# 一．Java内存区域

**1. 程序计数器**

当前字节码的行号指示器，字节码解释器，通过改变它，来选取下一个字节码。

**2.虚拟机栈**

我们平时经常会说heap和stack就是我们说的stack栈，存放局部变量，线程私有。

**3.本地方法栈**

Native方法栈

**4.堆**

唯一目的，存放对象实例，线程共有。

**5.方法区**

被虚拟机加载的类，常量，静态变量，即使编译器编译后的代码数据。

**6.常量池**

编译期生成的各种字面量和符号应用，string就是放在这。

**7.直接内存**

Native库，申请的堆外的内存。

# 二.对象的创建

## 2.1创建的步骤

对象的创建,分成两个部分,虚拟机控制一部分,构造函数控制一部分,可以分成6步

1. 检测常量池,如果常量池有这个对象,那么取这个对象end
2. 检测这个类,是否有被加载,解析和初始化,如果没有,先解析这个类
3. 分配内存
4. 初始化所有字段,为默认值,0,false,null
5. 执行类的静态块,赋值静态变量
6. 构造函数

以上,完成一个对象的构造.其中,1~4是虚拟机完成的,56是代码可控的.

## 2.2分配内存相关

### 2.2.1方法

**1.指针碰撞**

假设内存是规整的,空闲的全放一边,非空闲的全放一边,那么只要简单的移动指针位置就可以了,这个叫指针碰撞.

**2.空闲列表**

内存并不规整,那就要维护一个列表,告知哪些内存可用,哪些内存不可用.分配时候,找到一个足够大的,分配内存,并且更新列表.

内存是否规整,跟gc相关.

### 2.2.2同步

在多个线程中,即使只是改动指针位置,也有可能产生,A,B线程,同时申请了一块内存,这就涉及到同步.有两个方案:

1. 每次new,都要同步.
2. 给线程分配本地缓冲TLAB(thread local allocation buffer),new在本地缓冲上,更新到主存上时候,再同步.

## 2.3内存对象的布局

**1.对象头**

对象头包含2个部分:

运行时数据：hashcode,,lock标志,gc年代......

