# 内联优化

为了保证程序的执行高效与安全，现代编译器并不会将程序员的代码直接翻译成相应地机器码，它需要做一系列的检查与优化。Go编译器默认做了很多相关工作，例如**未使用的引用包检查、未使用的声明变量检查、有效的括号检查、逃逸分析、内联优化、删除无用代码**等。本文重点讨论内联优化相关内容。

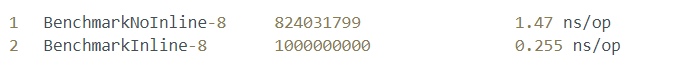
## 内联

在《详解逃逸分析》一文中，我们分析了栈分配内存会比堆分配高效地多，那么，我们就会希望对象能尽可能被分配在栈上。在Go中，一个goroutine会有一个单独的栈，栈又会包含多个栈帧，**栈帧**是函数调用时在栈上为函数所分配的区域。但其实，函数调用是存在一些固定开销的，例如维护**帧指针寄存器BP**、**栈溢出检测**等。因此，对于一些**代码行比较少**的**函数**，编译器倾向于将它们在**编译期展开**从而消除**函数调用**，这种行为就是**内联**。

### ****性能对比****

首先，看一下函数内联与非内联的性能差异。

在程序代码中，想要禁止编译器内联优化很简单，在函数定义前一行添加//go:noinline即可。以下是性能对比结果

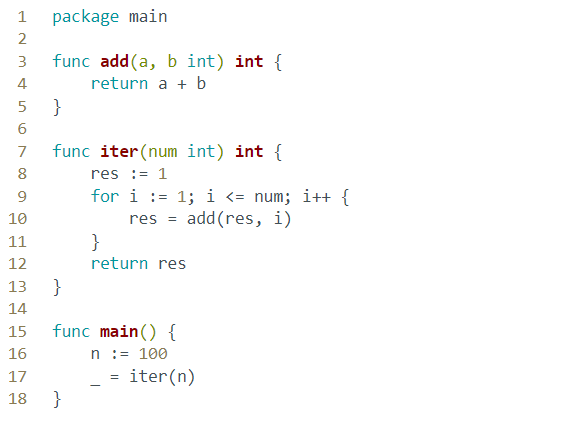


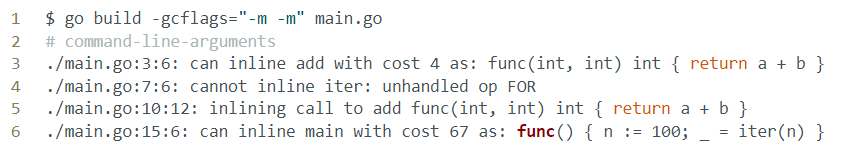
因为函数体内部的执行逻辑非常简单，此时内联与否的性能差异主要体现在函数调用的固定开销上。显而易见，该差异是非常大的。

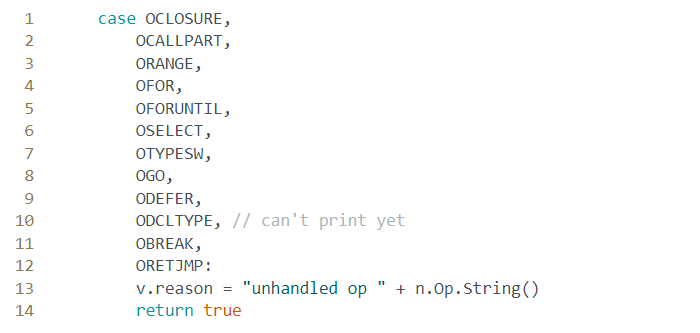
### ****内联场景****

此时，爱思考的读者可能就会产生疑问：既然内联优化效果这么显著，是不是所有的函数调用都可以内联呢？答案是不可以。因为内联，其实就是将一个函数调用原地展开，替换成这个函数的实现。当该函数被多次调用，就会被多次展开，这会增加编译后二进制文件的大小。而非内联函数，只需要保存一份函数体的代码，然后进行调用。所以，在空间上，一般来说使用内联函数会导致生成的可执行文件变大（但需要考虑内联的代码量、调用次数、维护内联关系的开销）。

问题来了，编译器内联优化的选择策略是什么？

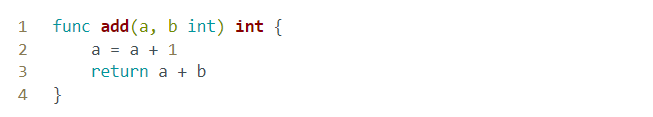
假设源码文件为main.go，可通过执行go build -gcflags="-m -m" main.go命令查看编译器的优化策略。

通过以上信息，可知编译器判断add函数与main函数都可以被内联优化，并将add函数内联。同时可以注意到的是，iter函数由于存在循环语句并不能被内联：cannot inline iter: unhandled op FOR。实际上，除了for循环，还有一些情况不会被内联，例如闭包，select，for，defer，go关键字所开启的新goroutine等，详细可见src/cmd/compile/internal/gc/inl.go相关内容。



