**Java 注解**

**概念**

提供一种为程序元素设置元数据的方法。

**基本原则**

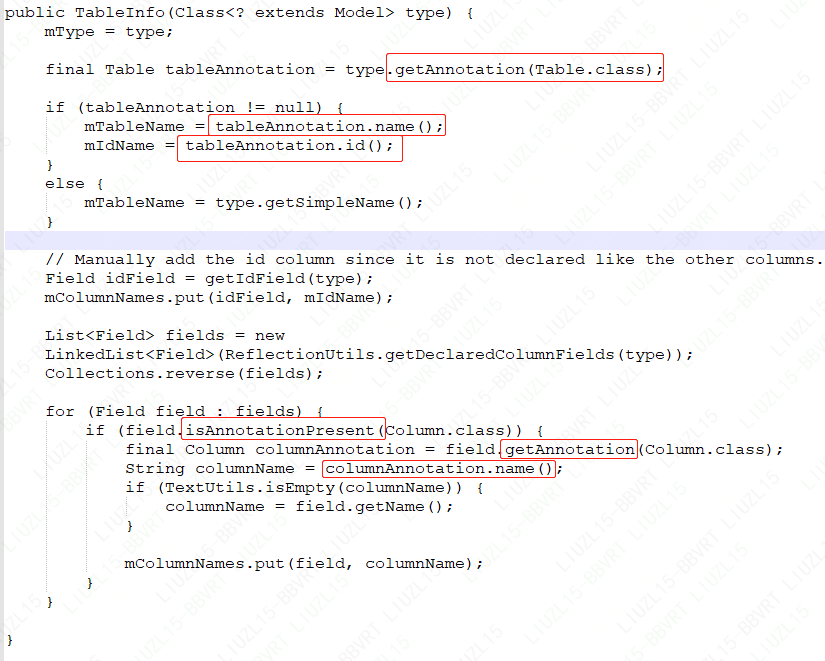
注解不能直接干扰程序代码的运行，无论增加或删除注解，代码都能够正常运行。

**使用范围**

Java中的类、方法、变量、参数、包都可以被注解。

**工作原理**

注解的逻辑实现是元数据的用户来处理的。注解仅仅提供它定义的属性（类/方法/变量/参数/包）的信息，注解的用户来读取这些信息并实现必要的逻辑。当使用java中的注解时（比如@Override、@Deprecated、@SuppressWarnings）JVM就是用户，它在字节码层面工作。如果是自定义的注解，它的用户是每个使用注解的类，通过反射来获取注解类中的每个属性：



**特别说明**

1. 注解仅仅是元数据，和业务逻辑无关。逻辑实现由用户来处理。
2. javadoc中的@author、@version、@param、@return、@deprecated、@hide、@throws、@exception、@see是标记，并不是注解

**注解的作用**

每当你创建**描述符性质的类或者接口**时,一旦其中**包含重复性**的工作，就可以考虑使用注解来简化与自动化该过程。

1. 格式检查

告诉编译器信息，比如被@Override标记的方法如果不是父类的某个方法，IDE会报错；

1. 减少配置

运行时动态处理，得到注解信息，实现代替配置文件的功能；

1. 减少重复工作

**元注解**

标记其他注解的注解。

@Documented、@Retention、@Target、@Inherited、@Repeatable**（JDK1.8）**就是元注解，它们的作用是负责注解其它注解，主要是描述注解的一些属性，任何注解都离不开元注解（包括元注解自身，通过元注解可以自定义注解），元注解的用户是JDK，JDK已经帮助我们实现了这四个注解的逻辑。这四个注解在JDK的java.lang.annotation包中。

