18 | 极致优化（上）：如何实现高性能的 C 程序？

2022-01-24 于航

《深入C语言和程序运行原理》 课程介绍

讲述：于航

时长 13:34　大小 12.44M

你好，我是于航。

我在开篇词中曾提到过，使用 C 语言正确实现的程序可以享受到最高的运行时性能。因此，如何编写具有“最高”执行性能的代码，是每个 C 程序员都在竭尽所能去探索的一个问题。那么，接下来的两讲，我们就来看看，如何编写高质量的 C 代码，来让我们的程序达到最佳的运行状态。

这一讲，我主要会为你介绍四个优化 C 代码的技巧，它们分别是利用高速缓存、利用代码内联、利用 `restrict` 关键字，以及消除不必要内存引用。

如何衡量程序的运行性能？

在开始正式介绍常用的性能优化技巧前，我们首先需要知道如何衡量一个应用程序的运行性能。

通常我们可以采用最简单和直观的两个宽泛指标，即内存使用率和运行时间。对于具有相同功能的两个程序，在完成相同任务时，哪一个的**内存使用率更低，且运行时间更短，则程序的整体性能相对更好**。

我们可以将这两个指标再进一步细分。比如程序的内存使用率，可以细分为程序在单位时间内的主内存使用率、L1-L3 高速缓存使用率、程序堆栈内存使用率等指标。而运行时间也可以细分为程序总体运行时间（墙上时钟时间）、用户与操作系统时间（CPU 时间），以及空闲时间与 IO 等待时间等。

至于这些指标的制定和使用方式，属于性能监控的另一个话题，这里我们就不展开了。但你需要知道的是，无论程序多么复杂，运行时间和内存消耗这两个指标都是可用于观察程序性能情况的基本指标。

这一讲和下一讲，我们会主要讨论可用于优化程序性能的具体编码方式。而采用这些编码方式能够保证你的代码在所有运行环境中，都拥有一个相对稳定且高效的执行情况。但你也需要注意，程序的实际运行可能会受到操作系统、编译器、网络环境等多方面的影响，因此，即使采用这些编码方式，也并不一定保证程序可以在所有执行环境上都能够获得较大的性能提升。

另外需要说明的是，这一讲我会以程序对应的 x86-64 汇编代码为例，来深入介绍各种优化技巧背后的逻辑。不过，这些基于 C 代码的性能优化策略都是相通的，只要掌握了它们的原理，在不同的平台体系上，你都可以做到举一反三，运用自如。下面，我们就来看具体的编码技巧吧。

技巧一：利用高速缓存

相信你在选购电脑时，经常会遇到 L1、L2、L3 高速缓存这几个指标，由此可见这些概念在衡量计算机整体质量中的重要性。那么，它们究竟代表什么呢？

我们都清楚缓存的重要性，通过将常用数据保存在缓存中，当硬件或软件需要再次访问这些数据时，便能以相对更快的速度进行读取和使用。而在现代计算机体系中，L1、L2、L3（某些体系可能有 L4 缓存，但不普遍）一般分别对应于 CPU 芯片上的三个不同级别的高速缓存，这些缓存的数据读写速度依次降低，但可缓存数据的容量却依次升高。而这些位于 CPU 芯片上的缓存，则主要用于临时存放 CPU 在最近一段时间内经常使用的数据和指令。

L1 到 L3 缓存直接由 CPU 硬件进行管理，当 CPU 想要读取某个数据时，它会按照一定规则优先从 L1 缓存中查找，如果没有找到，则再查找 L2 缓存，以此类推。而当在所有高速缓存中都没有找到目标数据时，CPU 便会直接从内存中读取。与直接从 L1 中读取相比，这个过程花费的时间会多出上百倍，所以你可以很清楚地看到优先将数据存放在高速缓存中的重要性。

高速缓存之所以能够提升性能，一个重要的前提便在于“局部性（locality）原理”。该原理通常被分为两个方面，即“时间局部性”和“空间局部性”，它们的内容分别如下所示。