在上文提到过，内联只针对小代码量的函数而言，那么到底是小于多少才算是小代码量呢？

此时，我将上面的add函数，更改为如下内容



执行go build -gcflags="-m -m" main.go命令，得到信息

1./main.go:3:6: can inline add with cost 9 as: func(int, int) int { a = a + 1; return a + b }

对比之前的信息

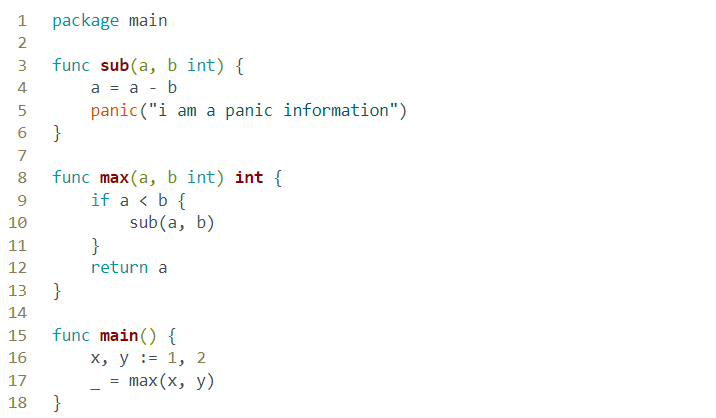
./main.go:3:6: can inline add with cost 4 as: func(int, int) int { return a + b }

可以发现，存在cost 4与cost 9的区别。这里的数值代表的是抽象语法树AST的节点，a = a + 1包含的是5个节点。Go函数中超过80个节点的代码量就不再内联。例如，如果在add中写入16个a = a + 1，则不再内联。

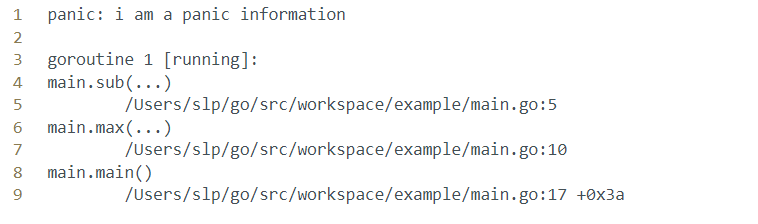
./main.go:3:6: cannot inline add: function too complex: cost 84 exceeds budget 80

### ****内联表****

内联会将函数调用的过程抹掉，这会引入一个新的问题：代码的堆栈信息还能否保证。举个例子，如果程序发生panic，内联之后的程序，还能否准确的打印出堆栈信息？看以下例子。

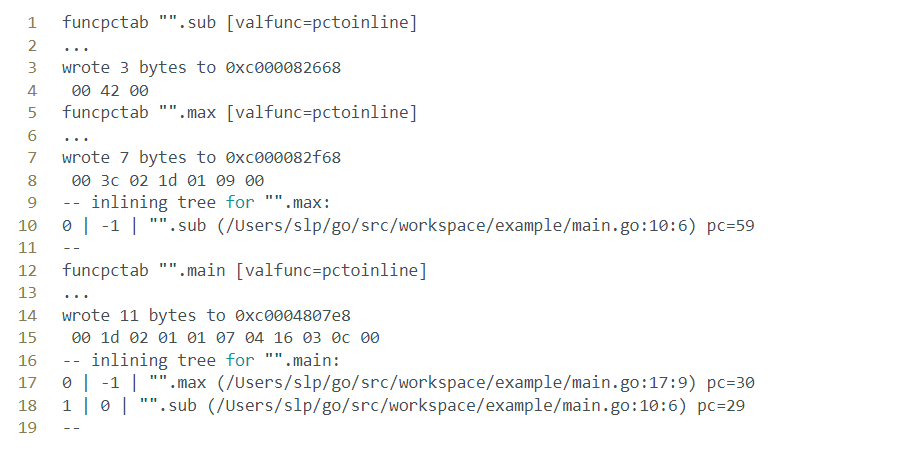


在该代码样例中，max函数将被内联。执行程序，输出结果如下



可以发现，panic依然输出了正确的程序堆栈信息，包括源文件位置和行号信息。那，Go是如何做到的呢？

这是由于Go内部会为每个存在内联优化的goroutine维持一个内联树（inlining tree），该树可通过 go build -gcflags="-d pctab=pctoinline" main.go 命令查看



### 内联控制

Go程序编译时，默认将进行内联优化。我们可通过-gcflags="-l"选项全局禁用内联，与一个-l禁用内联相反，如果传递两个或两个以上的-l则会打开内联，并启用更激进的内联策略。如果不想全局范围内禁止优化，则可以在函数定义时添加 //go:noinline 编译指令来阻止编译器内联函数。