



下载APP



13 | 编译期优化：只有修改业务代码才能提升系统性能？

2021-06-15 尉刚强

《性能优化高手课》

课程介绍 >



讲述：尉刚强

时长 21:02 大小 19.28M



你好，我是尉刚强。

我们知道，所谓的编译，就是把我們编写的软件代码，变成计算机可以识别的汇编代码的过程，而这个编译过程会直接影响到最终运行的软件性能。所以今天这节课，我们就一起来聊聊编译期优化对软件性能的影响。

事实上，编译期优化是做软件性能优化时最常见的优化手段，它的最大优势就是可以在不用修改业务代码的场景下来提升软件的性能。另外，它也可以让开发人员以较低的成本获取一定的性能收益，所以编译期优化也算是高性能软件系统研发中不可或缺的一个环节。



不过同时它也存在**局限性**，那就是需要开发人员对语言实现和底层编译过程都有比较深入的理解，否则就会很容易漏掉一些优化方向，导致发挥不出最佳的性能优化效果。举个简单的例子，在我以前参与的一个 C++ 性能优化项目中，只是帮忙调整配置了关于内联相关的编译配置，就直接给产品带来了比较明显的性能提升，而团队之前的性能优化就没有考虑过这个方向。

而且除此之外，不同的编程语言，受制于其语言内置设计机制的差异，导致在编译期优化时的关注点和优化方法也有很大的不同。比如，有些编程语言（如 Java、Python、Ruby 等）将内存管理内置到了语言中，那么在做编译期优化时就需要重点关注内存空间这部分的优化。另外，由于编译期优化和语言的相关性很大，我也不可能逐一介绍。

所以在今天的课程中，我会**基于 C/C++ 和 Java 这两门语言**，来给你展开讲解编译期优化中比较常用的优化手段和关键原则，以此帮助你明确应该按照怎样的步骤与思路去开展编译期优化工作，从而让你能够在实际的软件研发过程中，选择合适的优化手段来帮助提升产品性能。

这里，我之所以选择 C/C++ 和 Java，一是因为这两门语言是通用语言中的典型代码，二是因为二者也正好代表了两种优化类型，也就是其最后执行指令分别是在虚拟机上执行和在 OS 上执行。这样，你在理解了 C/C++ 和 Java 的编译期优化手段之后，再去思考其他编程语言的编译期优化手段，就可以融会贯通了。

C/C++ 是传统编译型语言中的一个代表，它与底层硬件比较贴近，编译期优化配置手段也比较丰富。所以接下来，我们就先来看下针对 C/C++ 开发高性能系统时，应该如何在编译期进行调优。

C/C++ 编译期优化

首先，C/C++ 在不同操作系统中所使用的编译工具并不是统一的，比如有 GCC、Clang 等，这节课我主要是基于最通用的 **GCC 编译器**来给你展开介绍。而如果你的产品使用的是其他编译工具链，其大部分的编译优化手段和方法也都是相似的，所以你也可以借鉴参考。

那么下面，我就根据使用 GCC 支撑的编译优化角度，来给你介绍下 C/C++ 在编译期优化时最关键的一些编译配置手段和方法，主要包括编译期优化选项配置、编译辅助函数、C++ 语言特殊优化。

编译期优化选项配置

对于 GCC 来说，它提供的针对编译优化的选项配置开关主要有这几种：-O1，-O2，-O3，-Os，-Of，-Og 等。

其中，-O1，-O2，-O3 分别代表不同的编译优化级别，-Os 代表的是对 Size 的优化，-Of 代表的是 fast 的极速优化，-Og 代表支持 Debug 的优化，这些都属于 GCC 上最常用的编译配置开关，更详细的你可以参考 [官方文档](#)。

而在调整配置这些开关之前，你需要明白这些编译配置会对编译生成的汇编指令产生哪些影响，以避免盲目调整。然后在调整这些配置时，你还需要做好针对编译花费的时间长短、生成的二进制执行程序的大小、程序执行的性能、程序的可调试能力之间的权衡。比如说：

