# 概念

## 什么是注解（Annotation）

**Annotation（注解）**就是Java提供了一种元程序中的元素关联任何信息和着任何元数据（metadata）的途径和方法。

Annotion(注解)是一个接口，程序可以通过反射来获取指定程序元素的Annotion对象，然后通过Annotion对象来获取注解里面的元数据。

Annotation(注解)是JDK5.0及以后版本引入的。它可以用于创建文档，跟踪代码中的依赖性，甚至执行基本编译时检查。从某些方面看，annotation就像修饰符一样被使用，并应用于包、类型、构造方法、方法、成员变量、参数、本地变量的声明中。这些信息被存储在Annotation的“name=value”结构对中。

Annotation的成员在Annotation类型中以无参数的方法的形式被声明。其方法名和返回值定义了该成员的名字和类型。在此有一个特定的默认语法：允许声明任何Annotation成员的默认值：一个Annotation可以将name=value对作为没有定义默认值的Annotation成员的值，当然也可以使用name=value对来覆盖其它成员默认值。这一点有些近似类的继承特性，父类的构造函数可以作为子类的默认构造函数，但是也可以被子类覆盖。

Annotation能被用来为某个程序元素（类、方法、成员变量等）关联任何的信息。需要注意的是，这里存在着一个基本的规则：Annotation不能影响程序代码的执行，无论增加、删除 Annotation，代码都始终如一的执行。另外，尽管一些annotation通过java的反射api方法在运行时被访问，而java语言解释器在工作时忽略了这些annotation。正是由于java虚拟机忽略了Annotation，导致了annotation类型在代码中是“不起作用”的； 只有通过某种配套的工具才会对annotation类型中的信息进行访问和处理。本文中将涵盖标准的Annotation和meta-annotation类型，陪伴这些annotation类型的工具是java编译器（当然要以某种特殊的方式处理它们）。

## 什么是metadata（元数据）

元数据从metadata一词译来，就是“关于数据的数据”的意思。

元数据的功能作用有很多，比如：你可能用过Javadoc的注释自动生成文档。这就是元数据功能的一种。总的来说，元数据可以用来创建文档，跟踪代码的依赖性，执行编译时格式检查，代替已有的配置文件。如果要对于元数据的作用进行分类，目前还没有明确的定义，不过我们可以根据它所起的作用，大致可分为三类：

1. 编写文档：通过代码里标识的元数据生成文档
2. 代码分析：通过代码里标识的元数据对代码进行分析
3. 编译检查：通过代码里标识的元数据让编译器能实现基本的编译检查

在Java中元数据以标签的形式存在于Java代码中，元数据标签的存在并不影响程序代码的编译和执行，它只是被用来生成其它的文件或针在运行时知道被运行代码的描述信息。

综上所述：

1. 元数据以标签的形式存在于Java代码中。
2. 元数据描述的信息是类型安全的，即元数据内部的字段都是有明确类型的。
3. 元数据需要编译器之外的工具额外的处理用来生成其它的程序部件。
4. 元数据可以只存在于Java源代码级别，也可以存在于编译之后的Class文件内部。

## Annotation和Annotation类型

### Annotation

Annotation使用了在java5.0所带来的新语法，它的行为十分类似public、final这样的修饰符。每个Annotation具有一个名字和成员个数>=0。每个Annotation的成员具有被称为name=value对的名字和值（就像javabean一样），name=value装载了Annotation的信息。

### Annotation类型

Annotation类型定义了Annotation的名字、类型、成员默认值。一个Annotation类型可以说是一个特殊的java接口，它的成员变量是受限制的，而声明Annotation类型时需要使用新语法。当我们通过java反射api访问Annotation时，返回值将是一个实现了该 annotation类型接口的对象，通过访问这个对象我们能方便的访问到其Annotation成员。后面的章节将提到在java5.0的 java.lang包里包含的3个标准Annotation类型。

## 注解的分类

根据注解参数的个数，我们可以将注解分为三类：

1. 标记注解：一个没有成员定义的Annotation类型被称为标记注解。这种Annotation类型仅使用自身的存在与否来为我们提供信息。比如后面的系统注解@Override;
2. 单值注解
3. 完整注解

