深入JVM系列-逃逸分析、同步省略、栈上分配、标量替换

Question

- 是不是所有的对象和数组都会在堆内存分配空间?

逃逸分析

逃逸分析 (Escape Analysis) 是目前 Java 虚拟机中比较前沿的优化技术。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。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要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，它可能被外部方法所引用，称为**方法逃逸**。甚至还有可能被外部线程访问到，譬如赋值给类变量或可以在其他线程中访问的实例变量，称为**线程逃逸**。

方法逃逸

什么是方法逃逸？举个例子吧

```
public static StringBuffer craeteStringBuffer(String s1, String s2) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb;
}
```

StringBuffer sb是一个方法内部变量，上述代码中直接将sb返回，这样这个StringBuffer有可能被其他方法所改变，这样它的作用域就不只是在方法内部，虽然它是一个局部变量，称其逃逸到了方法外部。

上述代码如果想要StringBuffer sb不逃出方法，可以这样写：

```
public static String createStringBuffer(String s1, String s2) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb.toString();
}
```

不直接返回 StringBuffer，那么StringBuffer将不会逃逸出方法。

线程逃逸

```
public class EscapeTest {

    public static Object globalVariableObject;

    public Object instanceObject;

    public void globalVariableEscape(){
        globalVariableObject = new Object(); //静态变量,外部线程可见,发生逃
逸
    }

    public void instanceObjectEscape(){
        instanceObject = new Object(); //赋值给堆中实例字段,外部线程可见,发生
逃逸
    }

    public Object returnObjectEscape(){
        return new Object();  //返回实例,外部线程可见，发生逃逸
    }

}
```

上述代码 中 globalVariableObject 对象暴露在外部，这样外部线程可见可以访问到，发生线程逃逸。

如果避免发生线程逃逸可以改写下面代码：

```
public void noEscape(){
        synchronized (new Object()){
            //仅创建线程可见,对象无逃逸
        }
        Object noEscape = new Object();  //仅创建线程可见,对象无逃逸
    }
}
```

优化

使用逃逸分析，编译器可以对代码做如下优化：

- 将堆分配转化为栈分配。如果某个对象在子程序中被分配，并且指向该对象的指针永远不会逃逸，该对象就可以在分配在栈上，而不是在堆上。在有垃圾收集的语言中，这种优化可以降低垃圾收集器运行的频率。
- 同步消除。如果发现某个对象只能从一个线程可访问，那么在这个对象上的操作可以不需要同步。
- 分离对象或标量替换。如果某个对象的访问方式不要求该对象是一个连续的内存结构，那么对象的部分（或全部）可以不存储在内存，而是存储在CPU寄存器中。

开启和关闭逃逸分析

在Java代码运行时，通过JVM参数可指定是否开启逃逸分析.

- `-XX:+DoEscapeAnalysis` 表示开启逃逸分析
- `-XX:-DoEscapeAnalysis` 表示关闭逃逸分析

> 从jdk 1.7开始已经默认开始逃逸分析，如需关闭，需要指定 `-XX:-DoEscapeAnalysis`

同步省略

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够被一个线程访问而没有被发布到其他线程。

如果同步块所使用的锁对象通过这种分析被证实只能够被一个线程访问，那么JIT编译器在编译这个同步块的时候就会取消对这部分代码的同步。这个取消同步的过程就叫同步省略，也叫锁消除。

如以下代码：

```
public void a() {
    Object obj = new Object();
    synchronized(obj) {
        System.out.println(obj);
    }
}
```

代码中对 obj 这个对象进行加锁，但是 obj 对象的生命周期只在a()方法中，并不会被其他线程所访问到，所以在JIT编译阶段就会被优化掉。优化成：

```
public void a() {
    Object obj = new Object();
    System.out.println(obj);
}
```

所以，在使用synchronized的时候，如果JIT经过逃逸分析之后发现并无线程安全问题的话，就会做锁消除。

同步省略对性能影响

以下列代码为例我们测试 同步消除对性能的影响

```
public class TestSync {
    public static void main(String[] args) throws IOException {
        long start = System.currentTimeMillis();
        for (int i = 0; i < 100000000; i++) {
            sync();
        }
        long end = System.currentTimeMillis();
        System.out.println("count = "+(end - start));
    }

    public static void sync() {
        byte[] b = new byte[65];
        synchronized (b) {
            b[0] = 1;
        }
    }
}
```

创建一个 sync() 内部创建一个 长度为 65 的字节数组 **默认数组长度大于64的是不会在栈上分配的**，我们都以堆上分配为例来测试锁消除带来的影响。 在main方法里面 执行 sync() 一亿次 并统计出执行时间。

