链滴

JVM_03 运行时数据区 2- 堆

作者：wlgzs-sjl

原文链接：https://ld246.com/article/1602299982632

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20180605.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<h2 id="1-核心概述">1.核心概述</h2>
一个进程对应一个 jvm 实例，一个运行时数据区，又包含多个线程，这些线程共享了方法区和
，每个线程包含了程序计数器、本地方法栈和虚拟机栈。

一个 jvm 实例只存在一个堆内存，堆也是 java 内存管理的核心区域。
Java 堆区在 JVM 启动的时候即被创建，其空间大小也就确定了。是 JVM 管理的最大一块内存
间。（堆内存的大小是可以调节的）
《Java 虚拟机规范》规定，堆可以处于物理上不连续的内存空间中，但在逻辑上它应该被视为连
的。
所有的线程共享 java 堆，在这里还可以划分线程私有的缓冲区。（TLAB:Thread Local Allocatio
Buffer）.（面试问题：堆空间一定是所有线程共享的么？不是，TLAB 线程在堆中独有的）
《Java 虚拟机规范》中对 java 堆的描述是：所有的对象实例以及数组都应当在运行时分配在堆
。

从实际使用的角度看，“几乎”所有的对象的实例都在这里分配内存。（‘几乎’是因为可能存
在栈上）

数组或对象永远不会存储在栈上，因为栈帧中保存引用，这个引用指向对象或者数组在堆中的位
。
在方法结束后，堆中的对象不会马上被移除，仅仅在垃圾收集的时候才会被移除。
堆，是 GC(Garbage Collection，垃圾收集器)执行垃圾回收的重点区域。

<h3 id="1-1-配置jvm及查看jvm进程">1.1 配置 jvm 及查看 jvm 进程</h3>

编写 HeapDemo/HeapDemo1 代码
<pre><code class="highlight-chroma"><span class="highlight
cl">public class HeapDemo {
    public static vo
d main(String[] args) {
        System.out.pr
ntln("start...");
        try {
            Thread.sle
p(1000000);
        } catch (Inter
uptedException e) {
            e.printStac
Trace();
        }
        System.out.pr
ntln("end...");
    }
}
</code></pre>

首先对虚拟机进行配置，如图 Run-Edit configurations

<img src="https://ld246.com/images/img-loading.svg" alt="1725a5fe12fef584.png" data
src="https://b3logfile.com/file/2020/10/1725a5fe12fef584-9d80b4eb.png?imageView2/2/inte
lace/1/format/jpg">

在 jdk 的 bin 目录下下找到 jvisualvm 运行（或者直接终端运行 jvisualvm），查看进程，可以到我们设置的配置信息。

<img src="https://ld246.com/images/img-loading.svg" alt="1725a6053aec9841.png" dat-src="https://b3logfile.com/file/2020/10/1725a6053aec9841-8b08ba43.png?imageView2/2/interlace/1/format/jpg">

可以看到 HeapDemo 配置-Xms10m， 分配的 10m 被分配给了新生代 3m 和老年代 7m

<img src="https://ld246.com/images/img-loading.svg" alt="1725a616c775325d.png" daa-src="https://b3logfile.com/file/2020/10/1725a616c775325d-f8e84c0f.png?imageView2/2/interlace/1/format/jpg">
<h3 id="1-2-堆的细分内存结构">1.2 堆的细分内存结构</h3>
JDK 7 以前： 新生区 + 养老区 + 永久区

Young Generation Space：又被分为 Eden 区和 Survior 区 Young/New
Tenure generation Space：Old/Tenure
Permanent Space：Perm

<img src="https://ld246.com/images/img-loading.svg" alt="1725a665af15dfb5.png" dat-src="https://b3logfile.com/file/2020/10/1725a665af15dfb5-85802d9c.png?imageView2/2/interlace/1/format/jpg">
JDK 8 以后： 新生区 + 养老区 + 元空间

Young Generation Space：又被分为 Eden 区和 Survior 区 Young/New
Tenure generation Space：Old/Tenure
Meta Space：Meta

