秒懂，Java 注解 （Annotation）你可以这样学

“乔布斯重新定义了手机，罗永浩重新定义了傻逼”。

过去的一年多，这句话的流行产生了巨大的正能量：让支持我们的人之间有了更强的认同感，让讨厌我们嚣张跋扈但又不能把我们怎么样的人得到了心灵上的抚慰，让没听说过我们的人感觉到这家公司不知道为什么总是被人们拿来跟苹果相提并论，让本打算下班的我锤同事看了之后（我会时不时的转发一次 ^_^）咬牙切齿地说一声 x 你妈的然后坐下来又敲上几个小时代码……

在这里对这句话的原创者宋一成老师致谢，光荣属于宋老师。

这处图片引自老罗的博客。为了避免不必要的麻烦，首先声明我个人比较尊敬老罗的。至于为什么放这张图，自然是为本篇博文服务，接下来我自会说明。好了，可以开始今天的博文了。

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。

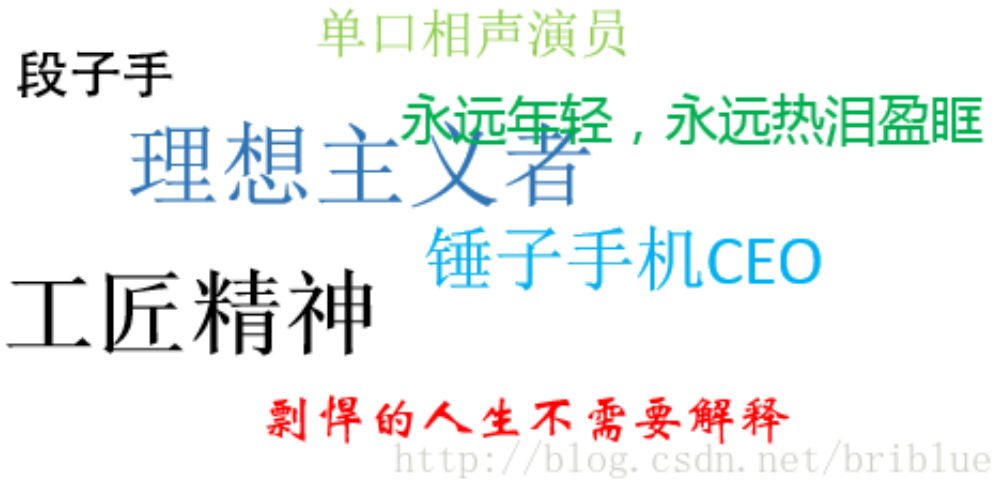
注解如同标签

回到博文开始的地方，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。

注解语法

因为平常开发少见，相信有不少的人员会认为注解的地位不高。其实同 classs 和 interface

一样，注解也属于一种类型。它是在 Java SE 5.0 版本中开始引入的概念。

注解的定义

注解通过 `@interface` 关键字进行定义。

```
public @interface TestAnnotation {
}
```

它的形式跟接口很类似，不过前面多了一个 @ 符号。上面的代码就创建了一个名字为 TestAnnotaion 的注解。

你可以简单理解为创建了一张名字为 TestAnnotation 的标签。

注解的应用

上面创建了一个注解，那么注解的的使用方法是什么呢。

```
@TestAnnotation
public class Test {
}
```

创建一个类 Test,然后在类定义的地方加上 @TestAnnotation 就可以用 TestAnnotation 注解这个类了。

你可以简单理解为将 TestAnnotation 这张标签贴到 Test 这个类上面。

不过，要想注解能够正常工作，还需要介绍一下一个新的概念那就是元注解。

元注解

元注解是什么意思呢?

