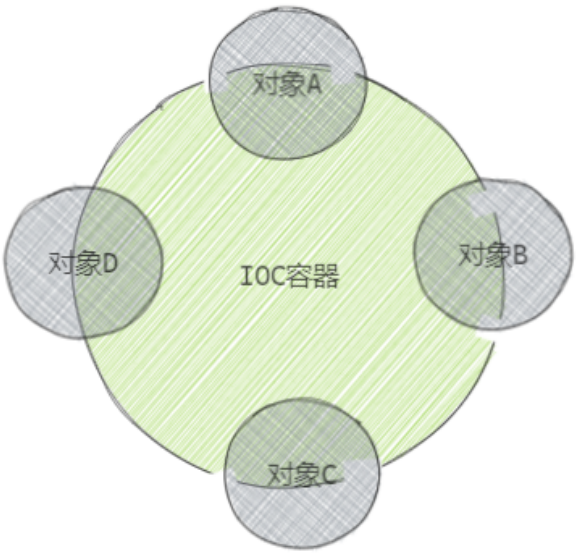
```
// 实例化（创建对象）
Object obj = new Object();
// 属性注入（DI）
obj.setXxx("");

// 初始化（在创建对象时 需要被调用的方法 init() | motheod()）
init();
// AOP（动态代理）
```

3、IoC[DI]

3-1、IoC思想

- 耦合问题（OOP ==> 无法避免对象间的耦合关系）
- 放大耦合问题（项目规模++ ==> 耦合关系越来越复杂）
- IoC容器（控制反转）（容器来帮我们管理bean 实现对象间的最大解耦）
  - XML、注解（常用）
  - 反射机制（工厂模式）
  - IoC容器 ==> 工厂（配置文件-对象）

对象间的耦合关系



- DI
  - 依赖注入，程序在运行时由IoC容器来动态注入对象需要的外部依赖
  - Spring 的 IoC 有两种方式（API层面）：一个是xml方式、另一种是注解的方式
  - 依赖注入的方式：构造方法注入、setter方法注入、接口注入

3-2、IoC接口

BeanFactory（getBean()）

ApplicationContext E BeanFactory

3-3、IoC实现

- IoC要做什么事情?
  - 哪些类的对象需要放入IoC容器?
  - 怎样去获得类对象的实例?
  - 怎样给对象实例的属性赋值?
  - 创建好的对象存入容器?

3-4、手写IoC源码分析

WEB-INF下的配置文件web.xml，入口就是dispatcherServlet，其init()方法在容器启动时就会被调用

3-4-1、加载配置文件

> application.properties文件

```
scanPackage=com.gupaoedu.vip.demo
```

> 进入`<servlet>`标签 GPDispatcherServlet - init() 方法

```
// com.gupaoedu.vip.spring.framework.webmvc.servlet.GPDispatcherServlet#init【Line: 110】
public void init(ServletConfig config) throws ServletException {
    // 读取config.getInitParameter("contextConfigLocation")的内容
    applicationContext = new
GPApplicationContext(config.getInitParameter("contextConfigLocation"));
}
```

> GPApplicationContext() 构造器

```
public GPApplicationContext(String ... configLocations) {
    //1、加载配置文件
    reader = new GPBeanDefinitionReader(configLocations);
}
```

> GPBeanDefinitionReader() 构造器

```
public GPBeanDefinitionReader(String... locations){
    //1、加载Properties文件
    doLoadConfig(locations[0]);
    //2、扫描相关的类
    doScanner(contextConfig.getProperty("scanPackage"));
}
```

> doLoadConfig() 方法

```
//根据contextConfigLocation的路径去ClassPath下找到对应的配置文件
private void doLoadConfig(String contextConfigLocation) {
    InputStream is = this.getClass().getClassLoader().getResourceAsStream(
        contextConfigLocation.replaceAll("classpath:",""));
    // ...
}
```

> doScanner() 方法

```
//扫描ClassPath下符合包路径规则所有的Class文件
private void doScanner(String scanPackage) {
    URL url = this.getClass().getClassLoader().getResource("/" +
scanPackage.replaceAll("\\.","/"));
    File classPath = new File(url.getFile());
    // ...
}
```

