# 概述

把字节码看作是程序语言的一种中间表示形式（Intermediate Representation，IR）的话，那编译器无论在何时、在何种状态下把Class文件转换成与本地基础设施（硬件指令集、操作系统）相关的二进制机器码，它都可以视为整个编译过程的后端。

在Java世界里，提前编译（Ahead Of Time，AOT）和即时编译（Just In Time，JIT）是两大热门研究对象。

无论是提前编译器抑或即时编译器，都不是Java虚拟机必需的组成部分，《Java虚拟机规范》中从来没有规定过虚拟机内部必须要包含这些编译器，更没有限定或指导这些编译器应该如何去实现。但是，后端编译器编译性能的好坏、代码优化质量的高低却是衡量一款商用虚拟机优秀与否的关键指标之一，它们也是商业Java虚拟机中的核心，是最能体现技术水平与价值的功能。

# 即时编译器

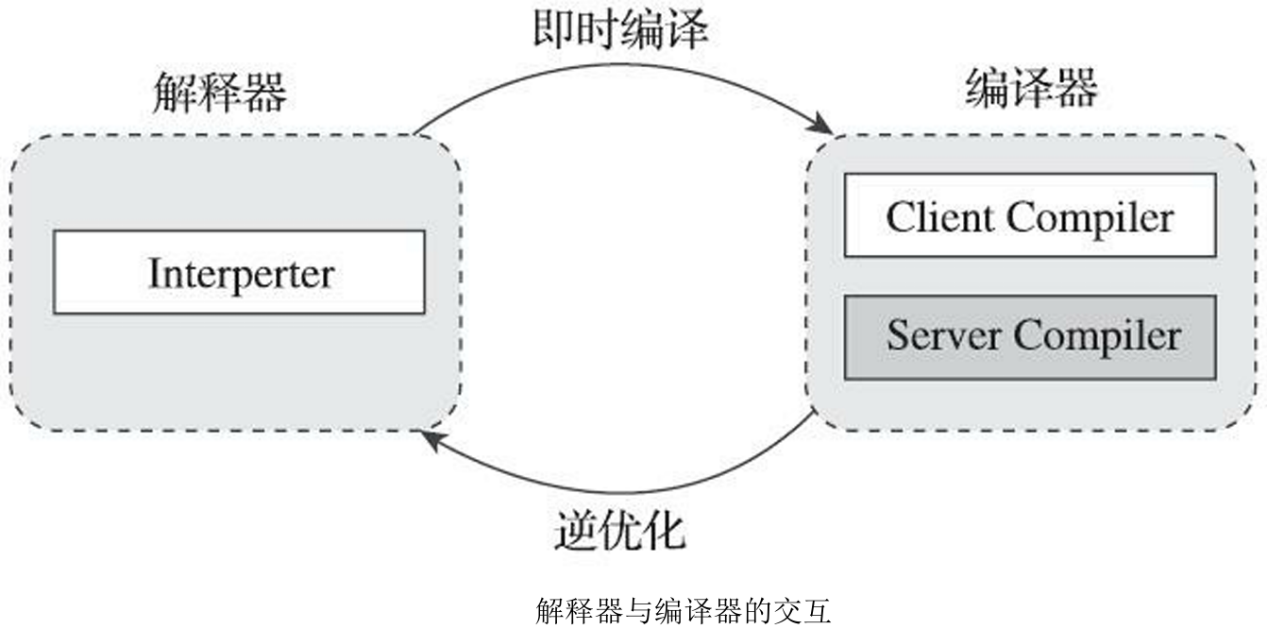
主流的两款商用Java虚拟机（HotSpot、OpenJ9）里，Java程序最初都是通过解释器（Interpreter）进行解释执行的，当虚拟机发现某个方法或代码块的运行特别频繁，就会把这些代码认定为“热点代码”（Hot Spot Code），为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成本地机器码，并以各种手段尽可能地进行代码优化，运行时完成这个任务的后端编译器被称为即时编译器。

## 解释器与编译器

尽管并不是所有的Java虚拟机都采用解释器与编译器并存的运行架构，但目前主流的商用Java虚拟机，譬如HotSpot、OpenJ9等，内部都同时包含解释器与编译器，解释器与编译器两者各有优势：

* 当程序需要迅速启动和执行的时候，解释器可以首先发挥作用，省去编译的时间，立即运行。当程序启动后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码，这样可以减少解释器的中间损耗，获得更高的执行效率。
* 当程序运行环境中内存资源限制较大，可以使用解释执行节约内存（如部分嵌入式系统中和大部分的JavaCard应用中就只有解释器的存在），反之可以使用编译执行来提升效率。
* 同时，解释器还可以作为编译器激进优化时后备的“逃生门”（如果情况允许，HotSpot虚拟机中也会采用不进行激进优化的客户端编译器充当“逃生门”的角色），让编译器根据概率选择一些不能保证所有情况都正确，但大多数时候都能提升运行速度的优化手段，当激进优化的假设不成立，如加载了新类以后，类型继承结构出现变化、出现“罕见陷阱”（Uncommon Trap）时可以通过逆优化（Deoptimization）退回到解释状态继续执行。

在整个Java虚拟机执行架构里，解释器与编译器经常是相辅相成地配合工作，其交互关系如图所示：



HotSpot虚拟机中内置了两个（或三个）即时编译器，其中有两个编译器存在已久，分别被称为“客户端编译器”（Client Compiler）和“服务端编译器”（Server Compiler），或者简称为C1编译器和C2编译器（部分资料和JDK源码中C2也叫Opto编译器），第三个是在JDK 10时才出现的、长期目标是代替C2的Graal编译器。

在分层编译（Tiered Compilation）的工作模式出现以前，HotSpot虚拟机通常是采用解释器与其中一个编译器直接搭配的方式工作，程序使用哪个编译器，只取决于虚拟机运行的模式，HotSpot虚拟机会根据自身版本与宿主机器的硬件性能自动选择运行模式，用户也可以使用“-client”或“-server”参数去强制指定虚拟机运行在客户端模式还是服务端模式。

无论采用的编译器是客户端编译器还是服务端编译器，解释器与编译器搭配使用的方式在虚拟机中被称为“混合模式”（Mixed Mode），用户也可以使用参数“-Xint”强制虚拟机运行于“解释模式”（Interpreted Mode），这时候编译器完全不介入工作，全部代码都使用解释方式执行。

另外，也可以使用参数“-Xcomp”强制虚拟机运行于“编译模式”（Compiled Mode），这时候将优先采用编译方式执行程序，但是解释器仍然要在编译无法进行的情况下介入执行过程。可以通过虚拟机的“ -version”命令的输出结果显示出这三种模式：

