# MyEclipse相关设置

* 低版本的java不能运行高版本的javac编译的文件。
* 选择运行时的编译器：Window>Preferences>Java>Complier
* Java编译器5.0就是1.5，JDK1.5.0即JDK5.0，MyEclipse6.6自带的运行环境是5.0的
* 选择运行时的JAVA运行环境：Window>Preferences>Java>Installed JREs>添加>标准VM>JRE 主目录，如选择：C:\Java\jdk1.6.0\jre—>单击完成即可。
* 设置了编译器版本还不行，还要看看编译后的运行环境版本是什么，运行环境的版本也要设置一样，或者更高才可以，因为高版本编译的\*.class文件无法在低版本的运行环境下运行。
* 在创建项目的时候也能看到运行环境:  
  
* 查看个体工程的运行环境

1. 右击工程名>Run As>Run Configurations>右边窗口的JRE选项卡
2. 右击工程名>Preferences>Run/Debug Settins>选择右边窗口的类>Edit>JRE

## MyEclipse调式

public static void main(String args[]) {

int x=0;

x++;

System.out.println(x);

}

在x++;语句前面双击增加一个断点，在输出语句前面双击增加一个断点，右击代码窗口，选择Debug As>Java Application,然后在代码窗口选择x++中的x，然后右击>Watch,然后点击工具栏上的Step Over(F6)就可以查看该x变量的值的变化情况了。

## 代码模板

选中代码右击>Surround With

设置：Window>Preferences>Java>Editor>Templates>New：

Name处输入模板名称，Pattern处输入：

**try**{

${line\_selection}//代表当前选择的内容

}

**finally**{

${cursor}//代表光标定位到这里

}

单击OK>Aplly>OK。

## 从一个工作间导出一个工程到另一个工作间

1. 把这个工程的整个目录复制到新工作间根目录下
2. File>Import>General>Existing Projects into Workspace>Next>Browse…>选择工作间根目录或工程的根目录>确定> 在Projects勾选要导入的工程>Finish
3. 注意：如果导进来的工程的JRE System Libray的安装目录与我们的工作间的不一样就会出问题，解决方法：右击工程名＞Build　Path＞Configure Build Path…>切换到Libraries选项卡＞选择库>Remove，然后再加入我们自己的库即可。
4. Libraries选项卡各按钮介绍：
   1. Add JARs… 增加单独的一个JAR包，且这个JAR包是存放在工程目录下的。
   2. Add External JARs… 增加一个单独的JAR包，且这个JAR包不是存放在工程目中的。
   3. Add Library… 增加一个JAR包库，点击Add Library…后，选择库即可，其中“User　Library”选项为用户自定义的jar包库，选择该项然后按Next即可看到自定义的用户库，如果要增加一个，可单击User Libraries… >New>输入一个库名>OK>Add JARs..增加jar包进来即可。

# 静态导入

* + import语句可导入一个类或某个包中的所有类
  + import static 语句导入一个类中的某个静态方法或所有静态方法

实例：  
**package** cn.itcast.day1;

**import** **static** java.lang.Math.*max*; //静态导入一个类中的一个方法

**import** **static** java.lang.Math.\*; //静态导入一个类中的所有方法

**public** **class** StaticImport {

**public** **static** **void** main(String args[])

{

System.*out*.println(Math.*max*(3, 6));//静态导入后可真直接使用：max(3,6)，无需加类名

System.*out*.println(Math.*abs*(3 - 6));//求会对值

}

}

# 可变参数

特点：

1. 只能在参数列表的最后
2. …位于变量类型和变量名之间，前后有无空格都可以
3. 调用可变参数的方法时，编译器为该可变参数隐含创建一个数组，在方法体中以数组的形式访问可变参数。

当一个方法要接受的参数不固定时可以使用可变参数，如一个add()方法，可接受任意个数字进行相加，代码如下：

**public** **class** VariableParameter {

**public** **static** **void** main(String[] args) {

System.*out*.println(*add*(1, 2));

System.*out*.println(*add*(1,2,3));

}

**public** **static** **int** add(**int** x,**int** ... args)

{

**int** sum=x;

**for**(**int** i=0;i<args.length;i++)

{

sum+=args[i];

}

**return** sum;

}

}

块注释快捷键：选中要注释的代码，按Ctrl+Shift+/

# 增强for循环

语法：for(type 变量名: 集合变量名){ … }

功能：用变量名去逐个取出集合中的元素，每取出一个元素就执行一次{ }中的语句

注意事项：迭代变量必须在( )中定义！

集合变量可以是数组或实现了Iterable接口的集合类

举例：

**public** **static** **int** add(**int** x,**int** ... args) {

**int** sum=x;

**for**(**int** arg:args) {

sum+=arg;

}

**return** sum;

}

# 基本数据类型的自动装箱与拆箱（这是1.5版本的特性）

自动装箱： Integer num = 12; // 自动把一个基本数据类型装成了一个Integer对象类型再赋给num引用变量

自动拆箱： System.out.println(num + 12); // 把Integer类型的num变量自动转换成基本数据类型再与12相加

基本数据类型的对象缓存：

Integer num1 = 12;

Integer num2 = 12;

System.out.println(num1 == num2); // true

Integer num1 = 129;

Integer num2 = 129;

System.out.println(num1 == num2); // false

基本数据类型int转换为Integer类型，如果这个数的大小在1字节大小之间（-128~127），则把这个数字放到缓存池里面，如果下次用到就直接在缓存池取出，而不用再重新分配。这就叫享元模式（flyweight）

Integer num1 = Integer.valueOf(12);

Integer num2 = Integer.valueOf(12);

System.out.println(num1 == num2); // true

# 枚举

枚举就是要让某个类型的变量的取值只能为若干个固定值中的一个，否则，编译器就会报错。枚举可以让编译器在编译时就可以控制源程序中填写的非法值，普通变量的方式在开发阶段无法实现这一目标。

普通类如何实现枚举功能：

* 私有的构造方法
* 每个元素分别用一个公有的静态成员变量表示
* 可以有若干公有方法或抽象方法，例如，要提供nextDay方法必须是抽象的。

星期天用0表示，星期一到星期六用1~6表示。

英文，简写，中文

Monday , Mon 星期一

Tuesday, Tue 星期二

Wednesday，Wed 星期三

Thursday，Thu 星期四

Friday，Fri 星期五

Saturday，Sat 星期六

Sunday，Sun 星期日

**实例：模拟枚举的功能，让一个类只能产生固定的实例。**

**public** **class** EnumTest {

**public** **static** **void** main(String[] args) {

WeekDay mon = WeekDay.*MON*;

WeekDay nextday = WeekDay.*MON*.nextDay();

System.*out*.println(nextday);//打印一个实例对象名会自动调用该对象的toString()方法。

}

}

**public** **abstract** **class** WeekDay {

**private** WeekDay(){}

**public** **static** **final** WeekDay *MON* = **new** WeekDay(){

**public** WeekDay nextDay() {

**return** *SUN*;

}

};

**public** **static** **final** WeekDay *SUN* = **new** WeekDay(){

**public** WeekDay nextDay() {

**return** *MON*;

}

};

**public** **abstract** WeekDay nextDay();

**public** String toString() {

**return** **this** == *MON*?"MON":"SUN";

}

}

**枚举的基本应用**

* 扩展：枚举类的values，valueOf，name，toString，ordinal等方法
* 总结：枚举是一种特殊的类，其中的每个元素都是该类的一个实例对象。

实例：

**public** **class** EnumTest {

**public** **static** **void** main(String[] args) {

WeekDay weekday=WeekDay.*MON*;

System.*out*.println(weekday);

System.*out*.println(weekday.toString());

System.*out*.println(weekday.name());//返回这个实例名

System.*out*.println(weekday.ordinal());//打印MON元素在枚举列表中的序号，第一位从0开始

System.*out*.println(weekday.getClass());//返回这个实例的类名

//以下是枚举的静态方法

System.*out*.println(WeekDay.*valueOf*("SAT"));//通过指定字符串返回一个枚举实例。

System.*out*.println(WeekDay.*values*()[0]);//把所有的枚举元素封装成一个数组返回

}

**public** **enum** WeekDay {

*SUN*(1),*MON*(),*TUE*,*WED*,*THU*,*FRI*,*SAT*; //此处的分号可要可不要,如果还有成员的话，那么分号必须要

//如果有还有其他类成员，则这个元素列表要放在所有类成员的最前面

//在元素后面加（参数），则可以指定调用哪个构造方法

**private** WeekDay(){ //枚举的构造方法必须是私有的

System.*out*.println("First");

}

**private** WeekDay(**int** day){

System.*out*.println("Second");

}

}

}

注：内部类可以有4个修饰符，而外部类只能有public和默认两种修饰符

内部类编译后的文件形式为： 外部类名$内部类名

匿名内部类编译后的形式为： 外部类名$1

外部类名$2

还有可能是这样的：外部类名$内部类名&1

外部类名$内部类名&2

外部类名$内部类名&3

**实例：定义一个交通灯枚举**

功能：实现只有红、绿、黄三种灯，实现多久切换成另一个灯且另一个灯是哪个灯

**public** **enum** TrafficLamp {

*RED*(60){

**public** TrafficLamp nextLamp() {

**return** *GREEN*;

}

},

*GREEN*(40) {

**public** TrafficLamp nextLamp() {

**return** *YELLOW*;

}

},

*YELLOW*(5){

**public** TrafficLamp nextLamp() {

**return** *RED*;

}

};

