**一、RTTI的概念以及Class对象作用**

　　　RTTI（Run-Time Type Identification）运行时类型识别，对于这个词一直是C++中的概念，至于Java中出现RTTI的说法则是源于《Thinking in java》一书，其作用是在运行时识别一个对象的类型和类的信息。

　　这里分为两种：

　　1、传统的“RTTI”，它假定我们在编译期已经知道了所有类型（在没有反射机制创建和使用类对象时，一般都是编译期已确定其类型，如new对象时该类必须已定义好）；

　　2、反射机制，它允许我们在运行时发现和使用类型的信息。

　　在java中用来表示运行时类型信息对应类就是Class类，Class类也是一个实实在在的类，存在于JDK的java.lang包中，其部分源码如下：

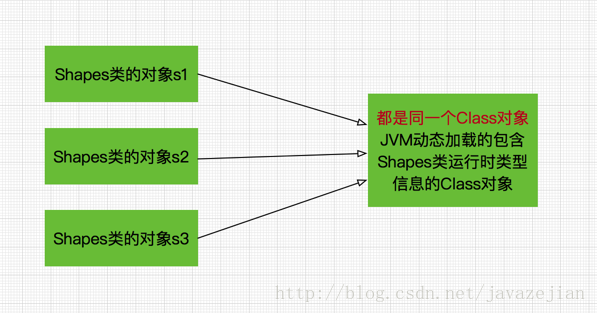
|  |
| --- |
| 1 public final class Class<T> implements java.io.Serializable,GenericDeclaration,Type, AnnotatedElement {  2 private static final int ANNOTATION= 0x00002000;  3 private static final int ENUM = 0x00004000;  4 private static final int SYNTHETIC = 0x00001000;  5  6 private static native void registerNatives();  7 static {  8 registerNatives();  9 }  10  11 /\*  12 \* Private constructor. Only the Java Virtual Machine creates Class objects.（私有构造，只能由JVM创建该类）  13 \* This constructor is not used and prevents the default constructor being  14 \* generated.  15 \*/  16 private Class(ClassLoader loader) {  17 // Initialize final field for classLoader. The initialization value of non-null  18 // prevents future JIT optimizations from assuming this final field is null.  19 classLoader = loader;  20 } |

Class类被创建后的对象就是Class对象，注意，Class对象表示的是自己手动编写类的类型信息，比如创建一个Shapes类，那么，JVM就会创建一个Shapes对应Class类的Class对象，该Class对象保存了Shapes类相关的类型信息。

　　实际上在java中每个类都有一个Class对象，每当我们编写并且编译一个新创建的类就或产生一个对应Class对象并且这个Class对象会被保存在同名.class文件里（编译后的字节码文件保存的就是Class对象），那么为什么需要这样一个Class对象呢？

　　是这样的，当我们new一个新对象或者引用静态成员变量时，Java虚拟机中的类加载器子系统会将对应Class对象加载到JVM中，然后JVM再根据这个类型信息相关的Class对象创建我们需要实例对象或者提供静态变量的引用值。

　　需要特别注意的是，手动编写的每个Class类，无论创建多少个实例对象，在JVM中都只有一个Class对象，即在内存中每个类有且只有一个相对应的Class对象，挺拗口，通过下图理解内存中简易现象图：



到这里我们也就可以得出如下几点信息：

　　　　1、Class类也是类的一种，与class关键字是不一样的。

　　　　2、手动编写的类被编译后会产生一个Class对象，其表示的是创建的类的类型信息，而且对象保存在同名.class的文件中（字节码文件），比如创建一个Shapes类，编译shapes类后就会创建其包含Shapes类相关类型信息的Class对象，并保存在Shapes.class字节码文件中。

　　　　3、每个通过关键字class标识的类，在内存中有且只有一个与之对应的Class对象来描述其类型信息，无论创建多少个实例对象，其依据的都是用一个Class对象。

　　　　4、Class类只存私有构造函数，因此对应Class对象只能有JVM创建和加载

　　　　5、Class类的对象作用是运行时提供或获得某个对象的类型信息，这点对于反射技术很重要。

**二、Class对象的加载及其获取方式**

**1、Class对象的加载**

　　所有的类都是在对其第一次使用时动态加载到JVM中，当程序创建第一个对类的静态成员引用时，就会加载这个被使用的类（实际上加载的就是这个类的字节码文件），包含new（构造函数也是类的静态方法）

　　java程序在它们开始运行之前并非完全加载到内存的，类加载器首先检查对象是否已被加载（类的实例对象创建时依据class对象中类型信息完成的），如果没有，默认的类加载器就会先根据类名查找.class文件，在这个类的字节码文件被加载时，它们必须接受相关验证，以确保其没有被破坏且不包含不良Java代码（这是java的安全机制检测），完全没有问题后会被动态加载到内存中，此时相当于Class对象也就被载入了内存了，同时也就可以被用来创建这个类的所有实例对象。

|  |
| --- |
| 1 class Candy {  2 static { System.out.println("Loading Candy"); }  3 }  4  5 class Gum {  6 static { System.out.println("Loading Gum"); }  7 }  8  9 class Cookie {  10 static { System.out.println("Loading Cookie"); }  11 }  12  13 public class SweetShop {  14 public static void print(Object obj) {  15 System.out.println(obj);  16 }  17 public static void main(String[] args) {  18 print("inside main");  19 new Candy();  20 print("After creating Candy");  21 try {  22 Class.forName("com.wangjun.Gum");  23 } catch(ClassNotFoundException e) {  24 print("Couldn't find Gum");  25 }  26 print("After Class.forName(\"com.wangjun.Gum\")");  27 new Cookie();  28 print("After creating Cookie");  29 }  30 } |

在上述代码中，每个类Candy、Gun、Cookie都存在一个static语句，这个语句会在类第一次被加载时执行，这个语句的作用就是告诉我们该类在什么时候被加载，执行结果是：

|  |
| --- |
| 1 inside main  2  3 Loading Candy  4  5 After creating Candy  6  7 Loading Gum  8  9 After Class.forName("com.wangjun.Gum")  10  11 Loading Cookie  12  13 After creating Cookie |

