**项目背景**

网关服务作为统一接入服务，是大部分服务的统一入口。为了避免成功瓶颈，需要对其进行尽可能地优化。因此，特别总结一下 golang 后台服务性能优化的方式，并对网关服务进行优化。

技术背景：

* 基于 tarsgo 框架的 http 接入服务，下游服务使用 tarsgo 协议进行交互

**性能指标**

网关服务本身没有业务逻辑处理，仅作为统一入口进行请求转发，因此我们主要关注下列指标

* 吞吐量：每秒钟可以处理的请求数
* 响应时间：从客户端发出请求，到收到回包的总耗时

**定位瓶颈**

一般后台服务的瓶颈主要为 CPU，内存，IO 操作中的一个或多个。若这三者的负载都不高，但系统吞吐量低，基本就是代码逻辑出问题了。

在代码正常运行的情况下，我们要针对某个方面的高负载进行优化，才能提高系统的性能。golang 可通过 benchmark 加 pprof 来定位具体的性能瓶颈。

**benchmark 简介**

go test -v gate\_test.go -run=none -bench=. -benchtime=3s -cpuprofile cpu.prof -memprofile mem.prof

* -run 知道单次测试，一般用于代码逻辑验证
* -bench=. 执行所有 Benchmark，也可以通过用例函数名来指定部分测试用例
* -benchtime 指定测试执行时长
* -cpuprofile 输出 cpu 的 pprof 信息文件
* -memprofile 输出 heap 的 pprof 信息文件。
* -blockprofile 阻塞分析，记录 goroutine 阻塞等待同步（包括定时器通道）的位置
* -mutexprofile 互斥锁分析，报告互斥锁的竞争情况

**benchmark 测试用例常用函数**

* b.ReportAllocs() 输出单次循环使用的内存数量和对象 allocs 信息
* b.RunParallel() 使用协程并发测试
* b.SetBytes(n int64) 设置单次循环使用的内存数量

**pprof 简介**

**生成方式**

* **runtime/pprof**: 手动调用如runtime.StartCPUProfile或者runtime.StopCPUProfile等 API 来生成和写入采样文件，灵活性高。主要用于本地测试。
* **net/http/pprof**: 通过 http 服务获取 Profile 采样文件，简单易用，适用于对应用程序的整体监控。通过 runtime/pprof 实现。主要用于服务器端测试。
* **go test**: 通过 go test -bench . -cpuprofile cpuprofile.out生成采样文件，主要用于本地基准测试。可用于重点测试某些函数。

**查看方式**

* go tool pprof [options][binary] ...
  + --text 纯文本
  + --web 生成 svg 并用浏览器打开（如果 svg 的默认打开方式是浏览器)
  + --svg 只生成 svg
  + --list funcname 筛选出正则匹配 funcname 的函数的信息
  + **-http=":port" 直接本地浏览器打开 profile 查看（包括 top，graph，火焰图等）**
* go tool pprof -base profile1 profile2
  + 对比查看 2 个 profile，一般用于代码修改前后对比，定位差异点。
* 通过命令行方式查看 profile 时，可以在命令行对话中，使用下列命令，查看相关信息
  + flat flat% sum% cum cum%  
    5.95s 27.56% 27.56% 5.95s 27.56% runtime.usleep4.97s 23.02% 50.58% 5.08s 23.53% sync.(\*RWMutex).RLock4.46s 20.66% 71.24% 4.46s 20.66% sync.(\*RWMutex).RUnlock2.69s 12.46% 83.70% 2.69s 12.46% runtime.pthread\_cond\_wait1.50s 6.95% 90.64% 1.50s 6.95% runtime.pthread\_cond\_signal
  + flat: 采样时，该函数正在运行的次数\*采样频率(10ms)，即得到估算的函数运行”采样时间”。这里不包括函数等待子函数返回。
  + flat%: flat / 总采样时间值
  + sum%: 前面所有行的 flat% 的累加值，如第三行 sum% = 71.24% = 27.56% + 50.58%
  + cum: 采样时，该函数出现在调用堆栈的采样时间，包括函数等待子函数返回。因此 flat <= cum
  + cum%: cum / 总采样时间值
  + topN [-cum] 查看前 N 个数据：
* list ncname 查看某个函数的详细信息，可以明确具体的资源（cpu，内存等）是由哪一行触发的。

**pprof 接入 tarsgo**

1. 服务中 main 方法插入代码cfg := tars.GetServerConfig()  
   profMux := &tars.TarsHttpMux{}profMux.HandleFunc("/debug/pprof/", pprof.Index)profMux.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)profMux.HandleFunc("/debug/pprof/profile", pprof.Profile)profMux.HandleFunc("/debug/pprof/symbol", pprof.Symbol)profMux.HandleFunc("/debug/pprof/trace", pprof.Trace)tars.AddHttpServant(profMux, cfg.App+"."+cfg.Server+".ProfObj")
2. taf 管理平台中，添加 servant：ProfObj （名字可自己修改)
3. 发布服务

**查看 tasrgo 服务的 pprof**

1. 保证开发机能直接访问到 tarsgo 节点部署的 ip 和 port。
2. 查看 profile(http 地址中的 ip,port 为 ProfObj 的 ip 和 port)# 下载cpu profile  
   go tool pprof http://ip:port/debug/pprof/profile?seconds=120 # 等待120s，不带此参数时等待30s# 下载heap profilego tool pprof http://ip:port/debug/pprof/heap# 下载goroutine profilego tool pprof http://ip:port/debug/pprof/goroutine# 下载block profilego tool pprof http://ip:port/debug/pprof/block# 下载mutex profilego tool pprof http://ip:port/debug/pprof/mutex# 下载20秒的trace记录（遇到棘手问题时，查看trace会比较容易定位)curl http://100.97.1.35:10078/debug/pprof/trace?seconds=20 > trace.outgo tool trace trace.out 查看
3. 直接在终端中通过 pprof 命令查看
4. sz 上面命令执行时出现的Saved profile in /root/pprof/pprof.binary.alloc\_objects.xxxxxxx.xxxx.pb.gz到本地
5. 在本地环境，执行go tool pprof -http=":8081" pprof.binary.alloc\_objects.xxxxxxx.xxxx.pb.gz 即可直接通过http://localhost:8081页面查看。包括topN，火焰图信息等,会更方便一点。

