1.介绍:

注解的原理:

注解本质是一个继承了Annotation 的特殊接口，其具体实现类是Java 运行时生成的动态代理类。而我们通过反射获取注解时，返回的是Java 运行时生成的动态代理对象$Proxy1。通过代理对象调用自定义注解（接口）的方法，会最终调用AnnotationInvocationHandler 的invoke 方法。该方法会从memberValues 这个Map 中索引出对应的值。而memberValues 的来源是Java 常量池。

元注解:

java.lang.annotation 提供了四种元注解，专门注解其他的注解（在自定义注解的时候，需要使用到元注解）：

@Documented – 注解是否将包含在JavaDoc中

@Retention – 什么时候使用该注解

@Target – 注解用于什么地方

@Inherited – 是否允许子类继承该注解

1.）@Retention – 定义该注解的生命周期

- RetentionPolicy.SOURCE：在编译阶段丢弃。这些注解在编译结束之后就不再有任何意义，所以它们不会写入字节码。@Override, @SuppressWarnings都属于这类注解。

- RetentionPolicy.CLASS：在类加载的时候丢弃。在字节码文件的处理中有用。注解默认使用这种方式
- RetentionPolicy.RUNTIME：始终不会丢弃，运行期也保留该注解，因此可以使用反射机制读取该注解的信息。我们自定义的注解通常使用这种方式。

2.）Target – 表示该注解用于什么地方。默认值为任何元素，表示该注解用于什么地方。可用的ElementType 参数包括

- ElementType.CONSTRUCTOR: 用于描述构造器
- ElementType.FIELD: 成员变量、对象、属性（包括enum实例）
- ElementType.LOCAL_VARIABLE: 用于描述局部变量
- ElementType.METHOD: 用于描述方法
- ElementType.PACKAGE: 用于描述包
- ElementType.PARAMETER: 用于描述参数
- ElementType.TYPE: 用于描述类、接口(包括注解类型) 或enum声明

3.)@Documented – 一个简单的Annotations 标记注解，表示是否将注解信息添加在java 文档中。

4.)@Inherited – 定义该注释和子类的关系

@Inherited 元注解是一个标记注解，@Inherited 阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited 修饰的annotation 类型被用于一个class，则这个annotation 将被用于该class 的子类。

自定义注解：

自定义注解类编写的一些规则:

1. Annotation 型定义为@interface, 所有的Annotation 会自动继承java.lang.Annotation这一接口,并且不能再去继承别的类或是接口.
2. 参数成员只能用public 或默认(default) 这两个访问权修饰
3. 参数成员只能用基本类型byte、short、char、int、long、float、double、boolean八种基本数据类型和String、Enum、Class、annotations等数据类型，以及这一些类型的数组.
4. 要获取类方法和字段的注解信息，必须通过Java的反射技术来获取 Annotation 对象，因为你除此之外没有别的获取注解对象的方法
5. 注解也可以没有定义成员,，不过这样注解就没啥用了

PS:自定义注解需要使用到元注解

2.使用:

方法一： 指定处理类(常用)

使用@Constraint注解,指定处理类

一：

定义注解，用于入参时判断枚举

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD,ElementType.PARAMETER})
@Documented
@Constraint(validatedBy=EnumDeal.class)// validatedBy 接收的是数组,如果有多个,可以用{}包含处理类
public @interface EnumCheck {
    /**
        * 枚举之外的值
        */
       String[] value() default {};

       /**
        * 枚举类
        */
       Class<? extends Enum<?>> enumClass();
 /**
        * 用来判断的方法名,默认 getEnum
        * <p>方法入参类型应当与当前判断属性类型一致</p>
        */
       String enumMethod() default "getEnum";