元注解是可以注解到注解上的注解，或者说元注解是一种基本注解，但是它能够应用到其它的注解上面。

如果难于理解的话，你可以这样理解。元注解也是一张标签，但是它是一张特殊的标签，它的作用和目的就是给其他普通的标签进行解释说明的。

元标签有 @Retention、@Documented、@Target、@Inherited、@Repeatable 5 种。

@Retention

Retention 的英文意为保留期的意思。当 @Retention 应用到一个注解上的时候，它解释说明了这个注解的的存活时间。

它的取值如下：
- RetentionPolicy.SOURCE 注解只在源码阶段保留，在编译器进行编译时它将被丢弃忽视。
- RetentionPolicy.CLASS 注解只被保留到编译进行的时候，它并不会被加载到 JVM 中。
- RetentionPolicy.RUNTIME 注解可以保留到程序运行的时候，它会被加载进入到 JVM 中，所以在程序运行时可以获取到它们。

我们可以这样的方式来加深理解，@Retention 去给一张标签解释的时候，它指定了这张标签张贴的时间。@Retention 相当于给一张标签上面盖了一张时间戳，时间戳指明了标签张贴的时间周期。

```
@Retention(RetentionPolicy.RUNTIME)
public @interface TestAnnotation {
}
```

上面的代码中，我们指定 TestAnnotation 可以在程序运行周期被获取到，因此它的生命周期非常的长。

@Documented

顾名思义，这个元注解肯定是和文档有关。它的作用是能够将注解中的元素包含到 Javadoc 中去。

@Target

Target 是目标的意思，@Target 指定了注解运用的地方。

你可以这样理解，当一个注解被 @Target 注解时，这个注解就被限定了运用的场景。

类比到标签，原本标签是你想张贴到哪个地方就到哪个地方，但是因为 @Target 的存在，它张贴的地方就非常具体了，比如只能张贴到方法上、类上、方法参数上等等。@Target 有下面的取值

- ElementType.ANNOTATION_TYPE 可以给一个注解进行注解
- ElementType.CONSTRUCTOR 可以给构造方法进行注解
- ElementType.FIELD 可以给属性进行注解
- ElementType.LOCAL_VARIABLE 可以给局部变量进行注解
- ElementType.METHOD 可以给方法进行注解
- ElementType.PACKAGE 可以给一个包进行注解
- ElementType.PARAMETER 可以给一个方法内的参数进行注解
- ElementType.TYPE 可以给一个类型进行注解，比如类、接口、枚举

@Inherited

Inherited 是继承的意思，但是它并不是说注解本身可以继承，而是说如果一个超类被 @Inherited 注解过的注解进行注解的话，那么如果它的子类没有被任何注解应用的话，那么这个子类就继承了超类的注解。
说的比较抽象。代码来解释。

```
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@interface Test {}


@Test
public class A {}


public class B extends A {}
```

注解 Test 被 @Inherited 修饰，之后类 A 被 Test 注解，类 B 继承 A,类 B 也拥有 Test 这个注解。

可以这样理解：

老子非常有钱，所以人们给他贴了一张标签叫做富豪。

老子的儿子长大后，只要没有和老子断绝父子关系，虽然别人没有给他贴标签，但是他自然也是富豪。

老子的孙子长大了，自然也是富豪。

这就是人们口中戏称的富一代，富二代，富三代。虽然叫法不同，好像好多个标签，但其实事情的本质也就是他们有一张共同的标签，也就是老子身上的那张富豪的标签。

@Repeatable

Repeatable 自然是可重复的意思。@Repeatable 是 Java 1.8 才加进来的，所以算是一个新的特性。

什么样的注解会多次应用呢？通常是注解的值可以同时取多个。

举个例子，一个人他既是程序员又是产品经理,同时他还是个画家。

```
<span id="wizkm_highlight_tmp_span">@interface Persons {
    Person[]  value();
}


@Repeatable(Persons.class)
@interface Person{
    String role default "";
}


@Person(role="artist")
@Person(role="coder")
@Person(role="PM")
public class SuperMan{

}</span>
```

注意上面的代码，@Repeatable 注解了 Person。而 @Repeatable 后面括号中的类相当于一个容器注解。

什么是容器注解呢？就是用来存放其它注解的地方。它本身也是一个注解。

我们再看看代码中的相关容器注解。

```
@interface Persons {
    Person[]  value();
}
```

按照规定，它里面必须要有一个 value 的属性，属性类型是一个被 @Repeatable 注解过的注解数组，注意它是数组。

如果不好理解的话，可以这样理解。Persons 是一张总的标签，上面贴满了 Person 这种同类型但内容不一样的标签。把 Persons 给一个 SuperMan 贴上，相当于同时给他贴了程序员、产品经理、画家的标签。

