Golang程序测试与性能优化

杨辰@adxmi

2016-12-16

Golang 代码质量管理

- 单元测试文件: *_test.go
  - go test -run regexp
  - table-driven tests
- 覆盖率测试: go test -cover
- 可测试的例子: func Example*()
  - godoc文档相关
  - 确保能够正常编译
- godoc文档
- 格式化及检查工具: gofmt -s, golint, go vet, ...

理想的项目状态:

性能测试

```
func BenchmarkItoa(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = strconv.Itoa(i) // op
    }
}
```

```
> go test -run $^ -bench Itoa -benchmem -cpu 2
BenchmarkItoa-2  20000000  66.8 ns/op  7 B/op  1 allocs/op
```

- 理解结果
- 小心编译器优化及初始化开销影响结果
- 高并发测试: -parallel n

profile

- “runtime/pprof”
- “net/http/pprof”
- go tool pprof
- 定位热点调用路径和瓶颈, 针对性优化
- 例子

GC 机制

GOGC 环境变量:

- 当新分配的内存大小/ 上次GC存活的内存大小超过这个阈值时, 触发GC
- 默认100: 内存占用翻倍时, 触发GC
- 权衡: 内存占用和GC时间
- runtime.ReadMemStats(m *MemStats) 提取GC信息
- 优化: 面向GC编程

堆分配，栈分配？

- 栈上分配优点？
- 编译器会做逃逸分析，优先栈上创建对象
  - `var T` 不一定在栈上
  - `new(T)` 不一定在堆上

array vs slice

- array ~ [2]uintptr 地址+ 长度
- slice ~ [3]uintptr 底层array信息+ 当前位置
- 理解扩容降容机制: runtime/slice.go
- 常见错误: make([]T, N) != make([]T, 0, N)
- 优化: 预留足够长度, 避免拷贝
- 优化: copy(dst, src) vs dst = append(dst, src...)

string vs []byte

- string: immutable byte array
- string 和[]byte 互转发生拷贝操作
- 避免string 连接, 避免[]byte 和string 转换
- 编译器会做一定的优化
- unsafe.Pointer trick

