目录

Java注解简介

Annotation 中文译过来就是注解、标释的意思，在 Java 中注解是一个很重要的知识点，但经常还是有点让新手不容易理解。

我个人认为，比较糟糕的技术文档主要特征之一就是：用专业名词来介绍专业名词。 比如：

> Java 注解用于为 Java 代码提供元数据。作为元数据，注解不直接影响你的代码执行，但也有一些类型的注解实际上可以用于这一目的。Java 注解是从 Java5 开始添加到 Java 的。 这是大多数网站上对于 Java 注解，解释确实正确，但是说实在话，我第一次学习的时候，头脑一片空白。这什么跟什么啊？听了像没有听一样。因为概念太过于抽象，所以初学者实在是比较吃力才能够理解，然后随着自己开发过程中不断地强化练习，才会慢慢对它形成正确的认识。

我在写这篇文章的时候，我就在思考。如何让自己或者让读者能够比较直观地认识注解这个概念？是要去官方文档上翻译说明吗？我马上否定了这个答案。

后来，我想到了一样东西————墨水，墨水可以挥发、可以有不同的颜色，用来解释注解正好。

不过，我继续发散思维后，想到了一样东西能够更好地代替墨水，那就是印章。印章可以沾上不同的墨水或者印泥，可以定制印章的文字或者图案，如果愿意它也可以被戳到你任何想戳的物体表面。

但是，我再继续发散思维后，又想到一样东西能够更好地代替印章，那就是标签。标签是一张便利纸，标签上的内容可以自由定义。常见的如货架上的商品价格标签、图书馆中的书本编码标签、实验室中化学材料的名称类别标签等等。

并且，往抽象地说，标签并不一定是一张纸，它可以是对人和事物的属性评价。也就是说，标签具备对于抽象事物的解释。

墨水　　印章　　标签

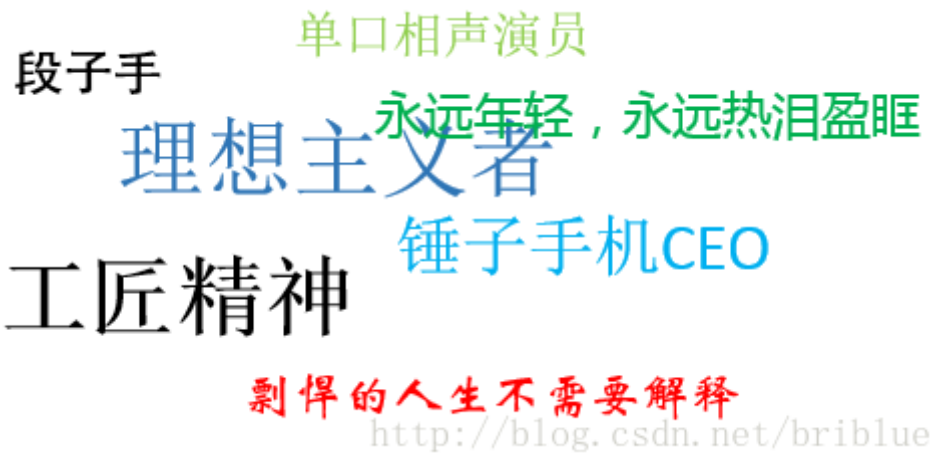
所以，基于如此，我完成了自我的知识认知升级，我决定用标签来解释注解。

注解如同标签

之前某新闻客户端的评论有盖楼的习惯，于是 “乔布斯重新定义了手机、罗永浩重新定义了傻X” 就经常极为工整地出现在了评论楼层中，并且广大网友在相当长的一段时间内对于这种行为乐此不疲。这其实就是等同于贴标签的行为。 在某些网友眼中，罗永浩就成了傻X的代名词。

广大网友给罗永浩贴了一个名为“傻x”的标签，他们并不真正了解罗永浩，不知道他当教师、砸冰箱、办博客的壮举，但是因为“傻x”这样的标签存在，这有助于他们直接快速地对罗永浩这个人做出评价，然后基于此，罗永浩就可以成为茶余饭后的谈资，这就是标签的力量。

而在网络的另一边，老罗靠他的人格魅力自然收获一大批忠实的拥泵，他们对于老罗贴的又是另一种标签。



老罗还是老罗，但是由于人们对于它贴上的标签不同，所以造成对于他的看法大相径庭，不喜欢他的人整天在网络上评论抨击嘲讽，而崇拜欣赏他的人则会愿意挣钱购买锤子手机的发布会门票。

我无意于评价这两种行为，我再引个例子。

《奇葩说》是近年网络上非常火热的辩论节目，其中辩手陈铭被另外一个辩手马薇薇攻击说是————“站在宇宙中心呼唤爱”，然后贴上了一个大大的标签————“鸡汤男”，自此以后，观众再看到陈铭的时候，首先映入脑海中便是“鸡汤男”三个大字，其实本身而言陈铭非常优秀，为人师表、作风正派、谈吐举止得体，但是在网络中，因为娱乐至上的环境所致，人们更愿意以娱乐的心态来认知一切，于是“鸡汤男”就如陈铭自己所说成了一个撕不了的标签。

我们可以抽象概括一下，标签是对事物行为的某些角度的评价与解释。

到这里，终于可以引出本文的主角注解了。

初学者可以这样理解注解：想像代码具有生命，注解就是对于代码中某些鲜活个体的贴上去的一张标签。简化来讲，注解如同一张标签。

在未开始学习任何注解具体语法而言，你可以把注解看成一张标签。这有助于你快速地理解它的大致作用。如果初学者在学习过程有大脑放空的时候，请不要慌张，对自己说：

注解，标签。注解，标签。

Java 注解概述

什么是注解?