**GC Trace**

golang 具备 GC 功能，而 GC 是最容易被忽视的性能影响因素。尤其是在本地使用 benchmark 测试时，由于时间较短，占用内存较少。往往不会触发 GC。而一旦线上出现 GC 问题，又不太好定位。目前常用的定位方式有两种：

**本地 gctrace**

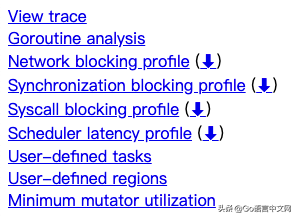
* 在执行程序前加 GODEBUG=gctrace=1，每次 gc 时会输出一行如下内容gc 1 @0.001s 11%: 0.007+1.5+0.004 ms clock, 0.089+1.5/2.8/0.27+0.054 ms cpu, 4->4->3 MB, 5 MB goal, 12 P  
  scvg: inuse: 4, idle: 57, sys: 62, released: 57, consumed: 4 (MB)
  + 也通过日志转为图形化：GODEBUG=gctrace=1 godoc -index -http=:6060 2> stderr.log  
    cat stderr.log | gcvis
  + inuse：使用了多少 M 内存
  + idle：剩下要清除的内存
  + sys：系统映射的内存
  + released：释放的系统内存
  + consumed：申请的系统内存
  + gc 1 表示第 1 次 gc
  + @0.001s 表示程序执行的总时间
  + 11% 表示垃圾回收时间占用总的运行时间百分比
  + 0.007+1.5+0.004 ms clock 表示工作线程完成 GC 的 stop-the-world,sweeping,marking 和 waiting 的时间
  + 0.089+1.5/2.8/0.27+0.054 ms cpu 垃圾回收占用 cpu 时间
  + 4->4->3 MB 表示堆的大小，gc 后堆的大小，存活堆的大小
  + 5 MB goal 整体堆的大小
  + 12 P 使用的处理器数量
  + scvg: inuse: 4, idle: 57, sys: 62, released: 57, consumed: 4 (MB) 表示系统内存回收信息
  + 采用图形化的方式查看：https://github.com/davecheney/gcvisGODEBUG=gctrace=1 go test -v \*.go -bench=. -run=none -benchtime 3m |& gcvis

**线上 trace**

在线上业务中添加**net/http/pprof**后，可通过下列命令采集 20 秒的 trace 信息

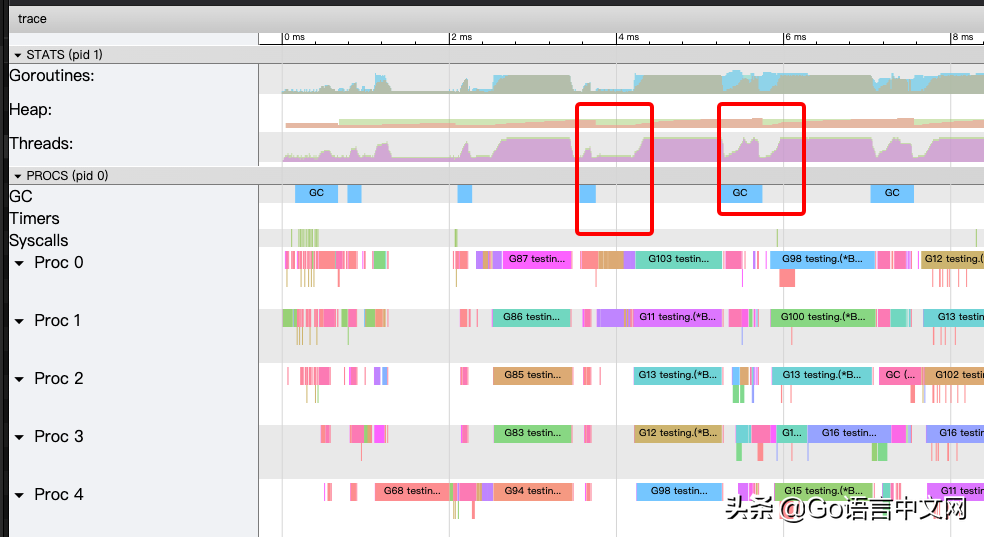
curl http://ip:port/debug/pprof/trace?seconds=20 > trace.out

再通过go tool trace trace.out 即可在本地浏览器中查看 trace 信息。

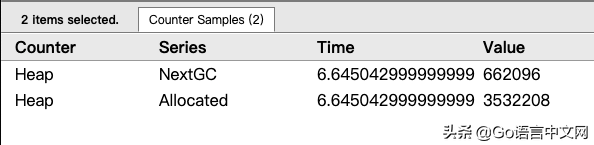


* View trace：查看跟踪
* Goroutine analysis：Goroutine 分析
* Network blocking profile：网络阻塞概况
* Synchronization blocking profile：同步阻塞概况
* Syscall blocking profile：系统调用阻塞概况
* Scheduler latency profile：调度延迟概况
* User defined tasks：用户自定义任务
* User defined regions：用户自定义区域
* Minimum mutator utilization：最低 Mutator 利用率

