1. 注解

1.1 注解（注释，标注，Annotation）的作用

如果要对于注解的作用进行分类，我们可以根据它所起的作用，大致可分为三类：

编写文档：通过代码里标识的元数据生成文档。

代码分析：通过代码里标识的元数据对代码进行分析。

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。

1.2 基本内置注释

@Override 注释能实现编译时检查，你可以为你的方法添加该注释，以声明该方法是用于覆盖父类中的方法。如果该方法不是覆盖父类的方法，将会在编译时报错。例如我们为某类重写 toString() 方法却写成了 tostring() ，并且我们为该方法添加了@Override 注释，那么编译是无法通过的。

@Deprecated 的作用是对不应该在使用的方法添加注释，当编程人员使用这些方法时，将会在编译时显示提示信息，它与 javadoc 里的 @deprecated 标记有相同的功能。

@SuppressWarnings 与前两个注释有所不同，你需要添加一个参数才能正确使用，这些参数值都是已经定义好了的，我们选择性的使用就好了，参数如下：

deprecation 使用了过时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

1.3 定制注释类型

我们可以自定义注解类型，如下例：

```
public @interface NewAnnotation {
}
```

1.4 使用定制的注释类型

我们已经成功地创建好一个注释类型 NewAnnotation，现在让我们来尝试使用它。

```
public class AnnotationTest {
    @NewAnnotation
    public static void main(String[] args) {
    }
}
```

1.5 添加变量

J2SE 5.0 里，我们了解到内置注释@SuppressWarnings() 是可以使用参数的，那么自定义注释能不能定义参数个数和类型呢？答案是当然可以，但参数类型只允许为基本类型、String 、Class 、枚举类型、数组等，并且参数不能为空。我们来扩展 NewAnnotation，为之添加一个 String 类型的参数，示例代码如下：

```
public @interface NewAnnotation {
    String value();//在写注解的时候value参数中value=可以省略，其它参数名不能省
}
```

使用该注释的代码如下，该注释的使用有两种写法。


```
public class AnnotationTest {
    @NewAnnotation("Just A Test")
    public static void main(String[] args) {
        sayHello();
    }
    @NewAnnotation(value="sayHello")
    public static void sayHello(){
    }
}
```

1.6 为变量赋默认值

但还是很容易理解的，我们先定义一个枚举类型，然后将参数设置为该枚举类型，并赋予默认值。

```
public @interface Greeting {
    public enum FontColor{
        RED,GREEN,BLUE
    }
    String title();
    String content();
    FontColor fontColor() default FontColor.RED;
}
```

这样用：

```
@Greeting(title="健康",content="你最近身体好吗
",fontColor=FontColor.BLUE)
    public static void sayHello(String name){
    }
```

1.7 限定注释使用范围 （在注解定义上使用@Target，这个@Target 注解是用来限定其他注解的。）

当我们的自定义注释不断的增多也比较复杂时，就会导致有些开发人员使用错误。为此，Java 提供了一个ElementType 枚举类型来控制每个注释的使用范围，比如说某些注释只能用于普通方法，而不能用于构造函数等。下面是 Java 定义的 ElementType 枚举：

```
package java.lang.annotation;
public enum ElementType {
    TYPE,/** Class, interface (including annotation type), or enum
declaration */
    FIELD,/** Field declaration (includes enum constants) */
    METHOD,/** Method declaration */
    PARAMETER,/** Parameter declaration */
    CONSTRUCTOR,/** Constructor declaration */
    LOCAL_VARIABLE,/** Local variable declaration */
    ANNOTATION_TYPE,/** Annotation type declaration */
    PACKAGE /** Package declaration */
}
```

下面我们来修改Greeting 注释，为之添加限定范围的语句，这里我们称它为目标（Target ）使用方法也很简单，如下：

```
@Target({ElementType.METHOD,ElementType.CONSTRUCTOR})
public @interface Greeting {
}
```

正如上面代码所展示的，我们只允许 Greeting 注释标注在普通方法和构造函数上，使用在包申明、类名等时，会提示错误信息。

1.8 注释保持性策略 （注解@Retention 也是用来修饰其他注解的。）

```
package java.lang.annotation;
public enum RetentionPolicy {
    /**
```

```
     * Annotations are to be discarded by the compiler.
     */
    SOURCE,

    /**
     * Annotations are to be recorded in the class file by the compiler
     * but need not be retained by the VM at run time.  This is the default
     * behavior.
     */
    CLASS,

    /**
     * Annotations are to be recorded in the class file by the compiler
and
     * retained by the VM at run time, so they may be read reflectively.
     */
    RUNTIME
}
```