> 对于很多初次接触的开发者来说应该都有这个疑问？Annontation是Java5开始引入的新特征，中文名称叫注解。它提供了一种安全的类似注释的机制，用来将任何的信息或元数据（metadata）与程序元素（类、方法、成员变量等）进行关联。为程序的元素（类、方法、成员变量）加上更直观更明了的说明，这些说明信息是与程序的业务逻辑无关，并且供指定的工具或框架使用。Annontation像一种修饰符一样，应用于包、类型、构造方法、方法、成员变量、参数及本地变量的声明语句中。

Java注解是附加在代码中的一些元信息，用于一些工具在编译、运行时进行解析和使用，起到说明、配置的功能。注解不会也不能影响代码的实际逻辑，仅仅起到辅助性的作用。包含在 java.lang.annotation 包中。

注解的用处

1、生成文档。这是最常见的，也是java 最早提供的注解。常用的有@param @return 等
2、跟踪代码依赖性，实现替代配置文件功能。比如Dagger 2依赖注入，未来java开发，将大量注解配置，具有很大用处;
3、在编译时进行格式检查。如@override 放在方法前，如果你这个方法并不是覆盖了超类方法，则编译时就能检查出。

注解的原理（接口+反射+动态代理）

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$Proxy1。通过代理对象调用自定义注解（接口）的方法，会最终调用AnnotationInvocationHandler的invoke方法。该方法会从memberValues这个Map中索引出对应的值。而memberValues的来源是Java常量池。（在使用注解的那个类的class文件常量池中，并不在自定义注解的class文件常量池中，也不在jdk为自定义注解生成的动态代理$Proxy1的常量池中）

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。

JDK里的注解

JDK 内置注解 先来看几个 Java 内置的注解，让大家热热身。

@Override 演示

```
class Parent {
    public void run() {
    }
}

class Son extends Parent {
    /**
     * 这个注解是为了检查此方法是否真的是重写父类的方法
     * 这时候就不用我们用肉眼去观察到底是不是重写了
     */
    @Override
    public void run() {
    }
}
```

@Deprecated 演示 class Parent {

```
/**
 * 此注解代表过时了，但是如果可以调用到，当然也可以正常使用
 * 但是，此方法有可能在以后的版本升级中会被慢慢的淘汰
 * 可以放在类，变量，方法上面都起作用
 */
@Deprecated
```

```
public void run() {
}
}

public class JDKAnnotationDemo {
    public static void main(String[] args) {
        Parent parent = new Parent();
        parent.run(); // 在编译器中此方法会显示过时标志
    }
}
```

@SuppressWarnings 演示 class Parent {

```
// 因为定义的 name 没有使用，那么编译器就会有警告，这时候使用此注解可以屏蔽掉警告
// 即任意不想看到的编译时期的警告都可以用此注解屏蔽掉，但是不推荐，有警告的代码最好还是处理一下
@SuppressWarnings("all")
private String name;
}
```

@FunctionalInterface 演示 /**

- 此注解是 Java8 提出的函数式接口，接口中只允许有一个抽象方法
- 加上这个注解之后，类中多一个抽象方法或者少一个抽象方法都会报错 */ @FunctionalInterface interface Func { void run(); }

注解处理器实战

注解处理器 注解处理器才是使用注解整个流程中最重要的一步了。所有在代码中出现的注解，它到底起了什么作用，都是在注解处理器中定义好的。概念：注解本身并不会对程序的编译方式产生影响，而是注解处理器起的作用；注解处理器能够通过在运行时使用反射获取在程序代码中的使用的注解信息，从而实现一些额外功能。前提是我们自定义的注解使用的是 RetentionPolicy.RUNTIME 修饰的。这也是我们在开发中使用频率很高的一种方式。

我们先来了解下如何通过在运行时使用反射获取在程序中的使用的注解信息。如下类注解和方法注解。

类注解 Class aClass = ApiController.class; Annotation[] annotations = aClass.getAnnotations();

```
for(Annotation annotation : annotations) {
    if(annotation instanceof ApiAuthAnnotation) {
        ApiAuthAnnotation apiAuthAnnotation = (ApiAuthAnnotation) annotation;
        System.out.println("name: " + apiAuthAnnotation.name());
        System.out.println("age: " + apiAuthAnnotation.age());
    }
```

```
}
方法注解
Method method = ... //通过反射获取方法对象
Annotation[] annotations = method.getDeclaredAnnotations();

for(Annotation annotation : annotations) {
    if(annotation instanceof ApiAuthAnnotation) {
        ApiAuthAnnotation apiAuthAnnotation = (ApiAuthAnnotation) annotation;
        System.out.println("name: " + apiAuthAnnotation.name());
        System.out.println("age: " + apiAuthAnnotation.age());
    }
}
```

此部分内容可参考: 通过反射获取注解信息

注解处理器实战 接下来我通过在公司中的一个实战改编来演示一下注解处理器的真实使用场景。 需求: 网站后台接口只能是年龄大于 18 岁的才能访问，否则不能访问 前置准备: 定义注解（这里使用上文的完整注解），使用注解（这里使用上文中使用注解的例子） 接下来要做的事情: 写一个切面，拦截浏览器访问带注解的接口，取出注解信息，判断年龄来确定是否可以继续访问。

在 dispatcher-servlet.xml 文件中定义 aop 切面

```
<aop:config>
    <!--定义切点，切的是我们自定义的注解-->
    <aop:pointcut id="apiAuthAnnotation"
expression="@annotation(cn.caijiajia.devops.aspect.ApiAuthAnnotation)"/>
    <!--定义切面，切点是 apiAuthAnnotation，切面类即注解处理器是 apiAuthAspect，
主处理逻辑在方法名为 auth 的方法中-->
    <aop:aspect ref="apiAuthAspect">
        <aop:around method="auth" pointcut-ref="apiAuthAnnotation"/>
    </aop:aspect>
</aop:config>
```

切面类处理逻辑即注解处理器代码如

```
@Component("apiAuthAspect")
public class ApiAuthAspect {

    public Object auth(ProceedingJoinPoint pjp) throws Throwable {
        Method method = ((MethodSignature) pjp.getSignature()).getMethod();
        ApiAuthAnnotation apiAuthAnnotation =
method.getAnnotation(ApiAuthAnnotation.class);
        Integer age = apiAuthAnnotation.age();
        if (age > 18) {
            return pjp.proceed();
        } else {
            throw new RuntimeException("你未满18岁，禁止访问");
        }
```

```
    }
}
```

不同类型的注解

类注解

你可以在运行期访问类，方法或者变量的注解信息，下是一个访问类注解的例子：

```
 Class aClass = TheClass.class;
Annotation[] annotations = aClass.getAnnotations();

for(Annotation annotation : annotations){
    if(annotation instanceof MyAnnotation){
        MyAnnotation myAnnotation = (MyAnnotation) annotation;
        System.out.println("name: " + myAnnotation.name());
        System.out.println("value: " + myAnnotation.value());
    }
}
```

