2021-06性能优化记录

由 余堤堤创建, 最后修改于昨天4:51 下午

预备知识 pprof工具使用

1. 获取CPU Profiling

2. 使用top命令和火焰图, 明确程序的cpu消耗概况(业务消耗+gc消耗)

知道cpu主要消耗在哪些接口, 哪些环节。

一般而言, 对于带gc的语言，cpu消耗主要包括2部分: 业务消耗+gc消耗。

2.1 使用top命令查看top10的cpu消耗都在哪些函数上。

```
go tool pprof pprof.product_server.samples.cpu.001.pb.gz
top
```

可以看到主要的cup消耗都在GC函数上。

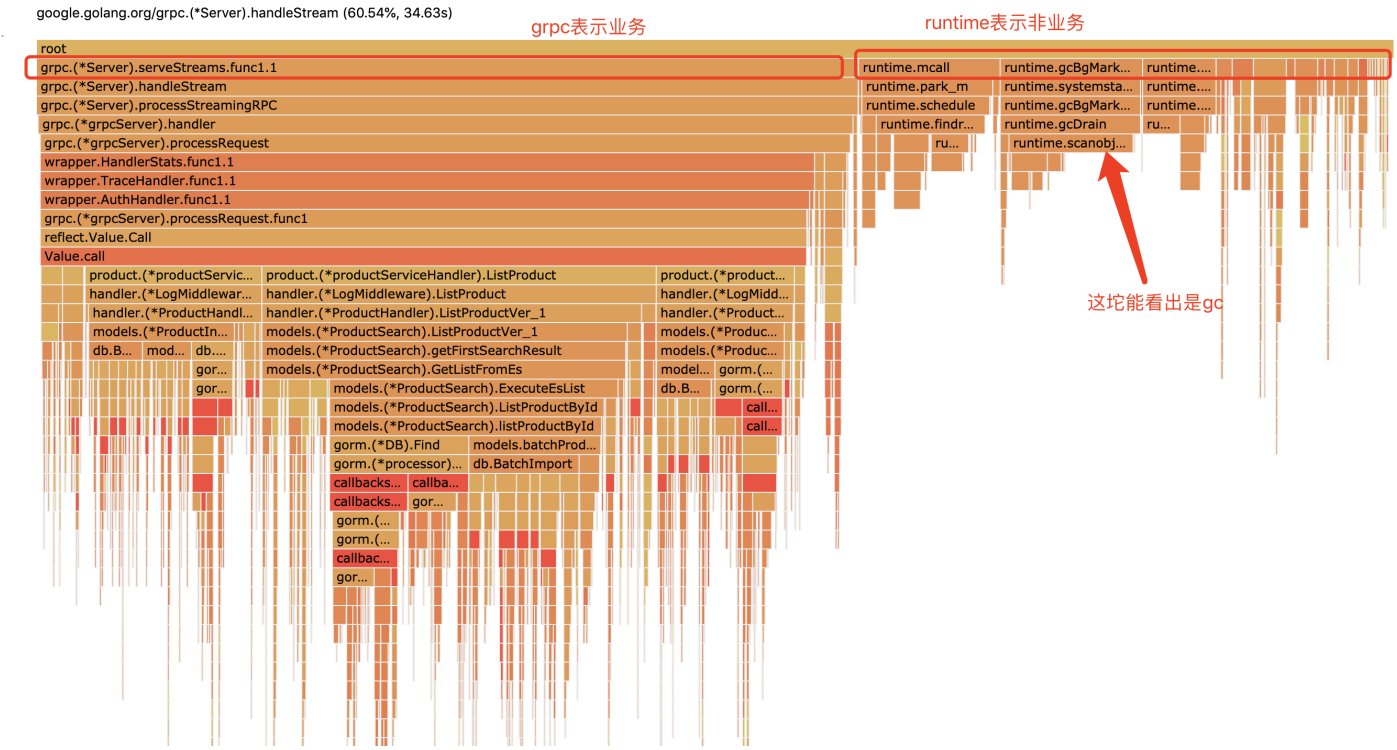
```
(base) Didis-MacBook-Pro-4:Downloads didiyu$ go tool pprof  pprof.product_server.samples.cpu.001.pb.gz
File: product_server
Type: cpu
Time: Jun 3, 2021 at 10:28am (CST)
Duration: 5mins, Total samples = 57.20s (19.06%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 21290ms, 37.22% of 57200ms total
Dropped 1381 nodes (cum <= 286ms)
Showing top 10 nodes out of 367
      flat  flat%   sum%        cum   cum%
    5300ms  9.27%  9.27%     5300ms  9.27%  runtime.futex
    4640ms  8.11% 17.38%     6390ms 11.17%  syscall.Syscall
    2910ms  5.09% 22.47%     5350ms  9.35%  runtime.scanobject
    1950ms  3.41% 25.87%     1950ms  3.41%  runtime.usleep
    1370ms  2.40% 28.27%     1370ms  2.40%  runtime.epollwait
    1330ms  2.33% 30.59%     6510ms 11.38%  runtime.mallocgc
    1150ms  2.01% 32.60%     1790ms  3.13%  runtime.heapBitsSetType
    1060ms  1.85% 34.46%     1060ms  1.85%  runtime.nextFreeFast
     820ms  1.43% 35.89%      820ms  1.43%  runtime.memclrNoHeapPointers
     760ms  1.33% 37.22%      790ms  1.38%  runtime.pageIndexOf
(pprof)
```