**@Target**

1. 作用

用于描述注解的使用范围，即被描述的注解可以用在什么地方。当注解未指定Target值时，则此注解可以用于任何元素之上，多个值使用{}包含并用逗号隔开，如下：



1. 取值

CONSTRUCTOR：构造器；

FIELD：描述域

LOCAL\_VARIABLE：局部变量；

METHOD：方法；

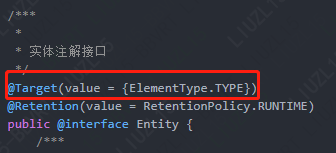
PACKAGE：包；

PARAMETER：参数;

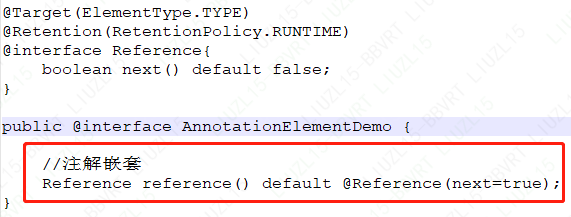
TYPE：类、接口(包括注解类型) 或enum声明。

TYPE\_PARAMETER：表明注解可以用于类型参数声明（**1.8新加入**）

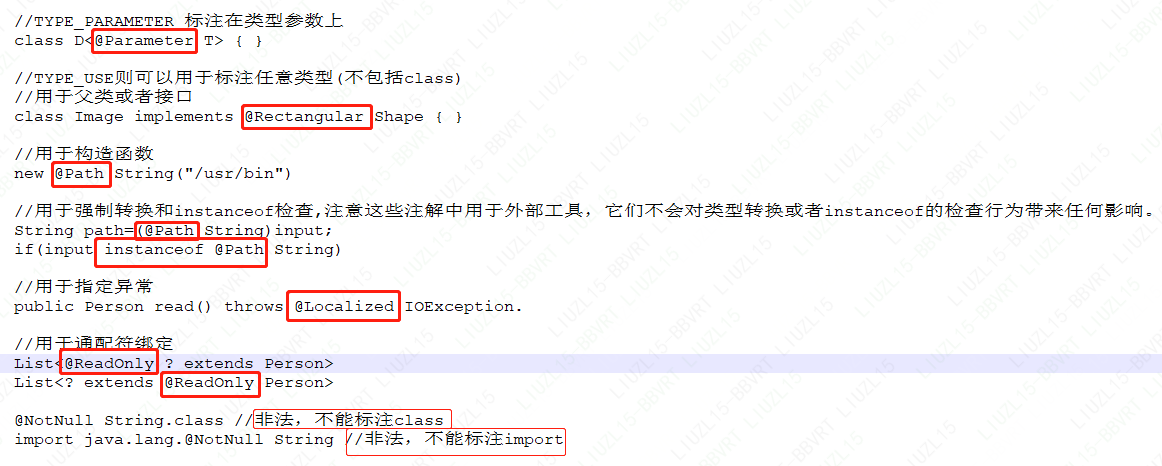
TYPE\_USE：类型使用声明（**1.8新加入**）



嵌套注解



在**Java8**中 ElementType 新增两个枚举成员，**TYPE\_PARAMETER** 和 **TYPE\_USE** ，在Java8前注解只能标注在一个声明(如字段、类、方法)上，Java8后，新增的TYPE\_PARAMETER可以用于标注**类型参数**，而TYPE\_USE则可以用于**标注任意类型(不包括class)**。



类型注解用来支持在Java的程序中做强类型检查，配合第三方插件工具（如Checker Framework），可以在编译期检测出runtime error（如UnsupportedOperationException、NullPointerException异常），避免异常延续到运行期才发现，从而提高代码质量，这就是类型注解的主要作用。

