链滴

编译及编译优化技术

作者：skyesx

原文链接：https://ld246.com/article/1472228358177

来源网站：链滴

<h2 id="toc_h2_0">JAVA编译期优化（编译原理没学太好...这部分内容后来再补齐）</h2>
编译过程

解析与填充符号表过程
插入式注解处理器的注解处理过程
分析与字节码生成的过程

<h3 id="toc_h3_1">解析与填充符号表</h3>

词法语法分析

词法分析（分析出一个个不可分割的编译语义最小单元TOKEN）
语法分析（通过TOKEN序列构造语法树）

填充符号表(???)

<h3 id="toc_h3_2">注解处理器</h3>
允许编写注解处理器，并在编译过程中，干预编译过程
<h3 id="toc_h3_3">语义分析及字节码生成</h3>

标注检查

<h2 id="toc_h2_4">运行期编译优化（以下若无指明，则基于HOTSPOT）</h2>
大多JAVA虚拟机执行程序的时候 都有两种形式，一种是 解析执行，一种是编译执行。
解析执行启动快，编译后，执行快，但需要额外的编译时间。
JIT编译器要编译出优化的更好的机器代码需要更多的时间，因此大多虚拟机都会采用分层编译。OTSPOT中分层编译包含以下层级

第0层，程序解析执行，解析器不开启性能监控功能，可触发1层编译
第1层，也称为C1编译（Client Complier）,C1编译将字节码翻译为本地代码，并进行一些简单靠的编译。如有必要，将会为代码加上 性能监控（profiling)功能。
第2层，也称为C2编译（Server Complier），C2编译会启用一些编译耗时较长的优化，甚至于会据性能统计数据进行一些不可靠的激进优化。

<h3 id="toc_h3_5">编译触发条件</h3>

多次被调用的方法
被多次执行的循环体（触发后执行On Stack Replacement）

探测多次执行的方法：

基于采样的热点探测。（检测栈中方法出现的频率）

好处是 实现简单，高效。但缺点是不太准确

基于计数器的热点探测(HotSpot使用的)

好处是 准确。缺点 实现较为复杂。

使用的计数器

方法调用计数器
回边计数器（统计方法内循环，OSR用）

达到 回边计数器 或者 方法计数器 阈值 后，会通知编译器进行编译，然后继续以解析方式执行下一次进入本方法时会检测是否存在编译版本，若有，则执行编译版本。
<h3 id="toc_h3_6">一些编译优化技术</h3>
<h4 id="toc_h4_7">公共子表达式消除</h4>
相同的计算不再重复执行
<h4 id="toc_h4_8">数组范围检测消除</h4>
通过 代码流分析 是否存在越界可能，若无，则去掉数组范围检测
<h4 id="toc_h4_9">方法内联</h4>

减少运行栈
为其他优化手段建立好基础
因 JAVA的实例非privae方法都是虚方法，因此其大多都不能内联。但JAVA虚拟机会采用一些激优化手段，如分析当前所有的类中，有没有多个虚方法的实现，若无，则进行内联。若后期突然出现。则回退到解析执行中。即使其有多个虚方法实现，虚拟机也会尝试进行内联，当内联后发现确实调了不同的虚方法，才会取消内联的实现

<h4 id="toc_h4_10">逃逸分析</h4>

分析对象动态作用域。若对象被传给了其他方法，那么称为 方法逃逸。若被外部线程也能访问到则称为线程逃逸。
若对象未逃逸可进行一些优化：

栈上分配（对象在栈上分配，减轻GC压力）
同步消除（对该对象的锁都消除）
标量替换（将对象分解为java值类型，方便栈上分配，方便进一步的一些优化）

<h3 id="toc_h3_11">JAVA与C/C++编译器对比</h3>
C/C++编译器的优势

编译不在运行时，对编译时间不敏感，可采用大规模的优化技术
JAVA是动态类型安全语言，虚拟机需要保证程序不会违反语言语义或者方位非结构化的内存。就是在运行时有很多的检查。而C/C++并没有这些东西
JAVA的虚方法比C/C++多的多，给 内联编译增加了难度。
JAVA是可动态扩展的语言，在编译时难以看清程序全貌。很多全局优化措施只能通过激进手段行
JAVA对象内存分配都在堆上，C/C++可在 堆 也可在 栈 上分配对象。在栈上的对象回收方便，堆上的对象由于是自己管理内存，因此回收效率也高些。

JAVA优势： 基于运行时数据的统计优化