你还可以像下面这样指定访问一个类的注解：

```
Class aClass = TheClass.class;
Annotation annotation = aClass.getAnnotation(MyAnnotation.class);

if(annotation instanceof MyAnnotation){
    MyAnnotation myAnnotation = (MyAnnotation) annotation;
    System.out.println("name: " + myAnnotation.name());
    System.out.println("value: " + myAnnotation.value());
}
```

方法注解

下面是一个方法注解的例子：

```
public class TheClass {
  @MyAnnotation(name="someName",  value = "Hello World")
  public void doSomething(){}
}
```

你可以像这样访问方法注解：

```
Method method = ... //获取方法对象
Annotation[] annotations = method.getDeclaredAnnotations();

for(Annotation annotation : annotations){
```

```
    if(annotation instanceof MyAnnotation){
        MyAnnotation myAnnotation = (MyAnnotation) annotation;
        System.out.println("name: " + myAnnotation.name());
        System.out.println("value: " + myAnnotation.value());
    }
}
```

你可以像这样访问指定的方法注解：

```
Method method = ... // 获取方法对象
Annotation annotation = method.getAnnotation(MyAnnotation.class);

if(annotation instanceof MyAnnotation){
    MyAnnotation myAnnotation = (MyAnnotation) annotation;
    System.out.println("name: " + myAnnotation.name());
    System.out.println("value: " + myAnnotation.value());
}
```

参数注解

方法参数也可以添加注解，就像下面这样：

```
public class TheClass {
  public static void doSomethingElse(
        @MyAnnotation(name="aName", value="aValue") String parameter){
  }
}
```

你可以通过 Method对象来访问方法参数注解：

```
Method method = ... //获取方法对象
Annotation[][] parameterAnnotations = method.getParameterAnnotations();
Class[] parameterTypes = method.getParameterTypes();

int i=0;
for(Annotation[] annotations : parameterAnnotations){
  Class parameterType = parameterTypes[i++];

  for(Annotation annotation : annotations){
    if(annotation instanceof MyAnnotation){
        MyAnnotation myAnnotation = (MyAnnotation) annotation;
        System.out.println("param: " + parameterType.getName());
        System.out.println("name : " + myAnnotation.name());
        System.out.println("value: " + myAnnotation.value());
    }
  }
}
```

需要注意的是 Method.getParameterAnnotations()方法返回一个注解类型的二维数组，每一个方法的参数包含一个注解数组。

变量注解

下面是一个变量注解的例子：

```
public class TheClass {

  @MyAnnotation(name="someName",  value = "Hello World")
  public String myField = null;
}
```

你可以像这样来访问变量的注解：

```
Field field = ... //获取方法对象</pre>
<pre>Annotation[] annotations = field.getDeclaredAnnotations();

for(Annotation annotation : annotations){
 if(annotation instanceof MyAnnotation){
 MyAnnotation myAnnotation = (MyAnnotation) annotation;
 System.out.println("name: " + myAnnotation.name());
 System.out.println("value: " + myAnnotation.value());
 }
}
```

你可以像这样访问指定的变量注解：

```
Field field = ...//获取方法对象</pre>
<pre>
Annotation annotation = field.getAnnotation(MyAnnotation.class);

if(annotation instanceof MyAnnotation){
 MyAnnotation myAnnotation = (MyAnnotation) annotation;
 System.out.println("name: " + myAnnotation.name());
 System.out.println("value: " + myAnnotation.value());
}
```

Java注解相关面试题

什么是注解？他们的典型用例是什么？

注解是绑定到程序源代码元素的元数据，对运行代码的操作没有影响。

他们的典型用例是：

- 编译器的信息 - 使用注解，编译器可以检测错误或抑制警告override
- 编译时和部署时处理 - 软件工具可以处理注解并生成代码，配置文件等。
- 运行时处理 - 可以在运行时检查注解以自定义程序的行为

描述标准库中一些有用的注解。

java.lang和java.lang.annotation包中有几个注解，更常见的包括但不限于此：

- @Override -标记方法是否覆盖超类中声明的元素。如果它无法正确覆盖该方法，编译器将发出错误
- @Deprecated - 表示该元素已弃用且不应使用。如果程序使用标有此批注的方法，类或字段，编译器将发出警告
- @SuppressWarnings - 告诉编译器禁止特定警告。在与泛型出现之前编写的遗留代码接口时最常用的
- @FunctionalInterface - 在Java 8中引入，表明类型声明是一个功能接口，可以使用Lambda Expression提供其实现

可以从注解方法声明返回哪些对象类型?

返回类型必须是基本类型，String，Class，Enum或数组类型之一。否则，编译器将抛出错误。

这是一个成功遵循此原则的示例代码：

```
enum Complexity {
    LOW, HIGH
}

public @interface ComplexAnnotation {
    Class<? extends Object> value();

    int[] types();

    Complexity complexity();
}
```

下一个示例将无法编译，因为Object不是有效的返回类型：

```
public @interface FailingAnnotation {
    Object complexity();
}
```

哪些程序元素可以注解?

注解可以应用于整个源代码的多个位置。它们可以应用于类，构造函数和字段的声明：

```
@SimpleAnnotation
public class Apply {
    @SimpleAnnotation
    private String aField;

    @SimpleAnnotation
    public Apply() {
        // ...
    }
}
```

方法及其参数：

```
@SimpleAnnotation
public void aMethod(@SimpleAnnotation String param) {
    // ...
}
```

局部变量，包括循环和资源变量：

```
@SimpleAnnotation
int i = 10;

for (@SimpleAnnotation int j = 0; j < i; j++) {
    // ...
}

try (@SimpleAnnotation FileWriter writer = getWriter()) {
    // ...
} catch (Exception ex) {
    // ...
}
```

其他注解类型：

```
@SimpleAnnotation
public @interface ComplexAnnotation {
    // ...
}
```

甚至包，通过package-info.java文件：

```
@PackageAnnotation
package com.baeldung.interview.annotations;
```

从Java 8开始，它们也可以应用于类型的使用。为此，注解必须指定值为ElementType.USE的@Target注解：

```
@Target(ElementType.TYPE_USE)
public @interface SimpleAnnotation {
    // ...
}
```

现在，注解可以应用于类实例创建：

```
new @SimpleAnnotation Apply();
```

类型转换：

```
aString = (@SimpleAnnotation String) something;
```

接口中：

```
public class SimpleList<T>
  implements @SimpleAnnotation List<@SimpleAnnotation T> {
    // ...
}
```

抛出异常上：

```
void aMethod() throws @SimpleAnnotation Exception {
    // ...
}
```

有没有办法限制可以应用注解的元素?

