# **jdk(java development Kit)**

### **java技术体系分为四个平台：**

java Card:支持小程序（Applets）运行在小内存设备上的平台

java me:主要是移动端；

java se：桌面应用，也称为J2SE

java ee:包含java se API外，对其做了大量扩充，这个版本叫J2SE（这些扩展一般以javax.\*作为包名开头，而已java.\*开头的包都是java SE的核心包，但由于历史原因，一些扩展包也也进入了核心包，因为核心包中也包含一些javax.\*的包名）

https://blog.csdn.net/xieyuooo/article/details/17452383

### java版本变迁

1,1995年5月23日,Oak语言改名为java，在SunWorld大会上正式发布java1.0版本；

2,1996年1月23日，JDK1.0发布，提供了一个纯解析执行的java虚拟机实现，代表技术：java虚拟机、Applet和AWT等

3,1997年2月19日，SUN发布了JDK1.1，提供了基础支持，技术代表有：jar文件格式、JDBC、JavaBean、RMI，语言支持：内部类（inner Class）和反射（Reflection）

4,1999年4月8日，一共发布了9个版本，1.10--1.18 ；并给每个工程版本命令，如宝石、南瓜、布鲁图 等；

5,1998年12月4日，发布JDK1.2，工程号：Playground(竞技场)，并拆分为3个方向，J2SE、J2EE、J2ME，代表技术：EJB、java Plug-in、java IDL、Swing等；第一次内置了JIT编译器（JDK1,2中并存过3个虚拟机，分别是：Classic VM、Hot Spot VM、Exact VM，其中Exact VM在Solaris平台出现过，其他两个都是内置JIT编译器的，之前版本所带的Classic VM只能以外挂形式使用JIT编译器

在语言和api级别上添加了strictfp关键字 和 及其强大的Collections集合；

6,1999年3月和7月，分别有JDK1.2.1和JDK1.2.2两个小版本

\*7,1999年4月27日，HotSpot虚拟机发布，又小公司开发，后被Sun公司收购，JDK1.3至今一直是JDK默认使用的虚拟机；

8，2000年5月8日，JDK1.3发布，增加一些类库，JNDI作为平台级服务提供等；

9,2002年2月13日，JDK1.4发布，真正走向成熟!技术特性：正则表达式、异常链、NIO、日志类、XML解析器、XSLT转换器等；

10,2002年，微软.NET Framework发布

11，JDK1.5发布，代号Tiger(老虎)，在语法易用性上做了重大改进，自动装箱、泛型、动态注解、枚举、可变长参数、遍历循环（foreach循环）等，并提供了java.util.concurrent并发包等； JDK1,.5后，命名改成JDK5、JDK6...

12,2006年12月11日，JDK1.6发布，代号Mustang(野马)，结束了J2EE、J2SE、J2ME的命名方式，改为JAVA SE6、JAVA EE6、JAVA ME6命名方式，改进：动态语言支持、提供编译API和微型HTTP服务的API，同时对java虚拟机内部做了大量改进，包括锁与同步、垃圾收集、类加载等算法

13,2006年11月13日在JavaOne大会上，sun宣堡把java开源，并建立OpenJDK组织对这些源码进行独立管理；

14,2009年2月19日，JDK1.7发布，代号Dolphin（海豚）;

15,2009年4月20日，Oracle宣布正式以74亿美元收购Sun公司，java商标从此归Oracle所有。。。

在JDK外围，出现了专为满足并行计算需求的计算框架，如Apache的Hadoop Map/Reduce，这是一个简单易懂的并行框架，能够运行在由上千个商用机器组成的大型集群上，并能以一种可靠的容错方式并行处理上TB级别的数据集；

# 自动内存管理机制

## 配置内存参数

-Xms20M 堆最小值20M

-Xmx20M 堆最大值20M

-Xmn10M 分配给新生代10M

-XX:SurvivorRatio=8 分配给新生代中的Eden Space区8m，如果共10M，from Survivor和to Survivor 就各1M

-Xoss 本地方法栈（对于Sun的HotSpa无效，因为虚拟机栈和本地方法栈是一起的）

-Xss10K 栈内存10K

-XX:PermSize=10M 方法区初始内存（包含常量池）（非堆）

-XX:MaxPermSize=10M 方法区最大内存（非堆）

-XX:MaxDirectMemorySize=10M 直接内存大小，不指定以Xmx 为准

-XX:+PrintGCDetail 打印GC时的内存，并且在程序结束时打印堆内存使用情况。

-verbose:class 跟踪类的加载和卸载信息

-XX:+TraceClassLoading 只根据类加载的信息

-XX:+TraceClassUnLoading 只跟踪类卸载的信息

-XX:PretenureSizeThreshold=1024 大于这个值的直接放入老年代，避免在新生代中大量内存拷贝

-XX:MaxTenuringThreshold=1 当对象在新生代的Survivor区每熬过一次Minor GC后，年龄加1，当年龄到达此配置值后就可以进入老年代中