**@Retention**

1. 作用

表示需要在什么级别保存该注解信息，用于描述注解的生命周期，即被描述的注解在什么范围内有效。

1. 取值

SOURCE:在源文件中有效，即源文件保留；

CLASS:在class文件中有效，即class保留；

RUNTIME:在运行时有效，即运行时保留

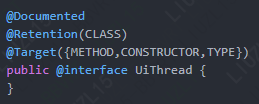
**@Documented**

1. 作用

用于描述其它类型的annotation应该被作为被标注的程序成员的公共API，因此可以被例如javadoc此类的工具文档化。

1. 取值

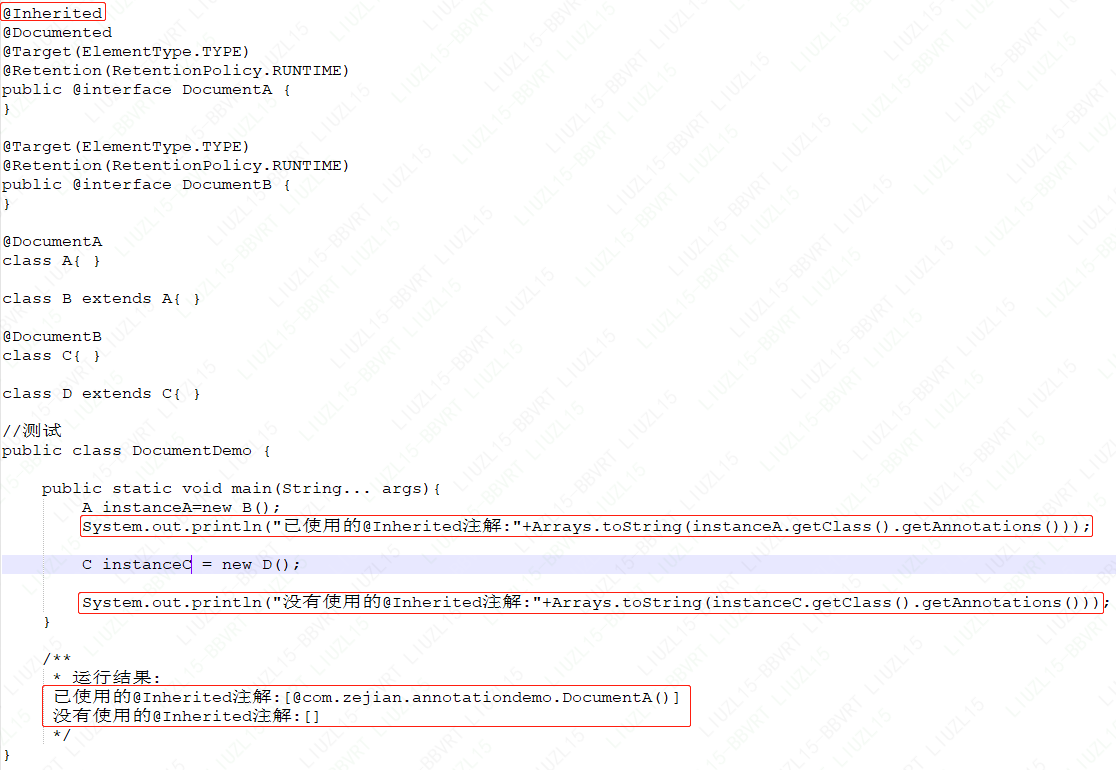
它属于标记注解，没有成员



**@Inherited**

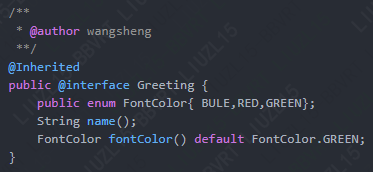
1. 作用

用于描述某个被标注的类型是可被继承的。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。并不是真的继承，只是可以让子类Class对象使用 getAnnotations() 获取父类被 @Inherited修饰的注解，如下：