2.2 使用火焰图查看先主要的cpu消耗

```
go tool pprof -http=:8080 pprof.product_server.samples.cpu.001.pb.gz
```

首先看业务消耗和非业务(gc)消耗的总体情况。

从图中可以看出 只有60%的cpu用于业务，还有大量cpu用到了非业务中, 比如调度和gc上。



3.然后从业务和非业务的cpu消耗两个角度具体分析

a .分析业务中哪块是最耗时，这样我们可以针对这块耗时的业务进行处理。

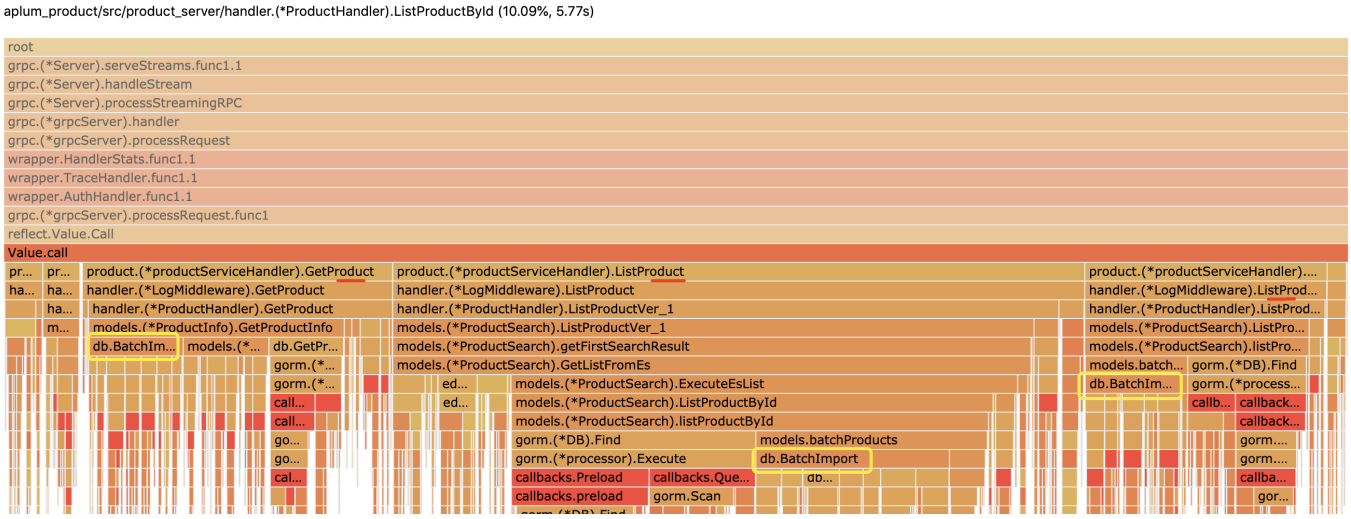
b. 需要清理的对象多，导致gc消耗高，肯定是分配的对象数量巨大，那么我们需要找到是哪些对象分配比较多，看能不能减少这些对象的分配。

3.1 进一步分析业务消耗cup的情况(a)

从图中可以看出，

业务中消耗的cpu主要集中在这3个接口, GetProduct,ListProduct,ListProductById。并且，主要都是消耗在数据库的查询上(因为gorm的在各个接口都占了不小的比例)