类型指针：它是哪个类的实例。（有的虚拟机，没有这个位）

**2.实例数据**

就是各个字段的数据。

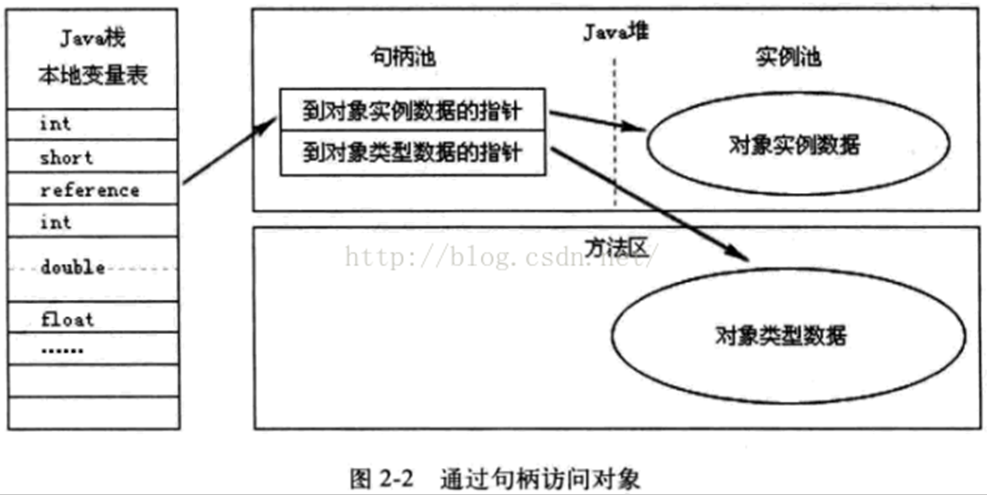
**3.填充位**

不是必然存在的。

## 2.4访问定位

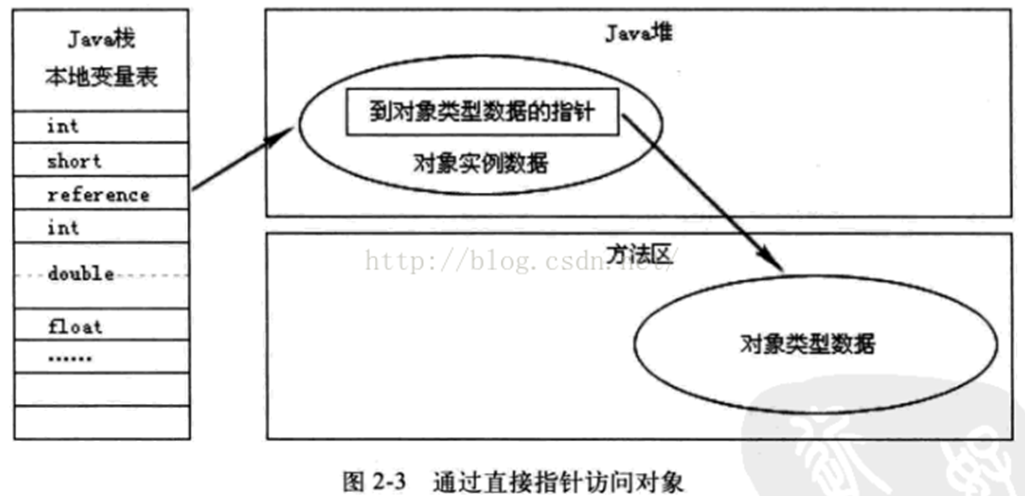
**1.句柄式**

优势在于，gc的时候，可能伴随着大量的对象移动，这个时候只需要改动句柄。



**2.直接式**

优势在于：快，节约了一次寻址的时间



# 三．垃圾收集

## 三个问题：

1.哪些需要回收

2.什么时候回收

3.怎么回收

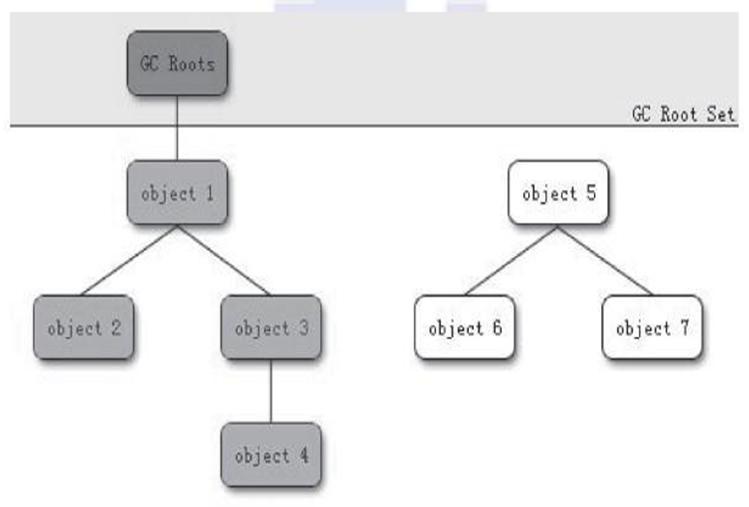
## 3.1对象死了没？

### 3.1.1引用计数

引用计数是错误的，不能解决循环引用的问题。

### 3.2.2可达性分析

从GC roots节点是否可以达到这个对象。



在Java语言中，可作为GC Roots的对象包括下面几种：

1.虚拟机栈（栈帧中的本地变量表）中引用的对象。

2.方法区中类静态属性引用的对象。

3.方法区中常量引用的对象。

4.本地方法栈中JNI（即一般说的Native方法）引用的对象

## 3.2 Finalize起死回生

回收一个对象的过程：

1. 可达性算法中，不可达的对象，判缓刑。
2. 缓刑的对象，如果没有复写finalize方法，判死刑，可以gc掉了
3. 缓刑的对象，如果有复写finalize方法，缓刑的对象执行finalize，如果在finalize中，又被人引用了，可以拯救自己
4. 在下一次可达性判断中，如果做过finalize的对象，又不可达了，就直接可以gc掉。

Finalize，不一定会执行完，因为如果finalzie太慢或者死循环的话，会造成F-queue对列永久等待，其他的finalize方法就没有机会执行。所以在这里，释放外部资源，也是不靠铺的。

### 3.3方法区，也是可以回收的

方法区，也是可以回收的。

## 3.3垃圾回收算法

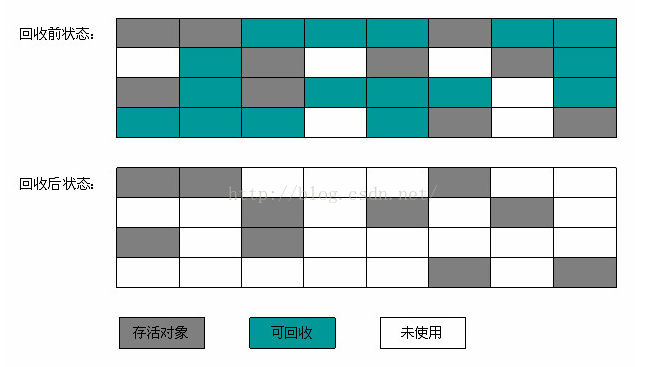
标记 - 清除算法  标记清除算法是最基础的收集算法，其他收集算法都是基于这种思想。标记清除算法分为“标记”和“清除”两个阶段：首先标记出需要回收的对象，标记完成之后统一清除对象