-XX:MaxTenuringThreshold=1 新生代第二次GC就会放入老年代中，默认15

新生代GC（Minor GC）：新生代垃圾收集动作，频繁，速度快

老年代GC（Major GC/Full GC）：老年代的GC，一般会伴随这一次新生代的GC，比新生代GC慢10倍以上。

-XX:-HandlePromotionFailure:允许担保失败

-Xverify:none关闭加载类过程中的校验过程

## JAVA内存区域与内存溢出异常

**java与c++之间有一堵有内存动态分配和垃圾收集技术所围成的高墙，墙外面的人想进去，墙里面的人却想出来。。。**

因为java程序员把内存控制交给了java虚拟机，出现内存泄露和溢出的话，排查错误将会成为一个艰难的工作！

java虚拟机在执行java程序过程中会把它所管理的内存划分为若干个不同的数据区域。

这些区域都有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而存在，有写区域则是依赖用户线程的启动和结束而建立和销毁。

### 运行时数据区域

java虚拟机管理的内存包括以下几个运行时数据区域：

**程序计数器**（Program Couter Register）、**虚拟机栈**（VM Stack）、**方法区**（Method Areaa）、**本地方法栈**（Native Method Stack）、**堆**（Heap）、

#### 程序计数器

是一块较小的内存，它的作用可以看做是当前线程所执行的字节码的行号指示器。（概念模型，各种虚拟机实现方式不同）字节码解释器工作时通过改变这个计数器的值来选取下一条需要执行的字节码指令，java虚拟机的多线程我那个是通过线程轮流切换并分配处理器执行时间的方式来实现的，一个处理器（多处理器说的是一个内核）只会执行一条线程的指令，所以每个线程都需要一个独立的程序计数器，我们称之为“线程私有”的内存。如果线程执行的是java方法，计数器记录的是虚拟机正在执行的字节码指令地址，如果是natvie方法，计数器为空，此内存是唯一一个在java虚拟机中没有规定任何out ofMemoryError情况的区域。

#### Java虚拟机栈

同计数器一样属于线程私有的，生命周期与线程相同。、

虚拟机栈描述的是java方法的内存模型：每个方法被执行的时候都会同时创建一个栈帧（栈帧是方法运行时的基础数据结构）（Stack Frame）用户存储局部变量表、操作栈、动态链接、方法出口等信息。每个方法的开始和结束对应一个栈帧在虚拟机中从入栈到出栈的过程。

常有人把java内存区分为堆内存和栈内存，这种比较粗糙，实际比较复杂，栈说的就是虚拟机栈，或者说虚拟机中的局部变量表部分，局部变量表存放了编译器可知的各种基本数据类型（8种）、对象引用（reference类型，不是对象本身可能是对象起始地址的引用地址，也可能指向一个代表对象句柄或其他相关的位置）和returnAddress类型（指向一条字节码指令地址）。64位长度的long和double类型会占2个局部变量空间（Slot），其余数据类型占1个。局部变量表所占空间在编译期完成分配。 会出现两种异常状态，请求的栈深度大于虚拟机允许的会抛出StackOverflowError异常。当支持内存动态扩展，但无法扩展到足够内存时候会抛出OutOfMemoryError异常。

#### 本地方法栈

与虚拟机栈类似，区别是本地方方法栈.是为虚拟机使用到的native方法服务的。各虚拟机实现方式不同，例如Sun的HotSpot虚拟机直接把方法栈和虚拟机栈合二为一了。

#### J**ava堆**

虚拟机中内存管理最大的一块，被所有线程共享，虚拟机启动时创建，唯一目的就是存放对象实例，但对着JIT编译器的发展，所有对象都在堆内存分配变的不那么绝对了。Java堆是垃圾收集器管理的主要区域，所以也也被称为“GC堆”。

内存回收角度看：由于现在收集器基本采用分代收集器算法，所以可细分为：新生代和老年代；再细一点有Eden空间、From Survivor空间、To Surivor空间等；

内存分配角度，可划分为多个线程私有的分配缓冲区（Thread Local Allocation Buffer，TLAB）；

无论如何划分都与内容无关，进一步划分只是为了更好的回收内存，或者更快分配内存。当前主流的虚拟机都是可扩展的，通过-Xms和-Xmx控制，如果没有可供扩展的内存，会抛出OutOfMemoryError异常。

#### 方法区

与java堆一样，各个线程共享的内存区域。

用于存储虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据；

很多人称为“永久代”，但并不等价，仅仅是因为HotSpot虚拟机团队把GC分带收集扩展至方法区；

虚拟机规范对这个区域很宽松，可以不实现垃圾收集，换言之，垃圾收集行为在这个区域较少出现，但并非永久。此区域内存回收的主要是针对常量池的回收和堆类型的卸载。但成绩不那么满意，在Sun公司的BUG列表中出现若干低版本导致的内存泄露。抛出

OutOfMemoryError异常。

#### 运行时常量池

好文：<http://blog.csdn.net/seu_calvin/article/details/52291082>

JDK6中

//堆中创建‘1’对象，常量池中生成‘1’

String s = new String("1") + newString("1");

//判断常量池中是否有‘11’，没有就生成，有就返回地址

s.intern();