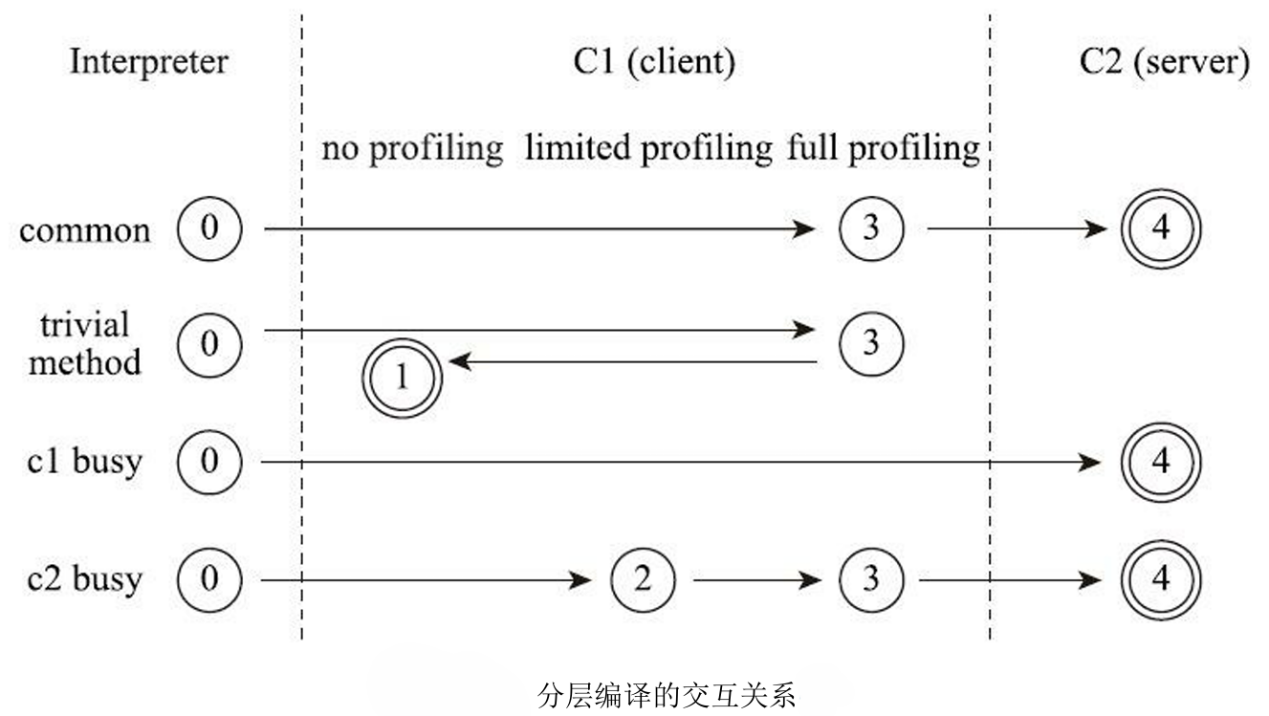
|  |
| --- |
| java -version  java -Xint -version  java -Xcomp -version |

由于即时编译器编译本地代码需要占用程序运行时间，通常要编译出优化程度越高的代码，所花费的时间便会越长；而且想要编译出优化程度更高的代码，解释器可能还要替编译器收集性能监控信息，这对解释执行阶段的速度也有所影响。为了在程序启动响应速度与运行效率之间达到最佳平衡，HotSpot虚拟机在编译子系统中加入了分层编译的功能，分层编译的概念其实很早就已经提出，但直到JDK 6时期才被初步实现，后来一直处于改进阶段，最终在JDK 7的服务端模式虚拟机中作为默认编译策略被开启。

分层编译根据编译器编译、优化的规模与耗时，划分出不同的编译层次，其中包括：

* 第0层。程序纯解释执行，并且解释器不开启性能监控功能（Profiling）。
* 第1层。使用客户端编译器将字节码编译为本地代码来运行，进行简单可靠的稳定优化，不开启性能监控功能。
* 第2层。仍然使用客户端编译器执行，仅开启方法及回边次数统计等有限的性能监控功能。
* 第3层。仍然使用客户端编译器执行，开启全部性能监控，除了第2层的统计信息外，还会收集如 分支跳转、虚方法调用版本等全部的统计信息。
* 第4层。使用服务端编译器将字节码编译为本地代码，相比起客户端编译器，服务端编译器会启用更多编译耗时更长的优化，还会根据性能监控信息进行一些不可靠的激进优化。

以上层次并不是固定不变的，根据不同的运行参数和版本，虚拟机可以调整分层的数量。各层次编译之间的交互、转换关系如图：



实施分层编译后，解释器、客户端编译器和服务端编译器就会同时工作，热点代码都可能会被多次编译，用客户端编译器获取更高的编译速度，用服务端编译器来获取更好的编译质量，在解释执行的时候也无须额外承担收集性能监控信息的任务，而在服务端编译器采用高复杂度的优化算法时，客户端编译器可先采用简单优化来为它争取更多的编译时间。

## 编译对象与触发条件

在运行过程中会被即时编译器编译的目标是“热点代码”，这里所指的热点代码主要有两类，包括：

* 被多次调用的方法：一个方法被调用得多了，方法体内代码执行的次数自然就多，它成为“热点代码”是理所当然的。
* 被多次执行的循环体：为了解决当一个方法只被调用过一次或少量的几次，但是方法体内部存在循环次数较多的循环体，这样循环体的代码也被重复执行多次，因此这些代码也应该认为是“热点代码”。

对于这两种情况，编译的目标对象都是整个方法体，而不会是单独的循环体。

* 第一种情况，由于是依靠方法调用触发的编译，那编译器理所当然地会以整个方法作为编译对象，这种编译也是虚拟机中标准的即时编译方式。
* 而对于后一种情况，尽管编译动作是由循环体所触发的，热点只是方法的一行）会稍有不同，编译时会传入执行入口点字节码序号（Byte Code Index，BCI）。这种编译方式因为编译发生在方法执行的过程中，因此被很形象地称为“栈上替换”（On Stack Replacement，OSR），即方法的栈帧还在栈上，方法就被替换了。

那到底多少次才算“多次”呢？

要知道某段代码是不是热点代码，是不是需要触发即时编译，这个行为称为“热点探测”（Hot Spot Code Detection），其实进行热点探测并不一定要知道方法具体被调用了多少次，目前主流的热点探测判定方式有两种：

* 基于采样的热点探测（Sample Based Hot Spot Code Detection）。采用这种方法的虚拟机会周期性地检查各个线程的调用栈顶，如果发现某个（或某些）方法经常出现在栈顶，那这个方法就是“热点方法”。基于采样的热点探测的好处是实现简单高效，还可以很容易地获取方法调用关系（将调用堆栈展开即可），缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。
* 基于计数器的热点探测（Counter Based Hot Spot Code Detection）。采用这种方法的虚拟机会为每个方法（甚至是代码块）建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值就认为它是“热点方法”。这种统计方法实现起来要麻烦一些，需要为每个方法建立并维护计数器，而且不能直接获取到方法的调用关系。但是它的统计结果相对来说更加精确严谨。

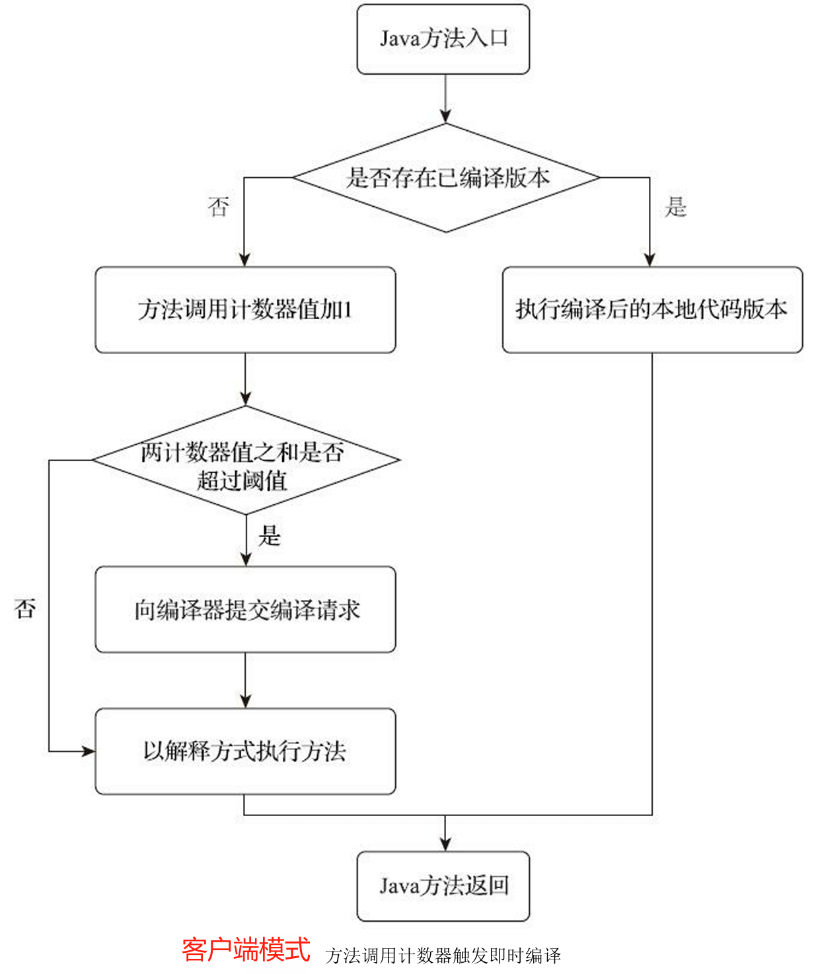
这两种探测手段在商用Java虚拟机中都有使用到，譬如J9用过第一种采样热点探测；