**private** **int** time;

**public** **abstract** TrafficLamp nextLamp();//因为这里定义为abstract方法后这个枚举类就是一个抽象类，

//所以不能直接用new产生这个枚举的实例，必须定义一个子类来产生实例，所以在枚举元素名的后面加{ 代码 }来

//产生匿名内置类，而这元素名就是这个匿名内置类的实例的引用变量。

**private** TrafficLamp(**int** time) {

**this**.time=time;

}

}

枚举只有一个成员时，就可以作为一种单例的实现方法。

# 反射

## Class

**反射的基石→Class类**

**java程序中的各个java类属于同一类事物，描述这类事物的Java类就是Class。**

Class类描述了哪些方面的信息呢？类的名字，类的访问属性，类所属的包名，字段名称的列表，方法名称的列表等等。

众多的人用Person表示，众多的Java类用Class表示

Person类它的实例对象就是张三、李四这样一个个具体的人

**Class类的各个实例对象又分别对应什么呢**：

* + 对应各个类在内存中的字节码，例如，Person类的字节码，ArrayList类的字节码，等等。
  + 一个类被类加载器加载到内存中，占用一片存储空间，这个空间里面的内容就是类的字节码，不同的类的字节码是不同的，所以它们在内存中的内容是不同的，这一个个的空间可以分别用一个个的对象来表示，这些对象显然具有相同的类型，这个类型是什么呢？就是Class类型

**如何得到各个字节对应的实例对象（Class类型）**

* + 类名.class，例如：System.class
  + 对象.getClass()，例如，new Date().getClass()
  + Class.forName(“类名”),例如：Classic.forName(“java.util.Date”);这里有两种情况：1、如果这个类已经加载过，这个类的字节码已经存在Java虚拟机中了，则直接返回。如果这个类还没有加载过，则类加载器先把这个类加载到Java虚拟机缓存起来，然后再返回这个类字节码。

Person person = new Person();

Date类

Math类  
Class cls1 = Date.class//字节码

Class cls2 = Person.class //字节码

person.getClass(); // 获取该实例所属的类的字节码

Class.forName(“java.lang.String”) // 返回String类的字节码

**public** **class** ReflectTest {

**public** **static** **void** main(String args[]) **throws** Exception {

String str="abc";

Class cls1 = str.getClass();

Class cls2 = String.**class**;

Class cls3 = Class.*forName*("java.lang.String");

System.*out*.println(cls1 == cls2);

System.*out*.println(cls1 == cls3);

}

}

### 原始数据类型都有对应的Class

有九种原始数据类型：八个基本类型和 void。这些类对象由 Java 虚拟机创建，与其表示的基本类型同名，即 boolean、byte、char、short、int、long、float、double和void.class。

参看Class.isPrimitive方法的帮助

如何知道一个类是否是原始数据类型：Class.isPrimitive()

Int.class = = Integer.TYPE，每一种原始数据类型的class都有包装类的TYPE保存了常量，如：Boolean.TYPE、Long.TYPE

数组类型的Class实例对象

Class.isArray() // 判断一个Class是否是数组类型

总之，只要是在源程序中出现的类型，都有各自的Class实例对象，例如int, int[],void等类型

**public** **static** **void** main(String args[]) **throws** Exception {

Class cls1 = str.getClass();

System.*out*.println(cls1.isPrimitive()); // 结果false，该方法：判定指定的 Class 对象是否表示一个基本类型。

System.*out*.println(**int**.**class**.isPrimitive()); // 结果true

System.*out*.println(**int**.**class** == Integer.**class**); // 结果 false

System.*out*.println(**int**.**class** == Integer.*TYPE*); // 结果true,TYPE属性代表Integer对象所包装的的基本类型的字节码

}

## 反射

反射就是把Java类中的各种成分映射成相应的java类。例如，一个个Java类用一个Class类的对象来表示，一个类中的组成部分：成员变量，方法，构造方法，包等等信息也用一个个的Java类来表示，就像汽车是一个类，汽车中的发动机，变速箱等等也是一个个的类。表示java类的Class类显然要提供一系列的方法，来获得其中的变量，方法，构造方法，修饰符，包等信息，这些信息就是用相应类的实例对象来表示，它们是Field、Method、Contructor、Package等等。

一个类中的每个成员都可以用相应的反射API类的一个实例对象来表示，通过调用Class类的方法可以得到这些实例对象后，得到这些实例对象后有什么用呢？怎么用呢？这正是学习和应用反射的要点。

## Constructor

* Constructor类代表某个类中的一个构造方法
  + 得到某个类所有的构造方法：

例子：Constructor[ ] constructors=Class.forName(“java.lang.String”).getConstructors();

* 得到某一个构造方法：
  + 例子：Constructor constructor=Class.forName( “java.lang.String” ).getConstructor(StringBuffer.class);//获得方法时要用到类型
* 创建实例对象：
  + 通常方式：String str = new String(new StringBuffer(“abc”))；
  + 反射方式：String str = (String)constructor.newInstance(new StringBuffer(“abc”));
* Class.newInstance()方法：
  + 例子：String object=(String)Class.forName(“java.lang.String”).newInstance();
  + 该方法内部先得到默认的构造方法，并缓存起来（说明反射是性能低的，不然就不会缓存起来了），然后用该构造方法创建实例对象。

## Field

Field类代表某个类中的一个成员变量

问题：得到的Field对象对应到类上面的成员变量，还是对应到对象上的成员变量？类只有一个，而该类的实例对象有多个，如果是与对象关联，那该关联哪个呢？所以字段Field代表的是x的定义,而不是具体的x变量。

**public** **class** **FieldTest** {

**private** **int** x;

**public** **int** y;

**public** FieldTest(**int** x, **int** y) {

**this**.x = x;

**this**.y = y;

}

**public** **static** **void** main(**String**[] args) **throws** Exception {

**FieldTest** ft = **new** **FieldTest**(3, 5);

**Field** fieldY = ft.getClass().getField("y");//该方法只能获取公共的字段

**System**.*out*.println(fieldY.get(ft));

**Field** fieldX = ft.getClass().getDeclaredField("x");//获取字段（包括公有的和私有的）

fieldX.setAccessible(**true**);//设置私有字段可以被访问

**System**.*out*.println(fieldX.get(ft));

}

}

注：字节码的比较用等号（=）比较，而不用equals。

实例：将一个对象中的所有String成员变量中的字母b改为a。

**public** **class** **FieldTest** {

**public** **String** str1="ball";

**public** **String** str2="basketball";

**public** **String** str3="itcast";

**public** **String** toString() {

**return** str1 + ":::" +str2 + ":::" + str3;

}

**public** **static** **void** main(**String**[] args) **throws** Exception {

**FieldTest** ft = **new** **FieldTest**();

ft.changeStringValue(ft);

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

}

**public** **void** changeStringValue(**Object** obj) **throws** Exception {

**Field** [] fields = obj.getClass().getFields();

**for**(**Field** field: fields) {

**if**(field.getType() == **String**.**class**) {

**String** oldStr = (**String**)field.get(obj);

**String** newStr = oldStr.replace('b', 'a');

field.set(obj, newStr);

}

}

}

}

**Method类**

Method类代表某个类中的一个成员方法

得到类中的某一个方法：

Method charAt = Class.forName(“java.lang.String”).getMethod(“charAt”, int.class);

调用方法：

通常方式：System.out.println(str.charAt(1));

反射方式：System.out.println(charAt.invoke(str,1));

如果传递给Method对象的invoke()方法的一个参数为null，这有着什么样的意义呢？说明该Method对象对应的是一个静态方法！

jdk1.4和jdk1.5的invoke方法的区别：

jdk1.5：public Object invoke（Object obj,Object… args）

jdk1.4：public Object invoke（Object obj,Object[] args），因为jdk1.4没有可变参数，所以要想接受多个对象只能使用数组

注意：参数类型是用Class来表示的，反射不能读非public的类

实例，调用用反射调用String类中的charAt方法

**String** s = "abcd";

**Method** str = **String**.**class**.getMethod("charAt", **int**.**class**);

**System**.*out*.println(str.invoke(s, 2));//输出 c，虽然这里给的参数是2，是一个基本类型，但是会自动装箱成Integer对象的。用JDK1.4来使用这个方法为：

Object[] array = new Object[]{2};

Str.invoke(s, array); // 按jdk1.4的接收数组的方式来调用

Str.invoke(s, new Object[]{2}); // 按jdk1.4的接收数组的方式来调用

//invoke(Object,Object …args)如果第一个参数为null，则说明调用的这个方法是静态方法，因为静态方法被调用时不需要产生对象，所以不用指定对象。

注：人在黑板上画圆，这里有3个对象：人、黑板、圆，那画圆的方法应该属于哪个对象上的方法呢？在圆对象上有圆心、半径的私有成员属性，而画圆需要这些属性，不可能让外面的对象来方法吧，所以画圆的方法属于圆对象。记住一句话，就是谁拥有数据，谁就拥有方法。

## 用反射方式调用main方法

目标：写一个程序，这个程序能够根据用户提供的类名，去执行该类中的main方法。

问题：启动Java程序的main方法的参数是一个字符串数组，通过 反射方式来调用这个main方法时，如何为invoke方法传递参数呢？按jdk1.5的语法，整个数组是一个参数，而按jdk1.4的语法，数组中的每个元素对应一个参数，当把一个字符串数组作为参数传递给invoke方法时，javac会到底按照哪种语法进行处理呢？jdk1.5肯定要兼容jdk1.4的语法，会按jdk1.4的语法进行处理，即把数组打散成若干个单独的参数。所以，在给main方法传递参数时，不能使用mainMethod.invoke(null,new String[]{"xxx"});javac只把它当作jdk1.4的语法进行理解。而不把它当作jdk1.5的语法解释，因此会出现参数类型不对的问题。

解决办法：

mainMethod.invoke(null,new Object[]{new String[]{“xx”}});

mainMethod.invoke(null,(Object) new String[]{“xx”});编译器会作特殊处理，编译时不把参数当作数组看待，也就不会把数组打散成若干个参数了。.

