JAVA反射学习之——基本学习

    反射通常用来做下在的事情：

    1. 动态加载类，获得类的信息(属性，方法，构造器)。

    2. 动态的构造对像。

    3. 动态的调用类和对像的方法。

    4. 动态的调用和处理属性（这于的处理是指修改）。

    5. 处理注解。

    首先，来看看什么是类的加载？

    程序在首次使用某个类时，如果该类还未被加载到内存中，JVM就会通过加载，连接，初始化三个步骤来对该类进行初始化，初始化的结果就是把类编译后的class文件加载到内存中，并为之创建一个java.lang.class对像，没错！是为class文件创建一个对像。关于JVM的一些特性后面再研究，今天先知道怎么用吧。

**一、获得Class对像的三种方式**

    上面说了，class文件加载到内存，并为之创建一个对像，那怎么得到这个对像呢？

**方法一：通过Object超类的getClass方法获得Class对像**

**方法二：调用类的class属性来获取该类对应的Class对像**

但上面两种方法都有它们自身的局限性，就是预先得知道了是哪个类，话说回来，你都知道了是哪个类了，为啥还要费事的得到类的class对像，然后再去操作它的属性和方法呢？

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. **public** **static** **void** main(String[] args)
2. {
3. Persion p1 = **new** Persion();
4. Persion p2 = **new** Persion();
6. //第一种方法得到类的Class文件对像
7. Class c1 = p1.getClass();
8. Class c2 = p2.getClass();
10. //第二种方法得到类的Class文件对像
11. Class c3 = Persion.**class**;
13. //通过打印结果表明,同一个类只有一个Class文件对像
14. //实际上是这样的,只要类被加载过一次，它就不会再被重新加载了
15. //所以事实上,c2和c3还是c1所指向的那个对像
16. System.out.println(p1 == p2); //false
17. System.out.println(c1 == c2); //true
18. System.out.println(c1 == c3); //true
19. }

   上面的两种方法我觉得知道就行了，并没有什么实用性吧，因为用到反射的地方，通常是为了使代码具有通用性，而不是针对某一个已知的特定类。

    方法三、使用Class类的forName(String clazzName)方法来动态加载类。

    简单的用法测试如下：

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. //第三种方法
2. Class c4 = Class.forName("反射.Persion");
4. //打印信息表明c4=c3
5. System.out.println(c4 == c3);

    值得说明的是，clazzName是一个带包名的完整的类名，不然运行会报错！

    查询JDK文档的Class类，具有如下描述：

    Class 类的实例表示正在运行的 Java 应用程序中的类和接口。枚举是一种类，注释是一种接口。每个数组属于被映射为 Class 对象的一个类，所有具有相同元素类型和维数的数组都共享该 Class 对象。基本的 Java 类型（boolean、byte、char、short、int、long、float 和 double）和关键字 void 也表示为 Class 对象。  
    Class 没有公共构造方法。Class 对象是在加载类时由 Java 虚拟机以及通过调用类加载器中的 defineClass 方法自动构造的”。

    JDK文档有两句简单的代码示例，通过以下代码简单测试

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. **public** **static** **void** printClassName(Object obj) {
3. //通过反射获得类的全名(带包名)
4. System.out.println( obj.getClass().getName());
6. //获得包名
7. System.out.println( obj.getClass().getPackage());
9. //通过反射获得类的包内名称
10. System.out.println( obj.getClass().getSimpleName());
11. }
13. **public** **static** **void** main(String[] args)
14. {
15. printClassName(**new** Persion());
16. }

    上面Class类的三个方法都非常有用的。从JDK文档摘录如下：

    1. String getName()   
        以 String 的形式返回此 Class 对象所表示的实体（类、接口、数组类、基本类型或 void）名称。   
    2. Package getPackage()     获取此类的包。   
    3. String getSimpleName()    返回源代码中给出的底层类的简称。

**二、从Class中获取信息**

    上面已经讲明了怎么来获得类的Class对像，那获得了Class对像用来干嘛呢？

**1. 通过Class对像获取构造方法并创建对像**

    构造方法分有参和无参之分，又有public和private等修饰，所以就对应有好几个方法。

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. //1.获取类的Class对像
2. Class clazz = Class.forName("反射.Persion");
3. //2.获取类的public无参构造方法
4. Constructor cons = clazz.getConstructor();
5. //3.通过类的无参构造方法返回实例
6. Object obj = cons.newInstance();

    最基本的方法和步骤就是上面所写，getConstructor是获得无参构造方法，而且，必须是public的！

    在实际当中，用的最多的，其实是下面两个：

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. //获得所有的public构造方法
2. Constructor[] cons1 = clazz.getConstructors();
3. //获得所有的构造方法(不管有参无参,public还是private等 )
4. Constructor[] cons2 = clazz.getDeclaredConstructors();

    然而，上面三个获取构造方法的声明中都是指定了变参列表的，所以我们在获取的时候，可以指定参数列表类型，从而获得得与之相匹配的构造方法来创建对像。

    典型代码如下所示：

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. Constructor cons = clazz.getConstructor(String.**class**,**int**.**class**,String.**class**);
2. Object obj = cons.newInstance("刘德华",50,"香港");

    需要注意的是，如果想使用私有的构造，那就要用getDecleardConstructor方法了。

     Constructor类有一个方法是getName()，以字符串的形式返回获取构造方法的名称，这样的话，在一些工具类的编写中，不用知道具体的是什么类而编写个有通用性的代码，但要注意的是，还要获得构造方法的参数类型！下面的一个方法就是用于此作用的，后面学习到再进行研究。

    Class<?>[] getParameterTypes()   
    按照声明顺序返回一组 Class 对象，这些对象表示此 Constructor 对象所表示构造方法的形参类型。