JDK6中7

//堆中创建‘1’对象，常量池中增加‘1’

String s = new String("1") + newString("1");

//将s的引用存入常量池中

s.intern();

方法区的一部分，Class文件中除了类版本、字段、方法、接口，还有一项是常量池，用于编译器生成的各种字面量符号引用，这部分在类加载后存放于常量池中。

除了保存Class文件中的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池中。

运行时常量池相对于Class文件常量池的另外一个特征是具备动态性，并不要求常量一定只能在编译器产生，也就是并非编译器Class文件常量池的内容才能进入常量池，运行期间也可能将新的常量放入池中，例如String类的intern（）方法。运行时常量池是方法区的一部分，会受到方法区的限制，无法申请到内存时会抛出OutOfMemoryError异常。

#### 直接内存

不是虚拟机运行时数据区的一部分，也不是java虚拟机规范中定义的内存区域，但经常被使用，可能导致OutOfMemoryError异常。JDK4中一如了NIT（New input/output）类，一种基于通道与缓冲区的I/O方式，也可以使用Native函数库直接分配堆外内存，然后通过一个存在java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作。提供性能，避免在java堆和Naviti堆中来回复制数据。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 区域名称 | 线程 | 数据类型 | 内存大小 | 异常信息 |
| 程序计数器 | 线程私有 | 当前线程字节码的行号指示器，Natvie方法为空 | 无 |  |
| Java虚拟机栈 | 线程私有 | 每个方法执行时都有一个栈帧，用于存储局部变量表（主要）、操作栈、动态链接、方法出口等。局部变量表包括（基本数据类型、对象引用、returnAddree类型）局部变量表在编译器完成分配 | -Xss10k |  |
| 本地方法栈 | 线程私有 |  | -Xoss(无效，HotSport虚拟机不区分虚拟机栈和本地方法栈) |  |
| Java堆 | 共享 | 对象实例 | -Xms20M -Mxs20M |  |
| 方法区 | 共享 | 存储已被加载的类信息、常量、静态变量、即时编译器编译的代码 | -XX:PermSize=10M -XX:MaxPermSize=20M |  |
| 运行时常量 | 共享 | 方法区的一部分，编译器生成的各种字面量和符合引用，具有动态性，如String.intern() | 同方法区 |  |
| 直接内存 | 共享 |  | -XX:MaxDirectMemorySize=10M |  |

### 对象访问

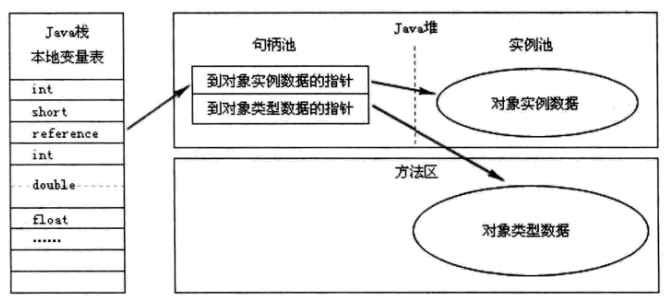
类似 Object obj = new Object() ;出现在方法体中，

Object obj 将反应到java栈的本地变量表中，作为一个reference引用类型出现，

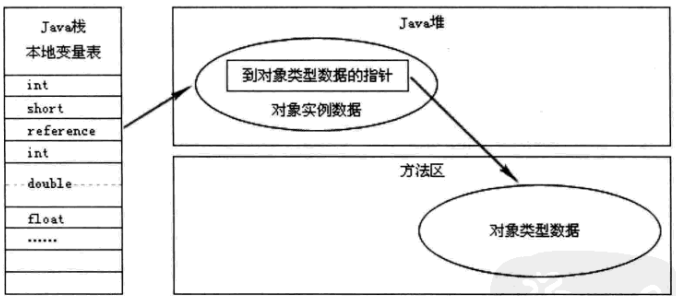
New Object() 将反应到java堆中，形成一块存储了Object类型所有实例数据值的结构化内存，长度不固定，另外，还必须包含能查找到此对象类型数据（如对象类型、父类、实现的接口、方法等）的地址信息，这些类型数据存储在方法区中。

reference引用类型在虚拟机中有两种访问方式：使用句柄 和直接指针。

使用句柄：堆中划分一块作为句柄池，reference中存储的是对象的句柄地址，句柄地址包含对象实例数据和类型数据各自的具体地址。



直接指针： reference中存储的就是对象地址。



句柄访问优势：reference中存储的是稳定的句柄地址，对象被移动时只改变句柄的实例数据指针，而reference本身不修改。

直接指针：速度更快，节省了一次指针定位的时间开销，Sun的HotSport虚拟机采用的直接指针方式。

### 虚拟机异常

在java虚拟机运行时数据区域中，除程序计数器外，其他几个运行时区域都有可能发生OutOfMemoryEerror异常 。

#### Java堆溢出

Java堆存储对象实例，所以只要不断创建对象实例，并保证对象一直有引用，堆总会溢出的。

设置堆大小的方法：-Xms20M -Xmx20M

List<OOMObject> obj = **new** ArrayList<OOMObject>();