我们可能对于 @Person(role="PM") 括号里面的内容感兴趣，它其实就是给 Person 这个注解的 role 属性赋值为 PM ，大家不明白正常，马上就讲到注解的属性这一块。

注解的属性

注解的属性也叫做成员变量。注解只有成员变量，没有方法。注解的成员变量在注解的定义中以“无形参的方法”形式来声明，其方法名定义了该成员变量的名字，其返回值定义了该成员变量的类型。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestAnnotation {

    int id();

    String msg();

}
```

上面代码定义了 TestAnnotation 这个注解中拥有 id 和 msg 两个属性。在使用的时候，我们应该给它们进行赋值。

赋值的方式是在注解的括号内以 value="" 形式，多个属性之前用 ， 隔开。

```
@TestAnnotation(id=3,msg="hello annotation")
public class Test {

}
1
2
3
4
```

需要注意的是，在注解中定义属性时它的类型必须是 8 种基本数据类型外加 类、接口、注解及它们的数组。

注解中属性可以有默认值，默认值需要用 default 关键值指定。比如：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestAnnotation {

    public int id() default -1;

    public String msg() default "Hi";

}
```

TestAnnotation 中 id 属性默认值为 -1，msg 属性默认值为 Hi。
它可以这样应用。

```
@TestAnnotation()
public class Test {}
1
2
```

因为有默认值，所以无需要再在 @TestAnnotation 后面的括号里面进行赋值了，这一步可以省略。

另外，还有一种情况。如果一个注解内仅仅只有一个名字为 value 的属性时，应用这个注解时可以直接接属性值填写到括号内。

```
public @interface Check {
    String value();
}
```

上面代码中，Check 这个注解只有 value 这个属性。所以可以这样应用。

```
@Check("hi")
```

```
int a;
```

这和下面的效果是一样的

```
@Check(value="hi")
int a;
```

最后，还需要注意的一种情况是一个注解没有任何属性。比如

```
public @interface Perform {}
```

那么在应用这个注解的时候，括号都可以省略。

```
@Perform
public void testMethod(){}
```

Java 预置的注解

学习了上面相关的知识，我们已经可以自己定义一个注解了。其实 Java 语言本身已经提供了几个现成的注解。

@Deprecated

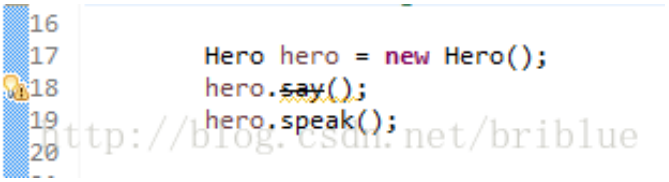
这个元素是用来标记过时的元素，想必大家在日常开发中经常碰到。编译器在编译阶段遇到这个注解时会发出提醒警告，告诉开发者正在调用一个过时的元素比如过时的方法、过时的类、过时的成员变量。

```
public class Hero {

    @Deprecated
    public void say(){
        System.out.println("Noting has to say!");
    }
```

```
7
8
9       public void speak(){
10          System.out.println("I have a dream!");
11      }
12
13
14  }
```

定义了一个 Hero 类，它有两个方法 say() 和 speak()，其中 say() 被 @Deprecated 注解。然后我们在 IDE 中分别调用它们。



可以看到，say() 方法上面被一条直线划了一条，这其实就是编译器识别后的提醒效果。

@Override

这个大家应该很熟悉了，提示子类要复写父类中被 @Override 修饰的方法

@SuppressWarnings

阻止警告的意思。之前说过调用被 @Deprecated 注解的方法后，编译器会警告提醒，而有时候开发者会忽略这种警告，他们可以在调用的地方通过 @SuppressWarnings 达到目的。

```
1   @SuppressWarnings("deprecation")
2   public void test1(){
3       Hero hero = new Hero();
4       hero.say();
5       hero.speak();
6   }
7   1
8   2
9   3
10  4
11  5
12  6
```

