**Java 反射**

**Class类**

Java程序在运行时，Java运行时系统一直对所有的对象进行所谓的运行时类型标识，即所谓的**RTTI**(多态是基于RTTI实现的)。这项信息纪录了每个对象所属的类。虚拟机通常使用运行时类型信息选准正确方法去执行，用来保存这些类型信息的类是Class类。Class类封装一个对象和接口运行时的状态，当装载类时，Class类型的对象自动创建。

Class类没有公共的构造方法，Class对象是在类加载的时候由Java虚拟机以及通过调用类加载器中的 defineClass 方法自动构造的，因此不能显式地声明一个Class对象。一个类被加载到内存并供我们使用需要经历如下三个阶段：

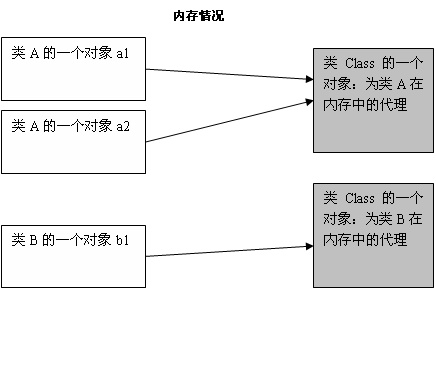
1. 加载，这是由类加载器（ClassLoader）执行的。通过一个类的全限定名来获取其定义的二进制字节流（Class字节码），将这个字节流所代表的静态存储结构转化为方法去的运行时数据接口，根据字节码在java堆中生成一个代表这个类的java.lang.Class对象。
2. 链接。在链接阶段将验证Class文件中的字节流包含的信息是否符合当前虚拟机的要求，为静态域分配存储空间并设置类变量的初始值（默认的零值），并且如果必需的话，将常量池中的符号引用转化为直接引用。
3. 初始化。到了此阶段，才真正开始执行类中定义的java程序代码。用于执行该类的静态初始器和静态初始块，如果该类有父类的话，则优先对其父类进行初始化。

所有的类都是在对其第一次使用时，动态加载到JVM中的（懒加载）。当程序创建第一个对类的静态成员的引用时，就会加载这个类。使用new创建类对象的时候也会被当作对类的静态成员的引用。因此java程序程序在它开始运行之前并非被完全加载，其各个类都是在必需时才加载的。这一点与许多传统语言都不同。动态加载使能的行为，在诸如C++这样的静态加载语言中是很难或者根本不可能复制的。

在类加载阶段，类加载器首先检查这个类的Class对象是否已经被加载。如果尚未加载，默认的类加载器就会根据类的全限定名查找.class文件。在这个类的字节码被加载时，它们会接受验证，以确保其没有被破坏，并且不包含不良java代码。一旦某个类的Class对象被载入内存，我们就可以它来创建这个类的所有对象。

**原理**

在object这个类中有一个方法：getclass().这个方法是用来取得该类已经被实例化了的对象的该类的引用，这个引用指向的是Class类的对象。我们自己无法生成一个Class对象（构造函数为private)，而 这个Class类的对象是在当各类被调入时，由 Java 虚拟机自动创建 Class 对象，或通过类装载器中的 defineClass 方法生成。我们生成的对象都会有个字段记录该对象所属类在CLass类的对象的所在位置。



**获取一个Class类对象**

**forName**

public class shapes{}

Class obj = Class.forName(“shapes”;)

**getClass**

public class shapes{}

shapes s1= new shapes();

Class obj = s1.getClass();

Class obj1 = s1.getSuperClass(); //获取shapes 类的父类的类型

**使用类的字面常量**

在编译时就会受到检查(因此不需要置于try语句块中)。 该方式不会自动地初始化该Class对象（这点和Class.forName方法不同）。类对象的初始化阶段被延迟到了对静态方法或者非常数静态域首次引用时才执行。

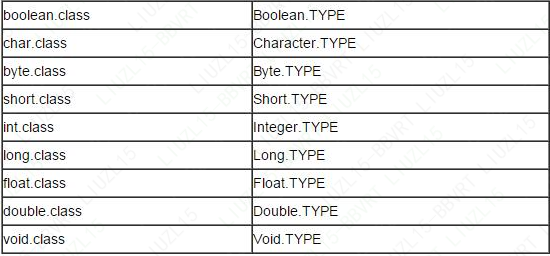
Class obj = String.class;

Class obj1 = int.class;

**注意**

使用这种办法生成Class类对象时，不会使JVM自动加载该类(如String类)。而其他办法会使得JVM初始化该类。

包装类中有一个字段TYPE，TYPE字段是一个引用，指向对应的基本数据类型的Class对象。如



说明：

如果一个字段被 static final 修饰，称为“编译时常量”，在调用这个字段的时候是不会对该类进行初始化的。因为被 static 和 final 修饰的字段，在编译时期就把结果放入了常量池中。