有，@ Target注解可用于此目的。如果我们尝试在不适用的上下文中使用注解，编译器将发出错误。

以下是仅将@SimpleAnnotation批注的用法限制为字段声明的示例：

```
@Target(ElementType.FIELD)
public @interface SimpleAnnotation {
    // ...
}
```

如果我们想让它适用于更多的上下文，我们可以传递多个常量：

```
@Target({ ElementType.FIELD, ElementType.METHOD, ElementType.PACKAGE })
```

我们甚至可以制作一个注解，因此它不能用于注解任何东西。当声明的类型仅用作复杂注解中的成员类型时，这可能会派上用场：

```
@Target({})
public @interface NoTargetAnnotation {
    // ...
}
```

什么是元注解?

元注解适用于其他注解的注解。

所有未使用@Target标记或使用它标记但包含ANNOTATION_TYPE常量的注解也是元注解：

```
@Target(ElementType.ANNOTATION_TYPE)
public @interface SimpleAnnotation {
    // ...
}
```

下面的代码会编译吗?

```
@Target({ ElementType.FIELD, ElementType.TYPE, ElementType.FIELD })
public @interface TestAnnotation {
    int[] value() default {};
}
```

不能。如果在@Target注解中多次出现相同的枚举常量，那么这是一个编译时错误。

删除重复常量将使代码成功编译：

```
@Target({ ElementType.FIELD, ElementType.TYPE})
```

参考文章

https://blog.fundodoo.com/2018/04/19/130.html
https://blog.csdn.net/qq_37939251/article/details/83215703
https://blog.51cto.com/4247649/2109129 https://www.jianshu.com/p/2f2460e6f8e7
https://blog.csdn.net/yuzongtao/article/details/83306182

以前，『XML』是各大框架的青睐者，它以松耦合的方式完成了框架中几乎所有的配置，但是随着项目越来越庞大，『XML』的内容也越来越复杂，维护成本变高。

于是就有人提出来一种标记式高耦合的配置方式，『注解』。方法上可以进行注解，类上也可以注解，字段属性上也可以注解，反正几乎需要配置的地方都可以进行注解。

关于『注解』和『XML』两种不同的配置模式，争论了好多年了，各有各的优劣，注解可以提供更大的便捷性，易于维护修改，但耦合度高，而 XML 相对于注解则是相反的。

追求低耦合就要抛弃高效率，追求效率必然会遇到耦合。本文意不再辨析两者谁优谁劣，而在于以最简单的语言介绍注解相关的基本内容。

注解的本质

「java.lang.annotation.Annotation」接口中有这么一句话，用来描述『注解』。

> The common interface extended by all annotation types
> 所有的注解类型都继承自这个普通的接口（Annotation）

这句话有点抽象，但却说出了注解的本质。我们看一个 JDK 内置注解的定义：

JDK自带的Override注解

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {

}
```

自定义注解

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@interface MyOverride {//自定义注解
}
```

反编译自定义的注解

```
interface se基础.demo.注解.MyOverride extends java.lang.annotation.Annotation {
  @Ljava/lang/annotation/Target;(value={Ljava/lang/annotation/ElementType;.METHOD})
  @Ljava/lang/annotation/Retention;(value=Ljava/lang/annotation/RetentionPolicy;.RUNTIME)
}
```

这是注解 @Override 的定义，其实它本质上就是：

```
public interface Override extends Annotation{

}
```

没错，注解的本质就是一个继承了 Annotation 接口的接口。有关这一点，你可以去反编译任意一个注解类，你会得到结果的。

一个注解准确意义上来说，只不过是一种特殊的注释而已，如果没有解析它的代码，它可能连注释都不如。

而解析一个类或者方法的注解往往有两种形式，一种是编译期直接的扫描，一种是运行期反射。反射的事情我们待会说，而编译器的扫描指的是编译器在对 java 代码编译字节码的过程中会检测到某个类或者方法被一些注解修饰，这时它就会对于这些注解进行某些处理。

典型的就是注解 @Override，一旦编译器检测到某个方法被修饰了 @Override 注解，编译器就会检查当前方法的方法签名是否真正重写了父类的某个方法，也就是比较父类中是否具有一个同样的方法签名。

这一种情况只适用于那些编译器已经熟知的注解类，比如 JDK 内置的几个注解，而你自定义的注解，编译器是不知道你这个注解的作用的，当然也不知道该如何处理，往往只是会根据该注解的作用范围来选择是否编译进字节码文件，仅此而已。

元注解

『元注解』是用于修饰注解的注解，通常用在注解的定义上，例如：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
```

```
}
```

这是我们 @Override 注解的定义，你可以看到其中的 @Target，@Retention 两个注解就是我们所谓的『元注解』，『元注解』一般用于指定某个注解生命周期以及作用目标等信息。

JAVA 中有以下几个『元注解』：

- @Target：注解的作用目标
- @Retention：注解的生命周期
- ~~@Documented：注解是否应当被包含在 JavaDoc 文档中~~
- ~~@Inherited：是否允许子类继承该注解：后两个元注解很少用~~

其中，@Target 用于指明被修饰的注解最终可以作用的目标是谁，也就是指明，你的注解到底是用来修饰方法的？修饰类的？还是用来修饰字段属性的。

@Target 的定义如下：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    /**
     * Returns an array of the kinds of elements an annota
     * can be applied to.
     * @return an array of the kinds of elements an annota
     * can be applied to
     */
    ElementType[] value();
}
```


我们可以通过以下的方式来为这个 value 传值：

```
@Target(value = {ElementType.FIELD})
```

被这个 @Target 注解修饰的注解将只能作用在成员字段上，不能用于修饰方法或者类。其中，ElementType 是一个枚举类型，有以下一些值：

- ElementType.TYPE：允许被修饰的注解作用在类、接口和枚举上
- ElementType.FIELD：允许作用在属性字段上
- ElementType.METHOD：允许作用在方法上
- ElementType.PARAMETER：允许作用在方法参数上
- ElementType.CONSTRUCTOR：允许作用在构造器上
- ElementType.LOCAL_VARIABLE：允许作用在本地局部变量上
- ElementType.ANNOTATION_TYPE：允许作用在注解上
- ElementType.PACKAGE：允许作用在包上