- **时间局部性**：被引用过一次的内存位置可能在不远的将来会再被多次引用；
- **空间局部性**：如果一个内存位置被引用了一次，那么程序很可能在不久的将来引用其附近的另一个内存位置。

局部性本身是处理器访问内存行为的一种趋势，因此，如果一个程序在设计时能够很好地满足局部性原理，其运行便可能会有更高的性能表现。接下来我们就看看，如何让程序设计尽量满足这些原则。

在这里，我给出了一段不满足局部性原理的代码，你可以先思考下这段代码有什么问题，有没有可以进一步提升性能的空间。

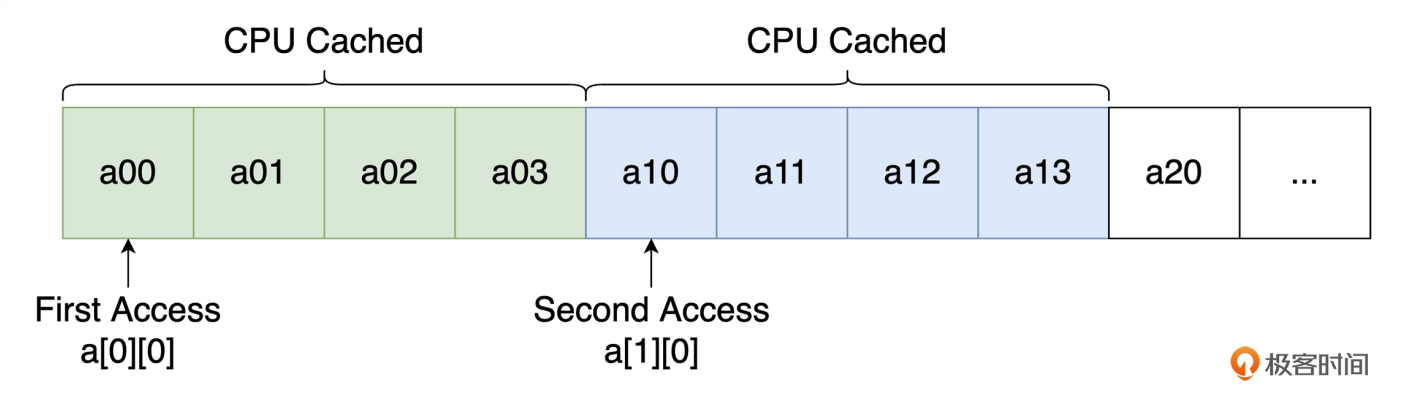
复制代码

```
#define M 128
#define N 64
int sum(int a[M][N]) {
  int i, j, sum = 0;
  for (j = 0; j < N; ++j) {
    for (i = 0; i < M; ++i) {
      sum += a[i][j];
    }
  }
  return sum;
}
```


可以看到的是，这段代码每次的内存访问过程（即访问数组 a 中的元素）都没有尽可能地集中在一段固定的内存区域上。相反，访存的过程发生在多个不同位置，且各个位置之间的跨度很大。这一点从代码中 sum 变量按照“列优先”顺序（先循环变量 i，后循环变量 j）访问数组元素的方式就可以得出。

我们知道，C 语言中二维数组的元素是按照“行优先”的方式存储的，也就是说，在上面的数组 a 中，连续不同的 j 将引用数组中连续的内存位置。CPU 在缓存数据时，会按照局部性原则，缓存第一次访问内存时其附近位置的一段连续数据。而列优先的访问方式则使得数组中数据的引用位置变得不连续，这在一定情况下可能会导致上一次已被放入高速缓存中的数据，在下一次数据访问时没有被命中。而由此带来的直接对内存的频繁访问，则会导致程序整体性能的降低。

