JVM_06 执行引擎（Execution Engine）

作者：wlgzs-sjl
原文链接：https://ld246.com/article/1603282966950
来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20190528.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<h2 id="执行引擎概述">执行引擎概述</h2>

执行引擎是 Java 虚拟机的核心组成部分之一
虚拟机是一个相对于“物理机”的概念，这两种机器都有代码执行能力，其区别是物理机的执行
擎是直接建立在处理器、缓存、指令集和操作系统层面上的，而虚拟机的执行引擎则是由软件自行实
的，因此可以不受物理条件制约地定制指令集与执行引擎的结构体系，能够执行那些不被硬件直接支
的指令集格式。
JVM 的主要任务是负责装载字节码到其内部，但字节码并不能够直接运行
操作系统之上，因为字节码指令并非等价于本地机器指令，它内部包含的仅仅只是一些能够被 JVM
识别的字节码指令、符号表和其他辅助信息
那么，如果想让一个 Java 程序运行起来、执行引擎的任务就是==将字节码指令解释/编译为对
平台上的本地机器指令才可以。简单来说，JVM 中的执行引擎充当了将高级语言翻译为机器语言的译
.
执行引擎的工作过程

从外观上来看，所有的 Java 虚拟机的执行引擎输入、输出都是一致的：输入的是字节码二进制
，处理过程是字节码解析执行的等效过程，输出的是执行结果。

<img src="https://ld246.com/images/img-loading.svg" alt="1729333fda0586ed.png" dat
-src="https://b3logfile.com/file/2020/10/1729333fda0586ed-2a12ea10.png?imageView2/2/in
erlace/1/format/jpg">

执行引擎在执行的过程中究竟需要执行什么样的字节码指令完全依赖于 PC 寄存器。
每当执行完一项指令操作后，PC 寄存器就会更新下一条需要被执行的指令地址。
当然方法在执行的过程中，执行引擎有可能会通过存储在局部变量表中的对象引用准确定位到存
在 Java 堆区中的对象实例信息，以及通过对象头中的元数据指针定位到目标对象的类型信息。

<h2 id="Java代码编译和执行过程">Java 代码编译和执行过程</h2>
大部分的程序代码转换成物理机的目标代码或虚拟机能执行的指令集之前，都需要经过下面图中
各个步骤：
<img src="https://ld246.com/images/img-loading.svg" alt="17293345b33057a0.png" da
a-src="https://b3logfile.com/file/2020/10/17293345b33057a0-b3793534.png?imageView2/2/
nterlace/1/format/jpg">
Java 代码编译是由 Java 源码编译器来完成，流程图如下所示：
<img src="https://ld246.com/images/img-loading.svg" alt="172933480ac44fb8.png" dat
-src="https://b3logfile.com/file/2020/10/172933480ac44fb8-bd508917.png?imageView2/2/in
erlace/1/format/jpg">
Java 字节码的执行是由 JVM 执行引擎来完成，流程图如下所示：
<img src="https://ld246.com/images/img-loading.svg" alt="1729334a1a335e4c.png" dat
-src="https://b3logfile.com/file/2020/10/1729334a1a335e4c-d614b11a.png?imageView2/2/in
erlace/1/format/jpg">
<h4 id="什么是解释器--Interpreter--什么是JIT编译器-">什么是解释器（ Interpreter），什么是 JIT
编译器？</h4>
解释器：当 Java 虚拟机启动时会根据预定义的规范对字节码采用逐行解释
方式执行，将每条字节码文件中的内容“翻译”为对应平台的本地机器指令执行。
JIT （Just In Time Compiler）编译器（即时编译器）：就是虚拟机将源
码直接编译成和本地机器平台相关的机器语言。
<h4 id="为什么说Java是半编译半解释型语言-">为什么说 Java 是半编译半解释型语言？</h4>
JDK1.0 时代，将 Java 语言定位为“解释执行”还是比较准确的。再后来，Java 也发展出可以
接生成本地代码的编译器。