<h2 id="2-设置堆内存大小与OOM">2.设置堆内存大小与 OOM</h2>

Java 堆区用于存储 java 对象实例，堆的大小在 jvm 启动时就已经设定好了，可以通过 "-Xmx"和"-Xms"来进行设置

-Xms 用于表示堆的起始内存，等价于 -XX:InitialHeapSize

-Xms 用来设置堆空间（年轻代 + 老年代）的初始内存大小

-X 是 jvm 的运行参数
ms 是 memory start

-Xmx 用于设置堆的最大内存，等价于 -XX:MaxHeapSize

一旦堆区中的内存大小超过 -Xmx 所指定的最大内存时，将会抛出 OOM 异常
通常会将-Xms 和-Xmx 两个参数配置相同的值，其目的就是为了能够在 java 垃圾回收机制清理堆区后不需要重新分隔计算堆区的大小，从而提高性能
默认情况下，初始内存大小：物理内存大小/64;最大内存大小：物理内存大小/4

手动设置：-Xms600m -Xmx600m

查看设置的参数：

方式一： 终端输入 jps ， 然后 jstat -gc 进程 id
方式二： （控制台打印）Edit Configurations->VM Options 添加： -XX:+PrintGCDetails

<h3 id="2-1-查看堆内存大小">2.1 查看堆内存大小</h3>
<pre><code class="highlight-chroma">public class HeapSpaceInitial {
    public static void main(String[] args) {

        //返回Java虚机中的堆内存总量
        long initialMemory = Runtime.getRuntime().totalMemory() / 1024 / 1024;
        //返回Java虚机试图使用的最大堆内存量
        long maxMemory = Runtime.getRuntime().maxMemory() / 1024 / 1024;

        System.out.println("-Xms : " + initialMemory + "M");//-Xms : 245M
        System.out.println("-Xmx : " + maxMemory + "M");//-Xmx : 3641M

        System.out.println("系统内存大小为：" + initialMemory * 64.0 / 1024 + "G");//系统内存大小为：15.3125G
        System.out.println("系统内存大小为：" + maxMemory * 4.0 / 1024 + "G");//系统内存大小为：14.22265625G

        try {
            Thread.sleep(1000000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
</code></pre>
<h3 id="2-2-堆大小分析">2.2 堆大小分析</h3>
设置堆大小为 600m，打印出的结果为 575m，这是因为幸存者区 S0 和 S1 各占据了 25m，但他们始终有一个是空的，存放对象的是伊甸园区和一个幸存者区。

<h2 id="3-年轻代与老年代">3.年轻代与老年代</h2>

存储在 JVM 中的 java 对象可以被划分为两类：

一类是生命周期较短的瞬时对象，这类对象的创建和消亡都非常迅速
另外一类对象是生命周期非常长，在某些情况下还能与 JVM 的生命周期保持一致

Java 堆区进一步细分可以分为年轻代（YoungGen）和老年代（OldGen）
其中年轻代可以分为 Eden 空间、Survivor0 空间和 Survivor1 空间（有时也叫 frmo 区，to 区

<img src="https://ld246.com/images/img-loading.svg" alt="1725a6811c74d76b.png" da
a-src="https://b3logfile.com/file/2020/10/1725a6811c74d76b-7b211b4b.png?imageView2/2/
nterlace/1/format/jpg">
配置新生代与老年代在堆结构的占比

默认-XX：NewRatio=2，表示新生代占 1，老年代占 2，新生代占整个堆的 1/3
可以修改-XX:NewRatio=4，表示新生代占 1，老年代占 4，新生代占整个堆的 1/5

<img src="https://ld246.com/images/img-loading.svg" alt="1725a685f630e792.png" dat
-src="https://b3logfile.com/file/2020/10/1725a685f630e792-ef25a0e7.png?imageView2/2/int
rlace/1/format/jpg">

在 hotSpot 中，Eden 空间和另外两个 Survivor 空间缺省所占的比例是 8：1：1（测试的时候是
6：1：1），开发人员可以通过选项 -XX:SurvivorRatio 调整空间比例，如-XX:SurvivorRatio=8
几乎所有的 Java 对象都是在 Eden 区被 new 出来的
绝大部分的 Java 对象都销毁在新生代了（IBM 公司的专门研究表明，新生代 80% 的对象都是
朝生夕死”的）
可以使用选项-Xmn 设置新生代最大内存大小（这个参数一般使用默认值就好了）