RetentionPolicy 的使用方法与 ElementType 类似，简单代码示例如下：

```
@Retention(RetentionPolicy.RUNTIME)
```

1.9 文档化功能

Java 提供的 Documented 元注释跟 Javadoc 的作用是差不多的，其实它存在的好处是开发人员可以定制 Javadoc 不支持的文档属性，并在开发中应用。它的使用跟前两个也是一样的，简单代码示例如下：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD,ElementType.CONSTRUCTOR})
public @interface Greeting {
}
```

值得大家注意的是，如果你要使用@Documented 元注释，你就得为该注释设置 RetentionPolicy.RUNTIME 保持性策略。

1.10 标注继承

它的作用是控制注释是否会影响到子类，简单代码示例如下：

```
@Inherited
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD,ElementType.CONSTRUCTOR})
public @interface Greeting {
}
```

1.11 读取注释信息

当我们想读取某个注释信息时，我们是在运行时通过反射来实现的，所以我们需要将保持性策略设置为 RUNTIME ，也就是说只有注释标记了@Retention(RetentionPolicy.RUNTIME) 的，我们才能通过反射来获得相关信息，并实现读取 AnnotationTest 类所有方法标记的注释并打印到控制台。

```
package com.bjpowernode.annotation;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;

public class AnnotationInfo {
    public static void main(String[] args) throws Exception {
        Method[] methods = Class.forName(
                "com.bjpowernode.annotation.AnnotationTest")
                .getDeclaredMethods();
        Annotation[] annotations;
```

```
        for (Method method : methods) {
            annotations = method.getAnnotations();
            for (Annotation annotation : annotations) {
                System.out.println(method.getName() + " : "
                        + annotation.annotationType().getName());
            }
        }
    }
}
```

主要掌握内容：

@Override

@Deprecated

@SuppressWarnings

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

Annotation(注解)

Annotation 对于程序运行没有影响,它的目的在于对编译器或分析工具说明程序的某些信息,您可以在包,类,方法,域成员等加上 Annotation.每一个 Annotation 对应一个实际的 Annotation 类型.

1.限定重载父类方法@Override

java.lang.Override 是 J2SE5.0 中标准的 Annotation 类型之一,它对编译器说明某个方法必须是重写父类中的方法.编译器得知这项信息后,在编译程序时如果发现被@Override 标注的方法并非重写父类中的方法,就会报告错误.例,如果在定义新类时想要重写 Object 类的 toString()方法,可能会写成这样:

```
public String ToString(){
    return "";
}
```

在编写 toString()方法时,因为输入错误或其他的疏忽,将之写成 ToString()了,编译这个类时并不会出现任何的错误,编译器不会知道您是想重写 toString()方法,只会以为是定义了一个新的 ToString()方法.可以使用 java.lang.Override 这个 Annotation 类型,在方法上加一个@Override 的 Annotation 这可以告诉编译器现在定义的这个方法,必须是重写父类中的方法.

```
@Override
public String toString(){
    return "";
}
```

java.lang.Override 是一个 Marker Annotation,简单地说就是用于标示的 Annotation，Annotation 名称本身表示了要给工具程序的信息。Annotation 类型与 Annotation 实际上是有区分的，Annotation 是 Annotation 类型的实例，例如@Override 是个 Annotation，它是 java.lang.Override 类型的一个实例，一个文件中可以有很多个@Override，但它们都是属于 java.lang.Override 类型。

2.标识方法为过时的@Deprecated

java.lang.Deprecated 也是 J2SE5.0 中标准的 Annotation 类型之一。它对编译器说明某个方法已经不建议使用。如果有开发人员试图使用或重写被@Deprecated 标示的方法，编译器必须提出警告信息。

```
@Deprecated
```



```
public String getSome(){
    return "some thing";
}
```

如果有人试图在继承这个类后重写 getSomething()方法，或是在程序中调用 getSomething()方法，则编译时会有警告出现。java.lang.Deprecated 也是一个 Marker Annotation 简单地说就是用于标示。

3. 抑制编译器警告 @SuppressWarnings

java.lang.SuppressWarnings 也是 J2SE5.0 中标准的 Annotation 类型之一，它对编译器说明某个方法中若有警告信息，则加以抑制，不用在编译完成后出现警告。例如：