**2. 获得成员属性**

    属性也称字段，同样的，有getField获得一个成员属性，有getFields获得所有的public型成员属性，有getDeclaredFields获得所有的成员属性。而Field的getName方法是获得成员属性的类中的名子。

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. //获取所有的成员属性
2. Field[] fs = clazz.getDeclaredFields();
4. //打印成员属性的类中的名子
5. **for**(Field f: fs ){
6. System.out.println(f.getName());
7. }

    那怎么获取和修改特定的属性呢？

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. //1.获取类的Class对像
2. Class clazz = Class.forName("反射.Persion");
4. //2.创建类的对像
5. Object obj = clazz.newInstance();
7. //3.获取特定的字段
8. Field name = clazz.getDeclaredField("name");
9. //4在操作字段前,首先要获得访问权限
10. name.setAccessible(**true**);
11. //5.使用set方法给特定的字段赋值
12. name.set(obj, "刘德华");
13. //6.同理肯定有get方法获得某个对像的字段值
14. System.out.println(name.get(obj));

    **3. 获得成员方法并使用**

 有了上面获取成员属性和构造器的知识，再看获取成员方法并使用的方法就so easy了。一个例子学会最基本的应用：

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46859213" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. //1.获取类的Class对像
2. Class clazz = Class.forName("反射.Persion");
4. //2.创建类的对像
5. Object obj = clazz.newInstance();
7. //3.获取所有的方法,并打印出方法名
8. Method[] ms = clazz.getDeclaredMethods();
9. **for**(Method m: ms){
10. System.out.println(m.getName());
11. }
13. //4.获取无参方法function1，并调用它
14. Method function1 = clazz.getDeclaredMethod("function1");
15. function1.invoke(obj);
17. //5.获取带参方法function2
18. Method function2= clazz.getDeclaredMethod("function2", String.**class**);
19. function2.invoke(obj, "Hello World");

     好了，上面就是反射的基本用法了，仅仅是使用而已，初步了解了反射能干什么。

    仅仅提供类名，就可以获得创建类的对像，获得类的属性和方法，通过反射创建的类，可以修改它的属性，也可以调用它的成员方法。反射的功能可谓强大。

JAVA反射学习之——深入研究(反射与泛型)

    通过上节的学习，了解了反射的基本功能，下面关于反射有如下几个需要注意的地方。在学习JAVA与泛型之前，先说两个小知识点。

**一、学会使用反射编写具有通用性功能的代码**

    设计一个方法，将Object对像中的propertyName属性的值设置为value，方法声明如下：

    public void setProperty(Object obj, String propertyName, Object value);

    用反射，完成具有通用性的功能：

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46861433" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46861433" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. **class** Persion{
2. **private** String name;
3. **private** **int** age;
5. @Override
6. **public** String toString()
7. {
8. **return** name + " --- " + age;
9. }
10. }
12. **public** **class** ReflectTest2
13. {
14. **public** **static** **void** setProperty(Object obj,String propertyName,Object value)
15. **throws** Exception
16. {
18. //获得类的class文件对像
19. Class clazz = obj.getClass();
21. //获得propertyName字段
22. Field field = clazz.getDeclaredField(propertyName);
23. //设置访问权限
24. field.setAccessible(**true**);
25. //给obj对像赋值
26. field.set(obj, value);
27. }
29. **public** **static** **void** main(String[] args) **throws** Exception
30. {
31. Persion p = **new** Persion();
32. System.out.println(p);
34. setProperty(p,"name","刘德华");
35. setProperty(p,"age",50);
37. System.out.println(p);
38. }
39. }

    从上面代码来看，利用反射 不但可以完成具有通用性的功能，而且，还是访问的private型字段，可见反射的好用之处。

    通过反射来获得配置文件的内容，如果配置文件中包含有类名，那我们就可以通过类名来获得类的class文件对像，从而调用该对像中的方法，如果应用层编写的是具有通用性的功能，那底层改变只需改一下配置文件就可以了。

**二、反射带来发灵活性，但同时也损失了性能**

    下面通过一个测试来比较一下，对于同一个类，（1）利用成员方法进行普通赋值；（2）利用反射动态调用类方法

赋值；（3）取消访问权限检查后，再利用反射动态调用成员方法赋值。

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46861433" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46861433" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. **class** Persion1{
2. **private** String name;
3. **private** **int** age;
5. **public** **void** setName(String name){
6. **this**.name=name;
7. }
8. }

11. **public** **class** ReflectTest3
12. {
13. **public** **static** **void** main(String[] args) **throws** Exception
14. {
15. Persion1 p = **new** Persion1();
17. //用普通的方法进行赋值操作
18. **long** start = System.currentTimeMillis();
19. **for**(**int** i = 0; i < 100000000L; i ++){
20. p.setName("张三");
21. }
22. **long** end = System.currentTimeMillis();
23. System.out.println("普通方法赋值1亿次,执行时间: " + (end-start) + "ms");
25. //利用反射动态赋值
26. Class clazz = p.getClass();
27. Method m = clazz.getDeclaredMethod("setName", String.**class**);
28. **long** start1 = System.currentTimeMillis();
29. **for**(**int** i = 0; i < 100000000L; i ++){
30. m.invoke(p, "张三");
31. }
32. **long** end1 = System.currentTimeMillis();
33. System.out.println("反射动态赋值1亿次,执行时间: " + (end1-start1) + "ms");
35. //利用反射禁止访问检查,动态赋值测试
36. m.setAccessible(**true**);
37. **long** start2 = System.currentTimeMillis();
38. **for**(**int** i = 0; i < 100000000L; i ++){
39. m.invoke(p, "张三");
40. }
41. **long** end2 = System.currentTimeMillis();
42. System.out.println("反射取消访问检查,动态赋值1亿次,执行时间: " + (end2-start2) + "ms");
43. }
45. }