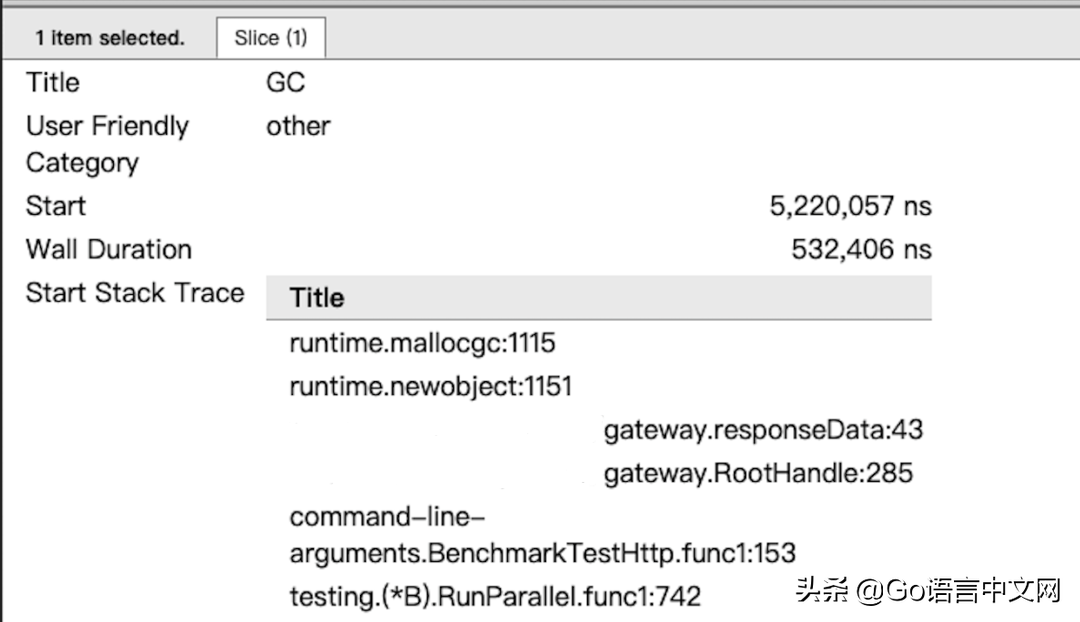
GC 相关的信息可以在 View trace 中看到



可通过点击 heap 的色块区域，查看 heap 信息。



点击 GC 对应行的蓝色色块，查看 GC 耗时及相关回收信息。



通过这两个信息就可以确认是否存在 GC 问题，以及造成高 GC 的可能原因。

**使用问题**

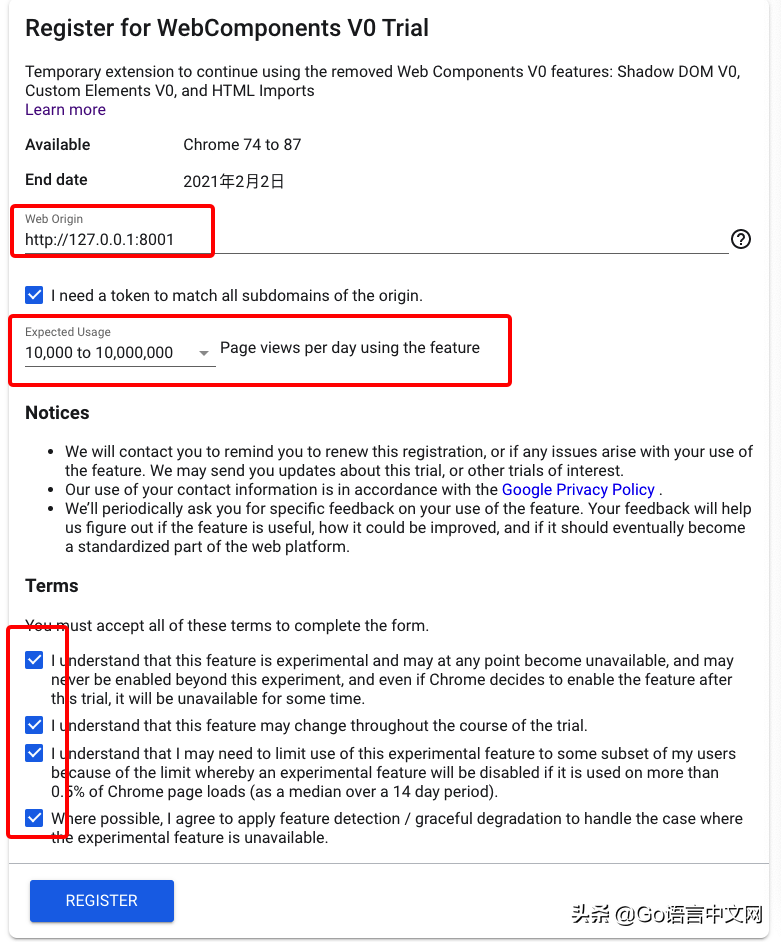
trace 的展示仅支持 chrome 浏览器。但是目前常用的 chrome 浏览器屏蔽了 go tool trace 使用的 HTML import 功能。即打开“view trace”时，会出现一片空白。并可以在 console 中看到警告信息：

HTML Imports is deprecated and has now been removed as of M80. See https://www.chromestatus.com/features/5144752345317376 and https://developers.google.com/web/updates/2019/07/web-components-time-to-upgrade for more details.

