**Annotation简介：**

1. Annotation表示注解，其本身是一个接口，定义在java.lang.annotation包中。只要 是注解都必须实现此接口。
2. 通过使用注解，程序开发人员可以在不改变原有逻辑的情况下，在源文件嵌入一些 补充的信息，并且不影响程序的运行。

**系统内建的Annotation：**

系统中主要有三个内建的Annotation，用户可以直接拿来使用：

1. @Override：方法覆写
2. @Deprecated：不赞成使用
3. @SuperessWarnings：压制警告

这三个注解都定义在java.lang包中，在使用时会自动导入。

**@Override：**

@Override在方法覆写时使用，用于保证方法覆写的正确性。且只能用在方法上。

**@Deprecated：**

@Deprecated表示不建议使用。如果一个类或方法使用了此注解声明，就表示该类或该 方法是不建议使用的。如果使用了，则在编译时出现警告，但并不影响程序的运行。

**@SuppressWarnings：**

@SuppressWarnings注解用来压制警告，例如声明泛型类对象时未指定泛型、使用了 @Deprecated注解的类或方法，这些在编译时都会产生警告。此时，就可以使用 @SuppressWarnings将其压制。

此注解是带有参数的注解，所以使用时需要传入参数，如下：

|  |  |
| --- | --- |
| 参数关键字 | 描述 |
| deprecated | （压制）使用了不赞成使用的类或方法的警告 |
| unchecked | （压制）执行了未检查的转换时警告 |
| fallthrough | （压制）swtich语句块中没有break时的警告 |
| path | （压制）在类路径、源文件路径不存在的路径警告 |
| serial | （压制）当在可序列化的类上缺少serialVersionUID定义时的警告 |
| finally | （压制）任何finally子句不能正常完成时的警告 |
| all | （压制）所有以上警告 |

压制一种警告时，@SuppressWarnings（“警告类型”）

压制多种警告时，@SuppressWarnings({“警告类型1”,…})

**自定义Annotation：**

除了系统提供内建的Annotation外，用户可以定义自己的Annotation。其定义格式为：

**[public] @interface Annotation名称{**

**数据类型 变量名称( );**

**}**

在程序中只要使用了@interface声明Annotation，那么此Annotation默认实现了 java.lang.annotation.Annotation接口。

1. 在自定义的Annotation如果设置了属性，但在使用此Annotation时必须传入相同 类型的参数。但也可以在定义属性的时候指定其默认值。如果没有设置内容的话， 则会将默认值给属性。

**[public] @interface Annotation名称{**

**数据类型 变量名称( ) default 默认值;**

**}**

1. 如果在Annotation中定义了多个属性，则在使用此Annotation时需要传入相同个数 的参数，并需要指明将哪个参数赋给哪个属性。
2. 如果要想为一个属性设置多个内容，则可以将一个属性定义成一个数组。

在定义Annotation时设置了属性，就必须在使用的时候设置内容，

1. 在自定义Annotation中也可以使用枚举来限制其属性的取值范围。

**通过反射取得Annotation：**

通过反射可以取得一个元素中的注解。包括取得元素的全部注解、取得指定的注解。

1. 取得元素的全部注解：

·通过反射取得该元素的对象

·使用取得的对象调用getAnnotations( )方法取得全部的注解，返回一个 Annotation对象数组，每一个Annotation对象表示一个注解，然后直 接输出。

1. 取得指元素的指定注解：

·通过反射取得该元素的对象

·使用取得的对象调用isAnnotationPresent(注解名.class)方法判断指定的注解 是否存在。

·如果存在，则声明一个该注解类对象，再通过元素对象调用getAnnotation(注 解名.class)方法取得其注解。

取得Annotation对象后，如果需要取得其属性的话，则通过Annotation对象调用 属性即可。例String s=注解对象.value( );

通过反射取得一个元素的Annotation时只能取得RUNTIME的方式声明的Annotation。

**元注解：** 元注解是在定义Annotation时用来声明注解的注解。元注解主义有四个@Retention、 @Target、@Document、@Inherited。

**@Rention注解：**

使用@Rention可以定义一个Annotation的保存范围。

在@Rention注解中，存在一个RententionPolicy枚举类型的属性value，无默认值，根 据传入的参数不同，其功能也不同。如下：

|  |  |
| --- | --- |
| 参数 | 描述 |
| SOURCE | 此Annotation的信息只会保留在源文件（\*.java）中 |
| CLASS | 此Annotation的信息只会保留在源文件（\*.java）和编译好的类文件中（\*.class） |
| RUNTIME | 此Annotation的信息不仅会保存在源文件（\*.java）和编译好的类文件（\*.class）中，在执行时还会被加载到JVM中 |

（对于系统内建的三个Annotation，@Override、@SuppressWarnings采用 的方式保存， @Deprecated采用（value=RUNTIME）的方式保存）