### 3.3.1标记-清除算法：

这是最基础的算法，其他的算法，都是按照这个算法改动而来的。

先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

缺点：回收了被标记的对象后，由于未经过整理，所以导致很多内存碎片



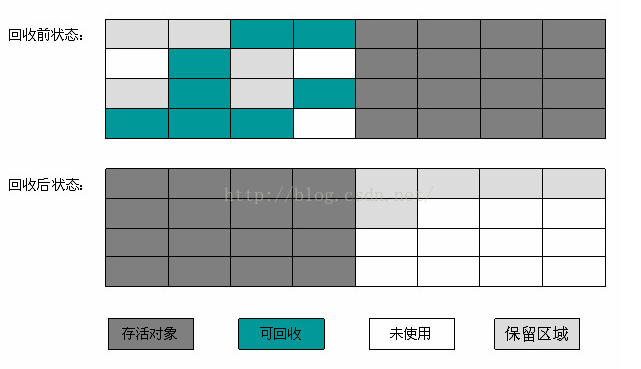
### 3.3.2停止-复制算法：

将可用的内存按容量划分为大小相等的两块（from，to），每次只是用其中一块（总有一块是空的【to区域】）。当这一苦熬的内存用完了，就将还存活着的对象复制到另外一块上面，然后把已使用过的内存空间一次清理完。

HotSpot虚拟机默认Eden和Survivor大小的比例是8:1，也就是每次新生代中可用的内存空间为整个新生代容量的90%，只有10%的内存时被浪费的。

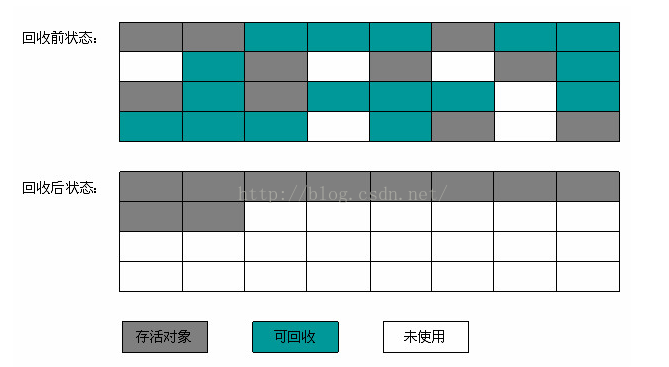
缺点：浪费内存空间，如果对象存活率较高时要执行较多的复制操作，效率降低。

图解：有一块内存区域是空的，一般是to区域。保留区域每次回收后都因为复制的时候让他们变为连续的地址空间，所有不产生内存碎片。



### 3.3 .3标记-整理算法

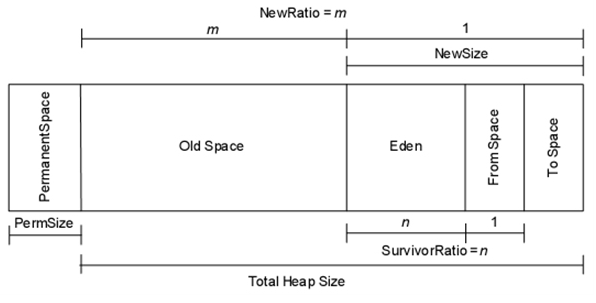
标记整理算法的“标记”过程和标记-清除算法一致，只是后面并不是直接对可回收对象进行整理，而是让所有存活的对象都向一段移动，然后直接清理掉端边界意外的内存。



## 3.4Heap内存模型

Heap的内存模型大致如下图，一块是 NEW Generation, 另一块是Old Generation. 在New Generation中，有一个叫Eden的空间，主要是用来存放新生的对象，还有两个Survivor Spaces（from,to）, 它们用来存放每次垃圾回收后存活下来的对象。在Old Generation中，主要存放应用程序中生命周期长的内存对象，还有个Permanent Generation，主要用来放JVM自己的反射对象，比如类对象和方法对象等。