　从结果来看，new一个Candy对象和Cookie对象，构造函数将被调用，属于静态方法的引用，Candy类的Class对象和Cookie的Class对象肯定会被加载，毕竟Candy实例对象的创建依据其Class对象。比较有意思的是

|  |
| --- |
| Class.forName("com.wangjun.Gum"); |

其中forName方法是Class类的一个static成员方法，记住所有的Class对象都源于这个Class类，因此Class类中定义的方法将适应所有Class对象。这里通过forName方法，我们可以获取到Gum类对应的Class对象引用。从打印结果来看，调用forName方法将会导致Gum类被加载（前提是Gum类从来没有被加载过）。

**2、Class.forName方法**

　　通过上述案例，我们也就知道Class.forName（）方法的调用将会返回一个对应类的Class对象，因此如果我们想要获取一个类的运行时类型信息并加以使用时，可以调用Class.forName()方法获取Class对象的引用，这样做的好处是无需通过持有该类的实例对象引用而去获取Class对象，如下的第二种方式是通过一个实例对象获取一个类的class对象，其中的getClass（）是从顶级类Object继承而来的，它将犯规表示该对象的实际类型的Class对象引用。

|  |
| --- |
| 1 public static void main(String[] args) {  2  3 try{  4 //通过Class.forName获取Gum类的Class对象  5 Class clazz=Class.forName("com.wangjun.Gum");  6 System.out.println("forName=clazz:"+clazz.getName());  7 }catch (ClassNotFoundException e){  8 e.printStackTrace();  9 }  10  11 //通过实例对象获取Gum的Class对象  12 Gum gum = new Gum();  13 Class clazz2=gum.getClass();  14 System.out.println("new=clazz2:"+clazz2.getName());  15  16 } |

注意调用forName方法时需要补货一个名称为ClassNotFoundException的异常，因为forName方法在编译器是无法检测到其传递的字符串对应的类是否是存在的，只能在程序运行时进行检查，如果不存在就会抛出ClassNotFoundException异常。

**3、Class字面常量**

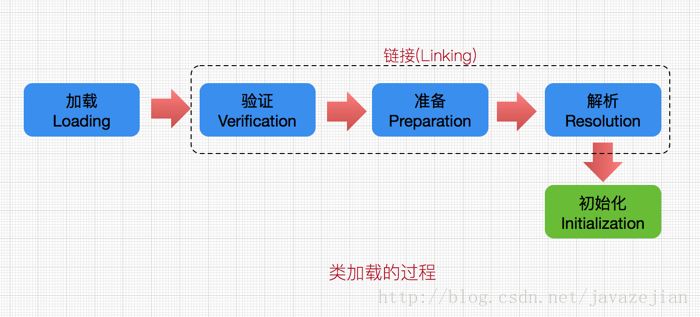
　　在java中存在另一种方式来生成Class对象的引用，它就是Class字面常量，如下：

|  |
| --- |
| 1 //字面常量的方式获取Class对象  2 Class clazz = Gum.class; |

这种方式相对前面两种方法更加简单，更安全。因为它在编译器就会受到编译器的检查同时由于无需调用forName方法效率也会更高，因为通过字面量的方法获取Class对象的引用不会自动初始化该类。更加有趣的是字面常量的获取Class对象引用方式不仅可以应用于普通的类，也可以应用用接口，数组以及基本数据类型，这点在反射技术应用传递参数时很有帮助，关于反射技术稍后会分析，由于基本数据类型还有对应的基本包装类型，其包装类型有一个标准字段TYPE，而这个TYPE就是一个引用，指向基本数据类型的Class对象，其等价转换如下，一般情况下更倾向使用.class的形式，这样可以保持与普通类的形式统一。

|  |
| --- |
| 1 boolean.class = Boolean.TYPE;  2 char.class = Character.TYPE;  3 byte.class = Byte.TYPE;  4 short.class = Short.TYPE;  5 int.class = Integer.TYPE;  6 long.class = Long.TYPE;  7 float.class = Float.TYPE;  8 double.class = Double.TYPE;  9 void.class = Void.TYPE; |

前面提到过，使用字面常量的方式获取Class对象的引用不会触发类的初始化，这里我们可能需要简单了解一下类加载的过程，如下：



1、加载：类加载过程的一个阶段：通过一个类的完全限定查找此类字节码文件，并利用字节码文件创建一个Class对象

2、链接：验证字节码的安全性和完整性，准备阶段正式为静态域分配存储空间，注意此时只是分配静态成员变量的存储空间，不包含实例成员变量，如果必要的话，解析这个类创建的对其他类的所有引用。

3、初始化：类加载最后阶段，若该类具有超类，则对其进行初始化，执行静态初始化器和静态初始化成员变量。

　　由此可知，我们获取字面常量的Class引用时，触发的应该是加载阶段，因为在这个阶段Class对象已创建完成，获取其引用并不困难，而无需触发类的最后阶段初始化。下面通过小例子来验证这个过程：

|  |
| --- |
| 1 import java.util.\*;  2  3 class Initable {  4 //编译期静态常量  5 static final int staticFinal = 47;  6 //非编期静态常量  7 static final int staticFinal2 =  8 ClassInitialization.rand.nextInt(1000);  9 static {  10 System.out.println("Initializing Initable");  11 }  12 }  13  14 class Initable2 {  15 //静态成员变量  16 static int staticNonFinal = 147;  17 static {  18 System.out.println("Initializing Initable2");  19 }  20 }  21  22 class Initable3 {  23 //静态成员变量  24 static int staticNonFinal = 74;  25 static {  26 System.out.println("Initializing Initable3");  27 }  28 }  29  30 public class ClassInitialization {  31 public static Random rand = new Random(47);  32 public static void main(String[] args) throws Exception {  33 //字面常量获取方式获取Class对象  34 Class initable = Initable.class;  35 System.out.println("After creating Initable ref");  36 //不触发类初始化  37 System.out.println(Initable.staticFinal);  38 //会触发类初始化  39 System.out.println(Initable.staticFinal2);  40 //会触发类初始化  41 System.out.println(Initable2.staticNonFinal);  42 //forName方法获取Class对象  43 Class initable3 = Class.forName("Initable3");  44 System.out.println("After creating Initable3 ref");  45 System.out.println(Initable3.staticNonFinal);  46 }  47 } |

执行结果：

|  |
| --- |
| 1 After creating Initable ref  2 47  3 Initializing Initable  4 258  5 Initializing Initable2  6 147  7 Initializing Initable3  8 After creating Initable3 ref  9 74 |