到这里，就找的了所有要加载到容器中的类，并存入list中。

3-4-2、封装成BeanDefinition

将扫描到的类全部封装成BeanDefinition并保存，BeanDefinition中有类的名称和类的全名称，将来反射机制可以通过全名称实例化对象

```
public GPApplicationContext(String ... configLocations) {
    // ...
    try {
        // 2、解析配置文件，将所有的配置信息封装成BeanDefinition对象
        List<GPBeanDefinition> beanDefinitions = reader.loadBeanDefinitions();
        // ...
    }catch (Exception e){ }
}
```

> loadBeanDefinitions() 方法

```
public List<GPBeanDefinition> loadBeanDefinitions(){
    List<GPBeanDefinition> result = new ArrayList<GPBeanDefinition>();
    // ...
    return result;
}
```

> doRegistBeanDefinition() 方法

```
public void doRegistBeanDefinition(List<GPBeanDefinition> beanDefinitions) throws Exception
{
    // 如果已经存在了，就不放入里面并且抛出异常
    for (GPBeanDefinition beanDefinition : beanDefinitions) {
        if(this.beanDefinitionMap.containsKey(beanDefinition.getFactoryBeanName())){
            throw new Exception("The " + beanDefinition.getFactoryBeanName() + " is
exists!!!");
        }
        this.beanDefinitionMap.put(beanDefinition.getFactoryBeanName(),beanDefinition);
    }
}
```

3-4-3、实例化这些类

> GPApplicationContext() 构造器

```
public GPApplicationContext(String ... configLocations) {
    try{
        // ...
        // 4、加载非延时加载的所有的Bean
        doLoadInstance();

    }catch (Exception e){
        e.printStackTrace();
    }
}
```

> doLoadInstance() 方法

```
private void doLoadInstance() {
    //循环调用getBean()方法
    for (Map.Entry<String,GPBeanDefinition> entry :
this.registry.beanDefinitionMap.entrySet()) {
        String beanName = entry.getKey();
        if(!entry.getValue().isLazyInit()) {
            getBean(beanName);
        }
    }
}
```

> getBean() 方法

```
@Override
public Object getBean(String beanName) {
    //1、先拿到BeanDefinition配置信息
    GPBeanDefinition beanDefinition = registry.beanDefinitionMap.get(beanName);

    //2、反射实例化对象
    Object instance = instantiateBean(beanName,beanDefinition);

    //3、将返回的Bean的对象封装成BeanWrapper
```

```
        GPBeanWrapper beanWrapper = new GPBeanWrapper(instance);

        //4、执行依赖注入
        populateBean(beanName,beanDefinition,beanWrapper);

        //5、保存到IoC容器中
        this.factoryBeanInstanceCache.put(beanName,beanWrapper);

        return beanWrapper.getWrappedInstance();
    }
```

一般来讲我们的成员变量注入其他的对象都是注入的是接口，所以接口的全名称也被保存。

4、注解

注解使用部分，比较基础，补充一些基础说明，大家参考

@Component

> 称之为Spring的注解，可以被ComponentScan扫描到

@ComponentScan

> **使用方法**
>
> @ComponentScan(value="包名")扫描该包下的类，扫描添加了@Component注解的类
>
> ComponentScan是一个可以重复配置的注解，可重复的意思是
>
> ```
> @ComponentScans({
>   @ComponentScan("com.gupaoedu.project.entity"),
>   @ComponentScan("com.gupaoedu.demo.annotaions.configures.lifecycle")
> })
> public class MyConfig {
> }
> ```
>
> 如果不加Configuration注解,则无法扫描到类，想要扫描到类，可以改为
>
> ```
> @ComponentScan("com.gupaoedu.project.entity")
> public class MyConfig {
> }
> ```
>
> **includeFilters**可以指定注解，指定类型，自定义规则等，表示加载相应的加了这个注解的类
>
> 指定注解：includeFilters = {@Filter(type = FilterType.ANNOTATION,value = {Controller.class})}
>
> 指定类型：includeFilters = {@Filter(type = FilterType.ASSIGNABLE_TYPE,value = {MyController.class})}
>
> 自定义规则：includeFilters = {@Filter(type = FilterType.CUSTOM,value = {GPTypeFilter.class})}
>
> 使用自定义规则时，设置useDefaultFilters = false，不使用默认的过滤器，才有效果