   String message() default "";
       Class<?>[] groups() default { };
   Class<? extends Payload>[] payload() default { };
  }
```

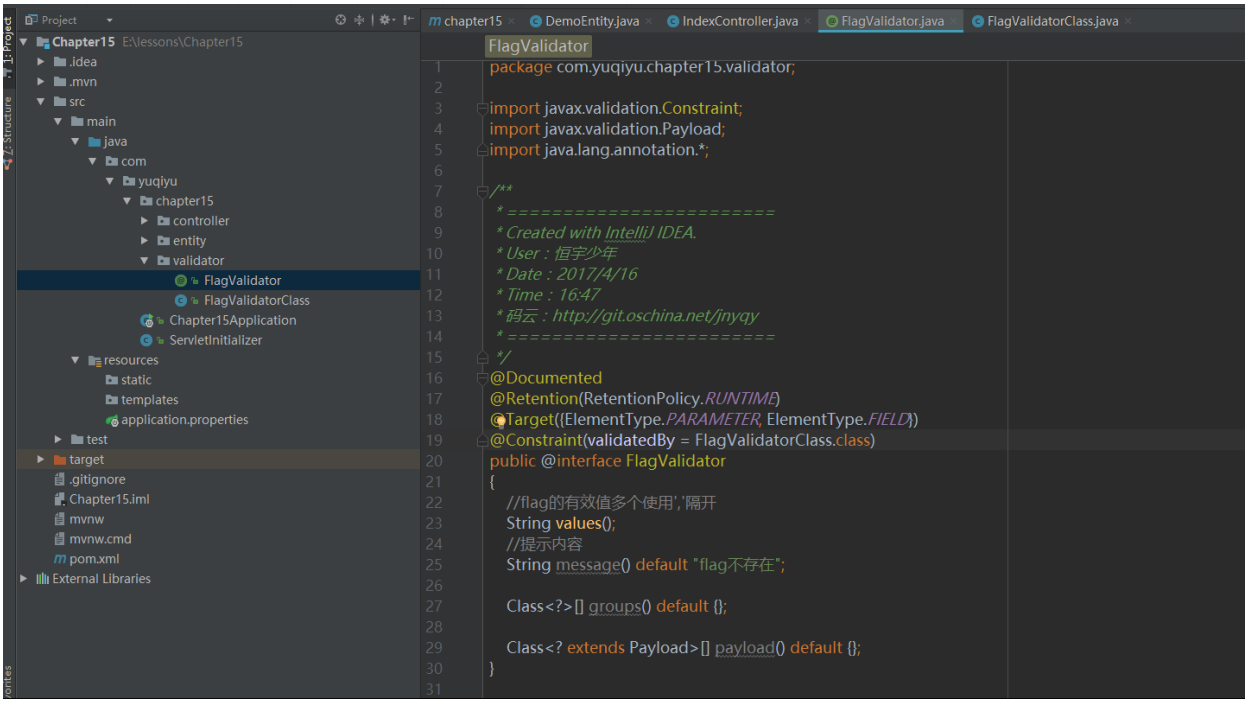
处理类,EnumDeal.java

```
public class EnumDeal implements ConstraintValidator<EnumCheck, Object> {

       private Class<?> enumClass;

       private String enumMethod;

       private List<String> values;
    @Override
       public void initialize(EnumCheck enumObj) {
        // enumObj  是注解里的值
              enumClass = enumObj.enumClass();
```