**while**(**true**){

obj.add(**new** OOMObject());

}

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

#### 虚拟机栈和本地方法栈溢出

HotSpot虚拟机不区分虚拟机栈和本地方法栈

设置栈内存大小方式：-Xss10k

线程请求深度大于虚拟机允许深度，抛出：StackOverflowError异常。（递归方式增加栈深度）

虚拟机在扩展栈时无法申请到，抛出：OutOfMemoryError异常。

**int** leng = 1 ;

**public** **void** c(){

leng++ ;

c();

}

**public** **static** **void** main(String[] args) **throws** Throwable {

VMStack vm = **new** VMStack();

**try** {

vm.c();

} **catch** (Throwable e) {

System.***out***.println(vm.leng);

**throw** e ;

}

}

Exception in thread "main" java.lang.StackOverflowError

#### 运行时常量池溢出

最简单的方式是String.intern()方法。

设置运行时常量池（方法区）内存大小方式：-XX:PermSize=10M -XX:MaxPermSize=10M

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

**int** j = 0 ;

**while**(**true**)

list.add(String.*valueOf*(j++).intern());

Exception in thread "main" java.lang.OutOfMemoryError: PermGen space

#### 方法区溢出

方法区存储着Class类信息，所以需要产生大量的类，例如用反射和动态代理，spring的动态代理类，用cglib进行动态代理

设置运行时常量池（方法区）内存大小方式：-XX:PermSize=10M -XX:MaxPermSize=10M

while(true){

Enhancer enhancer = new Enhancer();

enhancer.setSuperclass(OOMObject.class);

enhancer.setUseCache(false);

enhancer.setCallback(new MethodInterceptor() {

@Override

public Object intercept(Object arg0, Method arg1, Object[] arg2,

MethodProxy proxy) throws Throwable {

return proxy.invoke(arg0, arg2);

}

});

enhancer.create();

}

Exception in thread "main" java.lang.OutOfMemoryError: PermGen space

#### 本地直接内存溢出

## 垃圾收集器与内存分配策略

Gc需要完成的三件事情：哪些需要回收、什么时候回收、如何回收。

### 对象已死？

#### 引用技术算法（怎么确定对象死了的算法）

给对象添加一个引用计数器，每当有一个地方引用它时，计数器就加1，当引用失效时，计数器值就减1，为0的对象就是不可能在被使用的，存在一个问题，无法解决对象之间相互循环依赖的问题

#### 根搜索算法（怎么确定对象死了的算法）

Java和C#使用的算法，基本思路是通过一系列名为（GC Eoots）的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连时，则证明对象是不可用的。

可用于GC Roots回收算法的对象包括：

1. 虚拟机栈（栈帧中的本地变量表）中的引用对象
2. 方法区中的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Native方法的引用对象。

#### 生存还是死亡

跟搜索算法中，不可达的对象并非是非死不可的，他们会经过两次标记，第一次搜索后发现没有与GC Roots相连接的引用链，会被第一次标记并进行一次筛选，筛选条件是是否需要执行finalize（）方法，如果没有实现此方法或者方法已被执行都认为是没有必要执行此方法。

如果有需要执行那么会把他放在F—Queue队列中。

**public** **class** FinalizeGc {

**public** **static** FinalizeGc *gc* = **null** ;

@Override

**protected** **void** finalize() **throws** Throwable {

**super**.finalize();

System.***out***.println("finalize 重生");

*gc* = **this** ;

}

**public** **static** **void** isDead() **throws** InterruptedException{

*gc* = **null** ;

System.*gc*();

Thread.*sleep*(500);

**if**(*gc* != **null**){

System.***out***.println("I am still live");

}**else**{

System.***out***.println("I am dead!");

}

}

**public** **static** **void** main(String[] args) **throws** InterruptedException {

*gc* = **new** FinalizeGc();

*isDead*();

*isDead*();

}

}

finalize 重生

I am still live

I am dead!

第一次垃圾回收后重生是因为在finalize（）方法中将对象重新建立了引用链，一个对象的finalize方法只会执行一次，所以第二次就被FC了。

#### 回收方法区

其实方法区（永久代）是有垃圾回收的，只是性价比较新生代低很多，方法区主要回收两部分：废弃常量 和 无用的类。

判定无用类：

1. 该类所有实例都被回收，也就是java堆中不存在此类的实例
2. 加载该类的ClassLoader被回收
3. 该类对应的java.lang.Class对象没有被引用，无法通过反射访问该类。

只有满足上述3个条件，才可能被回收，是否一定回被回收，虚拟机提供了参数。

<http://blog.csdn.net/wanggg2760/article/details/51912491>

### 垃圾收集算法

#### 标记-清除算法

标记-清除是最基础的收集算法，算法分为标记和清除两个阶段：首先标记处所有需要回收的对象，在标记完成之后统一回收掉所有标记的对象，两个缺点：1，效率问题，2，空间问题，会有大量不连续的内存碎片

