[键入公司名称]

2014

Java文档

java

Me

[键入公司地址]

[一、 java编译与查看运行时内部结构 3](#_Toc485225633)

[二、 Java基本知识 4](#_Toc485225634)

[(一) java数据类型 4](#_Toc485225635)

[(二) 八种基本类型+void 4](#_Toc485225636)

[(三) Java引用类型 7](#_Toc485225637)

[(四) 类型转换 7](#_Toc485225638)

[(五) JVM的内存空间： 10](#_Toc485225639)

[三、 java核心机制 11](#_Toc485225640)

[(一) 虚拟机体系 11](#_Toc485225641)

[1) 数据类型 11](#_Toc485225642)

[2) 规格描述 12](#_Toc485225643)

[3) JVM碎片回收堆（无用单元收集堆） 14](#_Toc485225644)

[4) JVM存储区 14](#_Toc485225645)

[(二) 垃圾回收 15](#_Toc485225646)

[四、 java初始化顺序 16](#_Toc485225647)

[五、 java序列化与反序列化 17](#_Toc485225648)

[六、 java遍历 19](#_Toc485225649)

[(三) java遍历：for、do while、while、foreach、iterator的用法和区别 19](#_Toc485225650)

[(四) java遍历集合 20](#_Toc485225651)

[七、 java泛型 22](#_Toc485225652)

[(一) 泛型类： 22](#_Toc485225653)

[(二) 泛型方法： 23](#_Toc485225654)

[八、 java类型之间的关系判断（instanceof、isAssignableFrom） 25](#_Toc485225655)

[5) instanceof 25](#_Toc485225656)

[6) isAssignableFrom 25](#_Toc485225657)

[九、 Java知识外延 26](#_Toc485225658)

[(一) java.lang.Class类 26](#_Toc485225659)

[1) getName(),getSimpleName(),getCanonicalName ()三者之间的区别 26](#_Toc485225660)

[(二) 数据库访问部分 26](#_Toc485225661)

[1) javax.persistence.EntityManager; 26](#_Toc485225662)

[2) javax.persistence.Query 27](#_Toc485225663)

[3) 关于sql语句的说明 27](#_Toc485225664)

[4) 注意事项 27](#_Toc485225665)

# java编译与查看运行时内部结构

注：TestLambda.java 文件编写TestLambda类，并且其包名为package com.test;

编译：

1. 直接编译

javac com/test/TestLambda.java

1. 指定编码

javac -encoding UTF-8 com/test/TestLambda.java

查看运行时内部结构：

1. 直接反编译查看结构

javap -p com/test/TestLambda.class

1. 输出运行时生成的内部类和方法

java -Djdk.internal.lambda.dumpProxyClasses com/test/TestLambda

# Java基本知识

## java数据类型

数据类型

基本数据类型

引用数据类型

类(class)

接口(interface)

数组

注解

布尔型(boolean)

字符型(char)

数值型

整数类型（byte, short, int, long）

浮点类型（float, double）

returnAddress

注：数值类型不存在无符号的，它们的取值范围是固定的，不会随着机器硬件环境或者操作系统的改变而改变。

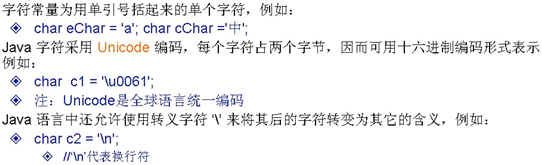
**returnAddress:用来实现Java程序中的finally子句**，Java程序员不能使用这个类型，它的值指向一条虚拟机指令的操作码

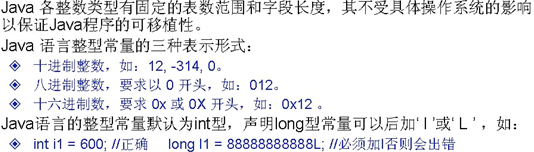
实际上，JAVA中还存在另外一种基本类型void，它也有对应的封装类 java.lang.Void，不过我们无法直接对它们进行操作。

## 八种基本类型+void

* 八种类型+void表示范围如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 封装类 | 存储空间 | 范围/说明 |
| boolean | Boolean | 不确定 | true或false；  Boolean. TRUE; Boolean.FALSE;  关于存储空间：理论值为1bit;  实际:boolean b; 4字节,等同于int;  boolean[] b: 8bit,等同于byte |
| char | Character | 2字节,16位  Character.SIZE | '\u0000'~'\uFFFF'；  Character. MIN\_VALUE; Character. MAX\_VALUE; |
| byte | Byte | 1字节,8位  Byte.SIZE | [-2^7,2^7-1]；  即：-128 ~ 127  Byte. MIN\_VALUE; Byte. MAX\_VALUE; |
| short | Short | 2字节,16位  Short.SIZE | [-2^15,2^15-1]；  即：-32768 ~ 32767  Short. MIN\_VALUE; Short. MAX\_VALUE; |
| int | Integer | 4字节,32位  Integer.SIZE | [-2^31,2^31-1]；  即：-2147483648 ~ 2147483647 共10位  Integer. MIN\_VALUE; Integer. MAX\_VALUE;  十六进制整型常量：以十六进制表示时，需以0x或0X开头，如0xff,0X9A。  八进制整型常量：八进制必须以0开头，如0123，034 |
| long | Long | 8字节,64位  Long.SIZE | [-2^63,2^63-1]；  即：-9223372036854775808到9223372036854775807共19位  Long. MIN\_VALUE; Long. MAX\_VALUE;  长整型必须以l/L作结尾，如9L,342L。 |
| float | Float | 4字节,32位  Float.SIZE | Float. MIN\_VALUE; Float. MAX\_VALUE;  数值必须以f/F做结尾 |
| double | Double | 8字节,64位  Double.SIZE | Double. MIN\_VALUE; Double. MAX\_VALUE;  小数可以以d/D结尾，默认类型，可以忽略 |
| void | Void |  |  |





注意：float、double两种类型的最小值与Float.MIN\_VALUE、 Double.MIN\_VALUE的值并不相同，实际上Float.MIN\_VALUE和Double.MIN\_VALUE分别指的是 float和double类型所能表示的最小正数。也就是说存在这样一种情况，0到±Float.MIN\_VALUE之间的值float类型无法表示，0 到±Double.MIN\_VALUE之间的值double类型无法表示。这并没有什么好奇怪的，因为这些范围内的数值超出了它们的精度范围。

在数学中0到1有无数个浮点数；而计算机是离散的，所以表示的时候有误差，计算机用精度（小数点后几位来表示正确），比较浮点数时a==0.1是不合适的，应该a-0.1==0;如果a是0.1,则即使有误差 a-0.1==0因为a和0.1都被表示为一个有误差的计算机二进制;

float与double表面相同的值，未必相等;

比如

0.1f == 0.1d ，返回false

1.1f == 1.1d ，返回false

1f == 1d ，返回true

Float和Double的最小值和最大值都是以科学记数法的形式输出的，结尾的"E+数字"表示E之前的数字要乘以10的多少倍。比如3.14E3就是3.14×1000=3140，3.14E-3就是3.14/1000=0.00314。

Java基本类型存储在栈中，因此它们的存取速度要快于存储在堆中的对应封装类的实例对象。从Java5.0（1.5）开始，JAVA虚拟机（Java Virtual Machine）可以完成基本类型和它们对应封装类之间的自动转换。因此我们在赋值、参数传递以及数学运算的时候像使用基本类型一样使用它们的封装类，但这并不意味着你可以通过基本类型调用它们的封装类才具有的方法。另外，所有基本类型（包括void）的封装类都使用了final修饰，因此我们无法继承它们扩展新的类，也无法重写它们的任何方法。