根据注解使用方法和用途，我们可以将Annotation分为三类：

1. JDK内置系统注解
2. 元注解
3. 自定义注解

### 内置系统注解

注解的语法比较简单，除了@符号的使用外，他基本与Java固有的语法一致，JavaSE中内置三个标准注解，定义在java.lang中：

**@Override**：用于修饰此方法覆盖了父类的方法;

**@Deprecated**：用于修饰已经过时的方法;

**@SuppressWarnnings**：用于通知java编译器禁止特定的编译警告。

下面我们依次看看三个内置标准注解的作用和使用场景。

#### @Override

限定重写父类方法。

@Override 是一个标记注解类型，它被用作标注方法。它说明了被标注的方法重载了父类的方法，起到了断言的作用。如果我们使用了这种Annotation在一个没有覆盖父类方法的方法时，java编译器将以一个编译错误来警示。这个annotaton常常在我们试图覆盖父类方法而确又写错了方法名时发挥威力。使用方法极其简单：在使用此annotation时只要在被修饰的方法前面加上@Override即可。下面的代码是一个使用@Override修饰一个企图重载父类的displayName()方法，而又存在拼写错误的实例：

#### @Deprecated

标记已过时。

同样Deprecated也是一个标记注解。当一个类型或者类型成员使用@Deprecated修饰的话，编译器将不鼓励使用这个被标注的程序元素。而且这种修饰具有一定的 “延续性”：如果我们在代码中通过继承或者覆盖的方式使用了这个过时的类型或者成员，虽然继承或者覆盖后的类型或者成员并不是被声明为 @Deprecated，但编译器仍然要报警。

***注意：@Deprecated这个annotation类型和javadoc中的 @deprecated这个tag是有区别的：前者是java编译器识别的，而后者是被javadoc工具所识别用来生成文档（包含程序成员为什么已经过 时、它应当如何被禁止或者替代的描述）。***

在java5.0，java编译器仍然象其从前版本那样寻找@deprecated这个javadoc tag，并使用它们产生警告信息。但是这种状况将在后续版本中改变，我们应在现在就开始使用@Deprecated来修饰过时的方法而不是 @deprecated javadoc tag。

#### @SuppressWarnnings

抑制编译器警告。

@SuppressWarnings 被用于有选择的关闭编译器对类、方法、成员变量、变量初始化的警告。在java5.0，sun提供的javac编译器为我们提供了-Xlint选项来使编译器对合法的程序代码提出警告，此种警告从某种程度上代表了程序错误。例如当我们使用一个generic collection类而又没有提供它的类型时，编译器将提示出"unchecked warning"的警告。通常当这种情况发生时，我们就需要查找引起警告的代码。如果它真的表示错误，我们就需要纠正它。例如如果警告信息表明我们代码中的switch语句没有覆盖所有可能的case，那么我们就应增加一个默认的case来避免这种警告。

有时我们无法避免这种警告，例如，我们使用必须和非generic的旧代码交互的generic collection类时，我们不能避免这个unchecked warning。此时@SuppressWarning就要派上用场了，在调用的方法前增加@SuppressWarnings修饰，告诉编译器停止对此方法的警告。

SuppressWarning不是一个标记注解。它有一个类型为String[]的成员，这个成员的值为被禁止的警告名。对于javac编译器来讲，被-Xlint选项有效的警告 名也同样对@SuppressWarings有效，同时编译器忽略掉无法识别的警告名。

SuppressWarnings注解的常见参数值的简单说明：

|  |  |
| --- | --- |
| **Type** | **Descption** |
| all | to suppress all warnings |
| boxing | to suppress warnings relative to boxing/unboxing operations |
| cast | to suppress warnings relative to cast operations |
| dep-ann | to suppress warnings relative to deprecated annotation |
| deprecation | to suppress warnings relative to deprecation |
| fallthrough | to suppress warnings relative to missing breaks in switch statements |
| finally | to suppress warnings relative to finally block that don’t return |
| hiding | to suppress warnings relative to locals that hide variable |
| incomplete-switch | to suppress warnings relative to missing entries in a switch statement (enum case) |
| nls | to suppress warnings relative to non-nls string literals |
| null | to suppress warnings relative to null analysis |
| rawtypes | to suppress warnings relative to un-specific types when using generics on class params |
| restriction | to suppress warnings relative to usage of discouraged or forbidden references |
| serial | to suppress warnings relative to missing serialVersionUID field for a serializable class |
| static-access | to suppress warnings relative to incorrect static access |
| synthetic-access | to suppress warnings relative to unoptimized access from inner classes |
| unchecked | to suppress warnings relative to unchecked operations |
| unqualified-field-access | to suppress warnings relative to field access unqualified |
| unused | to suppress warnings relative to unused code |