从图中可以看出，三个接口内部都调用db.BatchImport方法，耗时较多，如果可以优化该方法，性价比很高。

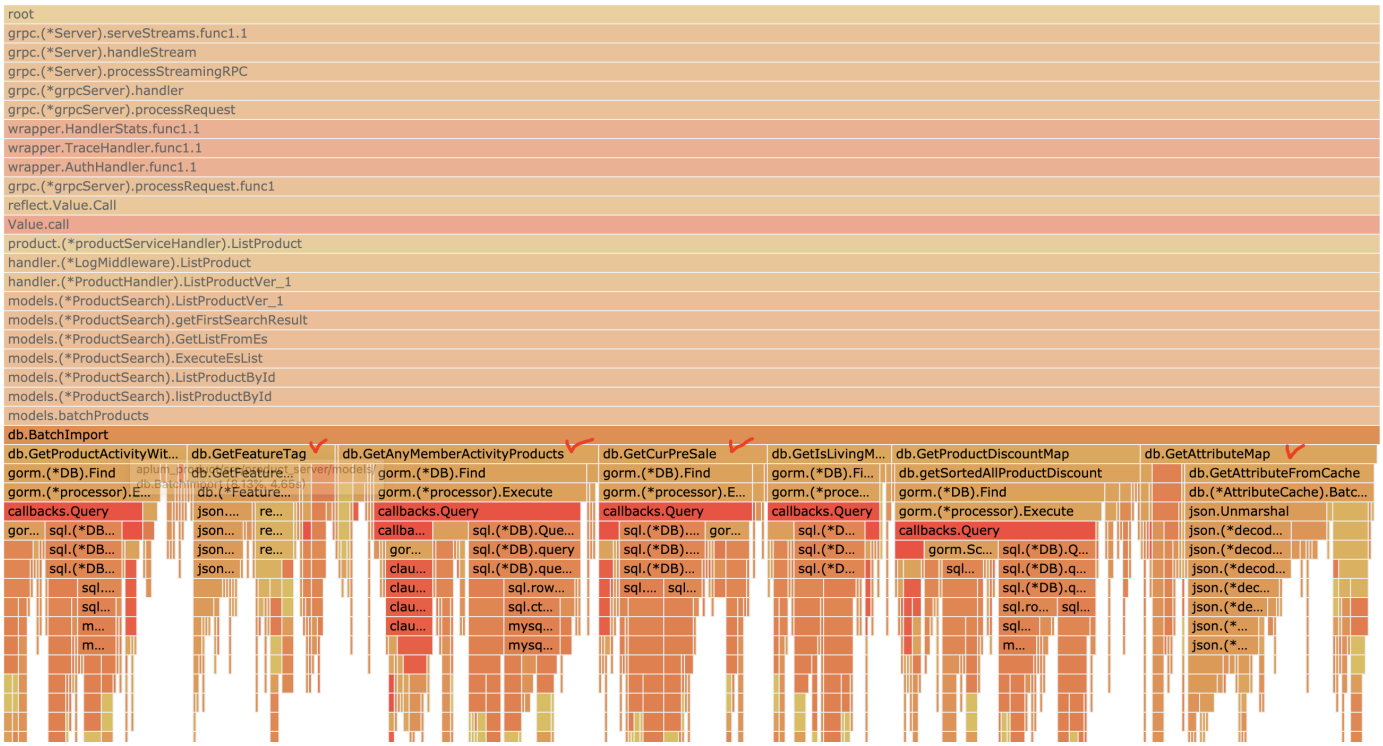


进一步查看db.BatchImport方法内部，可以发现主要是7个批量获取数据的方法，其中红勾的4个方法比较纯粹，考虑进行一定优化。

GetFeatureTag方法，GetAttributeMap方法: 不是价格相关的信息，可以考虑加缓存。

GetCurPreSale方法,GetAnyMemberActivityProducts方法: 从业务上说属于大部分情况都无效的sql查询，可以考虑通过某种标记，避免这2个无效的sql查询。

aplum_product/src/product_server/models/db.BatchImport (8.13%, 4.65s) 结合业务，这个函数的sql查询最多



3.2 进一步分析gc消耗cup的情况(b)