而在HotSpot虚拟机中使用的是第二种基于计数器的热点探测方法，为了实现热点计数，HotSpot为每个方法准备了两类计数器：方法调用计数器（Invocation Counter）和回边计数器（Back Edge Counter，“回边”的意思就是指在循环边界往回跳转）。当虚拟机运行参数确定的前提下，这两个计数器都有一个明确的阈值，计数器阈值一旦溢出，就会触发即时编译。

### 方法调用计数器

这个计数器就是用于统计方法被调用的次数，它的默认阈值在客户端模式下是1500次，在服务端模式下是10000次，这个阈值可以通过虚拟机参数-XX:CompileThreshold来人为设定。当一个方法被调用时，虚拟机会先检查该方法是否存在被即时编译过的版本，如果存在，则优先使用编译后的本地代码来执行。如果不存在已被编译过的版本，则将该方法的调用计数器值加一，然后判断方法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。一旦已超过阈值的话，将会向即时编译器提交一个该方法的代码编译请求。

如果没有做过任何设置，执行引擎默认不会同步等待编译请求完成，而是继续进入解释器按照解释方式执行字节码，直到提交的请求被即时编译器编译完成。当编译工作完成后，这个方法的调用入口地址就会被系统自动改写成新值，下一次调用该方法时就会使用已编译的版本了，整个即时编译的交互过程如图：



在默认设置下，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频率，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它提交给即时编译器编译，那该方法的调用计数器就会被减少一半，这个过程被称为方法调用计数器热度的衰减（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，这样只要系统运行时间足够长，程序中绝大部分方法都会被编译成本地代码。另外还可以使用-XX:CounterHalfLifeTime参数设置半衰周期的时间，单位是秒。

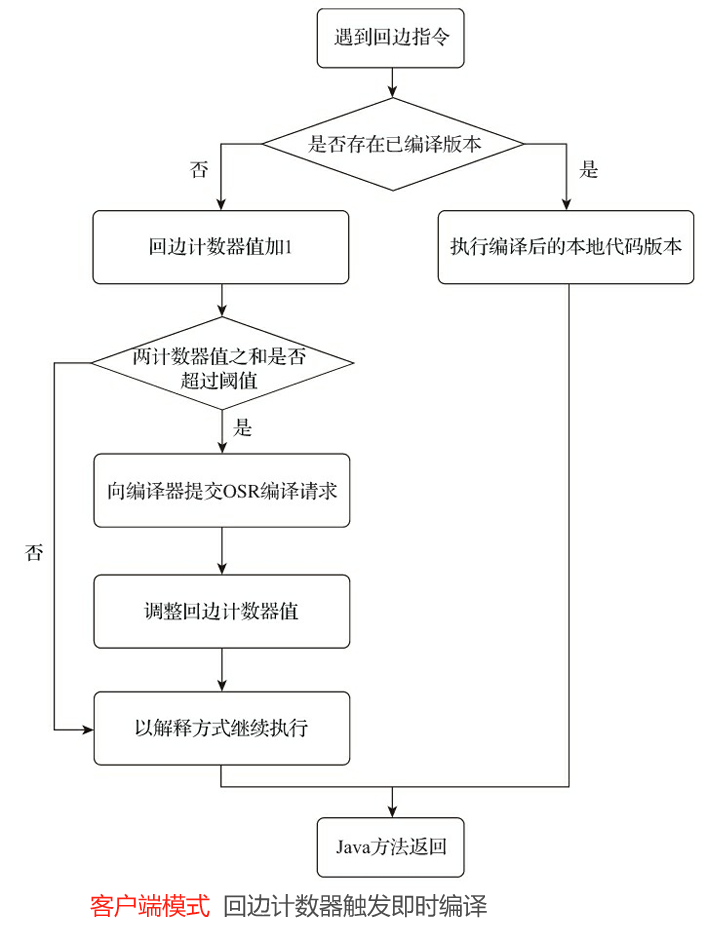
### 回边计数器

回边计数器，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就称为“回边（Back Edge）”，很显然建立回边计数器统计的目的是为了触发栈上的替换编译。

关于回边计数器的阈值，虽然HotSpot虚拟机也提供了一个类似于方法调用计数器阈值-XX:CompileThreshold的参数-XX:BackEdgeThreshold供用户设置，但是当前的HotSpot虚拟机实际上并未使用此参数，我们必须设置另外一个参数-XX:OnStackReplacePercentage来间接调整回边计数器的阈值，其计算公式有如下两种：

* 虚拟机运行在客户端模式下，回边计数器阈值计算公式为：方法调用计数器阈值（-XX:CompileThreshold）乘以OSR比率（-XX:OnStackReplacePercentage）除以100。其中-XX:OnStackReplacePercentage默认值为933，如果都取默认值，那客户端模式虚拟机的回边计数器的阈值为13995
* 虚拟机运行在服务端模式下，回边计数器阈值的计算公式为：方法调用计数器阈值（-XX:CompileThreshold）乘以（OSR比率（-XX:OnStackReplacePercentage）减去解释器监控比率（-XX:InterpreterProfilePercentage）的差值）除以100。其中-XX：OnStack ReplacePercentage默认值为140，-XX:InterpreterProfilePercentage默认值为33，如果都取默认值，那服务端模式虚拟机回边计数器的阈值为10700

当解释器遇到一条回边指令时，会先查找将要执行的代码片段是否有已经编译好的版本，如果有的话，它将会优先执行已编译的代码，否则就把回边计数器的值加一，然后判断方法调用计数器与回边计数器值之和是否超过回边计数器的阈值。当超过阈值的时候，将会提交一个栈上替换编译请求，并且把回边计数器的值稍微降低一些，以便继续在解释器中执行循环，等待编译器输出编译结果，整个执行过程如图：



与方法计数器不同，回边计数器没有计数热度衰减的过程，因此这个计数器统计的就是该方法循环执行的绝对次数。当计数器溢出的时候，它还会把方法计数器的值也调整到溢出状态，这样下次再进入该方法的时候就会执行标准编译过程。

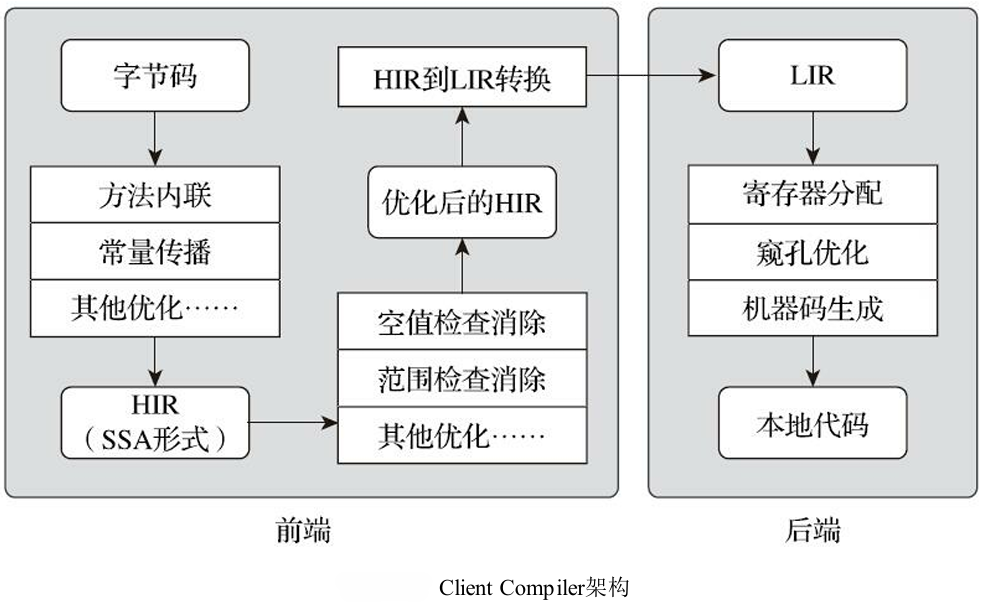
## 编译过程

在默认条件下，无论是方法调用产生的标准编译请求，还是栈上替换编译请求，虚拟机在编译器还未完成编译之前，都仍然将按照解释方式继续执行代码，而编译动作则在后台的编译线程中进行。用户可以通过参数-XX:-BackgroundCompilation来禁止后台编译，后台编译被禁止后，当达到触发即时编译的条件时，执行线程向虚拟机提交编译请求以后将会一直阻塞等待，直到编译过程完成再开始执行编译器输出的本地代码。