现在 JVM 在执行 Java 代码的时候，通常都会将解释执行与编译执行二者结合起来进行。

<h2 id="机器码-指令-汇编语言">机器码、指令、汇编语言</h2>
<h3 id="机器码">机器码</h3>

各种用二进制编码方式表示的指令，叫做机器指令码。开始，人们就用它编写程序，这就是机器语言。
机器语言虽然能够被计算机理解和接受，但和人们的语言差别太大，不易被人们理解和记忆，并用它编程容易出差错。
用它编写的程序一经输入计算机，CPU 直接读取运行，因此和其他语言编的程序相比，执行速度快。
机器指令与 CPU 紧密相关，所以不同种类的 CPU 所对应的机器指令也就不同。

<h3 id="指令">指令</h3>

由于机器码是由 0 和 1 组成的二进制序列，可读性实在太差，于是人们发明了指令。
指令就是把机器码中特定的 0 和 1 序列，简化成对应的指令（一般为英文简写，如 mov，inc），可读性稍好
由于不同的硬件平台，执行同一个操作，对应的机器码可能不同，所以不同的硬件平台的同一种令（比如 mov），对应的机器码也可能不同。

<h4 id="指令集">指令集</h4>

不同的硬件平台，各自支持的指令，是有差别的。因此每个平台所支持的指令，称之为对应平台指令集。
如常见的

x86 指令集，对应的是 x86 架构的平台
ARM 指令集，对应的是 ARM 架构的平台

<h3 id="汇编语言">汇编语言</h3>

由于指令的可读性还是太差，于是人们又发明了汇编语言。
在汇编语言中，用助记符（Mnemonics）代替机器指令的操作码，用地址符号（Symbol）或标（Label）代替指令或操作数的地址。
在不同的硬件平台，汇编语言对应着不同的机器语言指令集，通过汇编过程转换成机器指令。

由于计算机只认识指令码，所以用汇编语言编写的程序还必须翻译成机器指令码，计算机才能识和执行。

<h3 id="高级语言">高级语言</h3>

为了使计算机用户编程序更容易些，后来就出现了各种高级计算机语言。高级语言比机器语言、编语言更接近人的语言
当计算机执行高级语言编写的程序时，仍然需要把程序解释和编译成机器的指令码。完成这个过的程序就叫做解释程序或编译程序。

a-src="https://b3logfile.com/file/2020/10/1729335b173e3d31-14ec0d4c.png?imageView2/2/nterlace/1/format/jpg">
<h4 id="字节码">字节码</h4>

字节码是一种中间状态（中间码）的二进制代码（文件），它比机器码更抽象，需要直译器转译才能成为机器码
字节码主要为了实现特定软件运行和软件环境、与硬件环境无关。
字节码的实现方式是通过编译器和虚拟机器。编译器将源码编译成字节码，特定平台上的虚拟机将字节码转译为可以直接执行的指令。

字节码的典型应用为 Java bytecode

<h4 id="C-C--源程序执行过程">C、C++ 源程序执行过程</h4>
编译过程又可以分成两个阶段：编译和汇编。

编译过程：是读取源程序（字符流），对 之进行词法和语法的分析，将高级语言指令转换为功能效的汇编代码
汇编过程：实际上指把汇编语言代码翻译成目标机器指令的过程。

<img src="https://ld246.com/images/img-loading.svg" alt="172933609774cda3.png" da a-src="https://b3logfile.com/file/2020/10/172933609774cda3-92aa0620.png?imageView2/2/nterlace/1/format/jpg">
<h2 id="解释器">解释器</h2>
JVM 设计者们的初衷仅仅只是单纯地为了满足 Java 程序实现跨平台特性，因此避免采用静态编译的方式直接生成本地机器指令，从而诞生了实现解释器在运行时采用逐行解字节码执行程序的想法。
<img src="https://ld246.com/images/img-loading.svg" alt="172933694cc12661.png" dat -src="https://b3logfile.com/file/2020/10/172933694cc12661-876168ba.png?imageView2/2/interlace/1/format/jpg">

