链滴

关于 Ahead-of-Time Compilation 的调研与研究

作者：Lord-X

原文链接：https://ld246.com/article/1564725481130

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20181204.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
:rose::rose: 如果您觉得我的文章对您有帮助的话，记得在 GitHub 上 star 一波哈 :rose::rose:</
>
:rose::rose:<a href="https://ld246.com/forward?goto=https%3A%2F%2Fgithub.com%2FL
rd-X%2Fawesome-it-blog" target="_blank" rel="nofollow ugc">GitHub_awesome-it-blog</a
 :rose::rose:
<hr>
<h3 id="0-介绍">0 介绍</h3>
Ahead-of-Time Compilation，简称 AOT 编译，是在 Java9 中提供的一个功能，它能够事先
应用中或 JDK 中的字节码编译成机器码（提前做了即时编译器的事儿，但与 C1、C2 编译有很大差
），然后在启动应用时，使用这些编译好的机器码来加快应用启动速度，可以降低应用启动初期由即
编译器导致的 CPU 使用率飙高。
<h3 id="1-AOT-Quick-Start">1 AOT Quick Start</h3>
<h4 id="1-1-生成AOT-Library">1.1 生成 AOT Library</h4>
AOT 通过 jaotc 工具编译（在 $JAVA_HOME/bin 下），可以.class、java module、jar 等为
位进行编译，编译结果为一个.so 文件。jaotc 是通过 Graal 编译器生成机器码的。
例如，通过下面方式将一个 class 文件编译成 AOTLibrary：
<pre><code class="language-text highlight-chroma"><span cl
ss="highlight-cl">jaotc --output libHelloWorld.so HelloWorld.class
</code></pre>
通过下面的方式来编译 java.base 模块：
<pre><code class="language-text highlight-chroma"><span cl
ss="highlight-cl">jaotc --output libjava.base.so --module java.base
</code></pre>
<h4 id="1-2-使用AOT-Library">1.2 使用 AOT Library</h4>
生成的.so 文件使用起来非常方便，可通过以下两种方式使用：

将.so 文件放到 $JAVA_HOME/lib 下，JVM 启动时会自动加载
添加参数-XX:AOTLibrary，指定使用哪个 AOTLibrary

第二种方式，可通过如下方式使用：
<pre><code class="language-text highlight-chroma"><span cl
ss="highlight-cl">java -XX:AOTLibrary=./libHelloWorld.so,./libjava.base.so HelloWorld
</code></pre>
<h3 id="2-JVM对AOTLibrary的管理">2 JVM 对 AOTLibrary 的管理</h3>
JVM 将编译好的 AOTLibrary 当做 CodeCache 的一个扩展。当一个 Java class 被加载时，JVM
会先在 AOTLibrary 中看看有没有编译好的、与之匹配的 method，如果有就直接使用了。
但我们的代码可能是经常变化的，JVM 就需要识别这些变化的代码，并不再从 AOTLibrary 中
载这些类。这个功能是通过 class 指纹实现的。在 jaotc 编译 AOTLibrary 时，会同时生成每个 class
的指纹，并存储在编译而成的 AOTLibrary 中。
然后每次尝试从 AOTLibrary 中加载时，会比较 class 的指纹，从而实现这个功能。
此外，生成 AOTLibrary 时，jaotc 编译使用的 JDK 版本和用于 Java 应用启动的 JVM 版本必须
一致的。jaotc 后，JDK 版本会记录在 AOTLibrary 中，在 AOT 被 JVM 加载时会做检查，如果版本
同，会拒绝加载。
jaotc 和启动 JVM 用的运行时参数也必须是一致的，在 jaotc 中通过-J 来指定 Java 运行时需要
参数，例如：
<pre><code class="language-text highlight-chroma"><span cl
ss="highlight-cl">jaotc -J-XX:+UseParallelGC -J-XX:-UseCompressedOops --output libHello
orld.so HelloWorld.class
java -XX:+UsePara
lelGC -XX:-UseCompressedOops -XX:AOTLibrary=./libHelloWorld.so HelloWorld
</code></pre>
这些启动参数会被记录在生成的 AOT 文件中，JVM 加载 AOT 文件时会校验参数是否一致，如