**解决办法**

**申请 token**

* https://developers.chrome.com/origintrials/#/register\_trial/2431943798780067841 然后登录
* web origin 处填写 http://localhost:8001 端口只能是 8000 - 8003，支持 http 和 https。（也可以填写 127.0.0.1:8001,依赖于你浏览器中显示的地址，否则对不上的话，还要手动改一下)



* 点击注册后即可看到 token

**修改 trace.go**

* 编辑${GOROOT}/src/cmd/trace/trace.go 文件，在文件中找到 templTrace 然后在 标签的下一行添加<meta http-equiv="origin-trial" content="你复制的token">

**重新编译 go**

* ${GOROOT}/src 目录，执行 ./all.bash
* 若提示：ERROR: Cannot find go1.4\bin\go Set GOROOT\_BOOTSTRAP to a working Go tree >= Go 1.4 则需要先安装一个 go1.4 的版本，再通过它来编译 go。（下载链接https://dl.google.com/go/go1.4-bootstrap-20171003.tar.gz） 在 go1.4/src 下执行./make.bash. 指定 GOROOT\_BOOTSTRAP 为 go1.4 的根目录。然后就可以重新编译 go

**查看 trace**

go tool trace -http=localhost:8001 trace.out

若打开 view trace 还是空白,则检查一下浏览器地址栏中的地址，是否与注册时的一样。即注册用的 localhost 或 127.0.0.1 则地址栏中也要一样。

**常见性能瓶颈**

**业务逻辑**

出现无效甚至降低性能的逻辑。常见的有

* 逻辑重复：相同的操作在不同的位置做了多次或循环跳出的条件设置不当。
* 资源未复用：内存频繁申请和释放，数据库链接频繁建立和销毁等。
* 无效代码。

**存储**

未选择恰当的存储方式，常见的有：

* 临时数据存放到数据库中，导致频繁读写数据库。
* 将复杂的树状结构的数据用 SQL 数据库存储，出现大量冗余列，并且在读写时要进行拆解和拼接。
* 数据库表设计不当，无法有效利用索引查询，导致查询操作耗时高甚至出现大量慢查询。
* 热点数据未使用缓存，导致数据库负载过高，响应速度下降。

**并发处理**

并发操作的问题主要出现在资源竞争上，常见的有：

* 死锁/活锁导致大量阻塞，性能严重下降。
* 资源竞争激烈：大量的线程或协程抢夺一个锁。
* 临界区过大：将不必要的操作也放入临界区，导致锁的释放速度过慢，引起其他线程或协程阻塞。

**golang 部分细节简介**

在优化之前，我们需要对 golang 的实现细节有一个简单的了解，才能明白哪些地方有问题，哪些地方可以优化，以及怎么优化。以下内容的详细讲解建议查阅网上优秀的 blog。对语言的底层实现机制最好有个基本的了解，否则有时候掉到坑里都不知道为啥。

**协程调度**

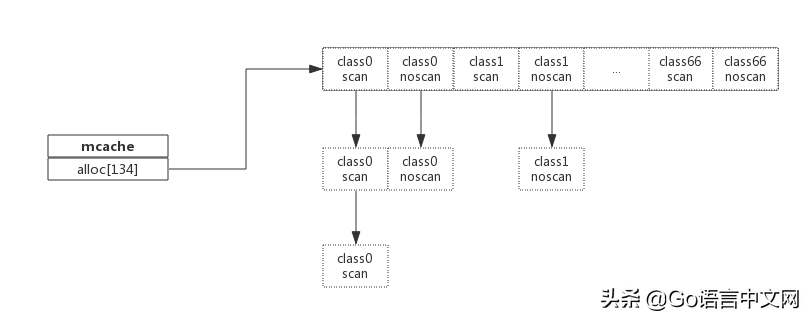
Golang 调度是非抢占式多任务处理，由协程主动交出控制权。遇到如下条件时，才有可能交出控制权

* I/O,select
* channel
* 等待锁
* 函数调用（是一个切换的机会，是否会切换由调度器决定）
* runtime.Gosched()

因此，若存在较长时间的 for 循环处理，并且循环内没有上述逻辑时，会阻塞住其他的协程调度。在实际编码中一定要注意。

**内存管理**

Go 为每个逻辑处理器（P）提供了一个称为**mcache**的本地内存线程缓存。每个 mcache 中持有 67 个级别的 mspan。每个 msapn 又包含两种：scan（包含指针的对象）和 noscan（不包含指针的对象）。在进行垃圾收集时，**GC 无需遍历 noscan 对象**。

