# 第二章、java内存区域和内存溢出异常

## 2.2、运行时数据区域

**虚拟机内存模型**



## 2.3、hotspot虚拟机对象相关

### 2.3.1、对象的创建过程



### 2.3.2、对象的内存布局

### 2.3.3、对象的访问

#### 2.3.3.1、句柄访问方式

Java堆会划分一块内存来作为句柄池，reference中存储的就是对象的句柄地址，句柄中包含了实例数据与类型数据的具体地址信息

优点：reference中存储的是稳定的句柄地址，在对象移动时，只会改变句柄中示例数据指针，而reference并不需要改变

#### 2.3.3.2、直接指针访问

Java堆对象的布局中就必须考虑如何防止访问类型数据的相关信息，而，reference中存储的直接就是对象地址。

优点：速度快，节省了一次指针定位的时间开销。由于对象的访问在java中非常频繁，积少成多。 Hotspot就是这种方式。

## 2.4、outtofmemorryerror异常实战

## 对象死了？

### 1.引用计数算法

每个对象都又一个计数器，没有一个地方引用对象，计数器+1。引用失效，则计数器-1.只要计数器为0，则对象将被回收。

**缺点**：不能解决相互引用的问题。A对象中有个引用B对象的成员，B对象中有个引用A对象的成员。然而其他地方都并没有引用。根据计数器算法，依然就无法被回收。

### 2.可达性分析算法

通过一系列被称为“GC Roots”的对象作为起始点，从这些节点开始望下搜索，搜索所走的路径称为引用链，当一个对象到GCRoots没有任何引用链相连（gc对象到这个对象不可达）的时候。则证明该对象不可用。将被回收。

可以作为GC Roots的对象:

1.虚拟机栈（栈帧中的本地变量表）中引用的对象

2.方法区中的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.再谈引用

**强引用**:普遍存在于代码中，如Objecta=new Object();只要引用还在，垃圾收集器永远不会回收这些被引用的对象

**软应用**:用来描述一些还有用，但是不是必须的对象。对于这些软引用关联的对象，在系统发生内存溢出异常之前，将会把这些对象列入回收范围内，进入第二次回收。如果内存还是不够，则再抛出内存溢出异常。

**弱引用**:用于描述非必须的对象，强度比软引用更弱，被弱引用关联的对象，只能生存到下一次垃圾收集发生之前，。当垃圾收集器工作时，无论当前内存是否足够，都会回收只被弱引用关联的对象。

**虚引用**:它是一种最弱的引用关系。一个对象是否有虚引用，完全不影响其生存时间构成影响，也无法通过虚引用来获取对象实例。需引用的唯一目的就是这个对象被收集器回收时到一个系统通知。

### 4.生存还是死亡

当一个对象经过可达性分析之后判定为**不可达**状态，那么它将会被标记并且进行一次筛选，筛选条件为“**是否有必要执行finalize方法**”。当对象没有覆盖finalize方法，或者已经执行过了finalize方法，以上两种情况，将被视为“没有必要执行”。

在被判定为有必要执行finalize时，那么这个对象会被放在一个叫做F-Queue的队列中，并在稍后有一个虚拟机创建的、低优先级的Finalizer线程去执行它的finalize方法。这时，已经被标记过一次的对象，则可以在finalize方法中进行一次自救，比如将this对象赋值引用链上的一个对象的某个成员变量上。

另外，finalize方法只会被系统执行一次。

### 5.方法区（永久代）的回收

方法区的垃圾回收效率较堆中的回收效率 较低。

方法区的回收主要两部分内容：废弃常量和无用的类

**废弃的常量判定**：常量池的任意一个字面量，在系统中没有任何一个对象引用该字面量，则被判定为无用，在gc时，会被清理出常量池。

**无用类的判定**：

1.该类的所有实例对象已经被回收

2.加载该类的ClassLoader已经被回收

3.该类对应的java.lang.Class对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法

虚拟机对满足以上条件的的类可以回收。但仅仅时可以。是否进行回收，HotSpot虚拟机提供了**-Xnoclassgc**参数进行控制，还可以使用**-verbose:class**以及**-XX:+TraceClassLoading、-XX:+TraceClassUnLoading**查看类加载和卸载信息。