**用普通的方法调用mian方法：**

class TestArguments {

public static void main(String[] args) {

for(String arg: args)

System.out.println(arg);

}

}

class InvokeMain {

public static void main(String[] args) {

TestArguments().main(new String[] {"abc","111"});

}

}

**用反射的方式调用mian方法：**

注：在“运行方式>运行配置”中的“自变量”选择卡中的“程序自变量”的内容即为运行类时传递给main函数的参数。

在使用Class.foName()获取一个类的字节码时，这里面的类名需要写完整方式，即包括包名。

class InvokeMain {

public static void main(String[] args) {

**String** startClassName = args[0];

**Class** startClassByteCode = **Class**.*forName*(startClassName);

**Method** mainMethod = startClassByteCode.getMethod("main", **String**[].**class**);

mainMethod.invoke(**null**, **new** **Object**[] {**new** **String**[] {"aaa","bbb","ccc"}});

}

}

可以看到使用反射调用main的好处，即编写代码时不知道要调用哪个类的main方法，在运行时你给一个类名作参数，给哪个类名就运行哪个类的main方法。

## 数组的反射

具有相同维数和元素的数组属于同一个类型，即具有相同的Class实例对象。

代表数组的Class实例对象的getSuperClass()方法返回父类为Object类对应的Class。

Array.asList(Object[] obj ),这是jdk1.4的

Array.asList(Object… obj ),这是jdk1.5的,由于高版本兼容低版本，所以当这个方法传递的对象是一个数组时，则使用jdk1.4的处理方式，如果给的是一个一个的对象参数，则用jdk1.5的处理方式。

基本类型的一维数组可以被当作Object类型使用，不能当作Object[]类型使用；非基本类型的一维数组，既可以当作Object类型使用，又可以当做Object[]类型使用。

Array.asList( )方法处理int[]和String[]时的差异。实例：

**int** [] a = **new** **int**[] {1,2,3};//这个数组只能当作一个Object类型使用

**String** [] strs = **new** **String**[] {"a","b","c"};

**System**.*out*.println(**Arrays**.*asList*(a));//输出为：[[I@1bc4459]

**System**.*out*.println(**Arrays**.*asList*(strs));//输出为：[a, b, c]

Array工具类用于完成对数组的反射操作。

实例：编写一个方法，参数为一个Object，当这个Object为一个对象时则打印该对象，如果为数组，则打印数组元素：

**public** **static** **void** main(**String**[] args) **throws** Exception {

**String**[] str1 = **new** **String**[] {"a","b","c"};

**String** str2 = "ddd";

*printObj*(str1);

*printObj*(str2);

}

**public** **static** **void** printObj(**Object** obj)

{

**Class** objClass = obj.getClass();

**if**(objClass.isArray()) {

**int** len = **Array**.*getLength*(obj);

**for**(**int** i = 0 ;i<len;i++)

**System**.*out*.println(**Array**.*get*(obj,i));

} **else**

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

}

思考：怎么得到数组中的元素类型呢？

如：int [] a = new int[3];这是取得数组a是什么类型的? 答：int[]类型，说明里面的元素全都是int类型

如：Object[] a = new Object[]{“a”,1};这里数组元素可以为String，也可以为int，所以也是无法取得数组a中装的元素是什么类型的，但是可以取得每个具体的数组中的元素的类型：a[0].getClass().getName();

## 反射的作用：实现框架功能

**框架与框架要解决的核心问题**

我做房子卖给用户住，由用户自己安装门窗和空调，我做的房子就是框架，用户需要使用我的框架，把门窗插入进我提供的框架中，框架与工具类有区别：

工具类：用户的类调用工具类

框架： 框架的类调用用户的类

**框架要解决的核心问题**

我在写框架（房子）时，你这个用户可能还在上小学，还不会写程序呢？我写的框架程序怎样能调用到你以后写的类（门窗）呢？

因为在写程序时无法知道要被调用的类名，所以，在程序中无法直接new某个类的实例对象，而要用反射方式来做。

**实例：将上面的例子用反射来实现,**

我们将要使用的集合名称保存到一个“config.properties”配置文件中，然后从这里读取使用这个集合类，这就是一个小框架实例应用。

其中ReflectPoint类的代码不变。

1. 右击项目>新建>文件>在文件名处输入：config.properties>确定，在代码窗口的左下角点击“source”,然后输入 className = java.util.HashSet
2. 其他代码如下：

**public** **class** **ReflectTest2** {

**public** **static** **void** main(**String**[] args) **throws** Exception {

**File** f = **new** **File**("config.properties");//这里一定要记住用完整的路径，但完整的路径不是给条固定不变的绝对路径，而是通过运算出来的绝对路径。

**InputStream** in = **new** **FileInputStream**(f);

**Properties** p = **new** **Properties**();

p.load(in);

**String** className = p.getProperty("className");

Collection collections = (Collection)**Class**.*forName*(className).newInstance();

**ReflectPoint** p1 = **new** **ReflectPoint**(3,3);

**ReflectPoint** p2 = **new** **ReflectPoint**(4,4);

**ReflectPoint** p3 = **new** **ReflectPoint**(3,3);

collections.add(p1);

collections.add(p1);

collections.add(p2);

collections.add(p3);

**System**.*out*.println(collections.size());

}

}

上面中的代码可以改为 ：

1. InputStream in = ReflectTest2.class.getClassLoader().getResourceAsStream("cn/itcast/day1/config.properties"); 这种方法有个缺点就是不能保存数据，因为没有输出流。

注：通常把配置文件放到java源文件的目录下，MyEclipse会把这个文件复制到java的class文件目录中的。

2、InputStream in = ReflectTest2.class.getResourceAsStream("config.properties");//这里的路径是相对于ReflectTest2.java文件所在的目录，如果这个config.properties文件放在ReflectTest2.java当前目录下面的resouce目录中，则应该写为：

InputStream in = ReflectTest2.class.getResourceAsStream("resource/config.properties");

这种写法是相对.java文件所在目录的相对目录的方法

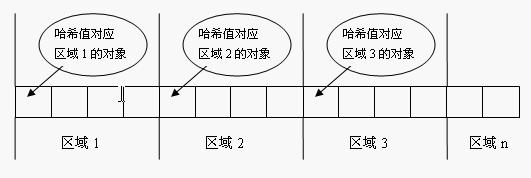
还有一种是相对.java文件根目录的相对目录的方法，如：

如果在参数的最前面加上了“/”，则代表是相对ReflectTest2.java的根目录的，如ReflectTest2.java在包：cn/itcast/day1中，即上面代码要写成这样：

InputStream in = ReflectTest2.class.getResourceAsStream("/cn/itcast/day1/config.properties ");

# HashSet类与hashCode方法分析

如果想查找一个集合中是否包含有某个对象，大概的程序代码怎样写呢？你通常是逐一取出每个元素与要查找的对象进行比较，当发现某个元素与要查找的对象进行equals方法比较的结果相等时，则停止继续查找并返回肯定的信息。假如一个集合中有一万个元素，并且没有包含要查找的对象时，则意味着你的程序需要从集合中取出一万个元素进行逐一比较才能得到结论，这是比较低效的。于是有人发明了一种哈希算法来提高从集合中查找元素的效率，这种方式将集合分成若干个存储区域，根据一个对象的哈希码就可以确定该对象应该存储在哪个区域，如下图：



HashSet就是采用哈希算法存取对象的集合，它内部采用对某个数字n进行取余的方式对哈希码进行分组和划分对象的存储区域，比如HashSet分成了8个存取区域，当一个对象要存进来时先进行哈希值计算，然后将该值对8取余，则余数为0-7，余数为哪一个就存到哪一个区域，Object类中定义了一个hashCode（）方法来返回每个Java的哈希码，当从HashSet集合中查找某个对象时，Java系统首先调用对象的hashCode()方法获得该对象的哈希码，然后根据哈希码找到相应的存储区域，并取出该存储区域内的每个元素与该对象进行equals方法比较，这样不用遍历集合中的所有元素就可以得到结论。可见，HashSet集合具有很好的对象检索性能，但是，HashSet集合存储对象的效率相对要低些，因为在向HashSet集合中添加一个对象时，要先计算出该对象的哈希码和根据这个哈希码确定该对象在集合中的存放位置。

所以，只有当一个集合的存在区域是采用哈希算法的，这时的hashCode方法算出来的哈希码才有价值，否则的话该方法是没有用。

为了保证一个类的实例对象能在HashSet正常存储，要求这个类的两个实例对象用equals（）方法比较的结果相等时，还要它们的哈希码也相等。就是说，如果obje1.equals(obj2)的结果为true，那么obj1.hashCode()==obj2.hashCode()也必须要为true。如果一个类的hashCode()方法没有遵循这个要求，那么，当这个类的两个实例对象用equals()方法比较的结果相等时，它们本来应该无法被同时存储进Set集合中，但是，如果将它们存储进HashSet集合时，由于它们的hashCode()方法的返回值不同，第二个对象首先按哈希码计算，可能会被放进与第一个对象不同的区域中，这样，它就不可能与第一个对象进行equals方法比较了，也就可能被存储进HashSet集合中了。Object类中的hashCode（）方法不能满足对象被存入到HashSet中的要求，因为它的返回值是通过对象的内存地址推算出来的，所以，如果两个对象的成员属性相同，但是如果这两个对象的内存地址不同则它们的hashCode码不同。

只有类的实例对象要被采用哈希算法进行存储和检索时，这个类才需要按要求覆盖hashCode方法。即使程序可能暂时不会用到当前类的hashCode方法，但是为它提供一个hashcode方法也不会有什么不好，没准以后什么时候又用到这个方法了，所以，通常要求hashCode方法和equals方法被同时覆盖。

提示：

1. 通常来说，一个类的两个实例对象用equals方法比较的结果相等时，它们的哈希码也必须相等，但反之则不成立，即equals方法比较的结果不相等时，它们的哈希码有可能相等，或者说哈希码相同的两个对象的equals方法比较的结果可以不等，例如，字符串“bb”和“Aa”的equals方法比较结果肯定不相等，但它们的hashCode方法返回值却相等。
2. 当一个对象被存储进HashSet集合中以后，就不能修改这个对象中的那些参与计算哈希值的字段（成员变量）了，否则，对象修改后的哈希值与最初存储进HashSet集合中的哈希值就不同了，在这种情况下，即使在contains方法使用该对象的当前引用作为参数去集合中检索对象，也将返回找不到对象的结果，这也会导致无法从HashSet集合中单独删除当前对象，从而造成内存泄露。