需要获取Memory Profiling。

用--inuse_space来分析程序常驻内存的占用情况；
用--alloc_objects来分析内存的临时分配情况；

```
// 生成Memory Profiling
./pprof http://localhost:8018/debug/pprof/allocs
// 命令行打开Memory Profiling
go tool pprof -alloc_objects pprof.product_server.alloc_objects.alloc_space.inuse_objects.inuse_space.001.pb.gz
// 网页打开
go tool pprof -http=:8081 -alloc_objects pprof.product_server.alloc_objects.alloc_space.inuse_objects.inuse_space.001.pb.gz
```

```
(base) Didis-MacBook-Pro-4:Downloads didiyu$ go tool pprof -alloc_objects pprof.product_server.alloc_objects.alloc_space.inuse_objects.inuse_space.001.pb.gz
File: product_server
Type: alloc_objects
Time: Jun 3, 2021 at 3:39pm (CST)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 209797439, 50.04% of 419232089 total
Dropped 933 nodes (cum <= 2096160)
Showing top 10 nodes out of 241
      flat  flat%   sum%        cum   cum%
  69783359 16.65% 16.65%   69783359 16.65%  reflect.New
  41796725  9.97% 26.62%   43287709 10.33%  github.com/go-sql-driver/mysql.(*binaryRows).readRow
  26802662  6.39% 33.01%   26802662  6.39%  reflect.packEface
  16726596  3.99% 37.00%   16726596  3.99%  strings.genSplit
  15826663  3.78% 40.77%   15826663  3.78%  github.com/go-sql-driver/mysql.(*mysqlConn).readColumns
   9387917  2.24% 43.01%  166594950 39.74%  gorm.io/gorm.Scan
   8490569  2.03% 45.04%   22018638  5.25%  aplum_product/src/product_server/models/db.(*TProduct).SetActivityRule
   8082677  1.93% 46.97%   38753972  9.24%  database/sql.convertAssignRows
   6914134  1.65% 48.62%   13041825  3.11%  github.com/json-iterator/go/extra.(*fuzzyIntegerDecoder).Decode
   5986137  1.43% 50.04%    5986137  1.43%  strings.(*Builder).grow
```

可以看到“SetActivityRule”函数，分配的内存占总的分配内存的2.03%(flat%列),在top10之内，算是很大的了。

使用(pprof) list SetActivityRule 命令查看SetActivityRule函数的内存分配情况，定位到188行:validBaseCondition

```
.     8148490    188:       isAddToActivityConf, passed := validBaseCondition(p, config)
```

```
(pprof) list SetActivityRule
Total: 419232089
ROUTINE ======================== aplum_product/src/product_server/models/db.(*TProduct).SetActivityRule in /Users/didiyu/Downloads/go/src/aplum_product_clear_micro/src/product_server/models/db/set_discount_product.go
   8490569   22018638 (flat, cum)  5.25% of Total
         .          .    177:
         .          .    178:// SetActivityRule 结合白名单，从有效活动配置获取匹配的活动，并标记参加了活动(p.ActivityMap)
         .          .    179:func (p *TProduct) SetActivityRule(curDiscount float64, whitelist map[string]bool) {
         .          .    180:ConfigLoop:
         .          .    181:   for i, config := range p.ActivityConfigs {
   5996623    5996623    182:           log.Debug("%v conf %v,%v", i, config.ID, config.Name)
         .          .    183:           if config.ProductType == ProductTypeNew {
         .          .    184:                   log.Debug("二手商品不参加新品的商城活动 configId:%v,pid:%v", config.ID, p.ID)
         .          .    185:                   continue
         .          .    186:           }
         .          .    187:           // 比validTypes更基础的校验规则
         .    8148490    188:           isAddToActivityConf, passed := validBaseCondition(p, config)
         .          .    189:           if !passed {
         .          .    190:                   continue ConfigLoop
         .          .    191:           }
         .          .    192:           // 只有白名单中的活动才能被匹配
         .          .    193:           if _, exist := whitelist[config.ActivityType]; !exist {
         .          .    194:                   continue
         .          .    195:           }
         .          .    196:           //校验规则
         .          .    197:           var activityIds []int32
         .    1624797    198:           validTypes := util.SplitToInt32Array(config.ValidTypes, ",")
         .          .    199:           actRuleValids := make([]func(p *TProduct) bool, 0)
         .          .    200:           for _, validType := range validTypes {
         .          .    201:                   switch validType {
         .          .    202:                   case ValidNolimit:
         .          .    203:                           actRuleValids = append(actRuleValids, validNolimit)
         .          .    204:                   case ValidBrand:
         .     136545    205:                           bids := util.SplitToInt32Array(config.Bids, ",")
         .          .    206:                           if len(bids) == 0 {
         .          .    207:                                   log.Error("待校验品牌id数组不存在:%s", config.Bids)
         .          .    208:                                   break ConfigLoop
         .          .    209:                           }
    120153     120153    210:                           brandMap := make(map[int32]bool)
                         211:                           for _, v := range bids {
```