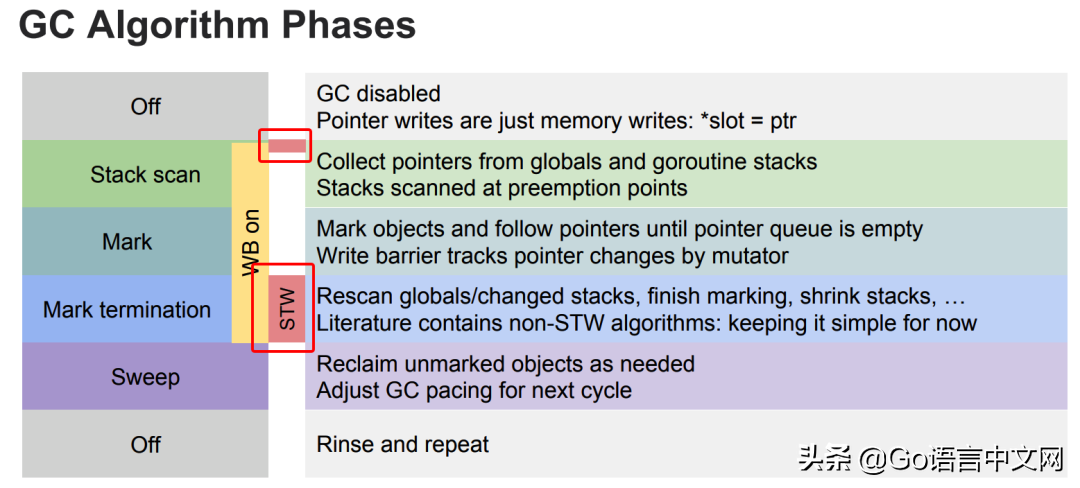


**GC 处理**

GC 的工作就是确定哪些内存可以释放，它是通过扫描内存查找内存分配的指针来完成这个工作的。GC 触发时机：

* 到达堆阈值：默认情况下，它将在堆大小加倍时运行，可通过 GOGC 来设定更高阈值（不建议变更此配置）
* 到达时间阈值：每两分钟会强制启动一次 GC 循环

为啥要注意 GC，是因为 GC 时出现 2 次 Stop the world，即停止所有协程，进行扫描操作。若是 GC 耗时高，则会严重影响服务器性能。



**变量逃逸**

注意，golang 中的栈是跟函数绑定的，函数结束时栈被回收。

**变量内存回收：**

* 如果分配在栈中，则函数执行结束可自动将内存回收；
* 如果分配在堆中，则函数执行结束可交给 GC（垃圾回收）处理；

而变量逃逸就意味着增加了堆中的对象个数，影响 GC 耗时。一般要尽量避免逃逸。

**逃逸分析不变性：**

1. 指向栈对象的指针不能存在于堆中；
2. 指向栈对象的指针不能在栈对象回收后存活；

在逃逸分析过程中，凡是发现出现违反上述约定的变量，就将其移到堆中。

**逃逸常见的情况：**

* 指针逃逸：返回局部变量的地址（不变性 2）
* 栈空间不足
* 动态类型逃逸：如 fmt.Sprintf,json.Marshel 等接受变量为...interface{}函数的调用，会导致传入的变量逃逸。
* 闭包引用

**包含指针类型的底层结构**

**string**

type StringHeader struct {

 Data uintptr

 Len  int

}

**slice**

type SliceHeader struct {

 Data uintptr

 Len  int

 Cap  int

}

**map**

type hmap struct {

 count     int

 flags     uint8

 B         uint8

 noverflow uint16

 hash0     uint32

 buckets    unsafe.Pointer

 oldbuckets unsafe.Pointer

 nevacuate  uintptr

 extra \*mapextra

}

这些是常见会包含指针的对象。尤其是 string，在后台应用中大量出现。并经常会作为 map 的 key 或 value。若数据量较大时，就会引发 GC 耗时上升。同时，我们可以注意到 string 和 slice 非常相似，从某种意义上说它们之间是可以直接互相转换的。这就可以避免 string 和[]byte 之间类型转换时，进行内存拷贝

**类型转换优化**

func String(b []byte) string {

 return \*(\*string)(unsafe.Pointer(&b))

}

func Str2Bytes(s string) []byte {

 x := (\*[2]uintptr)(unsafe.Pointer(&s))

 h := [3]uintptr{x[0], x[1], x[1]}

 return \*(\*[]byte)(unsafe.Pointer(&h))

}

**性能测试方式**

**本地测试**

将服务处理的核心逻辑，使用 go test 的 benchmark 加 pprof 来测试。建议上线前，就对整个业务逻辑的性能进行测试，提前优化瓶颈。

**线上测试**

一般 http 服务可以通过常见的测试工具进行压测，如 wrk，locust 等。taf 服务则需要我们自己编写一些测试脚本。同时，要注意的是，压测的目的是定位出服务的最佳性能，而不是盲目的高并发请求测试。因此，一般需要逐步提升并发请求数量，来定位出服务的最佳性能点。