默认的情况下，m=2，n=8。

3.5 回收逻辑

### 3.5.1回收步骤

1、绝大多数刚刚被创建的对象会存放在伊甸园空间（Eden）。

2、在伊甸园空间执行第一次GC（Minor GC）之后，存活的对象被移动到其中一个幸存者空间（Survivor）。

3、此后，每次伊甸园空间执行GC后，存活的对象会被堆积在同一个幸存者空间。

4、当一个幸存者空间饱和，还在存活的对象会被移动到另一个幸存者空间。然后会清空已经饱和的哪个幸存者空间。

5、在以上步骤中重复N次（N = MaxTenuringThreshold（年龄阀值设定，默认15））依然存活的对象，就会被移动到老年代。

我们需要重点记住的是，对象在刚刚被创建之后，是保存在伊甸园空间的（Eden）。那些长期存活的对象会经由幸存者空间（Survivor）转存到老年代空间（Old generation）。

也有例外出现，对于一些比较大的对象（需要分配一块比较大的连续内存空间）则直接进入到老年代。一般在Survivor 空间不足的情况下发生。

### 3.5.2老年代空间的构成与逻辑

老年代空间的构成其实很简单，它不像新生代空间那样划分为几个区域，它只有一个区域，里面存储的对象并不像新生代空间绝大部分都是朝闻道，夕死矣。这里的对象几乎都是从Survivor 空间中熬过来的，它们绝不会轻易的狗带。因此，Full GC（Major GC）发生的次数不会有Minor GC 那么频繁，并且做一次Major GC 的时间比Minor GC 要更长（约10倍）。

减轻survival区的压力，减少minorGc的频率

1. 大对象直接进入老年代，因为你的一个survival可能放不下，或者大对象直接占用了大量的survival，这样会导致minorGc过于频繁
2. 长期存活的对象进入老年态，默认第16次
3. 动态对象年龄判定，当某一个年龄的对象们，他们的总内存之和，大于survival区的一半，那么这个年龄，已经以上的年龄的对象，全都去老年态。

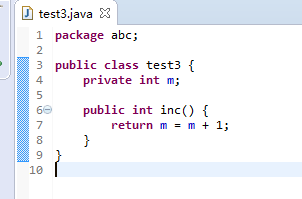
### 3.5.3空间分配担保

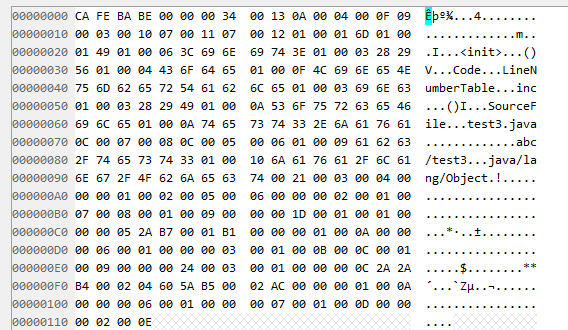
在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。

如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者HandlePromotionFailure设置不允许冒险，那这时也要改为进行一次Full GC。

# 类文件结构

## Java文件





字节码

CA FE BA BE 00 00 00 34 00 13 0A 00 04 00 0F 09

00 03 00 10 07 00 11 07 00 12 01 00 01 6D 01 00

01 49 01 00 06 3C 69 6E 69 74 3E 01 00 03 28 29

56 01 00 04 43 6F 64 65 01 00 0F 4C 69 6E 65 4E

75 6D 62 65 72 54 61 62 6C 65 01 00 03 69 6E 63

01 00 03 28 29 49 01 00 0A 53 6F 75 72 63 65 46

69 6C 65 01 00 0A 74 65 73 74 33 2E 6A 61 76 61

0C 00 07 00 08 0C 00 05 00 06 01 00 09 61 62 63

2F 74 65 73 74 33 01 00 10 6A 61 76 61 2F 6C 61

6E 67 2F 4F 62 6A 65 63 74 00 21 00 03 00 04 00

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

07 00 08 00 01 00 09 00 00 00 1D 00 01 00 01 00

00 00 05 2A B7 00 01 B1 00 00 00 01 00 0A 00 00

00 06 00 01 00 00 00 03 00 01 00 0B 00 0C 00 01

00 09 00 00 00 24 00 03 00 01 00 00 00 0C 2A 2A

B4 00 02 04 60 5A B5 00 02 AC 00 00 00 01 00 0A

00 00 00 06 00 01 00 00 00 07 00 01 00 0D 00 00

00 02 00 0E

基础数据类型u1，u2，u4，u8，分别表示1,2,4,8个字节的无符号数。

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| u4 | magic | 1 |
| u2 | minor\_version | 1 |
| u2 | major\_version | 1 |
| u2 | constant\_pool\_count | 1 |
| cp\_info | constant\_pool | constant\_pool\_count-1 |
| u2 | access\_flags | 1 |
| u2 | this\_class | 1 |
| u2 | super\_class | 1 |
| u2 | interfaces\_count | 1 |
| u2 | interfaces | interfaces\_count |
| u2 | fields\_count | 1 |
| field\_info | fields | fields\_count |
| u2 | methods\_count | 1 |
| method\_info | methods | methods\_count |
| u2 | attributes\_count | 1 |
| attribute\_info | attributes | attributes\_count |

