**Java Class对象**

@author ixenos

关键字：RTTI、动态绑定、动态加载、获得Class引用、泛型Class引用、newInstance的坑

**RTTI和动态绑定**

 　　RTTI即运行时类型识别 Run-Time Type Identification 或 Run-Time Type Information

　　例如，当把Shape对象放入List<Shape>的数组时会向上转型，但在向上转型为Shape的时候也会丢失Shape对象的具体类型，对于数组而言，他们只是Shape对象。从List<Shape>数组中取出元素时，这种容器（实际上它将所有元素当作Object持有）会自动将结果转型回Shape，这是RTTI最基本的使用形式，因为在Java中所有的类型转换都是在运行时进行正确性检查的，而这也是RTTI名字的含义：在运行时，识别一个对象的类型。

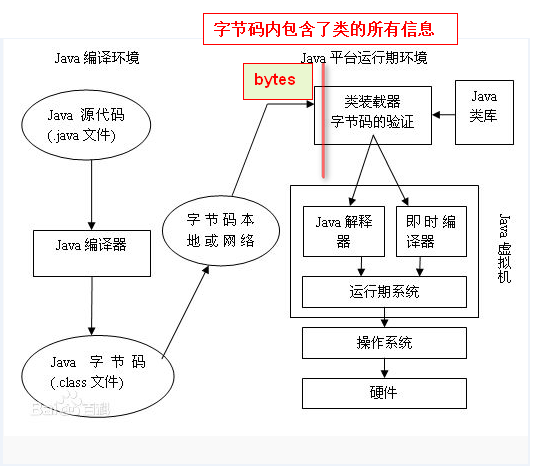
* 此例中RTTI类型转换并不彻底，这是因为目前我们只知道这个List<Shape>保存的都是Shape，这在**编译时**由容器和泛型系统来保证，在**运行时**由类型转换操作来保证
* 而类型转换后就是之前写的[多态(动态绑定)](http://www.cnblogs.com/ixenos/p/5677295.html)的事情了，因为这时候需要调用对象的方法，动态绑定确定域和方法的调用

 　　总结：**RTTI在运行时识别一个对象的类型，之后多态在运行时判断对象的实际类型来确定调用**

**Class对象和动态加载**

 　  要理解RTTI在Java中的工作原理，必须先知道类型信息在运行时是如何表示的，而这项工作由**Class对象的特殊对象**完成

　　Class对象就是用来创建类的所有的常规对象的，每个类都有一个Class对象，每当编译了一个新类就会产生一个Class对象（被保存在一个.class文件中），如下图所示：



**类加载器子系统：**为了生成类的对象，JVM使用**类加载器子系统**(Classloader)来加载**.class文件（内容是字节码）**

**类加载器链：**该子系统通常可以包含一条**类加载器链，**但只有一个**原生类加载器，它（指子系统）是JVM实现的一部分**

**原生类加载器：**原生类加载器加载的是所谓的**可信类，他们通常从本地盘加载的(包括Java API类)**

**额外的类加载器：**在这条链中通常不需要添加**额外的类加载器**，但如果由特殊要求（以某种特殊的方式加载类，以支持Web服务器应用，或者在网络中下载类）

**动态加载：**

　　　　所有类(XXX.class)都是在对其第一次使用时，动态加载到JVM中的。**当程序创建第一个对类的静态对象成员的引用时，就会加载这个类**（比如调用静态方法、静态域）

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

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 main(String[] args){

15 System.out.println("main");

16 new Candy();

17 System.out.println("After creating Candy");

18 try{

19 Class.forName("Gum");

20 }catch(ClassNotFoundException e){

21 System.out.println("Gum not founded");

22 }

23 System.out.println("After creating Gum");

24 new Cookie();

25 System.out.println("After creating Cookie");

26 }

27 }

28

29 ----------------------------------

30 输出：

31 main

32 Loading Candy

33 After creating Candy

34 Loading Gum

35 After creating Gum

36 Loading Cookie

37 After creating Cookie

[复制代码](javascript:void(0);)

　　　　　　从输出看出：Class对象仅在需要的时候才被加载，static初始化是在类加载时进行的

　　　　　　对Class.forName()的调用结果说明：如果类Gum还没被加载就加载它，那么在加载的过程中，Gum的static子句被执行

　　　　而当只是由构造器构造对象时，类就加载，这说明了**构造器也是类的静态方法**，虽然没有static关键字修饰。因此，使用new 操作符创建类的新对象也会被当作对类的静态成员的引用