@SafeVarargs

参数安全类型注解。它的目的是提醒开发者不要用参数做一些不安全的操作,它的存在会阻止编译器产生 unchecked 这样的警告。它是在 Java 1.7 的版本中加入的。

```
@SafeVarargs // Not actually safe!
    static void m(List<String>... stringLists) {
    Object[] array = stringLists;
    List<Integer> tmpList = Arrays.asList(42);
    array[0] = tmpList; // Semantically invalid, but compiles without warnings
    String s = stringLists[0].get(0); // Oh no, ClassCastException at runtime!
}
```

上面的代码中，编译阶段不会报错，但是运行时会抛出 ClassCastException 这个异常，所以它虽然告诉开发者要妥善处理，但是开发者自己还是搞砸了。

Java 官方文档说，未来的版本会授权编译器对这种不安全的操作产生错误警告。

@FunctionalInterface

函数式接口注解，这个是 Java 1.8 版本引入的新特性。函数式编程很火，所以 Java 8 也及时添加了这个特性。

函数式接口 (Functional Interface) 就是一个具有一个方法的普通接口。

比如

```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
```

```
    public abstract void run();
}
```

我们进行线程开发中常用的 Runnable 就是一个典型的函数式接口，上面源码可以看到它就被 @FunctionalInterface 注解。

可能有人会疑惑，函数式接口标记有什么用，这个原因是函数式接口可以很容易转换为 Lambda 表达式。这是另外的主题了，有兴趣的同学请自己搜索相关知识点学习。

注解的提取

博文前面的部分讲了注解的基本语法，现在是时候检测我们所学的内容了。

我通过用标签来比作注解，前面的内容是讲怎么写注解，然后贴到哪个地方去，而现在我们要做的工作就是检阅这些标签内容。 形象的比喻就是你把这些注解标签在合适的时候撕下来，然后检阅上面的内容信息。

要想正确检阅注解，离不开一个手段，那就是反射。

注解与反射。

注解通过反射获取。首先可以通过 Class 对象的 isAnnotationPresent() 方法判断它是否应用了某个注解

```
public boolean isAnnotationPresent(Class<? extends Annotation>
annotationClass) {}
```

然后通过 getAnnotation() 方法来获取 Annotation 对象。

```
 public <A extends Annotation> A getAnnotation(Class<A> annotationClass) {}
1
2
```

或者是 getAnnotations() 方法。

```
public Annotation[] getAnnotations() {}
```

前一种方法返回指定类型的注解，后一种方法返回注解到这个元素上的所有注解。

如果获取到的 Annotation 如果不为 null，则就可以调用它们的属性方法了。比如

```
@TestAnnotation()
public class Test {

    public static void main(String[] args) {

        boolean hasAnnotation =
Test.class.isAnnotationPresent(TestAnnotation.class);

        if ( hasAnnotation ) {
            TestAnnotation testAnnotation =
Test.class.getAnnotation(TestAnnotation.class);

            System.out.println("id:"+testAnnotation.id());
            System.out.println("msg:"+testAnnotation.msg());
        }

    }

}
```

