# 注解概述

## 元数据的概念

元数据，英文名称Metadata，又称中介数据、中继数据，是为描述数据的数据（data about data），主要是描述数据属性的信息。

例如如下学生表格

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 学生编号 | 学生姓名 | 性别 | 年龄 | 家庭住址 |
| A1001 | 避尘大王 | 男 | 300 | 狮驼岭 |
| A1002 | 避暑大王 | 男 | 500 | 狮驼岭 |
| A1003 | 避寒大王 | 男 | 700 | 狮驼岭 |

该表格的作者 ：小钻风

该表格的创建时间 ： 2019-9-9

该表格本身描述了一些学生信息，对于该表格来讲，其作者，创建时间等信息则是元数据，是为数据的数据。

## 注解的概念

注解，英文名称Annotation,是一种标记，这些标记可以在编译，类加载，运行时被读取，并执行相应的处理。通过使用注解开发人员可以在不改变原有逻辑的情况下，在源文件中嵌入一些补充信息。代码分析工具，开发工具和部署工具可以通过这些补充信息进行验证或者部署。

## 注解的作用

注解能够被用来为程序元素(类，方法，成员变量等)设置元数据。需要指出的是，注解不影响程序代码执行，无论增加，删除注解，代码都始终如一地执行。

如果希望让程序中的注解在运行时起一定作用，只能通过某种配套工具对注解中的信息进行访问和处理，而访问和处理的注解工具统称APT(Annotation Processing Tool)。

# 基本注解

Java 从Java 5开始支持注解。

了解了注解的概念以及作用以后，我们来学习一下Java提供的5个基本注解，这5个最基本的注解是：

* @Override
* @Deprecated
* @SuppressWarnings
* @SafeVarargs
* @FunctionalInterface

注意：使用注解时要在注解前加@符号。使用注解很简单，只需要把注解当成一个修饰符来修饰它支持的程序元素即可。

## @Override

@Override注解用来指定子类必须重写父类方法。也就是说如果一个方法被@Override注解所修饰，那么改方法必须是重写了父类的方法，否则编译报错。

示例代码

1. 声明Fu.java

**public** **class** Fu {

**public** **void** eat() {}

}

1. 声明Zi.java，继承Fu

**public** **class** Zi **extends** Fu {

@Override

**public** **void** eat() {}//只要不符合方法重写的规则就编译报错}

}

@Override注解主要是用来帮助开发者避免一些低级错误，例如在子类重写父类方法时候不小心将子类的方法名称给写错误。就像这样，本来想写的是execute()结果却写成了executer();这个时候就成了子类独有的方法，而不是子类重写父类的方法。

## @Deprecated

@Deprecated注解用于指示程序元素(类，方法等)已过时。

示例代码

**public** **class** Zi **extends** Fu {

@Deprecated

@Override

**public** **void** ~~eat~~() {}//可以看出来一个程序元素可以被多个注解修饰

}

## @SuppressWarnings

@SuppressWarnings注解用于取消编译器提示的警告信息。@SuppressWarnings会一直作用于该程序元素的所有子元素。例如使用@SuppressWarnings修饰某个类取消编译器的某个警告，同时又使用@SuppressWarnings修饰类中的某个方法取消编译器的另一个警告，那么该方法将会同时取消编译器的这两个警告。

### 抑制单一类型警告

编译器的警告信息有多种多样，在抑制单一类型警告这种情况中@SuppressWarnings只负责抑制一种类型的警告信息。

示例代码

@SuppressWarnings(value = "unchecked")

**public** **static** **void** main(String[] args) {

@SuppressWarnings(value = "rawtypes")

// 没有泛型的集合，编译器会给出警告信息,rawtypes用于取消编译给出new Arraylist()警告

List list = **new** ArrayList();

//没有泛型的集合添加元素，可能会引发unchecked异常，所以编译器会给出警告信息，unchecked用于取消添加元素时的警告信息

list.add(**new** Object());

}

### 抑制多种类型警告

为了方便起见，也可以让一个@SuppressWarnings注解抑制多种类型的警告信息。

示例代码

@SuppressWarnings(value = { "rawtypes", "unchecked" })

// 使用一个@SuppressWarnings注解抑制多种警告信息

**public** **static** **void** main(String[] args) {

List list = **new** ArrayList();

list.add(**new** Object());

}

### 抑制所有类型警告

顾名思义，就是让一个@SuppressWarnings注解抑制所有的类型的警告信息

示例代码

@SuppressWarnings(value = "all")