因此我们的结论是，上述代码的设计并没有很好地满足空间局部性，其运行效率并没有被最大化。你可以通过下图来形象地理解这个问题：



可以看到，当代码的内层循环第一次访问数组 a 时，假设 CPU 会将临近的 4 个元素放入到它的高速缓存中。而由于代码采用了“行优先”访问，因此当下一次访问数组元素时，位于高速缓存中的数据便不会被命中。以此类推，后续的每次访问都会产生这样的问题。

因此，为了能更好地利用 CPU 高速缓存，你可以参考这些原则来编写代码：

- 尽量确保定义的局部变量能够被多次引用；
- 在循环结构中，以较短的步长访问数据；
- 对于数组结构，使用行优先遍历；
- 循环体越小，循环迭代次数越多，则局部性越好。

还记得我在上一讲中提到的自定义数据对齐吗？通过合理使用 alignas 关键字，我们也可以优化某些特定场景下的代码，来让它在最大程度上利用高速缓存。来看下面这个例子：

复制代码

```
1 struct data {
2   char x;
```

```
  alignas(64) char y[118];
};
```

在这段代码中，我们定义了一个名为 data 的结构。它包含两个字段，一个为字符类型的 x，另一个为包含有 118 个元素的字符数组 y。而通过指定 `alignas(64)` 标识符，我们限制字段 y 在内存中的起始地址需要在 64 字节上对齐。

实际上，在计算机内部，高速缓存是以“缓存行（Cache Line）”的形式被组织的。也就是说，一大块连续的高速缓存会被分为多个组，每个组中有多个行，而每个行则具有固定大小。当发生缓存不命中时，CPU 会将数据从低层次缓存中以固定的“块大小（通常为缓存行大小）”为单位，拷贝到高层次缓存中。**而为了减少 CPU 需要进行的内存拷贝次数，我们希望连续的数据能被组织在尽可能少的缓存行中。**

另一方面，内存与高速缓存之间的映射关系一般与数据在内存中的具体地址有关。比如，对于采用“直接映射”方式的缓存来说，假设缓存行大小为 64 字节，若某段数据起始于内存中对齐到 64 字节的地址，而当它被拷贝到高速缓存中时，便会从缓存行的开头处开始放置数据，这在最大程度上减少了这段连续数据需要占用的缓存行个数（当从缓存行中间开始存放数据时，字段 y 可能需要占用三个缓存行）。上面那段代码便是如此。

技巧二：代码内联

第二种常用于性能优化的方式是代码内联（Inlining）。这种方式很好理解，下面我们直接来看个例子，由此理解内联的概念和它对程序运行的影响。

C99 标准引入了一个名为 `inline` 的关键字，通过该关键字，我们可以建议编译器，将某个方法的实现内联到它的实际调用处。来看下面这个简短的例子：

复制代码

```
#include <stdio.h>
static inline int foo() {
  return 10;
}
int main(void) {
  int v = foo();
  printf("Output is: %d\n", v);
  return 0;
}
```

