对 Go 程序进行可靠的性能测试

Changkun Ou
https://changkun.de/s/gobench/

Go 夜读系列 | *talkgo.org* | Talk Go | 第 83 期
March 26, 2020

主要内容

- 可靠的测试环境
- benchstat
- 例子与实践
  - 例1: 对代码块进行性能调优
  - 例2: Benchmark 的正确性分析
  - 例3: 其他的影响因素
- 假设检验的原理
- 局限与应对措施
- 总结

教科书式的性能测试方法论

在《Software Testing: Principles and Practices》一书中归纳的性能测试方法论：

1. 搜集需求
2. 编写测试用例
3. 自动化性能测试用例
4. 执行性能测试用例
5. 分析性能测试结果
6. 性能调优
7. **性能基准测试（Performance Benchmarking）**
8. 向客户推荐合适的配置

可靠的测试环境

什么是可靠的性能基准测试环境

影响测试环境的软硬件因素

- 硬件：CPU 型号、温度、IO 等
- 软件：操作系统版本、当前系统调度的负载等

指导思想

- 单次测量结果毫无意义，统计意义下可对比的结果是关键
  - 分析测试的场景、多次测量、决定统计检验的类型

- 可对比的结果是在可控的环境下得到的
  - 笔记本电脑 CPU 的执行效率受电源管理等因素影响，连续测试同一段代码可能先得到短暂的性能提升，而后由于温度的上升导致性能下降
  - 虚拟机或（共享）云服务器上可能受到宿主机资源分配等因素导致测量结果不稳定

性能基准测试的两个基本目标

可重复性：在其他外在条件不 变的情况下，性能度量 结果是稳定、可重复的（能复现的才叫 Bug）

可比较性：总是存在一个可以比 较的基本线（有比较才有伤害）

go test -bench

进行性能基准测试的方法

```
func BenchmarkFoo(b *testing.B) {
    for i := 0; i < b.N; i++ {
        foo() // 被测函数
    }
}
```

执行性能基准测试：

```
$ go test -bench=.
```

benchstat

benchstat

benchstat 的功能非常简单，作用只是对性能测试结果进行统计分析，对测量结果进行假设检验，从而消除结果的观测误差（observational error）。

```
$ go get golang.org/x/perf/cmd/benchstat
$ benchstat --help
usage: benchstat [options] old.txt [new.txt] [more.txt ...]
options:
  -alpha α
        设置显著性水平 α 的值（默认 0.05）
  -delta-test test
        设置显著性检验的类型，支持 utest/ttest/none（默认 utest）
  -geomean
        输出几何平均值
  -sort order
        对结果进行排序：[-]delta，[-]name，none（默认值 none）
```

benchstat 的原理：异常值消除+假设检验

当对一个性能基准测试 B 结果反复执行 n 次后，就能得到 b_1，…，b_n 个不同的结果；
在优化代码后，还能得到另外 m 个不同的结果 b_1'，…，b_m'。

一个稳定的基准测试，结果倾向于在某个值附近波动，于是通过通用的计算 **1.5 倍四分位距法则（1.5 x InterQuartile Range Rule）**的方法来消除异常值。

benchstat 的本质就是在消除异常值之后的两组浮点数之间进行**假设检验（Hypothesis Testing）**。

我们之后再来讨论假设检验。

```
type Metrics struct {
    Unit    string    // 性能测试的名称
    Values  []float64 // 某个性能测试的度量值
    RValues []float64 // 移除的异常值
    Min     float64   // RValues 的最小值
    Mean    float64   // RValues 的平均值
    Max     float64   // RValues 的最大值
}
func (m *Metrics) computeStats() {
    values := stats.Sample{Xs: m.Values}
    q1, q3 := values.Percentile(0.25), values.Percentile(0.75)
    lo, hi := q1-1.5*(q3-q1), q3+1.5*(q3-q1) // 计算结果的四分位距，并移除异常值
    for _, value := range m.Values {
        if lo <= value && value <= hi { m.RValues = append(m.RValues, value) }
    }
    // 求统计量
    m.Min, m.Max = stats.Bounds(m.RValues)
    m.Mean = stats.Mean(m.RValues)
}

... // 在 benchstat.Collection.Tables() 中
pval, _ := deltaTest(old, new) // 进行假设检验
```