**public** **static** **void** main(String[] args) {

List list = **new** ArrayList();

list.add(**new** Object());

}

### @SuppressWarnings的参数值

@SuppressWarnings注解定义在java.lang.SuppressWarnings中，用来抑制编译时的警告信息。使用它的时候需要添加一个参数，这些参数值都是已经定义好了的。具体看下表

|  |  |
| --- | --- |
| **参数名称** | **说明** |
| Deprecation | 使用了过时的类或者方法的警告 |
| Unchecked | 执行了未检查的转换的警告，例如使用集合时候未指定泛型 |
| fallthrough | switch语句中发生case穿透编译警告 |
| Path | 在类路径，源文件路径等中有不存在路径的编译警告 |
| Serial | 可序列化的类中缺少serialVersionUID定义的警告 |
| Finally | 任何finally子句不能完成时警告 |
| All | 关于以上所有的警告 |

## @SafeVarargs

在Java中，将一个不带泛型的对象赋值给一个带泛型的变量，这种情况称之为”堆污染(heap pollution)”。发生”堆污染”以后，代码并不是固定的运行出错，也不是固定的运行正常，而是根据实际情况来定的。

### “堆污染”示例

示例代码一：

**public** **static** **void** main(String[] args) {

List listA = **new** ArrayList<String>();

listA.add("小钻风");

List<Integer> listB = **new** ArrayList<>();

// "堆污染"代码，编译运行不报错

listB = listA;

// 运行报错，类转换异常

Integer integer = listB.get(0);

System.***out***.println(integer);

}

分析 ： 如果将ListB的泛型定义成String类型，那么运行的时候就不会报错。所以发生“堆污染”以后，代发是否运行报错是根据实际情况而定的。

### @SafeVarargs示例

注解@SafeVarargs就是用来取消编译器的堆污染警告信息。

示例代码二

Java不支持泛型数组，所以对于参数可变的方法，该方法的形式参数又带泛型，将更容易导致“堆污染”。因为Java在运行的时候会擦除泛型，因此程序在运行的时候会把List<Integer>…当成List[]处理，这就是发生“堆污染”的原因。

@SuppressWarnings(value = {"rawtypes", "unused" })

@SafeVarargs //取消堆污染警告信息

**public** **static** **void** addList(List<Integer>... lists) {

// java不支持泛型数组，这里已经发生堆污染

List[] listArray = lists;

List<String> aLists = **new** ArrayList<>();

listArray[0] = aLists;

aLists.add("小钻风");

//运行报错，将lists泛型修改成String，那么运行就又不报错了(修改以后记得用String接受哦)

Integer xiaozuanfeng = lists[0].get(0);

}

解析：取消编译器的堆污染警告信息并不意味着运行代码固定正常，它仅仅是取消编译器的警告信息而已。要始终记住，注解，只是一种符号，它帮助开发者在不改变原有逻辑的情况下进行补充，嵌入一些信息等操作。注解的删除，增加并不会影响代码的执行。

### 泛型数组

Java虽然不支持泛型数组，但是却可以声明变量是泛型数组类型，这样就可以将运行时期的错误提前到编译期间，保证代码运行正常。这也正是泛型的好处之一。

@SafeVarargs //取消堆污染编译警告信息

**public** **static** **void** addList(List<Integer>... lists) {

//java中不支持泛型数组，创建这种对象会报错

//new ArrayList<Integer>[10];

//跟上次相比，这次的变量类型是泛型数组，那么在编译期间就可以检查元素类型

List<Integer>[] listArray = lists;

List<String> aLists = **new** ArrayList<>();

//编译报错，避免运行时错误

//listArray[0] = aLists;

aLists.add("小钻风");

Integer xiaozuanfeng = lists[0].get(0);

}

### 小结

通过以上三个示例代码，可以总结出来，发生”堆污染”的代码运行时是否异常，要根据实际情况而定。基于这种情况，Java 7在定义方法的时候就发出”堆污染”警告，让开发者更早地注意到程序中可能存在的漏洞。当开发者不希望看到这个警告的时候，有如下三种解决方案

1. 使用@SafeVarargs注解修饰引发堆污染警告的方法或构造器
2. 使用@SuppressWarnings(“unchecked”)修饰
3. 在编译时使用-Xlint:varargs选项