```
            enumMethod = enumObj.enumMethod();
            values = Stream.of(enumObj.value()).collect(Collectors.toList());
        }
    @Override
      public boolean isValid(Object value, ConstraintValidatorContext context) {
       // value 是 要判断的值
            if (value == null || enumClass == null || enumMethod == null) {
                return false;
            }
          Class<?> valueClass = value.getClass();
            try {
                if(values.contains(value)) {
                    return true;
                }
              Method method = enumClass.getMethod(enumMethod, valueClass);
                Object result = enumClass.cast(method.invoke(null, value));// 激活方法并得到该方法的返回值
                return !Objects.isNull(result);
            } catch (Exception e) {
                e.printStackTrace();
            }
            return false;
        }
}
```

二：

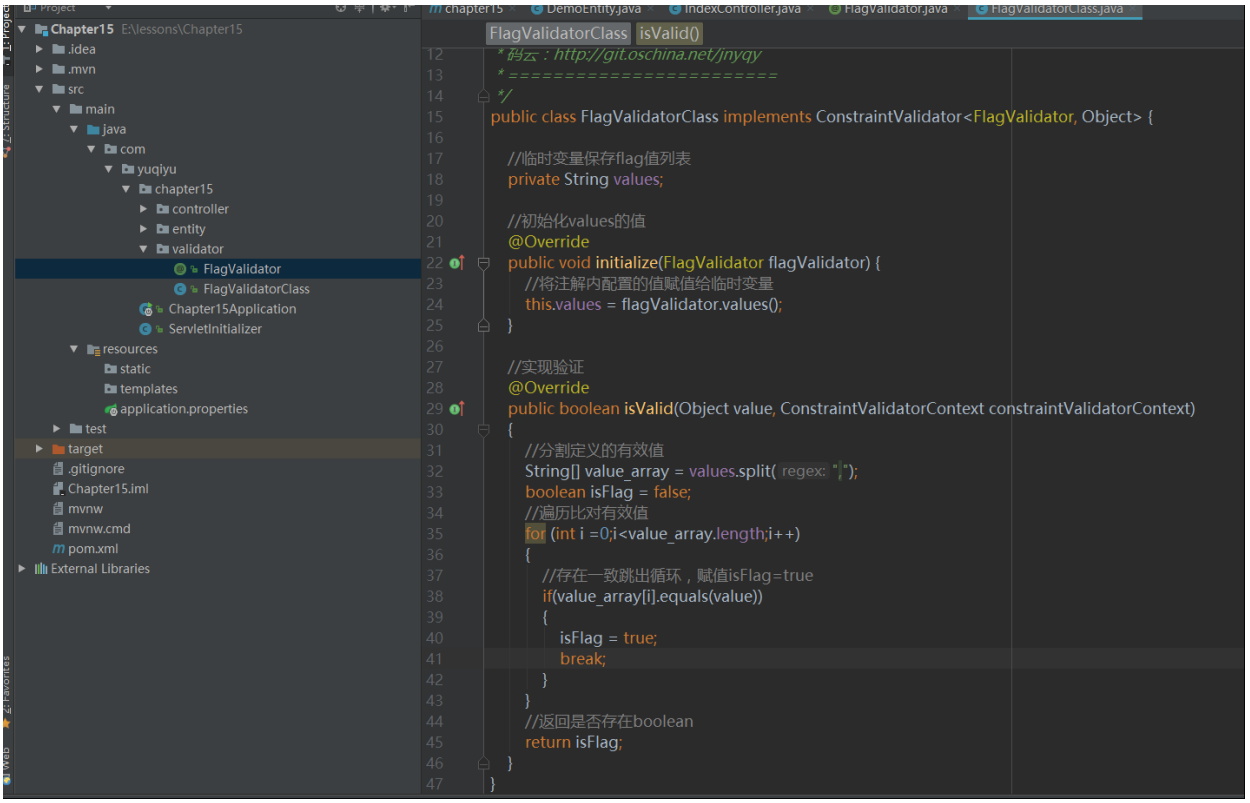
定义一个注解：



可以看到上图内有个@Constraint注解，里面传入了一个validatedBy的字段，这个就是我们自定义注解的实现类的类型，实现类代码如下图所示：

（原文是自定义注解用来校验的，hibernate-validator）

来自〈https://www.jianshu.com/p/e111d3fbc583〉



方法二: 利用反射

B门户中不是这样操作:

B门户是创建一个注解,不实现,手动调用方法,用反射机制找到使用了这个注解的属性(,一个非空注解:判空.)

```
//验证
public static Pair<Boolean, String> validateNotEmpty(Class<?> entity, Object obj) {

    Field[] fields = entity.getDeclaredFields();
    boolean flag = true;
    String message = "对象验空成功";
    log.debug("fields.length():{}", fields.length);
    for (Field field : fields) {
        if (field.isAnnotationPresent(NotEmpty.class)) {
            field.setAccessible(true);
            try {
                Object value = field.get(obj);
                log.debug("value:{}", value);
                field.setAccessible(false);
```

```
                if (null == value ||  "null".equals(value) || "".equals(value)) {
                    message = field.getName();
                    log.debug("对象 {}的字段{} 为空，请检查", obj, field.getName());
                    flag = false;
                }
            } catch (Exception e) {
                log.error("读取对象属性失败，失败的原因是:",e);
            }
        }
    }
    Pair<Boolean, String> result  = new Pair<Boolean, String>(flag,message);
    log.debug("result:{}", result);
    return result;
}
```

定义一个注解:

```
 @Target(FIELD)
@Retention(RUNTIME)
public @interface NotEmpty {

}
```

方法三: 利用AOP

利用切面, 对注解反射, 并处理

该注解, 用于打印日志

注解:

```
/**
 * 用于给方法打印日志
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Documented
public @interface PrintLog {
    /**
     * 方法描述
     */
    String value() default "";
}
```

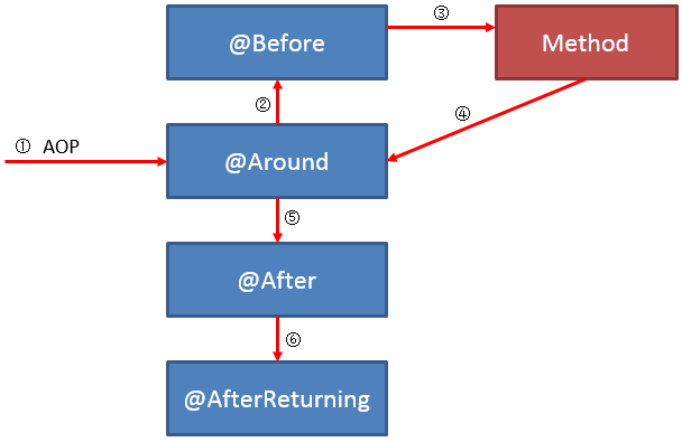
处理类

```
/**
 * 切面,主要用于打印日志
 */
@Aspect
@Component
@Slf4j
public class WebLoggerAspect {

    @Pointcut("@annotation(com.jht.cpr.manager.annotation.PrintLog)")
    public void log() {}
     @AfterThrowing(pointcut = "log()", throwing = "e")
    public void handle(JoinPoint joinPoint, Exception e) {
         log.error("{}-失败,入参:{}, 异常:
{}",getLogValue(joinPoint),JSON.toJSONString(joinPoint.getArgs()
[0]),ExceptionUtils.getStackTrace(e));
        writeContent(); // 错误时写默认返参
    }
    private void writeContent() {
        HttpServletResponse response = ((ServletRequestAttributes)
RequestContextHolder.getRequestAttributes()).getResponse();
        response.reset();
         response.setCharacterEncoding("UTF-8");
                try (PrintWriter writer = response.getWriter()) {
                  BaseResponse respMsg = new BaseResponse();
                       respMsg.setRespCode(CodeEnum.CPR00002.getCode());
                       respMsg.setRespMsg(CodeEnum.CPR00002.getMessage());
                       respMsg.setRespData(null);
                       writer.print(JSON.toJSONString(respMsg));
                       writer.flush();
   } catch (IOException e) {
                       e.printStackTrace();
                }
    }
     @Before("log()")
    public void doBefore(JoinPoint joinPoint) {
        log.info("{}-开始,入参:
{}",getLogValue(joinPoint),JSON.toJSONString(joinPoint.getArgs()[0]));
    }

     @AfterReturning(returning="resp",pointcut="log()")
    public void doAfter(JoinPoint joinPoint,BaseResponse resp) {
        log.info("{}-结束,返参:{}",getLogValue(joinPoint),resp);
    }

    @Around("log()")
    public Object around (ProceedingJoinPoint jp) {
        String funcMsg = getLogValue(jp);
```

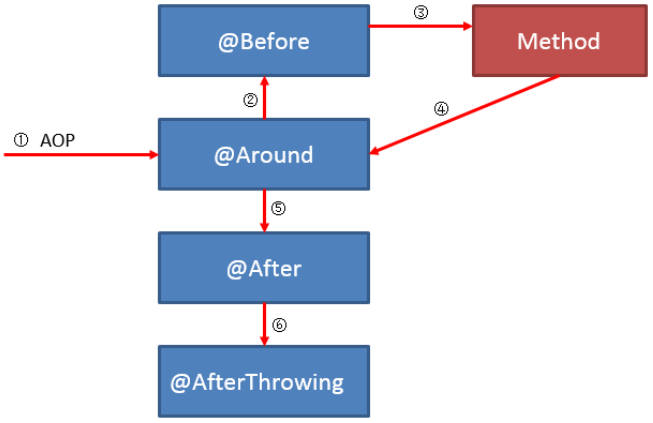
```
        log.info("{}-开始,入参:{}",funcMsg,ManagerUtil.toJSONString(jp.getArgs()));
        Object proceed = null;
        try {
                proceed = jp.proceed(jp.getArgs());
                log.info("{}-结束,返参:{}",funcMsg,JSON.toJSONString(proceed));
            } catch (Throwable e) {
                log.error("{}-失败,入参:{}, 异常:
{}",funcMsg,ManagerUtil.toJSONString(jp.getArgs()),ExceptionUtils.getStackTrace(e));