     下面是执行结果：

    普通方法赋值1亿次,执行时间: 645ms  
    反射动态赋值1亿次,执行时间: 48561ms  
    反射取消访问检查,动态赋值1亿次,执行时间: 3043ms

    从上面执行结果来看，这几种方法之间的性能差异是具大的，所以在程序中对性能有要求的地方，比如循环处，关键代码段，能不使用反射就不要使用，即使用了，也可以关掉访问权限检查以提高性能。  
  
   **三、反射与泛型**

**1. 使用反射越过泛型检查**

    首先，这个知识点的背景是这样的，如果你创建了一个集合，限制传入Integer类型，但是后面使用时，你想传入String类型，怎么办呢？先看几行代码的例子：

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46861433" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46861433" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. **public** **static** **void** main(String[] args) **throws** Exception
2. {
3. //创建一个List只能接收Integer类型
4. ArrayList<Integer> list = **new** ArrayList<Integer>();
6. list.add(100);
7. // 以下代码会报错
8. //list.add("hello");
10. //下面使用反射来越过泛型检查
11. Class clazz = list.getClass();
12. //获得List的add方法，在反射内部接收的都是Object对像
13. Method add = clazz.getDeclaredMethod("add", Object.**class**);
14. //执行add操作
15. add.invoke(list, "hello");
16. add.invoke(list, "world");
17. add.invoke(list, "java se");
19. System.out.println(list);
20. }

     对于泛型，在编写程序时都是给编译器看的，可以查看class文件的反编译结果，发现是没有泛型的，所以从理论上来说，对于上面的list对像，可以存储任意类型，在反射内部处理中，都是处理的Object类型，所以就有了以上代码。  
  
**2. 使用反射来获取泛型信息**

   java采用泛型擦除的机制来引入泛型，也就是说，泛型仅仅是给编译器javac看的，来确保数据的安全性和免去数据类型转换，但是，一旦编译完成，所有和泛型相关的东西都被擦除，这一点也可以从类编译的class文件反编译看到。

    在实际应用中，为了获得和泛型有关的信息，Java就新增了几种类型来代表不能被归一到Class类中的类型，但又和基本数据类型齐名的类型，通常使用的是如下两个：

    GenericType： 表示一种元素类型是参数化的类型或者类型变量的数组类型。

    ParameterizedType: 表示一种参数化的类型。

    为什么要引入这两种呢，实际上，在通过反射获得成员变量时，Field类有一个方法是getType，可以获得该字段的属性，但是这种属性如果是泛型就获取不到了，所以才引入了上面两种类型。

    看下面一个例子：

**[java]** [view plain](http://blog.csdn.net/fanwenjieok/article/details/46861433" \o "view plain" \t "/home/ryan/Documents\\x/_blank) [copy](http://blog.csdn.net/fanwenjieok/article/details/46861433" \o "copy" \t "/home/ryan/Documents\\x/_blank)

1. **class** student
2. {
3. **private** Map<String,Integer> score ;
5. }
7. **public** **class** ReflectTest5
8. {
9. **public** **static** **void** main(String[] args) **throws** Exception
10. {
11. Class<student> clazz = student.**class**;
13. Field f = clazz.getDeclaredField("score");
15. //通过getType方法只能获得普通类型
16. System.out.println("score的类型是：" + f.getType()); //打印Map
18. //1. 获得f的泛型类型
19. Type gType = f.getGenericType();
21. //2.如果gType是泛型类型对像
22. **if**(gType **instanceof** ParameterizedType)
23. {
24. ParameterizedType pType = (ParameterizedType)gType;
25. //获取原始类型
26. Type rType = pType.getRawType();
27. System.out.println("原始类型是: " + rType);
29. //获得泛型类型的泛型参数
30. Type[] gArgs = pType.getActualTypeArguments();
31. //打印泛型参数
32. **for**(**int** i=0; i < gArgs.length; i ++)
33. {
34. System.out.println("第"+ i +"个泛型类型是：" + gArgs[i]);
35. }
36. }
37. **else**{
38. System.out.println("获取泛型信息失败");
39. }
40. }
41. }

    Type是java.lang.reflect包下的一个接口，该接口代表所有类型的公共接口，Class是Type接口的实现类。

# **[深入理解Java：注解（Annotation）基本概念](http://www.cnblogs.com/peida/archive/2013/04/23/3036035.html)**

**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：  
　　　　第一，元数据以标签的形式存在于Java代码中。  
　　　　第二，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。  
　　　　第三，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。  
　　　　第四，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

**系统内置标准注解：**

　　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：

[IMG_256](javascript:void(0);)

public class Fruit {

public void displayName(){

System.out.println("水果的名字是：\*\*\*\*\*");

}

}

class Orange extends Fruit {

@Override

public void displayName(){

System.out.println("水果的名字是：桔子");

}

}

class Apple extends Fruit {

@Override

public void displayname(){

System.out.println("水果的名字是：苹果");

}

}

[IMG_257](javascript:void(0);)

　 Orange 类编译不会有任何问题，Apple 类在编译的时候会提示相应的错误。@Override注解只能用于方法，不能用于其他程序元素。

**@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。

　　值得注意，@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。

　　下面一段程序中使用了@Deprecated注解标示方法过期，同时在方法注释中用@deprecated tag 标示该方法已经过时，代码如下：

[IMG_258](javascript:void(0);)

class AppleService {

public void displayName(){

System.out.println("水果的名字是：苹果");

}

/\*\*

\* @deprecated 该方法已经过期，不推荐使用

\*/

@Deprecated

public void showTaste(){

System.out.println("水果的苹果的口感是：脆甜");

}

public void showTaste(int typeId){

if(typeId==1){

System.out.println("水果的苹果的口感是：酸涩");