<h2 id="4-图解对象分配过程">4.图解对象分配过程</h2>
<blockquote>
为新对象分配内存是件非常严谨和复杂的任务，JVM 的设计者们不仅需要考虑内存如何分配、
哪里分配的问题，并且由于内存分配算法与内存回收算法密切相关，所以还需要考虑 GC 执行完内存
收后是否会在内存空间中产生内存碎片。
</blockquote>

new 的对象先放伊甸园区。此区有大小限制。
当伊甸园的空间填满时，程序又需要创建对象，JVM 的垃圾回收器将对伊甸园区进行垃圾回收（
inor GC),将伊甸园区中的不再被其他对象所引用的对象进行销毁。再加载新的对象放到伊甸园区
然后将伊甸园中的剩余对象移动到幸存者 0 区。
如果再次触发垃圾回收，此时上次幸存下来的放到幸存者 0 区的，如果没有回收，就会放到幸存
1 区。
如果再次经历垃圾回收，此时会重新放回幸存者 0 区，接着再去幸存者 1 区。
啥时候能去养老区呢？可以设置次数。默认是 15 次。·可以设置参数：-XX:MaxTenuringThresh
ld=？进行设置。
在养老区，相对悠闲。当老年区内存不足时，再次触发 GC：Major GC，进行养老区的内存清理

若养老区执行了 Major GC 之后发现依然无法进行对象的保存，就会产生 OOM 异常。

总结
**
<pre><code class="highlight-chroma"><span class="highlight
cl">针对幸存者s0,s1区：复制之后有交换，谁空谁是to
</code></pre>

**
**
<pre><code class="highlight-chroma"><span class="highlight
cl">关于垃圾回收：频繁在新生区收集，很少在养老区收集，几乎不再永久区/元空间收集。
</code></pre>
**
<img src="https://ld246.com/images/img-loading.svg" alt="1725a6b5a5099b93.png" da
a-src="https://b3logfile.com/file/2020/10/1725a6b5a5099b93-1afcf579.png?imageView2/2/in
erlace/1/format/jpg">
<h3 id="4-2-对象分配的特殊情况">4.2 对象分配的特殊情况</h3>
<img src="https://ld246.com/images/img-loading.svg" alt="1725a6bccc4ef0bb.png" dat
-src="https://b3logfile.com/file/2020/10/1725a6bccc4ef0bb-fef7b536.png?imageView2/2/int
rlace/1/format/jpg">
<h3 id="4-3-代码举例">4.3 代码举例</h3>
<pre><code class="highlight-chroma"><span class="highlight
cl">public class HeapInstanceTest {
    byte[] buffer =
ew byte[new Random().nextInt(1024 * 200)];

    public static vo
d main(String[] args) {
        ArrayList<
eapInstanceTest> list = new ArrayList<HeapInstanceTest>();
        while (true) {
            list.add(n
w HeapInstanceTest());
            try {
                Thread.s
eep(10);
            } catch (Int
rruptedException e) {
                e.printS
ackTrace();
            }
        }
    }
}
</code></pre>
对应堆空间分配过程：
<img src="https://ld246.com/images/img-loading.svg" alt="1725a6c4bf09f89b.png" dat
-src="https://b3logfile.com/file/2020/10/1725a6c4bf09f89b-b6b8d97a.png?imageView2/2/in
erlace/1/format/jpg">
<h2 id="5-Minor-GC-Major-GC-Full-GC">5.Minor GC、Major GC、Full GC</h2>
<blockquote>
JVM 在进行 GC 时，并非每次都针对上面三个内存区域（新生代、老年代、方法区）一起回收
，大部分时候回收都是指新生代。
</blockquote>
针对 hotSpot VM 的实现，它里面的 GC 按照回收区域又分为两大种类型：一种是部分收集（Pa
tial GC），一种是整堆收集（Full GC）

部分收集：不是完整收集整个 Java 堆的垃圾收集。其中又分为：

新生代收集（Minor GC/Young GC）：只是新生代的垃圾收集

老年代收集（Major GC/Old GC）：只是老年代的垃圾收集

目前，只有 CMS GC 会有单独收集老年代的行为
注意，很多时候 Major GC 会和 Full GC 混淆使用，需要具体分辨是老年代回收还是整堆回收</l
>

混合收集（Mixed GC）：收集整个新生代以及部分老年代的垃圾收集

目前，之后 G1 GC 会有这种行为

整堆收集（Full GC）：收集整个 java 堆和方法区的垃圾收集

年轻代 GC（Minor GC）触发机制：

当年轻代空间不足时，就会触发 Minor GC，这里的年轻代满指的是 Eden 代满。Survivor 满不
引发 GC.(每次 Minor GC 会清理年轻代的内存，Survivor 是被动 GC，不会主动 GC)
因为 Java 队形大多都具备朝生夕灭的特性，所以 Monor GC 非常频繁，
般回收速度也比较快，这一定义既清晰又利于理解。
Minor GC 会引发 STW（Stop the World），暂停其他用户的线程，等垃圾回收结束，用户线
才恢复运行。