* 基本类型与封装类型优缺点

基本类型的优势：数据存储相对简单，运算效率比较高

封装类的优势：有的容易，比如集合的元素必须是对象类型，满足了java一切皆是对象的思想

* 一些常用的转义字符：

①\r表示接受键盘输入，相当于按下了回车键；

②\n表示换行；

③\t表示制表符，相当于Table键；

④\b表示退格键，相当于Back Space键；

⑤\'表示单引号；

⑥\''表示双引号；

⑦\\表示一个斜杠\。

数据类型之间的转换

1).简单类型数据间的转换,有两种方式:自动转换和强制转换,通常发生在表达式中或方法的参数传递时。

## Java引用类型

Java有 5种引用类型（对象类型）：**类、接口、数组、枚举、注解**

引用类型：底层结构和基本类型差别较大

## 类型转换

这里我们所说容量的"大"与"小",并不是指占用字节的多少,而是指表示值的范围的大小。

遵循原则：

* 值范围小的类型自动转换为值范围大的数据类型：数据类型按照值范围大小排序为

**(byte，short，char)->int->long->float->double**

* byte，short，char之间不会互相转换，他们之间的计算会首先转为int类型
* 值范围大的数据类型转换为值范围小的类型时，需加强制转换符，但可能造成精度降低或溢出；
* 有多种类型的数据混合计算时，系统会首先自动第将所有的数据转换成值范围最大的那一种，然后在进行计算;
* 方法调用时,实际参数较"小",而被调用的方法的形式参数数据又较"大"时(若有匹配的,当然会直接调用匹配的方法),系统也将自动将"小"数据转换成"大"数据,再进行方法的调用,自然,对于多个同名的重载方法,会转换成最"接近"的"大"数据并进行调用

①下面的语句可以在Java中直接通过：

byte b;int i=b; long l=b; float f=b; double d=b;

②如果低级类型为char型，向高级类型（整型）转换时，会转换为对应ASCII码值，例如

char c='c'; int i=c;

System.out.println("output:"+i);输出：output:99;

③对于byte,short,char三种类型而言，他们是平级的，因此不能相互自动转换，可以使用下述的强制类型转换。

short i=99 ; char c=(char)i; System.out.println("output:"+c);输出：output:c;

强制转换

将"大"数据转换为"小"数据时，你可以使用强制类型转换。即你必须采用下面这种语句格式： int n=(int)3.14159/2;可以想象，这种转换肯定可能会导致溢出或精度的下降。

2)表达式的数据类型自动提升, 关于类型的自动提升，注意下面的规则。

①所有的byte,short,char型的值将被提升为int型；

②如果有一个操作数是long型，计算结果是long型；

③如果有一个操作数是float型，计算结果是float型；

④如果有一个操作数是double型，计算结果是double型；

例， byte b; b=3; b=(byte)(b\*3);//必须声明byte。

3)封装类过渡类型转换

一般情况下，我们首先声明一个变量，然后生成一个对应的封装类，就可以利用封装类的各种方法进行类型转换了。例如：

①当希望把float型转换为double型时：

float f1=100.00f;

Float F1=new Float(f1);

double d1=F1.doubleValue();//F1.doubleValue()为Float类的返回double值型的方法

②当希望把double型转换为int型时：

double d1=100.00;

Double D1=new Double(d1);

int i1=D1.intValue();

简单类型的变量转换为相应的封装类，可以利用封装类的构造函数。即：Boolean(boolean value)、Character(char value)、Integer(int value)、Long(long value)、Float(float value)、Double(double value)

而在各个封装类中，总有形为××Value()的方法，来得到其对应的简单类型数据。利用这种方法，也可以实现不同数值型变量间的转换，例如，对于一个双精度实型类，intValue()可以得到其对应的整型变量，而doubleValue()可以得到其对应的双精度实型变量。

4)字符串与其它类型间的转换

其它类型向字符串的转换

①调用类的串转换方法:X.toString();

②自动转换:X+"";

③使用String的方法:String.volueOf(X);

字符串作为值,向其它类型的转换

①先转换成相应的封装器实例,再调用对应的方法转换成其它类型

例如，字符中"32.1"转换double型的值的格式为:new Float("32.1").doubleValue()。也可以用:Double.valueOf("32.1").doubleValue()

②静态parseXXX方法

String s = "1";

byte b = Byte.parseByte( s );

short t = Short.parseShort( s );

int i = Integer.parseInt( s );

long l = Long.parseLong( s );

Float f = Float.parseFloat( s );

Double d = Double.parseDouble( s );

③Character的getNumericValue(char ch)方法

5）Date类与其它数据类型的相互转换

整型和Date类之间并不存在直接的对应关系，只是你可以使用int型为分别表示年、月、日、时、分、秒，这样就在两者之间建立了一个对应关系，在作这种转换时，你可以使用Date类构造函数的三种形式：

①Date(int year, int month, int date)：以int型表示年、月、日

②Date(int year, int month, int date, int hrs, int min)：以int型表示年、月、日、时、分

③Date(int year, int month, int date, int hrs, int min, int sec)：以int型表示年、月、日、时、分、秒

在长整型和Date类之间有一个很有趣的对应关系，就是将一个时间表示为距离格林尼治标准时间1970年1月1日0时0分0秒的毫秒数。对于这种对应关系，Date类也有其相应的构造函数：Date(long date)。

获取Date类中的年、月、日、时、分、秒以及星期你可以使用Date类的getYear()、getMonth()、getDate()、getHours()、getMinutes()、getSeconds()、getDay()方法，你也可以将其理解为将Date类转换成int。

而Date类的getTime()方法可以得到我们前面所说的一个时间对应的长整型数，与封装类一样，Date类也有一个toString()方法可以将其转换为String类。

有时我们希望得到Date的特定格式，例如20020324，我们可以使用以下方法，首先在文件开始引入，

import java.text.SimpleDateFormat;

import java.util.\*;

java.util.Date date = new java.util.Date();

//如果希望得到YYYYMMDD的格式

SimpleDateFormat sy1=new SimpleDateFormat("yyyyMMDD");

String dateFormat=sy1.format(date);

//如果希望分开得到年，月，日

SimpleDateFormat sy=new SimpleDateFormat("yyyy");

SimpleDateFormat sm=new SimpleDateFormat("MM");

SimpleDateFormat sd=new SimpleDateFormat("dd");

String syear=sy.format(date);

String smon=sm.format(date);

String sday=sd.format(date);

总结：只有boolean不参与数据类型的转换

（1）.自动类型的转换：

a.常数在表数范围内是能够自动类型转换的

b.数据范围小的能够自动数据类型大的转换（注意特例）

int到float，long到float，long到double 是不会自动转换的，不然将会丢失精度

c.引用类型能够自动转换为父类的

d.基本类型和它们封装类型是能够互相转换的

（2）.强制类型转换：用圆括号括起来目标类型，置于变量前

## JVM的内存空间：

（1）. Heap 堆空间：分配对象 new Student（）

（2）. Stack 栈空间：临时变量 Student stu

（3）.Code 代码区 ：类的定义，静态资源 Student.class

eg：Student stu = new Student（）； //new 在内存的堆空间创建对象

stu.study(); //把对象的地址赋给stu引用变量

上例实现步骤：

a.JVM加载Student.class 到Code区

b.new Student()在堆空间分配空间并创建一个Student实例

c.将此实例的地址赋值给引用stu， 栈空间

# java核心机制

## 数据类型

* Java虚拟机支持Java语言的基本数据类型有8种，注意String不是基本数据类型如下：

boolean: //1字节有符号整数的补码

byte: //1字节有符号整数的补码

short: //2字节有符号整数的补码

int: //4字节有符号整数的补码

long: //8字节有符号整数的补码

float: //4字节IEEE754单精度浮点数

double: //8字节IEEE754双精度浮点数

char: //2字节无符号Unicode字符

**几乎所有的Java类型检查都是在编译时完成的**。上面列出的原始数据类型的数据在Java执行时不需要用硬件标记。操作这些原始数据类型数据的字节码（指令）本身就已经指出了操作数的数据类型，例如iadd、ladd、fadd和dadd指令都是把两个数相加，其操作数类型别是int、long、float和double。**虚拟机没有给boolean（布尔）类型设置单独的指令。boolean型的数据是由integer指令，包括integer返回来处理的。boolean型的数组则是用byte数组来处理的**。虚拟机使用IEEE754格式的浮点数。不支持IEEE格式的较旧的计算机，在运行Java数值计算程序时，可能会非常慢。