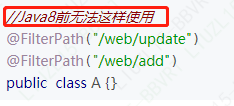
1. 取值

它属于标记注解，没有成员。



**@Repeatable**

**JDK 1.8 新加入**的，表示在同一个位置重复使用相同的注解。在没有该注解前，一般是无法在同一类型上使用相同的注解的，如：



定义

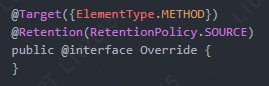


为了处理上述的新增注解，Java8还在AnnotatedElement接口新增了**getDeclaredAnnotationsByType()** 和 **getAnnotationsByType()**两个方法并在接口给出了默认实现，在指定@Repeatable的注解时，可以通过这两个方法获取到注解相关信息。但请注意，旧版API中的getDeclaredAnnotation()和 getAnnotation()是不对@Repeatable注解的处理的**(除非该注解没有在同一个声明上重复出现**)。注意**getDeclaredAnnotationsByType**方法获取到的注解**不包括父类**，其实当 getAnnotationsByType()方法调用时，其内部先执行了getDeclaredAnnotationsByType方法，只有当前类不存在指定注解时，getAnnotationsByType()才会继续从其父类寻找，但请注意如果@FilterPath和@FilterPaths**没有使用了@Inherited的话，仍然无法获取**。

**常见注解（Java 内置）**

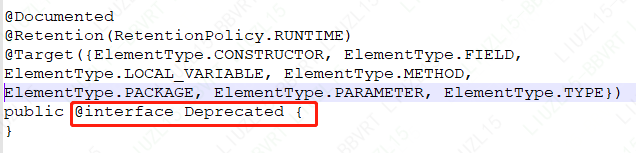
**@Override**

属于标记注解，不需要设置属性值；只能添加在方法的前面，用于标记该方法是复写的父类中的某个方法，如果在父类没有的方法前面加上@Override注解，编译器会报错



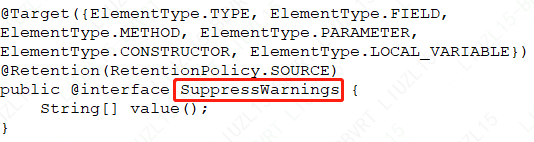
**@Deprecated**

属于标记注解，不需要设置属性值；可以对构造方法、变量、方法、包、参数标记，告知用户和编译器被标记的内容已不建议被使用，如果被使用，编译器会报警告，但不会报错，程序也能正常运行。



**@SuppressWarnings**

可以对构造方法、变量、方法、包、参数标记，用于告知编译器忽略指定的警告，不用再编译完成后出现警告信息。



其内部有一个String数组，主要接收值如下：

**deprecation**：使用了不赞成使用的类或方法时的警告；

**unchecked**：执行了未检查的转换时的警告，例如当使用集合时没有用泛型 (Generics) 来指定集合保存的类型;

**fallthrough**：当 Switch 程序块直接通往下一种情况而没有 Break 时的警告;

**path**：在类路径、源文件路径等中有不存在的路径时的警告;

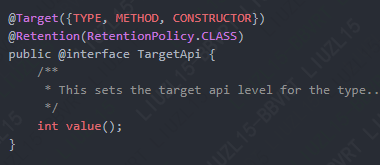
**serial**：当在可序列化的类上缺少 serialVersionUID 定义时的警告;

**finally**：任何 finally 子句不能正常完成时的警告;

**all**：关于以上所有情况的警告。

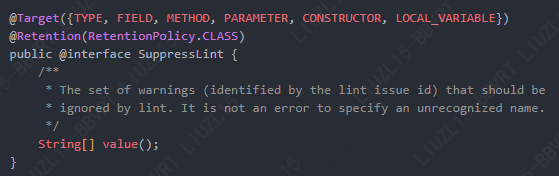
@TargetApi

可以对接口、方法、构造方法标记，如果在应用中指定minSdkVersion为8，但有地方需要使用API 11中的方法，为了避免编译器报错，在调用API11中方法的接口、方法或者构造方法前面加上@Target(11)，这样该方法就可以使用<=11的API接口了。虽然这样能够避免编译器报错，但在运行时需要注意，不能在API低于11的设备中使用该方法，否则会crash（可以获取程序运行设备的API版本来判断是否调用该方法）