不一致则不会加载这个 AOT 库，此时，如果-XX:+UseAOTStrictLoading 参数开启了，JVM 进程会接退出。
最后提一下，jaotc 并没有解决 class 依赖的问题，这些被依赖的类必须被添加到 classpath 中否则在编译 AOT 的过程中会抛出一个 ClassNotFoundException 异常。
<h3 id="3-AOT的编译模式">3 AOT 的编译模式</h3>
AOT 有两种编译模式，通过--compile-for-tiered 标记控制。添加这个标记的话，意味着使用层编译方式来生成 AOTLibrary，否则使用普通方式编译。

在没有开启的情况下，代码会被静态编译成机器码，这个编译过程是没有 profile 收集的，并且不会被 JIT 重新编译。
开启的情况下，会收集程序运行的 profiling 信息。其效果等同于 C1 编译器在 Tier2 层的 profiling 编译效果，如果代码在运行的过程中达到 C1 的 Tier3 层编译的阈值，会触发 C1 的重新编译，用收集所有的 profiling 信息，这些信息将用于 C2 编译器的编译优化。

可以看出，AOT 编译即使在开启分层编译模式的情况下，也只是能替代部分 C1 的编译工作，他法顶替 C2，最终还是要经过 C2 重新编译的。
所以，对于因为 C2 导致应用启动 CPU 飙高的应用来说，使用 AOT 的方式并不会提升应用启动性能。
<h3 id="4-案例-生成并使用java-base模块的AOTLibrary">4 案例：生成并使用 java.base 模块的 OTLibrary</h3>
因为 java.base 中的 method 量过大（大概 50000+ 个），所以在用 jaotc 生成时，要给足够内存，通过下面方式生成：
<pre><code class="language-text highlight-chroma">jaotc -J-XX:+UseCompressedOops -J-XX:+UseG1GC -J-Xmx4g --compile-fo
-tiered --info --compile-commands java.base-list.txt --output libjava.base-coop.so --module java.base
</code></pre>
--compile-commands 标记指定的文件可用于指定只编译哪些、或排除编译哪些方法。由于 java base 模块中的一些方法会导致编译失败，所以通过 java.base-list.txt 文件将之排除，这个文件内容下：
<pre><code class="language-text highlight-chroma">cat java.base-list.txt

# jaotc: java.lang.StackOverflowError
exclude sun.util.resources.LocaleNames.getContents()[[Ljava/lang/Object;
exclude sun.util.resources.TimeZoneNames.getContents()[[Ljava/lang/Object;
exclude sun.util.resources.cldr.LocaleNames.getContents()[[Ljava/lang/Object;
exclude sun.util.resources..*.LocaleNames_.*.getContents\(\)\[\[Ljava/lang/Object;
exclude sun.util.resources..*.LocaleNames_.*_.*.getContents\(\)\[\[Ljava/lang/Object;
exclude sun.util.resources..*.TimeZoneNames_.*.getContents\(\)\[\[Ljava/lang/Object;
exclude sun.util.resources..*.TimeZoneNames_.*_.*.getContents\(\)\[\[Ljava/lang/Object;
java.lang.Error: Trampoline must not be defined by the bootstrap classloader
exclude sun.reflect.misc.Trampoline.<clinit>()V
exclude sun.reflect

misc.Trampoline.invoke(Ljava/lang/reflect/Method;Ljava/lang/Object;[Ljava/lang/Object;)Ljava
lang/Object;
JVM asserts
exclude com.sun.c
ypto.provider.AESWrapCipher.engineUnwrap([BLjava/lang/String;I)Ljava/security/Key;
exclude sun.securi
y.ssl.*
exclude sun.net.R
gisteredDomain.<clinit>()V
Huge methods
exclude jdk.intern
l.module.SystemModules.descriptors()[Ljava/lang/module/ModuleDescriptor;
</code></pre>
生成 AOTLibrary 之后，通过下面方式加载并使用 AOTLibrary：
<pre><code class="language-text highlight-chroma"><span cl
ss="highlight-cl">java -XX:AOTLibrary=./libjava.base-coop.so,./libHelloWorld.so HelloWorld
</code></pre>
<blockquote>
PS1：由于 Java9 之后会默认使用 G1 收集器，并且默认开启 UseCompressedOops，所以启
时可以不用指定。
PS2：除了在启动时通过参数-XX:AOTLibrary 指定加载的 AOTLibrary 之外，也可以将生成的.so
文件放到 $JAVA_HOME/lib 下，JVM 启动时会自动扫描加载。
</blockquote>
<h3 id="5-Java中AOT相关的一些options">5 Java 中 AOT 相关的一些 options</h3>