程序的运行结果是：

```
id:-1
msg:
```

这个正是 TestAnnotation 中 id 和 msg 的默认值。

上面的例子中，只是检阅出了注解在类上的注解，其实属性、方法上的注解照样是可以的。同样还是要假手于反射。

```
@TestAnnotation(msg="hello")
public class Test {

    @Check(value="hi")
```

```
    int a;


    @Perform
    public void testMethod(){}


    @SuppressWarnings("deprecation")
    public void test1(){
        Hero hero = new Hero();
        hero.say();
        hero.speak();
    }


    public static void main(String[] args) {

        boolean hasAnnotation =
Test.class.isAnnotationPresent(TestAnnotation.class);

        if ( hasAnnotation ) {
            TestAnnotation testAnnotation =
Test.class.getAnnotation(TestAnnotation.class);
            //获取类的注解
            System.out.println("id:"+testAnnotation.id());
            System.out.println("msg:"+testAnnotation.msg());
        }


        try {
            Field a = Test.class.getDeclaredField("a");
            a.setAccessible(true);
            //获取一个成员变量上的注解
            Check check = a.getAnnotation(Check.class);

            if ( check != null ) {
                System.out.println("check value:"+check.value());
            }

            Method testMethod = Test.class.getDeclaredMethod("testMethod");

            if ( testMethod != null ) {
                // 获取方法中的注解
                Annotation[] ans = testMethod.getAnnotations();
                for( int i = 0;i < ans.length;i++) {
                    System.out.println("method testMethod
```

```
annotation:"+ans[i].annotationType().getSimpleName());
                }
            }
        } catch (NoSuchFieldException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            System.out.println(e.getMessage());
        } catch (SecurityException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            System.out.println(e.getMessage());
        } catch (NoSuchMethodException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            System.out.println(e.getMessage());
        }



    }

}
```

它们的结果如下：

```
id:-1
msg:hello
check value:hi
method testMethod annotation:Perform
```

需要注意的是，如果一个注解要在运行时被成功提取，那么@Retention(RetentionPolicy.RUNTIME) 是必须的。

注解的使用场景

我相信博文讲到这里大家都很熟悉了注解，但是有不少同学肯定会问，注解到底有什么用呢?

对啊注解到底有什么用?

我们不妨将目光放到 Java 官方文档上来。

文章开始的时候，我用标签来类比注解。但标签比喻只是我的手段，而不是目的。为的是让大家在初次学习注解时能够不被那些抽象的新概念搞懵。既然现在，我们已经对注解有所了解，我们不妨再仔细阅读官方最严谨的文档。

> 注解是一系列元数据，它提供数据用来解释程序代码，但是注解并非是所解释的代码本身的一部分。注解对于代码的运行效果没有直接影响。
> 注解有许多用处，主要如下：
> - 提供信息给编译器： 编译器可以利用注解来探测错误和警告信息
> - 编译阶段时的处理： 软件工具可以用来利用注解信息来生成代码、Html文档或者做其它相应处理。
> - 运行时的处理： 某些注解可以在程序运行的时候接受代码的提取

值得注意的是，注解不是代码本身的一部分。

如果难于理解，可以这样看。罗永浩还是罗永浩，不会因为某些人对于他“傻x”的评价而改变，标签只是某些人对于其他事物的评价，但是标签不会改变事物本身，标签只是特定人群的手段。所以，注解同样无法改变代码本身，注解只是某些工具的的工具。

还是回到官方文档的解释上，注解主要针对的是编译器和其它工具软件(SoftWare tool)。

当开发者使用了Annotation 修饰了类、方法、Field 等成员之后，这些 Annotation 不会自己生效，必须由开发者提供相应的代码来提取并处理 Annotation 信息。这些处理提取和处理 Annotation 的代码统称为 APT（Annotation Processing Tool)。

现在，我们可以给自己答案了，注解有什么用？给谁用？给 编译器或者 APT 用的。

如果，你还是没有搞清楚的话，我亲自写一个好了。

亲手自定义注解完成某个目的

我要写一个测试框架，测试程序员的代码有无明显的异常。

—— 程序员 A : 我写了一个类，它的名字叫做 NoBug，因为它所有的方法都没有错误。
—— 我：自信是好事，不过为了防止意外，让我测试一下如何?
—— 程序员 A: 怎么测试?
—— 我：把你写的代码的方法都加上 @Jiecha 这个注解就好了。
—— 程序员 A: 好的。

NoBug.java

```
1  package ceshi;
```

```java
import ceshi.Jiecha;


public class NoBug {

    @Jiecha
    public void suanShu(){
        System.out.println("1234567890");
    }
    @Jiecha
    public void jiafa(){
        System.out.println("1+1="+1+1);
    }
    @Jiecha
    public void jiefa(){
        System.out.println("1-1="+(1-1));
    }
    @Jiecha
    public void chengfa(){
        System.out.println("3 x 5="+ 3*5);
    }
    @Jiecha
    public void chufa(){
        System.out.println("6 / 0="+ 6 / 0);
    }

    public void ziwojieshao(){
        System.out.println("我写的程序没有 bug!");
    }

}
```