}

else if(typeId==2){

System.out.println("水果的苹果的口感是：绵甜");

}

else{

System.out.println("水果的苹果的口感是：脆甜");

}

}

}

public class FruitRun {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

Apple apple=new Apple();

apple.displayName();

AppleService appleService=new AppleService();

appleService.showTaste();

appleService.showTaste(0);

appleService.showTaste(2);

}

}

[IMG_259](javascript:void(0);)

　　AppleService类的showTaste() 方法被@Deprecated标注为过时方法，在FruitRun类中使用的时候，编译器会给出该方法已过期，不推荐使用的提示。

**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。  
　　annotation语法允许在annotation名后跟括号，括号中是使用逗号分割的name=value对用于为annotation的成员赋值。实例如下：

[IMG_260](javascript:void(0);)

public class FruitService {

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

public static List<Fruit> getFruitList(){

List<Fruit> fruitList=new ArrayList();

return fruitList;

}

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

public static List<Fruit> getFruit(){

List<Fruit> fruitList=new ArrayList();

return fruitList;

}

@SuppressWarnings("unused")

public static void main(String[] args){

List<String> strList=new ArrayList<String>();

}

}

[IMG_261](javascript:void(0);)

　　在这个例子中SuppressWarnings annotation类型只定义了一个单一的成员，所以只有一个简单的value={...}作为name=value对。又由于成员值是一个数组，故使用大括号来声明数组值。注意：我们可以在下面的情况中缩写annotation：当annotation只有单一成员，并成员命名为"value="。这时可以省去"value="。比如将上面方法getFruit()的SuppressWarnings annotation就是缩写的。

 　 SuppressWarnings注解的常见参数值的简单说明：

　　　　1.deprecation：使用了不赞成使用的类或方法时的警告；  
　　　　2.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;   
　　　　3.fallthrough：当 Switch 程序块直接通往下一种情况而没有 Break 时的警告;  
　　　　4.path：在类路径、源文件路径等中有不存在的路径时的警告;   
　　　　5.serial：当在可序列化的类上缺少 serialVersionUID 定义时的警告;   
　　　　6.finally：任何 finally 子句不能正常完成时的警告;   
　　　　7.all：关于以上所有情况的警告。

# **[深入理解Java：注解（Annotation）自定义注解入门](http://www.cnblogs.com/peida/archive/2013/04/24/3036689.html)**

要深入学习注解，我们就必须能定义自己的注解，并使用注解，在定义自己的注解之前，我们就必须要了解Java为我们提供的元注解和相关定义注解的语法。

**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：**

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

　　使用实例：

[IMG_256](javascript:void(0);)

@Target(ElementType.TYPE)public @interface Table {

/\*\*

\* 数据表名称注解，默认值为类名称

\* @return

\*/

public String tableName() default "className";

}

@Target(ElementType.FIELD)public @interface NoDBColumn {

}

[IMG_257](javascript:void(0);)

　　注解Table 可以用于注解类、接口(包括注解类型) 或enum声明,而注解NoDBColumn仅可用于注解类的成员变量。

**@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：**

　　　　1.SOURCE:在源文件中有效（即源文件保留）  
　　　　2.CLASS:在class文件中有效（即class保留）  
　　　　3.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。具体实例如下：

[IMG_258](javascript:void(0);)

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)public @interface Column {

public String name() default "fieldName";

public String setFuncName() default "setField";

public String getFuncName() default "getField";

public boolean defaultDBValue() default false;

}

[IMG_259](javascript:void(0);)

 　　Column注解的的RetentionPolicy的属性值是RUTIME,这样注解处理器可以通过反射，获取到该注解的属性值，从而去做一些运行时的逻辑处理

**@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。

[IMG_260](javascript:void(0);)

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documentedpublic @interface Column {

public String name() default "fieldName";

public String setFuncName() default "setField";

public String getFuncName() default "getField";

public boolean defaultDBValue() default false;

}

[IMG_261](javascript:void(0);)

**@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。

　　实例代码：

[IMG_262](javascript:void(0);)

/\*\*

\*

\* @author peida

\*

\*/

@Inheritedpublic @interface Greeting {

public enum FontColor{ BULE,RED,GREEN};

String name();

FontColor fontColor() default FontColor.GREEN;

}

[IMG_263](javascript:void(0);)

**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：**

　　　　1.所有基本数据类型（int,float,boolean,byte,double,char,long,short)  
　　　　2.String类型  
　　　　3.Class类型  
　　　　4.enum类型  
　　　　5.Annotation类型  
　　　　6.以上所有类型的数组

　　Annotation类型里面的参数该怎么设定:   
　　第一,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；　 　  
　　第二,参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;　　  
　　第三,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。

　　简单的自定义注解和使用注解实例：

[IMG_264](javascript:void(0);)

package annotation;

import java.lang.annotation.Documented;import java.lang.annotation.ElementType;import java.lang.annotation.Retention;import java.lang.annotation.RetentionPolicy;import java.lang.annotation.Target;

/\*\*

\* 水果名称注解

\* @author peida

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documentedpublic @interface FruitName {

String value() default "";

}

[IMG_265](javascript:void(0);)

[IMG_266](javascript:void(0);)

package annotation;

import java.lang.annotation.Documented;import java.lang.annotation.ElementType;import java.lang.annotation.Retention;import java.lang.annotation.RetentionPolicy;import java.lang.annotation.Target;

/\*\*

\* 水果颜色注解

\* @author peida

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documentedpublic @interface FruitColor {

/\*\*

\* 颜色枚举

\* @author peida

\*

\*/

public enum Color{ BULE,RED,GREEN};

/\*\*

\* 颜色属性

\* @return

\*/

Color fruitColor() default Color.GREEN;

