



下载APP



## 12 | 我们要先实现业务功能，还是先优化代码？

2021-06-12 尉刚强

《性能优化高手课》

[课程介绍 >](#)**讲述：尉刚强**

时长 17:47 大小 16.29M



你好，我是尉刚强，今天我们一起聊一聊高性能编码技术。

在做软件设计咨询工作的时候，我经常发现有很多高性能软件产品的研发团队，在软件开发阶段只关注和实现业务的特性功能，然后等功能交付之后，才开始花费很长的时间，对软件代码进行调整优化。

而且我在跟这些程序员接触的过程中，还观察到了一个比较有趣的现象，就是大家普遍认为在软件编码实现阶段，过早地考虑代码优化意义不大，而是应该等到功能开发完成后，再基于打点 Profiling（数据分析）去优化代码实现。



其实这个想法是否可取，曾经也困扰过我，但当我经历了很多低级编码所导致的性能问题之后，我发现原来高性能编码实现是有很大大价值的，而且这能让我更好地处理编码实现优

化与 Profiling 优化之间的关系。

所以今天这节课，我会和你一起探讨下应该如何去看待高性能编码这件事，然后我会给你具体讲讲，实现高性能编码的出发点和典型的最佳实践。通过今天课程的学习，你就可以建立起一套高性能编码实现的价值观，同时也会掌握实现高性能编码的思路和方法，从而支撑你开发出高性能的软件代码。

## 建立正确的高性能编码价值观

首先，提到高性能编码，你肯定听说过现代计算机科学的鼻祖高德纳 ( Donald Knuth ) 的那句名言：

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

我们应该忘掉那些效率低下的事情，告诫自己在 97% 的情况下：过早优化是万恶之源。但是，我们也不应该在关键的 3% 上错过优化机会。

—— [Computer Programming as an Art](#) (1974) P671.

不过我想，可能很多程序员都只记住了这句话的前半部分，“97% 的情况下，过早优化是万恶之源”，而没有注意到这句话还有后半句：**我们不应该放弃掉那关键的 3% 的优化机会。**

所以这样造成的后果就是：过度推崇不要对代码进行提前优化，并以此来作为编写低性能软件代码的借口。也就是说，现在我们在软件编码的过程中碰到的大多数问题，并不是由于过早优化导致的，而是因为在编写代码时对执行效率不关注所导致的。

其实，**在编写代码阶段去追求高性能实现的意识非常重要**，主要有两个原因。

**第一个原因**，就是可能原本只是一个很小的编码问题，却有可能引起软件比较大的性能问题。

就比如说，我曾经参与的一个 C++ 高性能软件开发项目，因为一位研发人员在编码中不小心将函数行参的引用符号写丢了，导致函数调用开销增大，软件版本性能明显下降。而且

这个问题还比较隐蔽，我们后面花了很大力气，通过在代码中增加了很多定位手段后，才发现问题。

所以高性能实现的**第二个原因**，就是一旦你错过了高性能编码这个窗口，将性能问题遗漏到软件生命周期的后期，很有可能会因为错过了当时编写代码的上下文，后面就很难再发现问题。这里，我也再给你举个例子来说明下。

可以看到，在如下所示的代码片段中，这个类的实例在接收到数据之后，会更新各个 Channel 中的数据量大小，然后对外提供了一个方法，来判断所有通道中是否存在数据：

 复制代码

```
1 public class ChannelGroup {
2     class Channel{
3         public String channelname;
4         public int dataSize;
5     }
6
7     Channel[] channels;
8
9     public Channels() {
10         channels = new Channel[10];
11     }
12
13     public receiveData(...) {...} // 收到数据更新Channel中信息，省略
14     public boolean hasData() { // 判断所有通道是否有数据。
15         for (Channel channel : channels) {
16             if (channel.dataSize > 0){
17                 return true;
18             }
19         }
20         return false;
21     }
22 }
```