### 元注解

元注解的作用就是负责注解其他注解。

Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。Java5.0定义的元注解：

1. **@Target**
2. **@Retention**
3. **@Documented**
4. **@Inherited**

这些类型和它们所支持的类在java.lang.annotation包中可以找到。下面我们看一下每个元注解的作用和相应分参数的使用说明。

#### @Target

@Target说明了Annotation所修饰的对象范围：Annotation可被用于 packages、types（类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数）。在Annotation类型的声明中使用了target可更加明晰其修饰的目标。

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）

取值(ElementType)有：

* CONSTRUCTOR:用于描述构造器
* FIELD:用于描述域
* LOCAL\_VARIABLE:用于描述局部变量
* METHOD:用于描述方法
* PACKAGE:用于描述包
* PARAMETER:用于描述参数
* TYPE:用于描述类、接口(包括注解类型) 或enum声明

#### @Retention

@Retention定义了该Annotation被保留的时间长短：某些Annotation仅出现在源代码中，而被编译器丢弃；而另一些却被编译在class文件中；编译在class文件中的Annotation可能会被虚拟机忽略，而另一些在class被装载时将被读取（请注意并不影响class的执行，因为Annotation与class在使用上是被分离的）。使用这个meta-Annotation可以对 Annotation的“生命周期”限制。

作用：表示需要在什么级别保存该注释信息，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）

取值（RetentionPoicy）有：

* 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）

Retention meta-annotation类型有唯一的value作为成员，它的取值来自java.lang.annotation.RetentionPolicy的枚举类型值。

#### @Documented

@Documented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。Documented是一个标记注解，没有成员。

#### @Inherited

@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。

注意：@Inherited annotation类型是被标注过的class的子类所继承。类并不从它所实现的接口继承annotation，方法并不从它所重载的方法继承annotation。

当@Inherited annotation类型标注的annotation的Retention是RetentionPolicy.RUNTIME，则反射API增强了这种继承性。如果我们使用java.lang.reflect去查询一个@Inherited annotation类型的annotation时，反射代码检查将展开工作：检查class和其父类，直到发现指定的annotation类型被发现，或者到达类继承结构的顶层。

### 自定义注解

使用@interface自定义注解时，自动继承了java.lang.annotation.Annotation接口，由编译程序自动完成其他细节。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。

#### 定义注解格式

public @interface 注解名 {定义体}

#### 注解参数的可支持数据类型

* 所有基本数据类型（int,float,boolean,byte,double,char,long,short)
* String类型
* Class类型
* enum类型
* Annotation类型
* 以上所有类型的数组

Annotation类型里面的参数该怎么设定:

1. 只能用public或默认(default)这两个访问权修饰.例如,String value();这里把方法设为defaul默认类型；
2. 参数成员只能用基本类型byte,short,char,int,long,float,double,boolean八种基本数据类型和 String,Enum,Class,annotations等数据类型,以及这一些类型的数组.例如,String value();这里的参数成员就为String;
3. 如果只有一个参数成员,最好把参数名称设为"value",后加小括号.例:下面的例子FruitName注解就只有一个参数成员。

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。这个约束使得处理器很难表现一个元素的存在或缺失的状态，因为每个注解的声明中，所有元素都存在，并且都具有相应的值，为了绕开这个约束，我们只能定义一些特殊的值，例如空字符串或者负数，一次表示某个元素不存在，在定义注解时，这已经成为一个习惯用法。

定义了注解，并在需要的时候给相关类，类属性加上注解信息，如果没有响应的注解信息处理流程，注解可以说是没有实用价值。如何让注解真真的发挥作用，主要就在于注解处理方法，下一步我们将学习注解信息的获取和处理！