**实例：**

往HashSet集合里装对象，只有对象的equals方法比较相同，且hashCode码相同时，这个对象才不会被存进去，但实际上，只要成员属性相同我们就认为他是同一个对象，不应该再存进去，所以我们要覆盖该对象的equals方法与hashCode方法来达到目的。

**public** **class** **ReflectTest2** {

**public** **static** **void** main(**String**[] args) {

//Collection collections = new ArrayList();

Collection collections = **new** **HashSet**();

**ReflectPoint** p1 = **new** **ReflectPoint**(3,3);

**ReflectPoint** p2 = **new** **ReflectPoint**(4,4);

**ReflectPoint** p3 = **new** **ReflectPoint**(3,3);

collections.add(p1);

collections.add(p2);

collections.add(p3);

collections.add(p1);

**System**.*out*.println(collections.size());

}

}

**class** **ReflectPoint** {

**int** x; **int** y;

**public** ReflectPoint(**int** x, **int** y) {

**this**.x = x;

**this**.y = y;

}

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + x;

result = prime \* result + y;

**return** result;

}

**public** **boolean** equals(**Object** obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

**ReflectPoint** other = (**ReflectPoint**) obj;

**if** (x != other.x)

**return** **false**;

**if** (y != other.y)

**return** **false**;

**return** **true**;

}

}

（1）当没有重写hashCode与equals时，Java默认的hashCode方法的计算是用对象的内存地址参与计算的，所以只要对象的内存地址不同，则hashCode码就不同。Java默认的equals方法是比较两个对象的引用，所以只有引用相同才会返回true；

上面例子的3个对象的内存地址都不同，所以他们的hashCode码都不一样，p1对象存了两次，但只存进去一次，因为是同一个对象，所以它的equals返回结果肯定为true，它的hashCode码也一定相等，所以不会被存两次。所以在这里能存进去3个对象。

（2）覆盖后，hashCode的计算是对象的成员变量参与计算的，只要成员变量相同，hashCode就相同。而且equals方法比较的也是成员变量，只要成员变量相同则返回true。

p1和p3的equasl返回为true，而且他们的hashCode计算后相同，所以只存进了p1。所以在这里能存进去2个对象（p1、p2）。

# 内省

## 内省介绍

内省的英文：Introspector

introspect /,ɪntrə(ʊ)'spekt/ n. 反省；内省

Introspector n. 内省

JavaBean是一种特殊的Java类，主要用于传递数据信息，这种Java类中的方法主要用于访问私有字段，且方法名符合某种命名规则setXXX、getXXX或isXXX。

JavaBean必须是public类型的，且要有一个public类型的无参构造函数，以便工具类创建这个在类的实例对象。

如果要在两个模块之间传递多个信息，可以将这些信息封装到一个JavaBean中，这种JavaBean的实例对象通常称这为值对象（Value Object,简称VO）。这些信息在类中用私有字段来存储，如果读取或设置这些字段的值，则需要通过一些相应的方法来访问。JavaBean的属性是根据其中的setter和getter方法来确定的，而不是根据其中的成员变量。如果方法名为setId，中文意思即为设置id，至于你把它存到哪个变量上并没有关系。如果方法名为getId，中文意思即为获取id，至于你从哪个变量上取也没有关系。换言之，只要set和get或is是成对的，即将并没有真正的变量也没关系，如下：

**public void** setColor(**int** color) {  
 System.***out***.println(**"设置Color成功"**);  
}  
**public int** getColor() {  
 **return** 255;  
}

这是一对方法setColor和getColor方法，所以我们可说这个类有一个color属性。

JavaBean属性名规则：删除set/get/is，剩下的内容就是属性名，但是剩下的第一个字母需要小写，如果剩下的内容的第二个字母是大写的，则第一个字母也保持大写不变。

* + - setId()的属性名：Id -> id ，Id中的第二位d为小写，所以第I变成小写
    - isLast()的属性名：Last -> last
    - getCPU的属性名：CPU -> CUP，CPU中的第二位P为大写，所以什么都不用变

总之，一个类被当作JavaBeam使用时，JavaBean的属性是根据方法名推断出来的，它根本看不到Java类内部的成员变量。

直接调用JavaBean的get方法即可获取属性值，一行代码的事，最简单的事情为什么要使用内省操作来变得复杂呢？举例：在JavaEE中，服务器有时会接收很多的参数，有时这些参数可以保存到一个对象中，我们知道把一个json转换为JavaBean很简单，可以通过Gson完成。但是把一些key-value的参数转换为JavaBean时，又有什么工具来完成呢？使用内省一行代码就可以完成，如果自己写反射的话会比较麻烦，你需要遍历所有的key，通过反射获取到key对应的方法，再设置值。

注：JavaBean的类必须是public类型的，方法也必须是public类型的，成双成对的方法才会被当成属性，比如getAge()/setAge(int age)，这样就可以认为有一个age属性，而如果只有一个方法，如只有getAddress()，则不会认为有address属性。

内省所在的包：java.beans

## PropertiDescriptor

操作JavaBean的内省类：PropertiDescriptor，属性描述器，描述了JavaBean中的一个属性，即把一个JavaBean的属性封装成一个对象

示例代码如下：

1、定义JavaBean类：

**public class** Person {  
 **private** String **name**;  
 **private int age**;  
 **public** Person(String name, **int** age) { **this**.**name** = name; **this**.**age** = age; }  
 **public** String getName() { **return name**; }  
 **public void** setName(String name) { **this**.**name** = name; }  
 **public int** getAge() { **return age**; }  
 **public void** setAge(**int** age) { **this**.**age** = age; }  
}

2、使用PropertyDescriptor类来获取JavaBean中的属性，并操作这个属性（读、写）

Person person = **new** Person(**"EvenDai"**, 28);  
String propertyName = **"name"**;  
PropertyDescriptor propertyDescriptor = **new** PropertyDescriptor(propertyName, Person.**class**);  
  
*// 操作属性：读*Object name = propertyDescriptor.getReadMethod().invoke(person);  
System.***out***.println(name);  
  
*// 操作属性：写*propertyDescriptor.getWriteMethod().invoke(person, **"Lily"**);  
System.***out***.println(person.getName());

3、实验单一方法：给Person增加两个单一的方法，如下：

**public** String getAddress() {  
 **return "广州"**;  
}

**public void** setColor(**int** color) {  
  
}

这里将会报异常，说找不到“setAddress”方法。

Person person = **new** Person(**"EvenDai"**, 28);  
PropertyDescriptor address = **new** PropertyDescriptor(**"address"**, Person.**class**);

这里将报找不到“isColor”方法，这有点奇怪，为什么不是getColor方法，我们给Person加上isColor方法后确实就编译通过了如，发下：

PropertyDescriptor address = **new** PropertyDescriptor(**"color"**, Person.**class**);

**public int** isColor() {  
 **return** 255;  
}

## BeanInfo

PropertyDescriptotr描述了JavaBean的一个属性，而BeanInfo则描述了JavaBean的所有属性，示例如下：

Person person = **new** Person(**"EvenDai"**, 28);  
BeanInfo beanInfo = Introspector.*getBeanInfo*(Person.**class**);  
**for** (PropertyDescriptor propertyDescriptor : beanInfo.getPropertyDescriptors()) {  
 System.***out***.println(propertyDescriptor.getName());  
 Object value = propertyDescriptor.getReadMethod().invoke(person);  
 System.***out***.println(value);  
}

注：这里侍有三个值输出，因为Object有一个getClass方法被当成了属性，但是并没有setClass的方法，这说明BeanInfo对象比较奇怪，只有单一的get或set方法也会被当成属性，这个比较诡异，实验如下：

在Person类中增加如下方法：

**public** String getAddress() {  
 **return "广州"**;  
}

**public void** setColor(**int** color) {  
  
}

再次运行上面的代码，会发现打印出属性名：address和color，调用address属性的getWriteMethod()和调用color属性的getReadMethod()方法将返回null，因为它们没有这些方法。

## BeanUtils

BeanUtils就是对反射和自省功能的包装，使其更易于使用。

官网：<https://commons.apache.org/proper/commons-beanutils/>

BeanUtils依赖于logging库，logging官网：<https://commons.apache.org/proper/commons-logging/>



使用说明：<https://commons.apache.org/proper/commons-beanutils/javadocs/v1.9.4/apidocs/org/apache/commons/beanutils/package-summary.html#package_description>

  
 在这个页面的左侧用下载连接，还有用户指南。

这是Apche提供的Java工具包，下载后解压，解压后有jar包和api文档，有3个jar包，最的那个是另外两个的总和，也就是如果你用到的功能少，可以使用简化版本的。这里我们用完整版的jar包，复制commons-beanutils.jar包，右击项目，新建一个文件夹为lib，右击该文件夹选择粘贴，然后右击该jar包，选择“构建路径->添加到构建路径”即可。用这种方式添加Jar包的好处是当把这个项目发给别人时jar包也跟着一起过去了。 beanutils工具又使用一了logging工具，这也是apache公司出品的，在JavaEE开发中非常常用，下载到后解压，也是使用最大的那个完整功能的jar。

### 示例代码

Person person = **new** Person(**"EvenDai"**, 28);  
System.***out***.println(BeanUtils.*getProperty*(person, **"name"**));  
BeanUtils.*setProperty*(person, **"name"**, **"Lily"**);  
System.***out***.println(person.getName());

此时运行将会报错，说找不到某个类，这是因为beanutils工具包是Apache开发的，但是这个包同时也用到了其他的包中的类，所以这里还要导入用到的logging工具包才行。

假如在上在面的例子中Person类中有一个Date成员变量叫birthday,而这个Date类有一个setTime方法，这个方法名符合JavaBean的规范，所以可以把Date当成一个JavaBean来使用，相当于Date有了一个time属性，示例如下：

**public class** Person {  
 **private** Date **birthday** = **new** Date();  
 **public** Date getBirthday() {**return birthday**; }  
 **public void** setBirthday(Date birthday) {**this**.**birthday** = birthday; }  
}

System.***out***.println(BeanUtils.*getProperty*(person, **"birthday.time"**));