-XX:+/-UseAOT

使用 AOT 编译的文件，默认开启。

-XX:AOTLibrary=<file>

指定 AOT 文件，多个文件之间用英文半角逗号分隔(,)。

-XX:+/-PrintAOT

在标准输出日志打印使用到的 AOTLibrary 中的 class 和 method。
<blockquote>
下面是一些具备诊断功能的 flag，使用时需要优先开启-XX:+UnlockDiagnosticVMOptions</p

</blockquote>

-XX:+/-UseAOTStrictLoading

启动时，如果没有任何一个 AOTLibrary 与当前的 JVM 环境匹配上，则 JVM 进程直接退出。</
>
这个可用于判定是否成功加载并使用了 AOTLibrary 编译的 class 或 method。
<blockquote>
此外，下面的一些参数可通过日志的形式打印 AOT 的使用情况，需要配合-Xlog 参数
</blockquote>

aotclassfingerprint

当 class 指纹不匹配时，打印日志。

aotclassload

当可用的 class 在 AOTLibrary 中找到时，打印日志。

aotclassresolve

当解析 AOT 中的 class 成功或失败时，打印日志。
<h3 id="6-jaotc的usage">6 jaotc 的 usage</h3>
使用格式如下：
<pre><code class="language-text highlight-chroma"><span cl
ss="highlight-cl">jaotc <options> <name or list>
</code></pre>
name 表示的是 class name 或 jar 文件，list 可以是 class、modules、jar 或包含 class 文件的
录。
另外，options 可以有如下选项：

--output <file>

输出的文件名，默认名称是"unnamed.so"。

--class-name <class names>

待编译的 java classes 列表。

--jar <jar files>

待编译的 jar files 列表。

--module <modules>

待编译的 java modules 列表。

--directory <dirs>

指定搜索目录，会从指定目录搜索待编译的文件。

--search-path <dirs>

搜索指定目录下的文件。

--compile-commands <file>

指定包含编译指令的文件，加载 AOT 时会执行这些指令进行文件过滤（exclude or compileOnl
），用法如下：
<pre><code class="language-text highlight-chroma"><span cl
ss="highlight-cl">exclude sun.util.resources..*.TimeZoneNames_.*.getContents\(\)\[\[Ljava/la
g/Object;
exclude sun.securi
y.ssl.*
compileOnly java.
ang.String.*
</code></pre>
exclude 用于排除指定的方法，被排除掉的不会被编译。
compileOnly 用于指定只编译哪些方法。

--compile-for-tiered

为分层编译生成带有 profiling 的代码，默认不生成。

--compile-with-assertions

用 Java 断言生成代码，默认不开启。

--compile-threads <number>

用于编译的线程数量，默认值：min(16, available_cpus)

--ignore-errors

忽略所有 class 加载时抛出的异常，默认情况下，如果编译过程中抛出异常，会直接退出编译。<
p>

--exit-on-error

编译发生 error 时，退出编译。默认情况下，会跳过编译失败的 method，其他的 method 编
仍会继续。

--info

打印编译时的信息。

--verbose

打印更全的编译时信息，在--info 开启的情况下有效。

--debug

打印所有细节编译时的信息，在--info 和--verbose 同时开启的情况下有效。

--help

打印 jaotc 的 usage 信息。

--version

打印版本信息。

-J<flag>

指定 JVM 运行时参数。
<h3 id="7-使用AOT的一些限制">7 使用 AOT 的一些限制</h3>

AOT 最初在 JDK9 作为一个实验性的功能出现，并且只能运行在 Linux x64 系统、64 位 JVM
境、Parallel 或 G1 环境下。
AOT 编译时和使用时必须使用相同的 JVM 参数。
AOT 的编译和使用必须在同一个系统环境下。
无法编译动态生成的 java code，例如 lambda 表达式。
AOT 不支持使用用户自定义类加载器加载的 class，因为在编译阶段无法知晓最终运行是会使用