@SuppressLint

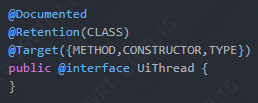
和@Target的功能差不多，但使用范围更广，主要用于避免在lint检查时报错



**自定义注解**

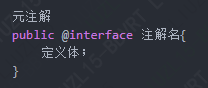
**特征**

1. 注解类会被 @interface 标记
2. 注解类的顶部会被@Documented、@Retention、@Target、@Inherited这四个注解标记（@Documented、@Inherited可选，@Retention、@Target必须要有）



在定义注解时，不能继承其他的注解或接口。@interface用来声明一个注解，其中的每一个方法实际上是声明了一个**配置参数**。方法的名称就是**参数的名称**，返回值类型就是**参数的类型**（返回值类型**只能是基本类型、Class、String、enum**）。可以通过**default**来声明参数的默认值。

**格式**



注解参数可支持的**数据类型**

1、所有基本数据类型（int,float,boolean,byte,double,char,long,short)；

2、String类型；

3、Class类型；

4、enum类型；

5、Annotation类型；

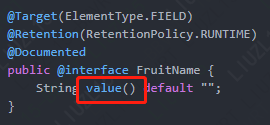
6、以上所有类型的数组。

**访问权修饰符**

只能用public 或者默认这两个访问权修饰。

**注意**

1. 如果注解类中只有一个成员，最好把方法名设置为"value"，比如



注解中定义了名为value的元素，并且在使用该注解时，如果该元素是唯一需要赋值的一个元素，那么此时无需使用key=value 的语法，而只需在括号内给出value元素所需的值即可。这可以应用于任何合法类型的元素，但这限制了元素名必须为 **value**。

1. 注解元素必须有确定的值，要么在定义注解的默认值中指定，要么在使用注解时指定，非基本类型的注解元素的值不可为null。因此, 使用空字符串或0作为默认值是一种常用的做法。

**注解处理器类库**

Java使用**Annotation**接口来代表程序元素前面的**注解**，该接口是所有Annotation类型的父接口。除此之外，Java在java.lang.reflect 包下新增了**AnnotatedElement**接口，主要用于表示目前正在VM中运行的程序中已使用注解的元素，通过该接口提供的方法可以利用反射技术来读取注解的信息，如反射包的Constructor类、Field类、Method类、Package类和Class类都实现了AnnotatedElement接口，接口含义如下：

Class：类定义

Constructor：构造器定义

Field：类的成员变量定义

Method：类的方法定义

Package：类的包定义

java.lang.reflect 包所有提供的反射API扩充了读取运行时Annotation信息的能力。当一个Annotation类型被定义为运行时的Annotation后，该注解才能是运行时可见，当class文件被装载时被保存在class文件中的Annotation才会被虚拟机读取。

**AnnotatedElement 接口是所有程序元素（Class、Method和Constructor）的父接口**，所以程序通过反射获取了某个类的AnnotatedElement对象之后，程序就可以调用该对象的如下四个个方法来访问Annotation信息：

1. <T extends Annotation> T getAnnotation(Class<T> annotationClass)

返回该程序元素上存在的、指定类型的注解，如果该类型注解不存在，则返回null。

1. Annotation[] getAnnotations()

返回该程序元素上存在的所有注解。可以获取到父类使用了 **@Inherited**的注解。

1. boolean isAnnotationPresent(Class<?extends Annotation> annotationClass)