使用(pprof) list validBaseCondition 命令 继续查看validBaseCondition函数的内存分配情况，定位到23行和42行。

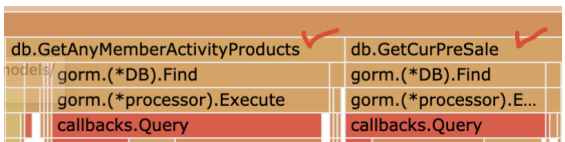
分别是因为split产生了大量对象和日志语句产生了大量对象。

```
(pprof) list validBaseCondition
Total: 419232089
ROUTINE ======================== aplum_product/src/product_server/models/db.validBaseCondition in /Users/didiyu/Downloads/go/src/aplum_product_clear_micro/src/product_server/models/db/promotion_not_join.go
   4227128    8148490 (flat, cum)  1.94% of Total
         .          .     18:// isAddToActivity: 是否提报的商城活动
         .          .     19:// passed: 是否能参加当前的商城活动
         .          .     20:func validBaseCondition(p *TProduct, conf *TActivityConf) (isAddToActivityConf, passed bool) {
         .          .     21:   // TODO 本次活动提报只针对个人卖家与vip卖家，但是mis预留此活动是针对个人卖家还是vip卖家还是b商户的选择条件
         .          .     22:   // 提报活动关联的满减|满折|限时折扣,不限制定价类型
         .    3047512     23:   for _, validType := range strings.Split(conf.ValidTypes, ",") {
         .     873850     24:           v, _ := strconv.Atoi(strings.TrimSpace(validType))
         .          .     25:           if v == ValidSellerDefinePrice { // 配置了规则21,则说明是提报活动的商城活动,(个人卖家与vip卖家)自主定价商品可以参加
         .          .     26:                   activityIds := util.SplitToInt32Array(conf.Aids, ",")
         .          .     27:                   if len(activityIds) == 0 {
         .          .     28:                           log.Error("提报商城活动活动页不存在,商城活动作废:id:%v,aids:%v", conf.ID, conf.Aids)
         .          .     29:                           return false, false
         .          .     30:                   } else {
         .          .     31:                           log.Debug("自主定价参加提报活动的商城活动:%v", conf.ID)
         .          .     32:                           return true, true
         .          .     33:                   }
         .          .     34:           }
         .          .     35:   }
         .          .     36:   // B端卖家:均不参加 满减满折| 限时折扣 | 爆款直降| 清仓价
         .          .     37:   // C端卖家(含vip)+自主定价:不参加 满减满折| 限时折扣 | 爆款直降| 清仓价
         .          .     38:   if (p.IsSellerC() && p.CooperateType == CooperateTypeSellerDefinePrice) || (p.IsSellerB()) {
         .          .     39:           if conf.ActivityType == FullOff || conf.ActivityType == FullReturn || conf.ActivityType == ConfDiscount ||
         .          .     40:                   conf.ActivityType == StraightDown || conf.ActivityType == Clearance {
         .          .     41:                   log.Debug("自主定价不参加.Appraise:%v,CooperateType:%v,conf:%v,ActivityType:%v",
   4227128    4227128     42:                           p.TSeller.Appraise, p.CooperateType, conf.ID, conf.ActivityType)
         .          .     43:                   return false, false
         .          .     44:           }
         .          .     45:   }
         .          .     46:   return false, true
         .          .     47:}
```

4.解决方案

问题	方案	方案的坏处	依据的火焰图

商品数据获取1.0:
1次8个商品,需要查询7个表,7*8=56个sql
	p1, a,b,c...表
	p2, a,b,c...表
	本来的伪代码: 每个商品都单独sql查表。
	for i := range 0-7{
	r.appen(getP(i))
	}
	func getP() p.Product{
	sql a set p.a
	sql b set p.b
	sql c set p.c
	...
	}
	修改为批量获取的伪代码: 所有商品同时用1个sql获取信息。
	bp := batchImport(0-7)
	for i := range 0-7 {
	p.a = bp[i].aList
	p.b= bp[i].bList
	...
	}
	type struct BatchResponse{
	aList map[pid]a
	bList map[pid]b
	...
	}
	func batchImport(pids) bp BatchResponse{
	batch sql a set bp.aList
	batch sql b set bp.bList
	...
	}
		1.代码就复杂一些。
		多出了组装商品信息的步骤。
		左侧红色的部分就是多出来的组装商品信息的代码。
			cpu