从输出结果来看，可以发现  
　　1.通过字面常量获取方式获取Initable类的Class对象并没有触发Initable类的初始化，这点也验证了前面的分析；

　　2.同时发现调用Initable.static final变量时也没有触发初始化，这是因为static final属于编译期静态常量，在编译阶段通过常量传播优化的方式将Initable类的常量static final存储到了一个称为NotInitialization类的常量池中，在以后对Initable类常量static final的引用实际都转化为对NotInitialization类对自身常量池的引用，所以在编译期后，对编译期常量的引用都将在NotInitialization类的常量池获取，这也就是引用编译期静态常量不会触发Initable类初始化的重要原因。但在之后调用了Initable.staticFinal2变量后就触发了Initable类的初始化，注意staticFinal2虽然被static和final修饰，但其值在编译期并不能确定，因此staticFinal2并不是编译期常量，使用该变量必须先初始化Initable类。

　　3.Initable2和Initable3类中都是静态成员变量并非编译期常量，引用都会触发初始化。至于forName方法获取Class对象，肯定会触发初始化，这点在前面已分析过。

到这几种获取Class对象的方式也都分析完，ok~,到此这里可以得出小结论：

获取Class对象引用的方式3种，通过继承自Object类的getClass方法，Class类的静态方法forName以及字面常量的方式”.class”。

　　其中实例类的getClass方法和Class类的静态方法forName都将会触发类的初始化阶段，而字面常量获取Class对象的方式则不会触发初始化。

　　初始化是类加载的最后一个阶段，也就是说完成这个阶段后类也就加载到内存中(Class对象在加载阶段已被创建)，此时可以对类进行各种必要的操作了（如new对象，调用静态成员等），注意在这个阶段，才真正开始执行类中定义的Java程序代码或者字节码。

　　关于类加载的初始化阶段，在虚拟机规范严格规定了有且只有5种场景必须对类进行初始化：

　　使用new关键字实例化对象时、读取或者设置一个类的静态字段(不包含编译期常量)以及调用静态方法的时候，必须触发类加载的初始化过程(类加载过程最终阶段)。

　　使用反射包(java.lang.reflect)的方法对类进行反射调用时，如果类还没有被初始化，则需先进行初始化，这点对反射很重要。

　　当初始化一个类的时候，如果其父类还没进行初始化则需先触发其父类的初始化。

　　当Java虚拟机启动时，用户需要指定一个要执行的主类(包含main方法的类)，虚拟机会先初始化这个主类

　　当使用JDK 1.7 的动态语言支持时，如果一个java.lang.invoke.MethodHandle 实例最后解析结果为REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，并且这个方法句柄对应类没有初始化时，必须触发其初始化(这点看不懂就算了，这是1.7的新增的动态语言支持，其关键特征是它的类型检查的主体过程是在运行期而不是编译期进行的，这是一个比较大点的话题，这里暂且打住)

# 二、理解泛化的Class对象引用

　　由于Class的引用总数指向某个类的Class对象，利用Class对象可以创建实例类，这也就足以说明Class对象的引用指向的对象确切的类型。在Java SE5引入泛型后，使用我们可以利用泛型来表示Class对象更具体的类型，即使在运行期间会被擦除，但编译期足以确保我们使用正确的对象类型。如下：

|  |
| --- |
| public class ClazzDemo {    public static void main(String[] args){  //没有泛型  Class intClass = int.class;    //带泛型的Class对象  Class<Integer> integerClass = int.class;    integerClass = Integer.class;    //没有泛型的约束,可以随意赋值  intClass= double.class;    //编译期错误,无法编译通过  //integerClass = double.class  }  } |

从代码可以看出，声明普通的Class对象，在编译器并不会检查Class对象的确切类型是否符合要求，如果存在错误只有在运行时才得以暴露出来。但是通过泛型声明指明类型的Class对象，编译器在编译期将对带泛型的类进行额外的类型检查，确保在编译期就能保证类型的正确性，实际上Integer.class就是一个Class<Integer>类的对象。面对下述语句，确实可能令人困惑，但该语句确实是无法编译通过的。

|  |
| --- |
| //编译无法通过  Class<Number> numberClass=Integer.class; |

我们或许会想Integer不就是Number的子类吗？然而事实并非这般简单，毕竟Integer的Class对象并非Number的Class对象的子类，前面提到过，所有的Class对象都只来源于Class类，看来事实确实如此。当然我们可以利用通配符“?”来解决问题：

|  |
| --- |
| Class<?> intClass = int.class;  intClass = double.class; |

这样的语句并没有什么问题，毕竟通配符指明所有类型都适用，那么为什么不直接使用Class还要使用Class<?>呢？这样做的好处是告诉编译器，我们是确实是采用任意类型的泛型，而非忘记使用泛型约束，因此Class<?>总是优于直接使用Class，至少前者在编译器检查时不会产生警告信息。当然我们还可以使用extends关键字告诉编译器接收某个类型的子类，如解决前面Number与Integer的问题：

|  |
| --- |
| 1 //编译通过！  2 Class<? extends Number> clazz = Integer.class;  3 //赋予其他类型  4 clazz = double.class;  5 clazz = Number.class; |

上述的代码是行得通的，extends关键字的作用是告诉编译器，只要是Number的子类都可以赋值。这点与前面直接使用Class<Number>是不一样的。实际上，应该时刻记住向Class引用添加泛型约束仅仅是为了提供编译期类型的检查从而避免将错误延续到运行时期。

# 三、关于类型转换的问题

　　在许多需要强制类型转换的场景，我们更多的做法是直接强制转换类型：

|  |
| --- |
| 1 public class ClassCast {  2  3 public void cast(){  4  5 Animal animal= new Dog();  6 //强制转换  7 Dog dog = (Dog) animal;  8 }  9 }  10  11 interface Animal{ }  12  13 class Dog implements Animal{ } |

之所可以强制转换，这得归功于RTTI，要知道在Java中，所有类型转换都是在运行时进行正确性检查的，利用RTTI进行判断类型是否正确从而确保强制转换的完成，如果类型转换失败，将会抛出类型转换异常。除了强制转换外，在Java SE5中新增一种使用Class对象进行类型转换的方式，如下：

|  |
| --- |
| 1 Animal animal= new Dog();  2 //这两句等同于Dog dog = (Dog) animal;  3 Class<Dog> dogType = Dog.class;  4 Dog dog = dogType.cast(animal) |