* 虚拟机支持的其它数据类型包括：

object //对一个Javaobject（对象）的4字节引用

returnAddress //4字节，用于jsr/ret/jsr-w/ret-w指令

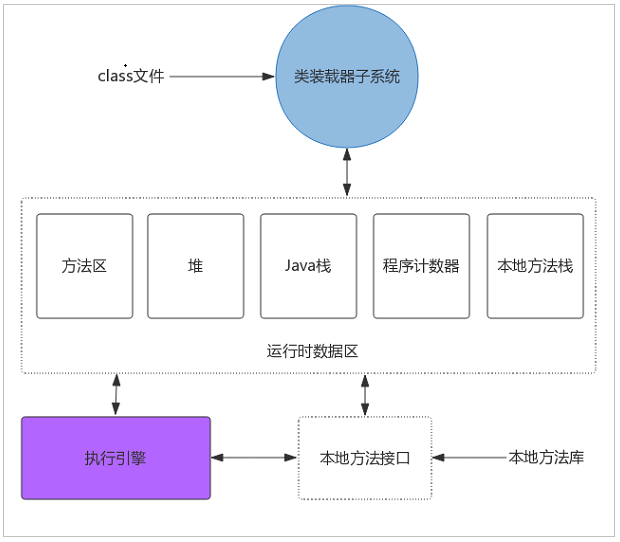
* 注：Java数组被当作object处理。

虚拟机的规范对于object内部的结构没有任何特殊的要求。**在Sun公司的实现中，对object的引用是一个句柄，其中包含一对指针：一个指针指向该object的方法表，另一个指向该object的数据。用Java虚拟机的字节码表示的程序应该遵守类型规定**。Java虚拟机的实现应拒绝执行违反了类型规定的字节码程序。

* 注：Java虚拟机由于字节码定义的限制似乎只能运行于32位地址空间的机器上。但是可以创建一个Java虚拟机，它自动地把字节码转换成64位的形式。从Java虚拟机支持的数据类型可以看出，Java对数据类型的内部格式进行了严格规定，这样使得各种Java虚拟机的实现对数据的解释是相同的，从而保证了Java的与平台无关性和可移植性。

## java虚拟机体系结构

在 Java虚拟机规范中，一个虚拟机实例的行为是分别按照子系统、内存区、数据类型和指令来描述的，这些组成部分一起展示了抽象的虚拟机的内部体系结构。



### class文件

Java class文件包含了关于类或接口的所有信息。class文件的“基本类型”如下：

|  |  |
| --- | --- |
| u1 | 1个字节，无符号类型 |
| u2 | 2个字节，无符号类型 |
| u4 | 4个字节，无符号类型 |
| u8 | 8个字节，无符号类型 |

class文件包含的内容：

ClassFile {

u4 magic; //魔数：0xCAFEBABE，用来判断是否是Java class文件

u2 minor\_version; //次版本号

u2 major\_version; //主版本号

u2 constant\_pool\_count; //常量池大小

u2 access\_flags; //类和接口层次的访问标志（通过|运算得到）

u2 this\_class; //类索引（指向常量池中的类常量）

u2 super\_class; //父类索引（指向常量池中的类常量）

u2 interfaces\_count; //接口索引计数器

u2 interfaces[interfaces\_count]; //接口索引集合

u2 fields\_count; //字段数量计数器

field\_info fields[fields\_count]; //字段表集合

u2 methods\_count; //方法数量计数器

u2 attributes\_count; //属性个数

method\_info methods[methods\_count]; //方法表集合

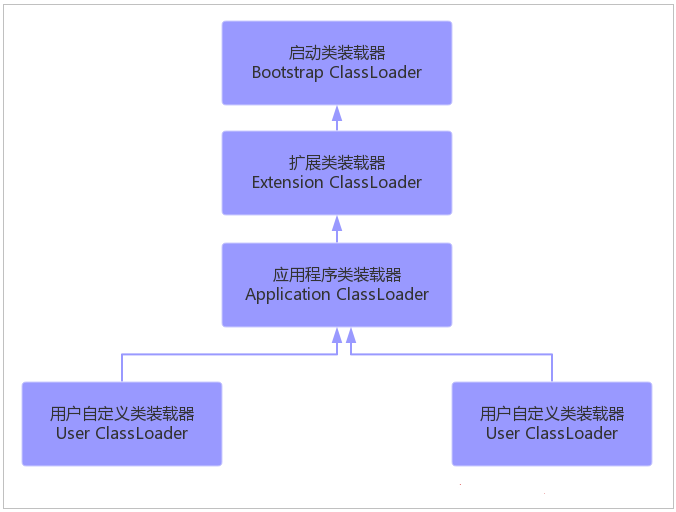
attribute\_info attributes[attributes\_count]; //属性表

cp\_info constant\_pool[constant\_pool\_count-1]; //常量池

}

### 类装载器子系统

类装载器子系统负责查找并装载类型信息。其实Java虚拟机有两种类装载器：系统装载器和用户自定义装载器。前者是Java虚拟机实现的一部分，后者则是Java程序的一部分。



* 启动类装载器（bootstrap class loader）：它用来加载 Java 的核心库，是用原生代码来实现的，并不继承自java.lang.ClassLoader。
* 扩展类装载器（extensions class loader）：它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
* 应用程序类装载器（application class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。
* 除了系统提供的类装载器以外，开发人员可以通过继承 java.lang.ClassLoader类的方式实现自己的类装载器，以满足一些特殊的需求。

　　类装载器子系统涉及Java虚拟机的其它几个组成部分以及来自java.lang库的类。ClassLoader定义的方法为程序提供了访问类装载器机制的接口。此外，对于每一个被装载的类型，Java虚拟机都会为它创建一个java.lang.Class类的实例来代表该类型。和其它对象一样，用户自定义的类装载器以及Class类的实例放在内存中的堆区，而装载的类型信息则位于方法区。

　　类装载器子系统除了要定位和导入二进制class文件外，还必须负责验证被导入类的正确性，为类变量分配并初始化内存，以及解析符号引用。这些动作还需要按照以下顺序进行：

1. 装载（查找并装载类型的二进制数据）
2. 连接（执行验证：确保被导入类型的正确性；准备：为类变量分配内存，并将其初始化为默认值；解析：把类型中的符号引用转换为直接引用）
3. 初始化（类变量初始化为正确初始值）

### 方法区

在Java虚拟机中，关于被装载的类型信息存储在一个方法区的内存中。当虚拟机装载某个类型时，它使用类装载器定位相应的class文件，然后读入这个class文件并将它传输到虚拟机中，接着虚拟机提取其中的类型信息，并将这些信息存储到方法区。**方法区也可以被垃圾回收器收集**，因为虚拟机允许通过用户定义的类装载器来动态扩展Java程序。