编译

使用 `javac` 命令将上述 `.java` 文件编译出 `.class` 文件

```
$ javac
```

关闭同步省略

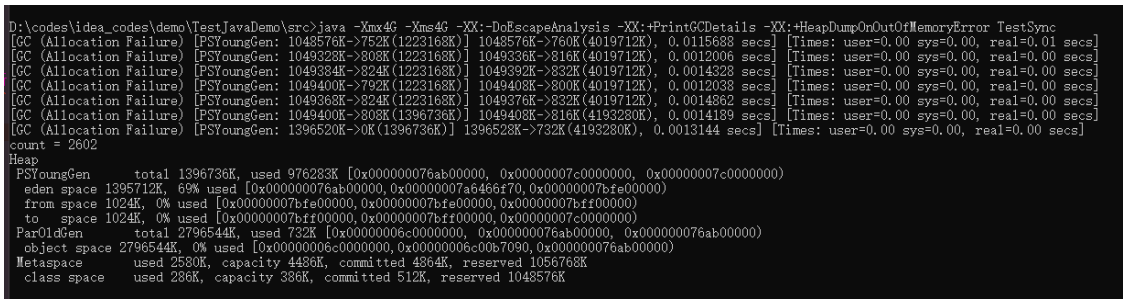
> 锁消除基于分析逃逸基础之上，开启锁消除必须开启逃逸分析

关闭逃逸分析执行上述编辑出 `.class` 文件

```
$ java -Xmx4G -Xms4G -XX:-DoEscapeAnalysis -XX:+PrintGCDetails -
XX:+HeapDumpOnOutOfMemoryError TestSync
```

注：

- `-Xmx4G -Xms4G`： 分配给 Java 堆大小为 4G
- `-XX:-DoEscapeAnalysis`：关闭逃逸分析
- `-XX:+PrintGCDetails`：打印出 `gc` 过程中详细信息
- `-XX:+HeapDumpOnOutOfMemoryError` 开启 `dump` 的选项

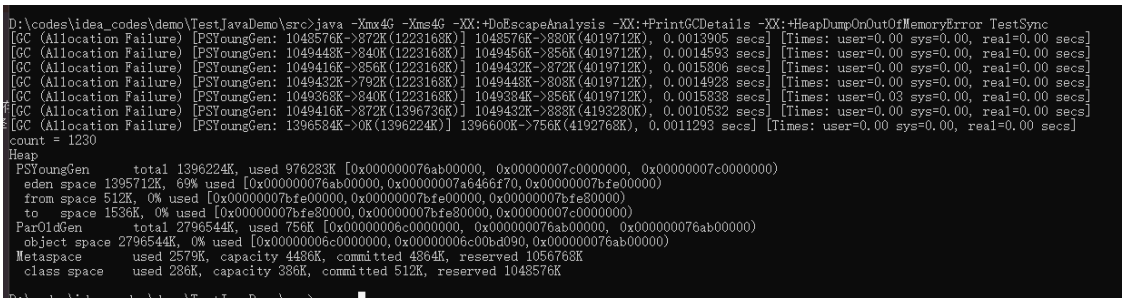


执行完毕耗时 2602 毫秒，并同时伴随着 `gc` 信息 在 Java heap 中占用的内存较高

开启同步省略

使用 `-XX:+DoEscapeAnalysis` 参数开启逃逸分析 执行上述代码

```
$ java -Xmx4G -Xms4G -XX:+DoEscapeAnalysis -XX:+PrintGCDetails -
XX:+HeapDumpOnOutOfMemoryError TestSync
```



开启逃逸分析执行上述代码 耗时 1230 毫秒

对象的栈上内存分配

我们知道，在一般情况下，对象和数组元素的内存分配是在堆内存上进行的。但是随着JIT编译器的日渐成熟，很多优化使这种分配策略并不绝对。JIT编译器就可以在编译期间根据逃逸分析的结果，来决定是否可以将对象的内存分配从堆转化为栈。

对象的栈上内存分配对性能影响

将上述代码修改成如下：

```
public class TestSync {
    public static void main(String[] args) throws IOException {
        long start = System.currentTimeMillis();
        for (int i = 0; i < 100000000; i++) {
            sync();
        }
        long end = System.currentTimeMillis();
        System.out.println("count = "+(end - start));
    }

    public static void sync() {
        byte[] b = new byte[2];
        synchronized (b) {
            b[0] = 1;
        }
    }
}
```