@Retention 用于指明当前注解的生命周期，它的基本定义如下：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    /**
     * Returns the retention policy.
     * @return the retention policy
     */
    RetentionPolicy value();
}
```


同样的，它也有一个 value 属性：

```
@Retention(value = RetentionPolicy.RUNTIME
```

自定义注解：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
@interface MyOverride {
}
```

反编译class文件中常量池仍然存在

```
RuntimeVisibleAnnotations
   #7 = Utf8               Ljava/lang/annotation/
Target;
   #8 = Utf8               value
   #9 = Utf8               Ljava/lang/annotation/
ElementType;
  #10 = Utf8               METHOD
  #11 = Utf8               Ljava/lang/annotation/
Retention;
  #12 = Utf8               Ljava/lang/annotation/
RetentionPolicy;
  #13 = Utf8               SOURCE
```

这里的 RetentionPolicy 依然是一个枚举类型，它有以下几个枚举值可取：

- RetentionPolicy.SOURCE：当前注解编译期可见，不会写入 class 文件
- RetentionPolicy.CLASS：类加载阶段丢弃，会写入 class 文件
- RetentionPolicy.RUNTIME：永久保存，可以反射获取

@Retention 注解指定了被修饰的注解的生命周期，一种是只能在编译期可见，编译后会被丢弃，一种会被编译器编译进 class 文件中，无论是类或是方法，乃至字段，他们都是有属性表的，而 JAVA 虚拟机也定义了几种注解属性表用于存储注解信息，但是这种可见性不能带到方法区，类加载时会予以丢弃，最后一种则是永久存在的可见性。

剩下两种类型的注解我们日常用的不多，也比较简单，这里不再详细的进行介绍了，你只需要知道他们各自的作用即可。@Documented 注解修饰的注解，当我们执行 JavaDoc 文档打包时会被保存进 doc 文档，反之将在打包时丢弃。@Inherited 注解修饰的注解是具有可继承性的，也就说我们的注解修饰了一个类，而该类的子类将自动继承父类的该注解。

JAVA 的内置三大注解

除了上述四种元注解外，JDK 还为我们预定义了另外三种注解，它们是：

- @Override
- @Deprecated
- @SuppressWarnings

@Override 注解想必是大家很熟悉的了，它的定义如下：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

它没有任何的属性，所以并不能存储任何其他信息。它只能作用于方法之上，编译结束后将被丢弃。

所以你看，它就是一种典型的『标记式注解』，仅被编译器可知，编译器在对 java 文件进行编译成字节码的过程中，一旦检测到某个方法上被修饰了该注解，就会去匹对父类中是否具有一个同样方法签名的函数，如果不是，自然不能通过编译。

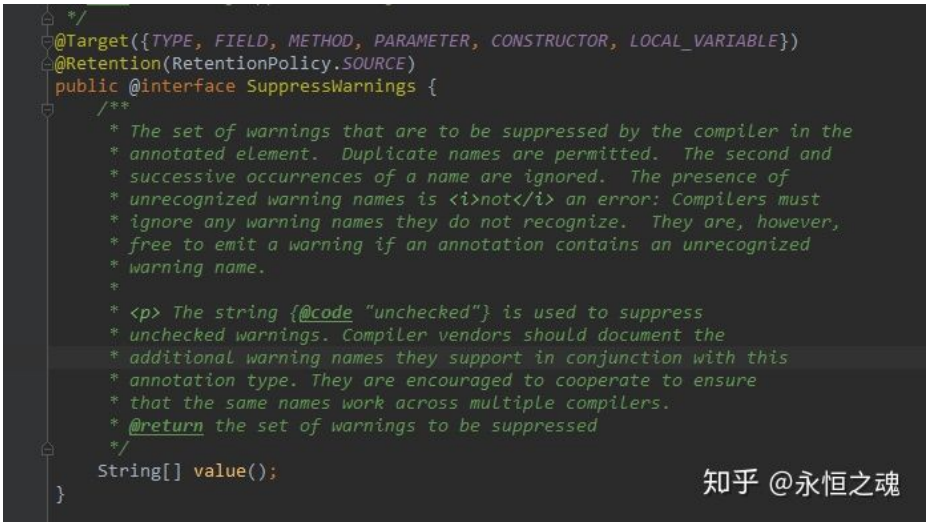
@Deprecated 的基本定义如下：

```
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})
public @interface Deprecated {
}
```


依然是一种『标记式注解』，永久存在，可以修饰所有的类型，作用是，标记当前的类或者方法或者字段等已经不再被推荐使用了，可能下一次的 JDK 版本就会删除。

当然，编译器并不会强制要求你做什么，只是告诉你 JDK 已经不再推荐使用当前的方法或者类了，建议你使用某个替代者。

@SuppressWarnings 主要用来压制 java 的警告，它的基本定义如下：



它有一个 value 属性需要你主动的传值，这个 value 代表一个什么意思呢，这个 value 代表的就是需要被压制的警告类型。例如：

```
public static void main(String[] args) {
    Date date = new Date(2018, 7, 11);
}
```

这么一段代码，程序启动时编译器会报一个警告。

> Warning:(8, 21) java: java.util.Date 中的 Date(int,int,int) 已过时

而如果我们不希望程序启动时，编译器检查代码中过时的方法，就可以使用 @SuppressWarnings 注解并给它的 value 属性传入一个参数值来压制编译器的检查。

```
@SuppressWarning(value = "deprecated")//压制deprecated修饰类的警告
 public static void main(String[] args) {
    Date date = new Date(2018, 7, 11);
}
```

这样你就会发现，编译器不再检查 main 方法下是否有过时的方法调用，也就压制了编译器对于这种警告的检查。

当然，JAVA 中还有很多的警告类型，他们都会对应一个字符串，通过设置 value 属性的值即可压制对于这一类警告类型的检查。

自定义注解的相关内容就不再赘述了，比较简单，通过类似以下的语法即可自定义一个注解。

```
public @interface InnotationName{

}
```

当然，自定义注解的时候也可以选择性的使用元注解进行修饰，这样你可以更加具体的指定你的注解的生命周期、作用范围等信息。

注解与反射

上述内容我们介绍了注解使用上的细节，也简单提到，「注解的本质就是一个继承了 Annotation 接口的接口」，现在我们就来从虚拟机的层面看看，注解的本质到底是什么。

首先，我们自定义一个注解类型：

```java
@Target(value = {ElementType.FIELD,ElementType.METHOD})
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Hello {
    String value();
}
```