很显然，第三种方式是很少使用，通常选择第一种或者第二种方式。尤其第一种方式，它是Java 7专门为抑制堆污染警告提供的。

## @FunctionalInterface

该注解的功能很简单，它用来指示某个接口必须是函数式接口。所以该注解只能修饰接口，不能修饰其它程序元素。

函数式接口就是为Java 8的Lambda表达式准备的，Java 8允许使用Lambda表达式创建函数式接口的实例，因此Java 8专门增加了@FunctionalInterface注解。

示例代码

@FunctionalInterface

**public** **interface** FunctionalInterfaceTest {

/\*

\* 只要接口中的抽象方法数量不是一，则编译报错

\* 所以放开test2()或者注释掉test()都会编译报错

\* 所以该注解就是用来帮助开发者避免一些低级错误

\*/

**public** **abstract** **void** test();

// public abstract void test2();

}

# JDK的元注解

元注解，英文名称Meta Annotation，是作用于注解的注解。换言之，元注解是用于声明，创建注解的时候用到的标记，符号。Jdk 在java.lang包下提供了五个基本注解(基本注解章节的五个注解)的同时，又在java.lang.annotation包下提供了六个元注解。这个六个元注解中只有五个元注解可以用于修饰其它注解，较为特殊的一个则是Java 8新增的重复注解。

这六个元注解分别是

* **@Retention**
* **@Target**
* **@Documented**
* **@Inherited**
* **@Repeatable**

## @Retention

设想一下，你要创建一个注解给其它开发者使用。正如你创建一个类给其它开发者使用一样，你必须指定该类的某些规则，比如指定该类只能连接MySQL数据库。注解也一样，在创建的时候需要指定该注解的某些规则，比如可以保留多长时间。

@Retention注解就是在声明创建一个注解的时候，指定这个被声明创建的注解可以保留多长时间。它里边有一个RetentionPolicy类型的value变量，该变量的值只有三个，不同的值代表不同的保留规则。

具体保留规则如下：

* **RetentionPolicy.CLASS**

编译器将注解记录在.class文件中。当运行Java程序时，JVM不能获取注解信息。这是默认值

* **RetentionPolicy.RUNTIME**

编译器将注解记录在.class文件中。当运行Java程序时，JVM也能获取注解信息。程序可以通过反射获取注解信息。

* **RetentionPolicy.SOURCE**

编译器直接丢弃注解。所以该注解只能保留在源代码文件中。

注意，使用@Retention注解必须为value变量赋值。

示例代码

//创建声明Test注解，该注解被使用的时候只能可以保留到运行阶段

@Retention(value = RetentionPolicy.***RUNTIME***)

**public** **@interface** Test {}

//创建声明Test注解，该注解编译器将直接丢弃，所以只能保留在源代码中

@Retention(RetentionPolicy.***SOURCE***)

**public** **@interface** Test{}

总结 ： 声明创建注解要使用@interface这种格式

使用注解的时候，如果只需要给注解的value属性赋值，直接将value属性的值写出即可，无需再使用key=value形式。

## @Target

@Target注解是在声明创建一个注解的时候，指示该注解可以作用于程序中的哪些元素。它里边也包含一个名为value的成员变量，value成员变量的值有如下几种

* ElementType.***ANNOTATION\_TYPE：***指定当前注解只能修饰其它注解
* ElementType.***CONSTRUCTOR：***指定当前注解只能修饰构造方法
* ElementType.***FIELD：***指定当前注解只能修饰成员变量
* ElementType.***LOCAL\_VARIABLE：***指定当前注解只能修饰局部变量
* ElementType.***METHOD：***指定当前注解只能修饰方法
* ElementType.***PACKAGE：***指定当前注解只能修饰包
* ElementType.***PARAMETER：***指定当前注解只能修饰参数
* ElementType.***TYPE：***指定当前注解可以修饰类，接口，其它注解，枚举等类型

示例代码

//创建声明Test注解，该注解只能修饰成员变量

@Target(ElementType.***FIELD***)

**public @interface** Test {}

//创建声明Test注解，该注解只能修饰方法

@Target(ElementType.***METHOD***)

**public @interface** Test {}

## @Documented

@Documented元注解是在声明创建一个注解的时候，指示该注解将被javadoc工具提取成文档。这样所有使用该注解的程序元素的API文档中都会包含该注解说明。

示例代码

声明一个使用@Documented注解的注解

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***METHOD***)