**@Target注解：**

如果一个Annotation没有明确地指出定义的位置，则在任何的位置都可以使用。而如 果希望一个自定义的Annotation只能在指定的位置上出现，则必须使用@Targer注解 进行声明。

在@Target注解中存在一个ElementsType[ ]枚举类型的变量，无默认值。使用时根据传 入的参数不同，则表示该Annotation的使用位置也不同，如下：

|  |  |
| --- | --- |
| 参数 | 描述 |
| ANNOTATION\_TYPE | 只能用在注释上 |
| CONSTRUCTOR | 只能用在构造方法上 |
| FIELD | 只能用在属性上 |
| LOCAL\_VARIABLE | 只能用在局部变量上 |
| METHOD | 只能用在方法上 |
| PACKAGE | 只能用在包的声明上 |
| TYPE | 只能用在类、接口、枚举类上 |

**@Documented注解：** 任何一个自定义的Annotation实际上都是通过@Documented进行注释的。使用 @Documented注解声明的Annotation，可以使用javadoc命令将其写成文档。

**@Inherited注解：** 使用@Inherited声明的Annotation表示该Annotation可以被子类所继承。

如果一个Annotation需要被子类继承，则在声明的时候使用@Inherited声明即可。

**Lamda表达式：**

Lamda表达式指的是应用在SAM（Single Abstract Method 含有一个抽象方法的接口）环境下的一种简化定义形式，可用于解决匿名内部类定义复杂的问题。其语法格式为：

**（参数）->方法体**

如果方法体是一条语句的话花括号最好将其忽略。

**@FunctionalInterface注解：**

1）在Lamda表达式已明确要求了是在接口上进行的一种操作，并且其接口只允许定义 有一个抽象方法。但是在一个项目中往往会定义大量的接口，为了分辨出哪些Lamda 表达式的使用接口，可以使用@FunctionlInterface注解声明，表示此接口是为Lamda 表达式服务的。这样的接口也称函数式接口。

2）从理论上来讲，如果一个接口只有一个抽象方法，写与不写@FunctionalInterface注 解是没有区别的，但从标准上来讲，最会还是写上。

3）在函数式接口中依然可以定义普通方法与静态方法。

简单来讲，@FunctionalInterface注解的作用是为了分辨哪些接口是为Lamda表达式而 定义的。

**方法引用：**

在Java中利用对象的引用传递可以实现不同的对象操作同一块堆内存空间，而在Java 中，通过函数式接口和Lamda表达式可以实现方法方法的引用。在Java中一共定义了 4中方法引用：

1. 引用静态方法：**类名称: :static 方法名称;**
2. 引用某个对象的方法：**实例化的对象: :普通方法;**
3. 引用特定类型的方法：**特定类: :普通方法;**
4. 引用构造方法：**类名称: :new;**

**引用静态方法：**

通过函数式接口利用Lamda表达式可以引用一个类中的静态方法，但需要注意的是函 数式接口中定义的抽象方法必须与引用的静态方法参数类型个数、返回值类型全部相同， 一般会使用泛型来指定。

**引用普通方法：**

通过函数式接口利用Lamda表达式可以引用一个类中的普通方法，但需要指定对象。

**引用特定类的方法：**

特定类的方法是需要特定类的支持，例如String类中CompareTo( )，因为String类实现 了Comparable接口。引用特定类的方法与引用静态方法的语法格式相同。

**引用构造方法：**

引用构造方法执行后会返回一个实例化的对象，所以在定义函数式接口时需要使用泛型 定义要产生的对象。

**内建函数式接口：**

1）在方法引用的操作中，可以发现不管如何操作，对于可能出现的函数式接口的方法 最多只会是有参数有返回值、无参数无返回值、无参数有返回值、判断真假4类。所以 为了方便开发者，从JDK1.8开始提供了java.util.function包，在这个包中提供了大量的 函数式接口，其中最主要的有四个，即功能型接口（Function）、消费型接口（Consumer）、 供给型接口（Supplier）、断言型接口（Predicate）。

**功能型接口：**

功能型接口指的是Function接口，其定义如下：  
 **@FunctionalInterface**

**public interface Function<T,R>{  
 public R apply(T t);  
 }**

主要作用：接收一个参数，并返回一个处理结果

**消费型接口：**

消费型接口指的是Consumer接口，其定义如下：

**@FunctionalInterface**

**public interface Consumer<T>{  
 public void accept(T t);  
 }**

主要作用：接收一个参数，不返回任何结果

**供给型接口：**

供给型接口指的是Supplier接口 ，其定义如下：

**@FunctionalInterface**

**public interface Supplier<T>{  
 public T get( );  
 }**

主要作用：不接受任何参数，但返回一个结果

**断言型接口：**

断言型接口指的是Predicate接口，其定义如下：

**@FunctionalInterface**

**public interface Predicate<T>{**

**public boolean test(T t);  
}**

主要作用：接收一个参数，返回一个布尔值