**特别说明**

其实对于任意一个Class对象，都需要由它的**类加载器**和这个**类本身**一同确定其在就Java虚拟机中的唯一性，也就是说，即使两个Class对象来源于同一个Class文件，只要加载它们的类加载器不同，那这两个Class对象就必定不相等。这里的“相等”包括了代表类的Class对象的equals（）、isAssignableFrom（）、isInstance（）等方法的返回结果，也包括了使用instanceof关键字对对象所属关系的判定结果。所以在java虚拟机中使用双亲委派模型来组织类加载器之间的关系，来保证Class对象的唯一性。

**使用Class类的对象生成目标类的实例**

**生成不精确的object实例**

获取一个Class类的对象后，可以用 newInstance() 函数来生成目标类的一个实例。然而，该函数并不能直接生成目标类的实例，只能生成object类的实例

Class obj= Class.forName(“shapes”);

Object shapesInstance = obj.newInstance();

**使用泛化Class引用生成带类型的目标类型**

Class<shapes> obj = shapes.class;

shapes newShape = obj.newInstance();

因为有了类型限制，所以使用泛化Class语法的对象引用不能指向别的类

Class obj1 = int.class;

Class<Integer> obj2 = int.class;

obj1 = double.class;

//obj2 = double.class; //这一行代码是非法的，obj2不能改指向别的类了

**有一个灵活的用法，使得可以用Class的对象指向基类的任何子类**。

Class<?> obj = int.class;

obj = double.class;

obj = shapes.class;

**使用这种泛型语法来构建你手头有的一个Class类的对象的基类对象时，必须采用以下的特殊语法**

public class shapes{}

class round extends shapes{}

Class<round> rclass = round.class;

Class<? super round> sclass = rclass.getSuperClass();

//Class<shapes> sclass = rclass.getSuperClass(); //**错误，必须使用Class<? super round>**

**Class 的方法**

**forName()**

(1)获取Class对象的一个引用，但引用的类还没有加载(该类的第一个对象没有生成)就加载了这个类。

(2)为了产生Class引用，forName()立即就进行了初始化。

**Object-getClass()**

获取Class对象的一个引用，返回表示该对象的实际类型的Class引用。

**getName()**

取全限定的类名(包括包名)，即类的完整名字。

com.cry.Test$inner

[Ljava.lang.String;

**getSimpleName()**

获取类名(不包括包名)

**getCanonicalName()**

返回更容易理解的表示，主要用于输出(toString)或log打印，大多数情况下和getName一样，但是在内部类、数组等类型的表示形式就不同了。

com.cry.Test.inner

java.lang.String[]

**isInterface()**

判断Class对象是否是表示一个接口

**getInterfaces()**

返回Class对象数组，表示Class对象所引用的类所实现的所有接口。

**getSupercalss()**

返回Class对象，表示Class对象所引用的类所继承的直接基类。应用该方法可在运行时发现一个对象完整的继承结构。

**newInstance()**

返回一个Oject对象，是实现“虚拟构造器”的一种途径。使用该方法创建的类，必须带有无参的构造器。

**getFields()**

获得某个类的所有的公共（public）的字段，包括继承自父类的所有公共字段。 类似的还有**getMethods**和**getConstructors**。

**getDeclaredFields**

获得某个类的自己声明的字段，即包括public、private和proteced，默认但是不包括父类声明的任何字段。类似的还有**getDeclaredMethods**和**getDeclaredConstructors**。

**getConstructors()**

返回每一个声明为公有的（Public）构造方法。如果知道要访问的构造方法的方法参数类型，可以用下面的方法获取指定的构造方法。如：

Constructor constructor = aClass.getConstructor(new Class[]{String.class})；

**概念**

JAVA反射机制

JAVA反射机制是在运行状态中，对于任意一个实体类，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方法和属性；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

程序运行时，允许改变程序结构或变量类型。

**使用**

**获取类的构造方法**

1. 获取类对象
2. 获取构造方法
   1. 获取所有 public 构造方法

Constructor[] constructors = mClass.getConstructors();

* 1. 获取本类所有构造方法

Constructor constructor = mClass.getDeclaredConstructors();

1. 获取构造方法参数

Class[] parameterTypes = constructor.getParameterTypes();

1. 实例化一个类，实参和形参必须一一对应

Constructor constructor = MyObject.class.getConstructor

MyObject myObject =(MyObject) constructor.newInstance(“constructor-arg1”);