@Documented

**public** **@interface** MyDocument {}

声明一个测试类，该类使用了@MyDocument注解

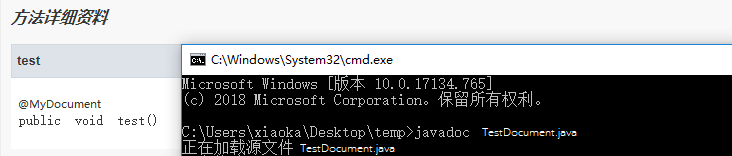
**public** **class** TestDocument {

@MyDocument

**public** **void** test() {}

}

效果图



可以看到，javadoc提取TestDocument.java成API文档的时候，将注解@MyDocument的说明信息也提取到了文档中。

## @Inherited

@Inherited注解是在声明创建一个注解的时候，指定该注解将具有继承性：如果某个类使用了该注解，则其子类也将自动被改注解所修饰。

示例代码

声明创建一个注解

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***TYPE***)

@Inherited

**public** **@interface** MyInherited {}

创建一个父类

@MyInherited

**public** **class** MyInheritedTestFu {}

所有的子类都将自动将@MyInherited注解给继承

创建子类进行测试

**public** **class** MyInheritedTestZi **extends** MyInheritedTestFu {

**public** **static** **void** main(String[] args) {

**boolean** a = MyInheritedTestZi.**class**.isAnnotationPresent(MyInherited.**class**);

System.***out***.println(a);

}

}

打印结果为true，可以看到虽然子类没有使用@MyInherited注解，却依然是被@MyInherited注解所修饰。如果将@MyInherited

的@Inherited元注解去掉，那么子类将无法从父类继承@MyInherited注解。打印结果将会是false.

## @Repeatable

某些情况下，一个程序元素需要使用多个同一种类型的注解。@Repeatable元注解，顾名思义，重复注解，就是在声明创建注解的时候，指定该注解可以被同一个程序元素多次使用。

例如一个公司的创始人，它既是CEO，又是产品经理，还要负责财务，招聘等等，具有多种角色身份。这种情况就可以使用重复注解。

示例代码

声明注解和注解容器

/\* 声明注解，该注解可以被程序元素重复使用 \*/

@Repeatable(Roles.**class**)

**public** **@interface** Role {

String value();

}

/\* 定义注解"容器" : Roles里边是Role的数组，这样Roles里边就可以存放多个Role \*/

**public** **@interface** Roles {

Role[] value();

}

使用重复注解

@Role("CEO")

@Role("UFO")

**public** **class** People { }

注意 ：

1. “容器”注解Roles的保留期必须比它所包含的注解Role的保留期更长，否则编译器会报错。
2. 重复注解是Java 8新增的元注解
3. Java 8以前不允许重复注解，所以之前都只能用如下形式来表达

//直接使用Roles来盛装Role

@Roles({ @Role("CEO"), @Role("UFO") })

**public** **class** People {}

1. 重复注解不是一个语言上的改变，只是编译器层面的改动，技术层面仍然是一样的。其本质依然是多个重复注解作为了“容器”注解的value成员变量的数组元素。本质代码请看注意3

# 自定义注解

在JDK的元注解一章，虽然没有正式介绍，但是相各位已经知道，声明一个注解很简单，就是使用@interface关键字即可。接下来，我们学习如何给自己创建的注解增加一些实际的功能。

## 自定义注解

我们首先来看一下自定义注解的注意事项 ：

* 定义注解需要使用@interface关键字，定义格式 ： public @interface 注解名称，其中public是权限修饰符。

示例代码

**pbulic @interface** Test {}

* 使用注解很简单，只需要将注解像public,final等修饰符一样，放在程序元素的前面或者上面即可。一般会选择放在程序元素的上面。默认情况下，注解可以修饰程序中的任何元素

示例代码

@Test

**public** **class** TestAnnotation {

@Test

**private** String name;

@Test

**public** **void** test1() {

System.***out***.println("test1");

}

}

* 定义注解可以为注解声明成员变量，格式 ：数据类型 成员变量名称(); 注意，跟声明方法形式很像，但这是注解的成员变量，不是方法。
* 注解中的成员变量只能是八种基本数据类型，以及String,Class,Enum等

示例代码

**@interface** Test {

//声明注解的成员变量 数据类型 成员变量名称();

String name();

**int** value();

String[] names();

}

* 使用注解的时候必须为注解中所有的成员变量赋值，否则编译报错。

示例代码

@Test(name = "小钻风", value = 300,names= {"避尘大王","避暑大王","避寒大王"})