1魔数与版本

魔术就是cafebaby，

CA FE BA BE 00 00 00 34 00 13 0A 00 04 00 0F 09

2版本号就是jdk版本号，1.8.0

CA FE BA BE 00 00 00 34 00 13 0A 00 04 00 0F 09

## 3常量池

常量池主要是三个东西：

1．类和接口的全限定名

2．字段的名称和描述符

3．方法的名称和描述符

下面分析常量池：

#### 1.限定长度

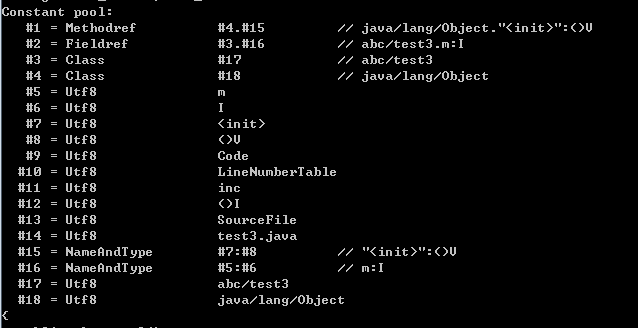
CA FE BA BE 00 00 00 34 00 13 0A 00 04 00 0F 09

咱们有0x0013个常量，也就是16个常量。注意常量池的长度2字节，上限是ffff，也就是说，常量的上限是65355，class类里面，但凡是有个一个标志位，来确定后续长度的，一般都是2个字节，比如方法的数量，上限也是65355.所以我们常说的，方法上限的限制，也就是来源于，java的class文件的结构问题。

#### 2.项目类型

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型 | 简介 | 项目 | 类型 | 描述 |
| CONSTANT\_Utf8\_info | utf-8缩略编码字符串 | tag | u1 | 值为1 |
| length | u2 | utf-8缩略编码字符串占用字节数 |
| bytes | u1 | 长度为length的utf-8缩略编码字符串 |
| CONSTANT\_Integer\_info | 整形字面量 | tag | u1 | 值为3 |
| bytes | u4 | 按照高位在前储存的int值 |
| CONSTANT\_Float\_info | 浮点型字面量 | tag | u1 | 值为4 |
| bytes | u4 | 按照高位在前储存的float值 |
| CONSTANT\_Long\_info | 长整型字面量 | tag | u1 | 值为5 |
| bytes | u8 | 按照高位在前储存的long值 |
| CONSTANT\_Double\_info | 双精度浮点型字面量 | tag | u1 | 值为6 |
| bytes | u8 | 按照高位在前储存的double值 |
| CONSTANT\_Class\_info | 类或接口的符号引用 | tag | u1 | 值为7 |
|  | index | u2 | 指向全限定名（参见备注一）常量项的索引 |
| CONSTANT\_String\_info | 字符串类型字面量 | tag | u1 | 值为8 |
| index | u2 | 指向字符串字面量的索引 |
| CONSTANT\_Fieldref\_info | 字段的符号引用 | tag | u1 | 值为9 |
| index | u2 | 指向声明字段的类或接口描述符CONSTANT\_Class\_info的索引项 |
| index | u2 | 指向字段描述符CONSTANT\_NameAndType\_info的索引项 |
| CONSTANT\_Methodref\_info | 类中方法的符号引用 | tag | u1 | 值为10 |
| index | u2 | 指向声明方法的类描述符CONSTANT\_Class\_info的索引项 |
| index | u2 | 指向名称及类型描述符CONSTANT\_NameAndType\_info的索引项 |
| CONSTANT\_InterfaceMethodref\_info | 接口中方法的符号引用 | tag | u1 | 值为11 |
| index | u2 | 指向声明方法的接口描述符CONSTANT\_Class\_info的索引项 |
| index | u2 | 指向名称及类型描述符CONSTANT\_NameAndType\_info的索引项 |
| CONSTANT\_NameAndType\_info | 字段或方法的部分符号引用 | tag | u1 | 值为12 |
| index | u2 | 指向该字段或方法名称常量项的索引 |
| index | u2 | 指向该字段或方法描述符常量项的索引 |