解释器真正意义上所承担的角色就是一个运行时“翻译者”，将字节码文件中的内容“翻译”为应平台的本地机器指令执行。
当一条字节码指令被解释执行完成后，接着再根据 PC 寄存器中记录的下一条需要被执行的字节指令执行解释操作。

在 Java 的发展历史里，一共有两套解释执行器，即古老的字节码解释器现在普遍使用的模板解释器。

字节码解释器在执行时通过纯软件代码模拟字节码的执行，效率非常低下。而模板解释器将每一字节码和一个模板函数相关联，模板函数中直接产生这条字节码执行时的机器码，从而很大程度上提了解释器的性能。

在 HotSpot VM 中，解释器主要由 Interpreter 模块和 Code 模块构成。

Interpreter 模块：实现了解释器的核心功能
Code 模块：用于管理 HotSpot JVM 在运行时生成的本地机器指令

<h4 id="现状">现状</h4>

由于解释器在设计和实现上非常简单，因此除了 Java 语言之外，还有许多高级语言同样也是基解释器执行的，比如 Python、 Perl、Ruby 等。但是在今天，基于解释器执行已经沦落为低效的代词，并且时常被一些 C/C++ 程序员所调侃。
为了解决这个问题，JVM 平台支持一种叫作即时编译的技术。即时编译的目的是避免函数被解释行，而是将整个函数体编译成为机器码，每次函数执行时，只执行编译后的机器码即可，这种方式可使执行效率大幅度提升。
不过无论如何，基于解释器的执行模式仍然为中间语言的发展做出了不可磨灭的贡献。

<h2 id="JIT编译器">JIT 编译器</h2>
<h3 id="HotSpot-VM-为何解释器与JIT编译器共存">HotSpot VM 为何解释器与 JIT 编译器共存</3>
java 代码的执行分类：

第一种是将源代码编译成字节码文件，然后在运行时通过解释器将字节码文件转为机器码执行</l>
第二种是编译执行（直接编译成机器码）。现代虚拟机为了提高执行效率，会使用即时编译技术JIT,Just In Time）将方法编译成机器码后再执行

HotSpot VM 是目前市面上高性能虚拟机的代表作之一。它采用解释器与即时编译器存的架构。在 Java 虚拟机运行时，解释器和即时编译器能够相互协作，各自取长补短，力去选择最合适的方式来权衡编译本地代码的时间和直接解释执行代码的时间。
在今天，Java 程序的运行性能早已脱胎换骨，已经达到了可以和 C/C++ 程序一较高下的地步。/p>
<h4 id="解释器依然存在的必要性">解释器依然存在的必要性</h4>
有些开发人员会感觉到诧异，既然 HotSpot VM 中已经内置 JIT 编译器了，那么为什么还需要使用解释器来“拖累”程序的执行性能呢？比如 JRockit VM 内部就不包含解释器，字节码全部都依即时编译器编译后执行。
首先明确：
当程序启动后，解释器可以马上发挥作用，省去编译的时间，立即执行。
编译器要想发挥作用，把代码编译成本地代码，需要一定的执行时间。但编译为本地代码后，执行效高。
所以：
尽管 JRockit VM 中程序的执行性能会非常高效，但程序在启动时必然需要花费更长的时间来进编译。对于服务端应用来说，启动时间并非是关注重点，但对于那些看中启动时间的应用场景而言，许就需要采用解释器与即时编译器并存的架构来换取一个平衡点。在此模式下，当 Java 虚拟器启时，解释器可以首先发挥作用，而不必等待即时编译器全部编译完成后再执行，这样可以省去许多不要的编译时间。随着时间的推移，编译器发挥作用，把越来越多的代码编译成本地代码，获得更高的行效率。
同时，解释执行在编译器进行激进优化不成立的时候，作为编译器的“逃生门”。
<h4 id="HostSpot-JVM的执行方式">HostSpot JVM 的执行方式</h4>
当虚拟机启动的时候，解释器可以首先发挥作用，而不必等待即时编译器全部编译完成再执行，样可以省去许多不必要的编译时间。并且随着程序运行时间的推移，即时编译器逐渐发挥作用，根据点探测功能，将有价值的字节码编译为本地机器指令，以换取更高的程序执行效率。
<h5 id="案例">案例</h5>
注意解释执行与编译执行在线上环境微妙的辩证关系。机器在热机状态可以承受的负载要大于机状态。如果以热机状态时的流量进行切流，可能使处于冷机状态的服务器因无法承载流量而假死。
在生产环境发布过程中，以分批的方式进行发布，根据机器数量划分成多个批次，每个批次的器数至多占到整个集群的 1/8。曾经有这样的故障案例：某程序员在发布平台进行分批发布，在输入布总批数时，误填写成分为两批发布。如果是热机状态，在正常情况下一半的机器可以勉强承载流量但由于刚启动的 JVM 均是解释执行，还没有进行热点代码统计和 JIT 动态编译，导致机器启动之后当前 1/2 发布成功的服务器马上全部宕机，此故障说明了 JIT 的存在。一阿里团队
<img src="https://ld246.com/images/img-loading.svg" alt="1729336fe2d19d44.png" da