```
@SuppressWarnings("unchecked")
public void add(){
    Map m = new HashMap();
    m.put("key", "value");
}
```

这样，编译器将忽略 unchecked 的警告，您也可以指定忽略多个警告：

@SuppressWarnings(value={"unchecked","deprecation"});

@SuppressWarnings 是所谓的 Single-Value Annotation，因为这样的 Annotation 只有一个成员，称为 value 成员，可在使用 Annotation 时作额外的信息指定。

4. 自定义 Annotation 类型

可以自定义 Annotation 类型，并使用这些自定义的 Annotation 类型在程序代码中使用 Annotation，这些 Annotation 将提供信息给程序代码分析工具。首先来看看如何定义 Marker Annotation，也就是 Annotation 名称本身即提供信息。对于程序分析工具来说，主要是检查是否有 Marker Annotation 的出现,并做出对应的动作。要定义一个 Annotation 所需的动作,就类似于定义一个接口，只不过使用的是@interface。

```
public @interface Debug {}
```

由于是一个 Marker Annotation，所以没有任何成员在 Annotation 定义中。编译完成后，就可以在程序代码中使用这个 Annotation。

```
public class SomeObject {
    @Debug
    public void doSome(){}
```



```
}
```

稍后可以看到如何在 Java 程序中取得 Annotation 信息(因为要使用 Java 程序取得信息，所以还要设置 meta-annotation，稍后会谈到），接着来看看如何定义一个 Single-Value Annotation，它只有一个 Value 成员。

```
public @interface UnitTest {
    String value();
}
```

实际上定义了 value()方法，编译器在编译时会自动产生一个 value 的域成员，接着在使用 UnitTest Annotation 时要指定值。如：

```
public class MathTool {
    @UnitTest("zhangsan")
    public static int sum(int a,int b){
        return a+b;
    }
}
```

@UnitTest("zhangsan")实际上是@UnitTest(value="zhangsan")的简便写法，value 也可以是数组值。如：

```
public @interface FunctionTest {
    String[] value();
}
```

在使用时，可以写成 @FunctionTest({"method1","method2"}) 这样的简便形式。或是 @FunctionTest(value={"method1","method2"})这样的详细形式. 也可以对 value 成员设置默认值，使用 default 关键词即可。

```
public @interface UnitTest {
    String value() default "NoSuchMethod";
}
```

这样如果使用@UnitTest2 时没有指定 value 值，则 value 默认就是 NoSuchMethod. 也可以为 Annotation 定义额外的成员，以提供额外的信息给分析工具，如：

```
public @interface Process {
    public enum Current {
        NONE, REQUIRE, ANALYSIS, DESIGN, SYSTEM
    };
```



```
    Current current() default Current.NONE;
    String tester();
    boolean ok();
}
```

运用:

```
public class Application {
 @Process(current=Process.Current.ANALYSIS,tester="dujubin",ok=true)
 public void doSomething(){}
}
```

5.meta-annotation

所谓meta-annotation 就是 Annotation 类型的数据，也就是 Annotation 类型的 Annotation。在定义 Annotation 类型时，为 Annotation 类型加上 Annotation 并不奇怪，这可以为处理 Annotation 类型的分析工具提供更多的信息。

5.1 告知编译器如何处理 annotation @Retention

java.lang.annotation.Retention 类型可以在您定义 Annotation 类型时，指示编译器该如何对待自定义的 Annotation 类型，编译器默认会将 Annotation 信息留在.class 文件中，但不被虚拟机读取，而仅用于编译器或工具程序运行时提供信息。在使用 Retention 类型时，需要提供 java.lang.annotation.RetentionPolicy 的枚举类型。

RetentionPolicy 的定义如下所示:

```
package java.lang.annotation;
public enum RetentionPolicy{
  SOURCE,//编译器处理完Annotation信息后就没有事了
  CLASS,//编译器将Annotation存储于class文件中，默认
  RUNTIME  //编译器将Annotation存储于class文件中，可由VM读入
}
```

RetentionPolicy 为 SOURCE 的例子是@SuppressWarnings,这个信息的作用仅在编译时期告知编译器来抑制警告，所以不必将这个信息存储在.class 文件中。

RetentionPolicy 为 RUNTIME 的时机，可以像是您使用 Java 设计一个程序代码分析工具，您必须让 VM 能读出 Annotation 信息，以便在分析程序时使用，搭配反射机制，就可以达到这个目的。

J2SE6.0 的 java.lang.reflect.AnnotatedElement 接口中定义有 4 个方法:

```
public Annotation getAnnotation(Class annotationType);
public Annotation[] getAnnotations();
public Annotation[] getDeclaredAnnotations();
public boolean isAnnotationPresent(Class annotationType);
```

Class,Constructor,Field,Method,Package 等类，都实现了 AnnotatedElement 接口，所以可以从这些类的实例上，分别取得标示于其上的 Annotation 与相关信息。由于在执行时读取 Annotation 信息，所以定义 Annotation 时必须设置 RetentionPolicy 为RUNTIME,也就是可以在 VM 中读取 Annotation 信息。

例:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface SomeAnnotation{
  String value();
  String name();
}
```

