Go1.0 到 1.22 的性能表现，提高了多少倍?

大家好，我是煎鱼。

五一假期时看到 @Ben Hoyt 大佬分享的文章《Go performance from version 1.0 to 1.22》，分享了他在这么多年来一直坚持不懈的对 Go 进行性能测试的记载。

今天基于此分享给大家，有所调整和精简。

原作者对 Go1.0 到 Go1.22 的所有 Go 版本进行了性能测试，包含了在 Go 1.20 中新增的性能分析引导优化（PGO）的结果。

Go 测试项目用的是 [GoAWK](http://cxyroad.com/"https://github.com/benhoyt/goawk") 项目，该项目用 Go 写的 AWK 解释器，支持 CSV。

GoAWK 使用案例如下：

```
$ go install github.com/benhoyt/goawk@latest

$ goawk 'BEGIN { print "foo", 42 }'
foo 42

$ echo 1 2 3 | goawk '{ print $1 + $3 }'
4

# Or use GoAWK's CSV and @"named-field" support:
$ echo -e 'name,amount\nBob,17.50\nJill,20\n"Boba Fett",100.00' | \
  goawk -i csv -H '{ total += @"amount" } END { print total }'
137.5
```

```

性能测试
---

接下来正式开始进行性能测试，主要是同项目多版本测试的模式。

原作者（下称：我）通过在两个 AWK 程序上运行 GoAWK 来测试这一点，这两个程序代表了使用 AWK 可以做的极端不同情况：I/O 操作与字符串处理，以及数值计算。

### countwords（字符串处理任务）

首先是 countwords，这是一个字符串处理任务，它计算输入中单词的频率并打印出带有计数的单词。这是 AWK 脚本的典型应用。输入是某本书的 10 倍串联版本。

以下是部分代码：

```
{
    for (i=1; i<=NF; i++)
        counts[tolower($i)]++
}

END {
    for (k in counts)
        print k, counts[k]
}
```

### sumloop（循环统计任务）

第二个程序是 sumloop，这是一个紧凑的循环，它多次将循环计数器加到一个变量上。

这个程序并不是 AWK 的典型用法，但它是测试 GoAWK 字节码解释器循环的
```

好方法。

以下是部分代码：

```
BEGIN {
    for (i=0; i<10000000; i++)
        sum += i+i+i+i+i+i
}
```

注：我不得不对 GoAWK 的代码做了一些微调，以便它能在旧版本的 Go 上编译。特别是对于 Go 1.0，因为它没有 bufio.Scanner，而 GoAWK 在很大程度上依赖它。我为 1.0 使用了 Go 1.1 的 bufio.Scanner 实现。

性能测试图表

图表中的计时数字是我在 x86-64 Linux 笔记本电脑上运行三次中的最佳时间（以秒为单位）。蓝线代表 countwords 程序，红线代表 sumloop 程序。

Go 各版本的性能测试结果的图表如下：

请注意，这次 Y 轴是对数的，目的是为了更清晰地看到最近版本中更微妙的改进。

图表中还包括了每个 Go 版本下的 GoAWK 二进制文件大小——那是浅灰色线。

一些要点

1、**Go 最大的改进出现在 1.3、1.5、1.7 和 1.12 版本中**。 在此之后的版本，性能提升变得非常渐进 —— 这意味着所有容易实现的优化早已完成。

2、Go 1.2 性能变差了，countwords 出现了一个奇怪的峰值：它从 1.1 的 7.5 秒增加到 1.2 的 25.5 秒（！），然后下降到 1.3 的 2.8 秒。

我通过分析并注意到 Go 运行时栈操作占据了运行时间的很大一部分，找出了 1.2 异常的原因。

以下是 pprof 输出的前几行：

```
$ go tool pprof --text ./goawk_1.2 go12.prof
Total: 1830 samples
     332  18.1%  18.1%      332  18.1% runtime.newstack
     296  16.2%  34.3%      296  16.2% runtime.memclr
     281  15.4%  49.7%      281  15.4% runtime.oldstack
     222  12.1%  61.8%      619  33.8%
github.com/benhoyt/goawk/interp.(*interp).execute
      91   5.0%  66.8%       91   5.0% runtime.lessstack
      75   4.1%  70.9%      133   7.3%
github.com/benhoyt/goawk/interp.(*interp).callBuiltin
      57   3.1%  74.0%       57   3.1% runtime.stackfree
      53   2.9%  76.9%       81   4.4% strings.FieldsFunc
      ...
```

这几乎可以肯定是由于在 1.3 中修复的[堆栈“热分裂”问题](http://cxyroad.com/”https://docs.google.com/document/u/0/d/1wAaf1rYoM4S4gtnPh0zOIGzWtrZFQ5suE8qr2sD8uWQ/mobilebasic?_immersive_translate_auto_translate=1”)所导致的性能下降，因为在该版本 Go 团队将 goroutine 栈的实现从旧的分段模型改为连续模型。

3、**PGO 仅将性能提高了几个百分点**，使用 Go 1.22，countwords 大约提高了 2%，sumloop 提高了 7%。我使用 PGO 编译了发布的 GoAWK 二进制文件。

4、**二进制文件的大小多年来一直保持相对稳定**，除了 1.2 版本中的大幅增

加。即使启用了 PGO，二进制文件的大小也只有大约 5% 的增加，所以我认为通常这是值得的。

结论
--

从测试结果来看，countwords 现在的运行速度比使用 Go 1.0 时快了大约 8 倍，而 sumloop 快了 24 倍。

Go 所编写的程序变快的原因有许多，包含但不限于 Go 团队和外部贡献者改进了编译器，并优化了运行时、垃圾回收器和标准库等。

煎鱼注：不得不说，感谢 Go 核心团队多年来的辛勤工作！让我们躺着升级个版本，就能实现性能的提高。

> 文章持续更新，可以搜【脑子进煎鱼了】阅读，本文 **GitHub** [github.com/eddycjy/blo...](http://cxyroad.com/”https://github.com/eddycjy/blog”) 已收录，学习 Go 语言可以看 [Go 学习地图和路线](http://cxyroad.com/ ”https://github.com/eddycjy/go-developer-roadmap”)，欢迎 Star 催更。

推荐阅读

* [Google 孵化了 3 个 Go 安全库，推荐使用！](http://cxyroad.com/”https://mp.weixin.qq.com/s/lOe036O20y7OW9cSeL-5UQ”)
* [Go 新提案：返回值应该明确使用或忽略？](http://cxyroad.com/”https://mp.weixin.qq.com/s/mnBnwQ_xaFdYAyN48KdTKw”)
* [Go 最大挑战、AI 方向、内部优先级？2024 H1 开发者报告发布](http://cxyroad.com/”https://mp.weixin.qq.com/s/JqDYD5PhtgmNlCWRK7Imsw”)
 原文链接: https://juejin.cn/post/7367404474815692837