> **excludeFilters**可以指定注解，指定类型，自定义规则等，表示排除相应的加了这个注解的类，与includeFilters相反

@Configuration和@Bean

> @Configuration和@Bean注解配套使用，方法加了一个这个@Bean注解，一般这样的方法返回的都是一个对象，我们就可以直接从这个类中拿到这个实例，并且是单例的，我们可以通过AnnotationApplicationContext context = new AnnotationApplicationContext(加了@Configuration注解的类)去获取到这个bean对象

```
@Configuration
public class MyConfig {

@Bean
public Person person(){
  return new Person("Tom",18);
}

}
```

```
public class MyTest {
@Test
public void test() {
  ApplicationContext app = new AnnotationConfigApplicationContext(MyConfig.class);
  Object bean = app.getBean("person");

  String [] beanNames = app.getBeanNamesForType(Person.class);
  System.out.println(Arrays.toString(beanNames));

}
}
```

如果不加@Configuration，然后类中有

```
@Configuration
public class MyConfig {

@Bean
public Person person1(){
  return new Person("Tom",18);
}
// 添加一个方法
@Bean
public Person person(){
  return person1();
}
}
```

```
// 比对两个bean
public class MyTest {
@Test
public void test() {
```

```
    ApplicationContext app = new AnnotationConfigApplicationContext(MyConfig.class);
    Object bean = app.getBean("person");
    Object bean1 = app.getBean("person1");
    System.out.println(bean==bean1);

    String [] beanNames = app.getBeanNamesForType(Person.class);
    System.out.println(Arrays.toString(beanNames));

}
}
```

会发现bean==bean1为false,因为第一次bean注解的实现方式是通过反射，方法名.invoke(类名,参数)得到的对象，方法person()被调用时是通过new Person("Tom",18)得到一个对象，方法person1()被调用又new Person("Tom",18)一下这个对象，因此两个不相等

如果加了Configuration则通过代理类的interceptor方法直接去容器中拿生成过的对象。因此只会有一个对象，那么就相等了

@Scope

@Scope("prototype")//多实例，IOC容器启动创建的时候，并不会创建对象放在容器在容器当中，当你需要的时候，需要从容器当中取该对象的时候，就会创建。

@Scope("singleton")//单实例 IOC容器启动的时候就会调用方法创建对象，以后每次获取都是从容器当中拿同一个对象（map当中）。
@Scope("request")//同一个请求创建一个实例
@Scope("session")//同一个session创建一个实例

@Lazy

延迟初始化

@Conditional(WinCondition.class)

条件主键，里面的类实现Condition，然后实现matches方法，返回的boolean值，如果为true,则执行该方法

@Primary

先定义一个Service,然后给该Service定义两个实现类

```
public interface MyService {

public void print();
}
```

```
@Service("myServiceImpl")
public class MyServiceImpl implements MyService {

@Autowired
private MyDao myDao;

public void print(){
  System.out.println(myDao);
}
}
```

```
@Service("yourServiceImpl")
public class YourServiceImpl implements MyService {

@Autowired
private MyDao myDao;

public void print(){
  System.out.println(myDao);
}
}
```

然后在Controller中引入该Service

```
@Controller
public class MyController {
@Autowired private MyService service;
}
```

定义一个Config配置类

```
@Configuration
@ComponentScan({
  "com.gupaoedu.project.controller",
  "com.gupaoedu.project.service",
  "com.gupaoedu.project.dao"
      })
public class MyConfig {


//    @Bean
//    @Primary
//    public MyService myService(){
//        return new MyServiceImpl();
//    }
//
//    @Bean
//    public MyService youService(){
//        return new YourServiceImpl();
//    }
```

```
}
```