上面的代码，有些方法上面运用了 @Jiecha 注解。

这个注解是我写的测试软件框架中定义的注解。

```java
package ceshi;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface Jiecha {

```

```
}
```

然后，我再编写一个测试类 TestTool 就可以测试 NoBug 相应的方法了。

```
package ceshi;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;



public class TestTool {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        NoBug testobj = new NoBug();

        Class clazz = testobj.getClass();

        Method[] method = clazz.getDeclaredMethods();
        //用来记录测试产生的 log 信息
        StringBuilder log = new StringBuilder();
        // 记录异常的次数
        int errornum = 0;

        for ( Method m: method ) {
            // 只有被 @Jiecha 标注过的方法才进行测试
            if ( m.isAnnotationPresent( Jiecha.class )) {
                try {
                    m.setAccessible(true);
                    m.invoke(testobj, null);

                } catch (Exception e) {
                    // TODO Auto-generated catch block
                    //e.printStackTrace();
                    errornum++;
                    log.append(m.getName());
                    log.append(" ");
                    log.append("has error:");
                    log.append("\n\r  caused by ");
                    //记录测试过程中，发生的异常的名称
                    log.append(e.getCause().getClass().getSimpleName());
                    log.append("\n\r");
```

```
                    //记录测试过程中，发生的异常的具体信息
                    log.append(e.getCause().getMessage());
                    log.append("\n\r");
                }
            }
        }


        log.append(clazz.getSimpleName());
        log.append(" has  ");
        log.append(errornum);
        log.append(" error.");

        // 生成测试报告
        System.out.println(log.toString());

    }

}
```

测试的结果是：

```
1234567890
1+1=11
1-1=0
3 x 5=15
chufa has error:

  caused by ArithmeticException

/ by zero

NoBug has  1 error.
```

提示 NoBug 类中的 chufa() 这个方法有异常，这个异常名称叫做 ArithmeticException，原因是运算过程中进行了除 0 的操作。

所以，NoBug 这个类有 Bug。

这样，通过注解我完成了我自己的目的，那就是对别人的代码进行测试。

所以，再问我注解什么时候用？我只能告诉你，这取决于你想利用它干什么用。

注解应用实例

注解运用的地方太多了，因为我是 Android 开发者，所以我接触到的具体例子有下：

JUnit

JUnit 这个是一个测试框架，典型使用方法如下：

```
public class ExampleUnitTest {
    @Test
    public void addition_isCorrect() throws Exception {
        assertEquals(4, 2 + 2);
    }
}
```

@Test 标记了要进行测试的方法 addition_isCorrect().

ButterKnife

ButterKnife 是 Android 开发中大名鼎鼎的 IOC 框架，它减少了大量重复的代码。

```
public class MainActivity extends AppCompatActivity {

    @BindView(R.id.tv_test)
    TextView mTv;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ButterKnife.bind(this);
    }
}
```

Dagger2

也是一个很有名的依赖注入框架。

Retrofit

很牛逼的 Http 网络访问框架

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Call<List<Repo>> listRepos(@Path("user") String user);
}

Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .build();

GitHubService service = retrofit.create(GitHubService.class);
```

当然，还有许多注解应用的地方，这里不一一列举。

总结

1. 如果注解难于理解，你就把它类同于标签，标签为了解释事物，注解为了解释代码。
2. 注解的基本语法，创建如同接口，但是多了个 @ 符号。
3. 注解的元注解。
4. 注解的属性。
5. 注解主要给编译器及工具类型的软件用的。
6. 注解的提取需要借助于 Java 的反射技术，反射比较慢，所以注解使用时也需要谨慎计较时间成本。

Java 反射机制中另外一个比较重要的概念就是动态代理了，写下这篇文章后，我一鼓作气，又写了这篇 《轻松学，Java 中的代理模式及动态代理》，有兴趣的同学可以一并阅读一下。

最后致敬老罗和陈铭，拿你们的事例为博文主题提供论点，只是基于技术视角，并没有一丝恶意和冒犯之心。