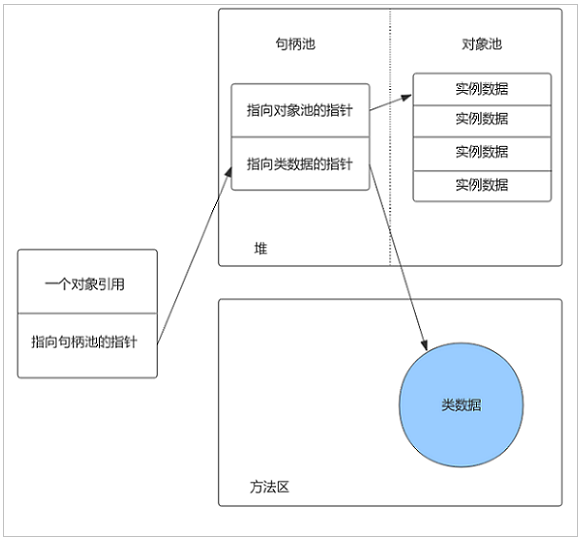
注：方法区中存放了以下信息：

* 类型的全限定名（如全限定名java.lang.Object）
* 类型的直接父类的全限定名
* 任何直接父接口的全限定名的有序列表
* 类型是类类型还是接口类型
* 类型的访问修饰符（public, abstract, final的某个子集）
* 该类型的常量池（一个有序集合，包括直接常量[string, integer和floating point常量]和对其它类型、字段和方法的符号引用）
* 除了常量以外的所有类（静态）变量
* 字段信息（字段名、类型、修饰符）
* 方法信息（方法名、返回类型、参数数量和类型、修饰符）
* 指向ClassLoader类的引用（每个类型被装载时，虚拟机必须跟踪它是由启动类装载器还是由用户自定义类装载器装载的）
* 指向Class类的引用（**对于每一个被装载的类型，虚拟机相应地为它创建一个java.lang.Class类的实例**。比如你有一个到java.lang.Integer类的对象的引用，那么只需要调用Integer对象引用的getClass()方法，就可以得到表示java.lang.Integer类的Class对象）

### 堆

Java程序在运行时创建的所有**类实例或数组**（数组在Java虚拟机中是一个真正的对象）都放在**同一个堆中**。由于**Java虚拟机实例只有一个堆空间，所以所有线程都将共享这个堆**。需要注意的是，**Java虚拟机有一条在堆中分配对象的指令，却没有释放内存的指令，因为虚拟机把这个任务交给垃圾收集器处理**。Java虚拟机规范并没有强制规定垃圾收集器，它只要求虚拟机实现必须“以某种方式”管理自己的堆空间。比如某个实现可能只有固定大小的堆空间，当空间填满，它就简单抛出OutOfMemory异常，根本不考虑回收垃圾对象的问题，但却是符合规范的。

Java虚拟机规范并没有规定Java对象在堆中如何表示，这给虚拟机的实现者决定怎么设计。一个可能的堆设计如下：



一个句柄池，一个对象池。一个对象的引用就是一个指向句柄池的本地指针。这种设计的好处有利于堆碎片的整理，当移动对象池中的对象时，句柄部分只需更改一下指针指向对象的新地址即可。缺点是每次访问对象的实例变量都要经过两次指针传递。

### Java栈

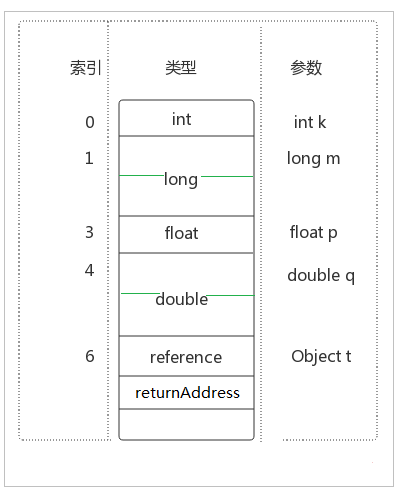
每当启动给一个线程时，Java虚拟机会为它分配一个Java栈。Java栈由许多栈帧组成，一个栈帧包含一个Java方法调用的状态。当线程调用一个Java方法时，虚拟机压入一个新的栈帧到该线程的Java栈中，当该方法返回时，这个栈帧就从Java栈中弹出。Java栈存储线程中Java方法调用的状态--包括局部变量、参数、返回值以及运算的中间结果等。Java虚拟机没有寄存器，其指令集使用Java栈来存储中间数据。这样设计的原因是为了保持Java虚拟机的指令集尽量紧凑，同时也便于Java虚拟机在只有很少通用寄存器的平台上实现。另外，基于栈的体系结构，也有助于运行时某些虚拟机实现的动态编译器和即时编译器的代码优化。

1. **栈帧**

**栈帧**由**局部变量区**、**操作数栈**和 **帧数据区**组成。当虚拟机调用一个Java方法时，它从**对应类的类型信息中**得到此方法的**局部变量区**和**操作数栈**的大小，并根据此分配**栈帧**内存，然后压入Java栈中。

1. 局部变量区

　　局部变量区被组织为以字长为单位、从0开始计数的数组。字节码指令通过从0开始的索引使用其中的数据。**类型为int, float, reference和returnAddress的值在数组中占据一项，而类型为byte, short和char的值在存入数组前都被转换为int值，也占据一项。但类型为long和double的值在数组中却占据连续的两项。**



1. 操作数栈

　　和局部变量区一样，操作数栈也是被组织成一个以字长为单位的数组。它通过标准的栈操作访问--压栈和出栈。由于程序计数器无法被程序指令直接访问，Java虚拟机的指令是从操作数栈中取得操作数，所以它的运行方式是基于栈而不是基于寄存器。虚拟机把操作数栈作为它的工作区，因为大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈。

1. 帧数据区

　　除了局部变量区和操作数栈，Java栈帧还需要**帧数据区来支持常量池解析、正常方法返回以及异常派发机制**。每当虚拟机要执行某个需要用到常量池数据的指令时，它会通过帧数据区中指向常量池的指针来访问它。除了常量池的解析外，帧数据区还要帮助虚拟机处理Java方法的正常结束或异常中止。如果通过return正常结束，虚拟机必须恢复发起调用的方法的栈帧，包括设置程序计数器指向发起调用方法的下一个指令；如果方法有返回值，虚拟机需要将它压入到发起调用的方法的操作数栈。为了处理Java方法执行期间的异常退出情况，帧数据区还保存一个对此方法异常表的引用。

### 程序计数器

对于一个运行中的Java程序而言，每一个线程都有它的程序计数器。程序计数器也叫PC寄存器。程序计数器既能持有一个本地指针，也能持有一个returnAddress。当线程执行某个Java方法时，程序计数器的值总是下一条被执行指令的地址。这里的地址可以是一个本地指针，也可以是方法字节码中相对该方法起始指令的偏移量。如果该线程正在执行一个本地方法，那么此时程序计数器的值是“undefined”。

### 本地方法栈

　　任何本地方法接口都会使用某种本地方法栈。**当线程调用Java方法时，虚拟机会创建一个新的栈帧并压入Java栈。当它调用的是本地方法时，虚拟机会保持Java栈不变，不再在线程的Java栈中压入新的栈**，虚拟机只是简单地动态连接并直接调用指定的本地方法。

其中方法区和堆由该虚拟机实例中所有线程共享。当虚拟机装载一个class文件时，它会从这个class文件包含的二进制数据中解析类型信息，然后把这些类型信息放到方法区。当程序运行时，虚拟机会把所有该程序在运行时创建的对象放到堆中。

像其它运行时内存区一样，本地方法栈占用的内存区可以根据需要动态扩展或收缩。

### 执行引擎

　　在Java虚拟机规范中，执行引擎的行为使用指令集定义。实现执行引擎的设计者将决定如何执行字节码，实现可以采取解释、即时编译或直接使用芯片上的指令执行，还可以是它们的混合。

