1. 后端编译
2. 概述

将Class文件转换成二进制机器码的过程为编译过程的后端，有即时编译和提前编译两种，但是这两种编译器都不是Java虚拟机必须的组成部分。

1. 即时编译器
2. 概述

最初的Java虚拟机中都是通过解释执行的，当虚拟机发现某个方法或者代码块的运行特别频繁时，会将这些代码编译成本地机器码，并进行代码优化，这都是通过即时编译器完成的。

1. 解释器与编译器

主流的Java虚拟机中同时包含了解释器和编译器，当需要程序快速启动和运行时，解释器可以首先发挥作用，省去编译的时间；当程序运行一段时间后，编译器可以发挥作用，将更多的代码编译成本地代码，减少解释器的中间消耗，获得更高的执行效率。

当内存限制较大时，使用解释执行节约内存；反之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的后备逃生门。如果激进优化不成立时，可以通过逆优化退回到解释状态继续执行，因此解释器和编译器通常是相辅相成的。

1. 客户端编译器和服务端编译器

HotSpot虚拟机中有三个即时编译器，一个是客户端编译器（C1），另一个是服务端编译器（C2），第三个是JDK10才出现的Graal编译器用于替代C2。在分层编译出现之前，HotSpot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汇分局自身版本与硬件性能自动选择运行模式，如客户端模式和服务端模式，或者是混合模式。

1. 分层编译

由于即使编译器编译本地代码需要占用程序运行时间，且优化程度越高，花费时间越长，且解释器还需要替编译器收集性能监控信息，来优化编译，这会对解释执行阶段的速度有所影响。为了平衡程序启动相应速度与运行效率，HotSpot在编译子系统种加入了分层编译的功能。

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：

1. 第0层，纯解释执行，不开启性能监控；
2. 第1层，解释器+客户端编译器，不开启性能监控；
3. 第2层，解释器+客户端编译器，仅开启方法及回边次数统计监控；
4. 第3层，解释器+客户端编译器，开启全部性能监控，增加分支跳转，虚方法调用版本等；
5. 第4层，解释器+服务端编译器，更多耗时长的优化以及不可靠的激进优化；
6. 热点代码

被即时编译器编译的目标是热点代码，所谓的热点代码有两种，一种是被多次调用的方法，另一种是被多次执行的循环体，即方法内部的循环体循环次数过多。对于这两种热点代码，编译的目标对象都是整个方法体，对于第一种方法代码，编译整个方法体当然合理；对于第二种循环体，依然编译整个方法，只是执行入口（从方法第几条字节码执行开始执行）稍有不同，编译时会传入执行入口点字节码序号，这种编译方法因为发生在方法执行过程中，称为栈上替换，即方法的栈帧还在栈上，方法就被替换了。

对于热点代码的判定以及即时编译何时被触发，有两种判定方式：

1. 基于采样的热点探测，虚拟机会周期性地检查各个线程的调用栈顶，如果发现某些方法经常出现在栈顶，则判定为热点方法。这种方法简单高效，还容易获取方法调用关系，但是对于方法的热度难以确定，容易受到线程阻塞的影响。
2. 基于计数器的热点探测，为每个方法（代码块）建立计数器，统计方法的执行次数，若超过阈值，则判定是热点方法。

计数器也分为两种，一种是方法调用计数器，另一种是回边计数器，回边的意思激素是在循环边界往回跳转，当计数器的阈值溢出时，就会触发即时编译。

1. 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，即一段时间内调用的次数，当超过一定时间，如果方法的调用次数不足以提交给即时编译器编译，则计数器减半，这个过程称为方法调用计数器热度的衰减，这段时间称为半衰周期，在垃圾收集时进行衰减。方法调用时，首先检查是否存在被即时编译过的版本，如果不存在则将该方法的调用器计数+1，判断方法调用计数器和回边计数器之和是否超过方法调用计数器的阈值，如果超过，则向即时编译器提交一个该方法的代码编译请求。
2. 回边计数器，统计一个方法中循环体代码执行的次数，同样，当解释器遇到回边指令，也会查找是否有已经编译好的版本，后面的步骤同上。但是，则提交编译请求后，会把回边计数器的值降级，以便继续在解释器中执行循环。回边计数器没有热度衰减，因此统计的是该方法循环执行的绝对次数，当计数器溢出时，会把方法计数器的值也调整到溢出状态，则下次进入该方法直接执行标准编译过程。
3. 编译过程

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，虚拟机在编译器未完成编译之前，仍然按照解释方式继续执行代码，而编译动作在后台的编译线程中运行。对于客户端和服务端编译器来说，编译过程是有所差别的。

1. 客户端编译器

简单快读的三段式编译器，关注点在于局部性的优化，放弃了耗时较长的全局优化手段。

第一阶段，平台独立的前端将字节码构造成一种高级中间代码表示（HIR），使用静态单分配（SSA）的形式代表码值，使得一些在HIR的构造过程之中和之后的优化更易实现。在此之前编译器已经在字节码上完成了基础优化，如方法内联，常量传播。

第二阶段，平台相关的后端从HIR中产生低级中间代码表示（LIR），在此之前在HIR中完成另外的优化，如空值检查消除，范围检查消除，以便让HIR达到更高效的代码表示形式。

第三阶段，平台相关的后端使用线性扫描算法在LIR上分配寄存器，并在LIR上做窥孔优化，产生机器代码。

1. 服务端编译器

专门面向服务端，能够容忍很高优化复杂度，执行大部分经典的优化动作以及激进的优化，虽然他的编译速度相对于客户端编译器会慢很多，但是它的代码质量提高很多，大幅减少了本地代码的执行事件，从而抵消了额外的编译时间开销。

1. 提前编译

提前编译有两条分支，一条是在程序运行之前将代码编译成机器码的静态翻译工作；另一条是将原本即时编译器在运行时做的编译工作提前做好并且缓存下来。由于即时编译要占用程序运行时间和运算资源，提前编译的作用就是节省时间，提高性能，由于提前编译没有了执行时间和资源限制的压力，可以采用重负载的优化手段。

但是，即时编译相对于提前编译也是有以下的优势的：

1. 性能分析制导优化

解释器会收集性能监控信息，依照监控信息来精确优化。

1. 激进预测性优化

即时编译在激进优化后可以逆优化回到低级编译器甚至是解释器上执行，不会出现灾难后果。

1. 链接时优化

Java天生就是动态链接的，Class文件在运行期被加载到虚拟机内存中，然后在即时编译器里产生优化后的本地代码。但是对于提前编译的语言和程序，如C++的程序要调用某个动态链接库的方法，就会有明显的隔阂，难以优化。