这里我们指定了 Hello 这个注解只能修饰字段和方法，并且该注解永久存活，以便我们反射获取。

之前我们说过，虚拟机规范定义了一系列和注解相关的属性表，也就是说，无论是字段、方法或是类本身，如果被注解修饰了，就可以被写进字节码文件。属性表有以下几种：

- RuntimeVisibleAnnotations：运行时可见的注解
- RuntimeInVisibleAnnotations：运行时不可见的注解
- RuntimeVisibleParameterAnnotations：运行时可见的方法参数注解
- RuntimeInVisibleParameterAnnotations：运行时不可见的方法参数注解
- AnnotationDefault：注解类元素的默认值

给大家看虚拟机的这几个注解相关的属性表的目的在于，让大家从整体上构建一个基本的印象，注解在字节码文件中是如何存储的。

所以，对于一个类或者接口来说，Class 类中提供了以下一些方法用于反射注解。

- getAnnotation：返回指定的注解
- isAnnotationPresent：判定当前元素是否被指定注解修饰
- getAnnotations：返回所有的注解
- getDeclaredAnnotation：返回本元素的指定注解
- getDeclaredAnnotations：返回本元素的所有注解，不包含父类继承而来的

方法、字段中相关反射注解的方法基本是类似的，这里不再赘述，我们下面看一个完整的例子。

首先，设置一个虚拟机启动参数，用于捕获 JDK 动态代理类。

> -Dsun.misc.ProxyGenerator.saveGeneratedFiles=true

然后 main 函数。

```java
public class Test {
    @Hello("hello")
    public static void main(String[] args) throws NoSuchMethodException {
        Class cls = Test.class;
        Method method = cls.getMethod( name: "main", String[].class);
        Hello hello = method.getAnnotation(Hello.class);
    }
}
```


我们说过，注解本质上是继承了 Annotation 接口的接口，而当你通过反射，也就是我们这里的 getAnnotation 方法去获取一个注解类实例的时候，其实 JDK 是通过动态代理机制生成一个实现我们注解（接口）的代理类。

我们运行程序后，会看到输出目录里有这么一个代理类，反编译之后是这样的：

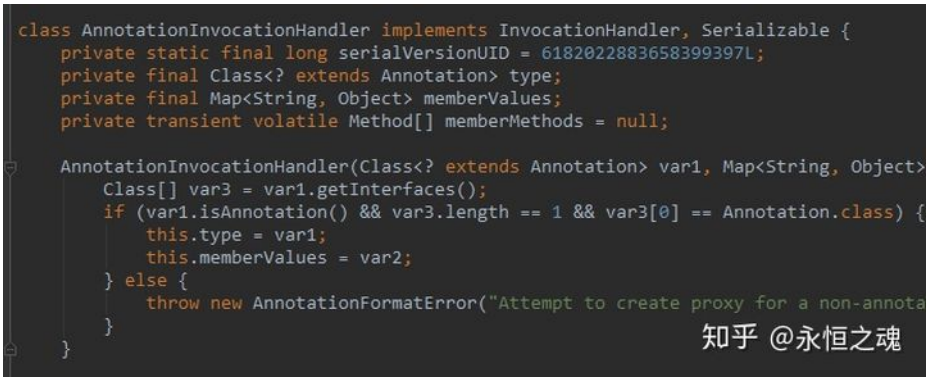
```java
public final class $Proxy1 extends Proxy
    implements Hello
{

    public $Proxy1(InvocationHandler invocationhandler)
    {
        super(invocationhandler);
    }
```

```
public final String value()
{
    try
    {
        return (String)super.h.invoke(this, m3, null);
    }
    catch(Error _ex) { }
    catch(Throwable throwable)
    {
        throw new UndeclaredThrowableException(throwable);
    }
}
```


代理类实现接口 Hello 并重写其所有方法，包括 value 方法以及接口 Hello 从 Annotation 接口继承而来的方法。

而这个关键的 InvocationHandler 实例是谁?

AnnotationInvocationHandler 是 JAVA 中专门用于处理注解的 Handler， 这个类的设计也非常有意思。



这里有一个 memberValues，它是一个 Map 键值对，键是我们注解属性名称，值就是该属性当初被赋上的值。

这里只贴注解自定义方法的调用部分代码

```
class AnnotationInvocationHandler implements InvocationHandler, Serializable {
    private final Map<String, Object> memberValues; //保存调用注解类中常量池中注解key=方法名字和value=方法返回值。比如Hello接口使用时注解 key=方法名字value value是我们定义的返回值=hello
    public Object invoke(Object var1, Method var2, Object[] var3) {var1是注解的代理类,var2是调用注解的方法
        String var4 = var2.getName();//获取调用注解方法名字,value
        Class[] var5 = var2.getParameterTypes();
        Object var6 = this.memberValues.get(var4);//从map中获取注解方法返回值（就是调用者使用注解时候，传递的数值）map.get("value")等于hello
        return var6;
}
```

自定义注解

```
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnocation {
    String description();
    int length();
}
```

使用注解的主类

```
public class DemoTest {
  //使用我们的自定义注解
  @MyAnnocation(description = "yanzhongxin", length = 12) //被存放到DemoTest常量池中，被AnnotationInvocationHandler 存储在hashmap中，key1="descirption",value1="yanzhongxin",key2="length",value2=12
  private String username;
    public static void main(String[] args) throws Exception {
        // 获取类模板
        Class c = DemoTest.class;
        // 获取所有字段
        for(Field f : c.getDeclaredFields()){
            // 判断这个字段是否有MyField注解
            if(f.isAnnotationPresent(MyAnnocation.class)){//获取Annoation方法时才动态代理生成proxy
                MyAnnocation annotation = f.getAnnotation(MyAnnocation.class);
                System.out.println("字段:[" + f.getName() +
                        "], 描述:[" + annotation.description() + "], 长度:[" +
                        annotation.length() +"]"
                );
                System.out.println(annotation);
            }
        }

    }
}
```

打印

```
字段:[username], 描述:[yanzhongxin], 长度:[12]
@se基础.demo.注解.MyAnnocation(description=yanzhongxin, length=12)
```

