# 类及类加载器相关

## 类加载的过程

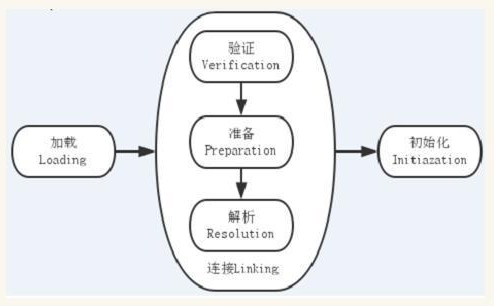
当程序主动使用某个类时，该类还未被加载到内存，系统会通过以下三个步骤来加载类。

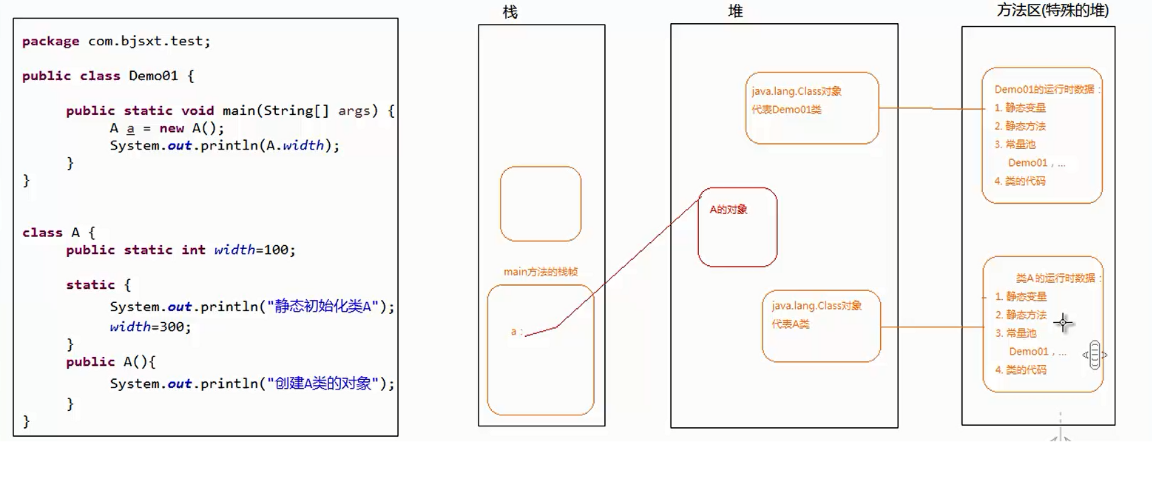
①类的加载：将.class文件读入到内存，并为之创建一个Class对象。由类加载器完成。

②类的连接：将类的二进制文件合并到JRE中。

③类的初始化：由JVM进行类的初始化。

Class对象的由来是将class文件读入内存，并为之创建一个Class对象。





## 类加载器的分类

**①**系统类加载器

ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();

②扩展类加载器

ClassLoader parent = ClassLoader.getSystemClassLoader().getParent();

③引导类加载器：负责加载核心类库，无法直接获取

ClassLoader parent = ClassLoader.getSystemClassLoader().getParent().getParent();

## 类加载的时机

①源文件程序只有在需要使用该程序代码时，才会被加载。

②类的代码只有在初次使用时才会被加载。

    <1>通常发生于创建第一个对象的时候

    <2>访问static域和static对象时也会被加载。

    <3>创建对象引用不加载类。

<4>可以归结为：访问static成员时，类会被初始化，因为构造器也是隐式的静态方法。因此更准确的讲，类在任何static成员访问时加载。

## 类加载过程分为：类的主动引用和类的被动引用

类的主动引用（一定会发生类的初始化）

--new一个类的对象

--调用类的静态成员（除了final常量）和静态方法

--使用java.lang.reflect包的方法对类进行反射调用

--当初始化一个类，如果其父类没有被初始化，则先初始化他的父类

--当要执行某个程序时，一定先启动main方法所在的类

类的被动引用（不会发生类的初始化）

--当访问一个静态变量时，只有真正生命这个静态变量的类才会被初始化（通过子类引用父类的静态变量，不会导致子类初始化）

--通过数组定义类应用，不会触发此类的初始化  A[] a = new A[10];