-Og 在程序中加入了定位调试信息，方便你跟踪定位问题；

从 -O1 到 -O3，编译出来的软件执行速度会越来越快，但也会导致编译时间越来越长，同时如果程序出现异常，你再回头基于汇编指令分析定位问题的难度也会加大；

-Os 在 -O2 的基础之上，会优化生成的目标文件的大小，所以它关闭了可能会使目标文件变大的部分优化选项；

-Of 则会开启所有 -O3 的编译开关，可能会对不符合标准的程序进行优化，所以潜在触发程序异常的概率会更高。

那么对于发布的软件版本来说，通常建议至少可以打开 -O2 级别，如果你的软件对性能要求比较高的话，也可以打开到 -O3（但这样可能会碰到一些因编译期优化而引起的故障问题）。在我之前做过的多个性能优化项目中，通过调整编译期优化开关，都可以给生成的软件带来 10%~20% 不等的性能收益。

其实，以上介绍每一级别的优化配置，都对应了一系列的详细编译配置，你可以根据业务代码进行更深入的配置和分析，具体的你可以参考 [GCC 的官方文档](#)。但是你也知道，对编译选项进行更详细的调整优化，其实性价比已经不太高了，一般情况下不建议你一定要这样做。

那么，在做好了编译期优化的选型配置之后，我们还需要选择编译辅助函数来优化生成软件的执行速度。下面我们就继续来看看，C/C++ 是如何与编译器进行配合，来实现更好的

编译期优化效果的。

编译辅助函数

其实，对编译器而言，如果它掌握的信息越多，那么编译生成的汇编指令的执行效率就越高。所以不少编译器都提供了编译期优化辅助函数，支持从代码中获取更多额外信息来指导编译。


但这里**你需要注意**的是，通常这些编译辅助函数并不在编程语言标准中，而是由编译器自定义的，所以使用前你需要谨慎选择。此外，如果使用编译辅助函数来优化生成软件的执行速度，也存在不少局限性，比如说：

使用内置辅助函数会污染到业务代码实现，导致代码可读性变差；

还有可能因为测试场景与产品运行环境的差异，导致编译辅助函数优化的实际效果并不理想；

如果因为引入了新的业务需求或代码重构，导致软件代码发生变化，也很容易引起内置辅助函数的优化成果失效。

不过，在一些对性能要求非常苛刻的场景下，你可能不得不需要使用这种手段。那么对于 C/C++ 语言来说，早期比较常用的编译辅助函数，应该是 **likely** 和 **unlikely** 宏，具体如下所示：

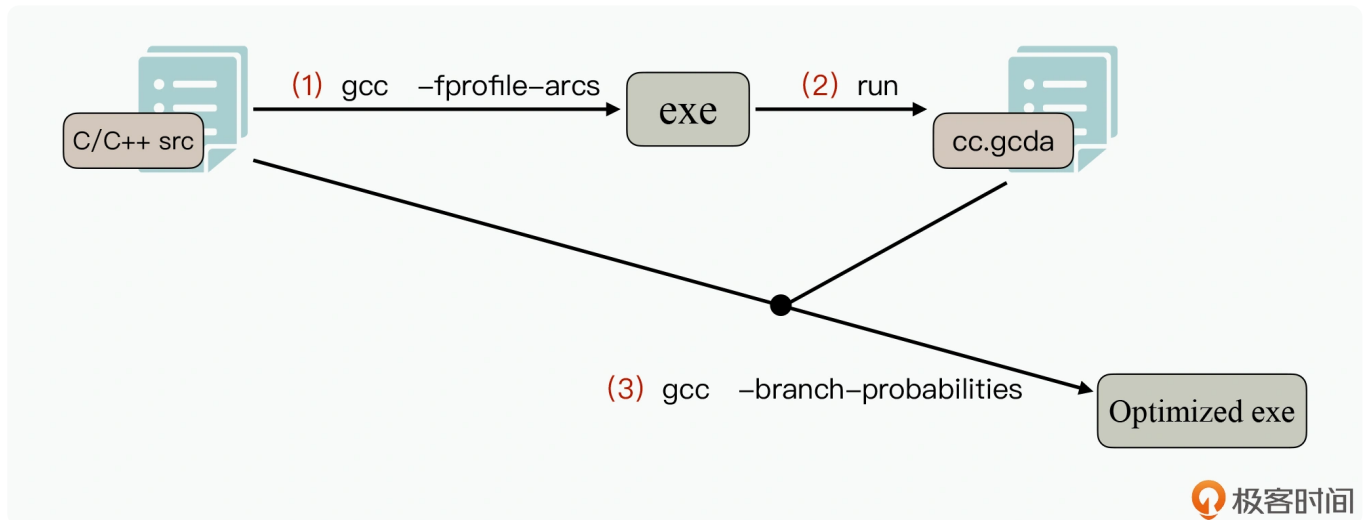
 复制代码

```
1 #define likely(x) __builtin_expect(!!(x), 1)
2 #define unlikely(x) __builtin_expect(!!(x), 0)
```