　　执行引擎可以理解成一个抽象的规范、一个具体的实现或一个正在运行的实例。抽象规范使用指令集规定了执行引擎的行为。具体实现可能使用多种不同的技术--包括软件方面、硬件方面或树种技术的结合。作为运行时实例的执行引擎就是一个线程。

　　运行中Java程序的每一个线程都是一个独立的虚拟机执行引擎的实例。从线程生命周期的开始到结束，它要么在执行字节码，要么执行本地方法。

### 指令集

　　方法的字节码流由Java虚拟机的指令序列构成。每一条指令包含一个单字节的操作码，后面跟随0个或多个操作数。操作码表示需要执行的操作；操作数向Java虚拟机提供执行操作码需要的额外信息。当虚拟机执行一条指令时，可能使用当前常量池中的项、当前帧的局部变量中的值或者位于当前帧操作数栈顶端的值。

　　抽象的执行引擎每次执行一条字节码指令。Java虚拟机中运行的程序的每个线程（执行引擎实例）都执行这个操作。执行引擎取得操作码，如果操作码有操作数，就取得它的操作数。它执行操作码和跟随的操作数规定的动作，然后再取得下一个操作码。这个执行字节码的过程在线程完成前将一直持续，通过从它的初始方法返回，或者没有捕获抛出的异常都可以标志着线程的完成。

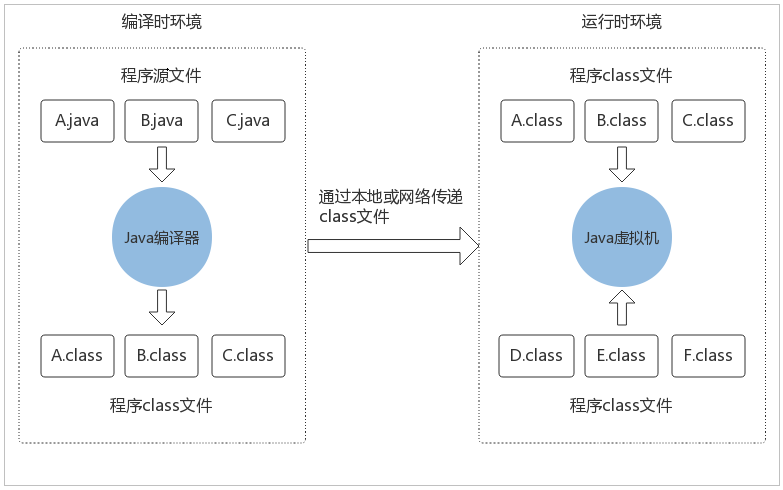
### 本地方法接口

　　Java本地接口，也叫JNI（Java Native Interface），是为可移植性准备的。本地方法接口允许本地方法完成以下工作：

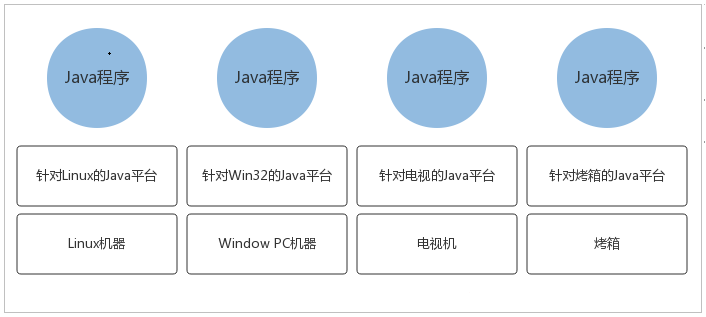
* 传递或返回数据
* 操作实例变量
* 操作类变量或调用类方法
* 操作数组
* 对堆的对象加锁
* 装载新的类
* 抛出异常
* 捕获本地方法调用Java方法抛出的异常
* 捕获虚拟机抛出的异步异常
* 指示垃圾收集器某个对象不再需要

## Java程序执行流程

　　Java程序的执行依赖于编译环境和运行环境。源码代码转变成可执行的机器代码，由下面的流程完成：

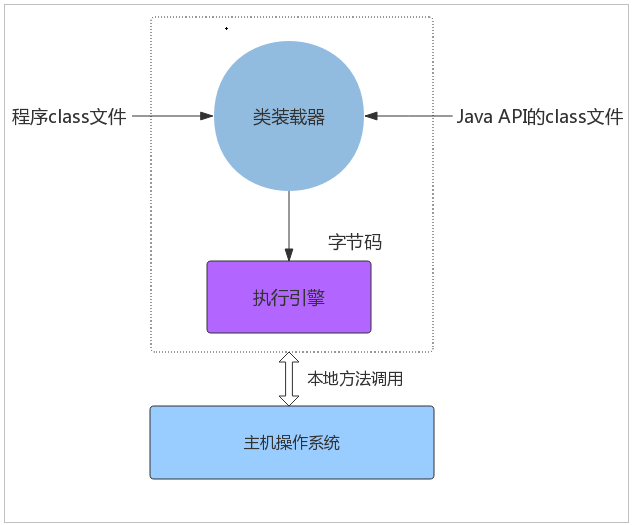


Java技术的核心就是Java虚拟机，因为所有的Java程序都在虚拟机上运行。Java程序的运行需要Java虚拟机、Java API和Java Class文件的配合。Java虚拟机实例负责运行一个Java程序。当启动一个Java程序时，一个虚拟机实例就诞生了。当程序结束，这个虚拟机实例也就消亡。



Java的跨平台特性，因为它有针对不同平台的虚拟机。

程序装载过程，如下图：



当Java虚拟机由主机操作系统上的软件实现时，Java程序通过调用本地方法和主机进行交互。Java方法由Java语言编写，编译成字节码，存储在class文件中。本地方法由C/C++/汇编语言编写，编译成和处理器相关的机器代码，存储在动态链接库中，格式是各个平台专有。所以本地方法是联系Java程序和底层主机操作系统的连接方式。

　　由于Java虚拟机并不知道某个class文件是如何被创建的，是否被篡改一无所知，所以它实现了一个class文件检测器，确保class文件中定义的类型可以安全地使用。class文件检验器通过四趟独立的扫描来保证程序的健壮性：

* + class文件的结构检查
  + 类型数据的语义检查
  + 字节码验证
  + 符号引用验证

　　Java虚拟机在执行字节码时还进行其它的一些内置的安全机制的操作，他们作为Java编程语言保证Java程序健壮性的特性，同时也是Java虚拟机的特性：

* + 类型安全的引用转换
  + 结构化的内存访问
  + 自动垃圾收集
  + 数组边界检查
  + 空引用检查

## 虚拟机体系

Java虚拟机是一个想象中的机器，在实际的计算机上通过软件模拟来实现。Java虚拟机有自己想象中的硬件，如处理器、堆栈、寄存器等，还具有相应的指令系统。

Java虚拟机规范定义了一个抽象的——而非实际的——机器或处理器。这个规范描述了一个指令集，一组寄存器，一个堆栈，一个“垃圾堆”，和一个方法区。一旦一个Java虚拟机在给定的平台上运行，任何Java程序（编译之后的程序，称作字节码）都能在这个平台上运行。Java虚拟机（JVM）可以以一次一条指令的方式来解释字节码（把它映射到实际的处理器指令），或者字节码也可以由实际处理器中称作just-in-time的编译器进行进一步的编译。Java虚拟机包括一套字节码指令集、一组寄存器、一个栈、一个垃圾回收堆和一个存储方法域。