那么在看完代码之后，**你觉得这段代码实现中的方法 hasData，可以算是高性能实现吗？**如果你只是根据这段代码实现来进行分析，会发现它好像没有啥性能问题。毕竟，针对一个只有 10 个元素的数组来说，使用二分法查找来提升查找速度的必要性不太大。

好，我们可以先暂且这样认为，然后再接着来做个假设：在一个真实的编写代码过程中，有这样一个潜在的上下文信息，那就是绝大多数的业务场景下是第三个通道中收到数据。

那么针对这种情况，如果你再使用从前向后的顺序遍历，肯定就不是性能最佳的实现方式了，而是应该先判断第三个通道中的数据。

所以通过这两个例子，你应该就明白了，如果在编码实现阶段并没有从高性能实现的角度出发，而是意图在后续通过打点数据分析来优化解决问题，几乎是不太可能的。

其实，就我的思考和实践经验来说，**在开发一个高性能软件系统的时候，在编码阶段考虑高性能的实现方法，与完成业务功能后再进行代码调优之间并不矛盾，这二者应该要被同等地重视起来。**因为前期的高性能编码实现过程，很多都是由人来主观控制的，所以可能会由于判断不准确或者实现过程不小心，引入一些低效率的代码实现。这样一来，后期通过热点代码分析以及代码调优的过程，就是不能省略的。

而且说实话，我心目中优秀的软件代码，应该是集代码简洁与性能于一身，而如果是对编码性能不屑一顾的话，我想通常这样的程序员也写不出非常高质量的代码。

好了，在理解了应该如何去看待高性能编码之后，接下来的问题就是，如何才能掌握实现高性能编码的方法，下面我们就具体来看看。

## 高性能编码实现方法

其实在软件开发的过程中，高性能编码实现的方法和技术非常多，不同的编程语言之间也会存在一些差异，很难在一节课中介绍完整。

所以，今天我主要是**从编写的代码映射到执行过程的角度**，来给你介绍四种高性能编码实现方法，以及对应的实现原则和手段，分别是循环实现、函数方法实现、表达式实现以及控制流程实现。在实际的软件编码过程中，你也可以根据这样的角度和思路，尝试去理解和分析软件代码的运行态过程，逐步积累和完善高性能编码的实现手法。

好，接下来我们就从循环实现开始，来看看这种高性能实现的原则和方法。

## 高性能循环实现

我们都知道，在编写代码的时候，循环体内的代码会被执行很多遍，所以它的代码开销会被放大，经常会出现于热点代码中。也就是说，如何实现高效循环是实现高性能编码最重要的一步。

那么编写高效循环代码的重要参考原则都有哪些呢？我认为主要有两个，下面我们就具体了解下。

### 第一点，尽量避免对循环起始条件和终止条件的重复计算。

为了让你更容易理解这个原则，我先带你来看一个高效循环的反例。在下面这个代码示例中，实现的功能是循环遍历并更新字符串中的值，你会发现在循环执行的过程中，`strlen`被调用了很多次，所以性能比较低。

[复制代码](#)

```
1 void updateStr(char* str)
2 {
3     for(int i = 0; i<strlen(str); i++)
4     {
5         str[i]= '*';
6     }
7 }
```

那么，针对这种情况，我们就应该在循环开始时，将字符串长度值保存在一个变量中，从而避免重复计算。修改好的代码如下：

[复制代码](#)

```
1 void updateStr(char* str)
2 {
3     int length = strlen(str);
4     for(int i = 0; i< length; i++)
5     {
6         str[i]= '*';
7     }
8 }
```

### 第二点，尽量避免循环体中存在重复的计算逻辑。

我们同样也来看一个反模式的代码示例。在下面这段代码的实现过程中，`x*y`的值并没有发生变化，但是在循环体中被执行了很多遍。

[复制代码](#)

```
1 void initData(int[] data, int length, int x, int y){
```



```
2     for(int i = 0; i < length; i++)
3     {
4         data[i] = x * y + 100;
5     }
6 }
```