在测试

```
public class MyTest {
@Test
public void test(){
  ApplicationContext app = new AnnotationConfigApplicationContext(MyConfig.class);

  MyService myService = app.getBean(MyService.class);

  System.out.println(myService);


}
}
```

报错如下：expected single matching bean but found 2: myServiceImpl,yourServiceImpl

解决方法：在配置类中分别配置两个Service的Bean，然后使用Primary注解指定默认使用哪一个

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

@Bean
public MyService youService(){
return new YourServiceImpl();
}
```

第二种方法就是

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

@Bean
public MyService youService(){
return new YourServiceImpl();
}
```

在Controller类中做修改

```
@Controller
public class MyController {
@Autowired
@Qualifier(value = "myServiceImpl")
private MyService service;
public String test(){
  return service.print();
}
}
```

测试

```
public class MyTest {
@Test
public void test(){
  ApplicationContext app = new AnnotationConfigApplicationContext(MyConfig.class);


  MyController service = app.getBean(MyController.class);
  System.out.println(service.test());

}
}
```

@Qualifier

@Qualifier 注解作用就是为了给Bean打上一个标记，用来查找bean

```
@Configuration
@ComponentScan({
  "com.gupaoedu.project.controller",
  "com.gupaoedu.project.service",
  "com.gupaoedu.project.dao"
      })
public class MyConfig {

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

@Qualifier
@Bean
public MyService youService(){
  return new YourServiceImpl();
}

}
```

定义一个Controller

```
@Controller
public class MyController {
@Autowired
@Qualifier(value = "yourServiceImpl")
private MyService service;

@Qualifier
@Autowired
private List<MyService> list = new ArrayList<MyService>();

public String test(){
  return service.print();
}

public void getList(){
  System.out.println("list"+list.toString());
}
}
```

测试类

```
public class MyTest {
@Test
public void test(){
  ApplicationContext app = new AnnotationConfigApplicationContext(MyConfig.class);


  MyController service = app.getBean(MyController.class);
  service.getList();

}
}
```

可以看到，通过Qualifier注解，我们可以找到打上了Qualifier注解标记的并，并且保存到list中

@PropertySource

就是把注解中配置的文件中的key-value的数据映射到具体的实体对象上

@Import导入外部资源

1.新建一个TestA

```
public class TestA {

public void fun(String str) {
  System.out.println(str);
}

public void printName() {
  System.out.println("类名 : " +Thread.currentThread().getStackTrace()
[1].getClassName());
}
}
```

2.新建一个ImportConfig,在类上面加上@Configuration，加上@Configuration是为了能让Spring 扫描到这个类，并且直接通过@Import引入TestA类

```
@Import({TestA.class})
@Configuration
public class ImportConfig {
}
```

ImportSelector 的实现

1.新建TestC.class

```
public class TestC {
public void fun(String str) {
  System.out.println(str);
}

public void printName() {
  System.out.println("类名 : " + Thread.currentThread().getStackTrace()
[1].getClassName());
}
}
```

2.新建SelfImportSelector.class 实现ImportSelector 接口,注入TestC.class

```
public class SelfImportSelector implements ImportSelector {
@Override
public String[] selectImports(AnnotationMetadata importingClassMetadata) {
  return new String[]{"com.test.importdemo.TestC"};
}
}
```

3.ImportConfig上面引入SelfImportSelector.class

```
@Import({TestA.class,TestB.class,SelfImportSelector.class})
@Configuration
public class ImportConfig {
}
```

ImportBeanDefinitionRegistrar

1.新建TestD.class

```
public class TestD {
public void fun(String str) {
  System.out.println(str);
}

public void printName() {
  System.out.println("类名 : " + Thread.currentThread().getStackTrace()
[1].getClassName());
}
}
```

2.新建SelfImportBeanDefinitionRegistrar.class,实现接口ImportBeanDefinitionRegistrar,注入TestD.class

```
public class SelfImportBeanDefinitionRegistrar implements
ImportBeanDefinitionRegistrar {
@Override
public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
BeanDefinitionRegistry registry) {
  RootBeanDefinition root = new RootBeanDefinition(TestD.class);
  registry.registerBeanDefinition("testD", root);
}
}
```