<img src="https://ld246.com/images/img-loading.svg" alt="1725a6cf9f5f7dda.png" data
src="https://b3logfile.com/file/2020/10/1725a6cf9f5f7dda-00b3c4e9.png?imageView2/2/inter
ace/1/format/jpg">

老年代 GC(Major GC/Full GC)触发机制

指发生在老年代的 GC,对象从老年代消失时，Major GC 或者 Full GC 发生了
出现了 Major GC，经常会伴随至少一次的 Minor GC（不是绝对的，在 Parallel Scavenge 收
器的收集策略里就有直接进行 Major GC 的策略选择过程）

也就是老年代空间不足时，会先尝试触发 Minor GC。如果之后空间还不足，则触发 Major GC</
i>

Major GC 速度一般会比 Minor GC 慢 10 倍以上，STW 时间更长
如果 Major GC 后，内存还不足，就报 OOM 了

Full GC 触发机制

触发 Full GC 执行的情况有以下五种

① 调用 System.gc()时，系统建议执行 Full GC，但是不必然执行
② 老年代空间不足
③ 方法区空间不足

④ 通过 Minor GC 后进入老年代的平均大小大于老年代的可用内存
⑤ 由 Eden 区，Survivor S0（from）区向 S1（to）区复制时，对象大小大于 To Space 可用内，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

说明：Full GC 是开发或调优中尽量要避免的，这样暂停时间会短一些。

<h2 id="6-堆空间分代思想">6.堆空间分代思想</h2>
为什么要把 Java 堆分代？不分代就不能正常工作了么

经研究，不同对象的生命周期不同。70%-99% 的对象都是临时对象。

新生代：由 Eden、Survivor 构成（s0,s1 又称为 from to），to 总为空
老年代：存放新生代中经历多次依然存活的对象

其实不分代完全可以，分代的唯一理由就是优化 GC 性能。如果没有分代那所有的对象都在一块，就如同把一个学校的人都关在一个教室。GC 的时候要找到哪些对象没用，样就会对堆的所有区域进行扫描，而很多对象都是朝生夕死的，如果分代的话，把新创建的对象放到一地方，当 GC 的时候先把这块存储“朝生夕死”对象的区域进行回收，这样就会腾出很大的空间出。

<h2 id="7-内存分配策略">7.内存分配策略</h2>

如果对象在 Eden 出生并经过第一次 Minor GC 后依然存活，并且能被 Survivor 容纳的话，将移动到 Survivor 空间中，把那个将对象年龄设为 1.对象在 Survivor 区中每熬过一次 MinorGC，年就增加一岁，当它的年龄增加到一定程度（默认 15 岁，其实每个 JVM、每个 GC 都有所不同）时，会被晋升到老年代中

对象晋升老年代的年龄阈值，可以通过选项 -XX：MaxTenuringThreshold 来设置

针对不同年龄段的对象分配原则如下：

优先分配到 Eden
大对象直接分配到老年代

尽量避免程序中出现过多的大对象

长期存活的对象分配到老年代
动态对象年龄判断

如果 Survivor 区中相同年龄的所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于年龄的对象可以直接进入到老年代。无需等到 MaxTenuringThreshold 中要求的年龄

空间分配担保

-XX: HandlePromotionFailure

<h3 id="代码示例">代码示例</h3>
分配 60m 堆空间，新生代 20m ，Eden 16m， s0 2m， s1 2m，buffer 对象 20m，Eden 区法存放 buffer， 直接晋升老年代。
<pre><code class="highlight-chroma">/** 测试：大对象直接进入老年代
 * -Xms60m -Xmx0m -XX:NewRatio=2 -XX:SurvivorRatio=8 -XX:+PrintGCDetails
 */
public class YoungOldAreaTest {
    // 新生代 20m Eden 16m， s0 2m， s1 2m
    // 老年代 40m
    public static void main(String[] args) {
        //Eden 区无存放buffer 晋升老年代
        byte[] buffer = new byte[1024 * 1024 * 20];//20m
    }
}
</code></pre>
日志输出
<img src="https://ld246.com/images/img-loading.svg" alt="1725a6dfdffbe6e8.png" data-src="https://b3logfile.com/file/2020/10/1725a6dfdffbe6e8-cd0a4089.png?imageView2/2/interlace/1/format/jpg">
<h2 id="8-为对象分配内存-TLAB-线程私有缓存区域-">8.为对象分配内存：TLAB（线程私有缓存域）</h2>
<img src="https://ld246.com/images/img-loading.svg" alt="1725a6e5328e2947.png" data-src="https://b3logfile.com/file/2020/10/1725a6e5328e2947-121e521b.png?imageView2/2/interlace/1/format/jpg">
为什么有 TLAB（Thread Local Allocation Buffer）