由于 RetentionPolicy 为 RUNTIME，编译器在处理 SomeAnnotation 时，会将 Annotation 及给定的相关信息编译至.class 文件中，并设置为 VM 可以读出 Annotation 信息。接下来：

```
public class SomeClass{
    @SomeAnotation(value="annotation value1",name="annotation name1")
    public void doSomething(){}
}
```

现在假设要设计一个源代码分析工具来分析所设计的类，一些分析时所需的信息已经使用 Annotation 标示于类中了，可以在执行时读取这些 Annotation 的相关信息。例:

```
public class AnalysisApp {
    public static void main(String [] args) throws NoSuchMethodException{
      Class<SomeClass> c = SomeClass.class;
      //因为SomeAnnotation标示于doSomething()方法上
      //所以要取得doSomething()方法的Method实例
      Method method = c.getMethod("doSomething");
      //如果SomeAnnotation存在
      if(method.isAnnotationPresent(SomeAnnotation.class){
```



```
        System.out.println("找到@SomeAnnotation");
        //取得SomeAnnotation
        SomeAnnotation annotation =
method.getAnnotation(SomeAnnotation.class);
        //取得value成员值
        System.out.println(annotation.value());
        //取得name成员值
        System.out.println(annotation.name());
      }else{
        System.out.println("找不到@SomeAnnotation");
      }
      //取得doSomething()方法上所有的Annotation
      Annotation[] annotations = method.getAnnotations();
      //显示Annotation名称
      for(Annotation annotation : annotations){
        System.out.println("Annotation名
称:"+annotation.annotationType().getName());
      }
    }
}
```

若 Annotation 标示于方法上，就要取得方法的 Method 代表实例，同样的，如果 Annotation 标示于类或包上,就要分别取得类的 Class 代表的实例或是包的 Package 代表的实例。之后可以使用实例上的 getAnnotation()等相关方法，以测试是否可取得 Annotation 或进行其他操作。

5.2 限定 annotation 使用对象 @Target

在定义 Annotation 类型时，使用 java.lang.annotation.Target 可以定义其适用的时机，在定义时要指定 java.lang.annotation.ElementType 的枚举值之一。

```
public enum ElementType{
    TYPE,//适用class,interface,enum
    FIELD,//适用于field
```

```
    METHOD,//适用于method
    PARAMETER,//适用method上之parameter
    CONSTRUCTOR,//适用constructor
    LOCAL_VARIABLE,//适用于区域变量
    ANNOTATION_TYPE,//适用于annotation类型
    PACKAGE,//适用于package
}
```

举例，假设定义 Annotation 类型时，要限定它只能适用于构造函数与方法成员，则:

```
@Target({ElementType.CONSTRUCTOR,ElementType.METHOD})
public @interface MethodAnnotation{}
```

将 MethodAnnotation 标示于方法之上，如:

```
public class SomeoneClass{
    @MethodAnnotation
    public void doSomething(){}
}
```

5.3 要求为 API 文件的一部分 @Documented

在制作 Java Doc 文件时，并不会默认将 Annotation 的数据加入到文件中.Annnotation 用于标示程序代码以便分析工具使用相关信息，有时 Annotation 包括了重要的信息，您也许会想要在用户制作 Java Doc 文件的同时,也一并将 Annotation 的信息加入到 API 文件中。所以在定义 Annotation 类型时，可以使用 java.lang.annotation.Documented.例:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface TwoAnnotation{}
```

使用 java.lang.annotation.Documented 为定义的 Annotation 类型加上 Annotation 时，必须同时使用 Retention 来指定编译器将信息加入.class 文件,并可以由 VM 读取,也就是要设置 RetentionPolicy 为 RUNTIME。接着可以使用这个 Annotation，并产生 Java Doc 文件,这样可以看到文件中包括了@TwoAnnotation 的信息.

5.4 子类是否可以继承父类的 annotation @Inherited

在定义 Annotation 类型并使用于程序代码上后，默认父类中的 Annotation 并不会被继承到子类中。可以在定义 Annotation 类型时加上 java.lang.annotation.Inherited 类型的 Annotation,这让您定义的 Annotation 类型在被继承后仍

可以保留至子类中。

```
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface ThreeAnnotation{
  String value();
  String name();
}
```

可以在下面的程序中使用@ThreeAnnotation:

```
public class SomeoneClass{
    @ThreeAnnotation(value = "unit",name = "debug1")
    public void doSomething(){
    }
}
```

如果有一个类继承了 SomeoneClass 类，则@ThreeAnnotation 也会被继承下来。