**注：BeanUtils**.*getProperty*(person, "birthday.time")这句代码就相当于获取person对象的birthday属性，再获取birthday属性的time属性。即获取person对象的birthday属性的time属性。这如果用前面的反射方法来自己做将是非常麻烦的事。

BeanUtils中还有PropertyUtils类可操作JavaBean，示例如下：

Person person = **new** Person(**"EvenDai"**, 28);  
System.***out***.println(PropertyUtils.*getProperty*(person, **"name"**));  
PropertyUtils.*setProperty*(person, **"name"**, **"Lily"**);  
System.***out***.println(person.getName());

使用PropertyUtils.get/setProperty（）方法，用法与BeanUtils几一样，区别：

1、PropertyUtils.setProperty()方法要求的参数为对象属性本身的类型，如对象属性类型为int，则set时需要的参数类型就需要int，get时返回的结果也是int。当然系统会装箱成Integer

BeanUtil.setProperty（）方法要求的参数String，如果对象属性是基本数据类型，我们传String它内部会自动转换。

2、PropertyUtils.getProperty()法是返回的是属性本身的类型

BeanUtil.getProperty（）方法返回的是字符串类型，即使属性本身是基本类型，也以String的类型舞蹈。

总结就是PropertyUtils的get/set方法是针对对象属性本身的类型进行操作的，而BeanUtils的get/set方法是针对字符串进行操作的。

3、BeanUtils中的其它方法：

* 1. static opyeProprties(dest, orig)把一个类上的属性复制到另一个类
  2. desribe(bean) 把bean中的属性封装到map中，属性名为key，值为value
  3. Populate(bean, map)，与describe方法功能相反，所map中的值封装到bean属性中。
  4. setProperty(map, key, value)修改map中指定key的值为指定的value

populate演示：

Map<String, String> map = **new** HashMap<>();  
map.put(**"name"**, **"Lily"**);  
map.put(**"age"**, **"18"**);  
Person person = **new** Person();  
BeanUtils.*populate*(person, map);  
System.***out***.println(person);

可以看到，age传的是String类型，BeanUtils会自动帮我们转换为int类型。

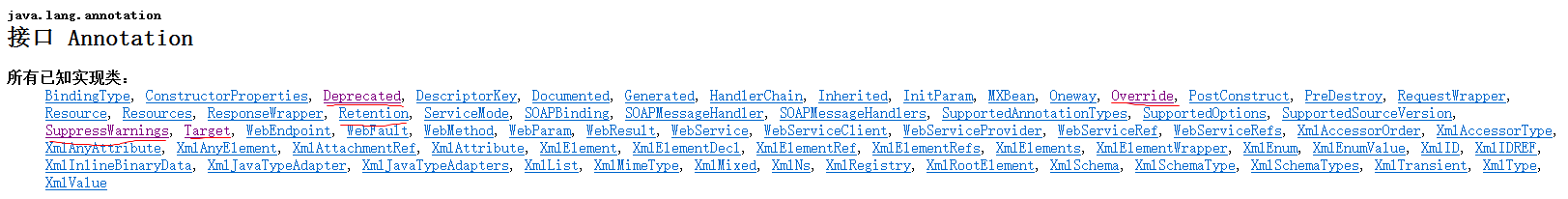
# 注解

注解功能非常重要，现在的开发模式基本上都是基于注解的，如EJB、Spring、Struts2和Hibnate都是基于注解的

小知识点：类名一般是名词，或者形容词 + 名词，整体来说是一个名词。而方法名一般是动词，或者动词 + 名称，整体代表一个动作。

Java中的注解在java.lang包中，还有java.lang.annotation包中，注解的超类为java.lang.annotation.Annotation，但是我们在开发一个注解时不需要像实现接口那样写一个implement

## Java中常用的注解

通过查看超类Annotation可方便查找到这些常注解：  


### SuppressWarnings

**public static void** main(String[] args) {  
 System.*~~runFinalizersOnExit~~*(**true**);  
}

在命令行窗口编译时会提示说这个方法已经过时的提示。在方法上加入：@SuppressWarnings(**"deprecation"**)，则在编译时就不会提示了，意思为，我知道它过时，但是我就需要使用过时的，你别提示我了，所以@SuppressWarnings(**"deprecation"**)就是告诉java编译器或者告诉ide我知道过时了，不要提示我了。

### Deprecated

当自己编写的一个方法，过时的时候又不想删，就可以加注解，以表示该方法已过时，如：

@Deprecated  
**public static void** sayHello() {  
 System.***out***.println(**"Hello!"**);  
}

则调用了这个方法后，在编译时就会提示过时。

### Override

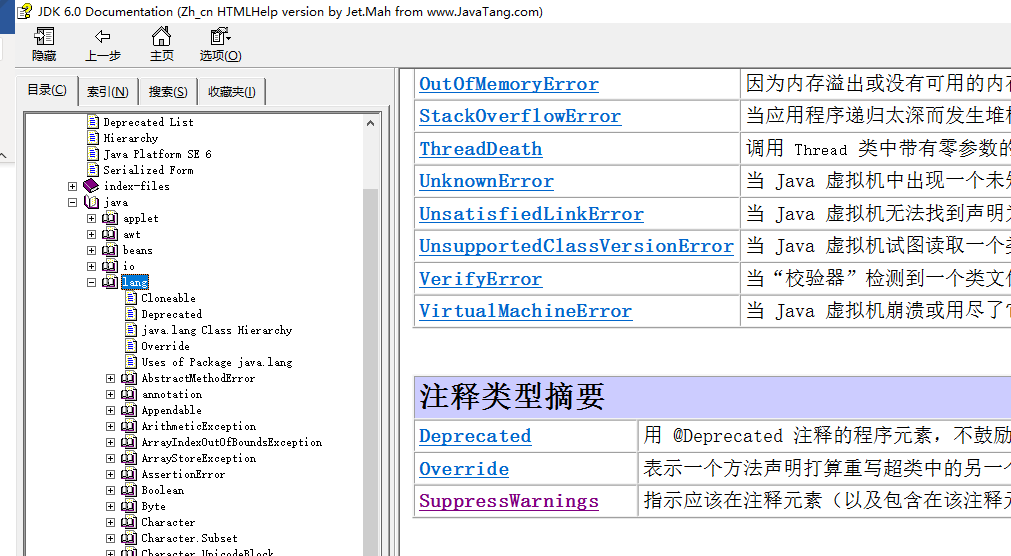
当覆盖一个方法时，要求参数型要一样，而有时候我们不小心写到不同的类型，所以运行时就会出错，而这种错误往往又很难发现原因，这时就可以在方法前面加上注解：@Override ，表示这个方法是覆盖的，当覆盖的方法类型不同时，则在编译时就会报错，这样就容易查找原因了。

比如Person类在覆盖equals方法时，方法中接收的参数类型是Object，如果你写成了Person，则不是覆盖，则你在使用equals时会发现结果不正确，这个还容易排查，难的是，比如HashMap之类的集合，他们会使用equals来比较对象，以进行删除或者查找操作，如果我们的equals覆盖不正确，你会发现删除或者查找操作不正确，但是你明明发现集合中有这样的元素啊，怎么删除不掉，或者查找不出来呢？这时就不容易找原因了，你可能会怀疑是不是equals中的逻辑不正确啊，认真看了几分钟发现逻辑也是OK的，这时你可能就想不到是equals的问题了。

### 注解的总结

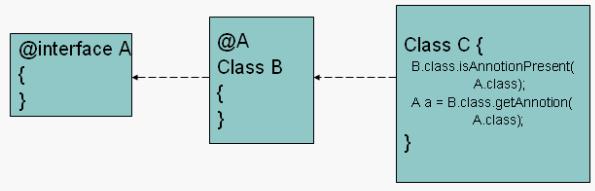
注解相当于一种标记，在程序中加了注解就等于为程序打上了某种标记，没加，则等于没有某种标记，以后，javac编译器、开发工具和其他程序可以用反射来了解你的类及各种元素上有无何种标记，看你有什么标记，就去干相应的事。标记可以加在包、类、方法、方法的参数以及局部变量上。

看java.lang包，可以看到后面提供了最基本的annotaion，共有3个：

* Deprecated
* Override
* SuppressWarning  
  

## 创建注解

注解类 应用了“注解类”的类 对“应用了注解类的类”进行反射操作的类



上面三个图的关系

1. 要使用注解，就要先定义一个或者说开发一个注解类
2. 在需要的类上使用注解类
3. 写一个类检查你这个类是否使用了注解，如果没有检查的类，则使用注解就没有意义了，使用注解就是为了让有一个类可以检查它并做出相应的操作。

实例：

1、创建注解

**public** @**interface** MyAnnotation {  
}

2、使用这个注释：

@MyAnnotation  
**public class** Main {  
  
}

3、编写一个类来检查Main类上什么注解，检查类当然要使用反射了：

这时运行没有打印出任何东西，证明那个if语句返回为false，这是怎么回事呢？把MyAnnotation类上加入这个注解即可：改为如下即可：@Retention(RetentionPolicy.***RUNTIME***)

**public static void** main(String[] args) {  
 **if** (Main.**class**.isAnnotationPresent(MyAnnotation.**class**)) {  
 MyAnnotation annotation = Main.**class**.getAnnotation(MyAnnotation.**class**);  
 System.***out***.println(annotation);  
 }  
}

## Retention注解

Retention是一个元注解，注解的注解称为元注解，信息的信息称为元信息。

Retention用于指定注解保留在什么阶段。

注：class文件中的内容不是字节码，只有经过类加载器验证、处理等加载到内存里的才是字节码。

Retention元注解有三种取值：RetentionPolicy.SOURCE、RetentionPolicy.CLASS、RetentionPolicy.RUNTIME，分别对应java源文件、class文件、内存中的字节码，

在编译java源文件的时候有可能会去掉注解，在把class文件加载到内存中时也有可能会去掉注解，所以想把注解保留到什么阶段就要进行说明，如果没有说明，则默认保留在.class阶段。