例子与实践

例 1：sync.Map.Delete 的一个优化

在 sync.Map 中存储一个值，然后再并发的删除该值：

```
func BenchmarkDeleteCollision(b *testing.B) {
    benchMap(b, bench{
        setup: func(_ *testing.B, m mapInterface) { m.LoadOrStore(0, 0) },
        perG: func(b *testing.B, pb *testing.PB, i int, m mapInterface) {
            for ; pb.Next(); i++ { m.Delete(0) }
        },
    })
}
```

优化 src/sync/map.go

```
275 -delete(m.dirty, key)
275 +e, ok = m.dirty[key]
276 +m.missLocked()
```

```
$ git stash
$ go test -run=none -bench=BenchmarkDeleteCollision -count=20 | tee old.txt
$ git stash pop
$ go test -run=none -bench=BenchmarkDeleteCollision -count=20 | tee new.txt
$ benchstat old.txt new.txt
name                                    old time/op  new time/op  delta
DeleteCollision/*sync_test.DeepCopyMap-8  104ns ± 0%   103ns ± 1%     ~     (p=0.383 n=20+20)
DeleteCollision/*sync_test.RWMutexMap-8   67.6ns ± 2%  68.2ns ± 2%  +0.89%  (p=0.009 n=20+20)
DeleteCollision/*sync.Map-8               94.2ns ± 2%   5.7ns ± 2%  -93.98%  (p=0.000 n=20+19)
```

有一个样本被作为异常值剔除了

例 2：测试代码错误

创建一颗红黑树，并依次将 0 … n 插入到这颗红黑树中：

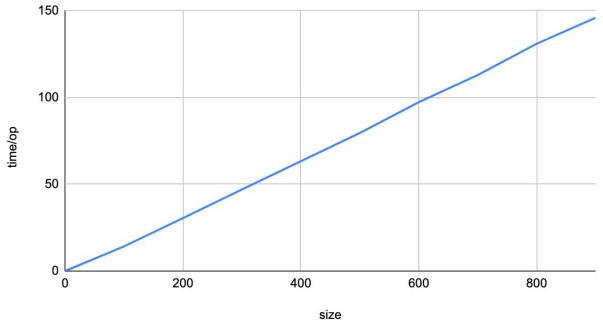
```
func BenchmarkRBTree_PutWrong(b *testing.B) {
    for size := 0; size < 1000; size += 100 {
        b.Run(fmt.Sprintf("size-%d", size), func(b *testing.B) {
            tree := ds.NewRBTree(func(a, b interface{}) bool {
                if a.(int) < b.(int) { return true }
                return false
            })


            for i := 0; i < b.N; i++ {
                for n := 0; n < size; n++ { tree.Put(n, n) }
            }
        })
    }
}
```

为什么插入的性能是线性的？红黑树的插入性能不是 O(log(n)) 吗？

代码写错了……吧……？

```
name                          time/op
RBTree_PutWrong/size-0-8      0.65ns ± 0%
RBTree_PutWrong/size-100-8    14.2µs ± 3%
RBTree_PutWrong/size-200-8    30.5µs ± 0%
RBTree_PutWrong/size-300-8    47.0µs ± 0%
RBTree_PutWrong/size-400-8    63.3µs ± 0%
RBTree_PutWrong/size-500-8    79.6µs ± 0%
RBTree_PutWrong/size-600-8    97.3µs ± 0%
RBTree_PutWrong/size-700-8     113µs ± 0%
RBTree_PutWrong/size-800-8     131µs ± 0%
RBTree_PutWrong/size-900-8     146µs ± 0%
```