堆区是线程共享区域，任何线程都可以访问到堆区中的共享数据。
由于对象实例的创建在 JVM 中非常频繁，因此在并发环境下从堆区中划分内存空间是线程不安的。
为避免多个线程操作同一地址，需要使用加锁等机制，进而影响分配速度。

什么是 TLAB

从内存模型而不是垃圾收集的角度，对 Eden 区域继续进行划分，JVM 为每个线程分配了一个私缓存区域，它包含在 Eden 空间内。
多线程同时分配内存时，使用 TLAB 可以避免一系列的非线程安全问题，同时还能够提升内存分的吞吐量，因此我们可以将这种内存分配方式称之为快速分配策略。
所有 OpenJDK 衍生出来的 JVM 都提供了 TLAB 的设计。

说明

尽管不是所有的对象实例都能够在 TLAB 中成功分配内存，但 JVM 明确是是将 TLAB 作为内存配的首选。
在程序中，开发人员可以通过选项“-XX:UseTLAB“设置是否开启 TLAB 空间。

默认情况下，TLAB 空间的内存非常小，仅占有整个 EDen 空间的 1%，当然我们可以通过选项“ XX:TLABWasteTargetPercent “ 设置 TLAB 空间所占用 Eden 空间的百分比大小。
一旦对象在 TLAB 空间分配内存失败时，JVM 就会尝试着通过使用加锁机制确保数据操作的原子性，从而直接在 Eden 空间中分配内存。

<h3 id="TLAB对象分配过程">TLAB 对象分配过程</h3>

<h2 id="9-小结堆空间的参数设置">9.小结堆空间的参数设置</h2>

-XX:PrintFlagsInitial: 查看所有参数的默认初始值
-XX:PrintFlagsFinal：查看所有的参数的最终值（可能会存在修改，不再是初始值）

具体查看某个参数的指令：

jps：查看当前运行中的进程
jinfo -flag SurvivorRatio 进程 id： 查看新生代中 Eden 和 S0/S1 空间的比例

-Xms: 初始堆空间内存（默认为物理内存的 1/64）
-Xmx: 最大堆空间内存（默认为物理内存的 1/4）
-Xmn: 设置新生代大小（初始值及最大值）
-XX:NewRatio: 配置新生代与老年代在堆结构的占比
-XX:SurvivorRatio：设置新生代中 Eden 和 S0/S1 空间的比例
-XX:MaxTenuringThreshold：设置新生代垃圾的最大年龄(默认 15)
-XX:+PrintGCDetails：输出详细的 GC 处理日志

打印 gc 简要信息：① -XX:+PrintGC ② -verbose:gc

-XX:HandlePromotionFailure：是否设置空间分配担保

<h4 id="说明">说明</h4>
在发生 Minor Gc 之前，虚拟机会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的总间。

如果大于，则此次 Minor GC 是安全的
如果小于，则虚拟机会查看-XX:HandlePromotionFailure 设置值是否允许担保失败。（JDK 7 后的规则 HandlePromotionFailure 可以认为就是 true）