服务端编译器和客户端编译器的编译过程是有所差别的。对于客户端编译器来说，它是一个相对简单快速的三段式编译器，主要的关注点在于局部性的优化，而放弃了许多耗时较长的全局优化手段。

* 第一个阶段，一个平台独立的前端将字节码构造成一种高级中间代码表示（High-Level Intermediate Representation，HIR，即与目标机器指令集无关的中间表示）。HIR使用静态单分配（Static Single Assignment，SSA）的形式来代表代码值，这可以使得一些在HIR的构造过程之中和之后进行的优化动作更容易实现。在此之前编译器已经会在字节码上完成一部分基础优化，如方法内联、常量传播等优化将会在字节码被构造成HIR之前完成。
* 在第二个阶段，一个平台相关的后端从HIR中产生低级中间代码表示（Low-Level Intermediate Representation，LIR，即与目标机器指令集相关的中间表示），而在此之前会在HIR上完成另外一些优化，如空值检查消除、范围检查消除等，以便让HIR达到更高效的代码表示形式。
* 最后的阶段是在平台相关的后端使用线性扫描算法（Linear Scan Register Allocation）在LIR上分配寄存器，并在LIR上做窥孔（Peephole）优化，然后产生机器代码。

客户端编译器大致的执行过程如图：



而服务端编译器则是专门面向服务端的典型应用场景，并为服务端的性能配置针对性调整过的编译器，也是一个能容忍很高优化复杂度的高级编译器，几乎能达到GNU C++编译器使用-O2参数时的优化强度。它会执行大部分经典的优化动作，如：无用代码消除（Dead Code Elimination）、循环展开（Loop Unrolling）、循环表达式外提（Loop Expression Hoisting）、消除公共子表达式（Common Subexpression Elimination）、常量传播（Constant Propagation）、基本块重排序（Basic BlockReordering）等，还会实施一些与Java语言特性密切相关的优化技术，如范围检查消除（Range Check Elimination）、空值检查消除（Null Check Elimination，不过并非所有的空值检查消除都是依赖编译器优化的，有一些是代码运行过程中自动优化了）等。另外，还可能根据解释器或客户端编译器提供的性能监控信息，进行一些不稳定的预测性激进优化，如守护内联（Guarded Inlining）、分支频率预测（Branch Frequency Prediction）等。

服务端编译采用的寄存器分配器是一个全局图着色分配器，它可以充分利用某些处理器架构（如RISC）上的大寄存器集合。以即时编译的标准来看，服务端编译器无疑是比较缓慢的，但它的编译速度依然远远超过传统的静态优化编译器，而且它相对于客户端编译器编译输出的代码质量有很大提高，可以大幅减少本地代码的执行时间，从而抵消掉额外的编译时间开销，所以也有很多非服务端的应用选择使用服务端模式的HotSpot虚拟机来运行。

# 提前编译器

提前编译在Java技术体系中并不是新事物。1996年JDK 1.0发布，Java有了正式的运行环境，第一个可以使用外挂即时编译器的Java版本是1996年7月发布的JDK 1.0.2，而Java提前编译器的诞生并没有比这晚多少。仅几个月后，IBM公司就推出了第一款用于Java语言的提前编译器（IBM High Performance Compiler for Java）。在1998年，GNU组织公布了著名的GCC家族（GNU Compiler Collection）的新成员GNU Compiler for Java（GCJ，2018年从GCC家族中除名），这也是一款Java的提前编译器，而且曾经被广泛应用（GCJ其实包含了整个Java运行时，里面也有解释器和即时编译器存在）。在OpenJDK流行起来之前，各种Linux发行版带的Java实现通常就是GCJ。

但是提前编译很快又在Java世界里沉寂了下来，因为当时Java的一个核心优势是平台中立性，其宣传口号是“一次编译，到处运行”，这与平台相关的提前编译在理念上就是直接冲突的。GCJ出现之后在长达15年的时间里，提前编译这条故事线上基本就再没有什么大的新闻和进展了。类似的状况一直持续至2013年，直到在Android的世界里，剑走偏锋使用提前编译的ART（Android Runtime）横空出世。ART一诞生马上就把使用即时编译的Dalvik虚拟机按在地上使劲蹂躏，仅经过Android 4.4一个版本的短暂交锋之后，ART就迅速终结了Dalvik的性命，把它从Android系统里扫地出门。（ART干掉Dalvik之后，到Android 7.0时其内部也加入了解释执行和即时编译）

尽管Android并不能直接等同于Java，但两者毕竟有着深厚渊源，提前编译在Android上的革命与崛起也震撼到了Java世界。在某些领域、某些人眼里，只要能获得更好的执行性能，什么平台中立性、字节膨胀（指提前编译的本地二进制码的体积会明显大于字节码的体积）、动态扩展（指提前编译通常要求程序是封闭的，不能在外部动态加载新的字节码），一切皆可舍弃，唯一的问题就只有“提前编译真的会是获得更高性能的银弹吗？”

## 提前编译的优劣得失

现在提前编译产品和对其的研究有着两条明显的分支：

* 一条分支是做与传统C、C++编译器类似的，在程序运行之前把程序代码编译成机器码的静态翻译工作；
* 另外一条分支是把原本即时编译器在运行时要做的编译工作提前做好并保存下来，下次运行到这些代码（譬如公共库代码在被同一台机器其他Java进程使用）时直接把它加载进来使用。

先说第一条，这是传统的提前编译应用形式，它在Java中存在的价值直指即时编译的最大弱点：即时编译要占用程序运行时间和运算资源。即使现在先进的即时编译器已经足够快，以至于能够容忍相当高的优化复杂度了（譬如Azul公司基于LLVM的Falcon JIT，就能够以相当于Clang-O3的优化级别进行即时编译；又譬如OpenJ9的即时编译器Testarossa，它的静态版本同时也作为C、C++语言的提前编译器使用，优化的复杂度自然也支持得非常高）；即使现在先进的即时编译器架构有了分层编译的支持，可以先用快速但低质量的即时编译器为高质量的即时编译器争取出更多编译时间，但是，无论如何，即时编译消耗的时间都是原本可用于程序运行的时间，消耗的运算资源都是原本可用于程序运行的资源，这个约束从未减弱，更不会消失，始终是悬在即时编译头顶的达摩克利斯之剑。

举个更具体的例子来帮助理解这种约束：在编译过程中最耗时的优化措施之一是通过“过程间分析”（Inter-Procedural Analysis，IPA，也经常被称为全程序分析，即Whole Program Analysis）来获得诸如某个程序点上某个变量的值是否一定为常量、某段代码块是否永远不可能被使用、在某个点调用的某个虚方法是否只能有单一版本等的分析结论。这些信息对生成高质量的优化代码有着极为巨大的价值，但是要精确（譬如对流敏感、对路径敏感、对上下文敏感、对字段敏感）得到这些信息，必须在全程序范围内做大量极耗时的计算工作，目前所有常见的Java虚拟机对过程间分析的支持都相当有限，要么借助大规模的方法内联来打通方法间的隔阂，以过程内分析（Intra-Procedural Analysis，只考虑过程内部语句，不考虑过程调用的分析）来模拟过程间分析的部分效果；要么借助可假设的激进优化，不求得到精确的结果，只求按照最可能的状况来优化，有问题再退回来解析执行。但如果是在程序运行之前进行的静态编译，这些耗时的优化就可以放心大胆地进行了，譬如Graal VM中的Substrate VM，在创建本地镜像的时候，就会采取许多原本在HotSpot即时编译中并不会做的全程序优化措施以获得更好的运行时性能，反正做镜像阶段慢一点并没有什么大影响。同理，这也是ART打败Dalvik的主要武器之一，连副作用也是相似的。在Android 5.0和6.0版本，安装一个稍微大一点的Android应用都是按分钟来计时的，以至于从Android 7.0版本起重新启用了解释执行和即时编译（但这已与Dalvik无关，它彻底凉透了），等空闲时系统再在后台自动进行提前编译。