**获取类的所有变量信息**

1. 获取并输出类的名称

Class mClass = SonClass.class;

1. 获取所有变量
   1. 获取所有public访问权限得到变量

Field[] fields = mClass.getFields();

* 1. 获取所有本类声明的变量（任何访问权限）

Field[] fields = mClass.getDeclaredFields();

//获取指定变量，参数为变量名

Field field = mClass.getDeclaredField(“xxx”);

1. 遍历变量并输出变量信息

for(Field field:fields){

//获取访问权限

int modifiers = field.getModifiers();

//变量类型

field.getType();

//变量名

field.getName();

}

**说明**

getFields() 与 getDeclaredFields()之间的区别

1. getFields() 方法，输出 SonClass 类以及其所继承的父类( 包括 FatherClass 和 Object ) 的 public 方法。**注：Object 类中没有成员变量，所以没有输出**。
2. getDeclaredFields() ， 输出 SonClass 类的所有成员变量，不问访问权限。

**获取类的所有方法信息**

1. 获取所有 public 访问权限的方法（本类及其父类）

Method[] methods = mClass.getMethods();

1. 获取所有本类的方法（任何访问权限）

Method[] methods = mClass.getDeclaredMethods();

//获取指定方法，参数为方法名和形参类型

Method method = mClass.getDeclaredMethod(“xxxx”,new Class[]{String.class});

或

Method method = mClass.getDeclaredMethod(“xxxx”, String.class);

1. 遍历所有方法

//获取方法的访问权限

int modifiers = method.getModifiers();

//获取返回值类型和名称

Class returnType = method.getRuturnType();

String typeName = returnType.getName();

//获取方法的所有参数

Parameter[] parameters = method.getParameters();

String paraTypeName = parameter.getType().getName();

//获取方法抛出的异常

Class[] exceptionTypes = method.getExceptionTypes();

String excepTypeName = exceptionType.getName();

**访问或操作类的私有变量和方法**

**访问私有方法**

1. 获取指定的私有方法

Method privateMethod =mClass.getDeclaredMethod("privateMethod", String.class, int.class);

方法参数也可以这么写：new Class[]{String,class,int.class}，没有参数就是null。

1. 获取私有方法的访问权（只是获取访问权，并不是修改实际权限）

//**如果不加会报异常 IllegalAccessException**

privateMethod.setAccessible(true);

1. 使用invoke 反射调用私有方法

//testClass 要操作的对象

//后面两个参数传实参

privateMethod.invoke(testClass,”Java Reflect”,666);

**修改私有变量**

1. 获取私有变量

Field privateField = mClass.getDeclaredField(“XXX”);

1. 获取私有变量的访问权

privateField.setAccessible(true);

1. 获取私有变量

testClass.getXXX();

1. 修改私有变量

//调用 set(object,value) 修改变量的值为 YYY

privateField.set(testClass,”YYY”);

**修改私有常量**

常量是指使用 final 修饰符修饰的成员属性，与变量的区别就在于有无 final 关键字修饰。Java 虚拟机（JVM）在编译 .java 文件得到 .class 文件时，会优化我们的代码以提升效率。其中一个优化就是：JVM 在编译阶段会把引用常量的代码替换成具体的常量值。但是，**并不是所有常量都会优化**。经测试对于 int 、long 、boolean 以及 String 这些**基本类型 JVM 会优化**，而对于 Integer 、Long 、Boolean 这种**包装类型，或者其他诸如 Date 、Object 类型**则不会被优化。

**对于基本类型的静态常量，JVM在编译阶段会把引用此常量的代码换成具体的常量值。**

针对这种问题，进行修改。

**方法一**

java允许我们声明常量时不赋值，但必须在构造函数中赋值。如：

public class TestClass{

private final String FINAL\_VALUE;

//构造函数内为常量赋值

public TestClass(){

this.FINAL\_VALUE =”FINAL”;

}

//………

}

执行修改

Field finalField = mClass.getDeclaredField(“FINAL\_VALUE”);

finalField.setAccessible(true);

//before

finalField.get(testClass);

//修改

finalField.set(testClass,”Modified”);

//after

finalField.get(testClass);

///// **修改成功**////

**说明**：

将赋值放在构造函数中，构造函数是我们运行时 new 对象才会调用的，所以不会直接为常量赋值，在编译阶段将 getFinalValue() 方法优化为返回常量值，而是指向 FINAL\_VALUE，这样在运行阶段通过反射修改的值就有意义了。

**方法二**

将声明常量的语句改为使用三目表达式赋值：

private final String FIANL\_VALUE =null == null ?”FINAL”:null;

三目表达式是在运行时刻计算出来的，编译时刻不会计算，也就不会被优化。