判断该程序元素上是否包含指定类型的注解，存在则返回true，否则返回false。

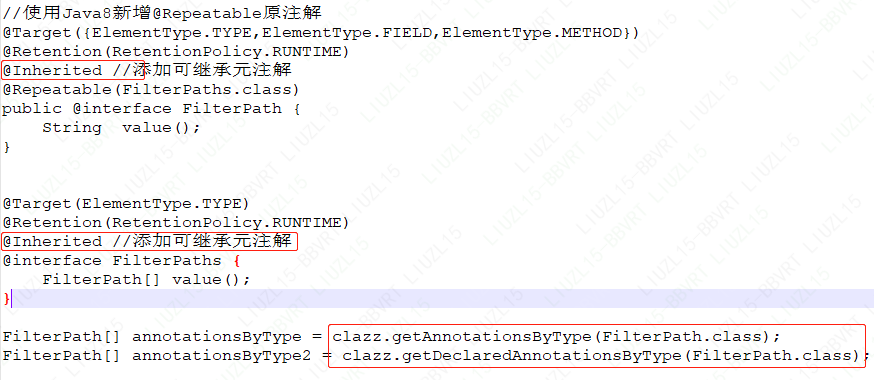
1. Annotation[] getDeclaredAnnotations()

返回直接存在于此元素上的所有注解。与此接口中的其他方法不同，该方法将忽略继承的注释。该方法的调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响。

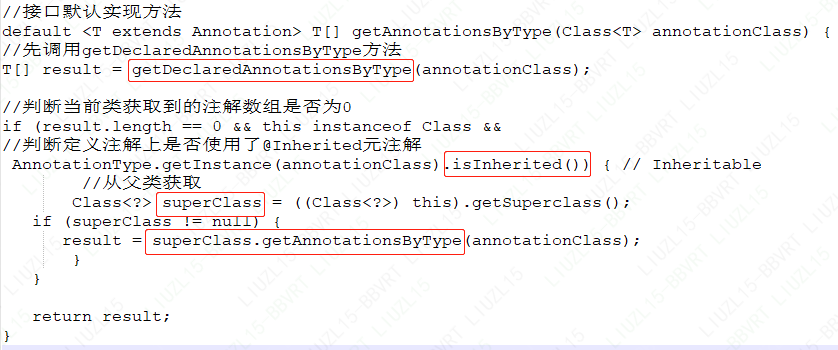
1. getDeclaredAnnotationsByType() 和 getAnnotationsByType()

在指定@Repeatable的注解时，可以通过这两个方法获取到注解相关信息。但请注意，旧版API中的getDeclaredAnnotation()和 getAnnotation()是不对@Repeatable注解的处理的(除非该注解没有在同一个声明上重复出现)。注意getDeclaredAnnotationsByType方法获取到的注解不包括父类，其实当 getAnnotationsByType()方法调用时，其内部先执行了getDeclaredAnnotationsByType方法，只有当前类不存在指定注解时，getAnnotationsByType()才会继续从其父类寻找，但请注意如果@FilterPath和@FilterPaths没有使用了@Inherited的话，仍然无法获取。

使用



**getAnnotationsByType源码实现**



**Android Annotation Support包中的注解介绍**

1. Nullness注解

**@Nullable**:用于标记方法参数或者返回值可以为空；

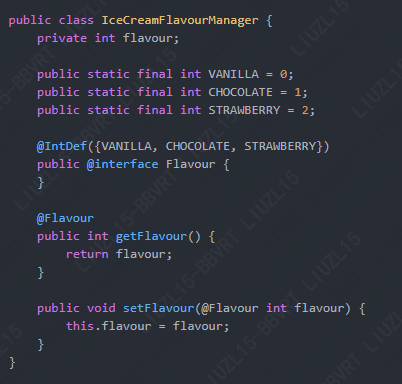
**@NonNull**:用于标记方法参数或者返回值不能为空，如果为空编译器会报警告；

1. 资源类型注解

主要用于标记方法的参数必须要是指定的资源类型，如果不是，IDE就会报错；因为资源文件都是静态的，所以在编写代码时IDE就知道传值是否错误，可以避免传的资源id错误导致运行时异常。资源类型注解包括@AnimatorRes、@AnimRes、@AnyRes、@ArrayRes、@BoolRes、@ColorRes、@DimenRes、@DrawableRes、@FractionRes、@IdRes、@IntgerRes、@InterpolatorRes、@LayoutRes、@MenuRes、@PluralsRes、@RawRes、@StringRes、@StyleableRes、@StyleRes、@TransitionRes、@XmlRes。