再说第二条，本质是给即时编译器做缓存加速，去改善Java程序的启动时间，以及需要一段时间预热后才能到达最高性能的问题。这种提前编译被称为动态提前编译（Dynamic AOT）或者索性就大大方方地直接叫即时编译缓存（JIT Caching）。在目前的Java技术体系里，这条路径的提前编译已经完全被主流的商用JDK支持。在商业应用中，这条路径最早出现在JDK 6版本的IBM J9虚拟机上，那时候在它的CDS（Class Data Sharing）功能的缓存中就有一块是即时编译缓存。不过这个缓存和CDS缓存一样是虚拟机运行时自动生成的，直接来源于J9的即时编译器，而且为了进程兼容性，很多激进优化都不能肆意运用，所以编译输出的代码质量反而要低于即时编译器。

真正引起业界普遍关注的是OpenJDK/OracleJDK 9中所带的Jaotc提前编译器，这是一个基于Graal编译器实现的新工具，目的是让用户可以针对目标机器，为应用程序进行提前编译。HotSpot运行时可以直接加载这些编译的结果，实现加快程序启动速度，减少程序达到全速运行状态所需时间的目的。这里面确实有比较大的优化价值，试想一下，各种Java应用最起码会用到Java的标准类库，如java.base等模块，如果能够将这个类库提前编译好，并进行比较高质量的优化，显然能够节约不少应用运行时的编译成本。这的确是很好的想法，但实际应用起来并不是那么容易，原因是这种提前编译方式不仅要和目标机器相关，甚至还必须与HotSpot虚拟机的运行时参数绑定。譬如虚拟机运行时采用了不同的垃圾收集器，这原本就需要即时编译子系统的配合（典型的如生成内存屏障代码）才能正确工作，要做提前编译的话，自然也要把这些配合的工作平移过去。至于前面提到过的提前编译破坏平台中立性、字节膨胀等缺点当然还存在，这里就不重复了。尽管还有许多困难，但提前编译无疑已经成为一种极限榨取性能（启动、响应速度）的手段，且被官方JDK关注，相信日后会更加灵活、更加容易使用，就如已经相当成熟的CDS（AppCDS需要用户参与）功能那样，几乎不需要用户介入，可自动完成。

最后，还要思考一个问题：提前编译的代码输出质量，一定会比即时编译更高吗？提前编译因为没有执行时间和资源限制的压力，能够毫无顾忌地使用重负载的优化手段，这当然是一个极大的优势，但即时编译难道就没有能与其竞争的强项了吗？当然是有的，尽管即时编译在时间和运算资源方面的劣势是无法忽视的，但其依然有自己的优势。接下来便要开始即时编译器的绝地反击了，介绍三种即时编译器相对于提前编译器的天然优势。

* 首先，是性能分析制导优化（Profile-Guided Optimization，PGO）。HotSpot的即时编译器时就多次提及在解释器或者客户端编译器运行过程中，会不断收集性能监控信息，譬如某个程序点抽象类通常会是什么实际类型、条件判断通常会走哪条分支、方法调用通常会选择哪个版本、循环通常会进行多少次等，这些数据一般在静态分析时是无法得到的，或者不可能存在确定且唯一的解，最多只能依照一些启发性的条件去进行猜测。但在动态运行时却能看出它们具有非常明显的偏好性。如果一个条件分支的某一条路径执行特别频繁，而其他路径鲜有问津，那就可以把热的代码集中放到一起，集中优化和分配更好的资源（分支预测、寄存器、缓存等）给它。
* 其次，是激进预测性优化（Aggressive Speculative Optimization），这也已经成为很多即时编译优化措施的基础。静态优化无论如何都必须保证优化后所有的程序外部可见影响（不仅仅是执行结果）与优化前是等效的，不然优化之后会导致程序报错或者结果不对，若出现这种情况，则速度再快也是没有价值的。然而，相对于提前编译来说，即时编译的策略就可以不必这样保守，如果性能监控信息能够支持它做出一些正确的可能性很大但无法保证绝对正确的预测判断，就已经可以大胆地按照高概率的假设进行优化，万一真的走到罕见分支上，大不了退回到低级编译器甚至解释器上去执行，并不会出现无法挽救的后果。只要出错概率足够低，这样的优化往往能够大幅度降低目标程序的复杂度，输出运行速度非常高的代码。譬如在Java语言中，默认方法都是虚方法调用，部分C、C++程序员（甚至一些老旧教材）会说虚方法是不能内联的，但如果Java虚拟机真的遇到虚方法就去查虚表而不做内联的话，Java技术可能就已经因性能问题而被淘汰很多年了。实际上虚拟机会通过类继承关系分析等一系列激进的猜测去做去虚拟化（Devitalization），以保证绝大部分有内联价值的虚方法都可以顺利内联。内联是最基础的一项优化措施。
* 最后，是链接时优化（Link-Time Optimization，LTO），Java语言天生就是动态链接的，一个个Class文件在运行期被加载到虚拟机内存当中，然后在即时编译器里产生优化后的本地代码，这类事情在Java程序员眼里看起来毫无违和之处。但如果类似的场景出现在使用提前编译的语言和程序上，譬如C、C++的程序要调用某个动态链接库的某个方法，就会出现很明显的边界隔阂，还难以优化。这是因为主程序与动态链接库的代码在它们编译时是完全独立的，两者各自编译、优化自己的代码。这些代码的作者、编译的时间，以及编译器甚至很可能都是不同的，当出现跨链接库边界的调用时，那些理论上应该要做的优化——譬如做对调用方法的内联，就会执行起来相当的困难。如果刚才说的虚方法内联让C、C++程序员理解还算比较能够接受的话（其实C++编译器也可以通过一些技巧来做到虚方法内联），那这种跨越动态链接库的方法内联在他们眼里可能就近乎于离经叛道了（但实际上依然是可行的）。

经过以上的讨论，应该能够理解提前编译器的价值与优势所在了，但忽略具体的应用场景就说它是万能的银弹，那肯定是有失偏颇的，提前编译有它的应用场景，也有它的弱项与不足，相信未来很长一段时间内，即时编译和提前编译都会是Java后端编译技术的共同主角。

# 编译器优化技术

编译器的目标虽然是做由程序代码翻译为本地机器码的工作，但其实难点并不在于能不能成功翻译出机器码，输出代码优化质量的高低才是决定编译器优秀与否的关键。

## 优化技术概览

OpenJDK的官方Wiki上，HotSpot虚拟机设计团队列出了一个相对比较全面的、即时编译器中采用的优化技术列表，其中有不少经典编译器的优化手段，也有许多针对Java语言，或者说针对运行在Java虚拟机上的所有语言进行的优化。