### 规格描述

JVM定义了控制Java代码解释执行和具体实现的五种规格，它们是：

\*JVM指令系统

\*JVM寄存器

\*JVM栈结构

\*JVM碎片回收堆

\*JVM存储区

1. **JVM指令系统**

JVM指令系统同其他计算机的指令系统极其相似。**Java指令也是由操作码和操作数两部分组成。操作码为8位二进制数，操作数进紧随在操作码的后面，其长度根据需要而不同,当长度大于8位时，操作数被分为两个以上字节存放。**JVM采用了"big endian"的编码方式来处理这种情况，即高位bits存放在低字节中。这同 Motorola及其他的RISC CPU采用的编码方式是一致的，而与Intel采用的"little endian"的编码方式即低位bits存放在低位字节的方法不同。Java指令系统是以Java语言的实现为目的设计的，其中包含了用于调用方法和监视多线程系统的指令。**Java的8位操作码的长度使得JVM最多有256种指令，java1.6及以上版本已使用了160多种操作码**。

1. **JVM寄存器**

所有的CPU均包含用于保存系统状态和处理器所需信息的寄存器组。如果虚拟机定义较多的寄存器，便可以从中得到更多的信息而不必对栈或内存进行访问，这有利于提高运行速度。然而，**如果虚拟机中的寄存器比实际CPU的寄存器多，在实现虚拟机时就会占用处理器大量的时间来用常规存储器模拟寄存器，这反而会降低虚拟机的效率**。针对这种情况，**JVM只设置了4个最为常用的寄存器**。它们是：

pc程序计数器、

optop操作数栈顶指针、

frame当前执行环境指针、

vars指向当前执行环境中第一个局部变量的指针；

注：**所有寄存器均为32位**。pc用于记录程序的执行。optop,frame和vars用于记录指向Java栈区的指针。

1. **JVM栈结构**

作为基于栈结构的计算机，Java栈是JVM存储信息的主要方法。**当JVM得到一个Java字节码应用程序后，便为该代码中每一个类的每一个方法创建一个栈框架，以保存该方法的状态信息**。每个栈框架包括以下三类信息：**局部变量区，运行环境，操作数栈**。

* **局部变量区**：用于存储一个类的方法中所用到的局部变量。vars寄存器指向该变量表中的第一个局部变量。每个Java方法使用一个固定大小的局部变量集。它们按照与vars寄存器的字偏移量来寻址。**局部变量都是32位的。长整数和双精度浮点数占据了两个局部变量的空间，却按照第一个局部变量的索引来寻址。**（例如，一个具有索引n的局部变量，如果是一个双精度浮点数，那么它实际占据了索引n和n+1所代表的存储空间。）**虚拟机规范并不要求在局部变量中的64位的值是64位对齐的。**虚拟机提供了把局部变量中的值装载到操作数栈的指令，也提供了把操作数栈中的值写入局部变量的指令。
* **运行环境**：在运行环境中包含的信息用于**动态链接、正常的方法返回、异常传播**。用于保存解释器对Java字节码进行解释过程中所需的信息。它们是：上次调用的方法、局部变量指针和操作数栈的栈顶和栈底指针。执行环境是一个执行一个方法的控制中心。例如：如果解释器要执行iadd（整数加法），首先要从frame寄存器中找到当前执行环境，而后便从执行环境中找到操作数栈，从栈顶弹出两个整数进行加法运算，最后将结果压入栈顶。

1. **动态链接**

运行环境包括对指向当前类和当前方法的解释器符号表的指针，用于支持方法代码的动态链接。方法的class文件代码在引用要调用的方法和要访问的变量时使用符号。动态链接把符号形式的方法调用翻译成实际方法调用，装载必要的类以解释还没有定义的符号，并把变量访问翻译成与这些变量运行时的存储结构相应的偏移地址。动态链接方法和变量使得方法中使用的其它类的变化不会影响到本程序的代码。

1. **正常的方法返回**

如果当前方法正常地结束了，在执行了一条具有正确类型的返回指令时，调用的方法会得到一个返回值。执行环境在正常返回的情况下用于恢复调用者的寄存器，并把调用者的程序计数器增加一个恰当的数值，以跳过已执行过的方法调用指令，然后在调用者的执行环境中继续执行下去。

1. **异常传播**

异常情况在Java中被称作Error（错误）或Exception（异常）,是Throwable类的子类，在程序中的原因是：

①**动态链接错**，如无法找到所需的class文件。

②**运行时错**，如对一个空指针的引用。

③**程序使用了throw语句**。

注：当异常发生时，Java虚拟机采取如下措施：

1. **定位catch（在方法内部）**：检查与当前方法相联系的catch子句表。每个catch子句包含其有效指令范围，能够处理的异常类型，以及处理异常的代码块地址。
2. **异常类型匹配（在方法内部）**：与异常相匹配的catch子句应该符合下面的条件：造成异常的指令在其指令范围之内，发生的异常类型是其能处理的异常类型的子类型。如果找到了匹配的catch子句，那么系统转移到指定的异常处理块处执行；如果没有找到异常处理块，重复寻找匹配的catch子句的过程，直到当前方法的所有嵌套的catch子句都被检查过。
3. **处理异常（在方法内部, 前两步成功）：**由于虚拟机从第一个匹配的catch子句处继续执行，所以catch子句表中的顺序是很重要的。因为Java代码是结构化的，因此总可以把某个方法的所有的异常处理器都按序排列到一个表中，对任意可能的程序计数器的值，都可以用线性的顺序找到合适的异常处理块，以处理在该程序计数器值下发生的异常情况。
4. **异常传播：**如果找不到匹配的catch子句，那么当前方法得到一个"未截获异常"的结果并返回到当前方法的调用者，好像异常刚刚在其调用者中发生一样。如果在调用者中仍然没有找到相应的异常处理块，那么这种错误传播将被继续下去。**如果错误被传播到最顶层，那么系统将调用一个缺省的异常处理块。**

* **操作数栈**：用于存储运算所需操作数及运算的结果。

操作数栈区机器指令只从操作数栈中取操作数，对它们进行操作，并把结果返回到栈中。选择栈结构的原因是：在只有少量寄存器或非通用寄存器的机器（如Intel486）上，也能够高效地模拟虚拟机的行为。**操作数栈是32位的**。它用于给方法传递参数，并从方法接收结果，也用于支持操作的参数，并保存操作的结果。例如，iadd指令将两个整数相加。相加的两个整数应该是操作数栈顶的两个字。这两个字是由先前的指令压进堆栈的。这两个整数将从堆栈弹出、相加，并把结果压回到操作数栈中。

每个原始数据类型都有专门的指令对它们进行必须的操作。**每个操作数在栈中需要一个存储位置，除了long和double型，它们需要两个位置**。操作数只能被适用于其类型的操作符所操作。例如，压入两个int类型的数，如果把它们当作是一个long类型的数则是非法的。在Sun的虚拟机实现中,这个限制由字节码验证器强制实行。但是，有少数操作（操作符dupe和swap），用于对运行时数据区进行操作时是不考虑类型的。

1. **JVM碎片回收堆（无用单元收集堆）**

Java类的实例所需的存储空间是在堆上分配的。解释器具体承担为类实例分配空间的工作。解释器在为一个实例分配完存储空间后，便开始记录对该实例所占用的内存区域的使用。一旦对象使用完毕，便将其回收到堆中。在Java语言中，除了new语句外没有其他方法为一对象申请和释放内存。对内存进行释放和回收的工作是由Java运行系统承担的。这允许Java运行系统的设计者自己决定碎片回收的方法。在SUN公司开发的Java解释器和Hot Java环境中，碎片回收用后台线程的方式来执行。这不但为运行系统提供了良好的性能，而且使程序设计人员摆脱了自己控制内存使用的风险。