1. 类型定义注解

这类注解用于检查“魔幻数”，很多时候，我们使用整型常量代替枚举类型（性能考虑），例如我们有一个IceCreamFlavourManager类，它具有三种模式的操作：VANILLA，CHOCOLATE和STRAWBERRY。我们可以定义一个名为@Flavour的新注解，并使用**@IntDef指定它可以接受的值类型**：



1. 线程注解

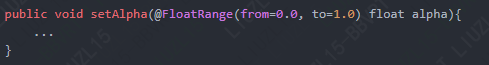
用于标记指定的方法、类（如果一个类中的所有方法都有相同的线程需求，就可以对这个类进行注解，比如View.java就被@UIThread所标记）只能在指定的线程类中被调用，包括：@UiThread、@MainThread、@WorkerThread、@BinderThread。

1. GRB颜色值注解

用于标记传递的颜色值必须是整型值，并且不能是color资源ID；当你的API期望一个颜色资源的时候，可以用**@ColorRes**标注，但是当你有一个相反的使用场景时，这种用法就不可用了，因为你并不是期望一个颜色资源id，而是一个真实的RGB或者ARGB的颜色值。在这种情况下，你可以使用**@ColorInt**注解，表示你期望的是一个代表颜色的整数值：

1. 值约束注解

用于标记参数必须是指定类型的值，并且值的范围必须在**约束的范围**内，包括@Size、@IntRange、@FloatRange。（**from、to**）

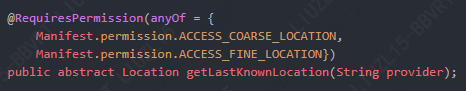


1. 权限注解

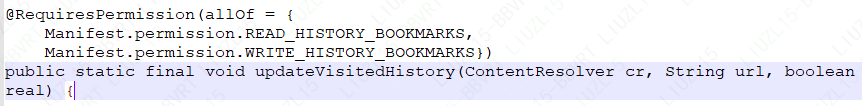
如果你的方法需要调用者有特定的权限，你可以使用**@RequiresPermission**注解。



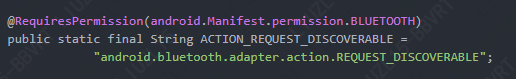
如果你至少需要权限集合中的一个，你可以使用**anyOf**属性。



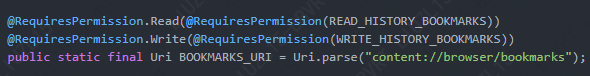
如果你同时需要多个权限，你可以用**allOf**属性。



对于intents的权限，可以直接在定义的**intent常量字符串字段**上标注权限需求(他们通常都已经被**@SdkConstant**注解标注过了)



对于**content providers**的权限，你可能需要单独的标注读和写的权限访问，所以可以用**@Read**或者**@Write**标注每一个权限需求



1. 复写方法注解

如果你的API允许使用者重写你的方法，但你又需要你自己的方法(父方法)在重写的时候也被调用，这时候你可以使用**@CallSuper**标注。



用了这个后，当重写的方法没有调用父方法时，工具就会给予警告提示。

1. 返回值注解

如果你的方法有返回值，你期望调用者用这个值做些事情，那么你可以使用@CheckResult注解标注这个方法。

你并不需要为每个非空方法都进行标注。它主要的目的是帮助哪些容易被混淆，难以被理解的API的使用者。

比如，可能很多开发者都对String.trim()一知半解，认为调用了这个方法，就可以让字符串改变以去掉空白字符。如果这个方法被@CheckResult标注，工具就会对那些没有使用trim()返回结果的调用者发出警告。