[PerformanceTacticIndex - PerformanceTacticIndex - OpenJDK Wiki](https://wiki.openjdk.org/display/HotSpot/PerformanceTacticIndex)

|  |  |
| --- | --- |
| **类型** | **优化技术** |
| 编译器策略（compiler tactics） | 延迟编译（delayed compilation） |
| 分层编译（tiered compilation） |
| 栈上替换（on-stack replacement） |
| 延迟优化（delayed reoptimization） |
| 程序依赖图表示（program dependence graph representation） |
| 静态单赋值表示（static single assignment representation） |
| 基于性能监控的优化技术（profile-based techniques） | 乐观空值断言（optimistic nullness assertions） |
| 乐观类型断言（optimistic type assertions） |
| 乐观类型增强（optimistic type strengthening） |
| 乐观数组长度增强（optimistic array length strengthening） |
| 裁剪未被选择的分支（untaken branch pruning） |
| 乐观的多态内联（optimistic N-morphic inlining） |
| 分支频率预测（branch frequency prediction） |
| 调用频率预测（call frequency prediction） |
| 基于证据的优化技术（proof-based techniques） | 精确类型推断（exact type inference） |
| 内存值推断（memory value inference） |
| 内存值跟踪（memory value tracking） |
| 常量折叠（constant folding） |
| 重组（reassociation） |
| 操作符退化（operator strength reduction） |
| 空值检查消除（null check elimination） |
| 类型检测退化（type test strength reduction） |
| 类型检测消除（type test elimination） |
| 代数简化（algebraic simplification） |
| 公共子表达式消除（common subexpression elimination） |
| 整数范围类型（integer range typing） |
| 数据流敏感重写（flow-sensitive rewrites） | 条件常量传播（conditional constant propagation） |
| 基于流承载的类型缩减转换（flow-carried type narrowing） |
| 无用代码消除（dead code elimination） |
| 语言相关的优化技术（language-specific techniques） | 类型继承关系分析（class hierarchy analysis） |
| 去虚拟机化（devirtualization） |
| 符号常量传播（symbolic constant propagation） |
| 自动装箱消除（autobox elimination） |
| 逃逸分析（escape analysis） |
| 锁消除（lock elision） |
| 锁膨胀（lock fusion） |
| 消除反射（de-reflection） |
| 内存及代码位置变换（memory and placement transformation） | 表达式提升（expression hoisting） |
| 表达式下沉（expression sinking） |
| 冗余存储消除（redundant store elimination） |
| 相邻存储合并（adjacent store fusion） |
| 交汇点分离（merge-point splitting） |
| 循环变换（loop transformations） | 循环展开（loop unrolling） |
| 循环剥离（loop peeling） |
| 安全点消除（safepoint elimination） |
| 迭代范围分离（iteration range splitting） |
| 范围检查消除（range check elimination） |
| 循环向量化（loop vectorization） |
| 全局代码调整（global code shaping） | 内联（inlining (graph integration)） |
| 全局代码外提（global code motion） |
| 基于热度的代码布局（heat-based code layout） |
| Switch调整（switch balancing） |
| 控制流图变换（control flow graph transformation） | 本地代码编排（local code scheduling） |
| 本地代码封包（local code bundling） |
| 延迟槽填充（delay slot filling） |
| 着色图寄存器分配（graph-coloring register allocation） |
| 线性扫描寄存器分配（linear scan register allocation） |
| 复写聚合（copy coalescing） |
| 常量分裂（constant splitting） |
| 复写移除（copy removal） |
| 地址模式匹配（address mode matching） |
| 指令窥孔优化（instruction peepholing） |
| 基于确定有限状态机的代码生成（DFA-based code generator） |

通过大家熟悉的Java代码变化来展示其中几种优化技术是如何发挥作用的。不过首先需要明确一点，即时编译器对这些代码优化变换是建立在代码的中间表示或者是机器码之上的，绝不是直接在Java源码上去做的，这里只是为了方便讲解，使用了Java语言的语法来表示这些优化技术所发挥的作用。

最初代码：

|  |
| --- |
| static class B {  int value;  final int get() {  return value;  }  }  public void foo() {  y = b.get();  // ...do stuff...  z = b.get();  sum = y + z;  } |

第一个要进行的优化是方法内联，它的主要目的有两个：一是去除方法调用的成本（如查找方法版本、建立栈帧等）；二是为其他优化建立良好的基础。方法内联膨胀之后可以便于在更大范围上进行后续的优化手段，可以获取更好的优化效果。因此各种编译器一般都会把内联优化放在优化序列最靠前的位置。内联后：

|  |
| --- |
| public void foo() {  y = b.value;  // ...do stuff...  z = b.value;  sum = y + z;  } |

第二步进行冗余访问消除（Redundant Loads Elimination），假设代码中间注释掉的“…do stuff…”所代表的操作不会改变b.value的值，那么就可以把“z=b.value”替换为“z=y”，因为上一句“y=b.value”已经保证了变量y与b.value是一致的，这样就可以不再去访问对象b的局部变量了。如果把b.value看作一个表达式，那么也可以把这项优化看作一种公共子表达式消除（Common Subexpression Elimination），优化后的代码：

|  |
| --- |
| public void foo() {  y = b.value;  // ...do stuff...  z = y;  sum = y + z;  } |

第三步进行复写传播（Copy Propagation），因为这段程序的逻辑之中没有必要使用一个额外的变量z，它与变量y是完全相等的，因此我们可以使用y来代替z。复写传播之后：

|  |
| --- |
| public void foo() {  y = b.value;  // ...do stuff...  y = y;  sum = y + y;  } |

第四步进行无用代码消除（Dead Code Elimination），无用代码可能是永远不会被执行的代码，也可能是完全没有意义的代码。因此它又被很形象地称为“Dead Code”， “y=y”是没有意义的，把它消除后：

|  |
| --- |
| public void foo() {  y = b.value;  // ...do stuff...  sum = y + y;  } |

经过四次优化之后，两份代码所达到的效果是一致的，但是前者比后者省略了许多语句，体现在字节码和机器码指令上的差距会更大，执行效率的差距也会更高。

* 最重要的优化技术之一：方法内联。
* 最前沿的优化技术之一：逃逸分析。
* 语言无关的经典优化技术之一：公共子表达式消除。
* 语言相关的经典优化技术之一：数组边界检查消除。

## 方法内联

内联被业内戏称为优化之母，因为除了消除方法调用的成本之外，它更重要的意义是为其他优化手段建立良好的基础，内联对其他优化手段的巨大价值：没有内联，多数其他优化都无法有效进行。例如下面testInline()方法的内部全部是无用的代码，但如果不做内联，后续即使进行了无用代码消除的优化，也无法发现任何“Dead Code”的存在。如果分开来看，foo()和testInline()两个方法里面的操作都有可能是有意义的。

|  |
| --- |
| public static void foo(Object obj) {  if (obj != null) {  System.out.println("do something");  }  }  public static void testInline(String[] args) {  Object obj = null;  foo(obj);  } |

方法内联的优化行为理解起来是没有任何困难的，不过就是把目标方法的代码原封不动地“复制”到发起调用的方法之中，避免发生真实的方法调用而已。

但实际上Java虚拟机中的内联过程却远没有想象中容易，甚至如果不是即时编译器做了一些特殊的努力，按照经典编译原理的优化理论，大多数的Java方法都无法进行内联。

无法内联的原因是Java方法解析和分派调用，只有使用invokespecial指令调用的私有方法、实例构造器、父类方法和使用invokestatic指令调用的静态方法才会在编译期进行解析。

除了上述四种方法之外（最多再除去被final修饰的方法这种特殊情况，尽管它使用invokevirtual指令调用，但也是非虚方法，《Java语言规范》中明确说明了这点），其他的Java方法调用都必须在运行时进行方法接收者的多态选择，它们都有可能存在多于一个版本的方法接收者，简而言之，Java语言中默认的实例方法是虚方法。

对于一个虚方法，如果不依赖上下文的话，编译器静态地去做内联的时候很难确定应该使用哪个方法版本。假如有ParentB和SubB是两个具有继承关系的父子类型，并且子类重写了父类的get()方法，那么b.get()是执行父类的get()方法还是子类的get()方法，这应该是根据实际类型动态分派的，而实际类型必须在实际运行到这一行代码时才能确定，编译器很难在编译时得出绝对准确的结论。

更糟糕的情况是，由于Java提倡使用面向对象的方式进行编程，而Java对象的方法默认就是虚方法，可以说Java间接鼓励了程序员使用大量的虚方法来实现程序逻辑。根据上面的分析可知，内联与虚方法之间会产生“矛盾”，那是不是为了提高执行性能，就应该默认给每个方法都使用final关键字去修饰呢？C和C++语言的确是这样做的，默认的方法是非虚方法，如果需要用到多态，就用virtual关键字来修饰，但Java选择了在虚拟机中解决这个问题。