RBTree.Put 性能（错误形式）



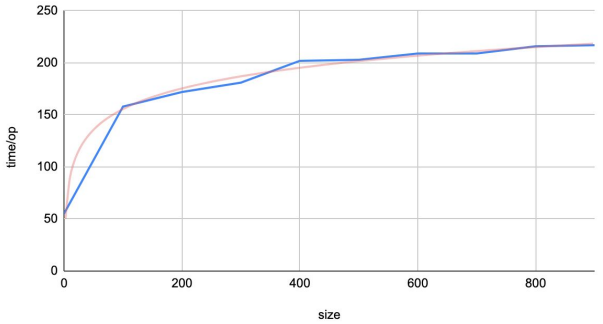
例 2:测试代码错误(续)

红黑树的插入性能是指当树的大小为 n-1 插入第 n 个值时的性能:

```
func BenchmarkRBTree_Put(b *testing.B) {
    for size := 0; size < 1000; size += 100 {
        b.Run(fmt.Sprintf("size-%d", size), func(b *testing.B) {
            tree := ds.NewRBTree(func(a, b interface{}) bool {
                if a.(int) < b.(int) { return true }
                return false
            })
+           for n := 0; n < size-1; n++ { tree.Put(n, n) }
+           b.ResetTimer()
            for i := 0; i < b.N; i++ {
                tree.Put(n, n)
            }
        })
    }
}
```

```
name                       time/op
RBTree_Put/size-0-8      55.2ns ± 0%
RBTree_Put/size-100-8     158ns ± 0%
RBTree_Put/size-200-8     172ns ± 0%
RBTree_Put/size-300-8     181ns ± 0%
RBTree_Put/size-400-8     202ns ± 0%
RBTree_Put/size-500-8     203ns ± 0%
RBTree_Put/size-600-8     209ns ± 0%
RBTree_Put/size-700-8     209ns ± 0%
RBTree_Put/size-800-8     216ns ± 0%
RBTree_Put/size-900-8     217ns ± 0%
```

RBTree.Put 性能（正确形式）



例 3:编译器优化

编译器优化产生的直接影响是测量的目标不准确，这一点在 C++ 编译器中相当严重。编译器优化是一个比较大的话题，有很多可以深入讨论的内容，以后有机会再表。只举比较简单的一例。**comp1 和 comp2 一样快吗？**

```go
func comp1(s1, s2 []byte) bool {
    return string(s1) == string(s2)
}

func comp2(s1, s2 []byte) bool {
    return conv(s1) == conv(s2)
}

func conv(s []byte) string {
    return string(s)
}
```

假设检验的原理

（相当不严谨地）回顾

- 总体：所有满足某些共同性质的值的集合（共同性质：接口）
- 样本：从总体中随机抽取的个体
- 频率：n 次试验中，某个事件发生的次数除以总的试验次数
- 大数定理：当试验次数 $n \to \infty$ 时，频率一定收敛到某个值
- 概率：频率收敛到的值，性质之一：$0 \leq P(A) \leq 1$
- 独立：两个事件互不影响，性质之一：$P(AB) = P(A)P(B)$
- 随机变量：是一个函数，参数是所有可能的样本，返回值是这些样本的取值，例如 $P(X = 2) = 0.25$
- 期望：随机变量以其概率为权重的加权平均值，即 $E(X) = \sum_i x_i p_i$
- 方差：样本取值与期望之间的「距离」，距离定义为差的平方和，即 $Var(X) = \sum_i (x_i - E(X))^2$
- 概率密度函数：是一个函数，参数是随机变量取值，返回值是随机变量取得该值的概率
- 累积分布函数：随机变量取值小于某个值的概率
- 正态分布：一种特殊的概率密度函数 $N(\mu, \sigma^2)$
- 中心极限定理：**无穷多个独立的随机变量的和服从正态分布**