商品数据获取2.0 batchImport的多个批量sql查询耗费大量资源	思路 通过，加缓存，减少与数据库的交互。 1.内存缓存(异步刷新缓存) 商城活动表，品牌品类表 2.redis缓存 feature, attribute表 Q:使用内存缓存还是redis外部缓存? A:优先考虑内存缓存。 因为redis缓存需要额外的序列化和网络交互，而内存缓存只是多用了一些内存以及重启后缓存丢失。 比如商品活动表，缓存量不大且生命周期也不长，每5秒刷新全量缓存。 即使重启后，加载缓存的代价也很小，对性能基本无影响。 不过，以下情况需要使用redis缓存。 1.缓存量很大或者生命周期很长 如果服务重启缓存丢失导致性能波大，或者缓存量大短时间内无法预热完成，那么就用redis缓存，服务重启不会导致缓存丢失。 比如feature, attribute涉及几十万商品，预热时间较长，重启后缓存清空导致性能波动大。 总结: 就性能优化而言，能用内存缓存就用内存缓存。	内存缓存: 1.同一时刻, 不同机器(进程)间，缓存数据有较大概率不一样。 所以需要选择对缓存一致性要求不大的业务数据做内存缓存。 2.增加了内存消耗 redis缓存: 1.序列化和网络开销 2.引入一个新的外部依赖 没有特别大的弊端。	
商品数据获取3.0 batchImport存在几个大部分时间都注定没有结果(无效)sql查询	思路 结合业务特点吗，加标记(黑白名单)，跳过注定没有查询结果的sql。 eg: 预售,会员价 // 会员价只有在每月16日全天才需要获取数据。 	适用范围小，只有特定业务能用	
split产生了大量对象	使用map完成split输入和输出的映射，空间换时间。		alloc, alloc_objects

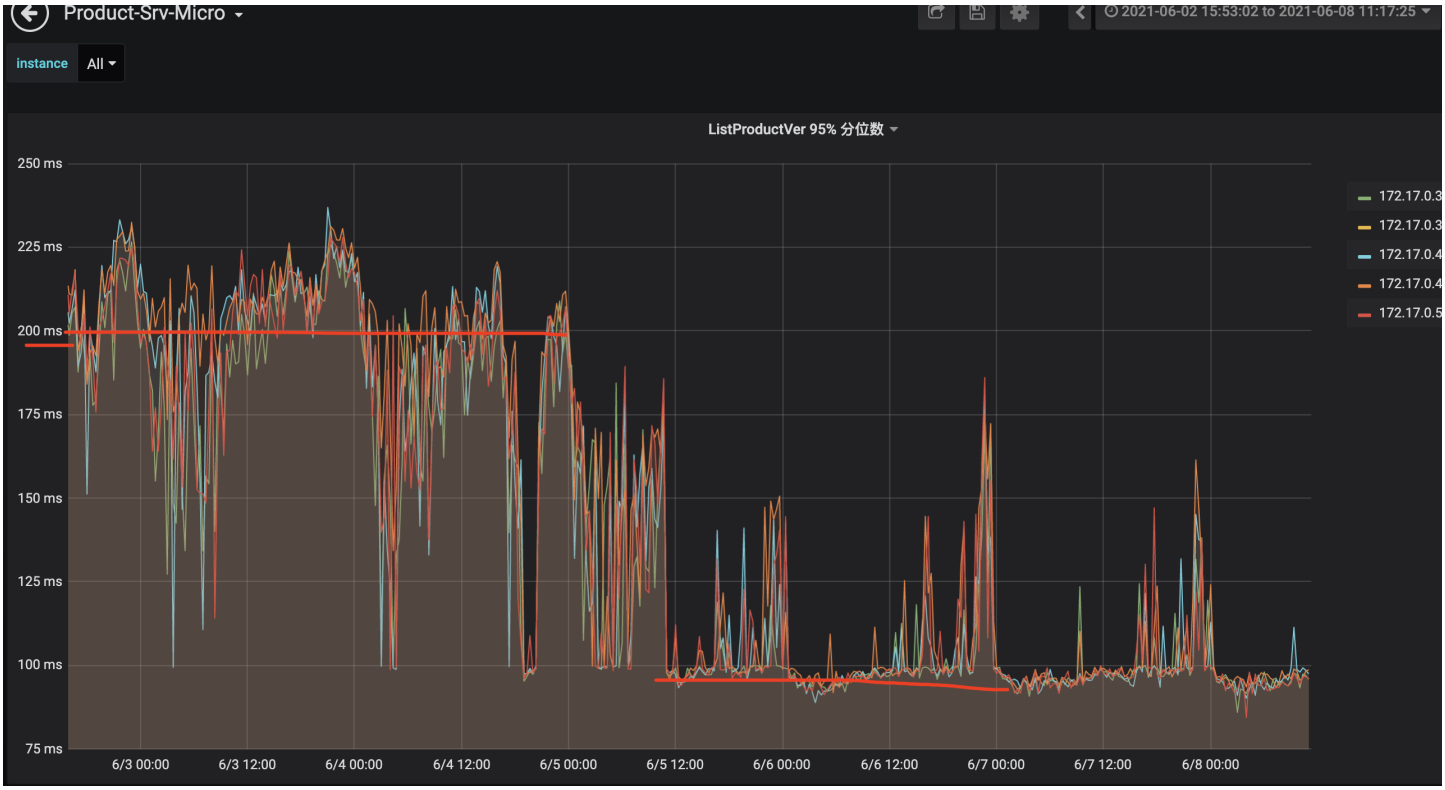
日志语句产生了大量对象	`if log.IsDebug() { // 加一个判断,避免执行线上执行log.Debug`
	`  log.Debug("%v conf %v,%v", i, config.ID, config.Name)`
	`}`
			alloc, alloc_objects