我们看一下javap的结果，然后继续分析



##### 第一个常量 #1

他的tag是0A表示这是一个CONSTANT\_Methodref\_info，方法信息。

CA FE BA BE 00 00 00 34 00 13 0A 00 04 00 0F 09

它的结构如下u2的类名索引，u2的方法名索引。

那么表示第一个常量就是

CA FE BA BE 00 00 00 34 00 13 0A 00 04 00 0F 09

它的类名是0004，表示指向第4个常量，

CA FE BA BE 00 00 00 34 00 13 0A 00 04 00 0F 09

它的方法名是000F，表示指向第15个常量

CA FE BA BE 00 00 00 34 00 13 0A 00 04 00 0F 09

第一个常量 #1结束

##### 第二个常量 #2

他的tag是0A表示这是一个09，表示这是一个CONSTANT\_Fieldref\_info，字段。

CA FE BA BE 00 00 00 34 00 13 0A 00 04 00 0F 09

00 03 00 10 07 00 11 07 00 12 01 00 01 6D 01 00

它的结构如下u2的类型索引（包括基本类型和对象），u2的描述索引。

那么整个第二个常量就是

CA FE BA BE 00 00 00 34 00 13 0A 00 04 00 0F 09

00 03 00 10 07 00 11 07 00 12 01 00 01 6D 01 00

它的类型索引00 03指向第三个常量

00 03 00 10 07 00 11 07 00 12 01 00 01 6D 01 00

它的描述索引指向 00 10 第十个常量

00 03 00 10 07 00 11 07 00 12 01 00 01 6D 01 00

第一个常量 #2结束

##### 第三个常量

他的tag是0A表示这是一个07，表示这是一个CONSTANT\_Class\_info字段。

00 03 00 10 07 00 11 07 00 12 01 00 01 6D 01 00

它的结构包含一个u2指向类的全限定名的索引.

那么整个第三个常量就是

00 03 00 10 07 00 11 07 00 12 01 00 01 6D 01 00

类的全限定名索引是00 11 指向第17个常量

00 03 00 10 07 00 11 07 00 12 01 00 01 6D 01 00

第三个常量#3结束

常量池先分析到这，可以看上面javap出来的表。

### 访问标志

这个类的访问标志，是不是共有的，是不是接口之类的东西，具体看下表。

|  |  |  |
| --- | --- | --- |
| 标志名称 | 标志值 | 含义 |
| ACC\_PUBLIC | 0x0001 | 是否为public类型 |
| ACC\_FINAL | 0x0010 | 是否被声明为final，只有类可设置 |
| ACC\_SUPER | 0x0020 | 是否允许使用invokespecial字节码指令，JDK1.2以后编译出来的类这个标志为真 |
| ACC\_INTERFACE | 0x0200 | 标识这是一个接口 |
| ACC\_ABSTRACT | 0x0400 | 是否为abstract类型，对于接口和抽象类，此标志为真，其它类为假 |
| ACC\_SYNTHETIC | 0x1000 | 标识别这个类并非由用户代码产生 |
| ACC\_ANNOTATION | 0x2000 | 标识这是一个注解 |
| ACC\_ENUM | 0x4000 | 标识这是一个枚举 |