@Retention默认值为RetentionPolicy.CLASS

@SuppressWarnings默认值为RetentionPolicy. SOURCE

@Override默认值为RetentionPolicy. SOURCE

@Deprecated默认值为RetentionPolicy. RUNTIME

SuppressWarnings与Override都是给ide用的，ide直接扫描源代码上有没有这些注解而进行处理，如发现有Override，则判断覆盖是否合法，不合法直接提示覆盖错误，并不需要保留在字节码，这样可以节省内存。

而Deprecated注解是保存到运行时的，因为我们会使用到一些别人编译好的类，那时已经是class文件，但是我们在调用这些class中的过时方法时依然有得到过时提示。说明编译器或者ide是扫描了class，把class加载到运行时来判断是否使用了Depreated注解。

通过JDK文档查看这几个注解有描述它们保存在什么阶段。

## Target注解

@Target(ElementType.***METHOD***)  
@Retention(RetentionPolicy.***RUNTIME***)  
**public** @**interface** MyAnnotation {  
 String color() **default "yello"**;  
}

如上面的注解只能使用在方法上，用到其他地方时ide就会报错。Target注解的属性是一个数组，所以我们可以指定多个应用范围，如下：

@Target({ElementType.***METHOD***, ElementType.***TYPE***})

这样，我们的注解就可以同时应用在方法上，也能应用在类上。

如果没有指定Target注解，则默认该注解可以用于任何元素。

允许加载在类上为什么是ElementType.TYPE而不是ElementType.CLASS呢？因为TYPE更精准，Type 是 Java 编程语言中所有类型的公共高级接口。它们包括原始类型、参数化类型、数组类型、类型变量和基本类型、接口、枚举、数组，这些像Class，但又不是Class，但都是一些Java类型。也就是说Type就代表了可以加在Class上，也能加在接口、枚举等类型上面。Class只是Java中的一种类型而已。

通过查看ElementType这个枚举可知注解都可以用到哪些类型上面，如下：  


## 注解的属性

一个注解相当于一个胸牌，如果你胸前贴了胸牌，就是传智播客的学生，否则就不是。如果想区分出是传智播客哪个班的学生，这时候可以为胸牌再增加一个属性来进行区分。加了属性的标记效果为：@MyAnnotation(color=”read”)

### 定义注解的属性

@Retention(RetentionPolicy.***RUNTIME***)  
**public** @**interface** MyAnnotation {  
 String color(); *// 定义了一个color属性，看起来像方法*}

### 应用注解的属性

@MyAnnotation(color = **"red"**)  
**public class** Main {  
}

上面给注解定义了一个color属性，之后使用这个注解时就必须要设置这个color属性，否则ide就会报错

### 获取注解的属性

MyAnnotation annotation = Main.**class**.getAnnotation(MyAnnotation.**class**);  
System.***out***.println(annotation.color());

可以认为@MyAnnotation是MyAnnotaion类的一个实例对象，上面通过Class的getAnnotation方法为获取到实例对象

### 注解属默认省值

@Retention(RetentionPolicy.***RUNTIME***)  
**public** @**interface** MyAnnotation {  
 String color() **default "yello"**;   
}

有了默认值之后，我们在使用这个注解时，就可以不设置这个属性

### 注解的value属性

@Retention(RetentionPolicy.***RUNTIME***)  
**public** @**interface** MyAnnotation {  
 String color() **default "yello"**;  
 String value();  
}

如果在使用注解时只需要设置value属性（比如只有一个value属性，或者有其它属性，但是其它属性都有默认值），则在设置注解的这个value属性时可以省略属性名，直接写上值即可：

@MyAnnotation(**"value属性的值"**)  
**public class** Main {  
   
}

比如之前看到的一些注解就使用了value属性：

* @Retention(RetentionPolicy.RUNTIME) value属性类型为枚举
* @SuppressWarnings("deprecation") value属性类型为String
* @Target({ElementType.METHOD, ElementType.TYPE}) value属性类型为数组

### 注解的高级属性

数组类型的属性：

属性定义：int[] arrayAttr() default{1,2,3};

属性使用：@MyAnnotation(arrayAttr={2,3,4});

如果数组属性中只有一个元素，这里候属性值部分可心省略大括号：@MyAnnotation(arrayAttr=2)，如果属性如是value的话，还可以省略属性名：@MyAnnotation(2)

枚举类型的属性

属性定义：ColorEnum color() default ColorEnum.RED;

属性使用：@MyAnnotation(color=ColorEnum.GREEN)

注解类型的属性(即元注解)

1. 定义元注解，和定义普通注解没什么区别：

**public** @**interface** MetaAnnotation {  
 String value();  
}

1. 在注解上使用元注解：

@Retention(RetentionPolicy.***RUNTIME***)  
**public** @**interface** MyAnnotation {  
 String color() **default "yello"**;  
 MetaAnnotation annotationAttr() **default** @MetaAnnotation(**"aaa"**);  
}

3、在类上使用这个拥有注解属性的注解  
   
 4、可以认为@MetaAnnotation("bbb")是MetaAnnotation类的一个实例对象，获取这个对象的value属性，如下：

@MyAnnotation(annotationAttr = @MetaAnnotation(**"bbb"**))  
**public class** Main {  
}

**public static void** main(String[] args) {  
 MyAnnotation annotation = Main.**class**.getAnnotation(MyAnnotation.**class**);  
 MetaAnnotation metaAnnotation = annotation.annotationAttr();  
 String value = metaAnnotation.value();  
 System.***out***.println(value);  
}

注解的详细语法可以通过看java语言规范了解，即看java的language specification，比如枚举中的属性的类型都可以是哪些，可以查看这个规范中了解，这些可用类型为：基本数据类型、String、枚举、Class、注解，和这些类型对应的数组类型

# 泛型

泛型是jdk1.5的所有新特性中最难深入掌握的部分，不过，我们在实际应用中有需要掌握得那么深入，掌握泛型中一些最基本的内容就差不多了。

ArrayList可心存储各种类型的对象，如：

**public** **static** **void** main(**String**[] args) {

**ArrayList** als = **new** **ArrayList**();

als.add(1);

als.add(3L);

als.add("abc");

**System**.*out*.println(als.get(2));

}

而ArrayList<String> als = new ArrayList<String>();代表ArrayList只能存储String类型的对象。这是在集合中使用泛型。

**在JDK文档中的类名后有“<T>”或“<E>”说明这个类支持泛型**

//在反射中使用泛型

//Constructor<String>指示这个构造函数是String类的构造函数，所以用这个构造函数newInstance的对象就是String对象，不需要再进行转换，代码如下：

**Constructor**<**String**> constructor = **String**.**class**.getConstructor(**StringBuffer**.**class**);

**String** str = constructor.newInstance(**new** **StringBuffer**("ddd"));

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

泛型是提供给javac编译器使用的，可心限定集合中的输入类型，让编译器挡住源程序中的非法输入，编译器编译带类型说明的集合时会去除掉“类型”信息，使程序运行效率不受影响，对于参数化的泛型类型，getClass()方法的返回值和原始类型完全一样。由于编译生成的字节码会去掉泛型的类型信息，只要能跳过编译器，就可心往某个泛型集合中加入其他类型的数据，例如，用反射得到集合，再调用其add方法即可。

在JDK1.5中，你还可以按可以按原来的方式将各种不同类型的数据装到一个集合中，但编译器会报告unchecked警告。

ArrayList<E>类定义和ArrayList<Integer>类引用中涉及如下术语：

整个ArrayList<E>称为泛型类型

ArrayList<E>中的E称为类型变量或类型参数

整个ArrayList<Interger>称为参数化的类型

ArrayList<Integer>中的Integer称为类型参数的实例或实际类型参数

ArrayList<Integer>中的<>读为typeof

ArrayList称为原始类型

参数化类型与原始类型的兼容性：

参数化类型可以引用一个原始类型的对象，编译报告警告，例如，

Collection<String> c = new Vector();

原始类型可以引用一个参数化类型的对象，编译报告警告，例如，

Collection<String> c = new Vector<String>();

参数化类型不考虑类型参数的继承关系：

Vector<String> v = new Vector<Object>();//错误！

Vector<Object> v = new Vector<String>();//也错误！

假设Vector<String> v = new Vector<Object>();可以的话，那么以后从v中取出的对象当作String用，而V实际指向的对象中可以机器翻译任意的类型对象；

假设Vector<Object> v = new Vector<String>();可以的话，那么以后可以向v中加入任意的类型对象，而v实际指向的集合中只能装String类型的对象。

在创建数组实例时，数组的元素不能使用参数化的类型，例如，下面语句有错误：

Vector<Integer> VectorList[] = new Vecotr<Integer>[10]；

思考下面的代码会报错吗？

Vector v1 = new Vector<String>();

Vector<Object> v = v1；

这是不会报错的，因为编译阶段是一句一句编译的，编译第一句的时候不会报错，接着编译第二句的时候，检查到达把一个参数化类型引用变量指向一个原始类型引用变量，这当然是可以的，所以不会报错。

**泛型中的？通配符，表示参数化类型可以为任意类型**

实例：设计一个方法，可以打印任意参数化类型的集体对象。

Public static void main(String args[])

{

Collection<**String**> collection = **new** **ArrayList**();

collection.add("abc");

*printCollection*(collection);

}

**public** **static** **void** printCollection(Collection<?> collection)

{

//collection.add(1);这句代码会报错，比如传进来的是一个参数化类型为String的集体，这是不合理的。

collection.size();//这句代码可以通过

for(Object obj: collection) //打印集合中的每一个对象

{

System.out.println(obj);

}

}

怎么知道collection.size();这个方法是通用的呢？查JDA文档，只要方法的参数里没有<E>即为通用的，即这个方法是跟参数化的类型无关的

**总结：**使用？通配符可以引用各种参数化的类型，？通配符定义的变量主要用作引用，可以调用与参数化无关的方法，不能调用与参数化有关的方法，在查JDK文档时，在参数里如果有<E>则表示这个方法与类的参数化有关。

**泛型中的？通配符的扩展**

* **限定通配符的上边界：**
  + 正确：Vector<? extends Number> x = new Vector<Integer>();//限定为Number或其子类
  + 错误：Vector<? extends Number> x = new Vector<String>();
* **限定通配符的下边界：**
  + 正确：Vector<? super Integer> x = new Vector<Number>();//限定为Number或父类
  + 正确：Vector<? super Integer> x = new Vector<Byte>();
* **提示：限定通配符总是包括自己。**