利用Class对象的cast方法，其参数接收一个参数对象并将其转换为Class引用的类型。这种方式似乎比之前的强制转换更麻烦些，确实如此，而且当类型不能正确转换时，仍然会抛出ClassCastException异常。源码如下：

|  |
| --- |
| 1 public T cast(Object obj) {  2 if (obj != null && !isInstance(obj))  3 throw new ClassCastException(cannotCastMsg(obj));  4 return (T) obj;  5 } |

# 四、instanceof 关键字与isInstance方法

　　关于instanceof 关键字，它返回一个boolean类型的值，意在告诉我们对象是不是某个特定的类型实例。如下，在强制转换前利用instanceof检测obj是不是Animal类型的实例对象，如果返回true再进行类型转换，这样可以避免抛出类型转换的异常(ClassCastException)

|  |
| --- |
| 1 public void cast2(Object obj){  2 if(obj instanceof Animal){  3 Animal animal= (Animal) obj;  4 }  5 } |

而isInstance方法则是Class类中的一个Native方法，也是用于判断对象类型的，看个简单例子：

|  |
| --- |
| 1 public void cast2(Object obj){  2 //instanceof关键字  3 if(obj instanceof Animal){  4 Animal animal= (Animal) obj;  5 }  6  7 //isInstance方法  8 if(Animal.class.isInstance(obj)){  9 Animal animal= (Animal) obj;  10 }  11 } |

事实上instanceOf 与isInstance方法产生的结果是相同的。对于instanceOf是关键字只被用于对象引用变量，检查左边对象是不是右边类或接口的实例化。如果被测对象是null值，则测试结果总是false。一般形式：

|  |
| --- |
| 1 //判断这个对象是不是这种类型  2 obj.instanceof(class) |

而isInstance方法则是Class类的Native方法，其中obj是被测试的对象或者变量，如果obj是调用这个方法的class或接口的实例，则返回true。如果被检测的对象是null或者基本类型，那么返回值是false;一般形式如下：

|  |
| --- |
| 1 //判断这个对象能不能被转化为这个类  2 class.inInstance(obj) |

最后这里给出一个简单实例，验证isInstance方法与instanceof等价性：

|  |
| --- |
| 1 class A {}  2  3 class B extends A {}  4  5 public class C {  6 static void test(Object x) {  7 print("Testing x of type " + x.getClass());  8 print("x instanceof A " + (x instanceof A));  9 print("x instanceof B "+ (x instanceof B));  10 print("A.isInstance(x) "+ A.class.isInstance(x));  11 print("B.isInstance(x) " +  12 B.class.isInstance(x));  13 print("x.getClass() == A.class " +  14 (x.getClass() == A.class));  15 print("x.getClass() == B.class " +  16 (x.getClass() == B.class));  17 print("x.getClass().equals(A.class)) "+  18 (x.getClass().equals(A.class)));  19 print("x.getClass().equals(B.class)) " +  20 (x.getClass().equals(B.class)));  21 }  22 public static void main(String[] args) {  23 test(new A());  24 test(new B());  25 }  26 } |

执行结果：

|  |
| --- |
| 1 Testing x of type class com.wangjun.A  2 x instanceof A true  3 x instanceof B false //父类不一定是子类的某个类型  4 A.isInstance(x) true  5 B.isInstance(x) false  6 x.getClass() == A.class true  7 x.getClass() == B.class false  8 x.getClass().equals(A.class)) true  9 x.getClass().equals(B.class)) false  10 ---------------------------------------------  11 Testing x of type class com.wangjun.B  12 x instanceof A true  13 x instanceof B true  14 A.isInstance(x) true  15 B.isInstance(x) true  16 x.getClass() == A.class false  17 x.getClass() == B.class true  18 x.getClass().equals(A.class)) false  19 x.getClass().equals(B.class)) true |

到此关于Class对象相关的知识点都分析完了，下面将结合Class对象的知识点分析反射技术。

**五、理解反射技术**

　　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。一直以来反射技术都是Java中的闪亮点，这也是目前大部分框架(如Spring/Mybatis等)得以实现的支柱。在Java中，Class类与java.lang.reflect类库一起对反射技术进行了全力的支持。在反射包中，我们常用的类主要有Constructor类表示的是Class 对象所表示的类的构造方法，利用它可以在运行时动态创建对象、Field表示Class对象所表示的类的成员变量，通过它可以在运行时动态修改成员变量的属性值(包含private)、Method表示Class对象所表示的类的成员方法，通过它可以动态调用对象的方法(包含private)，下面将对这几个重要类进行分别说明。

**1、Constructor类及其用法**

Constructor类存在于反射包(java.lang.reflect)中，反映的是Class 对象所表示的类的构造方法。获取Constructor对象是通过Class类中的方法获取的，Class类与Constructor相关的主要方法如下：

|  |  |  |
| --- | --- | --- |
| 方法返回值 | 方法名称 | 方法说明 |
| static Class<?> | forName(String className) | 返回与带有给定字符串名的类或接口相关联的 Class 对象。 |
| Constructor<T> | getConstructor(Class<?>... parameterTypes) | 返回指定参数类型、具有public访问权限的构造函数对象 |
| Constructor<?>[] | getConstructors() | 返回所有具有public访问权限的构造函数的Constructor对象数组 |
| Constructor<T> | getDeclaredConstructor(Class<?>... parameterTypes) | 返回指定参数类型、所有声明的（包括private）构造函数对象 |
| Constructor<?>[] | getDeclaredConstructor() | 返回所有声明的（包括private）构造函数对象 |
| T | |  | | --- | | newInstance() | | 创建此 Class 对象所表示的类的一个新实例。 |