根据上面的表格，测试类的访问标志为

ACC\_PUBLIC | ACC\_SUPER =

0x0001 | 0x0020 = 0x0021

字段位置如下：

CA FE BA BE 00 00 00 34 00 13 0A 00 04 00 0F 09

00 03 00 10 07 00 11 07 00 12 01 00 01 6D 01 00

01 49 01 00 06 3C 69 6E 69 74 3E 01 00 03 28 29

56 01 00 04 43 6F 64 65 01 00 0F 4C 69 6E 65 4E

75 6D 62 65 72 54 61 62 6C 65 01 00 03 69 6E 63

01 00 03 28 29 49 01 00 0A 53 6F 75 72 63 65 46

69 6C 65 01 00 0A 74 65 73 74 33 2E 6A 61 76 61

0C 00 07 00 08 0C 00 05 00 06 01 00 09 61 62 63

2F 74 65 73 74 33 01 00 10 6A 61 76 61 2F 6C 61

6E 67 2F 4F 62 6A 65 63 74 00 21 00 03 00 04 00

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

07 00 08 00 01 00 09 00 00 00 1D 00 01 00 01 00

00 00 05 2A B7 00 01 B1 00 00 00 01 00 0A 00 00

00 06 00 01 00 00 00 03 00 01 00 0B 00 0C 00 01

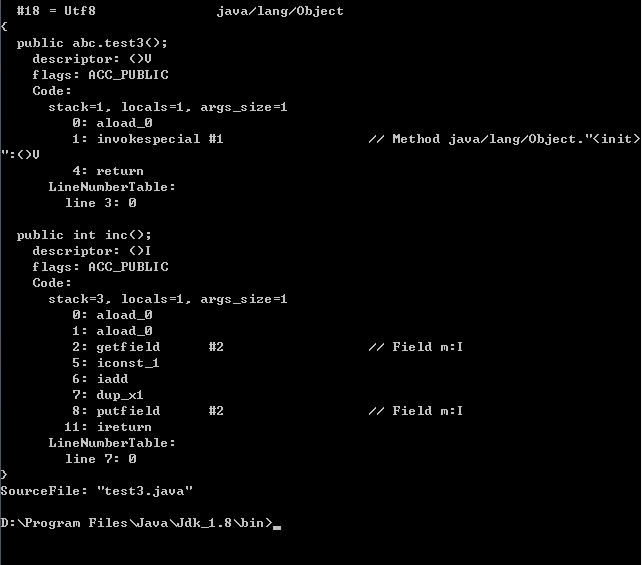
00 09 00 00 00 24 00 03 00 01 00 00 00 0C 2A 2A

B4 00 02 04 60 5A B5 00 02 AC 00 00 00 01 00 0A

00 00 00 06 00 01 00 00 00 07 00 01 00 0D 00 00

00 02 00 0E

Javap解析如下：



### 类索引

这个类是啥，父类是啥，怎么访问之类的，包括自己名字，父类名字，接口列表。

自己名字，常量池#3

6E 67 2F 4F 62 6A 65 63 74 00 21 00 03 00 04 00

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

父类名字，常量池#4

6E 67 2F 4F 62 6A 65 63 74 00 21 00 03 00 04 00

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

没有接口，数量为0

6E 67 2F 4F 62 6A 65 63 74 00 21 00 03 00 04 00

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

### 字段集合

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| u2 | access\_flags | 1 |
| u2 | name\_index | 1 |
| u2 | descriptor\_index | 1 |
| u2 | attributes\_count | 1 |
| attribute\_info | attributes | attributes\_count |

字段修饰符放在**access\_flags**中，占2字节，其值为0x0002，可见这个字段由private修饰，与访问标志位十分相似

|  |  |  |
| --- | --- | --- |
| 标志名称 | 标志值 | 含义 |
| ACC\_PUBLIC | 0x0001 | 字段是否为public |
| ACC\_PRIVATE | 0x0002 | 字段是否为private |
| ACC\_PROTECTED | 0x0004 | 字段是否为protected |
| ACC\_STATIC | 0x0008 | 字段是否为static |
| ACC\_FINAL | 0x0010 | 字段是否为final |
| ACC\_VOLATILE | 0x0040 | 字段是否为volatile |
| ACC\_TRANSIENT | 0x0080 | 字段是否为transient |
| ACC\_SYNTHETIC | 0x1000 | 字段是否为编译器自动产生 |
| ACC\_ENUM | 0x4000 | 字段是否为enum |

分析字段

先是长度，00 01，说明只有一个字段。

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

访问类型是 00 02说明私有的。

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

名字索引是 00 05常量#5，utf8的m

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

描述索引 00 06常量#6，utf8的I，说明是int型

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

属性表 00 00 说明没有属性

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

### 方法表集合

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| u2 | access\_flags | 1 |
| u2 | name\_index | 1 |
| u2 | descriptor\_index | 1 |
| u2 | attributes\_count | 1 |
| attribute\_info | attributes | attributes\_count |

数据项的含义非常相似，仅在访问标志位和属性表集合中的可选项上有略微不同

由于ACC\_VOLATILE标志和ACC\_TRANSIENT标志不能修饰方法，所以**access\_flags**中不包含这两项，同时增加ACC\_SYNCHRONIZED标志、ACC\_NATIVE标志、ACC\_STRICTFP标志和ACC\_ABSTRACT标志

|  |  |  |
| --- | --- | --- |
| 标志名称 | 标志值 | 含义 |
| ACC\_PUBLIC | 0x0001 | 字段是否为public |
| ACC\_PRIVATE | 0x0002 | 字段是否为private |
| ACC\_PROTECTED | 0x0004 | 字段是否为protected |
| ACC\_STATIC | 0x0008 | 字段是否为static |
| ACC\_FINAL | 0x0010 | 字段是否为final |
| ACC\_SYNCHRONIZED | 0x0020 | 字段是否为synchronized |
| ACC\_BRIDGE | 0x0040 | 方法是否是由编译器产生的桥接方法 |
| ACC\_VARARGS | 0x0080 | 方法是否接受不定参数 |
| ACC\_NATIVE | 0x0100 | 字段是否为native |
| ACC\_ABSTRACT | 0x0400 | 字段是否为abstract |
| ACC\_STRICTFP | 0x0800 | 字段是否为strictfp |
| ACC\_SYNTHETIC | 0x1000 | 字段是否为编译器自动产生 |