为了解决虚方法的内联问题，Java虚拟机首先引入了一种名为类型继承关系分析（Class Hierarchy Analysis，CHA）的技术，这是整个应用程序范围内的类型分析技术，用于确定在目前已加载的类中，某个接口是否有多于一种的实现、某个类是否存在子类、某个子类是否覆盖了父类的某个虚方法等信息。这样，编译器在进行内联时就会分不同情况采取不同的处理：如果是非虚方法，那么直接进行内联就可以了，这种的内联是有百分百安全保障的；如果遇到虚方法，则会向CHA查询此方法在当前程序状态下是否真的有多个目标版本可供选择，如果查询到只有一个版本，那就可以假设“应用程序的全貌就是现在运行的这个样子”来进行内联，这种内联被称为守护内联（Guarded Inlining）。不过由于Java程序是动态连接的，说不准什么时候就会加载到新的类型从而改变CHA结论，因此这种内联属于激进预测性优化，必须预留好“逃生门”，即当假设条件不成立时的“退路”（Slow Path）。假如在程序的后续执行过程中，虚拟机一直没有加载到会令这个方法的接收者的继承关系发生变化的类，那这个内联优化的代码就可以一直使用下去。如果加载了导致继承关系发生变化的新类，那么就必须抛弃已经编译的代码，退回到解释状态进行执行，或者重新进行编译。

假如向CHA查询出来的结果是该方法确实有多个版本的目标方法可供选择，那即时编译器还将进行最后一次努力，使用内联缓存（Inline Cache）的方式来缩减方法调用的开销。这种状态下方法调用是真正发生了的，但是比起直接查虚方法表还是要快一些。内联缓存是一个建立在目标方法正常入口之前的缓存，它的工作原理大致为：在未发生方法调用之前，内联缓存状态为空，当第一次调用发生后，缓存记录下方法接收者的版本信息，并且每次进行方法调用时都比较接收者的版本。如果以后进来的每次调用的方法接收者版本都是一样的，那么这时它就是一种单态内联缓存（Monomorphic Inline Cache）。通过该缓存来调用，比用不内联的非虚方法调用，仅多了一次类型判断的开销而已。但如果真的出现方法接收者不一致的情况，就说明程序用到了虚方法的多态特性，这时候会退化成超多态内联缓存（Megamorphic Inline Cache），其开销相当于真正查找虚方法表来进行方法分派。

所以说，在多数情况下Java虚拟机进行的方法内联都是一种激进优化。事实上，激进优化的应用在高性能的Java虚拟机中比比皆是，极为常见。除了方法内联之外，对于出现概率很小（通过经验数据或解释器收集到的性能监控信息确定概率大小）的隐式异常、使用概率很小的分支等都可以被激进优化“移除”，如果真的出现了小概率事件，这时才会从“逃生门”回到解释状态重新执行。

## 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，它与类型继承关系分析一样，并不是直接优化代码的手段，而是为其他优化措施提供依据的分析技术。

逃逸分析的基本原理是：分析对象动态作用域，当一个对象在方法里面被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他方法中，这种称为方法逃逸；甚至还有可能被外部线程访问到，譬如赋值给可以在其他线程中访问的实例变量，这种称为线程逃逸；从不逃逸、方法逃逸到线程逃逸，称为对象由低到高的不同逃逸程度。

如果能证明一个对象不会逃逸到方法或线程之外（换句话说是别的方法或线程无法通过任何途径访问到这个对象），或者逃逸程度比较低（只逃逸出方法而不会逃逸出线程），则可能为这个对象实例采取不同程度的优化，如：

* 栈上分配（Stack Allocations）：在Java虚拟机中，Java堆上分配创建对象的内存空间几乎是Java程序员都知道的常识，Java堆中的对象对于各个线程都是共享和可见的，只要持有这个对象的引用，就可以访问到堆中存储的对象数据。虚拟机的垃圾收集子系统会回收堆中不再使用的对象，但回收动作无论是标记筛选出可回收对象，还是回收和整理内存，都需要耗费大量资源。如果确定一个对象不会逃逸出线程之外，那让这个对象在栈上分配内存将会是一个很不错的主意，对象所占用的内存空间就可以随栈帧出栈而销毁。在一般应用中，完全不会逃逸的局部对象和不会逃逸出线程的对象所占的比例是很大的，如果能使用栈上分配，那大量的对象就会随着方法的结束而自动销毁了，垃圾收集子系统的压力将会下降很多。栈上分配可以支持方法逃逸，但不能支持线程逃逸。
* 标量替换（Scalar Replacement）：若一个数据已经无法再分解成更小的数据来表示了，Java虚拟机中的原始数据类型（int、long等数值类型及reference类型等）都不能再进一步分解了，那么这些数据就可以被称为标量。相对的，如果一个数据可以继续分解，那它就被称为聚合量（Aggregate），Java中的对象就是典型的聚合量。如果把一个Java对象拆散，根据程序访问的情况，将其用到的成员变量恢复为原始类型来访问，这个过程就称为标量替换。假如逃逸分析能够证明一个对象不会被方法外部访问，并且这个对象可以被拆散，那么程序真正执行的时候将可能不去创建这个对象，而改为直接创建它的若干个被这个方法使用的成员变量来代替。将对象拆分后，除了可以让对象的成员变量在栈上（栈上存储的数据，很大机会被虚拟机分配至物理机器的高速寄存器中存储）分配和读写之外，还可以为后续进一步的优化手段创建条件。标量替换可以视作栈上分配的一种特例，实现更简单（不用考虑整个对象完整结构的分配），但对逃逸程度的要求更高，它不允许对象逃逸出方法范围内。
* 同步消除（Synchronization Elimination）：线程同步本身是一个相对耗时的过程，如果逃逸分析能够确定一个变量不会逃逸出线程，无法被其他线程访问，那么这个变量的读写肯定就不会有竞争，对这个变量实施的同步措施也就可以安全地消除掉。

关于逃逸分析的研究论文早在1999年就已经发表，但直到JDK 6，HotSpot才开始支持初步的逃逸分析，而且到现在这项优化技术尚未足够成熟，仍有很大的改进余地。不成熟的原因主要是逃逸分析的计算成本非常高，甚至不能保证逃逸分析带来的性能收益会高于它的消耗。如果要百分之百准确地判断一个对象是否会逃逸，需要进行一系列复杂的数据流敏感的过程间分析，才能确定程序各个分支执行时对此对象的影响。前面介绍即时编译、提前编译优劣势时提到了过程间分析这种大压力的分析算法正是即时编译的弱项。可以试想一下，如果逃逸分析完毕后发现几乎找不到几个不逃逸的对象，那这些运行期耗用的时间就白白浪费了，所以目前虚拟机只能采用不那么准确，但时间压力相对较小的算法来完成分析。

C和C++语言里面原生就支持了栈上分配（不使用new操作符即可），而C#也支持值类型，可以很自然地做到标量替换（但并不会对引用类型做这种优化）。在灵活运用栈内存方面，确实是Java的一个弱项。在现在仍处于实验阶段的Valhalla项目里，设计了新的inline关键字用于定义Java的内联类型，目的是实现与C#中值类型相对标的功能。有了这个标识与约束，以后逃逸分析做起来就会简单很多。

下面将通过一系列Java伪代码的变化过程来模拟逃逸分析是如何工作的。初始代码如下：

|  |
| --- |
| // 完全未优化的代码  public int test(int x) {  int xx = x + 2;  Point p = new Point(xx, 42);  return p.getX();  } |

省略了Point类的代码，这就是一个包含x和y坐标的POJO类型。

第一步，将Point的构造函数和getX()方法进行内联优化：

|  |
| --- |
| // 步骤1：构造函数内联后的样子  public int test(int x) {  int xx = x + 2;  Point p = point\_memory\_alloc(); // 在堆中分配P对象的示意方法  p.x = xx; // Point构造函数被内联后的样子  p.y = 42  return p.x; // Point::getX()被内联后的样子  } |