下面看一个简单例子来了解Constructor对象的使用：

|  |
| --- |
| 1 package reflect;  2  3 import java.io.Serializable;  4 import java.lang.reflect.Constructor;  5  6 /\*\*  7 \* Created by wangjun on 2017/5/1.  8 \* Blog : http://blog.csdn.net/javawangjun [原文地址,请尊重原创]  9 \*/  10 public class ReflectDemo implements Serializable{  11 public static void main(String[] args) throws Exception {  12  13 Class<?> clazz = null;  14  15 //获取Class对象的引用  16 clazz = Class.forName("reflect.User");  17  18 //第一种方法，实例化默认构造方法，User必须无参构造函数,否则将抛异常  19 User user = (User) clazz.newInstance();  20 user.setAge(20);  21 user.setName("Rollen");  22 System.out.println(user);  23  24 System.out.println("--------------------------------------------");  25  26 //获取带String参数的public构造函数  27 Constructor cs1 =clazz.getConstructor(String.class);  28 //创建User  29 User user1= (User) cs1.newInstance("xiaolong");  30 user1.setAge(22);  31 System.out.println("user1:"+user1.toString());  32  33 System.out.println("--------------------------------------------");  34  35 //取得指定带int和String参数构造函数,该方法是私有构造private  36 Constructor cs2=clazz.getDeclaredConstructor(int.class,String.class);  37 //由于是private必须设置可访问  38 cs2.setAccessible(true);  39 //创建user对象  40 User user2= (User) cs2.newInstance(25,"lidakang");  41 System.out.println("user2:"+user2.toString());  42  43 System.out.println("--------------------------------------------");  44  45 //获取所有构造包含private  46 Constructor<?> cons[] = clazz.getDeclaredConstructors();  47 // 查看每个构造方法需要的参数  48 for (int i = 0; i < cons.length; i++) {  49 //获取构造函数参数类型  50 Class<?> clazzs[] = cons[i].getParameterTypes();  51 System.out.println("构造函数["+i+"]:"+cons[i].toString() );  52 System.out.print("参数类型["+i+"]:(");  53 for (int j = 0; j < clazzs.length; j++) {  54 if (j == clazzs.length - 1)  55 System.out.print(clazzs[j].getName());  56 else  57 System.out.print(clazzs[j].getName() + ",");  58 }  59 System.out.println(")");  60 }  61 }  62 }  63  64  65 class User {  66 private int age;  67 private String name;  68 public User() {  69 super();  70 }  71 public User(String name) {  72 super();  73 this.name = name;  74 }  75  76 /\*\*  77 \* 私有构造  78 \* @param age  79 \* @param name  80 \*/  81 private User(int age, String name) {  82 super();  83 this.age = age;  84 this.name = name;  85 }  86  87 //..........省略set 和 get方法  88 } |

运行结果：

|  |
| --- |
| 1 User [age=20, name=Rollen]  2 --------------------------------------------  3 user1:User [age=22, name=xiaolong]  4 --------------------------------------------  5 user2:User [age=25, name=lidakang]  6 --------------------------------------------  7 构造函数[0]:private reflect.User(int,java.lang.String)  8 参数类型[0]:(int,java.lang.String)  9 构造函数[1]:public reflect.User(java.lang.String)  10 参数类型[1]:(java.lang.String)  11 构造函数[2]:public reflect.User()  12 参数类型[2]:() |

**1.1关于Constructor类本身一些常用方法如下(仅部分，其他可查API)**

|  |  |  |
| --- | --- | --- |
| 回值 | 方法名称 | 方法说明 |
| Class<T> | getDeclaringClass() | 返回 Class 对象，该对象表示声明由此 Constructor 对象表示的构造方法的类,其实就是返回真实类型（不包含参数） |
| Type[] | getGenericParameterTypes() | 按照声明顺序返回一组 Type 对象，返回的就是 Constructor对象构造函数的形参类型。 |
| String | getName() | 以字符串形式返回此构造方法的名称。 |
| Class<?>[] | getParameterTypes() | 按照声明顺序返回一组 Class 对象，即返回Constructor 对象所表示构造方法的形参类型 |
| T | newInstance(Object... initargs) | 使用此 Constructor对象表示的构造函数来创建新实例 |
| String | toGenericString() | 返回描述此 Constructor 的字符串，其中包括类型参数。 |

代码演示如下：

|  |
| --- |
| 1 Constructor cs3=clazz.getDeclaredConstructor(int.class,String.class);  2  3 System.out.println("-----getDeclaringClass-----");  4 Class uclazz=cs3.getDeclaringClass();  5 //Constructor对象表示的构造方法的类  6 System.out.println("构造方法的类:"+uclazz.getName());  7  8 System.out.println("-----getGenericParameterTypes-----");  9 //对象表示此 Constructor 对象所表示的方法的形参类型  10 Type[] tps=cs3.getGenericParameterTypes();  11 for (Type tp:tps) {  12 System.out.println("参数名称tp:"+tp);  13 }  14 System.out.println("-----getParameterTypes-----");  15 //获取构造函数参数类型  16 Class<?> clazzs[] = cs3.getParameterTypes();  17 for (Class claz:clazzs) {  18 System.out.println("参数名称:"+claz.getName());  19 }  20 System.out.println("-----getName-----");  21 //以字符串形式返回此构造方法的名称  22 System.out.println("getName:"+cs3.getName());  23  24 System.out.println("-----getoGenericString-----");  25 //返回描述此 Constructor 的字符串，其中包括类型参数。  26 System.out.println("getoGenericString():"+cs3.toGenericString()); |

输出结果

|  |
| --- |
| 1  2 -----getDeclaringClass-----  3 构造方法的类:reflect.User  4 -----getGenericParameterTypes-----  5 参数名称tp:int  6 参数名称tp:class java.lang.String  7 -----getParameterTypes-----  8 参数名称:int  9 参数名称:java.lang.String  10 -----getName-----  11 getName:reflect.User  12 -----getoGenericString-----  13 getoGenericString():private reflect.User(int,java.lang.String) |

其中关于Type类型这里简单说明一下，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。getGenericParameterTypes 与 getParameterTypes 都是获取构成函数的参数类型，前者返回的是Type类型，后者返回的是Class类型，由于Type顶级接口，Class也实现了该接口，因此Class类是Type的子类，Type 表示的全部类型而每个Class对象表示一个具体类型的实例，如String.class仅代表String类型。由此看来Type与 Class 表示类型几乎是相同的，只不过 Type表示的范围比Class要广得多而已。当然Type还有其他子类，如：

　　TypeVariable：表示类型参数，可以有上界，比如：T extends Number

　　ParameterizedType：表示参数化的类型，有原始类型和具体的类型参数，比如：List<String>

　　WildcardType：表示通配符类型，比如：?, ? extends Number, ? super Integer

　　通过以上的分析，对于Constructor类已有比较清晰的理解，利用好Class类和Constructor类，我们可以在运行时动态创建任意对象，从而突破必须在编译期知道确切类型的障碍。