--引用常量(final类型)不会触发此类的初始化（常量在编译阶段就存入调用类的常量池中了）

## 获取Class对象的四种方式

**①**

Class clazz = Person.class;

②

Person person = new Person();

Class<? extends Person> clazz = person.getClass();

③

Class<?> clazz = Class.forName("com.nucsoft.refletciton.Person");

④

String className = "com.nucsoft.jdbc.Person";

ClassLoader classLoader = Person.class.getClassLoader();

Class<?> clazz = classLoader.loadClass(className);

# 反射

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

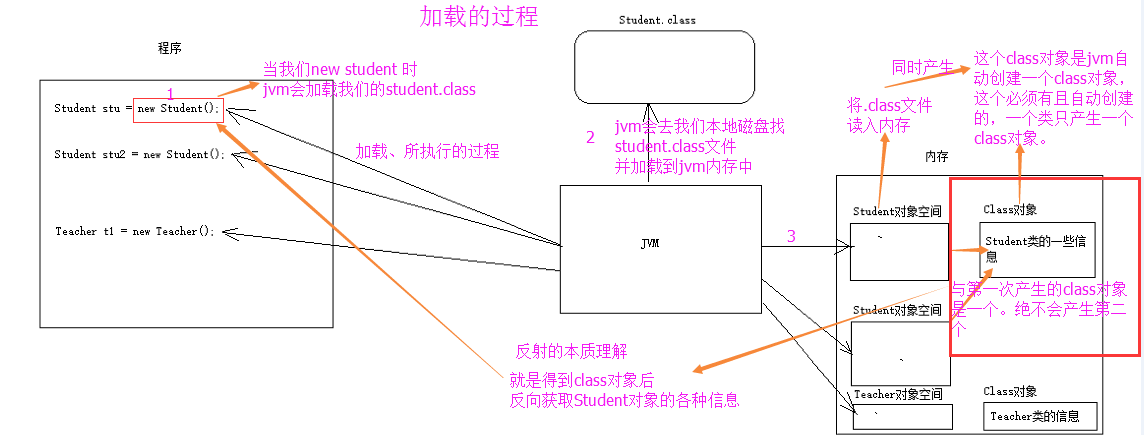
要想解剖一个类,必须先要获取到该类的字节码文件对象。而解剖使用的就是Class类中的方法.所以先要获取到每一个字节码文件对应的Class类型的对象.

**（使用的前提条件：必须先得到代表的字节码的Class，Class类用于表示.class文件（字节码））**

## 为什么用反射

需要在运行时才得知并使用编译时完全未知的类，创建其对象，改变其属性，调用其方法。

## 什么是反射



允许程序在运行时，借助Reflection API取得任何类的内部信息，并直接操纵其属性和方法。

## 通过反射创建实例

①默认是通过无参构造器来创建的

String className = "com.nucsoft.jdbc.Person";

Class<?> clazz = Class.forName(className);

Object newInstance = clazz.newInstance();

②也可以通过有参构造器来创建

Class<?> personClass = Class.forName("com.nucsoft.refletciton.Person");

Constructor<?> constructor = personClass.getConstructor(int.class, String.class);

Object obj = constructor.newInstance(2, "wangwu");

## 通过反射获取属性，改变属性

①getField()方法只能获取public修饰的属性

②getDeclaredField()方法能获取任意修饰符修饰的属性对象(Field对象)。但是在修改私有属性的时候，需要更改访问权限。

Class<?> personClass = Class.forName("com.nucsoft.refletciton.Person");

Field personName = personClass.getDeclaredField("personName");

personName.setAccessible(true);

personName.set(person, "zhangsan");

## 通过反射获取方法，调用方法

getMethod() 方法只能获取public修饰的方法

②getDeclaredMethod() 方法可以任意修饰符修饰的方法对象(Method对象)。但是在调用非公有方法时，需要设置访问权限。且通过method.invoke(obj, args)来调用、

Method sayMethod = personClass.getDeclaredMethod("say");

sayMethod.setAccessible(true);

sayMethod.invoke(person);