注意：由于 taf 平台具备扩容功能，因此为了更准确的测试，我们应该在测试前关闭要测试节点的自动扩容。

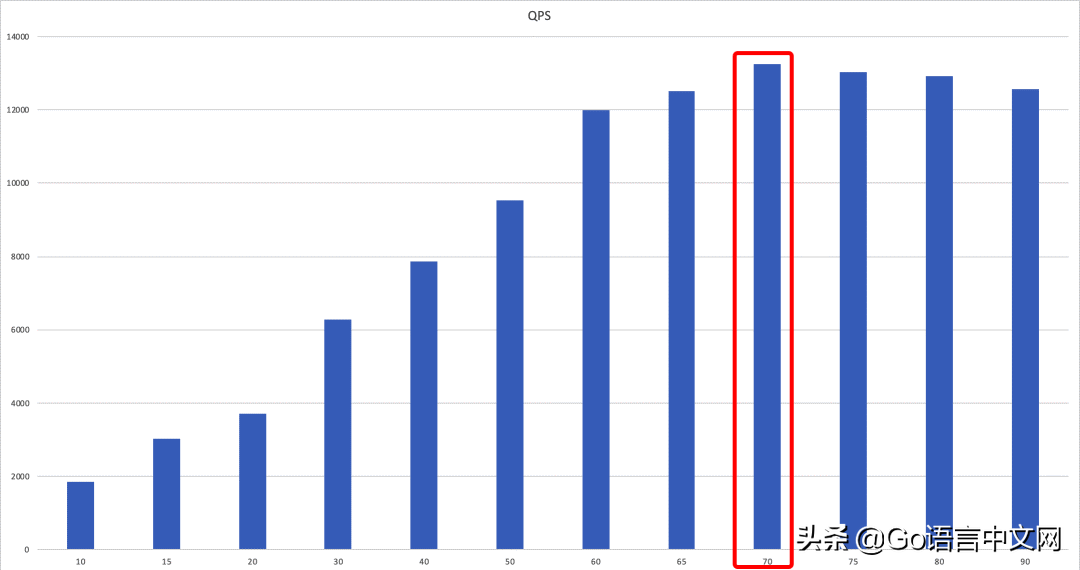
**实际项目优化**

为了避免影响后端服务，也为了避免后端服务影响网关自身。因此，我们需要在压测前，将对后端服务的调用屏蔽。

* 测试准备：屏蔽远程调用：下游服务调用，健康度上报，统计上报，远程日志。以便关注网关自身性能。

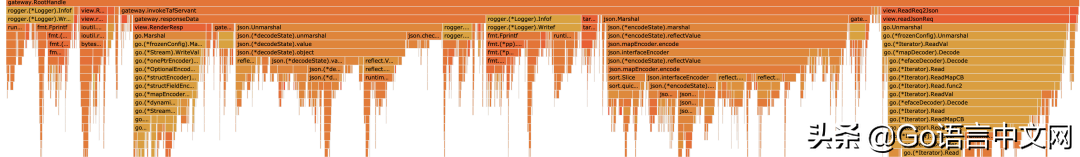
**QPS 现状**

首先看下当前业务的性能指标，使用 wrk 压测网关服务



可以看出，在总链接数为 70 的时候，QPS 最高，为 13245。

**火焰图**



根据火焰图我们定位出 cpu 占比较高的几个方法为：

* json.Marshal
* json.Unmarshal
* rogger.Infof

为了方便测试，将代码改为本地运行，并通过 benchmark 的方式来对比修改前后的差异。

由于正式环境使用的 golang 版本为 1.12，因此本地测试时，也要使用同样的版本。

**benchmark**

Benchmark 50000000 3669 ns/op 4601 B/op 73 allocs/op

查看 cpu 和 memory 的 profile，发现健康度上报的数据结构填充占比较高。这部分逻辑基于 tars 框架实现。暂时忽略，为避免影响其他测试，先注释掉。再看看 benchmark。

Benchmark 500000 3146 ns/op 2069 B/op 55 allocs/op

**优化策略**

**JSON 优化**

先查看 json 解析的部分，看看是否有优化空间

**请求处理**

//RootHandle view.ReadReq2Json readJsonReq 中进行json解析

type GatewayReqBody struct {

 Header  GatewayReqBodyHeader   `json:"header"`

 Payload map[string]interface{} `json:"payload"`

}

func readJsonReq(data []byte, req \*model.GatewayReqBody) error {

 dataMap := make(map[string]interface{})

 err := jsoniter.Unmarshal(data, &dataMap)

 ...

  headerMap, ok := header.(map[string]interface{})

  businessName, ok := headerMap["businessName"]

  qua, ok := headerMap["qua"]

  sessionId, ok := headerMap["sessionId"]

  ...

  payload, ok := dataMap["payload"]

  req.Payload, ok = payload.(map[string]interface{})

}

这个函数本质上将 data 解析为 model.GatewayReqBody 类型的结构体。但是这里却存在 2 个问题

1. 使用了复杂的解析方式，先将 data 解析为 map，再通过每个字段的名字来取值，并进行类型转换。
2. Req.Playload 解析为一个 map。但又未使用。我们看看后面这个 payload 是用来做啥。确认是否为无效代码。

func invokeTafServant(resp http.ResponseWriter, gatewayHttpReq \*model.GatewayHttpReq) {

 ...

  payloadBytes, err := json.Marshal(gatewayHttpReq.ReqBody.Payload)

 if err == nil {

  commonReq.Payload = string(payloadBytes)

 } else {

  responseData(gatewayHttpReq, StatusInternalServerError, "封装json异常", "", resp)

  return

 }

  ...

 }

后续的使用中，我们可以看到，又将这个 payload 转为 string。因此，我们可以确定，上面的 json 解析是没有意义，同时也会浪费资源（payload 数据量一般不小）。

**优化方法**

* golang 自带的 json 解析性能较低，这里我们可以替换为github.com/json-iterator来提升性能
* 在 golang 中，遇到不需要解析的 json 数据，可以将其类型声明为json.RawMessage. 即，可以将上述 2 个方法优化为

type GatewayReqBody struct {

 Header  GatewayReqBodyHeader `json:"header"`

 Payload json.RawMessage      `json:"payload"`

}

func readJsonReq(data []byte, req \*model.GatewayReqBody) error {

 err := jsoniter.Unmarshal(data, req)

 if err != nil {

  return jsonParseErr

 }

 for k, v := range req.Header.Qua {

  req.Header.Qua[k] = v

  if len(req.Header.QuaStr) == 0 {

   req.Header.QuaStr = k + "=" + v

  } else {

   req.Header.QuaStr += "&" + k + "=" + v

  }

 }

 return nil

}

func invokeTafServant(resp http.ResponseWriter, gatewayHttpReq \*model.GatewayHttpReq) {

 commonReq.Payload = string(gatewayHttpReq.ReqBody.Payload)

}

* 这里注意！出现了 string 和[]byte 之间的类型转换.为了避免内存拷贝，这里将 string()改为上面的类型转换优化中所定义的转换函数，即commonReq.Payload = encode.String(gatewayHttpReq.ReqBody.Payload)

**回包处理**

type GatewayRespBody struct {

 Header  GatewayRespBodyHeader  `json:"header"`

 Payload map[string]interface{} `json:"payload"`

}

func responseData(gatewayReq \*model.GatewayHttpReq, code int32, message string, payload string, resp http.ResponseWriter) {

 jsonPayload := make(map[string]interface{})

 if len(payload) != 0 {

  err := json.Unmarshal([]byte(payload), &jsonPayload)

  if err != nil {

   ...