#### 复制算法

将内存分为大小相等的两块，当一块内存用完了，就将还存活的对象复制到另一块上面，然后再把已经使用过的内存空间一次清理掉。代价是将原来内存缩小一半，现在商业虚拟机都采用这种收集方式来回收新生代，

#### 标记-整理算法

根据老年代的特点，提出了类似标记-清除的算法的另一种算法，当标记要清理的垃圾后，不马上清理，而是让所有存活对象都向一端移动，然后直接清理掉端边界以外的内存

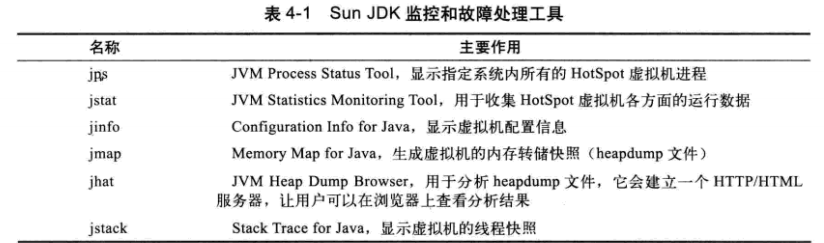
#### 分代收集算法

没有什么新的思想，只是根据对象的存活周期的不同将内存划分为几块，一般是把java堆分为新生代和老年代，新生代中用有大批会死去，所有选用复制算法，老年代中存活率高，所有选用标记-清理算法。

### 内存分配与回收策略

1. 对象内存分配往大的方向上说就是在堆上分配内存。
2. 对象再Eden区中分配，当Eden区中没有足够的空间进行分配时，虚拟机将进行一次Minor GC；（MinorGC：新生代GC；MajorGC/FullGC：老年代GC）
3. 需要大量连续内存空间的java对象会直接放入老年代，这样可以避免在新生代中大量内存拷贝，-XX:PretenureSizeThreshold=1024 大于这个值的直接放入老年代，怕遇到朝生熄灭的大对象。
4. 如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代
5. 在发生MinorGC时，虚拟机会检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间大小，如果大于，则改为直接进行一次FullGC。如果小于，则查看HandlePromotionFailure设置是否允许担保失败，如果允许，那只会进行MinorGC，如果不允许，则也要进行一次FullGC ；取平均值是种概率事件