方法个数 00 02，表示2个方法

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

07 00 08 00 01 00 09 00 00 00 1D 00 01 00 01 00

访问标志 00 01，表示public

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

07 00 08 00 01 00 09 00 00 00 1D 00 01 00 01 00

名称 00 07 表示常量池 #7 <init>

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

07 00 08 00 01 00 09 00 00 00 1D 00 01 00 01 00

描述 00 08 表示常量池 #8 （）V 表示无参数，无返回值

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

07 00 08 00 01 00 09 00 00 00 1D 00 01 00 01 00

属性表长度为 00 01表示有一个属性

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

07 00 08 00 01 00 09 00 00 00 1D 00 01 00 01 00

属性值为 00 09 表示这是一段 Code

00 00 01 00 02 00 05 00 06 00 00 00 02 00 01 00

07 00 08 00 01 00 09 00 00 00 1D 00 01 00 01 00

在Class文件、属性表、方法表中都可以包含自己的属性表集合，用于描述某些场景的专有信息

与Class文件中其它数据项对长度、顺序、格式的严格要求不同，属性表集合不要求其中包含的属性表具有严格的顺序，并且只要属性的名称不与已有的属性名称重复，任何人实现的编译器可以向属性表中写入自己定义的属性信息。虚拟机在运行时会忽略不能识别的属性，为了能正确解析Class文件，虚拟机规范中预定义了虚拟机实现必须能够识别的9项属性：

|  |  |  |
| --- | --- | --- |
| 属性名称 | 使用位置 | 含义 |
| Code | 方法表 | Java代码编译成的字节码指令 |
| ConstantValue | 字段表 | final关键字定义的常量值 |
| Deprecated | 类文件、字段表、方法表 | 被声明为deprecated的方法和字段 |
| Exceptions | 方法表 | 方法抛出的异常 |
| InnerClasses | 类文件 | 内部类列表 |
| LineNumberTale | Code属性 | Java源码的行号与字节码指令的对应关系 |
| LocalVariableTable | Code属性 | 方法的局部变量描述 |
| SourceFile | 类文件 | 源文件名称 |
| Synthetic | 类文件、方法表、字段表 | 标识方法或字段是由编译器自动生成的 |

每种属性均有各自的表结构。这9种表结构有一个共同的特点，即均由一个u2类型的属性名称开始，可以通过这个属性名称来判段属性的类型

**Code属性**：Java程序方法体中的代码经过Javac编译器处理后，最终变为字节码指令存储在Code属性中。当然不是所有的方法都必须有这个属性（接口中的方法或抽象方法就不存在Code属性），Code属性表结构如下：

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| u2 | attribute\_name\_index | 1 |
| u4 | attribute\_length | 1 |
| u2 | max\_stack | 1 |
| u2 | max\_locals | 1 |
| u4 | code\_length | 1 |
| u1 | code | code\_length |
| u2 | exception\_table\_length | 1 |
| exception\_info | exception\_table | exception\_table\_length |
| u2 | attributes\_count | 1 |
| attribute\_info | attributes | attributes\_count |

max\_stack：操作数栈深度最大值，在方法执行的任何时刻，操作数栈深度都不会超过这个值。虚拟机运行时根据这个值来分配栈帧的操作数栈深度

max\_locals：局部变量表所需存储空间，单位为Slot，2个字节。并不是所有局部变量占用的Slot之和，当一个局部变量的生命周期结束后，其所占用的Slot将分配给其它依然存活的局部变量使用，按此方式计算出方法运行时局部变量表所需的存储空间

code\_length和code：用来存放Java源程序编译后生成的字节码指令。code\_length代表字节码长度，code是用于存储字节码指令的一系列字节流。

每一个指令是一个u1类型的单字节，当虚拟机读到code中的一个字节码（一个字节能表示256种指令，Java虚拟机规范定义了其中约200个编码对应的指令），就可以判断出该字节码代表的指令，指令后面是否带有参数，参数该如何解释，虽然code\_length占4个字节，但是Java虚拟机规范中限制一个方法不能超过65535条字节码指令，如果超过，Javac将拒绝编译

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 数量 |
| u2 | attribute\_name\_index | 1 |
| u4 | attribute\_length | 1 |
| u2 | max\_stack | 1 |
| u2 | max\_locals | 1 |
| u4 | code\_length | 1 |
| u1 | code | code\_length |
| u2 | exception\_table\_length | 1 |
| exception\_info | exception\_table | exception\_table\_length |
| u2 | attributes\_count | 1 |
| attribute\_info | attributes | attributes\_count |