struct

- 和C语言struct类似
- struct{} 什么鬼?
  - 长度为0的对象均指向runtime.zerobase变量
- 要知道内存对齐
- 例子

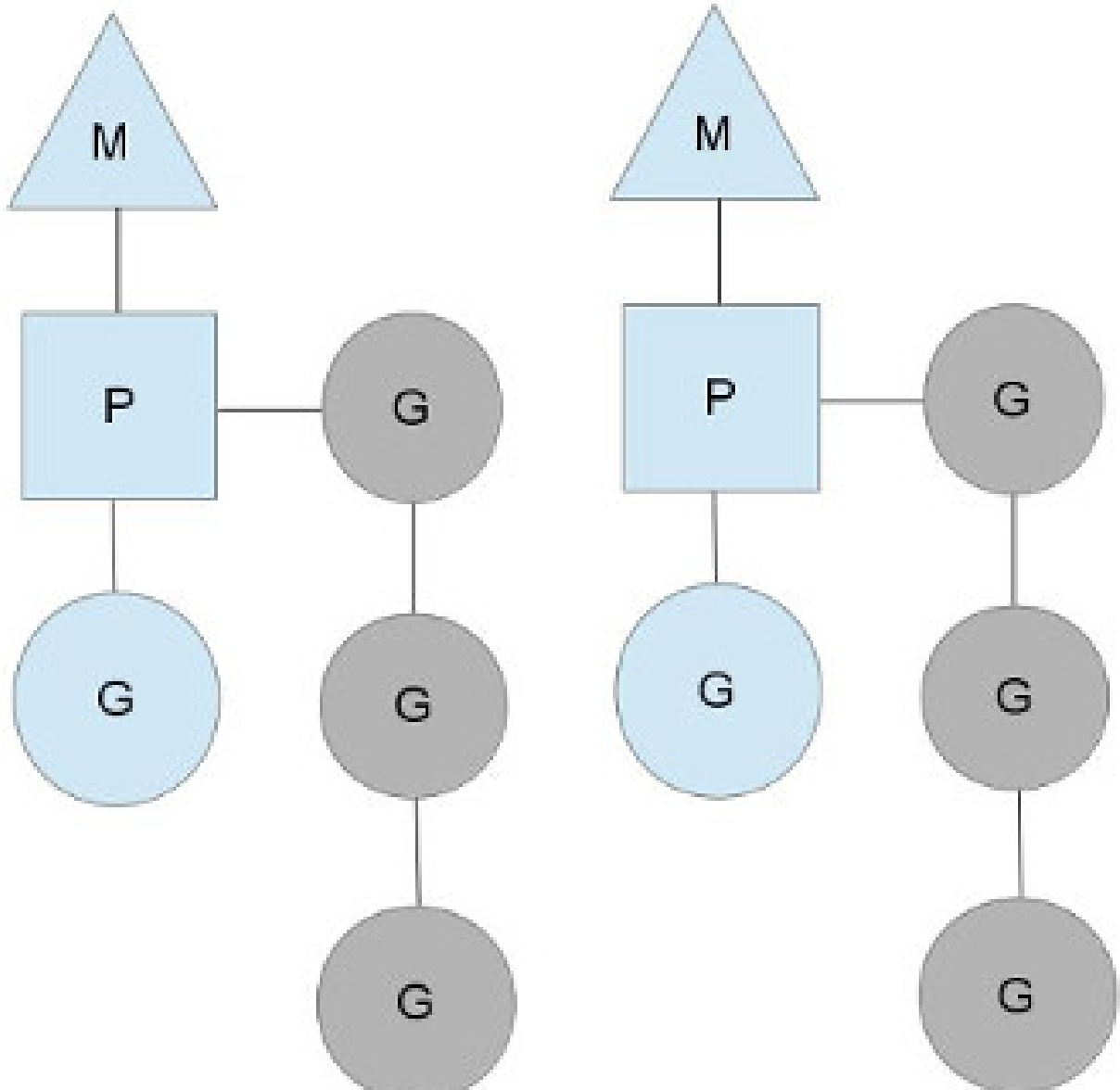
map

- 倍增扩容, 不会降容
- 扩容后, “懒拷贝”以降低延迟
- 优先选择int作为键值, string次之, 有针对性优化
- 用map[K]struct{}作为集合结构
- 创建时预留足够大小, 避免拷贝
- 小数据(<128)直接在map对象里存储

存值还是存指针?

- 对象指针/引用语义优点:
  - 作为参数传递时只拷贝地址, 开销小
  - 直接修改内容
  - 一处修改, 所有引用地方均知晓, 数据信息传递的桥梁
- 非指针对象/值语义优点:
  - 有利于内存局部性
  - 对于slice, channel, map类型, GC不扫描非指针类型成员, 减少GC负担
  - 减少堆内存分配, 降低GC对象引用维护负担
  - immutable object 有利于并发编程
- 推荐用noCopy标记不可拷贝对象

理解goroutine 调度



- G: goroutine, 非执行体, 仅保存并发任务的上下文信息
- P: Processor / Context
  - GOMAXPROCS 控制并发度
  - 有私有分配内存空间, G队列, 以避免争用
- M: 系统线程
  - 不定数目
  - 必须独占一个P, 无锁执行, 消费G工作队列, 实现跨线程调度
  - block时(如系统调用) 被换出
- 每个P一个G队列, 全局一个G队列
- 优化: prefer fixed size worker pool to goroutine
  - 可以保证全部消费完毕
  - 避免过载

defer

- defer 在函数返回前一次调用
  - 可以修改返回值
- 注意变量绑定, 不要取到错误的值
- 优化: 避免defer
  - 可读性vs 性能
- 优化: 整合多个defer操作到一起

channel

- 并发安全的有界队列
- 注意: 生产者端避免堵塞, 消费者端要消费干净再推出
- select会对所有channel加锁
- chan T to chan []T 批量读写优化
- 避免滥用channel, 不要忘记同步原语, 原子操作

对象池(object pool)

- sync.Pool
  - 并非全局, 对于每个P有本地pool
- channel 用作对象池(leaky buffer)
- 评估锁的负面影响

避免使用reflect

优化思路:

- 生成定制化编解码函数
  - e.g. ffjson
- 缓存类型信息
  - e.g. github.com/gorilla/schema

接口“短路”优化

“短路”: 基于接口间调用关系的实现的特化版本

例子:

- prefer `io.WriteString(w, s)` to `w.Write([]byte(s))`
  - 如有可能, 实现io.stringWriter
- io.Copy
  - io.ReaderFrom interface
  - io.WriterTo interface
- cipher.cbcEncAble

提供不需要申请内存对象的接口

- 使用面向[]byte, io.Writer, . . . 的方法签名, 避免需要创建中间变量的实现
- e.g.
  - strconv.Append*(dst []byte, . . .) []byte
  - fmt.Fprintf(w io.Writer, . . .)
- copy 方法针对append([]byte, “string”. . .) 有优化

其他优化

- 减少系统调用
  - 取时间优化
    - 避免频繁调用time.Now
    - 低精度场合全局clock
- 发现并改写热点路径函数
  - e.g. beego/logs/logger.go formatTimeHeader

Go调用汇编和C(CGO)

- 调用汇编目的: 使用硬件指令集优化
- 例子: src/crypto
- CGO方式调用现有C库, 或者用C重写部分模块
  - 理解CGO调用开销

case study: fasthttp vs net/http

- 都使用了sync.Pool

fasthttp 优化:

- 丧心病狂地用[]byte, 拒绝string
- 加读写缓冲区, 避免直接读写net.Conn
- “懒”, Header等相关请求信息需要时再解析
- 使用workerpool处理请求, 不单独起goroutine

参考资料

- go test 相关
  - https://golang.org/pkg/testing/
  - https://blog.golang.org/cover
  - https://blog.golang.org/examples
  - https://blog.golang.org/subtests
- GC相关
  - Go GC: Prioritizing low latency and simplicity
  - Go GC: Latency Problem Solved
  - Go 1.4+ Garbage Collection (GC) Plan and Roadmap
  - Go 1.5 concurrent garbage collector pacing

- http://www.zenlife.tk/tags?name=golang
- https://dave.cheney.net/category/golang
- http://jmoiron.net/blog/go-performance-tales/
- https://golang.org/doc/effective_go.html
- https://morsmachine.dk/go-scheduler
- https://github.com/golang/go/wiki/CompilerOptimizations

写在最后的

- 架构设计优于算法
- 算法优于语言层面优化
- 追求清晰简短的代码
- 日常写代码要有性能意识
- 先性能测试，定位瓶颈，避免过度优化