}

[IMG_267](javascript:void(0);)

[IMG_268](javascript:void(0);)

package annotation;

import annotation.FruitColor.Color;

public class Apple {

@FruitName("Apple")

private String appleName;

@FruitColor(fruitColor=Color.RED)

private String appleColor;

public void setAppleColor(String appleColor) {

this.appleColor = appleColor;

}

public String getAppleColor() {

return appleColor;

}

public void setAppleName(String appleName) {

this.appleName = appleName;

}

public String getAppleName() {

return appleName;

}

public void displayName(){

System.out.println("水果的名字是：苹果");

}

}

[IMG_269](javascript:void(0);)

**注解元素的默认值：**

　　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。例如：

package annotation;

import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

/\*\*

\* 水果供应者注解

\* @author peida

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface FruitProvider {

/\*\*

\* 供应商编号

\* @return

\*/

public int id() default -1;

/\*\*

\* 供应商名称

\* @return

\*/

public String name() default "";

/\*\*

\* 供应商地址

\* @return

\*/

public String address() default "";

}

　　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！

# **[深入理解Java：注解（Annotation）--注解处理器](http://www.cnblogs.com/peida/archive/2013/04/26/3038503.html)**

　　如果没有用来读取注解的方法和工作，那么注解也就不会比注释更有用处了。使用注解的过程中，很重要的一部分就是创建于使用注解处理器。Java SE5扩展了反射机制的API，以帮助程序员快速的构造自定义注解处理器。

**注解处理器类库(java.lang.reflect.AnnotatedElement)：**

　　Java使用Annotation接口来代表程序元素前面的注解，该接口是所有Annotation类型的父接口。除此之外，Java在java.lang.reflect 包下新增了AnnotatedElement接口，该接口代表程序中可以接受注解的程序元素，该接口主要有如下几个实现类：

　　Class：类定义  
　　Constructor：构造器定义  
　　Field：累的成员变量定义  
　　Method：类的方法定义  
　　Package：类的包定义

　　java.lang.reflect 包下主要包含一些实现反射功能的工具类，实际上，java.lang.reflect 包所有提供的反射API扩充了读取运行时Annotation信息的能力。当一个Annotation类型被定义为运行时的Annotation后，该注解才能是运行时可见，当class文件被装载时被保存在class文件中的Annotation才会被虚拟机读取。  
　　AnnotatedElement 接口是所有程序元素（Class、Method和Constructor）的父接口，所以程序通过反射获取了某个类的AnnotatedElement对象之后，程序就可以调用该对象的如下四个个方法来访问Annotation信息：

　　方法1：<T extends Annotation> T getAnnotation(Class<T> annotationClass): 返回改程序元素上存在的、指定类型的注解，如果该类型注解不存在，则返回null。  
　　方法2：Annotation[] getAnnotations():返回该程序元素上存在的所有注解。  
　　方法3：boolean is AnnotationPresent(Class<?extends Annotation> annotationClass):判断该程序元素上是否包含指定类型的注解，存在则返回true，否则返回false.  
　　方法4：Annotation[] getDeclaredAnnotations()：返回直接存在于此元素上的所有注释。与此接口中的其他方法不同，该方法将忽略继承的注释。（如果没有注释直接存在于此元素上，则返回长度为零的一个数组。）该方法的调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响。

　　一个简单的注解处理器：

[IMG_256](javascript:void(0);)

/\*\*\*\*\*\*\*\*\*\*\*注解声明\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* 水果名称注解

\* @author peida

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documentedpublic @interface FruitName {

String value() default "";

}

/\*\*

\* 水果颜色注解

\* @author peida

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documentedpublic @interface FruitColor {

/\*\*

\* 颜色枚举

\* @author peida

\*

\*/

public enum Color{ BULE,RED,GREEN};

/\*\*

\* 颜色属性

\* @return

\*/

Color fruitColor() default Color.GREEN;

}

/\*\*

\* 水果供应者注解

\* @author peida

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documentedpublic @interface FruitProvider {

/\*\*

\* 供应商编号

\* @return

\*/

public int id() default -1;

/\*\*

\* 供应商名称

\* @return

\*/

public String name() default "";

/\*\*

\* 供应商地址

\* @return

\*/

public String address() default "";

}

/\*\*\*\*\*\*\*\*\*\*\*注解使用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public class Apple {

@FruitName("Apple")

private String appleName;

@FruitColor(fruitColor=Color.RED)

private String appleColor;

@FruitProvider(id=1,name="陕西红富士集团",address="陕西省西安市延安路89号红富士大厦")

private String appleProvider;

public void setAppleColor(String appleColor) {

this.appleColor = appleColor;

}

public String getAppleColor() {

return appleColor;

}

public void setAppleName(String appleName) {

this.appleName = appleName;

}

public String getAppleName() {

return appleName;

}

public void setAppleProvider(String appleProvider) {

this.appleProvider = appleProvider;

}

public String getAppleProvider() {

return appleProvider;

}

public void displayName(){

System.out.println("水果的名字是：苹果");

}

}

/\*\*\*\*\*\*\*\*\*\*\*注解处理器\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public class FruitInfoUtil {

public static void getFruitInfo(Class<?> clazz){

String strFruitName=" 水果名称：";

String strFruitColor=" 水果颜色：";

String strFruitProvicer="供应商信息：";

Field[] fields = clazz.getDeclaredFields();

for(Field field :fields){

if(field.isAnnotationPresent(FruitName.class)){

FruitName fruitName = (FruitName) field.getAnnotation(FruitName.class);

strFruitName=strFruitName+fruitName.value();

System.out.println(strFruitName);

}

else if(field.isAnnotationPresent(FruitColor.class)){

FruitColor fruitColor= (FruitColor) field.getAnnotation(FruitColor.class);

strFruitColor=strFruitColor+fruitColor.fruitColor().toString();