 　　　  因此，**Java程序（包含很多类）在它开始运行之前并非被完全加载，其各个部分（.class）是在必需时才加载的**

**类加载器对于动态加载：**

　　　　类加载器首先检查这个类的Class对象是否已经加载。

**如果未加载**，默认的类加载器由类型查找.class文件（例如某个额外的类加载器可能会在数据库中查找字节码（.class文件的内容））

**如果已加载**，则Class对象将接受验证，确保没被破坏且不包含不良代码（Java的安全防范措施）

**一旦某个类的Class对象被载入内存，它就被用来创建这个类的所有对象**

**获得Class对象的引用**

**获得Class对象引用的两种方式：**

**1.使用方法**

**只知道对应类型名时，**使用Class.forName(String name)

**持有对应类型对象的引用时，**使用该对象的getClass()（属于根类Object的一部分）

**2.使用类字面常量**

　　例如 Fancy.class

　　类字面常量应用于：普通类、接口、数组、基本数据类型（对于包装类，还有标准字段TYPE，这是个引用，指向基本数据类型的Class对象，例如boolean.class等价于Boolean.TYPE）

　　使用".class"来创建Class对象的引用时，**不会自动地初始化该Class对象**

**补充**：为了使用类而做的准备工作包括三个步骤：

　　1.加载：类加载器查找字节码，从字节码创建一个Class对象

　　2.链接：验证字节码，为静态域分配存储空间，解析此类对其他类的所有引用

　　3.初始化：若该类有超类，对其初始化，执行静态初始化器和静态初始化块

　　可以看到，**初始化被延迟到了对静态方法（构造器是隐式静态）或者非常数(final)静态域进行首次引用时才执行**

**\*\*\***static final int = 47 是**编译期常量，不需要对类进行初始化就可以读取**

　　　　\*\*\*static final int = Random.nextInt(100)  是运行时常量，不是编译期常量

　　　　\*\*\*static int = 47 是静态域而已，不是常量，更不是编译期常量

**泛型化的Class引用**

**作用：**

1.对Class引用所指向的Class对象的类型进行限定（也就是说类型参数表示的就是类的类型，比如Class<T> = T.class）

2.让编译器强制执行额外的类型检查

**通配符：**

而为了在使用泛化的Class引用时放松限制，使用通配符，这是Java泛型的一部分

1.**Class<?>**：优于平凡的Class，即使他们是等价的，Class<?>若使用了一个非具体的引用，编译器将警报

2.**Class<? extends T>**：为了创建一个限定为某种类型(或该类型的子类型)的Class引用，使用通配符与extends关键字结合创建一个范围

[复制代码](javascript:void(0);)

1 public class BoundClass{

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

3 Class<? extends Number> bound = int.class;

4 bound = double.class; //可引用该Class对象

5 bound = Number.class;　//可引用该Class对象

6 }

7 }

[复制代码](javascript:void(0);)

3.**Class的newInstance()**方法：

　　1.普通类Class对象的newInstance方法，返回Object类型的对象

　　2.普通泛型类Class对象的newInstance方法，返回确切类型的对象

　　3.**坑**：如果手头的是超类，即通配符类型由super修饰<? super FancyToy>，返回Object类型的对象

　　　　因为**编译器**将只允许你声明的**超类引用是“某个类，它是FancyToy的超类”，这种含糊性的定义**，所以只能返回Object保证安全

1 Class<? super FancyToy> up = ftClass.getSuperclass();

2 //下面编译不通过:

3 //Class<Toy> up2 = ftClass.getSuperclass();

4

5 //只能返回Object

6 Object obj = up.newInsance();

 　　　　编译不通过的是因为getSuperclass返回的是**超类引用，必须是个含糊的定义，由Class<? super T>接受**

**getSuperclass**

public Class<? super [T](http://www.cnblogs.com/)> **getSuperclass**()

返回表示此 Class 所表示的实体（类、接口、基本类型或 void）的超类的 Class。如果此 Class 表示Object 类、一个接口、一个基本类型或 void，则返回 null。如果此对象表示一个数组类，则返回表示该Object 类的 Class 对象。

**返回：**

此对象所表示的类的超类。

**Class对象的常用方法**

 　　常用方法：

　　1.Class.forName(String name) String必须使用全限定名（包括包名）

[复制代码](javascript:void(0);)

1 Class c = null;

2 try{

3 c = Class.forName("com.xenos.jxen"); //使用全限定名

4 }catch(ClassNotFoundException e){

5 System.out.println("can not find jxen");

6 System.exit(1);

7 }

[复制代码](javascript:void(0);)

　　2.Class的getInterfaces() 返回Class对象，表示在感兴趣的Class对象中包含的接口

　　3.Class的getSuperClass() 查询Class对象的直接基类，如果使用泛型不能用Class<T>类型接受，要用Class<? super T>类型来接受，返回你可以用来进一步查询的Class对象，因此可以在运行时发现一个对象完整的继承结构

　　4.Class的newInstance() 江湖人称“虚拟构造器”，允许你声明：“我不知道你的确切类型，但是无论如何都要你正确创建自己”，另外创建的类要有默认构造器（无参构造器）