将 byte[] 长度 65 修改为 2

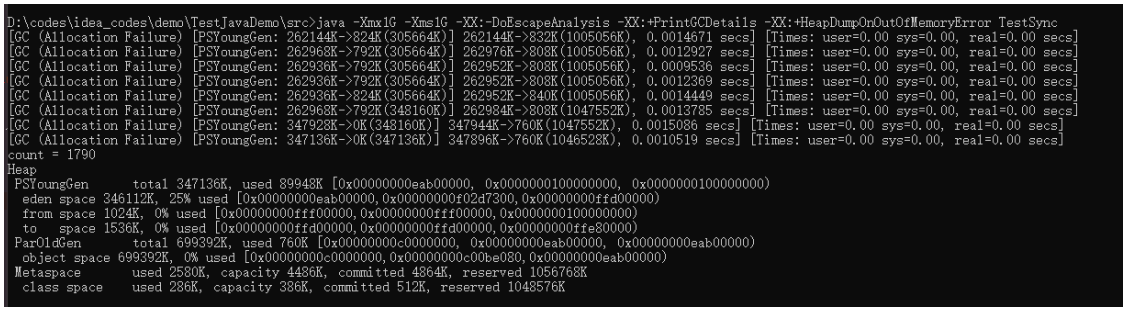
关闭逃逸分析

执行命令

```
$ java -Xmx1G -Xms1G -XX:-DoEscapeAnalysis -XX:+PrintGCDetails -
XX:+HeapDumpOnOutOfMemoryError TestSync
```

我们将 Java heap 大小调整到 1G 并同时打印出 `gc` 信息

结果:



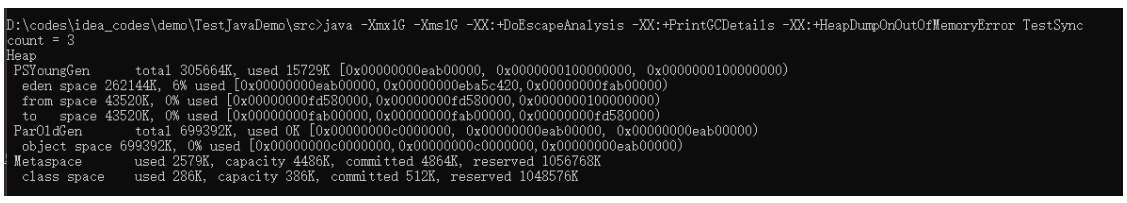
可以看到 执行上述代码 耗时 1790 毫秒 并伴随着大量的 `gc` 信息

开启逃逸分析

执行命令

```
$ java -Xmx1G -Xms1G -XX:+DoEscapeAnalysis -XX:+PrintGCDetails -
XX:+HeapDumpOnOutOfMemoryError TestSync
```

结果:



可以看到 执行上述代码 耗时 3 毫秒 并且没有发现 `gc` 信息

> 除了通过 将 Java heap 内存调小 >根据GC的次数来分析，也能发现，开启了逃逸分析之后，在运行期间，GC次数会明显减少。 >正是因为很多堆上分配被优化成了栈上分配，所以GC次数有了明显的减少。 还可以 通过 Java 命令 jmap 查看对象个数来验证对象在栈上分配内存。

两种编译器的区别

- 使用client编译器时，默认执行为1500次才认为是热代码
- 使用server编译器时，默认执行为10000次才认为是热代码

上面的例子开启逃逸分析后，并不是所有的对象都直接在栈上分配，而是通过JIT分析此代码是热代码，才进行异步编译成本地机器码，并通过逃逸分析，把对象分配到栈上。（如果是server编译器：在前10000次循环和编译成本地机器码这段时间，对象都会在堆中分配对象，编译成本地机器码后才会在栈上分配）

结论

所以，如果以后再有人问你：是不是所有的对象和数组都会在堆内存分配空间?

那么你可以告诉他：不一定，随着JIT编译器的发展，在编译期间，如果JIT经过逃逸分析，发现有些对象没有逃逸出方法，那么有可能堆内存分配会被优化成栈内存分配。但是这也并不是绝对的。就像我们前面看到的一样，在开启逃逸分析之后，也并不是所有User对象都没有在堆上分配。

标量替换

标量和聚合量

标量即不可被进一步分解的量，而JAVA的基本数据类型就是标量（如：int，long等基本数据类型以及reference类型等），标量的对立就是可以被进一步分解的量，而这种量称之为聚合量。而在JAVA中对象就是可以被进一步分解的聚合量。

替换过程

通过逃逸分析确定该对象不会被外部访问，并且对象可以被进一步分解时，JVM不会创建该对象，而会将该对象成员变量分解若干个被这个方法使用的成员变量所代替。这些代替的成员变量在栈帧或寄存器上分配空间。

- `-XX:+EliminateAllocations` 开启标量替换。
- `-XX:-EliminateAllocations` 关闭标量替换。
- `-XX:+PrintEliminateAllocations` 查看标量替换情况 Server VM 支持