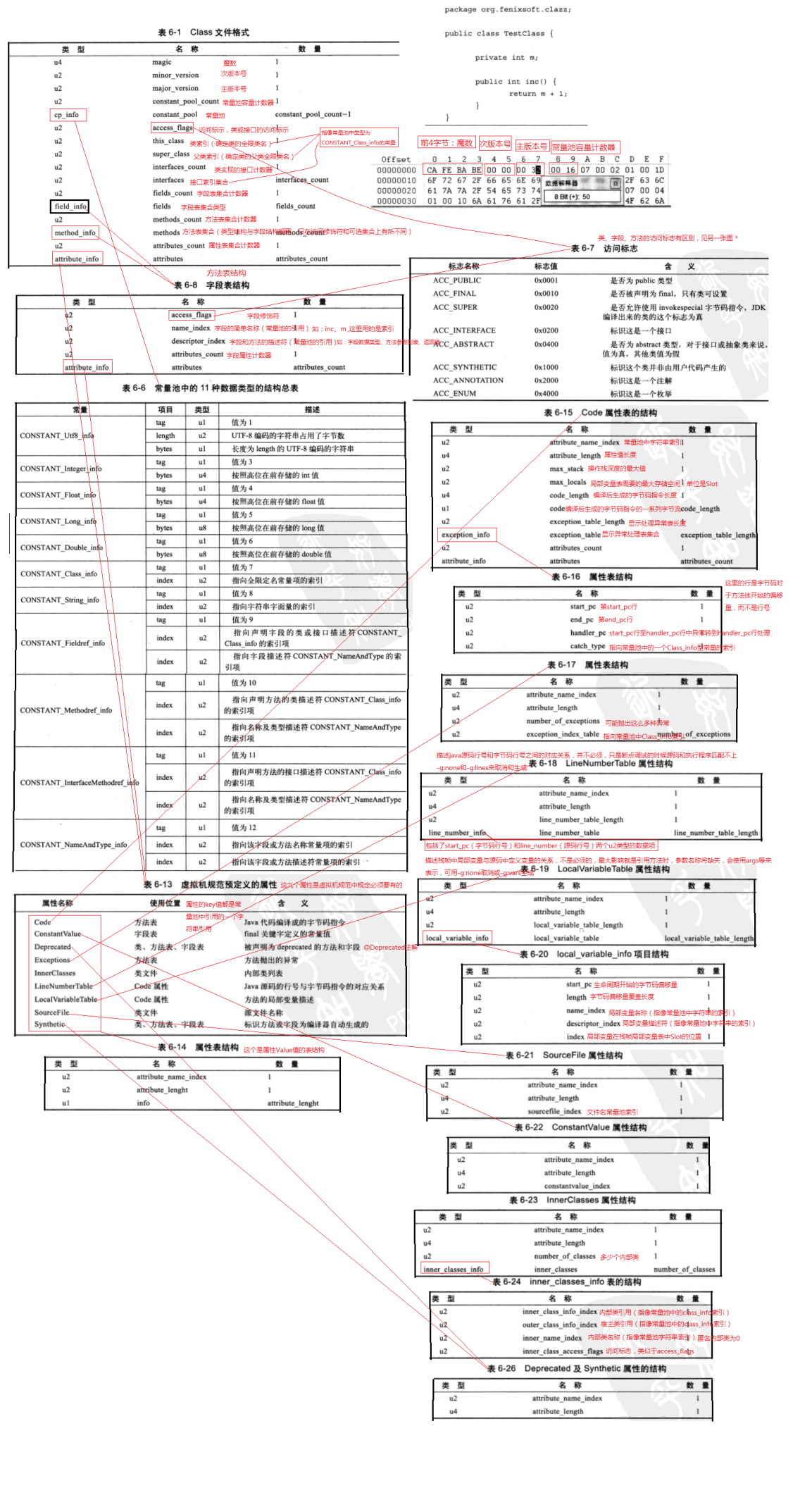
## 虚拟机性能监控与故障处理工具



# 虚拟机执行子系统

## 类文件结构

### Class文件结构

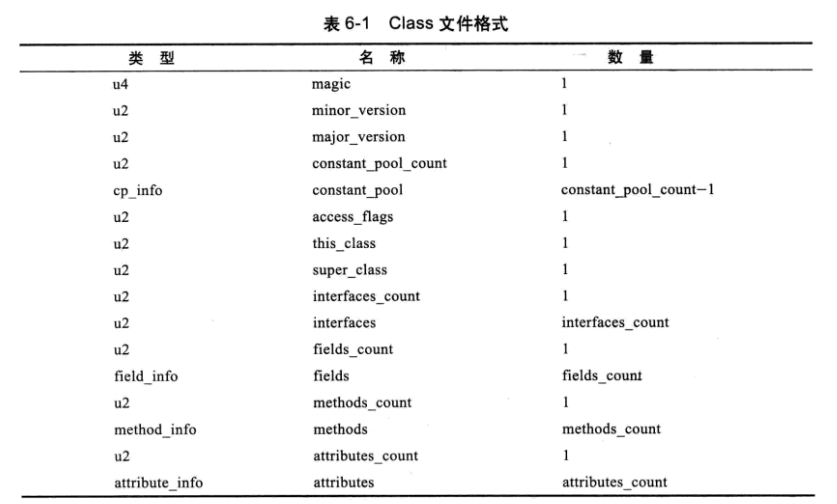


Class文件是一组以8位字节为基础单位的二进制流

Class文件采用一种类似C语言的伪结构来存储，只有两种数据类型：无符号数和表。

无符号数属于基本数据类型，以u1、u2、u4、u8代表1、2、4、8个字节，无符号数可以用来描述数字，索引引用，数量值，或按照UTF8构成的字符串值。

表是由多个无符号数或者其他表作为数据项构成的符合数据类型，习惯以\_info结尾，表用于描述有层次关系的复合结构数据，整个Class文件本质上就是一张表。



#### 常量池

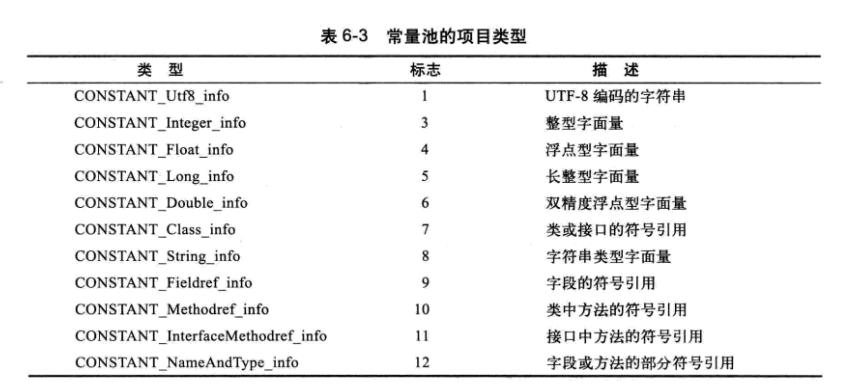
常量池是文件结构中与其他项目关联最多的数据类型，也是占用Class文件空间最大的数据项之一，起始位置是一个u2类型的容量计数器，从1而不是0开始，0代表不引用任何一个常量池项目时会用到，Class结构中只有常量池容量是从1计数，其他都是0

常量池中存放着两大类常量：字面量和符号引用。

字面量相当于常量，如：字符串、被声明为final的常量值。

符号引用包括三类：1，类和接口全限类名；2，字段的名称和描述符；3，方法的名称和描述符。

Java在编译时没有像C一样连接的步骤，而是在虚拟机加载Class文件时候进行动态连接。当虚拟机运行时，需要从常量池获得对应的符号引用，再在类创建时或运行时解析并翻译到具体的内存地址中。