在这段代码中，我们使用 `inline` 关键字标注了方法 foo，并在 main 函数内将 foo 方法的调用返回结果赋值给了变量 v。为了能够看清 `inline` 关键字对程序实际运行的影响，我们还需要查看上述 C 代码对应的汇编代码，具体如下所示：

```
.LC0:
        .string "Output is: %d\n"
main:
        sub     rsp, 8
        mov     esi, 10
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

可以看到，在这段汇编代码中，实际上只有 main 函数的机器指令实现，而 foo 函数的具体定义则已经被替换到了它在 main 函数中的实际调用位置处（对应 `mov esi, 10` 这一行）。

通过这种方式，程序不再需要使用 `call` 指令来调用 foo 函数。这样做的好处在于，可以省去 `call` 指令在执行时需要进行的函数栈帧创建和销毁过程，以节省部分 CPU 时钟周期。而通过这种方式得到的性能提升，通常在函数被多次调用的场景中更加显而易见。

`inline` 固然好用，但我们也要注意这一点：函数本身作为 C 语言中对代码封装和复用的主体，不恰当的内联也会导致程序可执行二进制文件的增大，以及操作系统加载程序时的效率变低。一般情况下，内联仅适用于那些本身实现较为短小，且可能会被多次调用的函数。同时，`inline` 关键字也仅是程序员对编译器提出的一个建议，具体是否会被采纳，还要看具体编译器的实现。而对于大部分常用编译器来说，在高优化等级的情况下，它们也会默认采用内联来对代码进行优化。

当然，除此之外，你也可以选择通过宏来进行预处理时的代码内联。采用这种方式的话，你需要将 C 代码封装成对应的宏，并在需要内联的地方展开。

技巧三：`restrict` 关键字

C99 标准新增了一个名为 `restrict` 的关键字，可以优化代码的执行。该关键字只能用于指针类型，用以表明该指针是访问对应数据的唯一方式。

在计算机领域，有一个名为 aliasing 的概念。这个概念是说内存中的某一个位置，可以通过程序中多于一个的变量来访问或修改其包含的数据。而这可能会导致一个潜在的问题：即当通过其中的某个变量修改数据时，便会导致所有与其他变量相关的数据访问发生改变。

因此，aliasing 使得编译器难以对程序进行过多的优化。而在 C 语言中，`restrict` 关键字便可以解决这个问题。当然，如果你学习过 Rust，这也是其所有权机制的核心内容。下面我们来看一个例子。

复制代码

```c
#include <stdio.h>
void foo(int* x, int* y, int* restrict z) {
  *x += *z;
  *y += *z;
}
int main(void) {
  int x = 10, y = 20, z = 30;
  foo(&x, &y, &z);
  printf("%d %d %d", x, y, z);
  return 0;
}
```

在这段代码中，函数 foo 共接收三个整型指针参数，它的功能是将第三个指针指向变量的值，累加到前两个指针指向的变量上。其中，第三个参数 z 被标记为了 `restrict`，这表明我们向编译器做出了这样一个承诺：即在函数体 foo 内部，我们只会使用变量 z 来引用传入函数第三个指针参数对应的内存位置，而不会发生 aliasing。这样做使得编译器可以对函数的机器码生成做进一步优化。

来看下上面这段 C 代码对应的汇编代码：

复制代码

```
foo:
        mov     eax, DWORD PTR [rdx]
        add     DWORD PTR [rdi], eax
        add     DWORD PTR [rsi], eax
        ret
```


我们可以发现，在使用 `restrict` 关键字标注了 foo 函数的第三个参数后，在为指针 y 进行值累加前，编译器不会再重复性地从内存中读取指针 z 对应的值（对应上面第一行代码）。而

这对程序的执行来说，无疑是一种性能上的优化。

另外你需要注意的是，若一个指针已被标记为 `restrict`，但在实际使用时却发生了 aliasing，此时的行为是未定义的。

技巧四：消除不必要的内存引用

在某些情况下，可能我们只需要对程序的结构稍作调整，便能在很大程度上提升程序的运行性能。你可以先看看下面这段代码，思考下优化方式：

复制代码

```
#define LEN 1024
int data[LEN] = { ... };
int foo(int* dest) {
  *dest = 1;
  for (int i = 0; i < LEN; i++) {
    *dest = *dest * data[i];
  }
}
```

在这段代码中，函数 foo 主要用来计算数组 data 中所有元素的乘积，并将计算结果值拷贝给指针 dest 所指向的整型变量。函数的逻辑很简单，但当我们仔细观察函数 foo 内部循环逻辑的实现时，便会发现问题所在。

在这个循环中，为了保存乘积在每一次循环时产生的累积值，函数直接将这个值更新到了指针 dest 指向的变量中。并且，在每次循环开始时，程序还需要再从该变量中将临时值读取出来。我们知道，从内存中读取数据的速度是慢于寄存器的。因此，这里我们可以快速想到一个优化方案。优化后的代码如下所示：

复制代码