然后，我们可以将这两个宏放在代码中对应语义的条件判断分支上，来提升代码分支预测效率。

在上节课我也介绍过了，代码分支预测出错有可能会引起 CPU 指令流水线不连续，这是目前影响 CPU 硬件性能发挥的重要因素之一。而使用这个内置函数，就是主动告知编译器哪个分支会被更大概率地执行，从而让编译器在生成指令时优先选择大概率的分支，以此进一步提升 CPU 的执行效率。

不过，现在 GCC 也提供了更加方便的手段来帮助分支预测，具体流程如下图所示。



我来给你介绍下它具体实现的功能：

第一步，GCC 编译时使用 **-fprofile-arcs 选项**，编译源代码，生成可以跟踪分支预测的可执行程序。

第二步，启动可执行程序，并运行正常业务逻辑，这时候会生成很多 cc.gcda 文件，它们会记录相关的分支预测信息。

第三步，GCC 编译时使用 **-fbranch-probabilities 选项**，重新编译源文件时，会读取 cc.gcda 分支预测文件，从而最终可以生成执行效率比较高的可执行程序。

另外，还有一个比较常用的编译辅助函数，**指令预取指令** `__builtin_prefetch`，它可以实现在代码执行的过程中，预先加载代码段或数据段到 Cache 中，从而达到降低数据或数据 Cache Miss 的概率。比如，你可以看看这个二分法查找代码，它在查询的过程中，就可以提前一步把接下来要对比的数据，先加载到 Cache 中来提升性能。

复制代码

```
1  int binarySearch(int *array, int number_of_elements, int key) {
2      int low = 0, high = number_of_elements-1, mid;
3      while(low <= high) {
4          mid = (low + high)/2;
5          // low path
6          __builtin_prefetch (&array[(mid + 1 + high)/2], 0, 1);
7          // high path
8          __builtin_prefetch (&array[(low + mid - 1)/2], 0, 1);
9          if(array[mid] < key)
10             low = mid + 1;
11         else if(array[mid] == key)
```



```
12         return mid;
13     else if(array[mid] > key)
14         high = mid-1;
15     }
16     return -1;
17 }
```

我们知道，Cache 预取手段是 CPU 执行性能的优化手段之一，这是因为在通常情况下，在 CPU 硬件上的 Cache 容量是非常有限的，一旦出现 Cache Miss，就会导致 CPU 的执行速度不能完全发挥出来。而这里所使用预取指令，其实就是**通过显式地告知编译器在哪些场景下，把哪些代码段或数据段加载到 Cache 中，从而提升 Cache 命中率来优化性能。**


C++ 语言特殊优化

其实在 GCC 中，还有一些专门针对 C++ 语言的编译选项配置，它们对生成的可执行程序的性能影响也很大。这里我主要给你介绍下比较关键的一些配置，你可以根据实际业务中针对有关 C++ 语言特性的使用情况，去选择配置。

首先，在这些配置中，**最重要的一个就是内联的优化配置。**

通常我们在编写代码时，会使用 inline 关键字来定义方法，而这样的方法是否可以真实地被内联掉，还依赖于编译器。因此，对于 C++ 语言来说，编译中是否开启内联选项，对性能影响是非常大的。

下面给出的是最常用的、与内联相关的几个编译选项配置和具体的含义，你可以参考：

 复制代码

- 1 `'-fno-inline'` 忽略代码中的 inline 关键字
- 2 `'-finline-functions'` 编译期尝试将'**简单**'函数集成到调用代码处
- 3 `'-fearly-inlining'` 加速编译 默认可用
- 4 `'-finline-limit=N'` gcc 默认限制内联函数的大小，使用该选项可以控制内联函数的大小

其次，由于目前在 C++ 中，新的语言特性越来越复杂，所以在开发高性能系统时，有很多的特性会因为可能产生的性能问题而很少使用。那么，你就可以**在编译过程中，将这些编译特性的支持开关关闭掉**，也可以进一步提升生成的可执行程序的性能。

