**为什么要引入注解？**

Java 5中引入，主要是为了提供一种机制，允许程序员直接在代码中写入关于代码的元数据。

元数据（metadata）：描述数据的数据。

在引入注解之前，程序员描述自己的代码没有统一的标准，比如通过注释，接口和xml等。但都不是太好的办法，因为都无法把代码和数据耦合起来。

因此Java急需一种新的元数据形式来描述代码，注解被引入。

**注解是什么？**

注解是一种元数据，用来描述代码，提供一些额外信息。

**注解能用来干什么？**

注解不包含任何功能逻辑，也不会直接影响被注解的代码。

某些系统或应用会根据注解信息，对被注解的代码执行不同的动作。

注解的主要用途有：

1. 为编译器提供信息。编译器产生警告或错误信息。
2. 生成文档。有些工具利用注解检测代码质量，例如FindBugs和PMD，有些工具利用注解生成报告。
3. 生成代码。利用注解自动生成代码或XML文件。
4. 运行时处理。运行时解析注解中的信息。

**注解怎样使用？**

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() default -1;

String msg() default “Hi”;

}

// 赋值的方式是在注解的括号内以 属性=””形式，多个属性之前用 ，隔开

@TestAnnotation(id=3,msg="hello annotation")

public class Test {

}

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 关键字指定。

指定了默认值，可以不在@TestAnnotation 后面的括号里面进行赋值。

如果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把属性值填写到括号内。

如果一个注解没有任何属性，在应用这个注解的时候，括号都可以省略。

**注解能被用在哪？**

packages，classes，methods，fields，variables，parameters

**JDK内置注解**

元注解：注解的注解。

1. @Retention。指明注解被保留的时间。

SOURCE：只在源代码中有效，编译器和JVM忽略。

CLASS：被编译器记录到class文件中，但不会被JVM在运行时保留。默认行为。

RUNTIME：被编译器记录到class文件中，并且会被JVM在运行时保留，通过反射可读取。

1. @Target。指定注解用在哪里。

ANNOTATION\_TYPE：用于注解类型声明。

CONSTRUCTOR：用于构造方法声明。

FIELD：用于成员变量声明。

LOCAL\_VARIABLE：用于局部变量声明。

METHOD：用于方法声明。

PACKAGE：用于包声明。

PARAMETER：用于方法参数声明。

TYPE：用于类，接口（包括注解类型），枚举声明。

TYPE\_PARAMETER：Java 8，用于Type参数声明。

TYPE\_USE：Java 8，用于Type。

1. @Documented。注解某个注解@A，被@A注解的元素会被Javadoc工具记录。默认不记录。
2. @Inherited。注解某个注解@B，@B注解某个类SuperClass，那么SuperClass的子类SubClass会继承注解@B。@B只能注解class类型，注解其他元素无效。默认不继承。
3. @Repeatable。Java 8引入。同一个注解可以重复多次用于某个元素。注解的值可以同时取多个的情形下会使用。例如，一个人他既是程序员又是产品经理,同时他还是个画家。

// Person.java

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.TYPE)

@Repeatable(Persons.class)

@interface Person {

String role() default "";

}

// Persons.java

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.TYPE)

@interface Persons {

// 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。

Person[] value();

}

// Test.java

@Person(role = "artist")

@Person(role = "coder")

@Person(role = "PM")

public class Test {

public static void main(String[] args) {

Annotation[] annotations = Test .class.getAnnotations();

for (Annotation annotation : annotations) {

System.out.println(annotation);

}

}

}

其他内置非元注解：

@Deprecated：指示被注解的元素不应该再被使用。使用被注解的元素编译器会生成一条警告信息。如果使用该注解，应该提供弃用该元素的说明和替代方案。

@SuppressWarnings：指示编译器不要产生警告信息。

@Override：指示被注解的方法重写或实现了父类的某个方法。重写或实现父类方式时并非一定要声明该注解，声明的好处是让编译器强制检查重写是否正确，若不正确编译器会产生错误信息。

@SafeVarargs：Java 7引入。用于有可变参数的构造方法或static/final成员方法。在方法参数中如果有可变长度参数，编译器会产生一条警告“Type safety: Potential heap pollution via varargs parameter xxx”，如果程序员确定对可变长度参数的操作都是安全的，可以给方法加上该注解，指示编译器已安全处理，不要产生警告。

// 程序员并没有对可变长度参数进行安全操作，使用@SafeVarargs虽然能压制警告，但运行仍会出// 错，再次说明了该注解只能给编译器提供信息，不会对被注解的程序的运行逻辑产生影响，该崩溃// 还是得崩溃。

static void m(List<String>... stringLists) {

Object[] array = stringLists;

List<Integer> tmpList = Arrays.asList(42);

array[0] = tmpList; // 语义非法，但编译器不会产生警告

String s = stringLists[0].get(0); // 运行时，ClassCastException

}

@FunctionalInterface：Java 8引入。函数式接口可以很容易转换为 Lambda 表达式。

**自定义注解**

**解析注解**

通过反射解析注解。

Class/Method/Field的对象提供如下方法：

// 某个类是否有annotationClass类型注解

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

// 获取类上的annotationClass注解

public <A extends Annotation> A getAnnotation(Class<A> annotationClass)

// 获取类上所有注解

public Annotation[] getAnnotations()

**注解中的继承**

**总结**

**<https://www.javacodegeeks.com/2014/11/java-annotations-tutorial.html>**

**<http://howtodoinjava.com/core-java/annotations/complete-java-annotations-tutorial/>**

**<https://zhuanlan.zhihu.com/p/27643133?utm_source=itdadao&utm_medium=referral>**