个类加载器加载这个 class。

<h3 id="8-实验">8 实验</h3>
<hr>
<pre><code class="language-bash highlight-chroma"><span c
ass="highlight-cl"># 使用AOT编译java.base模块
/Library/Java/Java
irtualMachines/jdk-11.0.3.jdk/Contents/Home/bin/jaotc -J-XX:+UseCompressedOops -J-XX:+
seG1GC -J-Xmx4g --compile-for-tiered --info --compile-commands java.base-list.txt --output
ibjava.base-coop.so --module java.base
</code></pre>
<hr>
<h4 id="8-1-第一次">8.1 第一次</h4>

JDK11 不带任何参数正常启动
启动接口性能：

<img src="https://ld246.com/images/img-loading.svg" alt="JDK11 启动性能.png" data-sr
="https://b3logfile.com/file/2019/08/JDK11%E5%90%AF%E5%8A%A8%E6%80%A7%E8%83%
D-d0159fa3.png?imageView2/2/interlace/1/format/jpg">
<h4 id="8-2-第二次">8.2 第二次</h4>

使用 jaotc 编译 AOTLibrary（带分层编译），只编译 java.base：

<pre><code class="language-text highlight-chroma"><span cl
ss="highlight-cl">jaotc --output libjava.base.tiered.so --compile-for-tiered --module java.bas

</code></pre>

使用 AOT 启动

<pre><code class="language-text highlight-chroma"><span cl
ss="highlight-cl">-XX:AOTLibrary=/data/coohua/logs/libjava.base.tiered.so
</code></pre>

启动性能

<img src="https://ld246.com/images/img-loading.svg" alt="分层编译 AOT 启动.png" data
src="https://b3logfile.com/file/2019/08/%E5%88%86%E5%B1%82%E7%BC%96%E8%AF%91
OT%E5%90%AF%E5%8A%A8-bb21b9ef.png?imageView2/2/interlace/1/format/jpg">

CPU

<img src="https://ld246.com/images/img-loading.svg" alt="分层编译 AOT 启动 CPU 消耗.
ng" data-src="https://b3logfile.com/file/2019/08/%E5%88%86%E5%B1%82%E7%BC%96%E8
AF%91AOT%E5%90%AF%E5%8A%A8CPU%E6%B6%88%E8%80%97-71f105a3.png?imageVie
2/2/interlace/1/format/jpg">
主要在 C2 编译器上。

与没有 AOT 的性能对比

<img src="https://ld246.com/images/img-loading.svg" alt="分层编译 AOT 接口性能对比.p
g" data-src="https://b3logfile.com/file/2019/08/%E5%88%86%E5%B1%82%E7%BC%96%E8
AF%91AOT%E6%8E%A5%E5%8F%A3%E6%80%A7%E8%83%BD%E5%AF%B9%E6%AF%94-2c

548df.png?imageView2/2/interlace/1/format/jpg">
看上去好了一些。
<h4 id="8-3-生成JFR-通过JMC进一步分析">8.3 生成 JFR，通过 JMC 进一步分析</h4>
通过下面的方式生成 JFR 信息，这条指令的意思是：在应用启动 20 秒后开始收集 JFR，收集 120 秒。
<pre><code class="language-text highlight-chroma"><span cl
ss="highlight-cl">-XX:StartFlightRecording=delay=20s,duration=120s,name=myrecording,fil
name=./record.jfr,settings=profile
</code></pre>
<blockquote>
PS: 关于 JFR 和 JMC，可参考：<a href="https://ld246.com/forward?goto=https%3A%2F%
Fgithub.com%2FLord-X%2Fawesome-it-blog%2Fblob%2Fmaster%2Fjava%2FJava%2520Fligh
%2520Recorder%25E5%2588%259D%25E6%258E%25A2.md" target="_blank" rel="nofollow
gc">Java Flight Recorder 初探
</blockquote>
分下面三种情况对比，主要对比 C2 的编译情况：