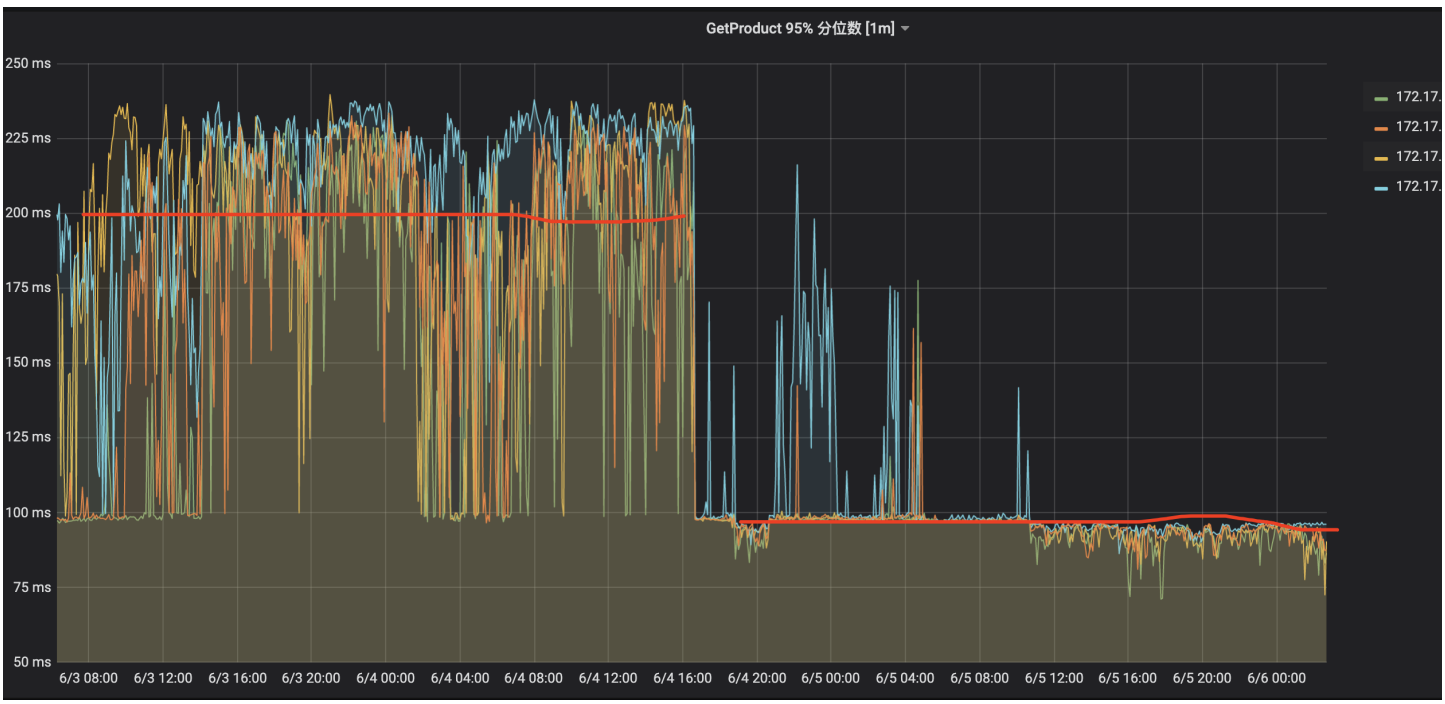
5.mysql预编译,一个意料之外的优化

当业务的数据库操作很多, 存在大量的mysql查询，此时可以考虑借助mysql的预编译机制提升外部组件的性能。

当是升级gorm也关注到了其发布日志的预编译部分，开始只是凭模糊的印象，认为其主要功能是防止sql注入的。

后面抱着试一试的心态启用了预编译，结果发现带来的性能提升非常巨大(接近100%)，超乎想象。





之前用的gorm v1版本，没有开预编译，gorm v2引入了预编译，使用如下。

```
db, err := gorm.Open(mysql.Open(DbInfo), &gorm.Config{
   Logger:      newLogger,
   PrepareStmt: true,
})
```