第二步，经过逃逸分析，发现在整个test()方法的范围内Point对象实例不会发生任何程度的逃逸，这样可以对它进行标量替换优化，把其内部的x和y直接置换出来，分解为test()方法内的局部变量，从而避免Point对象实例被实际创建，优化后的结果如下所示：

|  |
| --- |
| // 步骤2：标量替换后的样子  public int test(int x) {  int xx = x + 2;  int px = xx;  int py = 42  return px;  } |

第三步，通过数据流分析，发现py的值其实对方法不会造成任何影响，那就可以放心地去做无效代码消除得到最终优化结果，如下所示：

|  |
| --- |
| // 步骤3：做无效代码消除后的样子  public int test(int x) {  return x + 2;  } |

从测试结果来看，实施逃逸分析后的程序在MicroBenchmarks中往往能得到不错的成绩，但是在实际的应用程序中，尤其是大型程序中反而发现实施逃逸分析可能出现效果不稳定的情况，或分析过程耗时但却无法有效判别出非逃逸对象而导致性能（即时编译的收益）下降，所以曾经在很长的一段时间里，即使是服务端编译器，也默认不开启逃逸分析，甚至在某些版本（如JDK 6 Update 18）中还曾经完全禁止了这项优化，一直到JDK 7时这项优化才成为服务端编译器默认开启的选项。

如果有需要，或者确认对程序运行有益，用户也可以使用参数-XX:+DoEscapeAnalysis来手动开启逃逸分析，开启之后可以通过参数-XX:+PrintEscapeAnalysis来查看分析结果。有了逃逸分析支持之后，用户可以使用参数-XX:+EliminateAllocations来开启标量替换，使用+XX:+EliminateLocks来开启同步消除，使用参数-XX:+PrintEliminateAllocations查看标量的替换情况。

尽管目前逃逸分析技术仍在发展之中，未完全成熟，但它是即时编译器优化技术的一个重要前进方向，在日后的Java虚拟机中，逃逸分析技术肯定会支撑起一系列更实用、有效的优化技术。

## 公共子表达式消除

公共子表达式消除是一项非常经典的、普遍应用于各种编译器的优化技术，它的含义是：如果一个表达式E之前已经被计算过了，并且从先前的计算到现在E中所有变量的值都没有发生变化，那么E的这次出现就称为公共子表达式。对于这种表达式，没有必要花时间再对它重新进行计算，只需要直接用前面计算过的表达式结果代替E。

如果这种优化仅限于程序基本块内，便可称为局部公共子表达式消除（Local Common Subexpression Elimination），如果这种优化的范围涵盖了多个基本块，那就称为全局公共子表达式消除（Global Common Subexpression Elimination）。

下面举个简单的例子来说明它的优化过程：

|  |
| --- |
| int d = (c \* b) \* 12 + a + (a + b \* c); |

如果这段代码交给Javac编译器则不会进行任何优化，那生成的代码将如代码清单所示，是完全遵照Java源码的写法直译而成的。

|  |
| --- |
| # 未作任何优化的字节码  iload\_2 // b  imul // 计算b\*c  bipush 12 // 推入12  imul // 计算(c \* b) \* 12  iload\_1 // a  iadd // 计算(c \* b) \* 12 + a  iload\_1 // a  iload\_2 // b  iload\_3 // c  imul // 计算b \* c  iadd // 计算a + b \* c  iadd // 计算(c \* b) \* 12 + a + a + b \* c  istore 4 |

当这段代码进入虚拟机即时编译器后，它将进行如下优化：编译器检测到c\*b与b\*c是一样的表达式，而且在计算期间b与c的值是不变的。

因此这条表达式就可能被视为：

|  |
| --- |
| int d = E \* 12 + a + (a + E); |

这时候，编译器还可能（取决于哪种虚拟机的编译器以及具体的上下文而定）进行另外一种优化——代数化简（Algebraic Simplification），在E本来就有乘法运算的前提下，把表达式变为：

|  |
| --- |
| int d = E \* 13 + a + a; |

表达式进行变换之后，再计算起来就可以节省一些时间了。如果还对其他的经典编译优化技术感兴趣，可以参考《编译原理》（俗称龙书）中的相关章节。

## 数组边界检查消除

数组边界检查消除（Array Bounds Checking Elimination）是即时编译器中的一项语言相关的经典优化技术。我们知道Java语言是一门动态安全的语言，对数组的读写访问也不像C、C++那样实质上就是裸指针操作。如果有一个数组foo[]，在Java语言中访问数组元素foo[i]的时候系统将会自动进行上下界的范围检查，即i必须满足“i>=0&&i<foo.length”的访问条件，否则将抛出一个运行时异常：java.lang.ArrayIndexOutOfBoundsException。这对软件开发者来说是一件很友好的事情，即使程序员没有专门编写防御代码，也能够避免大多数的溢出攻击。

但是对于虚拟机的执行子系统来说，每次数组元素的读写都带有一次隐含的条件判定操作，对于拥有大量数组访问的程序代码，这必定是一种性能负担。

无论如何，为了安全，数组边界检查肯定是要做的，但数组边界检查是不是必须在运行期间一次不漏地进行则是可以“商量”的事情。例如下面这个简单的情况：数组下标是一个常量，如foo[3]，只要在编译期根据数据流分析来确定foo.length的值，并判断下标“3”没有越界，执行的时候就无须判断了。

更加常见的情况是，数组访问发生在循环之中，并且使用循环变量来进行数组的访问。如果编译器只要通过数据流分析就可以判定循环变量的取值范围永远在区间[0，foo.length)之内，那么在循环中就可以把整个数组的上下界检查消除掉，这可以节省很多次的条件判断操作。

把这个数组边界检查的例子放在更高的视角来看，大量的安全检查使编写Java程序比编写C和C++程序容易了很多，比如：数组越界会得到ArrayIndexOutOfBoundsException异常；空指针访问会得到NullPointException异常；除数为零会得到ArithmeticException异常……在C和C++程序中出现类似的问题，一个不小心就会出现Segment Fault信号或者Windows编程中常见的“XXX内存不能为Read/Write”之类的提示，处理不好程序就直接崩溃退出了。但这些安全检查也导致出现相同的程序，从而使Java比C和C++要做更多的事情（各种检查判断），这些事情就会导致一些隐式开销，如果不处理好它们，就很可能成为一项“Java语言天生就比较慢”的原罪。为了消除这些隐式开销，除了如数组边界检查优化这种尽可能把运行期检查提前到编译期完成的思路之外，还有一种避开的处理思路——隐式异常处理，Java中空指针检查和算术运算中除数为零的检查都采用了这种方案。

举个例子，程序中访问一个对象（假设对象叫foo）的某个属性（假设属性叫value），那以Java伪代码来表示虚拟机访问foo.value的过程为：

|  |
| --- |
| if (foo != null) {  return foo.value;  }else{  throw new NullPointException();  } |

在使用隐式异常优化之后，虚拟机会把上面的伪代码所表示的访问过程变为如下伪代码：

|  |
| --- |
| try {  return foo.value;  } catch (segment\_fault) {  uncommon\_trap();  } |

虚拟机会注册一个Segment Fault信号的异常处理器（伪代码中的uncommon\_trap()，务必注意这里是指进程层面的异常处理器，并非真的Java的try-catch语句的异常处理器），这样当foo不为空的时候，对value的访问是不会有任何额外对foo判空的开销的，而代价就是当foo真的为空时，必须转到异常处理器中恢复中断并抛出NullPointException异常。

进入异常处理器的过程涉及进程从用户态转到内核态中处理的过程，结束后会再回到用户态，速度远比一次判空检查要慢得多。当foo极少为空的时候，隐式异常优化是值得的，但假如foo经常为空，这样的优化反而会让程序更慢。幸好HotSpot虚拟机足够聪明，它会根据运行期收集到的性能监控信息自动选择最合适的方案。