1. **JVM存储区**

**JVM有两类存储区：常量缓冲池和方法区**。**常量缓冲池用于存储类名称、方法和字段名称以及串常量**。**方法区则用于存储Java方法的字节码**。方法区与传统语言中的编译后代码或是Unix进程中的正文段类似。它保存方法代码（编译后的java代码）和符号表。在当前的Java实现中，方法代码不包括在无用单元收集堆中，但计划在将来的版本中实现。每个类文件包含了一个Java类或一个Java界面的编译后的代码。可以说类文件是Java语言的执行代码文件。为了保证类文件的平台无关性，Java虚拟机规范中对类文件的格式也作了详细的说明。其具体细节请参考Sun公司的Java虚拟机规范。

对于这两种存储区域具体实现方式在JVM规格中没有明确规定。这使得Java应用程序的存储布局必须在运行过程中确定，依赖于具体平台的实现方式。

*JVM是为Java字节码定义的一种独立于具体平台的规格描述，是Java平台独立性的基础。尽管JVM还存在一些限制和不足，有待于进一步的完善，但无论如何，JVM的思想是成功的。对比分析：如果把Java原程序想象成我们的C++原程序，Java原程序编译后生成的字节码就相当于C++原程序编译后的80x86的机器码（二进制程序文件），JVM虚拟机相当于80x86计算机系统,Java解释器相当于80x86CPU。在80x86CPU上运行的是机器码，在Java解释器上运行的是Java字节码。*

*Java解释器相当于运行Java字节码的“CPU”，但该“CPU”不是通过硬件实现的，而是用软件实现的。Java解释器实际上就是特定的平台下的一个应用程序。只要实现了特定平台下的解释器程序，Java字节码就能通过解释器程序在该平台下运行，这是Java跨平台的根本。当前，并不是在所有的平台下都有相应Java解释器程序，这也是Java并不能在所有的平台下都能运行的原因，它只能在已实现了Java解释器程序的平台下运行。*

## 垃圾回收

# java初始化顺序

java类初始化顺序：

父类--静态变量、静态初始化块：按照定义的顺序初始化

子类--静态变量、静态初始化块：按照定义的顺序初始化

子类main方法

父类--变量、初始化块：按照定义的顺序初始化

父类--构造器

子类--变量、初始化块：按照定义的顺序初始化

子类--构造器

# java序列化与反序列化

1. serialVersionUID 生成规则：只有静态属性以及构造方法变化时不改变其值，其余皆改变
2. 当使用writeObject 与 readObject时：请注意写入属性的顺序和读取属性的顺序应该一致
3. 反序列化时：通过二进制流直接生成对象，并不调用构造方法；

生成类的方式：new、Class.forName("xxx")、反射、反序列化；

其中new、Class.forName("xxx")、反射都是调用构造器，而反序列化不是；

反序列化：不调用构造器，自身相当于构造器的另一种写法；

1. 如果继承自Serialiable则：

序列化函数：

private Object writeReplace()throws ObjectStreamException

private void writeObject(ObjectOutputStream out)throws IOException

注：系统总是先调用writeReplace()方法，然后调用writeObject方法；

反序列化函数：

private void readObject(ObjectInputStream in)throws IOException,ClassNotFoundException

private Object readResolve() throws ObjectStreamException

注：系统总是先调用readObject ()方法，然后调用readResolve方法；

注：如果writeReplace()方法改变了对象类型，那么将会调用真实类型的writeObject ，反序列化时调用真实类型的readObject 以及真实类型的readResolve ;

1. 如果继承自Externalizable则：

不像Serializable接口只是一个标记接口，里面的接口方法都是可选的（可实现可不实现，如果不实现则启用其自动序列化功能），而Externalizable接口不是一个标记接口，它强制你自己动手实现串行化和反串行化算法：

public interface Externalizable extends java.io.Serializable {

// 序列化

void writeExternal(ObjectOutput out) throws IOException;

/ / 反序列化

void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;

}

该接口直接继承了Serializable接口，其两个方法其实就对应了Serializable的writeObject和readObject方法，实现方式也是一模一样；

ObjectOutput和ObjectInput的使用方法和ObjectOutputStream和ObjectInputStream一模一样（readObject、readInt之类的，完全一模一样）

因此Externalizable就是强制实现版的Serializable罢了；

# java遍历

## java遍历：for、do while、while、foreach、iterator的用法和区别

相同点： 五个都可以用来遍历数组和集合

不同点：

1. 形式差别

do while的形式是

do{ xxxx }while(yyyy);

while的形式是

while(yyyy){ xxxxxx }

for的形式是

for（int i=0;i<arr.size();i++）{...}

foreach的形式是

for（int　i：arr）{...}

iterator的形式是

Iterator it = arr.iterator();

while(it.hasNext()){ object o =it.next(); ...}

1. 条件差别

for需要知道集合或数组的大小，而且需要是有序的，不然无法遍历；

foreach和iterator都不需要知道集合或数组的大小，他们都是得到集合内的每个元素然后进行处理；

1. 多态差别

for和foreach都需要先知道集合的类型，甚至是集合内元素的类型，即需要访问内部的成员，不能实现态；

iterator是一个接口类型，他不关心集合或者数组的类型，而且他还能随时修改和删除集合的元素，举个例子：

public void display（Iterator<object> it）{

while(it.hasNext()){

system.out.print(it.next()+"");

}

}

当我们需要遍历不同的集合时，我们只需要传递集合的iterator（如arr.iterator()）看懂了吧，这就 是iterator的好处，他不包含任何有关他所遍历的序列的类型信息，能够将遍历序列的操作与序列底层的结构分离。迭代器统一了对容器的访问方式。这也是接口的解耦的最好体现。

1. 用法差别

for循环一般用来处理比较简单的有序的，可预知大小的集合或数组

foreach可用于遍历任何集合或数组，而且操作简单易懂，他唯一的不好就是需要了解集合内部类型

iterator是最强大的，他可以随时修改或者删除集合内部的元素，并且是在不需要知道元素和集合的类型的情况下进行的（原因可参考第三点：多态差别），当你需要对不同的容器实现同样的遍历方式时，迭代器是最好的选择！

注：

for，while是一样的，形式上的不同

foreach，Iterator也是一样的，形式上的不同

性能肯定是用迭代器的好一些。迭代器能直接访问集合内部元素。而且有的集合没有索引，没法用for来遍历，只有用迭代器。

## java遍历集合

1. 对set的遍历

迭代遍历

Set<String> set = new HashSet<String>();

Iterator<String> it = set.iterator();

while (it.hasNext()) {

String str = it.next();

System.out.println(str);

}

for循环遍历：

for (String str : set) {

System.out.println(str);

}

优点还体现在泛型 假如 set中存放的是Object

for循环遍历：

Set<Object> set = new HashSet<Object>();

for (Object obj: set) {

if(obj instanceof Integer){

int aa= (Integer)obj;

}else if(obj instanceof String){

String aa = (String)obj

}

........

}

1. 对map的遍历

迭代遍历

Set set = map.keySet();

Iterator it = set.iterator();

while(it.hasNext()){

String str = it.next().toString();

}

Iterator it = paraMap.entrySet().iterator();

while (it.hasNext())

{

Map.Entry pairs = (Map.Entry)it.next();

System.out.println(pairs.getKey() + " = " + pairs.getValue());

}

for循环遍历：

Map<String,String> paraMap=new HashMap<>();

for(String dataKey : paraMap.keySet())

{

System.out.println(dataKey );

}

这里要注意的是,paraMap是怎么样定义的,如果是简单的Map paraMap = new HashMap();那前面的String就只能换成Object了.