在大量使用反射、动态代理、CGLib等byteCode框架、动态生成jsp以及OSGi这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

## 垃圾收集算法

### 1.标记-清除算法

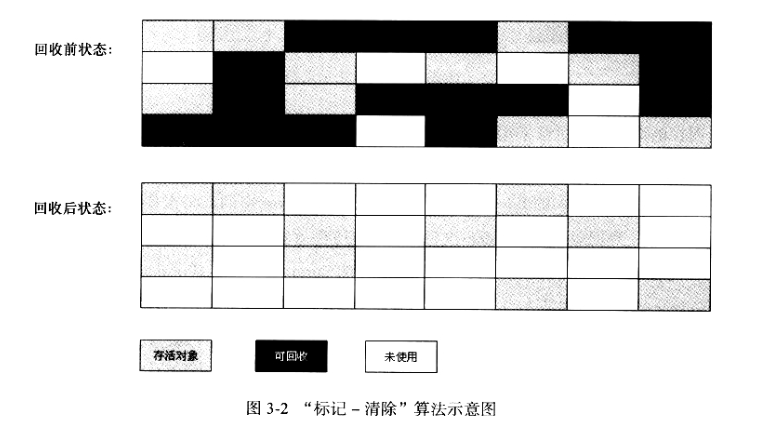
Mark-sweep算法最基础的垃圾收集算法。后续的算法都时基于该算法，弥补其不足实现的。

标记-清除算法缺点：

1.效率问题：标记和清除的效率都不高

2.空间问题：清除之后会产生大量不连续的的空间，会导致之后需要分配较大的对象时，无法找到足够大且连续的空间而不得不提前触发一次垃圾搜集操作。

算法示意图：



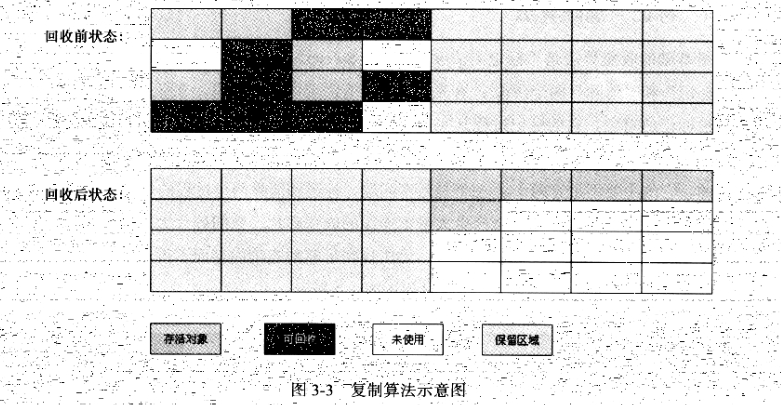
### 2.复制算法

为了解决效率问题，复制算法出现。它将内存划分为大小相等的两块。每次只使用其中一块，当这一块使用完了，就将还存活的对象复制到另一块上。然后再把自己已使用的内存空间全部一次性清除掉。

**优点**：每次都是整块内存进行回收，高效，也不存在不连续的问题。

**缺点**：将内存分为两块，所以实际用到的内存只有真正内存的一半。

**算法示意图**



经研究，新生代中98%的对象是‘朝生夕死‘，所以不需要按照1:1来划分空间。而是将新生代内存分为一块较大的Eden空间和两块较小的Survivor空间。每次使用Eden和其中一块Survivor空间，当回收时，将**Eden**和**survivor**还存活的对象复制到另一块Survivor空间中，最后一次性清理Eden和刚才用过的Survivor空间。HotSpot默认Eden和Survivor空间比例为8：1。所以浪费掉的内存只有10%。