常量池中每一项常量都是一个表，共有11种结构各不相同的表结构数据，共同特点：第一位是一个u1类型的标志位，代表这个常量属于哪种常量类型。

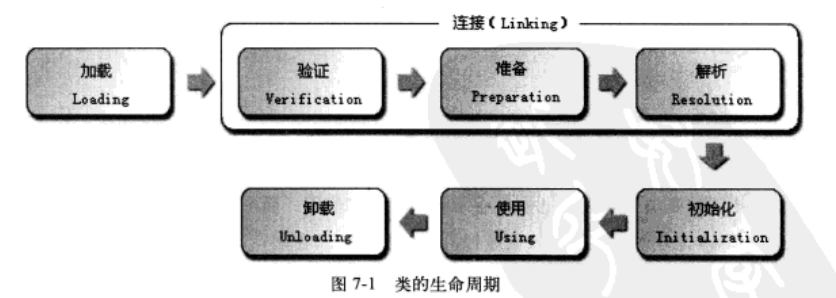
### 虚拟机类加载机制

虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验，转换解析和初始化，最终形成可以被虚拟机直接使用的java类型，这就是虚拟机的类加载机制。

Java语言是在程序运行时进行加载和连接的，所以java天生可以动态扩展。

解析阶段有可能在初始化之后才进行，为了支持java的运行时绑定

有且只有遇到new（实例化）、getstatic（读取静态字段）、putstatic（设置静态字段）、invokestatic（调用静态方法）这4种情况虚拟机才会对类进行初始化。



#### 虚拟机类加载机制

##### 加载Loading

1. 通过一个类的全限类名来获取定义此类的二进制字节流；（压缩包读取jar、war,网络读取applet，运行时生成Proxy代理，其他文件生成jsp，数据库读取等方式）
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
3. 在java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口

##### 验证

-Xverify:none关闭加载类过程中的校验过程

连接阶段的第一步，目的为了确保Class文件的字节流中包含的信息符合虚拟机要求。

一般完成四步骤：

文件格式验证：字节流是否符合Class文件格式的规范；（魔数、版本号、常量池类型等等）

元数据验证：对字节码描述的信息进行语义分析，保证符合java语言规范；（是否有父类等）

字节码验证：最复杂，进行数据流和控制流分析，对类的方法体进行校验

符号引用验证：将符号引用转化为直接引用，发生在连接的第三个阶段-解析阶段中发生，可以看做是对常量池中各种符号引用进行匹配性校验。

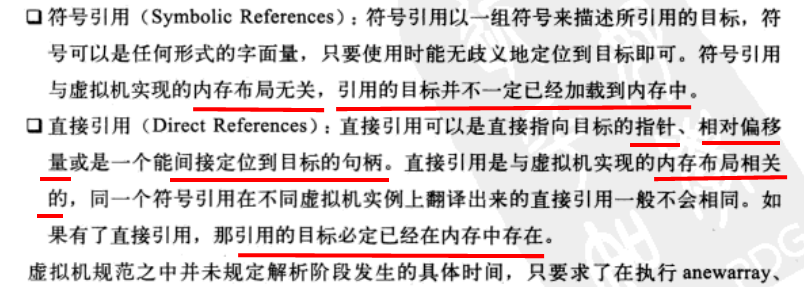
##### 准备

为类变量设置初始值并在方法区中分配内存（和加载中第二步混淆？），类变量是被static修饰的变量，不包括实例变量，被final修饰的变量会直接赋值。

##### 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

符号引用如Class文件中CONSTANT\_Class\_info、CONSTANT\_Fieldred\_info、CONSTANT\_Merthodred\_info等类型的常量。



##### 初始化

<client>()类构造器：jvm第一次加载的时候调用，静态变量初始化和静态块的执行

<init>()实例构造器：实例创建出来的时候执行，包括new 等..

初始化阶段是执行类构造器方法的过程,而类构造器的作用是静态变量初始化和静态块的执行。

静态块只能范文到定义在之前的变量，之后的变量只能赋值，不能访问。

<client>()不需要显示调用父类构造器，虚拟机自己会调

<client>()可以不生成，如果类中没有静态块和静态变量

接口和类都会生成<client>()，接口中的<client>()不需要先执行父类的<client>()方法，只有当父接口中变量被使用时，接口实现类初始化也不会执行接口的<client>()方法

虚拟机会保证<client>()在多线程中加锁和同步

#### 类加载器

通过一个类的全限类名来获取描述此类的二进制字节流的这个动作代码模块称为‘类加载器’

比较两个雷是否相等，只有在这两个类是由同一个类加载器加载的前提之下才有意义

虚拟机角度只有两种类加载器，一种是启动类加载器，用C++编写，是虚拟机一部分，另一种就是其他类加载器，继承自java.lang.ClassLoader

开发人员角度更新一点，分三种：

1. 启动类加载器：<JAVA\_HOME>\lib下或-Xbootclasspath参数下的，指定名称的类库名，开发者不可使用此加载器
2. 扩展类加载器：sun.misc.Launcher$ExtClassLoader实现，加载<JAVA\_HOME>\lib\ext下和java.ext.dirs系统变量下的类库，开发者可以使用此加载器
3. 应用程序类加载器：sun.misc.Launcher$AppClassLoader实现，是ClassLoader抽象类的getSystemClassLoader返回值，加载类路径上指定的类库