**2.Field类及其用法**

　　Field 提供有关类或接口的单个字段的信息，以及对它的动态访问权限。反射的字段可能是一个类（静态）字段或实例字段。同样的道理，我们可以通过Class类的提供的方法来获取代表字段信息的Field对象，Class类与Field对象相关方法如下：

|  |  |  |
| --- | --- | --- |
| 方法返回值 | 方法名称 | 方法说明 |
| Field | getDeclaredField(String name) | 获取指定name名称的(包含private修饰的)字段，不包括继承的字段 |
| Field[] | getDeclaredField() | 获取Class对象所表示的类或接口的所有(包含private修饰的)字段,不包括继承的字段 |
| Field | getField(String name) | 获取指定name名称、具有public修饰的字段，包含继承字段 |
| Field[] | getField() | 获取修饰符为public的字段，包含继承字段 |

下面的代码演示了上述方法的使用过程

|  |
| --- |
| 1 public class ReflectField {  2  3 public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException {  4 Class<?> clazz = Class.forName("reflect.Student");  5 //获取指定字段名称的Field类,注意字段修饰符必须为public而且存在该字段,  6 // 否则抛NoSuchFieldException  7 Field field = clazz.getField("age");  8 System.out.println("field:"+field);  9  10 //获取所有修饰符为public的字段,包含父类字段,注意修饰符为public才会获取  11 Field fields[] = clazz.getFields();  12 for (Field f:fields) {  13 System.out.println("f:"+f.getDeclaringClass());  14 }  15  16 System.out.println("================getDeclaredFields====================");  17 //获取当前类所字段(包含private字段),注意不包含父类的字段  18 Field fields2[] = clazz.getDeclaredFields();  19 for (Field f:fields2) {  20 System.out.println("f2:"+f.getDeclaringClass());  21 }  22 //获取指定字段名称的Field类,可以是任意修饰符的自动,注意不包含父类的字段  23 Field field2 = clazz.getDeclaredField("desc");  24 System.out.println("field2:"+field2);  25 }  26  27 }  28  29 class Person{  30 public int age;  31 public String name;  32 //省略set和get方法  33 }  34  35 class Student extends Person{  36 public String desc;  37 private int score;  38 //省略set和get方法  39 } |

输出结果：

|  |
| --- |
| 1 field:public int reflect.Person.age  2 f:public java.lang.String reflect.Student.desc  3 f:public int reflect.Person.age  4 f:public java.lang.String reflect.Person.name  5  6 ================getDeclaredFields====================  7 f2:public java.lang.String reflect.Student.desc  8 f2:private int reflect.Student.score  9 field2:public java.lang.String reflect.Student.desc |

上述方法需要注意的是，如果我们不期望获取其父类的字段，则需使用Class类的getDeclaredField/getDeclaredFields方法来获取字段即可，倘若需要连带获取到父类的字段，那么请使用Class类的getField/getFields，但是也只能获取到public修饰的的字段，无法获取父类的私有字段。下面将通过Field类本身的方法对指定类属性赋值，代码演示如下：

|  |
| --- |
| 1 //获取Class对象引用  2 Class<?> clazz = Class.forName("reflect.Student");  3  4 Student st= (Student) clazz.newInstance();  5 //获取父类public字段并赋值  6 Field ageField = clazz.getField("age");  7 ageField.set(st,18);  8 Field nameField = clazz.getField("name");  9 nameField.set(st,"Lily");  10  11 //只获取当前类的字段,不获取父类的字段  12 Field descField = clazz.getDeclaredField("desc");  13 descField.set(st,"I am student");  14 Field scoreField = clazz.getDeclaredField("score");  15 //设置可访问，score是private的  16 scoreField.setAccessible(true);  17 scoreField.set(st,88);  18 System.out.println(st.toString());  19  20 //输出结果：Student{age=18, name='Lily ,desc='I am student', score=88}  21  22 //获取字段值  23 System.out.println(scoreField.get(st));  24 // 88 |

其中的set(Object obj, Object value)方法是Field类本身的方法，用于设置字段的值，而get(Object obj)则是获取字段的值，当然关于Field类还有其他常用的方法如下：

|  |  |  |
| --- | --- | --- |
| 方法返回值 | 方法名称 | 方法说明 |
| void | set(Object obj, Object value) | 将指定对象变量上此 Field 对象表示的字段设置为指定的新值。 |
| Object | get(Object obj) | 返回指定对象上此 Field 表示的字段的值 |
| Class<?> | getType() | 返回一个 Class 对象，它标识了此Field 对象所表示字段的声明类型。 |
| boolean | isEnumConstant() | 如果此字段表示枚举类型的元素则返回 true；否则返回 false |
| String | toGenericString() | 返回一个描述此 Field（包括其一般类型）的字符串 |
| String | getName() | 返回此 Field 对象表示的字段的名称 |
| Class<?> | getDeclaringClass() | 返回表示类或接口的 Class 对象，该类或接口声明由此 Field 对象表示的字段 |
| void | setAccessible(boolean flag) | 将此对象的 accessible 标志设置为指示的布尔值,即设置其可访问性 |

　　　上述方法可能是较为常用的，事实上在设置值的方法上，Field类还提供了专门针对基本数据类型的方法，如setInt()/getInt()、setBoolean()/getBoolean、setChar()/getChar()等等方法，这里就不全部列出了，需要时查API文档即可。需要特别注意的是被final关键字修饰的Field字段是安全的，在运行时可以接收任何修改，但最终其实际值是不会发生改变的。

**3.Method类及其用法**

　　Method 提供关于类或接口上单独某个方法（以及如何访问该方法）的信息，所反映的方法可能是类方法或实例方法（包括抽象方法）。下面是Class类获取Method对象相关的方法：

|  |  |  |
| --- | --- | --- |
| 方法返回值 | 方法名称 | 方法说明 |
| Method | getDeclaredMethod(String name, Class<?>... parameterTypes) | 返回一个指定参数的Method对象，该对象反映此 Class 对象所表示的类或接口的指定已声明方法。 |
| Method[] | getDeclaredMethod() | 返回 Method 对象的一个数组，这些对象反映此 Class 对象表示的类或接口声明的所有方法，包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法。 |
| Method | getMethod(String name, Class<?>... parameterTypes) | 返回一个 Method 对象，它反映此 Class 对象所表示的类或接口的指定公共成员方法。 |
| |  | | --- | | Method[] | | getMethods() | 返回一个包含某些 Method 对象的数组，这些对象反映此 Class 对象所表示的类或接口（包括那些由该类或接口声明的以及从超类和超接口继承的那些的类或接口）的公共 member 方法。 |