提供参数-XX:SurvivorRatio=8 设置Eden区与Surivor区的比值

由于无法保证每次回收都只有不多于10%的对象存活，所以当Survivor空间不够时，就需要依赖其他内存（这里指老年代）进行分配担保。

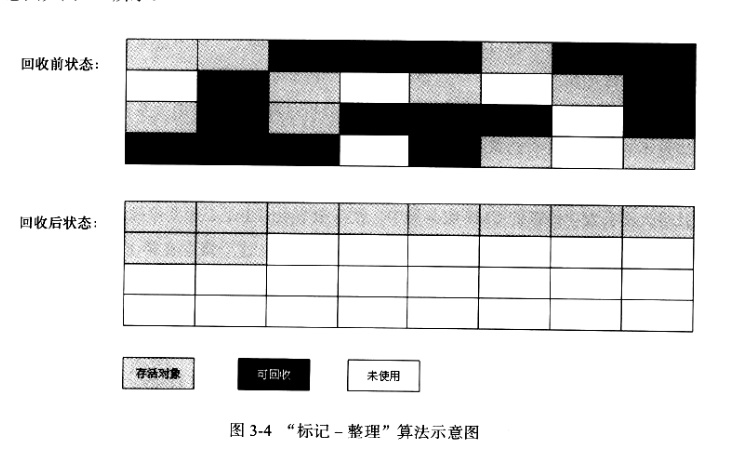
就是说如果另一块Survivor空间不足存放上一次新生代收集下来的对象，这些对象会通过**分配担保机制**进入老年代

### 3.标记整理算法

老年代存活的对象比较，所以每次复制的对象也多，浪费性能。所以老年代不能直接使用复制算法。

根据老年代特点，有人提出**标记整理算法。**标记与之前都标记是一样的，只是标记后不再直接清除，而是让所有的对象向一端移动，然后清除掉端边界以外的内存。

**算法示意图**



### 4.分代搜集算法

分代搜集算法：根据新生代和老年代不同的特点，采用不同的回收算法。

新生代每次保留的对象较少，就采用复制算法。

老年代每次回收的对象较少，就采用“标记-清理”或者“标记-整理”算法。

## 垃圾搜集器

Serial搜集器

Parnew收集器

Parallelscavenge收集器

Serialold收集器

Parallelold收集器

Cms收集器

## 内存分配与回收策略

### 1.大对象优先分配eden区

大多数情况下，对象在新生代Eden区分配。当空间不足时，会发生一次MinorGC。

虚拟机提供参数-XX：+PrintGCDetails,告诉虚拟gc时，打印内存回收日志

**Gc日志阅读**



33.125Gc发生的时间

GC gc类型

DefNew gc发生的区域

3324K->1521K(3712k) 当前区域 gc前的内存占用->gc后的内存占用（总内存大小）

0.0025925secs gc占用时间

3324k->1521（11904k） 堆内存的gc前的内存占用->gc后的内存占用（总内存大小）

### 2.大对象直接进入老年代

所谓大对象就是需要大片连续内存的java对象（如很长的字符串和数组）

大对象过多，容易导致提前触发gc

虚拟机提供参数 -XX：PretenureSizeThreshold，令大于这个值的对象，直接在老年代分配。这样做避免Eden区和Survivor区发生大量的内存复制。

### 3.长期存活的对线将进入老年代

虚拟机为每个对象定义了一个年龄计数器，如果对象在Eden出生并经过一次MinorGC后任然存活并且能被Survivor区容纳，将被移到Survivor空间，并且年龄设置为1.

对象在Survivor每熬过一次MinorGC，年龄就增加1.

当年龄达到一定值（默认是15），将被移到老年代。

提供参数-XX:MaxTenuringThreshold设置这个值

### 4.动态对象年龄判断

为了适应不同程序的内存情况，虚拟机不会要求对象一定要达到MaxTenuringThredshold才进入老年代

如果 Survivor空间相同年龄的对象占用内存总和超过Survivor区空间的一般，大于等于这个年龄的对象，将被移动到老年代

### 5.空间分配担保

在MinorGC发生前，虚拟会先检查老年代最大可用连续空间是否大于新生代所有对象总和，如果成立，那么MinorGC可以确保是安全的。

如果不成立，虚拟机会检查HandlerPromotionFailure设置值是否允许担保失败。

如果允许，那么虚拟机会检查老年代最大可用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，将尝试一次MinorGC，尽管会有风险。