因此，站在高性能编码实现的角度，我们可以把 $x*y$ 的值的计算过程搬移到循环体外部，从而减少这部分的冗余计算开销。

其实到这里，你可以记住一句话：**编写高效循环代码的本质，就是尽量让循环体中执行的代码越少越好，剥离掉所有可以冗余的重复计算。**

那么在具体的代码实现中，需要检查的循环优化点其实还有很多。比如，你还需要检查是否有重复的函数调用、多余的对象申请和构造、多余的局部变量定义，等等。所以，在编写循环代码时，你需要注意识别并剥离出这样的代码实现。

## 高性能函数方法实现

实现高性能的函数方法，有两个重要的出发点：尽量通过内联来减少运行期函数调用，尽量减少不必要的运行期多态。接下来，我就来给你讲解下为什么要从这两个点出发，以及该如何去做。

### 第一点，尽量通过内联来减少运行期函数调用。

所谓的“通过内联”，意思就是将代码直接插入到代码调用中执行，以此减少运行期函数调用。那为什么要减少生成真实的运行期函数呢？

这是因为函数调用本身会产生一些额外的性能开销。在函数调用的过程中，需要先把当前局部变量压栈，在调用结束后还需要出栈操作，同时还需要更新相关寄存器。所以当函数体内的逻辑很小时，所产生的额外开销占比会比较高。因此，针对比较小的函数方法，我们可以尽量采用内联实现，从而减少不必要的调用开销。

其实不同的编程语言，支撑函数方法内联的语法和机制有一定的差异。在 Java 语言的开发过程中，我推荐你尽量使用 `final` 来定义方法，因为这种场景下，Java 的 JIT 会有比较大的概率将这个代码方法内联掉。

而在 C++ 中，针对一些热点小函数，你可以使用 `Inline` 关键字来定义方法，这样就可以显式地告知编译器尽量将代码内联掉。

补充：在早期 C 语言的开发过程中，因为没有内联语法，程序员还经常使用编译宏来定义方法，以此减少真实方法的调用开销。

但是，最终编译器或解释器是否可以将代码内联掉，还会有很多隐式约束条件，所以你需要在编码实现中多加注意。

## 第二点，尽量减少不必要的运行期多态。

多态的本质就是函数指针，它需要在运行过程中获取内存中变量的值，来判断代码执行需要跳转到哪个位置。而这种运行期动态决定跳转地址，就很容易导致指令集流水线的中断，造成指令 Cache Miss 的概率增大，从而引起性能下降。


不过在 Java 语言中，因为类方法模式都是抽象的，所以我们可以将关键方法定义成静态方法，从而避免多态调用；对于 C++ 来说，在定义类方法的时候，我们可以根据需求来决定是否需要使用抽象方法，以此减少不必要的多态；而在 C 语言中，我们可以通过尽量避免使用不必要的函数指针，来减少运行期多态。

另外，在实现高性能函数方法的时候，还有一些要点你也要注意，比如尽量避免递归调用、尽量减少不必要的参数传递，等等。不过这些都是高性能编程的常识问题，所以这里我就不展开介绍了。

## 高性能表达式实现

其实，现在的编译器针对表达式级别的优化支持能力已经很强大了，比如说，如果你在编写代码的过程中，使用下面的乘法操作：

```
1 int y = x * 128;
```

 复制代码

那么，对于高性能的编译器来说（如新版的 GCC 9.x 等），就可以将这个乘法操作优化为移位操作，从而提升执行性能。

但是，我们在编码的过程中，并不能完全去依赖这种编译器的能力，因为一方面是编译器的优化能力是有边界的，另一方面在编写代码过程中，编译器对表达式的优化也只是举手之劳。