我举几个简单的例子。如果你开发的 C++ 代码中没有使用异常，可以使用 `-fno-exceptions` 编译选项来关闭。它不仅可以提升运行性能，还能减少编译生成的二进制文件大小。还有，如果你的代码中没有使用动态类型转换、`typeid` 运算符、`typeid` 等语法，那么你可以使用 `-fno-rtti` 选项关闭运行时 RTTI 机制，来提升性能，等等。

更换编译器

好，最后我要给你介绍的一项编译期调优手段，就是可以通过更换编译器来优化性能。

实际上，对 C/C++ 而言，由于 CPU 硬件的发展变化很快，同时厂家的编译优化技术差异也很大，就导致了编译器生成的二进制性能也会存在差异。比如说，Clang 编译出的软件执行速度，在大部分场景下都要比 GCC 要快，而不同的 GCC 版本之间的编译优化性能也存在差距。所以，在大部分业务场景下，你都可以尝试通过更换编译器或者编译器版本，来进一步的提升性能。

OK，前面我们介绍了 C/C++ 语言在编译期调优时的一些常用优化方法和手段。接下来，我们再来看看在 Java 语言当中，具体要如何更深入地挖掘出软件在编译期的性能优化点。

Java 的 JVM 优化

一般来说，使用 Java 语言来开发软件是基于 JVM 之上来运行的，这是因为 Java 语言将对象内存管理职责交给了 JVM，因此在很大程度上减轻了一线软件开发人员的负担。但同时，由于内存申请与释放操作对软件性能的影响非常大，所以**对 JVM 的堆空间配置**，就成为了 Java 性能调优中非常重要的手段之一。

而除此之外，在 JVM 中，我们可以**借助 JIT（即时编译器）**把 Java 生成的热点字节码，转换为可以直接在 CPU 上执行的机器码，从而提升软件执行速度的效果。与此同时，JIT 也对外提供了一些配置选项，方便我们直接对 JIT 的功能和过程进行配置，从而优化软件执行性能。

所以说，针对 JVM 的性能优化一般就主要集中在这两个点上，接下来，我就重点给你讲解这两个方面进行优化配置的思路和经验，方便你参考借鉴。

JVM 的堆空间配置

首先，对 JVM 的堆空间配置，实际上可以分为三个方向来进行，分别是针对堆内存资源大小、堆空间的内部分配和 JVM 中的 GC 算法选择。它们之间是递进的关系，在针对软件的启动配置优化过程中，你就应该按照这个顺序来进行优化配置。

1. JVM 堆内存资源配置

JVM 在启动时，可以配置使用的堆内存资源大小，从而对最终运行的软件性能产生比较直接的影响。那么，关于 JVM 堆内存资源的配置，这里我给你介绍最重要的两个参数，当你开发的软件在服务器部署运行时，使用的资源也会限制在这个资源空间范围内：

-Xms，初始堆大小配置；

-Xmx，最大堆内存配置。

这里你可能要问了，这两个参数具体要怎么使用呢？下面我就给你详细介绍下。

首先，我推荐你根据真实服务器的内存大小，留给 JVM 使用的最大内存空间，去最大化配置 JVM 使用的堆空间（-Xmx）。注意，如果这里 JVM 的最大堆空间配置比较小，就会容易出现 JVM 堆内存上频繁触发 GC，而服务器上还有空闲内存未被充分利用的场景。

其次，对于高吞吐量性能要求和低时延性能要求的软件来说，我比较推荐把 -Xms 和 -Xmx 设置成相同的值，这样可以在 JVM 启动时，就把相关堆内存创建好，以避免程序在运行期间再调整而引起时延抖动。

2. JVM 堆空间内部分配

JVM 的堆空间分为三个部分，分别是新生代、老生代和永久代。因此，你可以通过配置来改变应用的堆空间的分配比例，从而影响或改变软件应用的性能表现。

那么，关于 JVM 堆空间的内部分配，也有比较重要的三个参数需要你关注：

-XX:NewRatio：新生代和老生代的内存大小比例，如果配置 4，则表示新生代与老生代的空间占比是 1:4。

-XX:NewSize：新生代大小。

-XX:SurvivorRatio：Eden 和 Survivor 区的比率。

此外，在配置新生代与老生代的堆空间占比时，你需要重点考虑两个因素，一个是**业务软件中短期对象的占比状况**，另一个是**软件系统的时延要求是否足够高**。

如果业务软件中的短期对象比较多，我建议可以配置比较大的新生代堆空间占比，这样在一定程度上就可以减少触发 Minor GC 发生的概率。而系统针对时延要求很高的场景，为了避免新生代空间太大，触发 GC 后的执行时间长，造成业务的时延抖动比较大，你可以把新生代堆空间占比调小一些。