JDK动态代理为注解（接口）生成的动态代理类

```
final class $Proxy1 extends Proxy implements MyAnnocation {

  private static Method m1;
  private static Method m2;
  private static Method m5;
  private static Method m4;
  private static Method m3;
  private static Method m0;

  public $Proxy1(InvocationHandler var1) throws {
      super(var1);
  }

  public final boolean equals(Object var1) throws {
      return (Boolean)super.h.invoke(this, m1, new Object[]{var1});
  }

  public final String toString() throws {
      return (String)super.h.invoke(this, m2, (Object[])null);
  }

  public final Class annotationType() throws {
      return (Class)super.h.invoke(this, m5, (Object[])null);
  }

  public final String description() throws {

      return (String)super.h.invoke(this, m4, (Object[])null); super.h=AnnotationInvocationHandler注解处理类处理调用的方法
  }

  public final int length() throws {
      return (Integer)super.h.invoke(this, m3, (Object[])null);
  }

  public final int hashCode() throws {
      return (Integer)super.h.invoke(this, m0, (Object[])null);
  }

  static {
      m1 = Class.forName("java.lang.Object").getMethod("equals", Class.forName("java.lang.Object"));
      m2 = Class.forName("java.lang.Object").getMethod("toString");
      m5 = Class.forName("se基础.demo.注解.MyAnnocation").getMethod("annotationType");
      m4 = Class.forName("se基础.demo.注解.MyAnnocation").getMethod("description");注解中自定义的两个抽象方法
      m3 = Class.forName("se基础.demo.注解.MyAnnocation").getMethod("length");
      m0 = Class.forName("java.lang.Object").getMethod("hashCode");
  }
}
```

反编译的字节码中省略了很多try catch之类无关代码

注解底层原理总结：

1、（接口）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。

2、（反射）而我们通过反射获取注解时（注解类、方法、属性等），返回的是Java运行时生成的动态代理对象$Proxy1。

3、（动态代理）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。（在使用注解的那个类的class文件常量池中，并不在自定义注解的class文件常量池中，也不在jdk为自定义注解生成的动态代理$Proxy1的常量池中）

可以参考的阅读链接（并没有写上面的深入原理）：https://www.zhihu.com/question/24401191

相信很多人对Java中的注解都很熟悉，比如我们经常会用到的一些如@Override、@Autowired、@Service等，这些都是JDK或者诸如Spring这类框架给我们提供的。

在以往的面试过程中，我发现，关于注解的知识很多程序员都仅仅停留在使用的层面上，很少有人知道注解是如何实现的，更别提使用自定义注解来解决实际问题了。

但是其实，我觉得一个好的程序员的标准就是懂得如何优化自己的代码，那在代码优化上面，如何精简代码，去掉重复代码就是一个至关重要的话题，在这个话题领域，自定义注解绝对可以算得上是一个大大的功臣。

所以，**在我看来，会使用自定义注解 ≈ 好的程序员。**

那么，本文，就来介绍几个，作者在开发中实际用到的几个例子，向你介绍下如何使用注解来提升你代码的逼格。

基本知识

在Java中，注解分为两种，元注解和自定义注解。

很多人误以为自定义注解就是开发者自己定义的，而其它框架提供的不算，但是其实上面我们提到的那几个注解其实都是自定义注解。

关于”元”这个描述，在编程世界里面有都很多，比如”元注解”、”元数据”、”元类”、”元表”等等，这里的”元”其实都是从meta翻译过来的。

一般我们把**元注解理解为描述注解的注解，元数据理解为描述数据的数据，元类理解为描述类的类**...

所以，在Java中，除了有限的几个固定的”描述注解的注解”以外，所有的注解都是自定义注解。

在JDK中提供了4个标准的用来对注解类型进行注解的注解类（元注解），他们分别是:

```
@Target
@Retention
@Documented后两个用的元注解用的少
@Inherited
```

除了以上这四个，所有的其他注解全部都是自定义注解。

这里不准备深入介绍以上四个元注解的作用，大家可以自行学习。

本文即将提到的几个例子，都是作者在日常工作中真实使用到的场景，这例子有一个共同点，那就是都用到了Spring的AOP技术。

什么是AOP以及他的用法相信很多人都知道，这里也就不展开介绍了。

使用自定义注解做日志记录

不知道大家有没有遇到过类似的诉求，就是希望在一个方法的入口处或者出口处做统一的日志处理，比如记录一下入参、出参、记录下方法执行的时间等。

如果在每一个方法中自己写这样的代码的话，一方面会有很多代码重复，另外也容易被遗漏。

这种场景，就可以使用自定义注解+切面实现这个功能。

假设我们想要在一些web请求的方法上，记录下本次操作具体做了什么事情，比如新增了一条记录或者删除了一条记录等。

首先我们自定义一个注解：

```java
/**
 * Operate Log 的自定义注解
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface OpLog {

    /**
     * 业务类型，如新增、删除、修改
     *
     * @return
     */
    public OpType opType();

    /**
     * 业务对象名称，如订单、库存、价格
     *
     * @return
     */
    public String opItem();

    /**
     * 业务对象编号表达式，描述了如何获取订单号的表达式
     *
     * @return
     */
    public String opItemIdExpression();
}
```

因为我们不仅要在日志中记录本次操作了什么，还需要知道被操作的对象的具体的唯一性标识，如订单号信息。

但是每一个接口方法的参数类型肯定是不一样的，很难有一个统一的标准，那么我们就可以借助Spel表达式，即在表达式中指明如何获取对应的对象的唯一性标识。

有了上面的注解，接下来就可以写切面了。主要代码如下：

```
            if (parameterNames != null) {
                for (int i = 0; i < parameterNames.length; i++) {
                    context.setVariable(parameterNames[i], args[i]);
                }
            }

            // 将方法的resp当做变量放到context中，变量名称为该类名转化为小写字母开头的驼峰形式
            if (response != null) {
                context.setVariable(
                     CaseFormat.UPPER_CAMEL.to(CaseFormat.LOWER_CAMEL, response.getClass().getSimpleN
ame()),
                    response);
            }

            // 解析表达式，获取结果
            String itemId = String.valueOf(expression.getValue(context));

            // 执行日志记录
            handle(opLog.opType(), opLog.opItem(), itemId);
        }

        return response;
    }


    private void handle(OpType opType,  String opItem, String opItemId) {
      // 通过日志打印输出
      LOGGER.info("opType = " + opType.name() +",opItem = " +opItem + ",opItemId = " +opItemId);
    }
}
```

以上切面中，有几个点需要大家注意的：

1、使用@Around注解来指定对标注了OpLog的方法设置切面。 2、使用Spel的相关方法，通过指定的表示，从对应的参数中获取到目标对象的唯一性标识。 3、再方法执行成功后，输出日志。