                BaseResponse respMsg = new BaseResponse();
                respMsg.setRespCode(CodeEnum.CPR00002.getCode());
                respMsg.setRespMsg(CodeEnum.CPR00002.getMessage());
                respMsg.setRespData(null);
                return respMsg;
            }

            return proceed;
    }

    // 得到注解上的值
     private String getLogValue(JoinPoint joinPoint) {
            MethodSignature methodSignature = (MethodSignature)
joinPoint.getSignature();
           Method method = methodSignature.getMethod();
            PrintLog webLogger = method.getAnnotation(PrintLog.class);
            return webLogger.value();
        }
```

切面执行顺序：

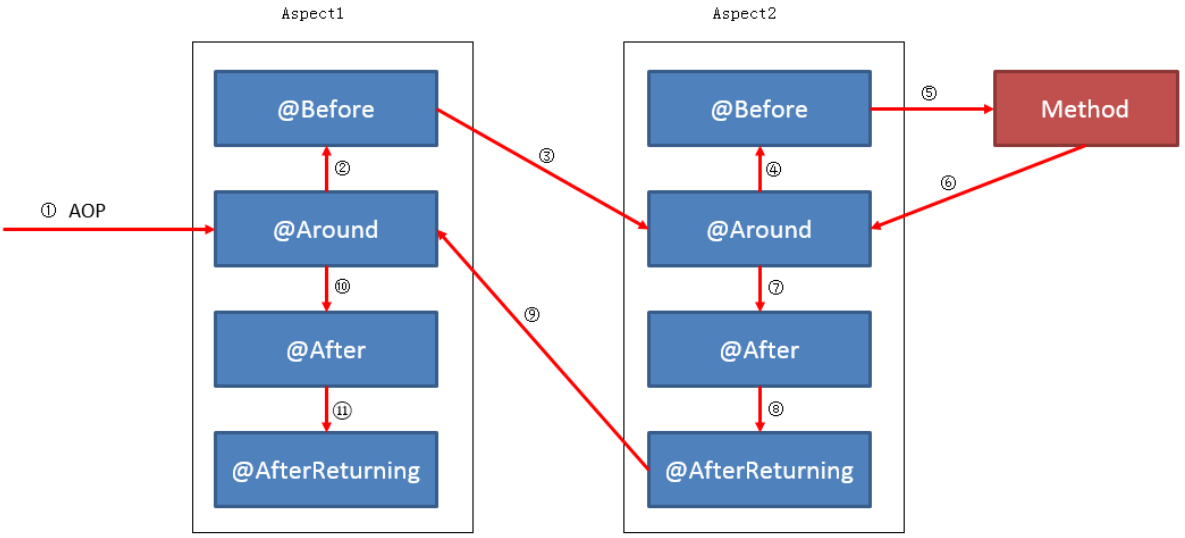
正常情况（最后一步不一样）：



异常情况（最后一步不一样）：



多个切面的情况：



来源：https://blog.csdn.net/rainbow702/article/details/52185827

3.组合注解:

就是在注解上直接使用其他注解,@SpringBootApplication就是个典型的组合注解,组合了@SpringBootConfiguration,@EnableAutoConfiguration,@ComponentScan

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
		@Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
		@Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

```
        @AliasFor(annotation = EnableAutoConfiguration.class)
        Class<?>[] exclude() default {};
    @AliasFor(annotation = EnableAutoConfiguration.class)
        String[] excludeName() default {};
    @AliasFor(annotation = ComponentScan.class, attribute = "basePackages")
        String[] scanBasePackages() default {};
    @AliasFor(annotation = ComponentScan.class, attribute =
"basePackageClasses")
        Class<?>[] scanBasePackageClasses() default {};
}
```

> 使用后,SpringBootApplication 就拥有三个注解的功能了.
> @AliasFor 表示 scanBasePackages 这个值注入到 类 ComponentScan 中的 basePackages 值中 ,相当于值传递

如果被组合的注解中有 没有默认值的属性,则不无法传达值 ,比如 注解 @ApiOperation("删除方法"),只能写死,如果写死就用处不大了,
所以在自定义注解时,尽量给每个值写默认值