  }

 }

 body := model.GatewayRespBody{

  Header: model.GatewayRespBodyHeader{

   Code:    code,

   Message: message,

  },

  Payload: jsonPayload,

 }

  data, err := view.RenderResp("json", &body)

  ...

  resp.WriteHeader(http.StatusOK)

 resp.Write(data)

}

同样的，这里的 jsonPayload，也是出现了不必要的 json 解析。我们可以改为

type GatewayRespBody struct {

 Header  GatewayRespBodyHeader  `json:"header"`

 Payload json.RawMessage `json:"payload"`

}

body := model.GatewayRespBody{

  Header: model.GatewayRespBodyHeader{

   Code:    code,

   Message: message,

  },

  Payload: encode.Str2Bytes(payload),

 }

然后在 view.RenderResp 方法中

func RenderResp(format string, resp interface{}) ([]byte, error) {

 if "json" == format {

  return jsoniter.Marshal(resp)

 }

 return nil, errors.New("format error")

}

**benchmark**

Benchmark 500000 3326 ns/op 2842 B/op 50 allocs/op

虽然对象 alloc 减少了，但单次操作内存使用增加了，且性能下降了。这就有点奇怪了。我们来对比一下 2 个情况下的 pprof。

**逃逸分析及处理**

**go tool pprof -base**

* cpu 差异 flat flat% sum% cum cum%  
  0.09s 1.17% 1.17% 0.40s 5.20% runtime.mallocgc0.01s 0.13% 1.30% 0.35s 4.55% /vendor/github.com/json-iterator/go.(\*Iterator).readObjectStart0 0% 1.30% 0.35s 4.55% /vendor/github.com/json-iterator/go.(\*twoFieldsStructDecoder).Decode
* mem 差异 flat flat% sum% cum cum%  
  478.96MB 20.33% 20.33% 279.94MB 11.88% gateway.RootHandle0 0% 20.33% 279.94MB 11.88% command-line-arguments.BenchmarkTestHttp.func10 0% 20.33% 279.94MB 11.88% testing.(\*B).RunParallel.func1

可以看出 RootHandle 多了 478.96M 的内存使用。通过 list RootHandle 对比 2 个情况下的内存使用。发现修改后的 RootHandle 中多出了这一行：475.46MB 475.46MB 158: gatewayHttpReq := model.GatewayHttpReq{} 这一般意味着变量 gatewayHttpReq 出现了逃逸。

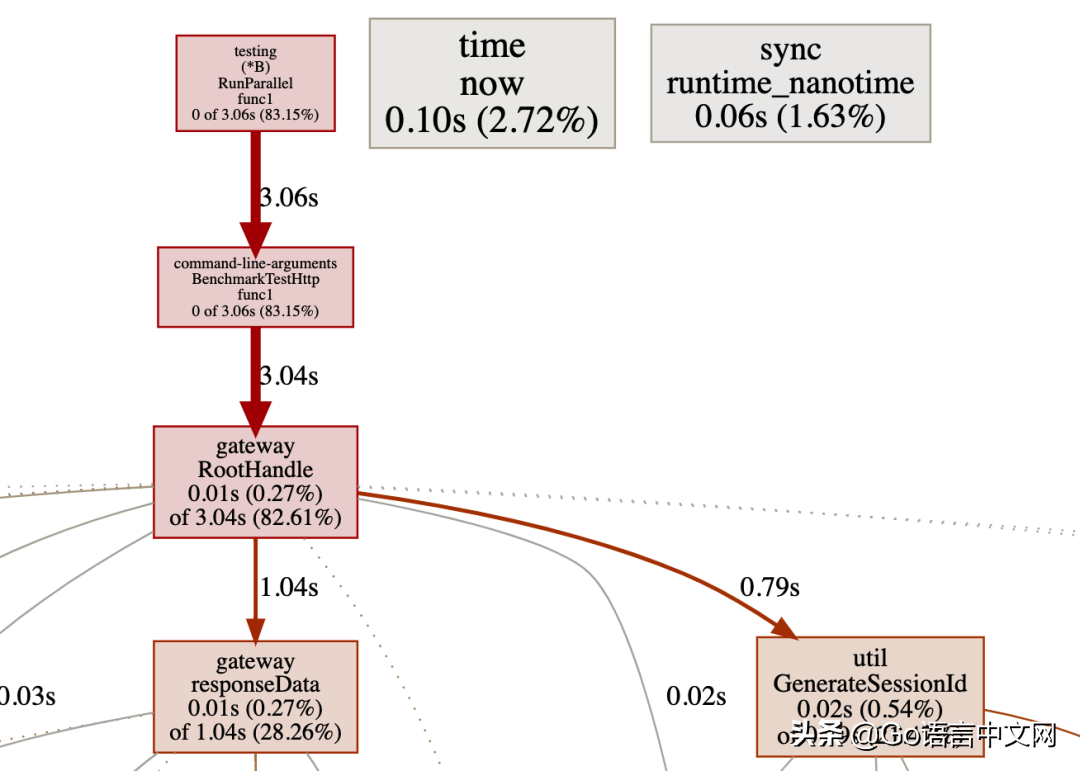
* go build -gcflags "-m -m" gateway/\*.gogateway/logic.go:270:26: &gatewayHttpReq escapes to heap可以看到确实出现了逃逸。这个对应的代码为err = view.ReadReq2Json(&gatewayHttpReq),而造成逃逸的本质是因为上面改动了函数 readJsonReq（动态类型逃逸，即函数参数为 interface 类型，无法在编译时确定具体类型的）func readJsonReq(data []byte, req \*model.GatewayReqBody) error {  
  err := jsoniter.Unmarshal(data, req) ...}因此，这里需要特殊处理一下，改为func readJsonReq(data []byte, req \*model.GatewayReqBody) error {var tmp model.GatewayReqBodyerr := jsoniter.Unmarshal(data, &tmp)...}

**benchmark**

Benchmark 500000 2994 ns/op 1892 B/op 50 allocs/op

可以看到堆内存使用明显下降。性能也提升了。再看一下 pprof，寻找下个瓶颈。

**cpu profile**



抛开 responeseData(他内部主要是日志打印占比高），占比较高的为 util.GenerateSessionId，先来看看这个怎么优化。

**随机字符串生成**

var letterRunes = []rune("0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")

func RandStringRunes(n int) string {

 b := make([]rune, n)

 for i := range b {

  b[i] = letterRunes[rand.Intn(len(letterRunes))]