##### 双亲委派模型

类加载器间的父子关系一般不会是继承关系，都是使用组合关系来复用代码。

工作模式：类加载器收到加载请求，先委派父类加载器完成，父类无法完成，子类才去加载，避免不同jar下同路径同名的类被不同的加载器加载

##### 破坏双亲委派模型

### 虚拟机字节码执行引擎

执行引擎在执行java代码的时候可能有解释执行（通过解释器执行）和编译执行（通过即时编译器产生代码执行）两种选择。

#### 运行时栈帧结构

栈帧（Stack Frame）是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈（Virtual Machine Stack）的栈元素。栈帧存储了方法的局部变量表、操作数栈、动态链接和犯法返回地址等信息。每一个方法从调用到完成对应着一个栈帧从虚拟机入栈到出栈的过程。

在编译代码的时候，栈帧中需要多大的局部变量、多深的操作数栈都已经完全确定了，并写入Code属性中，so，一个栈帧的内存不会受到运行期变量的影响。活动线程中，只有栈顶的栈帧有效，称为当前栈帧。

##### 局部变量表

是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。

Code属性的max\_locals数据项确定了分配最大的局部变量表的容量。

局部变量表的容量以变量槽（Slot）为最小单位。

局部变量表中第0位索引的Slot默认是用于传递方法所属对象实例的引用，方法中通过‘this’来访问这个隐藏的参数。

##### 操作数栈

后入先出，

同局部变量一样，操作数栈的最大深度也在编译的时候写入到Code属性的max\_stacks中。

如：算术运算时通过操作数栈进行，方法调用通过操作数栈进行参数传递

##### 动态连接

每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。字节码中方法调用指令就以常量池中指向方法的符号引用为参数。类加载时转为直接引用称为静态解析，运行期间转为直接引用称为动态链接。

##### 方法返回地址

调用一个方法后，有两种退出方式，一种是正常退出，一种异常退出，无论哪种都要恢复到它上层方法的执行状态，一般正常退出时，调用者PC计数器就可以作为返回地址，栈帧中可能会保存这个计数器值，异常退出时，通过异常处理器来确定

方法退出其实是当前方法栈帧的出栈，因此可能操作有：恢复上层方法局部变量表和操作数栈，返回值压入栈帧的操作数栈中，调整PC计数器指向方法下一行。

#### 方法调用

方法调用不等同于方法执行，方法调用任务是确定被调用方法的版本。

方法调用在Class文件中存储的只是符号引用，需要在类加载时确定目标方法的直接引用，这个特性给java带来了强大的动态扩展能力。

##### 解析

方法调用中，目标方法在Class文件中都是常量池的一个符号引用，

类加载的解析阶段，会将其中的一部分符号引用转化为直接引用，前期是：方法在程序真正运行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可改变的，这类方法的调用称为解析。主要有静态方法和私有方法两大类，这两种方法调用在类加载阶段进行解析。

Java虚拟机提供了4条方法调用字节码指令，分别是：

Invokestatic:调用静态方法，

Invokespecial:调用实例构造器<init>方法、私有方法和父类方法

Invokevirtual:调用所有的虚方法

Invokeinterface:调用接口方法，会在运行时再确定一个实现此接口的对象。

被Invokestatic、Invokespecial指令调用的方法，都可以在类加载的阶段，也就是解析阶段确定唯一然后把符号引用替换为直接引用。另外final修饰的方法也是非虚方法。

##### 分派

与解析阶段相对应的是分派，分派调用可能是静态的和动态的，根据宗量数分为单分派和多分派，两两组合就成了四种分派方式。

###### 静态分派

**public** **class** TestClass {

**static** **abstract** **class** HuMan{}

**static** **class** Man **extends** HuMan{}

**static** **class** WoMan **extends** HuMan{}

**public** **static** **void** sayHello(HuMan huMan){

System.***out***.println("say HuMan");

}

**public** **static** **void** sayHello(Man man){

System.***out***.println("say man");

}

**public** **static** **void** sayHello(WoMan woMan){

System.***out***.println("say woMan");

}

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

HuMan man = **new** Man();

HuMan woman = **new** WoMan();

TestClass.*sayHello*(man);

TestClass.*sayHello*(woman);

}

}

//outPut

//say HuMan

//say HuMan

Human man = new Man() ;

Human：称为变量的静态类型 或者 外观类型

Man：实际类型

静态类型的变化紧紧在使用时发生，变量本身的静态类型不会被改变，并且最终的静态类型是在编译器可知的；

而实际类型变化的结果再运行期才可确定，编译器并不知道

虚拟机在重载时是通过参数的静态类型而不是实际类型作为判断依据的。静态类型是编译器可知的，所以根据静态类型决定使用哪个重载的版本。

所有依赖静态类型来定位方法执行版本的分派动作，都称为静态分派，最典型的就是重载