* 额外的说明见演讲者备注

检验的类型

- 统计是一套在总体分布函数完全未知或者只知道形式、不知参数的情况下，为了由样本推断总体的某些未知特性，形成的一套方法论。
- 多次抽样：对同一个性能基准测试运行多次，根据中心极限定理，如果理论均值存在，则抽样噪声服从正态分布的。
- 当重复执行完某个性能基准测试后，benchstat 先帮我们剔除掉了一些异常值，我们得到了关于某段代码在可控的环境条件 E 下的性能分布的一组样本。

- 现在的问题是：
  - **非参数方法**：样本是否来自同一总体？总体是什么分布？两组样本在可控的测试环境下进行吗？
  - **参数方法**：如果总体分布已经确定，那么样本的变化是否显著？性能的基准测试前后，是否具有统计意义下的明显变化？

假设检验

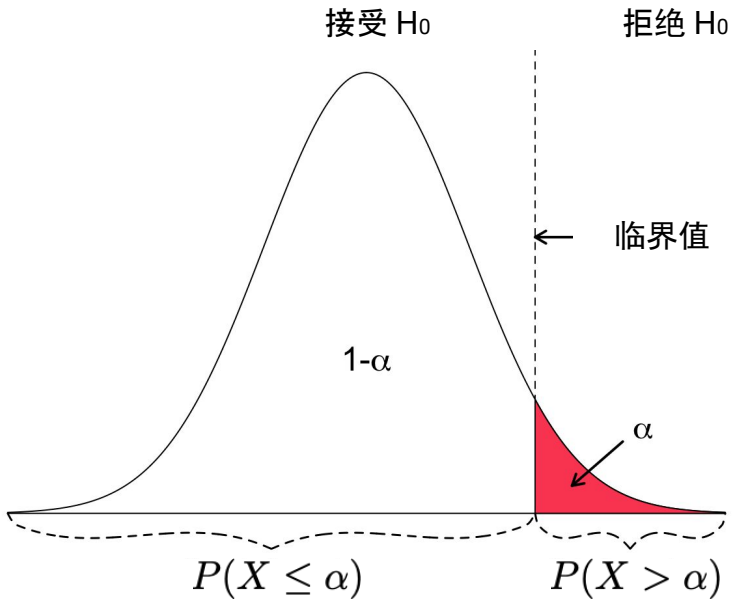
假设检验：利用样本判断对总体的假设是否成立的过程

零假设 **H_0**：想要驳回的论点

备择假设 **H_1**：拒绝零假设后的备用项，我们想要证明的论点

p 值：零假设发生的概率

显著性水平：可靠程度



例如：在性能基准测试中，

H_0：代码修改前后，性能没有提升

H_1：代码修改前后，性能有显著提升

`p < 0.05`：**H_0** 发生的概率小于 5%，在至少 95% 的把握下，性能有显著提升

假设检验的决策错误

假设检验的决策错误		
真实情况	**决策**	
	接受零假设	拒绝零假设
零假设为真	正确	犯第一类错误
零假设为假	犯第二类错误	正确

- 第一类错误：把对的判断成错的；第二类错误：把错的判断成对的
- 当样本不变时，减少犯某类错误的概率择会增加犯另一类错误的概率。控制第一类错误的概率，让它小于某个 p 值（0.05）称之为显著性检验
- 零假设　**H_0: 代码性能测试的均值没有显著变化** $\mu_0 - \mu_1 = 0$
- 备择假设 **H_1: 代码性能有显著变化** $\mu_0 - \mu_1 \neq 0$
    - 对性能提升持有保守态度，尽可能避免出现实际没有提升，但被判断为提升（第一类错误）