a-src="https://b3logfile.com/file/2020/10/1729336fe2d19d44-d321cbde.png?imageView2/2/nterlace/1/format/jpg">
<h3 id="JIT编译器-">JIT 编译器</h3>
<h4 id="概念解释">概念解释</h4>

Java 语言的“编译器” 其实是一段“不确定”的操作过程，因为它可能是指一个前端译器（其实叫“编译器的前端” 更准确一些）把.java 文件转变成.class 文件的过程；
也可能是指虚拟机的后端运行期编译器（JIT 编译器，Just In Time Compilr）把字节码转变成机器码的过程。
还可能是指使用静态提前编译器（AOT 编译器，Ahead Of Time Compile）直接把. java 文件编译成本地机器代码的过程。

<blockquote>
前端编译器： Sun 的 Javac、 Eclipse JDT 中的增量式编译器（ECJ）
JIT 编译器： HotSpot VM 的 C1、C2 编译器。
AOT 编译器： GNU Compiler for the Java （GCJ） 、Excelsior JET。
</blockquote>
<h4 id="热点代码及探测方式">热点代码及探测方式</h4>
当然是否需要启动 JIT 编译器将字节码直接编译为对应平台的本地机器指令，则需要根据代码被用执行的频率而定。关于那些需要被编译为本地代码的字节码，也被称之为“热点代码” ，JIT 编译在运行时会针对那些频繁被调用的“热点代码”做出深度优化，将其直接编译为对应平台的本地机器令，以此提升 Java 程序的执行性能。

一个被多次调用的方法，或者是一个方法体内部循环次数较多的循环体都可以被称之为“热点代”，因此都可以通过 JIT 编译器编译为本地机器指令。由于这种编译方式发生在方法的执行过程中，此也被称之为栈上替换，或简称为 OSR （On StackReplacement）编译。
一个方法究竟要被调用多少次，或者一个循环体究竟需要执行多少次循环才可以达到这个标准？然需要一个明确的阈值，JIT 编译器才会将这些“热点代码”编译为本地机器指令执行。这里主要依靠strong>热点探测功能。
目前 HotSpot VM 所采用的热点探测方式是基于计数器的热点探测。
采用基于计数器的热点探测，HotSpot VM 将会为每一个方法都建立 2 个不同类型的计数器，别为方法调用计数器（Invocation Counter） 和回边计数器（BackEdge Counter） 。