## 获取父类泛型

**Class clazz = Person.class;**

**Type type = clazz.getGenericSuperClass();**

**ParameterizedType types = (ParameterizedType)type;**

**for(Type t : types) {**

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

**}**

# 注解

## 什么是注解（Annotation）

**符合一定格式的语法 @XXX**

**注解：给jvm看的，给机器看的**

**注释：给程序员看的**

**主流应用：代替配置文件 例如servlet 3.0时，创建servlet时的方法**

**注解开发与配置文件的优缺点：优点 –》开发效率高，成本低，缺点：不利于后期维护 耦合性大 ----》MVC**

**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：

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");

}

}

　 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：

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);

}

}

　　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：

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>();

}

}

　　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。

## [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：

@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 {

}

　　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：

@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;

}

 　　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。

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface Column {

public String name() default "fieldName";

public String setFuncName() default "setField";

public String getFuncName() default "getField";

public boolean defaultDBValue() default false;

}

#### @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：

/\*\*

\*

\* @author peida

\*

\*/

@Inherited

public @interface Greeting {

public enum FontColor{ BULE,RED,GREEN};

String name();

FontColor fontColor() default FontColor.GREEN;

}

### 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：

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)

@Documented

public @interface FruitName {

String value() default "";

}

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)

@Documented

public @interface FruitColor {

/\*\*

\* 颜色枚举

\* @author peida

\*

\*/

public enum Color{ BULE,RED,GREEN};

/\*\*

\* 颜色属性

\* @return

\*/

Color fruitColor() default Color.GREEN;

}

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");

}

}

### 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：

1 package annotation;

2

3 import java.lang.annotation.Documented;

4 import java.lang.annotation.ElementType;

5 import java.lang.annotation.Retention;

6 import java.lang.annotation.RetentionPolicy;

7 import java.lang.annotation.Target;

8

9 /\*\*

10 \* 水果供应者注解

11 \* @author peida

12 \*

13 \*/

14 @Target(ElementType.FIELD)

15 @Retention(RetentionPolicy.RUNTIME)

16 @Documented

17 public @interface FruitProvider {

18 /\*\*

19 \* 供应商编号

20 \* @return

21 \*/

22 public int id() default -1;

23

24 /\*\*

25 \* 供应商名称

26 \* @return

27 \*/

28 public String name() default "";

29

30 /\*\*

31 \* 供应商地址

32 \* @return

33 \*/

34 public String address() default "";

35 }

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：

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

/\*\*

\* 水果名称注解

\* @author peida

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface FruitName {

String value() default "";

}

/\*\*

\* 水果颜色注解

\* @author peida

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @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)

@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 "";

}

/\*\*\*\*\*\*\*\*\*\*\*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

# 代理

Proxy，也就是“代理”了。意思就是，你不用去做，别人代替你去处理。比如说：赚钱方面，我就是我老婆的 Proxy；带小孩方面，我老婆就是我的 Proxy；家务事方面，没有 Proxy。

它在程序开发中起到了非常重要的作用，比如传说中的 AOP（面向切面编程），就是针对代理的一种应用。此外，在设计模式中，还有一个“代理模式”。在公司里要上外网，要在浏览器里设置一个 HTTP 代理。

凡是都要由浅入深，学习也不例外。先来一个 Hello World 吧：

public interface Hello {

void say(String name);

}

这是一个 Hello 接口，不用解释了，大家都懂的。赶紧上实现类吧：

public class HelloImpl implements Hello {

@Override

public void say(String name) {

System.out.println("Hello! " + name);

}

}

### 静态代理

这简直就是 So easy 嘛！但是有个问题：如果要在 println() 方法前面和后面分别需要处理一些逻辑，怎么做呢？把这些逻辑写死在 say() 方法里面吗？肯定不够优雅啦，菜鸟一般这样干，作为一名资深的程序员，我坚决不能这样做！

我要用代理！写一个 HelloProxy 类，让它去调用 HelloImpl 的 say() 方法，在调用的前后分别进行逻辑处理不就行了吗？赶紧搞一个吧：

public class HelloProxy implements Hello {

private HelloImpl helloImpl;

public HelloProxy() {

helloImpl = new HelloImpl();

}

@Override

public void say(String name) {

before();

helloImpl.say(name);

after();

}

private void before() {

System.out.println("Before");

}

private void after() {

System.out.println("After");