同样通过案例演示上述方法：

|  |
| --- |
| 1 public class ReflectMethod {  2  3  4 public static void main(String[] args) throws ClassNotFoundException, NoSuchMethodException {  5  6 Class clazz = Class.forName("reflect.Circle");  7  8 //根据参数获取public的Method,包含继承自父类的方法  9 Method method = clazz.getMethod("draw",int.class,String.class);  10  11 System.out.println("method:"+method);  12  13 //获取所有public的方法:  14 Method[] methods =clazz.getMethods();  15 for (Method m:methods){  16 System.out.println("m::"+m);  17 }  18  19 System.out.println("=========================================");  20  21 //获取当前类的方法包含private,该方法无法获取继承自父类的method  22 Method method1 = clazz.getDeclaredMethod("drawCircle");  23 System.out.println("method1::"+method1);  24 //获取当前类的所有方法包含private,该方法无法获取继承自父类的method  25 Method[] methods1=clazz.getDeclaredMethods();  26 for (Method m:methods1){  27 System.out.println("m1::"+m);  28 }  29 }  30  31 }  32  33 class Shape {  34 public void draw(){  35 System.out.println("draw");  36 }  37  38 public void draw(int count , String name){  39 System.out.println("draw "+ name +",count="+count);  40 }  41  42 }  43 class Circle extends Shape{  44  45 private void drawCircle(){  46 System.out.println("drawCircle");  47 }  48 public int getAllCount(){  49 return 100;  50 }  51 } |

输出结果：

|  |
| --- |
| 1 输出结果:  2 method:public void reflect.Shape.draw(int,java.lang.String)  3 m::public int reflect.Circle.getAllCount()  4 m::public void reflect.Shape.draw()  5 m::public void reflect.Shape.draw(int,java.lang.String)  6 m::public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException  7 m::public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException  8 m::public final void java.lang.Object.wait() throws java.lang.InterruptedException  9 m::public boolean java.lang.Object.equals(java.lang.Object)  10 m::public java.lang.String java.lang.Object.toString()  11 m::public native int java.lang.Object.hashCode()  12 m::public final native java.lang.Class java.lang.Object.getClass()  13 m::public final native void java.lang.Object.notify()  14 m::public final native void java.lang.Object.notifyAll()  15  16 =========================================  17 method1::private void reflect.Circle.drawCircle()  18  19 m1::public int reflect.Circle.getAllCount()  20 m1::private void reflect.Circle.drawCircle() |

在通过getMethods方法获取Method对象时，会把父类的方法也获取到，如上的输出结果，把Object类的方法都打印出来了。而getDeclaredMethod/getDeclaredMethods方法都只能获取当前类的方法。我们在使用时根据情况选择即可。下面将演示通过Method对象调用指定类的方法：

|  |
| --- |
| 1 Class clazz = Class.forName("reflect.Circle");  2 //创建对象  3 Circle circle = (Circle) clazz.newInstance();  4  5 //获取指定参数的方法对象Method  6 Method method = clazz.getMethod("draw",int.class,String.class);  7  8 //通过Method对象的invoke(Object obj,Object... args)方法调用  9 method.invoke(circle,15,"圈圈");  10  11 //对私有无参方法的操作  12 Method method1 = clazz.getDeclaredMethod("drawCircle");  13 //修改私有方法的访问标识  14 method1.setAccessible(true);  15 method1.invoke(circle);  16  17 //对有返回值得方法操作  18 Method method2 =clazz.getDeclaredMethod("getAllCount");  19 Integer count = (Integer) method2.invoke(circle);  20 System.out.println("count:"+count); |

输出结果：

|  |
| --- |
| 1 draw 圈圈,count=15  2 drawCircle  3 count:100 |

在上述代码中调用方法，使用了Method类的invoke(Object obj,Object... args)第一个参数代表调用的对象，第二个参数传递的调用方法的参数。这样就完成了类方法的动态调用。

|  |  |  |
| --- | --- | --- |
| 方法返回值 | 方法名称 | 方法说明 |
| Object | invoke(Object obj, Object... args) | 对带有指定参数的指定对象调用由此 Method 对象表示的底层方法。 |
| Class<?> | getReturnType() | 返回一个 Class 对象，该对象描述了此 Method 对象所表示的方法的正式返回类型,即方法的返回类型 |
| Type | getGenericReturnType() | 返回表示由此 Method 对象所表示方法的正式返回类型的 Type 对象，也是方法的返回类型。 |
| Class<?>[] | getParameterTypes() | 按照声明顺序返回 Class 对象的数组，这些对象描述了此 Method 对象所表示的方法的形参类型。即返回方法的参数类型组成的数组 |
| Type[] | getGenericParameterTypes() | 按照声明顺序返回 Type 对象的数组，这些对象描述了此 Method 对象所表示的方法的形参类型的，也是返回方法的参数类型 |
| String | getName() | 以 String 形式返回此 Method 对象表示的方法名称，即返回方法的名称 |
| boolean | isVarArgs() | 判断方法是否带可变参数，如果将此方法声明为带有可变数量的参数，则返回 true；否则，返回 false。 |
| String | toGenericString() | 返回描述此 Method 的字符串，包括类型参数。 |

　　getReturnType方法/getGenericReturnType方法都是获取Method对象表示的方法的返回类型，只不过前者返回的Class类型后者返回的Type(前面已分析过)，Type就是一个接口而已，在Java8中新增一个默认的方法实现，返回的就参数类型信息。

|  |
| --- |
| 1 public interface Type {  2 //1.8新增  3 default String getTypeName() {  4 return toString();  5 }  6 } |

而getParameterTypes/getGenericParameterTypes也是同样的道理，都是获取Method对象所表示的方法的参数类型，其他方法与前面的Field和Constructor是类似的。

**4、反射包中的Array类**

　　在Java的java.lang.reflect包中存在着一个可以动态操作数组的类，Array，它提供了动态创建和访问 Java 数组的方法。Array 允许在执行 get 或 set 操作进行取值和赋值。在Class类中与数组关联的方法是：

|  |  |  |
| --- | --- | --- |
| 方法返回值 | 方法名称 | 方法说明 |
| Class<?> | getComponentType() | 返回表示数组元素类型的 Class，即数组的类型 |
| boolean | isArray() | 判定此 Class 对象是否表示一个数组类。 |

　　java.lang.reflect.Array中的常用静态方法如下：

|  |  |  |
| --- | --- | --- |
| 方法返回值 | 方法名称 | 方法说明 |
| static Objec | set(Object array, int index) | 返回指定数组对象中索引组件的值。 |
| static int | getLength(Object array) | 以 int 形式返回指定数组对象的长度 |
| static object | newInstance(Class<?> componentType, int... dimensions) | 创建一个具有指定类型和维度的新数组。 |
| static Object | newInstance(Class<?> componentType, int length) | 创建一个具有指定的组件类型和长度的新数组。 |
| static void | set(Object array, int index, Object value) | 将指定数组对象中索引组件的值设置为指定的新值。 |