如果 HandlePromotionFailure=true,那么会继续检查老年代最大可用连续空间是否大于历次晋到老年代的对象的平均大小。

√ 如果大于，则尝试进行一次 Minor GC,但这次 Minor GC 依然是有风险的；
√ 如果小于，则改为进行一次 Full GC。

√ 如果 HandlePromotionFailure=false,则改为进行一次 Full GC。

在 JDK6 Update24 之后（JDK7），HandlePromotionFailure 参数不会再影响到虚拟机的空分配担保策略，观察 openJDK 中的源码变化，虽然源码中还定义了 HandlePromotionFailure 参数但是在代码中已经不会再使用它。JDK6 Update24 之后的规则变为只要老年代的连续空间大于新生对象总大小或者历次晋升的平均大小就会进行 Minor GC,否则将进行 Full GC。
<h2 id="10-堆是分配对象的唯一选择么-不是-">10.堆是分配对象的唯一选择么（不是）</h2>
在《深入理解 Java 虚拟机》中关于 Java 堆内存有这样一段描述：随着 JIT 编译期的发展与逃逸析技术逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化，所的对象都分配到堆上也渐渐变得不那么“绝对”了。
在 Java 虚拟机中，对象是在 Java 堆中分配内存的，这是一个普遍的常识。但是，有一种特殊况，那就是**如果经过逃逸分析（Escape Analysis)后发现，一个对象并没有逃逸出方法的话，那么可能被优化成栈上分配。**这样就无需在堆上分配内存，也无须进行垃圾回收了。这也是最常见的堆存储技术。
此外，前面提到的基于 OpenJDK 深度定制的 TaoBaoVM,其中创新的 GCIH(GCinvisible heap 技术实现 off-heap,将生命周期较长的 Java 对象从 heap 中移至 heap 外，并且 GC 不能管理 GCIH 部的 Java 对象，以此达到降低 GC 的回收频率和提升 GC 的回收效率的目的。

如何将堆上的对象分配到栈，需要使用逃逸分析手段。
这是一种可以有效减少 Java 程序中同步负载和内存堆分配压力的跨函数全局数据流分析算法。</l
>
通过逃逸分析，Java Hotspot 编译器能够分析出一个新的对象的引用的使用范围从而决定是否将这个对象分配到堆上。
逃逸分析的基本行为就是分析对象动态作用域：

当一个对象在方法中被定义后，对象只在方法内部使用，则认为没有发生逃逸。
当一个对象在方法中被定义后，它被外部方法所引用，则认为发生逃逸。例如作为调用参数传递其他地方中。

如何快速的判断是否发生了逃逸分析，就看 new 的对象实体是否有可能在方法外被调用。

<h3 id="代码分析">代码分析</h3>
<pre><code class="highlight-chroma"><span class="highlight
cl">public void method(){
    V v = new V();
    //use V
    //.....
    v = null;
}
</code></pre>
没有发生逃逸的对象，则可以分配到栈上，随着方法执行的结束，栈空间就被移除。
<pre><code class="highlight-chroma"><span class="highlight
cl">public static StringBuffer createStringBuffer(String s1,String s2){
    StringBuffer sb
= new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb;
}
</code></pre>
由于上述方法返回的 sb 在方法外被使用，发生了逃逸，上述代码如果想要 StringBuffer sb 不出方法，可以这样写：
<pre><code class="highlight-chroma"><span class="highlight
cl">public static String createStringBuffer(String s1,String s2){
    StringBuffer sb

```
= new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb.toStrig();
}
```