}

}

我将 HelloProxy 类实现了 Hello 接口（和 HelloImpl 实现相同的接口），并且在构造方法中 new 出一个 HelloImpl 类的实例。这样一来，我就可以在 HelloProxy 的 say() 方法里面去调用 HelloImpl 的 say() 方法了。更重要的是，我还可以在调用的前后分别加上 before() 与 after() 方法，在这两个方法里去实现那些前后逻辑。

用一个 main 方法来测试一下吧：

public static void main(String[] args) {

Hello helloProxy = new HelloProxy();

helloProxy.say("Jack");

}

运行后，打印出：

Before  
Hello! Jack  
After

轻而易举，我就写出了这么优雅的代码（暗自小嗨了一把）。

### Jdk动态代理

不久后，我在一本设计模式的书上看到，原来我写的这个 HelloProxy 就是所谓的“代理模式”啊！我只能说，自己和 GoF（四人帮）的距离有接近了一点。

于是我疯狂的使用“代理模式”，项目中到处都有 XxxProxy 的声影。直到有一天，架构师看到了我的代码，他惊呆了！他对我说：“你怎么这么喜欢用静态代理呢？你就不会用动态代理吗？给我全都重构了！”。

我表面上点了点头，说：“好的！”。其实我根本都不知道什么是“静态代理”，什么又是“动态代理”。我继续翻开我那本垫桌脚的设计模式，深入得研究了一 番，最后我才明白，原来我一直用的都是“静态代理”啊，怪不得架构师说我到处都是 XxxProxy 类了。我好恨我自己！一定要将这些垃圾 Proxy 都重构为“动态代理”。

于是我就是用 JDK 给我们提供的动态代理方案，写了一个 DynamicProxy：

public class DynamicProxy implements InvocationHandler {

private Object target;

public DynamicProxy(Object target) {

this.target = target;

}

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

before();

Object result = method.invoke(target, args);

after();

return result;

}

...

}

在 DynamicProxy 类中，我定义了一个 Object 类型的 target 变量，它就是被代理的目标对象，通过构造函数来初始化（现在流行叫“注入”了，我觉得叫“射入”也不错哦！构造函数初始化叫“正着射”，所以 reflect 方式就叫“反着射”，简称“反射”）。

言归正传，DynamicProxy 实现了 InvocationHandler 接口，那么必须实现该接口的 invoke 方法，参数不做解释，望文生义吧，是 JRE 给我们“射”进来的。在该方法中，直接通过反射去 invoke method，在调用前后分别处理 before 与 after，最后将 result 返回。

写一个 main() 方法看看实际怎么用吧：

public static void main(String[] args) {

Hello hello = new HelloImpl();

DynamicProxy dynamicProxy = new DynamicProxy(hello);

Hello helloProxy = (Hello) Proxy.newProxyInstance(

hello.getClass().getClassLoader(),

hello.getClass().getInterfaces(),

dynamicProxy

);

helloProxy.say("Jack");

}

没错，意思就是，用我写的这个通用的 DynamicProxy 类去包装 HelloImpl 实例，然后再调用 JDK 给我们提供的 Proxy 类的工厂方法 newProxyInstance() 去动态地创建一个 Hello 接口的代理类，最后调用这个代理类的 say() 方法。

运行一下，结果和以前一样，动态代理成功了。其实，动态代理就是帮我们自动生成 XxxProxy 类的法宝啊！

要注意的是，Proxy.newProxyInstance() 方法的参数实在是让我“蛋碎一地”！

参数1：ClassLoader  
参数2：该实现类的所有接口  
参数3：动态代理对象

调用完了还要来一个强制类型转换一下。

wocao！这一坨 shi 一定要想办法封装一下，避免再次发生到处都是 Proxy.newProxyInstance()，这样架构师又要骂我了。于是我将这个 DynamicProxy 重构了：

public class DynamicProxy implements InvocationHandler {

...

@SuppressWarnings("unchecked")

public <T> T getProxy() {

return (T) Proxy.newProxyInstance(

target.getClass().getClassLoader(),

target.getClass().getInterfaces(),

this

);

}

...