下面通过一个简单例子来演示这些方法

|  |
| --- |
| 1 import java.lang.reflect.Array;  2  3 public class ReflectArray {  4  5 public static void main(String[] args) throws ClassNotFoundException {  6 int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };  7 //获取数组类型的Class 即int.class  8 Class<?> clazz = array.getClass().getComponentType();  9 //创建一个具有指定的组件类型和长度的新数组。  10 //第一个参数:数组的类型,第二个参数:数组的长度  11 Object newArr = Array.newInstance(clazz, 15);  12 //获取原数组的长度  13 int co = Array.getLength(array);  14 //赋值原数组到新数组  15 System.arraycopy(array, 0, newArr, 0, co);  16 for (int i:(int[]) newArr) {  17 System.out.print(i+",");  18 }  19  20 //创建了一个长度为10 的字符串数组，  21 //接着把索引位置为6 的元素设为"hello world!"，然后再读取索引位置为6 的元素的值  22 Class clazz2 = Class.forName("java.lang.String");  23  24 //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  25 Object array2 = Array.newInstance(clazz2, 10);  26  27 //把字符串数组对象的索引位置为6的元素设置为"hello"  28 Array.set(array2, 6, "hello world!");  29  30 //获得字符串数组对象的索引位置为5的元素的值  31 String str = (String)Array.get(array2, 6);  32 System.out.println();  33 System.out.println(str);//hello  34 }  35  36 } |

输出结果：

|  |
| --- |
| 1 1,2,3,4,5,6,7,8,9,0,0,0,0,0,0,  2 hello world! |

通过上述代码演示，确实可以利用Array类和反射相结合动态创建数组，也可以在运行时动态获取和设置数组中元素的值，其实除了上的set/get外Array还专门为8种基本数据类型提供特有的方法，如setInt/getInt、setBoolean/getBoolean，其他依次类推，需要使用是可以查看API文档即可。除了上述动态修改数组长度或者动态创建数组或动态获取值或设置值外，可以利用泛型动态创建泛型数组如下：

|  |
| --- |
| 1 /\*\*  2 \* 接收一个泛型数组，然后创建一个长度与接收的数组长度一样的泛型数组，  3 \* 并把接收的数组的元素复制到新创建的数组中，  4 \* 最后找出新数组中的最小元素，并打印出来  5 \* @param a  6 \* @param <T>  7 \*/  8 public <T extends Comparable<T>> void min(T[] a) {  9 //通过反射创建相同类型的数组  10 T[] b = (T[]) Array.newInstance(a.getClass().getComponentType(), a.length);  11 for (int i = 0; i < a.length; i++) {  12 b[i] = a[i];  13 }  14 T min = null;  15 boolean flag = true;  16 for (int i = 0; i < b.length; i++) {  17 if (flag) {  18 min = b[i];  19 flag = false;  20 }  21 if (b[i].compareTo(min) < 0) {  22 min = b[i];  23 }  24 }  25 System.out.println(min);  26 } |

毕竟我们无法直接创建泛型数组，有了Array的动态创建数组的方式这个问题也就迎刃而解了。

|  |
| --- |
| 1 //无效语句，编译不通  2 T[] a = new T[]; |

ok~，到这反射中几个重要并且常用的类我们都基本介绍完了，但更重要是，我们应该认识到反射机制并没有什么神奇之处。当通过反射与一个未知类型的对象打交道时，JVM只会简单地检查这个对象，判断该对象属于那种类型，同时也应该知道，在使用反射机制创建对象前，必须确保已加载了这个类的Class对象，当然这点完全不必由我们操作，毕竟只能JVM加载，但必须确保该类的”.class”文件已存在并且JVM能够正确找到。关于Class类的方法在前面我们只是分析了主要的一些方法，其实Class类的API方法挺多的，建议查看一下API文档，浏览一遍，有个印象也是不错的选择，这里仅列出前面没有介绍过又可能用到的API：

|  |
| --- |
| 1 /\*\*  2 \* 修饰符、父类、实现的接口、注解相关  3 \*/  4  5 //获取修饰符，返回值可通过Modifier类进行解读  6 public native int getModifiers();  7 //获取父类，如果为Object，父类为null  8 public native Class<? super T> getSuperclass();  9 //对于类，为自己声明实现的所有接口，对于接口，为直接扩展的接口，不包括通过父类间接继承来的  10 public native Class<?>[] getInterfaces();  11 //自己声明的注解  12 public Annotation[] getDeclaredAnnotations();  13 //所有的注解，包括继承得到的  14 public Annotation[] getAnnotations();  15 //获取或检查指定类型的注解，包括继承得到的  16 public <A extends Annotation> A getAnnotation(Class<A> annotationClass);  17 public boolean isAnnotationPresent(Class<? extends Annotation> annotationClass);  18  19 /\*\*  20 \* 内部类相关  21 \*/  22 //获取所有的public的内部类和接口，包括从父类继承得到的  23 public Class<?>[] getClasses();  24 //获取自己声明的所有的内部类和接口  25 public Class<?>[] getDeclaredClasses();  26 //如果当前Class为内部类，获取声明该类的最外部的Class对象  27 public Class<?> getDeclaringClass();  28 //如果当前Class为内部类，获取直接包含该类的类  29 public Class<?> getEnclosingClass();  30 //如果当前Class为本地类或匿名内部类，返回包含它的方法  31 public Method getEnclosingMethod();  32  33 /\*\*  34 \* Class对象类型判断相关  35 \*/  36 //是否是数组  37 public native boolean isArray();  38 //是否是基本类型  39 public native boolean isPrimitive();  40 //是否是接口  41 public native boolean isInterface();  42 //是否是枚举  43 public boolean isEnum();  44 //是否是注解  45 public boolean isAnnotation();  46 //是否是匿名内部类  47 public boolean isAnonymousClass();  48 //是否是成员类  49 public boolean isMemberClass();  50 //是否是本地类  51 public boolean isLocalClass(); |