方法调用计数器用于统计方法的调用次数
回边计数器则用于统计循环体执行的循环次数

<h5 id="方法调用计数器">方法调用计数器</h5>

这个计数器就用于统计方法被调用的次数，它的默认阈值在 Client 模式
下是 1500 次，在 Server 模式下是 10000 次。超过这个阈值，就会触发 JIT 编译。
这个阈值可以通过虚拟机参数一 XX： CompileThreshold 来人为设定。
当一个方法被调用时， 会先检查该方法是否存在被 JIT 编译过的版本，如果存在，则优先使用编后的本地代码来执行。如果不存在已被编译过的版本，则将此方法的调用计数器值加 1，然后判断方调用计数器与回边计数器值之和是否超过方法调用计数器的阈值。如果已超过阈值，那么将会向即时译器提交一个该方法的代码编译请求。

<img src="https://ld246.com/images/img-loading.svg" alt="172933785afec215.png" dat-src="https://b3logfile.com/file/2020/10/172933785afec215-bc7f934a.png?imageView2/2/interlace/1/format/jpg">
热度衰减

如果不做任何设置，方法调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的执行

率，即一段时间之内方法被调用的次数。当超过一定的时间限度， 如果方法的调用次数仍然不足以让提交给即时编译器编译，那这个方法的调用计数器就会被减少一半，这个过程称为方法调用计数器热的衰减（Counter Decay） ，而这段时间就称为此方法统计的半衰周期（Counter Half Life Time）
进行热度衰减的动作是在虚拟机进行垃圾收集时顺便进行的，可以使用虚拟机参数
-XX：-UseCounterDecay 来关闭热度衰减，让方法计数器统计方法调用的绝对次数，这样，只要系运行时间足够长，绝大部分方法都会被编译成本地代码。
另外， 可以使用-XX： CounterHalfLifeTime 参数设置半衰周期的时间，单位是秒。

<h5 id="回边计数器">回边计数器</h5>
它的作用是统计一个方法中循环体代码执行的次数，在字节码中遇到控制流向后跳转的指令称为回边” （Back Edge）。显然，建立回边计数器统计的目的就是为了触发 OSR 编译。

<h4 id="HotSpot-VM-可以设置程序执行方式">HotSpot VM 可以设置程序执行方式</h4>
缺省情况下 HotSpot VM 是采用解释器与即时编译器并存的架构，当然开发人员可以根据具体应用场景，通过命令显式地为 Java 虚拟机指定在运行时到底是完全采用解释器执行，还是完全采用时编译器执行。如下所示：

-Xint： 完全采用解释器模式执行程序；
-Xcomp： 完全采用即时编译器模式执行程序。如果即时编译出现问题，解释器会介入执行。
-Xmixed：采用解释器 + 即时编译器的混合模式共同执行程序。

<h5 id="测试解释器模式和JIT编译模式">测试解释器模式和 JIT 编译模式</h5>
测试表明：

纯解释器模式速度最慢（JVM1.0 版本用的就是纯解释器执行）
混合模式速度更快


```
/**
 * 测试解释器模式 JIT编译模式
 *  -Xint  : 6520ms
 *  -Xcomp : 950ms
 *  -Xmixed : 936ms
 */
public class IntCompTest {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        testPrimeNumber(1000000);
        long end = System.currentTimeMillis();
        System.out.println("花费的时间为：" + (end - start));
    }
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d testPrimeNumber(int count){
</span></span><span class="highlight-line"><span class="highlight-cl">        for (int i = 0; i
&amp;lt; count; i++) {
</span></span><span class="highlight-line"><span class="highlight-cl">            //计算100
内的质数
</span></span><span class="highlight-line"><span class="highlight-cl">            label:for(in
j = 2;j &amp;lt;= 100;j++){
</span></span><span class="highlight-line"><span class="highlight-cl">                for(int k
= 2;k &amp;lt;= Math.sqrt(j);k++){
</span></span><span class="highlight-line"><span class="highlight-cl">                    if(j %
 == 0){
</span></span><span class="highlight-line"><span class="highlight-cl">                        con
inue label;
</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">                //System
out.println(j);
</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">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h4 id="HotSpot-VM-中的JIT分类">HotSpot VM 中的 JIT 分类</h4>
<p>在 HotSpot VM 中内嵌有两个 JIT 编译器，分别为 Client Compiler 和 Server Compiler，但多数情况下我们简称为 C1 编译器和 C2 编译器。开发人员可以通过如下命令显式指定 Java 虚拟机在行时到底使用哪一种即时编译器，如下所示：</p>
<ul>
<li>-client： 指定 Java 虚拟机运行在 Client 模式下，并使用 C1 编译器；
<ul>
<li>C1 编译器会对字节码进行<strong>简单和可靠的优化，耗时短</strong>。以达到更快的编译度。</li>
</ul>
</li>
<li>-server： 指定 Java 虚拟机运行在 Server 模式下，并使用 C2 编译器。
<ul>
<li>C2 进行<strong>耗时较长的优化，以及激进优化</strong>。但优化的代码执行效率更高。</l
>
</ul>
</li>
</ul>
<h5 id="C1和C2编译器不同的优化策略">C1 和 C2 编译器不同的优化策略</h5>
<ul>
<li>在不同的编译器上有不同的优化策略，C1 编译器上主要有方法内联，去虚拟化、冗余消除。
<ul>
<li>方法内联：将引用的函数代码编译到引用点处，这样可以减少栈帧的生成，减少参数传递以及跳过程</li>
<li>去虚拟化：对唯一的实现类进行内联</li>
<li>冗余消除：在运行期间把一些不会执行的代码折叠掉</li>
</ul>
</li>
<li>C2 的优化主要是在全局层面，逃逸分析是优化的基础。基于逃逸分析在 C2 上有如下几种优化
```

