1、Class对象

　　理解RTTI在Java中的工作原理，首先需要知道类型信息在运行时是如何表示的，这是由Class对象来完成的，它包含了与类有关的信息。Class对象就是用来创建所有“常规”对象的，Java使用Class对象来执行RTTI，即使你正在执行的是类似类型转换这样的操作。

　　每个类都会产生一个对应的Class对象，也就是保存在.class文件。所有类都是在对其第一次使用时，动态加载到JVM的，当程序创建一个对类的静态成员的引用时，就会加载这个类。Class对象仅在需要的时候才会加载，static初始化是在类加载时进行的。

复制代码

public class TestMain {

public static void main(String[] args) {

System.out.println(XYZ.name);

}

}

class XYZ {

public static String name = "luoxn28";

static {

System.out.println("xyz静态块");

}

public XYZ() {

System.out.println("xyz构造了");

}

}

复制代码

输出结果为：

　　类加载器首先会检查这个类的Class对象是否已被加载过，如果尚未加载，默认的类加载器就会根据类名查找对应的.class文件。

　　想在运行时使用类型信息，必须获取对象(比如类Base对象)的Class对象的引用，使用功能Class.forName(“Base”)可以实现该目的，或者使用base.class。注意，有一点很有趣，使用功能”.class”来创建Class对象的引用时，不会自动初始化该Class对象，使用forName()会自动初始化该Class对象。为了使用类而做的准备工作一般有以下3个步骤：

加载：由类加载器完成，找到对应的字节码，创建一个Class对象

链接：验证类中的字节码，为静态域分配空间

初始化：如果该类有超类，则对其初始化，执行静态初始化器和静态初始化块

复制代码

public class Base {

static int num = 1;

static {

System.out.println("Base " + num);

}

}

public class Main {

public static void main(String[] args) {

// 不会初始化静态块

Class clazz1 = Base.class;

System.out.println("------");

// 会初始化

Class clazz2 = Class.forName("zzz.Base");

}

}

复制代码

2、类型转换前先做检查

　　编译器将检查类型向下转型是否合法，如果不合法将抛出异常。向下转换类型前，可以使用instanceof判断。

复制代码

class Base { }

class Derived extends Base { }

public class Main {

public static void main(String[] args) {

Base base = new Derived();

if (base instanceof Derived) {

// 这里可以向下转换了

System.out.println("ok");

}

else {

System.out.println("not ok");

}

}

}

复制代码

3、反射：运行时类信息

　　如果不知道某个对象的确切类型，RTTI可以告诉你，但是有一个前提：这个类型在编译时必须已知，这样才能使用RTTI来识别它。Class类与java.lang.reflect类库一起对反射进行了支持，该类库包含Field、Method和Constructor类，这些类的对象由JVM在启动时创建，用以表示未知类里对应的成员。这样的话就可以使用Contructor创建新的对象，用get()和set()方法获取和修改类中与Field对象关联的字段，用invoke()方法调用与Method对象关联的方法。另外，还可以调用getFields()、getMethods()和getConstructors()等许多便利的方法，以返回表示字段、方法、以及构造器对象的数组，这样，对象信息可以在运行时被完全确定下来，而在编译时不需要知道关于类的任何事情。

　　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只是简单地检查这个对象，看它属于哪个特定的类。因此，那个类的.class对于JVM来说必须是可获取的，要么在本地机器上，要么从网络获取。所以对于RTTI和反射之间的真正区别只在于：

RTTI，编译器在编译时打开和检查.class文件

反射，运行时打开和检查.class文件

复制代码

public class Person implements Serializable {

private String name;

private int age;

// get/set方法

}

public static void main(String[] args) {

Person person = new Person("luoxn28", 23);

Class clazz = person.getClass();

Field[] fields = clazz.getDeclaredFields();

for (Field field : fields) {

String key = field.getName();

PropertyDescriptor descriptor = new PropertyDescriptor(key, clazz);

Method method = descriptor.getReadMethod();

Object value = method.invoke(person);

System.out.println(key + ":" + value);

}

}

复制代码

　　以上通过getReadMethod()方法调用类的get函数，可以通过getWriteMethod()方法来调用类的set方法。通常来说，我们不需要使用反射工具，但是它们在创建动态代码会更有用，反射在Java中用来支持其他特性的，例如对象的序列化和JavaBean等。

4、动态代理

　　代理模式是为了提供额外或不同的操作，而插入的用来替代”实际”对象的对象，这些操作涉及到与”实际”对象的通信，因此代理通常充当中间人角色。Java的动态代理比代理的思想更前进了一步，它可以动态地创建并代理并动态地处理对所代理方法的调用。在动态代理上所做的所有调用都会被重定向到单一的调用处理器上，它的工作是揭示调用的类型并确定相应的策略。以下是一个动态代理示例：

接口和实现类：

复制代码

public interface Interface {

void doSomething();

void somethingElse(String arg);

}

public class RealObject implements Interface {

public void doSomething() {

System.out.println("doSomething.");

}

public void somethingElse(String arg) {

System.out.println("somethingElse " + arg);

}

}

复制代码

动态代理对象处理器：

复制代码

public class DynamicProxyHandler implements InvocationHandler {

private Object proxyed;

public DynamicProxyHandler(Object proxyed) {

this.proxyed = proxyed;

}

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException {

System.out.println("代理工作了.");

return method.invoke(proxyed, args);

}

}

复制代码

测试类：

复制代码

public class Main {

public static void main(String[] args) {

RealObject real = new RealObject();

Interface proxy = (Interface) Proxy.newProxyInstance(

Interface.class.getClassLoader(), new Class[] {Interface.class},

new DynamicProxyHandler(real));

proxy.doSomething();

proxy.somethingElse("luoxn28");

}

}

复制代码

输出结果如下：

　　通过调用Proxy静态方法Proxy.newProxyInstance()可以创建动态代理，这个方法需要得到一个类加载器，一个你希望该代理实现的接口列表(不是类或抽象类)，以及InvocationHandler的一个实现类。动态代理可以将所有调用重定向到调用处理器，因此通常会调用处理器的构造器传递一个”实际”对象的引用，从而将调用处理器在执行中介任务时，将请求转发。