另外在一般情况下，当系统吞吐量比较大且时延要求不高的话，你就可以将新生代的堆空间配置得大一些。

3. JVM 中的 GC 算法选择及配置

JVM 中如何使用不同的 GC 算法，也会对软件应用的性能表现影响比较大，因此你需要根据业务性能要求差异，来选择配置合适的 GC 算法。

目前，JVM 中支持的 GC 算法种类比较多，这里我给你介绍其中最典型的三种，来了解下选择配置的方法。

ParallelScavenge (-XX:+UseParallelGC)，它是一个新生代收集器，如果你的业务对时延性能指标不是非常关注，可以考虑配置这种 GC 算法。

CMS (-XX:+UseConcMarkSweepGC)，它是一个老生代收集器，其 GC 执行时间会比较短，如果业务对响应时间要求比较高的话，你可以优先选择这个 GC 算法。

GarbageFirst G1 (-XX:+UseG1GC)，它是新生代和老生代都通用的收集器，在管理大的内存上有比较大的优势，因此当堆内存空间比较大时，推荐你去使用这种。

总之，每种 GC 算法在不同场景下的性能优势都是不一样的，你需要基于业务特性，来选择配置合适的 GC 算法。

JIT 优化

好，最后我们再来了解下 JIT 优化的相关配置，看看如何通过调整 JIT 的配置，来提升软件的执行速度。

首先，对于 JIT 来说，一般情况下包含了两种工作模式：

Client Compiler，简称 C1，-client 参数强制，代表**客户端模式**。

Server Compiler，简称 C2，-server 参数强制，代表**服务器模式**。

一般来说，使用 Client 可以获得更快的编译速度，而使用 Server 更容易获得较好的运行性能，所以在产品部署态，你可以先检查下 JVM 是否在服务器模式运行。而如果是在客户端模式下运行，你也可以通过显式地配置来运行服务器模式。

额外知识：GraalVM 是 Oracle 新开发的编译器，在目前的评测中，其生成代码的执行速度会优于 C2（服务器模式），你可以基于真实业务去对比分析下性能，看看是否需要采用。

然后，在 JIT 优化的过程中，也有一些比较重要的优化项，如果你需要对 JIT 进行更深入的优化配置来提升性能，就可以参考使用。

-XX:ReservedCodeCacheSize：调整代码缓存，避免因为缓存问题，导致无法进行 JIT 优化。

-XX:CompileThreshold：热点方法触发内联的阈值，当被执行次数超过这个门限值，才会进行内联优化。

-XX:UseFastAccessorMethods：是否针对 get 方法进行优化。

-XX:+UseCompressedOops：是否进行 64 位指针到 32 位指针的压缩优化等。

最后要注意，这里我只是给你简单介绍了下每个配置项的含义，你可以根据真实的业务软件，调整配置到更适合的值。

小结

在今天的课程中，我带你学习了在 C/C++ 和 Java 两门语言中，针对编译期优化可以使用的手段有哪些，并分享了这些手段在使用过程中的一些经验和建议。这里你要注意一点，

就是**你需要关注开发的业务软件的实现特点，以及它关注的性能指标是什么**，然后再利用这些编译期优化手段，来调整优化到一个比较好的性能效果。

最后呢，我还想重点强调下，这些经验与建议其实都是在一些特定业务场景下总结出来的，但不同行业的软件业务差异会比较大，可能一些建议并不一定适用于你的产品。所以，你在借鉴或者采纳这些经验和建议的时候，还需要先在业务中做进一步验证后，再去使用。

思考题

C/C++ 语言可以通过更换编译器来优化性能，那 Java 语言是不是也可以通过更换 JVM 来优化性能呢？

欢迎给我留言，分享你的思考和看法。如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 我们要先实现业务功能，还是先优化代码？

下一篇 14 | 内存使用篇：如何高效使用内存来优化软件性能？

更多学习推荐

Java 面试必考 300 题

最新汇总

限时免费领取



精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。