T 检验、Welch T 检验和 Mann-Whitney U 检验

两个总体均值差的检验 $H_0 : \mu_1 - \mu_2 = 0, H_1 : \mu_1 - \mu_2 \neq 0$

T 检验

参数检验，假设数据服从正态分布，且方差相同

Welch T 检验

参数检验，假设服从正态分布，方差一定不相同

Mann-Whitney U 检验

非参数检验，假设最少，最通用，只假设两组样本来自同一总体，只有均值上的差异（保守派）

当对数据的假设减少时，结论的不确定性就会增大，因此 p 值会相应的变大，进而使性能基准测试的条件更加严格。

局限与应对措施

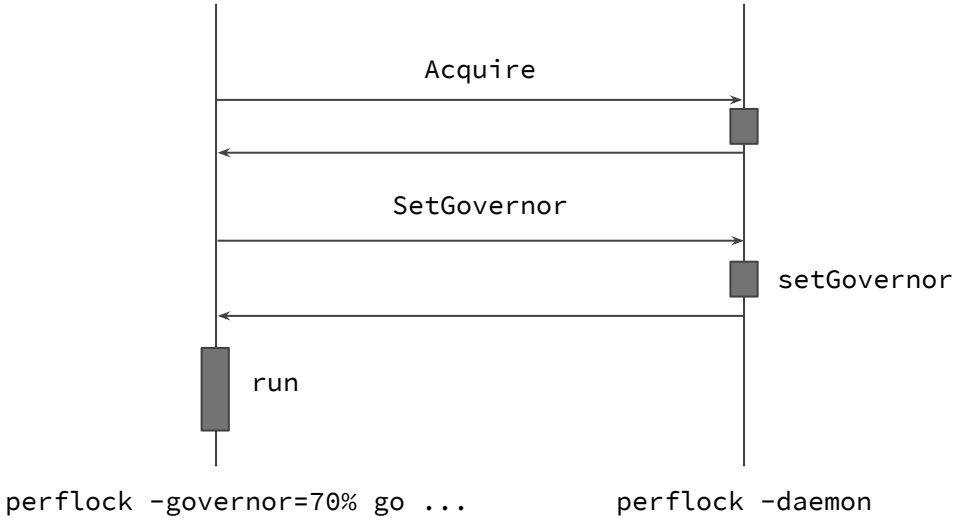
降低系统噪音：perflock

作用是限制 CPU 时钟频率，从而一定程度上消除系 统对性能测试程序的影响，减少结果的噪声，进而性能测量的结果方差更小也更加可靠，仅支持 Linux。

```
$ go get github.com/aclements/perflock/cmd/perflock
$ sudo install $GOPATH/bin/perflock /usr/bin/perflock
$ sudo -b perflock -daemon
$ perflock
Usage of perflock:
  perflock [flags] command...
  perflock -list
  perflock -daemon
  -daemon
        启动 perflock 守护进程
  -governor percent
        设置运行指令所占用的 CPU 频率比例，或 none 没有调整（默认 90%）
  -list
        列出当前正在等待执行的命令
  -shared
        在共享模式下获取锁（默认独占模式）

$ perflock -governor 70% go test -test=none -bench=.
```

perflock 的原理

在执行命令前，通知 perflock 守护进程，守护进程将 cpufreq 进行备份，调整到 perflock-client 指定的频率，再通知 perflock-client 开始执行 Benchmark



进行如下修改：

```
/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq
                                ==
/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq
                                ==
        target := (max-min)*percent/100 + min
```

使用 perflock 的注意事项

- **不要**在执行性能测试时强制 kill perflock daemon，否则 cpufreq 参数将不会恢复
- 只锁定了系统的 CPU 频率，并没有限制与系统中其他资源的占用情况，该被打断的依然会被打断
- **使用前检查 cpufreq 是否能够被修改**，不能在虚拟机上使用、不能在容器内使用
  - 无法获取 /sys/devices/system/cpu/cpu*/cpufreq
- ...