对整Map的key和value都进行循环,如下:

Java代码

复制代码 代码如下:

for(Map.Entry<String, Object> entry : paraMap.entrySet())

{

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

}

例子：

Map<String,String> map = new HashMap<String,String>(){};

map.put("a", "A");

map.put("b", "B");

map.put("c", "C");

//第一种

Set set = map.keySet();

Iterator it = set.iterator();

while(it.hasNext()){

    String str = it.next().toString();

}

//第二种

Set<Entry<String,String>> entryset = map.entrySet();

Iterator iter = entryset.iterator();

while(iter.hasNext()){

    Entry<String,String> entry = (Entry<String,String>)iter.next();

}

//第三种

for(Entry<String,String> entry : map.entrySet()){

    String strkey = entry.getKey();

    String strval = entry.getValue();

}

首先，使用entryset比使用keyset的效率要高。所以建议用后2种方法。

   第二和第三种的区别其实就是使用迭代器iteratior和使用一般的for语句的区别，和map本身倒没什么关系。最简单的判断方法，就是对要迭代的内容（本例是个map,如果是一个list同理）,在循环过程中，你需要进行add或者remove操作不。for语句的循环是不能做这样的操作的，只能使用迭代器。

   不过，个人认为，for语句虽然看上去代码少一点，但敲起来没iterator有感觉

# java泛型

## 泛型类：

**class** classType<T **extends** Number,V **extends** Object>

{

T t;

V v;

}

**class** classType2<T ,V **extends** Object>

{

T t;

V v;

}

**class** classType3<T **extends** Number,V >

{

T t;

V v;

}

## 泛型方法：

❶：

**public**<T> T getObject(Class<T>tc) **throws** InstantiationException,IllegalAccessException

{

T t= tc.newInstance();

// try {

// T t2= tc.newInstance();

// } catch (InstantiationException | IllegalAccessException e) {

// // **TODO** Auto-generated catch block

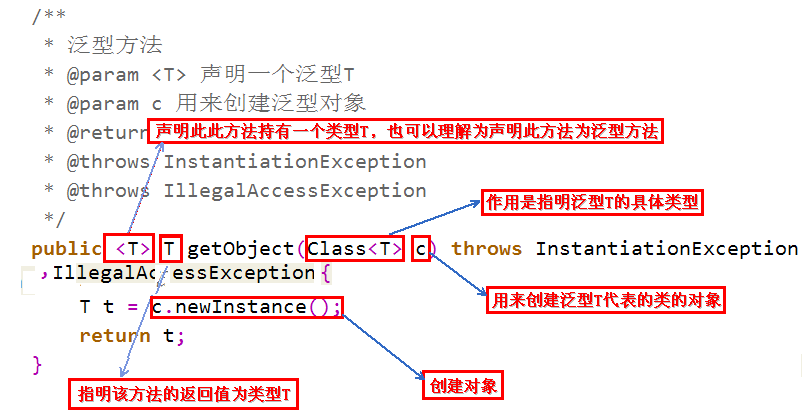
// e.printStackTrace();

// }

**return**t;

}

* 1. <T>：声明此方法有一个类型T，也可以理解为声明此方法为泛型方法
  2. Class<T>：作用是指明泛型T的具体类型，传值方法：getObject(xxxx.class);
  3. tc：用来创建泛型T代表的类的对象
  4. tc.newInstance()：创建泛型对象



❷：

**public**<T> Set<?> getT(Tt)

{

**returnnull**;

}

有通配符修饰的变量，可以遍历，不能修改，比如List<?> list,该list变量就只能够用于遍历；

# java类型之间的关系判断（instanceof、isAssignableFrom）

### instanceof

用法：

实例 instanceof 类名/接口名

作用：

判断该实例是否是（类名/接口名）或则其子类的实例；

另一个写法：

类名/接口名.class.isInstance(实例)

例：

List<String> list=new ArrayList<String>();

System.out.println(list.getClass().isInstance(new ArrayList<String>()));

System.out.println(list instanceof ArrayList);

System.out.println(ArrayList.class.isInstance(list));

System.out.println(list.getClass().isAssignableFrom(ArrayList.class));

System.out.println(ArrayList.class.isAssignableFrom(list.getClass()));

### isAssignableFrom

用法：

(类名1/接口名1).class.isAssignableFrom(类名2/接口名2)

作用：

判断（类名1/接口名1）是否是（类名2/接口名2）的父类或其自身

注：

instanceof：子-->父

isAssignableFrom：父-->子

# Java知识外延

## java.lang.Class类

### getName(),getSimpleName(),getCanonicalName ()三者之间的区别

|  |  |  |
| --- | --- | --- |
| 函数 | 非数组 | 数组 |
| getName() | 包.类名称 | [L包.类名称 |
| getSimpleName() | 类名称 | 类名称[] |
| getCanonicalName() | 包.类名称 | 包.类名称[] |

## 数据库访问部分

### javax.persistence.EntityManager;

String sql="select \* from picc\_gift\_info r where r.registerId=?registerId ";

javax.persistence.Query query = new javax.persistence.EntityManager().createNativeQuery(sql, GiftReceivedVo.class);

query.setParameter("registerId", registerId);

query.setFirstResult(start);

query.setMaxResults(limit);

List<GiftReceivedVo> list = (List<GiftReceivedVo>) query.getResultList();

1. 持久化实体persist()：

new EntityManager().persist(new GiftReceivedVo().setId(“100”))

1. 按主键查询find()：

GiftReceivedVo temp=new EntityManager().find(GiftReceivedVo.class,”100”);

1. 删除实体remove()：

new EntityManager().remove(new EntityManager().find(GiftReceivedVo.class,”100”));

1. 更新实体merge()：更新至数据库(处不处于管理模式都会更新！)

GiftReceivedVo temp=new EntityManager().find(GiftReceivedVo.class,”100”);

temp.setName(“大王哟”);

new EntityManager().merge(temp);

1. 更新实体flush()：更新至数据库（只更新处于管理模式的bean）

GiftReceivedVo temp=new EntityManager().find(GiftReceivedVo.class,”100”);

temp.setName(“大王哟”);

new EntityManager().flush (temp);

1. 从数据库中更新refresh()：与数据库同步

GiftReceivedVo temp=new EntityManager().find(GiftReceivedVo.class,”100”);

new EntityManager().refresh (temp);

1. 检测实体是否被管理contains()：好比flush()与merge()

GiftReceivedVo temp=new EntityManager().find(GiftReceivedVo.class,”100”);

Boolean b=new EntityManager().contains (temp);

b==true:处于管理模式；

b==false:未处于管理模式；

1. 分离管理的实体clear():

当处理了大量的实体后，这些实体都会处于实体管理器中，将会消耗大量的内存，使得程序运行变慢；如果要减少消耗，则可以使用clear()方法，将正在被管理的的实体从持久化内容中分离出来

new EntityManager().clear();

如果调用clear()方法，则之前所有对实体所做的任何改变都会被丢失，所以在调用clear()方法之前先调用flush()方法保存更改！

### javax.persistence.Query

关于占位符的问题：

* 如果是Query qry = new EntityManager().createNativeQuery(sql,XXXXX.class)则占位符一律是”?”;
* 如果是Query qry = new EntityManager().createQuery(sql,XXXXX.class)则占位符如果是数字，则为 “?1”；如果是字符串，则为”:XXXXXX”;

关于更新的问题：

* 如果是insert into 或者 update 则为：Query q = this.em.createNativeQuery(sql )；q.executeUpdate();

### 关于sql语句的说明

如果使用原生sql，则sql拼接语句中尽量不要出现\n等java类的转义字符，可能会出错！！！

### 注意事项

关于模糊查询：

Sql:like ?gift\_Name

query.setParameter("gift\_Name", "%"+map\_action.get("giftNameSearch").toString().trim()+"%");

关于in查询：

Sql:in (t1,t2……)

Java:使用循环赋值;