**public** **class** TestAnnotation {

@Test(name = "小钻风", value = 300,names= {"避尘大王","避暑大王","避寒大王"})

**private** String name;

@Test(name = "小钻风", value = 300,names= {"避尘大王","避暑大王","避寒大王"})

**public** **void** test1() {

System.***out***.println("test1");

}

}

* 可以为注解的成员变量声明默认值，格式 defalut value

示例代码

**@interface** Test {

String name();

**int** value();

String[] names() **default** {"避尘大王","避暑大王","避寒大王"};

}

* 使用注解的时候，带默认值的成员变量可以不赋值，这个时候使用的是注解中成员变量的默认值

示例代码

@Test(name = "小钻风", value = 300)

**public** **class** TestAnnotation {

@Test(name = "小钻风", value = 300)

**private** String name;

@Test(name = "小钻风", value = 300)

**public** **void** test1() {

System.***out***.println("test1");

}

}

* 使用注解的时候，如果只需要给注解的value成员变量赋值，则成员变量名称可以省略，只赋值即可

示例代码

@Test(name = "小钻风", value = 300) //value不能省略

**public** **class** TestAnnotation {

@Test(name="小钻风",value = 300) //value不能省略

**private** String name;

@Test(300) //value可以省略

**public** **void** test1() {

System.***out***.println("test1");

}

}

**@interface** Test {

String name() **default** "小钻风";

**int** value();

String[] names() **default** { "避尘大王", "避暑大王", "避寒大王" };

}

* Java.lang.Annotation接口是所有注解的父接口，所以，注解一旦声明创建，便自动继承java.lang.Annotation接口

总结 ： 根据注解是否包含成员变量，可以把注解分为两类

* 标记注解

没有定义成员变量的注解被称为标记。这种注解仅利用自身的存在与否来提供信息，例如@Override。

* 元数据注解

包含成员变量的注解可以接受更多的元数据，所以也被称为元数据注解。这种注解，其本身，其成员变量都可以用来提供信息

## 获取注解信息

使用注解修饰了程序元素以后，这些注解不会自己生效，必须由相应的工具类来提取并处理才能生效。这些工具类都在java.lang.reflect包中。从Java 5开始，java.lang.reflect包所提供的反射AIP增加了读取运行时注解的能力。[只有当定义注解的时候使用了@Retention(RetentionPolicy.RUNTIME)](mailto:只有当定义注解的时候使用了@Retention(RetentionPolicy.RUNTIME))修饰，该注解才会在运行时可见,JVM才会在装载.class文件时读取保存在.class文件中的注解。

示例代码

一：声明注解

@Retention(RetentionPolicy.***RUNTIME***)

**@interface** Test {

String name() **default** "小钻风";

**int** value();

String[] names() **default** { "避尘大王", "避暑大王", "避寒大王" };

}

二：创建测试类

@Test(name = "小钻风", value = 300) // value不能省略

**public** **class** TestAnnotation {

@Test(name = "小钻风", value = 300) // value不能省略

**private** String name;

@Test(300) // value可以省略

**public** **void** test1() {System.***out***.println("test1");}

**public** **static** **void** main(String[] args) throw Exception{

Class<TestAnnotation> classT = null;

classT = (Class<TestAnnotation>) Class.*forName*("com.annotation.TestAnnotation");

Method method = classT.getMethod("test1");

System.***out***.println(method);

//获取指定方法的所有注解

Annotation[] annotations = classT.getMethod("test1").getAnnotations();

**for** (Annotation annotation : annotations) {

System.***out***.println(annotation);

//将注解强转成所需要的注解类型

**if** (annotation **instanceof** Test) {

Test temp = (Test) annotation;

//通过注解对象的抽象方法来访问注解中的元数据

System.***out***.println("name :" + temp.name());

System.***out***.println("value :"+temp.value());

String[] names = temp.names();

**for** (String tempName : names) {

System.***out***.println("names :"+tempName);

}

}

}

}

## 小结

Class 、Constructor、 Method、 Field等程序元素均是AnnotatedElement接口的子类。当通过反射获取到某个程序元素以后，就可以通过其父接口AnnotatedElement提供的方法来获取该程序元素上的注解信息。

AnnotatedElement接口提供了如下几个方法帮助开发者获取注解信息

<T **extends** Annotation> T getAnnotation(Class<T> annotationClass);--返回程序元素上指定类型注解