没有 AOT 的 JFR（收集 1 分钟）
没有分层编译 AOT 的 JFR（收集 2 分钟）
分层编译 AOT 的 JFR（收集 2 分钟）

<h5 id="8-3-1-没有AOT的JFR">8.3.1 没有 AOT 的 JFR</h5>

C2 线程 CPU 时间片的消耗

右侧黄色部分表示 C2 的 CPU 消耗。
<img src="https://ld246.com/images/img-loading.svg" alt="没有 AOT 的 JFRC2 线程 CPU
时间片的消耗.png" data-src="https://b3logfile.com/file/2019/08/%E6%B2%A1%E6%9C%89AO
%E7%9A%84JFRC2%E7%BA%BF%E7%A8%8BCPU%E6%97%B6%E9%97%B4%E7%89%87%E7
9A%84%E6%B6%88%E8%80%97-0e63cbff.png?imageView2/2/interlace/1/format/jpg">

方法编译耗时

<img src="https://ld246.com/images/img-loading.svg" alt="没有 AOT 的 JFR 方法编译耗时
png" data-src="https://b3logfile.com/file/2019/08/%E6%B2%A1%E6%9C%89AOT%E7%9A%8
JFR%E6%96%B9%E6%B3%95%E7%BC%96%E8%AF%91%E8%80%97%E6%97%B6-d5a8d2c4.p
g?imageView2/2/interlace/1/format/jpg">
<h5 id="8-3-2-没有分层编译AOT的JFR">8.3.2 没有分层编译 AOT 的 JFR</h5>

C2 线程 CPU 时间片的消耗

<img src="https://ld246.com/images/img-loading.svg" alt="没有分层编译 AOT 的 JFRC2
程 CPU 时间片消耗.png" data-src="https://b3logfile.com/file/2019/08/%E6%B2%A1%E6%9C%
9%E5%88%86%E5%B1%82%E7%BC%96%E8%AF%91AOT%E7%9A%84JFRC2%E7%BA%BF%E
%A8%8BCPU%E6%97%B6%E9%97%B4%E7%89%87%E6%B6%88%E8%80%97-ebd1cfd8.png?
mageView2/2/interlace/1/format/jpg">

方法编译耗时

<img src="https://ld246.com/images/img-loading.svg" alt="没有分层编译 AOT 的 JFR 方
编译耗时.png" data-src="https://b3logfile.com/file/2019/08/%E6%B2%A1%E6%9C%89%E5%8
%86%E5%B1%82%E7%BC%96%E8%AF%91AOT%E7%9A%84JFR%E6%96%B9%E6%B3%95%E
%BC%96%E8%AF%91%E8%80%97%E6%97%B6-4b70aa5e.png?imageView2/2/interlace/1/fo
mat/jpg">

由于没有 AOT 的 JFR 收集了 1 分钟，所以这里的时间跨度是上面的两倍。等比缩小后，可以发两次在 CPU 时间片消耗上差不多，但这一次收集时间更长，出现了多个编译时间较长的方法。
<h5 id="8-3-3-分层编译AOT的JFR">8.3.3 分层编译 AOT 的 JFR</h5>

C2 线程 CPU 时间片的消耗

方法编译耗时

可看出跟没有分层编译是，没有明显差别。这也验证了 AOT 并不能优化 C2 的编译时间。
<h4 id="8-4-结论">8.4 结论</h4>
由于我们应用启动时消耗 CPU 时间片最多的是 C2 编译器，对于 AOT 来说，正如上文描述的那，他并不能降低 C2 的消耗，所以 AOT 无法解决我们这类场景的问题。
思考：
其实可以根据 AOT 编译的原理想一下他适用的场景。AOT 是一个静态编译过程，他是离线的，运行时编译，所以无法收集足够有效的 profiling 信息，所以他编译的结果肯定无法与 C2 相媲美。<b
>
同时官文中也说了，他可以大概等同于 C1 的 Tier2 编译的效果，在这一层上，会收集较少的 profilin 信息，然后 AOT 就直接将字节码编译成机器码。根据这个特点，他应该可以用于短时运行应用的启性能优化。对于长期跑到服务器，他应该是无能为力的。
<h3 id="9-参考">9 参考</h3>

JEP 295: Ahead-of-Time Compilation
解密新一代 Java JIT 编译器 Graa