逃逸分析

```
/**
 * 逃逸分析
 *
 *  如何快速的判是否发生了逃逸分析，就看new的对象实体是否有可能在方法外被调用。
 */
public class EscapAnalysis {

    public EscapeAalysis obj;

    /*
    方法返回Escapenalysis对象，发生逃逸
     */
    public EscapeAalysis getInstance(){
        return obj == null? new EscapeAnalysis() : obj;
    }
    /*
    为成员属性赋值发生逃逸
     */
    public void setbj(){
        this.obj = ne EscapeAnalysis();
    }
    //思考：如果当的obj引用声明为static的？仍然会发生逃逸。

    /*
    对象的作用域仅当前方法中有效，没有发生逃逸
     */
    public void usescapeAnalysis(){
        EscapeAnalyss e = new EscapeAnalysis();
    }
    /*
    引用成员变量的，发生逃逸
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">     */
</span></span><span class="highlight-line"><span class="highlight-cl">    public void use
scapeAnalysis1(){
</span></span><span class="highlight-line"><span class="highlight-cl">        EscapeAnalys
s e = getInstance();
</span></span><span class="highlight-line"><span class="highlight-cl">        //getInstance
).xxx()同样会发生逃逸
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h3 id="参数设置">参数设置</h3>
<ul>
<li>在 JDK 6u23 版本之后，HotSpot 中默认就已经开启了逃逸分析。</li>
<li>如果使用了较早的版本，开发人员可以通过
<ul>
<li>-XX:DoEscapeAnalysis 显式开启逃逸分析</li>
<li>-XX:+PrintEscapeAnalysis 查看逃逸分析的筛选结果</li>
</ul>
</li>
</ul>
<p><strong>结论</strong></p>
<p>开发中能使用局部变量的，就不要使用在方法外定义。</p>
<h3 id="代码优化">代码优化</h3>
<p>使用逃逸分析，编译器可以对代码做如下优化：</p>
<ol>
<li>栈上分配：将堆分配转化为栈分配。如果一个对象在子程序中被分配，要使指向该对象的指针永不会逃逸，对象可能是栈分配的候选，而不是堆分配</li>
<li>同步省略：如果一个对象被发现只能从一个线程被访问到，那么对于这个对象的操作可以不考虑步</li>
<li>分离对象或标量替换：有的对象可能不需要作为一个连续的内存结构存在也可以北方问道，那么象的部分（或全部）可以不存储在内存，而是存储在 CPU 寄存器中。</li>
</ol>
<h4 id="栈上分配">栈上分配</h4>
<ul>
<li>JIT 编译器在编译期间根据逃逸分析的结果，发现如果一个对象并没有逃逸出方法的话，就可能优化成栈上分配。分配完成之后，继续在调用栈内执行，最后线程结束，栈空间被回收，局部变量对也被回收。这样就无须机型垃圾回收了</li>
<li>常见的栈上分配场景：给成员变量赋值、方法返回值、实例引用传递</li>
</ul>
<p><strong>代码分析</strong></p>
<p>以下代码，关闭逃逸分析（-XX:-DoEscapeAnalysi），维护 10000000 个对象，如果开启逃逸析，只维护少量对象（JDK7 逃逸分析默认开启）</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">/**
</span></span><span class="highlight-line"><span class="highlight-cl"> * 栈上分配测试
</span></span><span class="highlight-line"><span class="highlight-cl"> * -Xmx1G -Xms1
 -XX:-DoEscapeAnalysis -XX:+PrintGCDetails
</span></span><span class="highlight-line"><span class="highlight-cl"> */
</span></span><span class="highlight-line"><span class="highlight-cl">public class StackA
location {
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">        long start =
ystem.currentTimeMillis();
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        for (int i = 0; i
&lt; 10000000; i++) {
</span></span><span class="highlight-line"><span class="highlight-cl">            alloc();
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">        // 查看执行
间
</span></span><span class="highlight-line"><span class="highlight-cl">        long end = S
stem.currentTimeMillis();
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("花费的时间为：" + (end - start) + " ms");
</span></span><span class="highlight-line"><span class="highlight-cl">        // 为了方便
看堆内存中对象个数，线程sleep
</span></span><span class="highlight-line"><span class="highlight-cl">        try {
</span></span><span class="highlight-line"><span class="highlight-cl">            Thread.sle
p(1000000);
</span></span><span class="highlight-line"><span class="highlight-cl">        } catch (Inter
uptedException e1) {
</span></span><span class="highlight-line"><span class="highlight-cl">            e1.printSta
kTrace();
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    private static vo
d alloc() {
</span></span><span class="highlight-line"><span class="highlight-cl">        User user =
ew User();//未发生逃逸
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    static class User
{
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h4 id="同步省略">同步省略</h4>
<ul>
<li>线程同步的代价是相当高的，同步的后果是降低并发性和性能。</li>
<li>在动态编译同步块的时候，JIT 编译器可以借助逃逸分析来判断同步块所使用的锁对象是否只能被一个线程访问而没有被发布到其他线程。如果没有，那么 JIT 编译器在编译这个同步块的时候就会消对这部分代码的同步。这样就能大大提高并发性和性能。这个取消同步的过程就叫同步省略，也叫<trong>锁消除。</strong></li>
</ul>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">/**
</span></span><span class="highlight-line"><span class="highlight-cl"> * 同步省略说明
</span></span><span class="highlight-line"><span class="highlight-cl"> */
</span></span><span class="highlight-line"><span class="highlight-cl">public class Synch
onizedTest {
</span></span><span class="highlight-line"><span class="highlight-cl">    public void f() {
</span></span><span class="highlight-line"><span class="highlight-cl">        Object hollis
= new Object();
</span></span><span class="highlight-line"><span class="highlight-cl">        synchronized
hollis) {
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">            System.out
println(hollis);
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">    //代码中对holli
这个对象进行加锁，但是hollis对象的生命周期只在f（）方法中
</span></span><span class="highlight-line"><span class="highlight-cl">    //并不会被其他
程所访问控制，所以在JIT编译阶段就会被优化掉。
</span></span><span class="highlight-line"><span class="highlight-cl">    //优化为↓
</span></span><span class="highlight-line"><span class="highlight-cl">    public void f2()

</span></span><span class="highlight-line"><span class="highlight-cl">        Object hollis
= new Object();
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln(hollis);
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h4 id="分离对象或标量替换">分离对象或标量替换</h4>
<ul>
<li>（标量 Scalar）是指一个无法再分解成更小的数据的数据。Java 中的原始数据类型就是标量。</l
>
<li>相对的，那些还可以分解的数据叫做<strong>聚合量(Aggregate)</strong>，Java 中对象就
聚合量，因为它可以分解成其他聚合量和标量。</li>
<li>在 JIT 阶段，如果经过逃逸分析，发现一个对象不会被外界访问的话，那么经过 JIT 优化，就会
这个对象拆解成若干个其中包含的若干个成员变量来替代。这个过程就是标量替换。</li>
</ul>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class ScalarTest {
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">        alloc();
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d alloc(){
</span></span><span class="highlight-line"><span class="highlight-cl">        Point point =
new Point(1,2);
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">class Point{
</span></span><span class="highlight-line"><span class="highlight-cl">    private int x;
</span></span><span class="highlight-line"><span class="highlight-cl">    private int y;
</span></span><span class="highlight-line"><span class="highlight-cl">    public Point(int
x,int y){
</span></span><span class="highlight-line"><span class="highlight-cl">        this.x = x;
</span></span><span class="highlight-line"><span class="highlight-cl">        this.y = y;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>以上代码，经过标量替换后，就会变成</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public static void alloc(){
</span></span><span class="highlight-line"><span class="highlight-cl">    int x = 1;
</span></span><span class="highlight-line"><span class="highlight-cl">    int y = 2;
```