**default** <T **extends** Annotation> T getDeclaredAnnotation(Class<T> annotationClass)

-- java8新增方法，返回直接修饰该程序元素，指定类型的注解

Annotation[] getAnnotations();-- 返回程序元素上所有的注解

Annotation[] getDeclaredAnnotations();-- 返回直接修饰该程序元素的所有注解

**default** <T **extends** Annotation> T[] getAnnotationsByType(Class<T> annotationClass)

--- java8新增方法，与重复注解有关，获取指定类型的多个注解

**default** <T **extends** Annotation> T[] getDeclaredAnnotationsByType(Class<T> annotationClass)

--- java8新增方法，与重复注解有关，获取直接修饰该程序元素，指定类型的多个注解

**default** **boolean** isAnnotationPresent(Class<? **extends** Annotation> annotationClass)

-- 判断该程序元素上是否存在指定类型的注解

示例代码 ： 获取重复注解

**public** **static** **void** main(String[] args) **throws** ClassNotFoundException {

Class<People> clazz = (Class<People>) Class.*forName*("com.annotation.People");

// java 8 以前获取重复注解的方式

Roles roles = clazz.getDeclaredAnnotation(Roles.**class**);

Role[] rolesArray = roles.value();

**for** (Role role : rolesArray) {

System.***out***.println(role.value());

}

// java 8新增获取重复注解的方式

Role[] roleArray = clazz.getDeclaredAnnotationsByType(Role.**class**);

**for** (Role role : roleArray) {

System.***out***.println(role.value());

}

}

# 类型注解

Java 8为ElementType枚举增加了TYPE\_PARAMETER和TYPE\_USE两个枚举值。[这样就允许定义注解时使用@Target(ElementType.TYPE\_USE)](mailto:这样就允许定义注解时使用@Target(ElementType.TYPE_USE))修饰，这种注解就是类型注解。

在Java 8以前，注解只能在定义程序元素的时候使用。从Java 8开始，类型注解可以在任何使用类型的地方使用。例如如下位置 ：

* 创建对象
* 类型转换
* 使用implements实现接口
* 使用throws 声明抛出异常

上面这些位置都会使用到类型，因此可以使用类型注解来修饰。

## TYPE\_PARAMETER

直译为类型变量，被该值所修饰，意味着声明的注解可以作用在泛型类，泛型接口，泛型方法上。

示例代码

//声明注解

// 声明注解

@Target({ ElementType.***TYPE\_PARAMETER***})

**public** **@interface** NotNull {}

//测试注解

**public** **class** TestNotNull<@NotNull T> {

**public** <@NotNull T> **void** testT(T x) {}

}

**public** **interface** A<@NotNull T> {}

## TYPE\_USE

示例代码

// 声明类型注解

@Target(ElementType.***TYPE\_USE***)

**public** **@interface** NotNull {}

使用类型注解

@NotNull // 声明类使用类型注解，在继承和实现的时候都使用类型注解

**public** **class** TestNotNull<@NotNull T> **extends** @NotNull HashMap<String, String> **implements** @NotNull Serializable {

@NotNull

**private** T t;

// 声明成员变量使用类型注解

@NotNull

**private** String test;

// 方法中使用类型注解，抛异常使用类型注解

**public** **static** **void** main(@NotNull String[] args) **throws** @NotNull Exception {

// 声明局部变量使用类型注解

@NotNull

String string;

// 创建对象使用类型注解

string = **new** @NotNull String("小钻风");

// 强制类型转换使用类型注解

Object object = (@NotNull Object) string;

System.***out***.println(object);

}

@NotNull // 声明方法使用类型注解， 泛型使用类型注解

**public** String test(List<@NotNull String> lists) {

**return** "";

}

// @NotNull 返回值为空意味着没有类型返回，所以不能够使用类型注解

**public** **void** test2() {}

**public** <@NotNull T> **void** testT(@NotNull T xddf) {

}

}

## 注意

在疯狂Java讲义一书中只明确提出了使用TYPE\_USE的才是类型注解，并没有提到使用TYPE\_PARAMETER也是类型注解，也没有否认使用TYPE\_PARAMETER不是类型注解。有任何问题，欢迎提出。

类型变量一词来源于网络博客，未曾在任何书籍中见到过对于类型变量的定义和解释，仅在测试中发现类型变量属于泛型定义，有任何问题，欢迎提出。