（server 模式下才会有这些优化，64 位系统默认就是 server 模式）

标量替换：用标量值代替聚合对象的属性值
栈上分配：对于未逃逸的对象分配对象在栈而不是堆
同步消除：清除同步操作，通常指 synchronized

分层编译（Tiered Compilation）策略：程序解释执行（不开启性能监控）可以触发 C1 编译将字节码编译成机器码，可以进行简单优化，也可以加上性能监控，C2 编译会根据性能监控信息进激进优化。
不过在 Java7 版本之后，一旦开发人员在程序中显式指定命令“一 server"时，默认将会开启层编译策略，由 C1 编译器和 C2 编译器相互协作共同来执行编译任务。
<h5 id="总结">总结</h5>

一般来讲，JIT 编译出来的机器码性能比解释器高。
C2 编译器启动时长比 C1 编译器慢，系统稳定执行以后，C2 编译器执行速度远远快于 C1 编译。

<h2 id="Graal编译器与AOT编译器">Graal 编译器与 AOT 编译器</h2>
<h3 id="Graal编译器">Graal 编译器</h3>

自 JDK10 起，HotSpot 又加入一个全新的即时编译器： Graal 编译器
编译效果短短几年时间就追评了 C2 编译器。未来可期。
目前，带着“实验状态"标签，需要使用开关参数
-XX： +UnlockExperimentalVMOptions 一 XX： +UseJVMCICompiler 去激活，才可以使用。</l
>

<h3 id="AOT编译器">AOT 编译器</h3>

jdk9 引入了 AOT 编译器（静态提前编译器，Ahead Of Time Compiler）
Java 9 引入了实验性 AOT 编译工具 jaotc。它借助了 Graal 编译器，将所输入的 Java 类文件转为机器码，并存放至生成的动态共享库之中。
所谓 AOT 编译，是与即时编译相对立的一个概念。我们知道，即时编译指的是在程序的运行过中，将字节码转换为可在硬件上直接运行的机器码，并部署至托管环境中的过程。而 AOT 编译指的是，在程序运行之前，便将字节码转换为机器码的过程。
最大好处： Java 虚拟机加载已经预编译成二进制库，可以直接执行。不必等待即时编译器的预，减少 Java 应用给人带来“第一次运行慢”的不良体验。
缺点：

破坏了 java"一次编译，到处运行”，必须为每个不同硬件、oS 编译对应的发行包。
降低了 Java 链接过程的动态性，加载的代码在编译期就必须全部已知。
还需要继续优化中，最初只支持 Linux x64 java base