}
</code></pre>
可以看到，Point 这个聚合量经过逃逸分析后，发现他并没有逃逸，就被替换成两个标量了。那标量替换有什么好处呢？就是可以大大减少堆内存的占用。因为一旦不需要创建对象了，那么就不再要分配堆内存了。
标量替换为栈上分配提供了很好的基础。
<h4 id="逃逸分析小结">逃逸分析小结</h4>

关于逃逸分析的论文在 1999 年就已经发表了，但直到 JDK1.6 才有实现，而且这项技术到如今并不是十分成熟的。
其根本原因就是无法保证逃逸分析的性能消耗一定能高于他的消耗。虽然经过逃逸分析可以做标替换、栈上分配、和锁消除。但是逃逸分析自身也是需要进行一系列复杂的分析的，这其实也是一个对耗时的过程。
一个极端的例子，就是经过逃逸分析之后，发现没有一个对象是不逃逸的。那这个逃逸分析的过就白白浪费掉了。
虽然这项技术并不十分成熟，但是它也是即时编译器优化技术中一个十分重要的手段。
注意到有一些观点，认为通过逃逸分析，JVM 会在栈上分配那些不会逃逸的对象，这在理论上是行的，但是取决于 JVM 设计者的选择。据我所知，Oracle HotspotJVM 中并未这么做，这一点在逃分析相关的文档里已经说明，所以可以明确所有的对象实例都是创建在堆上。
目前很多书籍还是基于 JDK7 以前的版本，JDK 已经发生了很大变化，intern 字符串的缓存和静变量曾经都被分配在永久代上，而永久代已经被元数据区取代。但是，intern 字符串缓存和静态变量不是被转移到元数据区，而是直接在堆上分配，所以这一点同样符合前面一点的结论：对象实例都是配在堆上。
年轻代是对象的诞生、生长、消亡的区域，一个对象在这里产生、应用、最后被垃圾回收器收集结束生命。
老年代防止长生命周期对象，通常都是从 Survivor 区域筛选拷贝过来的 Java 对象。当然，也有殊情况，我们知道普通的对象会被分配在 TLAB 上，如果对象较大，JVM 会试图直接分配在 Eden 其位置上；如果对象太大，完全无法在新生代找到足够长的连续空闲空间，JVM 就会直接分配到老年代
当 GC 只发生在年轻代中，回收年轻对象的行为被称为 MinorGC。当 GC 发生在老年代时则被为 MajorGC 或者 FullGC。一般的，MinorGC 的发生频率要比 MajorGC 高很多，即老年代中垃圾收发生的频率大大低于年轻代。