**泛型集合的综合案例**

能写出下面的实例即代表掌握了Java的泛型集合类：

**package** cn.itcast.day2;

**import** java.util.\*;

**public** **class** **GegericSimple** {

**public** **static** **void** main(**String**[] args) {

Map<**String**,**Integer**> maps = **new** **HashMap**();

maps.put("张三", 23);

maps.put("李四", 24);

maps.put("王五", 25);

//使用迭代打印出Map映射中的值对，但是Map没有实现Iterator接口，所以不能进行迭代，

//可以把这个Map转换成Set集合，Set实现了Iterator接口

Set<Map.Entry<**String**,**Integer**>> entrySet = maps.entrySet();

**for**(Map.Entry<**String**,**Integer**> entry: entrySet)

{

**System**.*out*.println(entry.getKey() + ":" + entry.getValue());

}

}

}

键-值对在JSP页面中也经常要对Set或Map集合进行迭代：

<c:forEach items=”${map}” var=”entry”>

${entry.key}:${entry.valu}

</c:froEach>

**由C++的模板函数引入自定义泛型**

**如下函数的结构很相似，仅类型不同：**

int add(int x , int y) { return x+y ; }

float add(float x , float y) { return x+y ; }

double add(double x , double y) { return x+y ; }

**C++用模板函数解决，只写一个通用的方法，它可适应各种类型，示意代码如下：**

Template<class T>

T add(T x, T y) { return (T) ( x +y ) }

**在Java中自定义泛型：**

**public** **class** **GegericSimple** {

**private** **static** <T> T add(T x,T y)

{

**return** **null**;

}

**public** **static** **void** main(**String**[] args) {

**Integer** i = *add*(2,3);//参数都为Integer，返回Interger

**Number** n = *add*(2,3.0);//参数都为Number，返回Number

**Object** o = *add*(2,"a");//参数都为Object，返回Object

}

}

自定义泛型应用实例：交换任意类型的数组中任意两个元素的值

**public** **static** **void** main(**String**[] args) {

**String**[] strs = **new** **String**[] {"a","b","c"};

**int**[] nums = **new** **int**[] {1,2,3};

*swap*(strs,1,2);

//swap(nums,1,2);//这句代码会报错，T类型的参数只能是引用变量型的，不能为基本类型

}

**public** **static** <T> **void** swap(T[] t,**int** i,**int** j)

{

T temp = t[i];

t[j] = t[i];

t[i] = temp;

}

Java中的泛型类型（或者泛型）类似于C++中的模板。但是这种相似性仅限于表面，Java语言中的泛型基本上完全是在编译器中实现，用于编译器执行类型检查和类型推断，然后生成普通的非泛型的字节码，这种实现技术称为 擦除（erasure）（编译器使用泛型类型信息保证类型安全，然后在生成字节码之前将其清除）。这是因为扩展虚拟机指令集来支持泛型被认为是无法接受的，这会为Java厂商升级其JVM造成难以逾越的障碍。所以，Java的泛型采用了可以完全在编译器中实现的擦除方法。

交换数组中的两个元素的位置的泛型方法语法定义如下：

Private static <T extends Exception> sayHello() throws T

{

Try{}

catch(Exception e)//这里用(T e)就是错误的

{ throw (T)e;}

}

用于放置泛型的类型参数的尖括号应出现在方法的其他所有修饰符之后和在方法的返回类型之前，也就是紧邻返回值之前。按照惯例，类型参数通常用个单个大写字母表示。

只有引用类型才能作为泛型方法的实际参数，swap(new int[3],3.5);语句会报告编译错误。

除了在应用泛型时可以使用extends限定符，在定义泛型时也可以使用extends限定符，例如，Class.getAnnotation()方法的定义，并且可以用&来指定多个边界，如<V extends Serializable & cloneable> void method(){}

普通方法、构造方法和静态方法中都可以使用泛型。编译器也不允许创建类型变量的数组。

也可以用类型变量表示异常，称为参数化的异常，可以用于方法throws列表中，但是不能用于catch子句中。如：

在泛型中可以同时有多个类型参数，在定义它们的尖括号中用逗号分隔，例如：

public static <K,V> getValue(K key){return map.get(key);}

**泛型方法的练习题**

* 编写一个泛型方法，自动将Object类型的对象转的成其他类型。

**public** **static** **void** main(**String**[] args) {

**Object** obj = **new** **String**("abc");

**String** str = *autoConvert*(obj);

}

**private** **static** <T> T autoConvert(**Object** obj)

{

**return** (T)obj;

}

* 定义一个方法，可以将任意类型的数组中的所有元素填充为相应类型的某个对象。

**public** **static** **void** main(**String**[] args) {

**String**[] strs2 = **new** **String**[5];

*fillArray*(strs2,"d");

**for**(**int** i = 0;i<strs2.length;i++)

**System**.*out*.println(strs2[i]);

}

**private** **static** <T> **void** fillArray(T[] arr,T obj) {

**for**(**int** i = 0 ;i < arr.length ; i++)

arr[i] = obj;

}

* 采用自定泛型的方式打印出任意参数化类型的集合中的所有内容。
  + 在这种情况下，前面的通配符方案要比泛型方法更有效，当一个类型变量用来表达两个参数之间或者参数和返回值之间的关系时，即同一个类型变量在方法签名的两处被使用，或者类型变量在方法体代码中也被使用而不仅在签名的时候使用，才需要使用泛型方法。

Public static void main(String args[])

{

Collection<**String**> collection = **new** **ArrayList**();

collection.add("abc");

*printCollection*(collection);

}

**public** **static** <T>**void** printCollection(Collection<T> collection,T obj)

{

collection.add(obj);//这句代码不会报错，这就是与用？的区别

for(Object obj: collection) //打印集合中的每一个对象

{

System.out.println(obj);

}

}

* 定义一个方法，把任意参数类型的集合中的数据安全的复制到相应类型的数组中。

如果使用如下形式定义：

static void copy(Collection a,Object[] b);

则有可能出现A类型的数据复制进B类型的数组中的情况。

使用泛型方法的定义形式为：

static <T> void copy (Collection<T> dest,T[] src);

static <T> void copy2 (T[] dest , T[] src)

在使用这两个方法时的区别：

Copy(new Vector<String>(),new String[10]);//不报错

Copy(new Vector<Date>() ,new String[10]);//报错

Copy2(new Date[10],new String[10]);//不报错

* 定义一个方法，把任意参数类型的一个数组中的数据安全地复制到相应类型的另一个数组中。

**类型参数的类型推断**

* 编译器判断泛型方法的实际类型参数的过程称为类型推断，类型推断是相对于知觉推断的，其实方法是一种复杂的过程。
* 根据调用泛型方法时实际传递的参数类型或返回值的类型来推断，具体规则如下：
  + 当某个类型变量只在整个参数列表中的所有参数和返回值中的一处被应用了，那么根据调用方法时该处的实际应用类型来确定，这很容易凭着感觉推断出来，即直接根据调用方法时传递的参数类型或返回值来决定泛型参数的类型，例如：

swap(new String[3],3,4)🡪static <E> void swap(E[] a, int i,int j)

* 当某个类型变量在整个参数列表中的所有参数和返回值中的多处被应用了，如果调用方法时这多处的实际应用类型都对应同一种类型来确定，这很容易凭着感觉推断出来，例如：

Add(3,5) 🡪static <T> T add(T a,T b)

* 当某个类型变量在整个参数列表中的所有参数和返回值中的多处被应用了，如果调用方法时这多处的实际应用类型对应到了不同的类型，且没有使用返回值，这时候取多个参数中的最大交集类型，例如，下面语句实际对应的类型就是Number了，编译没问题，只是运行时出问题：

fill(new Integer[3],3.5f) 🡪 static <T> void fill(T[] a,T b)

* 当某个类型变量在整个参数列表中的所有参数和返回值中的多处被应用了，如果调用方法时这多处的实际应用类型对应到了不同的类型，并且使用返回值，这时候优先考虑返回奉珠类型，例如，下面语句实际对应的类型就是Integer了，编译将报告错误，将变量x的类型改为float，对比eclipse报告的错误提示，接着再将变量x类型改为Number，则没有了错误：

int x = (3,3.5f) 🡪 static <T> T add(T a,Tb)

* 参数类型的类型推断具有传递性，下面第一种情况推断实际参数类型为Object，编译没有问题，而第二种情况则根据参数化的Vector类实例将类型变量直接确定为String类型，编译将出现问题：

Copy(new Integer[5],new String[5]) 🡪 static <T> void copy(T[] a,T[] b);

**定义泛型类型**

**public** **class** **GenericDao** {

**public** <T> **void** add(T t) {

}

**public** <T> T findById(**int** id)

{

**return** **null**;

}

**public** **static** **void** main(**String** args[])

{

**GenericDao** gd = **new** **GenericDao**();

gd.add(**new** **ReflectPoint**(3,5));//这里添加的是ReflectPoint

**String** str = gd.findById(1);//这里查的的却是String，显然是不对，但这里不会报错的

}

}

如果类的实例对象中的多处都要用到同一个泛型参数，即这些地方引用的泛型类型要保持同一个实际类型时，这时候就要采用泛型类型的方式进行定义，也就是类级别的泛型，语法格式如下：

//dao 是data access object的缩写，这个类的功能主要是crud (create、read、update、delete的缩写)，即增加、查询、更新、删除数据。

**public** **class** **GenericDao**<T> {

**public** **void** add(T t) { //增

}

**public** T findById(**int** id) //查

{

**return** **null**;

}

**public** T findByUserName(**String** name)//查

{

**return** **null**;

}

**public** **void** delete(T obj)//删

{

}

**public** **void** delete(**int** id)//删

{

}

**public** **void** update(T obj)//改

{

}

**public** Set<T> findByConditions(**String** where)

{

**return** **null**;

}

**public** **static** **void** main(**String** args[])

{

**GenericDao**<**ReflectPoint**> gd = **new** **GenericDao**<**ReflectPoint**>();

gd.add(**new** **ReflectPoint**(3,5));