如果小于，或者HandlerPromotionFailure不允许担保失败，这时将改为一个FullGC

## 类加载机制

### 1.类加载的时机

类的生命周期的七个阶段：加载、验证、准备、解析、初始化、使用、卸载

其中验证、准备、解析通称为连接

什么时候开始**加载**，java虚拟机规范并没有强行约定。

但对于**初始化**，虚拟机规范要求有且只有5种情况，会立即初始化（而加载、验证、准备必须在初始化之前）。

**解析**阶段则不一定要按顺序执行，这是为了支持java的运行时绑定。

**立刻初始化的五种情况**

1）遇到new、getstatic、putstatic、invokestatic这4条字节码的时候。相应的场景有实用**new实例化对象**、**读取或设置一个类的静态字段**（被final修饰，已经在编译期把结果放入常量池的静态字段除外）以及**调用一个类的静态方法**的时候

2）实用java.lang.reflect包进行反射的时候。如果类没有初始化，则需先初始化。

3）初始化一个子类时候，若父类没有初始化，则先初始化父类

4）当虚拟机启动时，用户指定要执行一个类的时候（包含main方法的类），虚拟机会先初始化这个主类。

5)当使用java 1.7动态语言支持的时候，如果一个java.lang.invoke.MethodHandler实例最后的解析结果REF\_getstatic、REF\_putstatic、REF\_invokestatic的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需先初始化

初始化借口的时候，第三种场景的时候，有区别：借口初始化的是时候，并不要求其父借口已经全部初始化了。

### 2.类加载的过程

#### 1）加载

加载的时候，虚拟机需要做三件事

1. 通过一个类的全限定名来获取定义此类的二进制字节流
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. 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口

由于第一点并没有生命字节流必须从class文件获取，所以有了后面jar、war、ear格式

#### 2）验证

验证主要是为了确保class文件中的字节流中包含的信息符合当前虚拟机的要求，不会危害自身行为安全。

如果所运行的代码（自己写的和第三方包）都已经反复验证过，可以使用**参数-Xverify:none**来关闭大部分的验证措施

1. **文件格式验证**

是否已魔数0xCAFEBABY开头

主次版本号是否在当前虚拟机处理范围内

常量池的常量是否有不被支持的常量类型

……

1. **元数据验证**

这个类是否有父类

这个类的父类是否继承了不被允许继承的类

如果不是抽象类，是否实现了父类或接口中要求实现的方法

类中的字段、方法是否与父类矛盾（例如覆盖了父类final字段或者出现不符合规则的重载）

1. **字节码验证**

通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的

1. **符号引用验证**

这一验证在解析中发生

可以看做是对自生以外（常量池中的各种符号引用）的信息进行校验。

符号引用中通过字符串描述的全限定名是否能找到对应类

符号引用的类、字段、方法的访问性（private、protected、public、default）是否可以被当前类访问到

#### 3）准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量的内存都将在方法区分配。

public static int value=123这个阶段初始化的值为0.而把value赋值123的操作是在编译后存放于类构造器<init>（）方法之中，所以把value赋值123，是在初始化阶段才回执行。

此阶段的其他基础数据类型都是初始化为该类型的零值

public static **final** value=123;此时初始化的就是123。赋值123则是在javac编译时生成了ConstantValue属性，立马赋值了

#### 4）解析

虚拟机将常量池内的符号引用替换成直接引用的过程

**符合引用**

以任意一组符号来描述所引用的目标，符号可以使任意形式的字面量，只要使用时能无歧义地定位目标既可以。符号引用与虚拟机实现的内存布局无关，引用的目标不一定已经加载到内存中

**直接引用**

可以使指向目标的指针、相对偏移量或者是一个能间接定位目标的句柄。与布局有关，直接引用的对象一定是内存中的。

#### 5）初始化

到了初始化阶段，才真正开始执行java代码（或者说字节码）。

是执行<clinit>()方法的时候.