多重比较谬误

Source: https://xkcd.com/882/

多重比较谬误

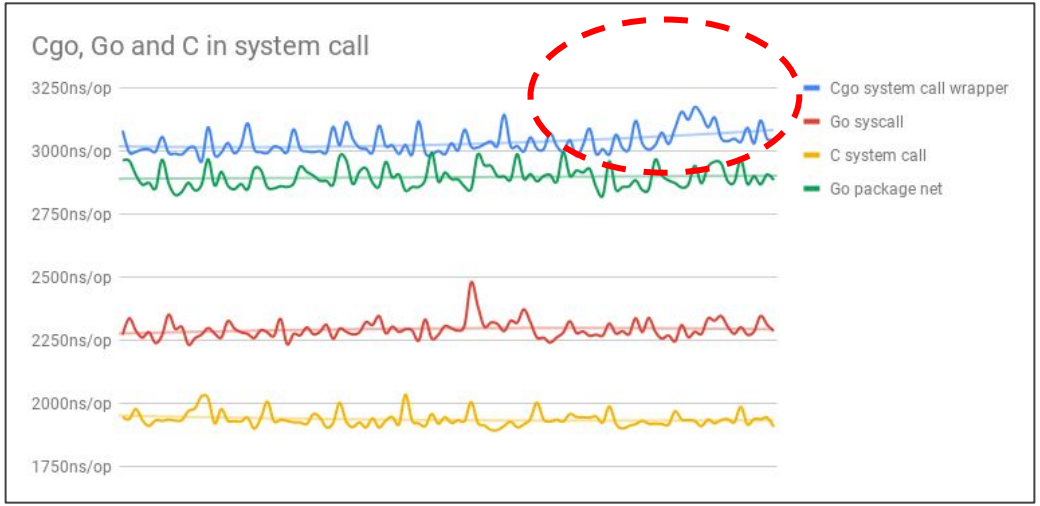
广泛比较两个不同群体的所有差异，从中找出具有差异的特征，宣称是造成两个群体不同的原因。

⇒

在不对代码进行优化的情况下，反复 对不同的性能测试结果样本进行显著性检验，直到找到能 够使 p 值能够满足显著性水平，宣称性能得到了提升。

机器过热

对结果进行回归，肉眼可见的性能下降



https://github.com/changkun/cgo-benchmarks/tree/master/syscall

总结

进行(严肃的)性能测试前的检查清单

- 限制系统资源，降低测试噪声：perflock
  - 限制 CPU 时钟频率：perflock
  - (如果需要)限制 runtime 消耗的内存上限: runtime.SetMaxHeap
  - 关闭无关程序和进程等等……
- 确定测试代码的正确性
  - 考虑 Goroutine 的终止性，当某些并发的工作发生在基准测试结束后，那么测量是不准确的
  - 考虑编译器进行了过度优化或基准测试代码本身编写错误导致测量程序不正确
- 实施性能基准测试
  - (如果需要)计算需要采样的次数，否则推荐 20 次
  - 使用 git stash 记录并撤销代码的修改，执行测试得到修改前的性能测试结果
  - 使用 git stash pop 恢复代码的修改内容，执行测试得到修改后的性能测试结果
  - 使用 benchstat 对前后测量到的性能测量进行假设检验
  - 验证结果有效性，例如确认结果的波动，比较随时间推移造成的性能回归等等

进一步阅读的参考文献

- https://dave.cheney.net/high-performance-go-workshop/dotgo-paris.html
  - 这是一篇很早之前的关于Go程序性能测试的文章，里面讲述了相当多有关性能调优、测试的主题，不仅局限于这次分享的主题
- https://github.com/golang/go/issues/27400
  - 这是一个未解决的Issue，目的是希望 Go 团队能够在 testing 包中使用文档来说明编译器优化的情况，进而避免基准测试测量不准确的问题
- https://github.com/golang/go/issues/23471
  - 这是一个未解决的Issue，目的是希望 Go 团队能够发布一篇官方文档来详述如何科学的对 Go 程序进行性能测试
  - 当然，本次分享的PPT 其实解决了这个问题 :)
- A Review and Comparison of Methods for Detecting Outliers in Univariate Data Sets, http://d-scholarship.pitt.edu/7948/1/Seo.pdf
  - 这篇论文比较了统计学中的一些异常值检测的方法
- Mann, Henry B., and Donald R. Whitney. "On a test of whether one of two random variables is stochastically larger than the other." *The annals of mathematical statistics* (1947): 50-60.
  - 这是 Mann-Whitney U 检验的原始论文
- Mytkowicz, Todd, et al. "Producing wrong data without doing anything obviously wrong!." *ACM Sigplan Notices* 44.3 (2009): 265-276.
  - 这篇文章介绍了适用因果分析和随机化的方法来检测并避免测量误差