System.out.println(strFruitColor);

}

else if(field.isAnnotationPresent(FruitProvider.class)){

FruitProvider fruitProvider= (FruitProvider) field.getAnnotation(FruitProvider.class);

strFruitProvicer=" 供应商编号："+fruitProvider.id()+" 供应商名称："+fruitProvider.name()+" 供应商地址："+fruitProvider.address();

System.out.println(strFruitProvicer);

}

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*输出结果\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/public class FruitRun {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

FruitInfoUtil.getFruitInfo(Apple.class);

}

}

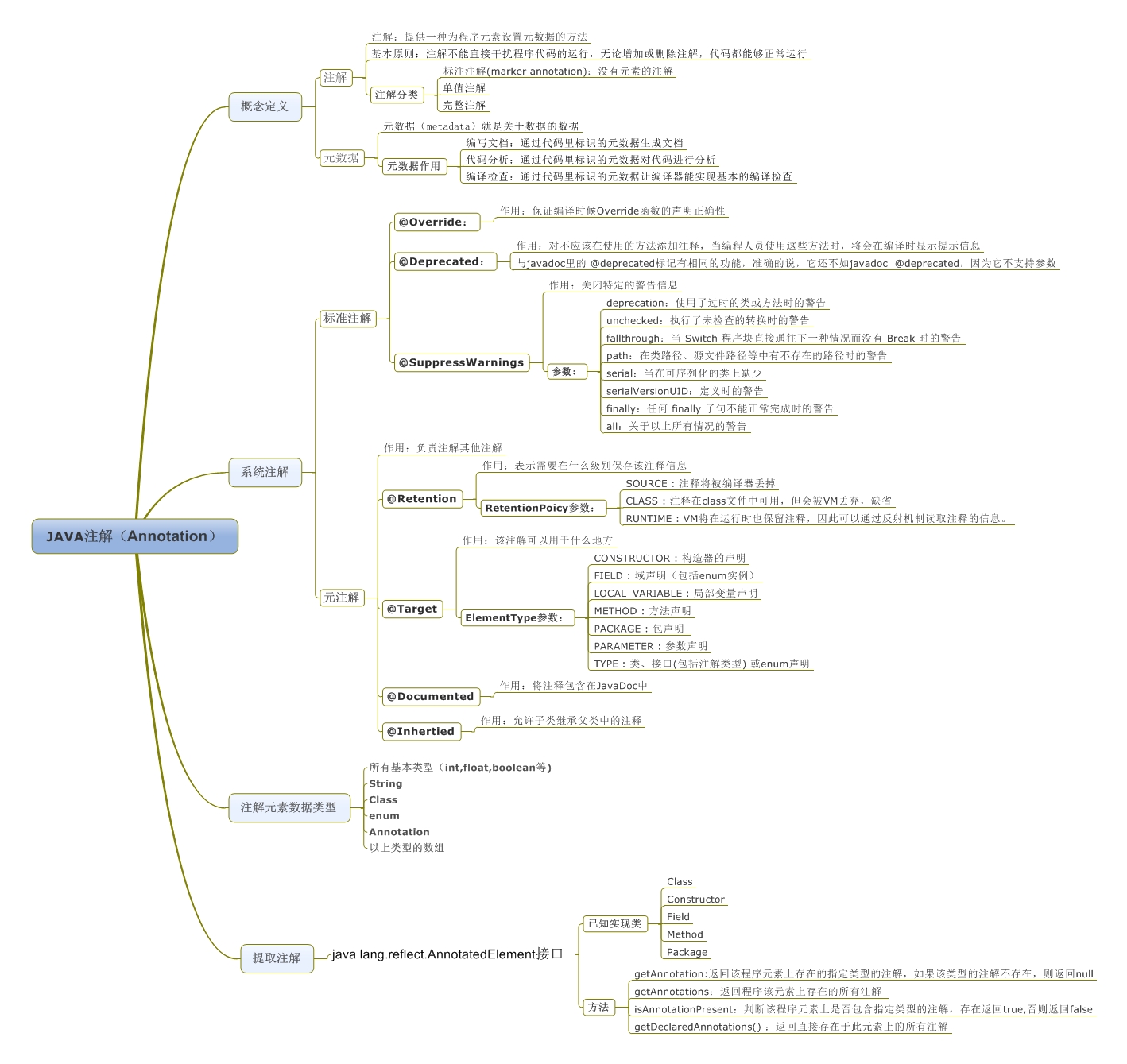
====================================

水果名称：Apple

水果颜色：RED

供应商编号：1 供应商名称：陕西红富士集团 供应商地址：陕西省西安市延安路89号红富士大厦

Java注解的基础知识点



### [Java Annotation注解继承方式说明](http://zxmsdyz.iteye.com/blog/1974249)

有关Annotation的继承说明：

1、JDK文档中的说明是：只有在类上应用的Annotation才能被继承，而实际应用时的结果是：除了类上应用的Annotation能被继承外，没有被重写的方法的Annotation也能被继承。

2、要注意的是：当方法被重写后，Annotation将不会被继承。

3、要使得Annotation 被继承，需要在Annotation中加标识@Inherited，并且如果要被反射应用的话，就需要还有个@Retention(RetentionPolicy.RUNTIME) 标识。

4、Annotation的继承不能应用在接口上。

特别说明下spring事务注解@Transactional：

@Transactional 注解应该只被应用到 public 可见度的方法上。 如果你在 protected、private 或者 package-visible 的方法上使用 @Transactional 注解，它也不会报错， 但是这个被注解的方法将不会展示已配置的事务设置。  
  