```
#define LEN 3
int data[LEN] = { 1,2,4 };
int foo(int* dest) {
  register int acc = 1;
  for (int i = 0; i < LEN; i++) {
    acc = acc * data[i];
  }
  *dest = acc;
}
```

在上面的代码中，我们一共做了两件事情：

1. 将循环中用于存放临时累积值的 “*dest” 替换为一个整型局部变量 “acc”；
2. 在定义时为该变量添加额外的 `register` 关键字，以建议编译器将该值存放在寄存器中，而非栈内存中。

通过消除不必要的内存引用，我们就能够减少程序访问内存的次数，进而提升一定的性能。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

这一讲，我主要介绍了可用于 C 代码优化的几种常见策略，它们分别是利用高速缓存、使用代码内联和 `restrict` 关键字，以及消除不必要的内存引用，具体如下：

- 高速缓存利用了 CPU 的局部性，使得满足局部性的程序可以更加高效地访问数据；
- 代码内联通过直接使用函数定义替换函数调用的方式，减少了程序调用 `call` 指令带来的额外开销；
- `restrict` 关键字通过限制指针的使用，避免 aliasing，进而给予了编译器更多的优化空间；
- 消除不必要的内存引用，则是通过减少程序与内存的交互过程，来进一步提升程序的运行效率。

思考题

最后，给你留个小作业：尝试了解一下什么是高速缓存的“抖动”，并在评论区告诉我你的理解。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

生成海报并分享

赞 1　提建议

上一篇　17丨标准库：断言、错误处理与对齐

下一篇　19丨极致优化（下）：如何实现高性能的 C 程序？

精选留言 (4)

Geek_98aed8
2022-02-12

高速缓存的抖动：
https://www.jianshu.com/p/3607c0f94526
函数中，因为CPU取块时，不同变量总不在同一个块中，导致每次都不命中，一直需要重新取；且反复取的块总是固定的几个，称为抖动
（本质是变量在内存中存放的方式不科学，这样理解？）

作者回复: 这个理解是没错的，本质上是由于程序设计正好与所在平台的高速缓存策略（包括物理特性）产生冲突导致的。比如 CSAPP 中的一个例子：

```
float foo(float x[8], float y[8]) {
  float sum = 0.0;
  int i;
  for (i = 0; i < 8; i++)
    sum += x[i] * y[i];
  return sum;
}
```

通常情况下，一个缓存行的大小即一个块大小。简单来看，假设在一共拥有 2 个高速缓存行（比如分在两个组），且每个缓存行大小为 16 字节的计算机中，上述代码逻辑在每一次交替访问数组 x 与 y 中的元素时，便可能会产生缓存抖动。但总的来看，缓存抖动在采用了“直接映射高速缓存”的体系中较为常见，而在“全相联高速缓存”及其他缓存策略中则相对较少发生。


ZR2021
2022-01-25

666，减少内存引用还是我头一次见到的优化方法，多谢老师！！！


白凤凰
2022-01-24

（当从缓存行中间开始存放数据时，字段 y 可能需要占用三个缓存行）。上面那段代码便是如此。请问老师，上面这段代码是指

```
struct data {
  char x;
   alignas(64) char y[118];
};
```

如果没有align(64)，怎么就知道y是从缓存行中间开始存放的呢？不一定占用三个缓存行啊。

作者回复: 是的，所以这里只是“可能”。因为对于这个 data 结构来说，由于它可以对齐到任意地址，因此编译器实际会如何布局，完全视程序整体实现而定。可以说是存在不确定的性能损耗风险。


i Love 3
2022-01-24

请问一下，高速缓存的“抖动”和false-sharing是一回事吗？是不是都可以使用数据填充的方法来解决？

作者回复: 看了一下你提到的 “False Sharing”，是的，两者的问题和解决方案实际上都是类似的，只不过是目标主体不同。一个是同个程序的不同代码，另一个是同个程序的不同线程。