所以这里，我给你总结了高性能表达式实现中几个比较重要的点，它们都属于简单的实现规则，你也可以在编写代码过程中参考注意下。

### 第一点，尽量将常量计算放到一起。

比如你可以看看下面的代码，这是一个包含了 3 个乘法运算的表达式：

```
1 int z = 32 * x * 432 * y;
```

[复制代码](#)

那么，如果将常量乘法计算放到一起，就很容易在编译期优化掉，从而就可以避免执行时再计算。

### 第二点，尽量将表达式简化，从而减少冗余运算开销。

我们同样来看一个例子。在下面这段代码示例中，两个表达式的实现逻辑是一样的，都是先乘法再加法，但是你会发现，第二个表达式少了一次乘法运算，所以它的执行性能会更加出色：

```
1 int z = x * x + y * x ; //两个乘法操作，一个加法操作
2 int z = x * (x+y); //一个乘法操作，一个加法操作
```

[复制代码](#)

### 第三点，尽量减少除法运算。

目前 CPU 中对除法计算的开销还比较大，因此如果可以优化为移位操作或者乘法操作，那么就都可以改善执行性能。

## 高性能控制流程实现



首先你要知道的是，控制流程代码在执行的过程中，CPU 执行会通过指令分支预测，提前将接下来的执行指令搬移到 Cache 中，如果预测失败，就有可能引起指令流水线中断，从而影响执行性能。

所以，你在编写控制流程代码的时候，就需要考虑一下如何才能更好地实现，以此来优化代码的执行性能。那么具体要如何做呢？

这里，我也给你分享一下我在实践过程中总结出来的经验，即**尽量减少不必要的分支判断**。这个原则是最重要、也是最容易被忽视的。为什么这么说呢？我们来看一个具体的例子。

在下面这段代码中，你可以发现 `x==2` 和 `x==3` 对应的分支场景都是一样的，但是它们也被放到了两个代码分支当中，所以这样执行起来不仅很低效，而且还存在重复代码：

[复制代码](#)

```
1  if ( 2 == x ) {    // 场景1
2      printf("case 1");
3  }
4  if ( 3 == x ) {    //场景1
5      printf("case 1");
6  }
7  if ( 4 == x ) {    //场景2
8      printf("case 2");
9  }
```

在日常的代码开发中，因为偷懒不想写一个组合的逻辑表达式，而增加分支逻辑的现象，其实是比较普遍的。所以说，我们在实际编写控制流程代码的时候，一定要注意尽量减少不必要的代码分支，这样才能有效地提升执行性能。

可是这里你可能还存在一个问题，就是如果深入去挖掘优化代码中的一些重复的分支逻辑，里面包含的门路还比较多。比如说，通过多态来避免代码中重复的 switch 分支逻辑，利用表驱动来减少 switch 逻辑和小的 for 循环平铺执行，等等。

所以这里，我给你一个小建议，就是在一些特殊场景下（比如 if 条件嵌套非常多的场景），你可以考虑使用 switch 来替换 if，这样也有可能改善代码的执行性能。

## 小结

今天这节课，我带你一起探讨了高性能编码的价值观，其实我的核心观念就是，**高性能编码实现需要和后期的代码热点调优一起互相配合**，而不是孤立地去看待其中一个，这样才会更容易开发出高性能的软件。

另外，在明确了高性能实现的价值之后，你还要清楚应该从哪些要点出发，去思考实现高性能编码，以及在高性能编码中针对一些典型业务场景的实现手段。你可以先理解和掌握这节课我给你分享的四种高性能编码实现的方法思路，然后按照这个思路，逐步积累和提  
升高性能编码的能力，从而帮助你最终开发出高性能的代码。

## 思考题

每一种编程语言都在不断发展，那么高性能编码手段是不是也要持续同步地更新呢？欢迎给我留言，分享你的思考和见解。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 11 | 如何针对特定业务场景设计数据结构和高性能算法？

下一篇 13 | 编译期优化：只有修改业务代码才能提升系统性能？

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。