**ReflectPoint** p = gd.findById(1);

}

}

类级别的泛型是根据引用该类名时指定的类型信息来参数化类型变量的，例如，下面两种方式都可以：

GenericDao<String> dao = null;

new GenericDao<String>();

注意：

在对泛型类型进行参数化时，类型参数的实例必须是引用类型，不能是基本类型。

当一个变量被声明为泛型时，只能被实例变量和方法调用（还有内嵌类型），而不能被静态变量和静态方法调用。因为静态成员是被所有参数化的类所共享的，所以静态成员不一叶叶类级别的类型参数。

问题：类中只有一个方法需要使用泛型，是使用类级别的泛型，还是使用方法级别的泛型？

答：类级别。

下面两个方法是否属于重载：

**public** **static** **void** applyVector(**Vector**<**Date**> d){}

**public** **static** **void** applyVector(**Vector**<**String**> d){}

答：不属于，在编译成字节码后，泛型的参数类型会被擦除掉，所以编译后这两个方法在字节码里面是一模一样的，所以，如果同时编写这两个方法编译时会报错。

**通过反射获取方法参数的参数化类型：**

**public** **static** **void** applyVector(**Vector**<**Date**> d){}

**public** **static** **void** main(**String** args[]){

**Method** genericMethod = **GenericTest**.**class**.getMethod("applyVector", **Vector**.**class**);

Type[] genericType = genericMethod.getGenericParameterTypes();//获取泛型参数类型

ParameterizedType pType = (ParameterizedType)genericType[0];//强制转换为“参数化类型”类的类型

Type[] actualTypes = pType.getActualTypeArguments();//获取参数的实际类型

Type rawType = pType.getRawType();//获取真实类型

**System**.*out*.println(actualTypes[0]);//返回Date

**System**.*out*.println(rawType);//返回Vector

}

**类加载器**

类加载器即指加载类的工具

Java虚拟机中可以安装多个类加载器，系统默认三个主要类加载器，每个类负责加载我写位置的类：

BootStrap，ExtClassLoader，;AppClassLoader

类加载器本身也是一个Java类，因为其他是Java类的类加载器本身也要被类加载器加载，显然必须有第一个类加载器不是Java类，这正是BootSrap,它是用C++写的。

Java虚拟机中的所有类装载器采用具有父子关系的树形结构进行组织，在实例化每个类装载器对象时，需要为其指定一个父级类装载器对象或者默认采用系统类装载器为其父级类加载

**实例1：**

**public** **class** **classLoaderTest** {

**public** **static** **void** main(**String**[] args) {

**System**.*out*.println(**classLoaderTest**.**class**.getClassLoader().getClass().getName());//结果为AppClassLoader

//System.out.println(System.class.getClassLoader().getClass().getName());//这句代码为报错因为：

**System**.*out*.println(**System**.**class**.getClassLoader());//结果为null

}

}

第二个输出语句会抛出NullPointerException，这个类存放的位置不同**classLoaderTest类**，而且这个System类是由BootStrap加载器加载的，这个加载器不是一个Java类，所以当然不能获得它的名字了。

先看一下上面的文件所有项目使用的是哪一版本的JDK，然后把这个文件导出为JAR，导出到对应的JDK版本的jdk1.7\jre\lib\ext\目录下，步骤：右击上面的文件在项目中的文件名🡪导出—>Java🡪JAR文件🡪在Java文件处输入：C:\Java\jdk1.6.0\jre\lib\ext\itcast.jar🡪完成🡪再次运行这个类，会发现每一个输出语句为：ExtClassLoader

**实例2：**验证BootStrap，ExtClassLoader，;AppClassLoader的父子关系：

**public** **static** **void** main(**String**[] args) {

**ClassLoader** loader = **classLoaderTest**.**class**.getClassLoader();

**while**(loader != **null**)

{

**System**.*out*.println(loader.getClass().getName());

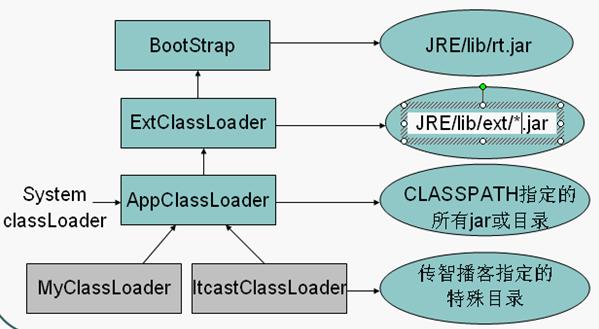
loader = loader.getParent();

}

**System**.*out*.println(loader);//输出null

}

**类加载器之间的父子关系和管辖范围图**

****

**类加载器的委托机制**

每个ClassLoader本身只能分别加载特定位置和目录中的类，但它们可以委托其他的类装载器去加载类，这就是类加载器的委托模式。类装载器一级级委托到BootSrap类加载器，当BootStrap无法加载当前所要加载的类时，然后才一级级回退到子孙类装载器去进行真正的加载。当回退到最初的类装载器时，如果它自己也不能完成类的装载，那就应该报告ClassNotFoundException异常了。

当Java虚拟机要加载一个类时，到底派出哪个类加载器去加载呢？

首先当前线程的类加载器去加载线程中的每一个类。

如果类A中引用了类B，Java虚拟机使用加载类A的类装载器来加载类B。还可以直接调用ClassLoader.loadClass()方法来指定某个类加载器去加载某个类。

每个类加载器加载类时，又先委托给其上级类加载器。

当所有祖宗类加载器没有加载到类，回到发起者类加载器，还加载不了，则抛ClassNotFoundException，不是再去找发起者类加载器的儿子，因为没有getChild方法，即使有，那有多个儿子，找哪一个呢？

**自定义加载器**

必须继承ClassLoad类

模板方法设计模式

实例：设计一个加密程序，用来加密.class文件：

**package** cn.itcast.day2;

**import** java.io.\*;

**public** **class** **MyClassLoader** {

**public** **static** **void** main(**String**[] args) **throws** Exception {

**String** srcPath = args[0];

**String** destDir = args[1];

**String** destFileName = srcPath.substring(

srcPath.lastIndexOf('\\') + 1);

**String** destPath = destDir + "\\" + destFileName;

**InputStream** in = **new** **FileInputStream**(srcPath);

**OutputStream** out = **new** **FileOutputStream**(destPath);

*cyhper*(in,out);

}

**public** **static** **void** cyhper(**InputStream** in,**OutputStream** out) **throws** Exception

{

**int** b = -1;

**while**((b = in.read()) != -1)

{

out.write(b^0xff);//与0xff异或后，则0变1，1变0；

}

in.close();

out.close();

}

}

运行这个类时给类传递一个包含.class文件的路径，和一个要保存到新位置的路径参数。接着把新生成的.class文件复制到原来的.class，接着在程序中调用这个类就会报错。

把加密的类解密：

//这个类用来加密

**import** java.util.Date;

**public** **class** **ClassLoaderAttachment** **extends** **Date** {

@Override

**public** **String** toString() {

**return** "hello,world!";

}

}

//自定义的类加载器

**import** java.io.\*;

**import** java.util.Date;

**public** **class** **MyClassLoader** **extends** **ClassLoader** {

**public** **static** **void** main(**String**[] args) **throws** Exception {

**String** srcPath = args[0];

**String** destDir = args[1];

**String** destFileName = srcPath.substring(

srcPath.lastIndexOf('\\') + 1);

**String** destPath = destDir + "\\" + destFileName;

**InputStream** in = **new** **FileInputStream**(srcPath);

**OutputStream** out = **new** **FileOutputStream**(destPath);

**System**.*out*.println(**new** **ClassLoaderAttachment**());

*cyhper*(in,out);

**MyClassLoader** myClassLoader =

**new** **MyClassLoader**("D:\\workspace8\\javaenhance\\itcastlib");

**Class** clazz = myClassLoader.findClass("ClassLoaderAttachment");

**Date** classLoaderAttachment = (**Date**)clazz.newInstance();//这里声明的类型不能为ClassLoaderAttachment，

**System**.*out*.println(classLoaderAttachment); //因为此时的这个类是一个乱码文件

}

**private** **String** classDir;

@Override

**protected** **Class**<?> findClass(**String** name) **throws** ClassNotFoundException {

**String** classPath = classDir + "\\" + name + ".class";

**try**

{

**InputStream** in = **new** **FileInputStream**(classPath);

**ByteArrayOutputStream** out = **new** **ByteArrayOutputStream**();

**MyClassLoader**.*cyhper*(in,out);//解密的类的字节码保存在out

**byte**[] bytes = out.toByteArray();//把out里的字节码拿出来

in.close();

out.close();

//下面方法将bytes数组 转换为Class类的实例。

**return** defineClass(bytes, 0,bytes.length);

} **catch**(**Exception** e)

{

e.printStackTrace();

}

**return** **super**.findClass(name);

}

**public** MyClassLoader(**String** classDir) {

**this**.classDir = classDir;

}

**public** **static** **void** cyhper(**InputStream** in,**OutputStream** out) **throws** Exception

{

**int** b = -1;

**while**((b = in.read()) != -1)

{

out.write(b^0xff);

}

in.close();

out.close();

}

}

一个类加载器的高级问题分析

编写一个能打印出自己的类加载器名称和当前类加载器的父子结构关系链的MyServlet，正常发布后，看到打印结果为WebAppClassLoader。

把MyServlet.class文件打jar包，放到ext目录中（tomcat服务器用哪个jdk就放到到个jdk的ext目录中），重启tomcat，发现找不到HttpServlet的错误。

把tomcat服务器中lib目录中的servlet.jar也放到jdk的ext目录中，问题解决了，打印的结果是ExtClassLoader。

父级类加载器加载的类无法引用只能被子级类加载器加载的类，如下图中子类WebAppclassLoader能加载HttpServlet，而父类ExtClassLoader却加载不了，原理如下图：