提问、交流链接：

https://github.com/talk-go/night/issues/564

Go 夜读 微信公众号

Q & A

一些小故事

如果你没有办法解释莫名其妙带来的性能提升……

```
https://github.com/golang/go/commit/a479a455489bc3600c004367f16c4d452705d2c9

Russ Cox committed on Nov 17, 2011

json.BenchmarkSkipValue              24630036     18557062   -24.66%

I cannot explain why BenchmarkSkipValue gets faster.
Maybe it is one of those code alignment things.
```

可能的解释：当代码的顺序被调整后，在二进制的表现形式可能发生变化，进而在缓存中的存储形式也可能发生变化，也有可能影响缓存的局部性，从而得到莫名其妙的性能提升……

正态分布的由来

本科的概率论通常会直接给出正态分布的定义，然后讲授中心极限定理。但实际上早年数学见是先研究出中心极限定理，而后发现正态分布的形式在后续研究中非常常见，就将其称之为正态分布。

更复杂的建模

机器过热问题可以通过更复杂的数学建模来解决(例如 MCMC)，右边的图中虽然存在机器过热问题，但实际上我们对数据的直观感受是，Go syscall 是显著慢于 C syscall 的

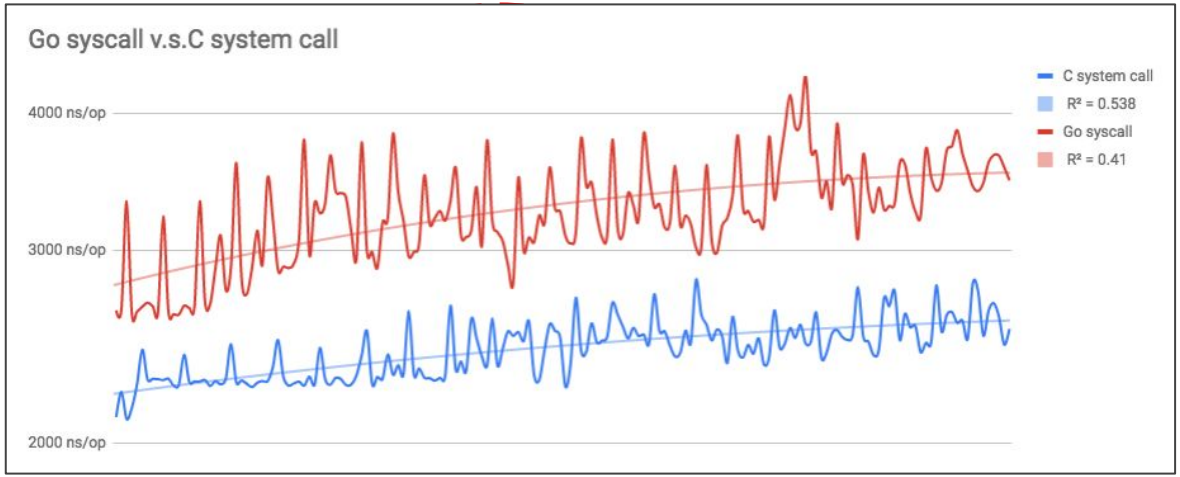
但实践中这种复杂的建模对性能测试有指导意义吗？也许没有

⇒

找一个稳定的机器比花哨的数学更重要

不过，倒是可以考虑如何自动检测样本数据中的这种现象来避免新手犯错。

对结果进行回归，肉眼可见的性能下降



https://github.com/changkun/cgo-benchmarks/tree/master/syscall