@Transactional 注解可以被应用于接口定义和接口方法、类定义和类的 public 方法上。然而，请注意仅仅 @Transactional 注解的出现不足于开启事务行为，它仅仅 是一种元数据，能够被可以识别 @Transactional 注解和上述的配置适当的具有事务行为的beans所使用。上面的例子中，其实正是 <tx:annotation-driven/>元素的出现 开启 了事务行为。  
  
Spring团队的建议是你在具体的类（或类的方法）上使用 @Transactional 注解，而不要使用在类所要实现的任何接口上。你当然可以在接口上使用 @Transactional 注解，但是这将只能当你设置了基于接口的代理时它才生效。因为注解是 不能继承 的，这就意味着如果你正在使用基于类的代理时，那么事务的设置将不能被基于类的代理所识别，而且对象也将不会被事务代理所包装（将被确认为严重的）。因此，请接受Spring团队的建议并且在具体的类上使用 @Transactional 注解。   
  
注意  
当使用 @Transactional 风格的进行声明式事务定义时，你可以通过 <tx:annotation-driven/> 元素的 "proxy-target-class" 属性值来控制是基于接口的还是基于类的代理被创建。如果 "proxy-target-class" 属值被设置为 "true"，那么基于类的代理将起作用（这时需要CGLIB库cglib.jar在CLASSPATH中）。如果 "proxy-target-class" 属值被设置为 "false" 或者这个属性被省略，那么标准的JDK基于接口的代理将起作用。

### [java.lang.reflect.Array的应用](http://yahaitt.iteye.com/blog/144779)

java.lang.Array 类提供了动态创建和访问数组元素的各种静态方法。例程  
ArrayTest 类的main()方法创建了一个长度为10 的字符串数组，接着把索引位置为5 的元素设为“hello”，然后再读取索引位置为5 的元素的值

一、作为一维数组示例代码：

这里第21行，也可以这样写：Array.setInt(array\_2,10,37);

java 代码

1. **package** com.test.reflect;
3. **import** java.lang.reflect.Array;
5. **public** **class** ArrayTest {
7. **public** **static** **void** main(String[] args) **throws** Exception {
9. Class clazz = Class.forName("java.lang.String");
11. //创建一个长度为10的字符串数组，在Java中数组也可以作为Object对象
12. Object array = Array.newInstance(clazz, 10);
14. //把字符串数组对象的索引位置为5的元素设置为"hello"
15. Array.set(array, 5, "hello");
17. //获得字符串数组对象的索引位置为5的元素的值
18. String str = (String)Array.get(array, 5);
19. System.out.println(str);//hello
20. }
22. }

二、作为多维数组对象示例代码

java 代码

1. **package** com.test.reflect;
3. **import** java.lang.reflect.Array;
5. **public** **class** ArrayTest2 {
7. **public** **static** **void** main(String[] args) {
9. //dims的长度就表示了数组的维数，如这里的是三维
10. **int**[] dims = **new** **int**[] {5,10,15};
11. //array其实就是个三维数组对象了
12. Object array = Array.newInstance(Integer.TYPE, dims);
14. //找到第一维下标为3的元素，此元素是个二维数组对象
15. Object array\_1 = Array.get(array, 3);
17. //找到第二维下标为5的元素，此元素是个一维数组对象
18. Object array\_2 = Array.get(array\_1, 5);
20. //找到第三维下标为10的元素，此元素是个int类型的数据，并给这个数据赋值为37
21. Array.set(array\_2, 10, 37);
23. **int**[][][] value = (**int**[][][])array;
24. System.out.println(value[3][5][10]);//37
26. }
28. }

上面的代码第21行也可以这样写：Array.setInt(array\_2,10,37);

### [@Documented Annotation的使用](http://yahaitt.iteye.com/blog/144587)

@Documented Annotation的使用：

@Documented Annotation的作用是在生成javadoc文档的时候将该Annotation也写入到文档中。

java 代码

1. **package** com.test;
3. **import** java.lang.annotation.Documented;
5. @Documented
6. **public** **@interface** DocumentTest {
7. String hello();
8. }

java 代码

1. **package** com.test;

4. **public** **class** DocumentClass {
5. /\*\*
6. \* this is method of doSomething
7. \*/
8. @DocumentTest(hello = "yahaitt")
9. **public** **void** doSomething()
10. {
11. System.out.println("do something");
12. }
14. /\*\*
15. \* this is method of say
16. \*/
17. **public** **void** say()
18. {
19. System.out.println("say");
20. }
21. }

生成的doc文件中如下：

### doSomething

@DocumentTest(hello="yahaitt")

public void ****doSomething****()

this is method of doSomething

### [有关Annotation的继承](http://yahaitt.iteye.com/blog/144565)

有关Annotation的继承说明：

1、JDK文档中的说明是：只有在类上应用的Annotation才能被继承，而实际应用时的结果是：除了类上应用的Annotation能被继承外，没有被重写的方法的Annotation也能被继承。

2、要注意的是：当方法被重写后，Annotation将不会被继承。

3、要使得Annotation 被继承，需要在Annotation中加标识@Inherited，并且如果要被反射应用的话，就需要还有个@Retention(RetentionPolicy.RUNTIME) 标识

4、Annotation的继承不能应用在接口上

代码一、实现类上的继承

java 代码

1. **package** com.test;
3. **import** java.lang.annotation.Inherited;
4. **import** java.lang.annotation.Retention;
5. **import** java.lang.annotation.RetentionPolicy;
7. @Inherited
8. @Retention(RetentionPolicy.RUNTIME)
9. **public** **@interface** InheritedTest {
10. String hello();
11. }

java 代码

1. **package** com.test;
3. @InheritedTest(hello = "yahaitt")
4. **public** **class** InheritedParent {
6. }

java 代码

1. **package** com.test;
3. **public** **class** InheritedChild **extends** InheritedParent {
5. }

java 代码

1. **package** com.test;
3. **public** **class** InheritedClassTest {
5. **public** **static** **void** main(String[] args) **throws** Exception
6. {
7. Class<inheritedchild></inheritedchild> c = InheritedChild.**class**;
8. **if**(c.isAnnotationPresent(InheritedTest.**class**))
9. {
10. InheritedTest it = c.getAnnotation(InheritedTest.**class**);
11. System.out.println(it.hello());//yahaitt
12. }
13. }
14. }