 }

 return string(b)

}

目前的生成方式使用的类型是 rune，但其实用 byte 就够了。另外，letterRunes 是 62 个字符，即最大需要 6 位的 index 就可以遍历完成了。而随机数获取的是 63 位。即每个随机数，其实可以产生 10 个随机字符。而不用每个字符都获取一次随机数。所以我们改为

const (

 letterBytes   = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

 letterIdxBits = 6

 letterIdxMask = 1<<letterIdxBits - 1

 letterIdxMax  = 63 / letterIdxBits

)

func RandStringRunes(n int) string {

 b := make([]byte, n)

 for i, cache, remain := n-1, rand.Int63(), letterIdxMax; i >= 0; {

  if remain == 0 {

   cache, remain = rand.Int63(), letterIdxMax

  }

  if idx := int(cache & letterIdxMask); idx < len(letterBytes) {

   b[i] = letterBytes[idx]

   i--

  }

  cache >>= letterIdxBits

  remain--

 }

 return string(b)

}

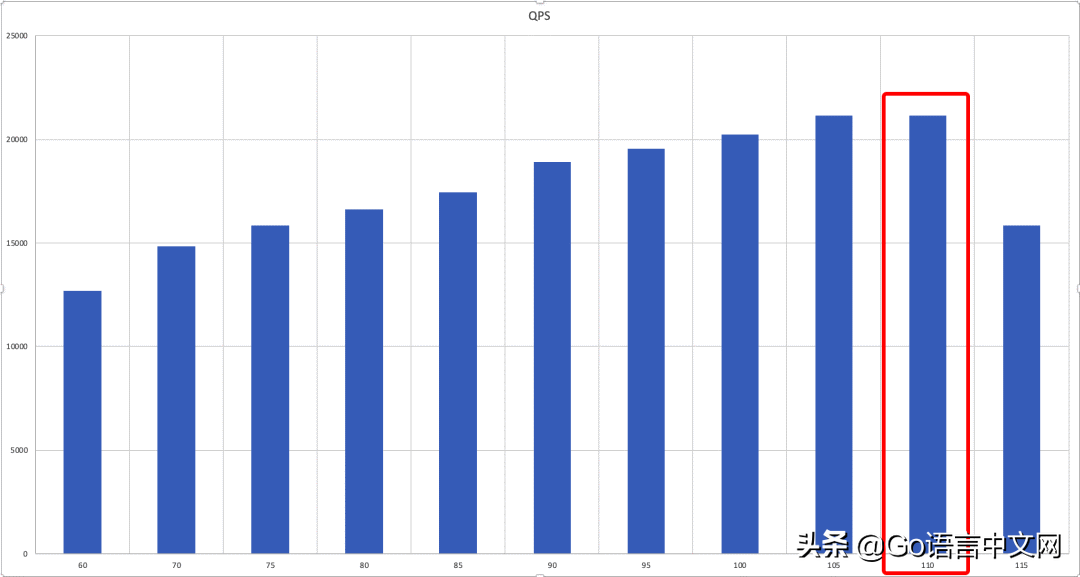
**benchmark**

Benchmark 1000000 1487 ns/op 1843 B/op 50 allocs/op

**类型转换及字符串拼接**

一般情况下，都会说将 string 和[]byte 的转换改为 unsafe；以及在字符串拼接时，用 byte.Buffer 代替 fmt.Sprintf。但是网关这里的情况比较特殊，字符串的操作基本集中在打印日志的操作。而 tars 的日志打印本身就是通过 byte.Buffer 拼接的。所以这可以避免。另外，由于日志打印量大，使用 unsafe 转换[]byte 为 string 带来的收益，往往会因为逃逸从而影响 GC，反正会影响性能。因此，不同的场景下，不能简单的套用一些优化方法。需要通过压测及结果分析来判断具体的优化策略。

**优化结果**



可以看到优化后，最大链接数为 110，最高 QPS 为**21153.35**。对比之前的**13245**，大约提升 60%。

**后续**

从 pprof 中可以看到日志打印，远程日志，健康上报等信息占用较多 cpu 资源，且导致多个数据逃逸（尤其是日志打印）。过多的日志基本等于没有日志。后续可考虑裁剪日志，仅保留出错时的上下文信息。

**总结**

* 性能查看工具 pprof,trace 及压测工具 wrk 或其他压测工具的使用要比较了解。
* 代码逻辑层面的走读非常重要，要尽量避免无效逻辑。
* 对于 golang 自身库存在缺陷的，可以寻找第三方库或自己改造。
* golang 版本尽量更新，这次的测试是在 golang1.12 下进行的。而 go1.13 甚至 go1.14 在很多地方进行了改进。比如 fmt.Sprintf，sync.Pool 等。替换成新版本应该能进一步提升性能。
* 本地 benchmark 结果不等于线上运行结果。尤其是在使用缓存来提高处理速度时，要考虑 GC 的影响。
* 传参数或返回值时，尽量按 golang 的设计哲学，少用指针，多用值对象，避免引起过多的变量逃逸，导致 GC 耗时暴涨。struct 的大小一般在 2K 以下的拷贝传值，比使用指针要快（可针对不同的机器压测，判断各自的阈值)。
* 值类型在满足需要的情况下，越小越好。能用 int8，就不要用 int64。
* 资源尽量复用,在 golang1.13 以上，可以考虑使用 sync.Pool 缓存会重复申请的内存或对象。或者自己使用并管理大块内存，用来存储小对象，避免 GC 影响（如本地缓存的场景)。