}

我在 DynamicProxy 里添加了一个 getProxy() 方法，无需传入任何参数，将刚才所说的那一坨 shi，放在这个方法中，并且该方法返回一个泛型类型，就不会强制类型转换了。方法头上加那个 @SuppressWarnings("unchecked") 注解表示忽略编译时的警告（因为 Proxy.newProxyInstance() 方法返回的是一个 Object，这里我强制转换为 T 了，这是向下转型，IDE 中就会有警告，编译时也会出现提示，很烦）。

好了，这下子使用 DynamicProxy 就简单了吧：

public static void main(String[] args) {

DynamicProxy dynamicProxy = new DynamicProxy(new HelloImpl());

Hello helloProxy = dynamicProxy.getProxy();

helloProxy.say("Jack");

}

确实简单用 2 行代理就去掉了前面的 7 行代码（省了 5 行），架构师看到了这样的代码肯定会表扬我！

经过一番代码重构后，我提交了所有的代码，架构师看到了，没有吱声…… 可我总算学会了动态代理。

用了这个 DynamicProxy 以后，我觉得它还是非常爽的，爽的地方是，接口变了，这个动态代理类不用动。而静态代理就不一样了，接口变了，实现类还要动，代理类也要动。但我也发现动 态代理并不是“万灵丹”，它也有搞不定的时候，比如说，我要代理一个没有任何接口的类，它就没有勇武之地了！这就是 JDK 给我们提供的动态代理，让我不知道该说什么了。

### CGLib代理

于是我又开始调研，能否代理没有接口的类呢？终于让我找到了这颗“银弹”！那就是 [CGLib](http://www.oschina.net/p/cglib) 这个类库。虽然它看起来不太起眼，但 Spring、Hibernate 这样牛逼的开源框架都用到了它。它就是一个在运行期间动态生成字节码的工具，也就是动态生成代理类了。说起来好高深，实际用起来一点都不难。我再搞一个 CGLibProxy 吧：

public class CGLibProxy implements MethodInterceptor {

public <T> T getProxy(Class<T> cls) {

        return (T) Enhancer.create(cls, this);

    }

public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {

before();

Object result = proxy.invokeSuper(obj, args);

after();

return result;

}

...

}

需要实现 CGLib 给我们提供的 MethodInterceptor 实现类，并填充 intercept() 方法。方法中最后一个 MethodProxy 类型的参数 proxy，值得注意！CGLib 给我们提供的是方法级别的代理，也可以理解为对方法的拦截（这不就是传说中的“方法拦截器”吗？）。这个功能对于我们这群屌丝程序员而言，如同雪中送炭 啊，此乃神器也！我们直接调用 proxy 的 invokeSuper() 方法，将被代理的对象 obj 以及方法参数 args 传入其中即可。

与 DynamicProxy 类似，我在 CGlibProxy 中也添加了一个泛型的 getProxy() 方法，便于我们可以快速地获取自动生成的代理对象。还是用一个 main() 方法来描述吧：

public static void main(String[] args) {

CGLibProxy cgLibProxy = new CGLibProxy();

HelloImpl helloProxy = cgLibProxy.getProxy(HelloImpl.class);

helloProxy.say("Jack");

}

仍然通过 2 行代码就可以返回代理对象了，与 JDK 动态代理不同的是，这里不需要任何的接口信息，对谁都可以生成动态代理对象（不管它是“屌丝”还是“高富帅”）。说它是神器，过分吗？

我一向都是以追求完美而著称，2 行代码返回代理对象，我觉得还是有些多余，我不想总是去 new 这个 CGLibProxy 对象，最好 new 一次，以后随时拿随时用。于是我想到了“单例模式”：

public class CGLibProxy implements MethodInterceptor {

private static CGLibProxy instance = new CGLibProxy();

private CGLibProxy() {

    }

public static CGLibProxy getInstance() {

return instance;

}

...

}

我加了以上几行代码，就搞定了！需要说明的是：这里有一个 private 的构造方法，就是为了限制外界不能再去 new 它了，换句话说，我在这里把它给“阉”了。

用一个 main() 方法来证明我的简单主义思想：

public static void main(String[] args) {

HelloImpl helloImpl = CGLibProxy.getInstance().getProxy(HelloImpl.class);

helloImpl.say("Jack");

}

没错吧？只需 1 行代码就可以获取代理对象了！