代码二、实现方法上的继承

java 代码

1. **package** com.test;
3. **import** java.lang.annotation.Inherited;
4. **import** java.lang.annotation.Retention;
5. **import** java.lang.annotation.RetentionPolicy;
7. @Inherited
8. @Retention(RetentionPolicy.RUNTIME)
9. **public** **@interface** InheritedTest {
10. String hello();
11. }

java 代码

1. **package** com.test;

4. **public** **class** InheritedParent {
6. @InheritedTest(hello = "yahaitt")
7. **public** **void** doSomething()
8. {
9. System.out.println("parent do something");
10. }
12. }

java 代码

1. **package** com.test;
3. **public** **class** InheritedChild **extends** InheritedParent {

6. }

java 代码

1. **package** com.test;
3. **import** java.lang.reflect.Method;
5. **public** **class** InheritedClassTest {
7. **public** **static** **void** main(String[] args) **throws** Exception
8. {
9. Class<inheritedchild></inheritedchild> c = InheritedChild.**class**;
10. Method method = c.getMethod("doSomething", **new** Class[]{});
11. **if**(method.isAnnotationPresent(InheritedTest.**class**))
12. {
13. InheritedTest it = method.getAnnotation(InheritedTest.**class**);
14. System.out.println(it.hello());//yahaitt
15. }
16. }
17. }

### [@Target Annotation的使用](http://yahaitt.iteye.com/blog/144562)

@Target Annotation的使用

@Target Annotation表示的是这个Annotation可以应用在哪里，是对Annotation应用的限制，如果没有这个标识的话，Annotation可以用在任何地方，比如类上，方法上，变量、属性上等，可以从java.lang.Enum.ElementType这个枚举里选择。

1、只能用在方法上的限制的应用

java 代码

1. **package** com.test;
3. **import** java.lang.annotation.ElementType;
4. **import** java.lang.annotation.Target;
6. @Target(ElementType.METHOD)
7. **public** **@interface** TargetTest {
8. String hello();
9. }

java 代码

1. **package** com.test;

4. **public** **class** TargetClass {
5. @TargetTest(hello = "abc")
6. **public** **void** doSomething()
7. {
8. System.out.println("do something");
9. }
10. }

2、只能用于类上的限制应用

java 代码

1. **package** com.test;
3. **import** java.lang.annotation.ElementType;
4. **import** java.lang.annotation.Target;
6. @Target(ElementType.TYPE)
7. **public** **@interface** TargetTest {
8. String hello();
9. }

java 代码

1. **package** com.test;

4. @TargetTest(hello = "abc")
5. **public** **class** TargetClass {
6. **public** **void** doSomething()
7. {
8. System.out.println("do something");
9. }
10. }

### [告知编译程序如何处理@Retention](http://yahaitt.iteye.com/blog/144455)

告知编译程序如何处理@Retention：

java.lang.annotation.Retention型态可以在您定义Annotation型态时，指示编译程序该如何对待您的自定义Annotation型态。

预定义上编译程序会将Annotation信息留在.class文档中，但不被虚拟机读取，而仅用于编译程序或工具程序运行时提供信息。

java.lang.annotation.RetentionPolicy 有三个枚举类型：CLASS、RUNTIME、SOURCE

只有当Annotation被指示成RUNTIME时，在运行时通过反射机制才能被JVM读取，否则，JVM是读取不到这个Annotation的。

java 代码

1. **package** com.test;
3. **import** java.lang.annotation.Retention;
4. **import** java.lang.annotation.RetentionPolicy;
6. @Retention(RetentionPolicy.RUNTIME)
7. **public** **@interface** RetentionTest {
8. String hello() **default** "hello";
9. String world();
10. }

java 代码

1. **package** com.test;
3. **public** **class** MyTest {
4. @RetentionTest(hello = "beijing", world = "shanghai")
5. @Deprecated
6. @SuppressWarnings("unchecked")
7. **public** **void** output()
8. {
9. System.out.println("output");
10. }
11. }

java 代码

1. **package** com.test;
3. **import** java.lang.annotation.Annotation;
4. **import** java.lang.reflect.Method;
6. **public** **class** ReflectRetentionTest {
8. **public** **static** **void** main(String[] args) **throws** Exception{
9. MyTest mt = **new** MyTest();
10. Class<mytest></mytest> c = MyTest.**class**;
11. Method method = c.getMethod("output",**new** Class[]{});
12. **if**(method.isAnnotationPresent(RetentionTest.**class**))
13. {
14. method.invoke(mt, **new** Object[]{});//output
16. RetentionTest  retentionTest = method.getAnnotation(RetentionTest.**class**);
18. System.out.println(retentionTest.hello());//beijing
19. System.out.println(retentionTest.world());//shanghai
20. }
22. Annotation[] annotations = method.getAnnotations();
23. **for**(Annotation annotation: annotations)
24. {
25. System.out.println(annotation.annotationType().getName());
26. }
27. //for循环里输出的结果是com.test.RetentionTest以及java.lang.Deprecated，而没有出来java.lang.SuppressWarnings
28. //因为java.lang.SuppressWarnings的Retention是被设置成RetentionPolicy.SOURCE类型的，所以在运行时是不会被虚拟机读取的。
29. }
30. }