3.ImportConfig类上加上导入SelfImportBeanDefinitionRegistrar.class

```
@Import({TestA.class,TestB.class,SelfImportSelector.class,
  SelfImportBeanDefinitionRegistrar.class})
@Configuration
public class ImportConfig {
}
```

```
@Autowired
TestD testD;

@Test
public void TestD() {
testD.printName();
}
```

@DependsOn

@DependsOn注解可以用来控制bean的创建顺序，该注解用于声明当前bean依赖于另外一个bean。所依赖的bean会被容器确保在当前bean实例化之前被实例化。

```
@Configuration
public class MyConfig {

@Bean
@DependsOn("beanB")
public BeanA beanA(){
  System.out.println("beanA================");
  return new BeanA();
}

@Bean
@DependsOn("beanC")
public BeanB beanB(){
  System.out.println("beanB================");
  return new BeanB();
}

@Bean
public BeanC beanC(){
  System.out.println("beanC================");
  return new BeanC();
}
}
```

测试类

```
public class Test {

public static void main(String[] args) {
  ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(MyConfig.class);

}
}
```

自定义注解

先定义类上面的注解

```java
// 表示作用在类上
@Target({ElementType.TYPE})
// 运行时的注解
@Retention(RetentionPolicy.RUNTIME)
// @Documented 将此注解包含在javadoc 中
@Documented
// 代表是spring的注解，会被spring扫描到
@Component
public @interface GupaoAnnotionClass {

    boolean value() default false;
}
```

```java
// 表示作用在方法上
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface GupaoAnnotionMethod {

    String value() default "wudasheng";
}
```

```java
@GupaoAnnotionClass
public class FourClass {

    @GupaoAnnotionMethod(value="fourClass")
    public void study(){
        System.out.println("study spring");
    }
}
```

```java
@Configuration
@ComponentScans({
        @ComponentScan("com.gupaoedu.project.gupaoentity"),
        @ComponentScan("com.gupaoedu.demo.annotaions.configures.gupaoannotion")
})
public class MyConfig {

}
```

```java
public class MyTest {
    @Test
    public void test() {
        ApplicationContext app = new AnnotationConfigApplicationContext(MyConfig.class);

        FourClass fourClass = (FourClass)app.getBean("fourClass");
        fourClass.study();
    }
}
```

```
@Component
public class GupaoBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        Object instance = null;
        if(bean.getClass().isAnnotationPresent(GupaoAnnotionClass.class)){
            GupaoAnnotionClass annotation =
bean.getClass().getAnnotation(GupaoAnnotionClass.class);
            if(annotation.value()){
                return bean;
            }
            Method[] declaredMethods = bean.getClass().getDeclaredMethods();
            for(Method method : declaredMethods){
                if(method.isAnnotationPresent(GupaoAnnotionMethod.class)) {
                    GupaoAnnotionMethod annotionMethod =
method.getAnnotation(GupaoAnnotionMethod.class);
                    if("fourClass".equals(annotionMethod.value())){
                        instance = getInstance(bean.getClass());
                    }
                }
            }
        }
        return instance;
    }

    public Object getInstance(Class<?> clazz) throws BeansException {
        //相当于Proxy，代理的工具类
        // 生成的代码是代理类的子类
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(clazz);
        enhancer.setCallback(new MethodInterceptor() {
            @Override
            public Object intercept(Object o, Method method, Object[] objects, MethodProxy
methodProxy) throws Throwable {
                System.out.println("准备开始学习");
                Object obj = methodProxy.invokeSuper(o,objects);
                System.out.println("一天就学会了");
                return obj;
            }
        });
        return enhancer.create();
    }
}
```

自定义注解使用很简单，主要在于需要处理的业务是否复杂。

四、课后总结

1. 理解 IoC 思想 及其 工作原理
2. Spring 注解的使用

五、下节预告

《Spring 循环依赖与Bean 生命周期》