有了以上的切面及注解后，我们只需要在对应的方法上增加注解标注即可，如：

```
 @RequestMapping(method = {RequestMethod.GET, RequestMethod.POST})
@OpLog(opType = OpType.QUERY, opItem = "order", opItemIdExpression = "#id")
public @ResponseBody
HashMap view(@RequestParam(name = "id") String id)
    throws Exception {
}
```

上面这种是入参的参数列表中已经有了被操作的对象的唯一性标识，直接使用 `#id` 指定即可。

如果被操作的对象的唯一性标识不在入参列表中，那么可能是入参的对象中的某一个属性，用法如下：

```
 @RequestMapping(method = {RequestMethod.GET, RequestMethod.POST})
@OpLog(opType = OpType.QUERY, opItem = "order", opItemIdExpression = "#orderVo.id")
public @ResponseBody
HashMap update(OrderVO orderVo)
    throws Exception {
}
```

以上，即可从入参的OrderVO对象的id属性的值获取。

如果我们要记录的唯一性标识，在入参中没有的话，应该怎么办呢？最典型的就是插入方法，插入成功之前，根本不知道主键ID是什么，这种怎么办呢？

我们上面的切面中，做了一件事情，就是我们把方法的返回值也会使用表达式进行一次解析，如果可以解析得到具体的值，可以是可以。如以下写法：

```
 @RequestMapping(method = {RequestMethod.GET, RequestMethod.POST})
@OpLog(opType = OpType.QUERY, opItem = "order", opItemIdExpression = "#insertResult.id")
public @ResponseBody
InsertResult insert(OrderVO orderVo)
    throws Exception {

    return orderDao.insert(orderVo);
}
```

以上，就是一个简单的使用自定义注解+切面进行日志记录的场景。下面我们再来看一个如何使用注解做方法参数的校验。

使用自定义注解做前置检查

当我们对外部提供接口的时候，会对其中的部分参数有一定的要求，比如某些参数值不能为空等。大多数情况下我们都需要自己主动进行校验，判断对方传入的值是否合理。

这里推荐一个使用HibernateValidator + 自定义注解 + AOP实现参数校验的方式。

首先我们会有一个具体的入参类，定义如下：

```
public class User {
    private String idempotentNo;
    @NotNull(
        message = "userName can't be null"
    )
    private String userName;
}
```

以上，对userName参数注明不能为null。

然后再使用hibernate validator定义一个工具类，用于做参数校验。

```
/**
 * 参数校验工具
 *
 * @author Hollis
 */
public class BeanValidator {

    private static Validator validator = Validation.byProvider(HibernateValidator.class).configure().fa
ilFast(true)
        .buildValidatorFactory().getValidator();

    /**
     * @param object object
     * @param groups groups
     */
    public static void validateObject(Object object, Class<?>... groups) throws ValidationException {
        Set<ConstraintViolation<Object>> constraintViolations = validator.validate(object, groups);
        if (constraintViolations.stream().findFirst().isPresent()) {
              throw new ValidationException(constraintViolations.stream().findFirst().get().getMessage
());
        }
    }
}
```

以上代码，会对一个bean进行校验，一旦失败，就会抛出ValidationException。

接下来定义一个注解：

```java
/**
 * facade接口注解， 用于统一对facade进行参数校验及异常捕获
 * <pre>
 *      注意，使用该注解需要注意，该方法的返回值必须是BaseResponse的子类
 * </pre>
 */

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Facade {

}
```

这个注解里面没有任何参数，只用于标注那些方法要进行参数校验。

接下来定义切面：

```java
/**
 * Facade的切面处理类，统一统计进行参数校验及异常捕获
 *
 * @author Hollis
 */
@Aspect
@Component
public class FacadeAspect {

    private static final Logger LOGGER = LoggerFactory.getLogger(FacadeAspect.class);

    @Autowired
    HttpServletRequest request;

    @Around("@annotation(com.hollis.annotation.Facade)")
    public Object facade(ProceedingJoinPoint pjp) throws Exception {

        Method method = ((MethodSignature)pjp.getSignature()).getMethod();
        Object[] args = pjp.getArgs();

        Class returnType = ((MethodSignature)pjp.getSignature()).getMethod().getReturnType();

        //循环遍历所有参数，进行参数校验
        for (Object parameter : args) {
            try {
                BeanValidator.validateObject(parameter);
            } catch (ValidationException e) {
                return getFailedResponse(returnType, e);
            }
        }
```

以上代码，和前面的切面有点类似，主要是定义了一个切面，会对所有标注@Facade的方法进行统一处理，即在开始方法调用前进行参数校验，一旦校验失败，则返回一个固定的失败的Response，特别需要注意的是，这里之所以可以返回一个固定的BaseResponse，是因为我们会要求我们的所有对外提供的接口的response必须继承BaseResponse类，这个类里面会定义一些默认的参数，如错误码等。

之后，只需要对需要参数校验的方法增加对应注解即可：

```
@Facade
public TestResponse query(User user) {


}
```

这样，有了以上注解和切面，我们就可以对所有的对外方法做统一的控制了。

其实，以上这个facadeAspect我省略了很多东西，我们真正使用的那个切面，不仅仅做了参数检查，还可以做很多其他事情。比如异常的统一处理、错误码的统一转换、记录方法执行时长、记录方法的入参出参等等。

总之，使用切面+自定义注解，我们可以统一做很多事情。除了以上的这几个场景，我们还有很多相似的用法，比如：

统一的缓存处理。如某些操作需要在操作前查缓存、操作后更新缓存。这种就可以通过自定义注解+切面的方式统一处理。

代码其实都差不多，思路也比较简单，就是通过自定义注解来标注需要被切面处理的累或者方法，然后在切面中对方法的执行过程进行干预，比如在执行前或者执行后做一些特殊的操作。

使用这种方式可以大大减少重复代码，大大提升代码的优雅性，方便我们使用。

但是同时也不能过度使用，因为注解看似简单，但是其实内部有很多逻辑是容易被忽略的。就像我之前写过一篇《Spring官方都推荐使用的@Transactional事务，为啥我不建议使用！(https://www.hollischuang.com/archives/5608)》中提到的观点一样，无脑的使用切面和注解，可能会引入一些不必要的问题。

不管怎么说，自定义注解却是是一个很好的发明，可以减少很多重复代码。快快在你的项目中用起来吧。