<clinit>()方法是由编译期自动收集类中的类变量的赋值动作和静态语句块中的语句合并产生的，编译期收集的顺序是由源码出现的顺序决定，静态语句块只能访问到定义在这之前的静态变量。之后的静态变量可以赋值，不能访问。

<clinit>()方法与类的构造函数不同，它不需要显示的调用父类构造器，虚拟机会保证在子类的<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。因此虚拟机中第一个执行的<clinit>()方法是Object的

由于父类的<clinit>()方法先执行，所以父类的静态代码块会先执行。

<clinit>()方法不是必须的。如果一个类没有静态语句块，也没有对变量的操作，编译期可以不为该类生成<clinit>()方法

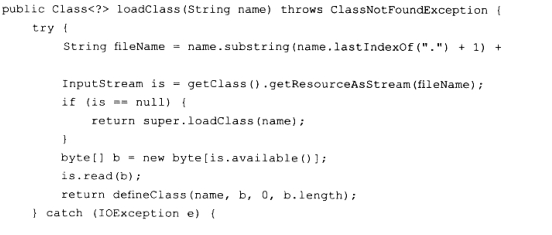
接口中没有静态语句块，但是有对变量的操作，所以也会生成<clinit>()方法。但与类不同，子接口的<clinit>()方法执行前，不必先执行父借口的<clinit>()方法。只有调用父借口的变量时才会使用。接口的实现类初始化的时候也不会执行接口的<clinit>()方法。

虚拟机会保证<clinit>()方法在多线程中的线程安全。

### 3.类加载器

若不是同一个类加载器加载的类（即时是同一个类，来源一个class文件）绝不相等。

自定义类加载器，实现ClassLoader，重写loadClass方法即可。



**启动类加载器**:负责将存放在<JAVA\_HOME>\lib目录中的，或这被-Xbootclasspath参数所指定的路径中的，并且被虚拟机所识别的类库加载到内存中。启动类加载器无法被java代码调用

**扩展类加载器**:它负责加载<JAVA\_HOME>\lib\ext目录中，或者被java.ext.dirs系统变量所指定的路径中的所有类库。开发者可以直接使用扩展类加载器。

**应用程序类加载器**:负责加载用户路径（classpath）上指定的类库。如果用户没有自定义加载器，一般情况下就是默认的加载器。

各个类加载器，是符合双亲委派模型的。在使用子类加载器加载类的时候，会去判断是否已经加载该类，若果没有，则调用父类的loadclass方法，父类无法加载，则再调用自己的loadclass方法。因此双亲委派模型下，各个类是有优先级的。

## 类文件结构

### 1.魔法数与版本号

#### 魔法数

Class文件头4个字节，表示魔法数，虚拟机检查它，判断该文件是否能够被加载到虚拟机，值为0XCAFEBABE

#### 版本号

5-6个字节是此版本号，7-8个字节是主版本号

Java的版本号的从45开始。周每发布一个大版本，就在者上面加1；

Java1.1 ---- 45

Java1.2 ---- 46

……

Java1.7 ---- 51

高版本的jdk能向下兼容低版本的class文件，低版本的jdk不能运行高本版的class文件（即便类的属性、方法都一样）

### 2.常量池

### 3访问标志

### 4类索引、父类索引与接口索引

### 5字段表集合

### 6方法表集合

### 7属性表集合

## Jvm参数调试

### 参数

**-Xms** 设置对内存最小值

**-Xmx** 设置堆内存最大值

**-XX:HeapDumpOnOutOfMemoryError**  虚拟机内存溢出时dump当前内存门快照

**-XX:HeapDumpPath** 内存的快照位置

**-Xoss** 设置本地方法栈（hotspot 不去分本地方法栈和虚拟机栈，所以该方法可能不生效，使用Xss参数设置栈容量）

**-Xss**:设置栈容量

**-XX:MaxPermSize** 最大方法区容量

**-XX:PermSize**

**-XX:MaxDirectMemorySize** 指定本机直接内存

**-versbose:gc** 查看垃圾回收过程

**-XX:+/- UserTLAB** 设置创建对象时，使用本地线程分配缓冲 保证分配内存时的线程安全

### 例子

设置对内存为10M，且不可扩展。内存溢出时，dump内存快照，存放路径d:\proj.bin

-Xms10m -Xmx10m

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=d:\proj.bin