Android中，Context#checkPermission这个方法已经被@CheckResult标注了。



这是非常重要的，因为有些使用context.checkPermission的开发者认为他们已经执行了一个权限 —-但其实这个方法仅仅只做了检查并且反馈一个是否成功的值而已。如果开发者使用了这个方法，但是又不用其返回值，那么这个开发者真正想调用的可能是这个Context#enforcePermission方法，而不是checkPermission。

1. 测试可见注解

你可以把这个注解标注到类、方法或者字段上，以便你在测试的时候可以使用他们。

**注解与配置文件的区别**

**配置文件**

1. 使用场合

外部依赖的配置，比如build.gradle中的依赖配置；

同一项目团队内部达成一致的时候；

非代码类的资源文件（比如图片、布局、数据、签名文件等）；

1. 优点

降低耦合，配置集中，容易扩展，比如Android应用多语言支持；

对象之间的关系一目了然，比如strings.xml；

xml配置文件比注解功能齐全，支持的类型更多，比如drawable、style等；

1. 缺点

繁琐；

类型不安全，比如R.java中的都是资源ID，用TextView的setText方法时传入int值时无法检测出该值是否为资源ID，但@StringRes可以；

**注解**

1. 使用场合

动态配置信息；

代为实现程序逻辑（比如xUtils中的@ViewInject代为实现findViewById）；

代码格式检查，比如Override、Deprecated、NonNull、StringRes等，便于IDE能够检查出代码错误；

1. 优点

在class文件中，提高程序的内聚性；

减少重复工作，提高开发效率，比如findViewById。

1. 缺点

如果对annotation进行修改，需要重新编译整个工程；

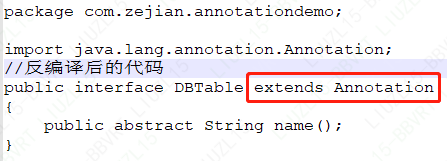
业务类之间的关系不如XML配置那样一目了然；

程序中过多的annotation，对于代码的简洁度有一定影响；

扩展性较差；

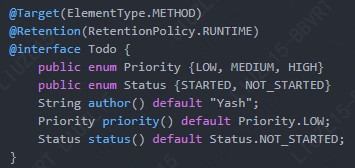
**特别说明**

注解是不支持继承的，因此**不能使用关键字 extends 来继承某个 @interface**，但注解在编译后，编译器会自动继承 java.lang.annotation.Annotation接口。反编译一个注解类：

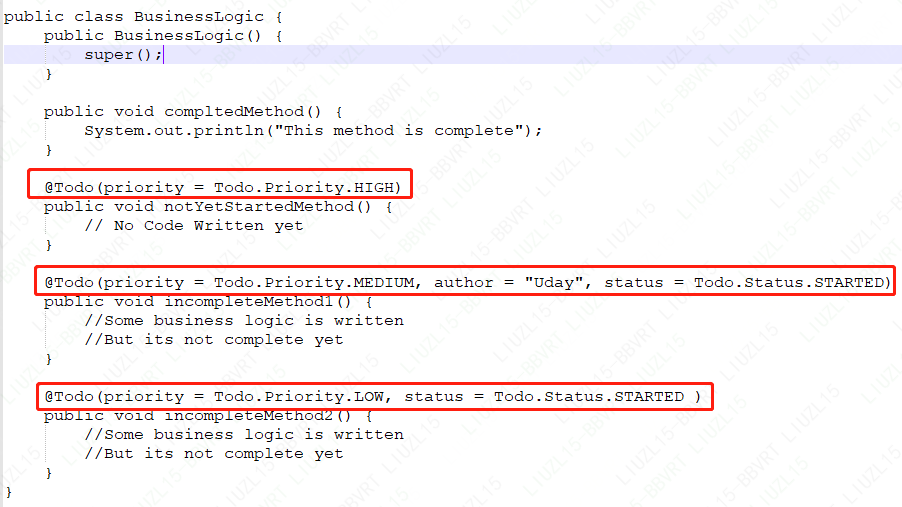


**示例**

1. 注解类



1. 使用注解的类



1. 解析注解信息