总结

实际问题

在面向对象的编程语言中，动态编译器特别适合使用逃逸分析。在传统的静态编译中，方法重写使逃逸分析变得不可能，任何调用方法可能被一个允许指针逃逸的版本重写。动态编译器可以使用重载信息来执行逃逸分析，并且当相关方法被动态代码加载重写时，会重新执行分析。[1]

Java编程语言的流行使得逃逸分析成为一个研究热点。Java的堆分配、内置线程和Sun HotSpot动态编译器的结合创建了一个关于逃逸分析优化的候选平台。逃逸分析最早是在Java标准版6中实现的。

现状

关于逃逸分析的论文在1999年就已经发表了，但直到JDK 1.6才有实现，而且这项技术到如今也并不是十分成熟的。

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量替换、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，这其实也是一个相对耗时的过程。

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程就白白浪费掉了。

虽然这项技术并不十分成熟，但是他也是即时编译器优化技术中一个十分重要的手段。

遇到的问题

在 Mac OS 上如果安装的 JDK 版本 是 1.8 执行开启逃逸分析 命令 会报如下错误:

```
sun.jvm.hotspot.debugger.DebuggerException: Can't attach symbolicator to

the process
sun.jvm.hotspot.debugger.DebuggerException:
sun.jvm.hotspot.debugger.DebuggerException: Can't attach symbolicator to
the process
    at
sun.jvm.hotspot.debugger.bsd.BsdDebuggerLocal$BsdDebuggerLocalWorkerThread
.execute(BsdDebuggerLocal.java:169)
    at
sun.jvm.hotspot.debugger.bsd.BsdDebuggerLocal.attach(BsdDebuggerLocal.java
:287)
    at sun.jvm.hotspot.HotSpotAgent.attachDebugger(HotSpotAgent.java:671)
    at
sun.jvm.hotspot.HotSpotAgent.setupDebuggerDarwin(HotSpotAgent.java:659)
    at sun.jvm.hotspot.HotSpotAgent.setupDebugger(HotSpotAgent.java:341)
    at sun.jvm.hotspot.HotSpotAgent.go(HotSpotAgent.java:304)
    at sun.jvm.hotspot.HotSpotAgent.attach(HotSpotAgent.java:140)
    at sun.jvm.hotspot.tools.Tool.start(Tool.java:185)
    at sun.jvm.hotspot.tools.Tool.execute(Tool.java:118)
    at sun.jvm.hotspot.tools.JInfo.main(JInfo.java:138)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:
62)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorIm
pl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at sun.tools.jinfo.JInfo.runTool(JInfo.java:108)
    at sun.tools.jinfo.JInfo.main(JInfo.java:76)
Caused by: sun.jvm.hotspot.debugger.DebuggerException: Can't attach
symbolicator to the process
    at sun.jvm.hotspot.debugger.bsd.BsdDebuggerLocal.attach0(Native
Method)
    at
sun.jvm.hotspot.debugger.bsd.BsdDebuggerLocal.access$100(BsdDebuggerLocal.
java:65)
    at
sun.jvm.hotspot.debugger.bsd.BsdDebuggerLocal$1AttachTask.doit(BsdDebugger
Local.java:278)
    at
sun.jvm.hotspot.debugger.bsd.BsdDebuggerLocal$BsdDebuggerLocalWorkerThread
.run(BsdDebuggerLocal.java:144)
```

这是 JDK 8 的一个bug 在官网 上有说明
https://bugs.openjdk.java.net/browse/JDK-8160376

解决方案： 升级 JDK 到 9 以上

参考

https://zh.wikipedia.org/wiki/逃逸分析

书籍

https://book.douban.com/subject/24722612/

深入理解Java中的逃逸分析

https://www.hollischuang.com/archives/2583

https://www.jianshu.com/p/04fcd0ea5af7

Java 命令

https://www.cnblogs.com/lleid/archive/2013/03/21/java.html

https://blog.csdn.net/fenglibing/article/details/6411953

JVM调优命令

https://www.cnblogs.com/ityouknow/p/5714703.html

Java获取 Pid

https://blog.csdn.net/testcs_dn/article/details/58240484

https://blog.csdn.net/yin_jw/article/details/32198177

关于Mac OS 上 JDK 8 执行命令问题

https://bugs.java.com/bugdatabase/view_bug.do?bug_id=7112802

https://bugs.openjdk.java.net/browse/JDK-8160376

https://stackoverflow.com/questions/53313510/run-jmap-command-failed-in-macos

https://stackoverflow.com/questions/57739532/error-attach-task-for-pid23990-failed-os-kern-failure