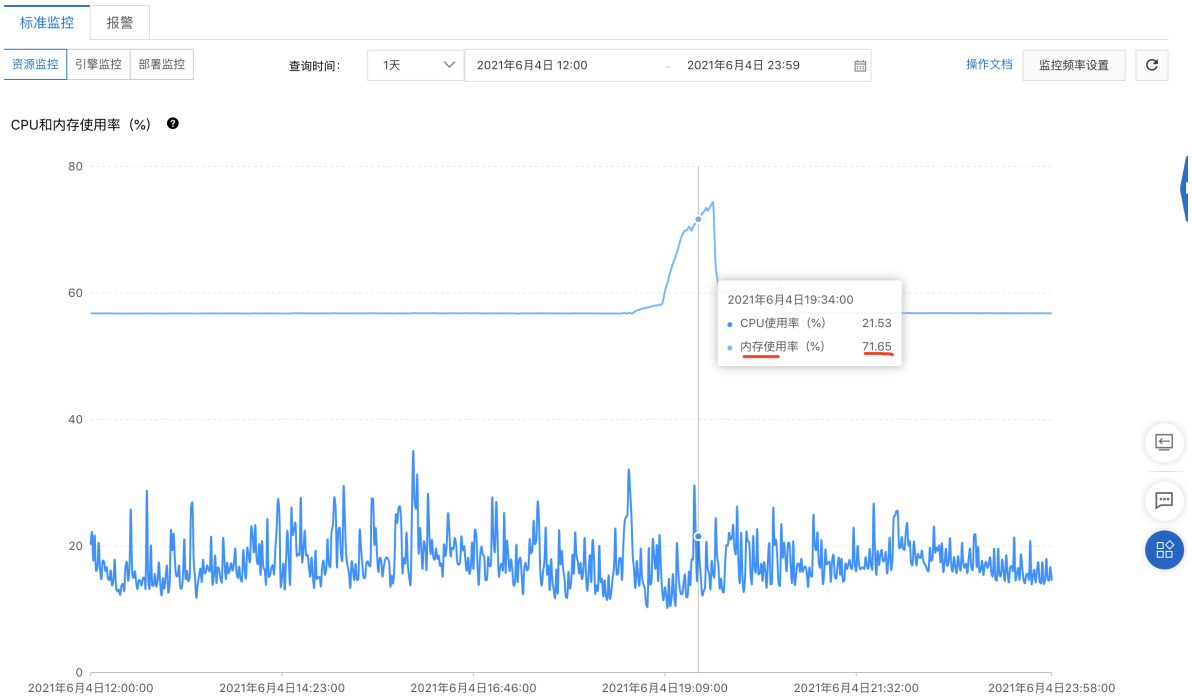
所以还是要多关注核心的第三方库的发布日志，如果有比较大的性能提升，可以考虑升级。

比如后续考虑升级qo的版本:

6.预编译也有弊端

1.开启预编译sql会增加mysql实例的内存使用。

从图中可以看到，内存使用率从58%增加到了71%。



2.mysql服务端存放的预编译语句数量是有限制的(max_prepared_stmt_count)。大量使用开启预编译语句，达到上线后，会报一下错误:

```
Error 1461: Can't create more than max_prepared_stmt_count statements (current value: 16382)","errorVerbose":"Error 1461: Can't
```

3.【这个是博客里面看到，具体是不是这样没有验证过】预编译语句的作用范围是具有session的，也就是说,2个session(数据库连接)查询1个相同的sql，那么在服务端会缓存2次。则剩余max_prepared_stmt_count-2可被其他服务使用。

之前不了解这一点，我们把thrift和micro两个服务都开启了预编译, 共计最多150*2=300个连接，然后就爆出了上述错误。为了处理上述错误，一方面增加了

`max_prepared_stmt_count`为3万多，另一方面，关闭了`thrift`的预编译，然后内存使用和预编译语句数量，就比较正常，不再出错了。

计算当前缓存的预编译语句数量参考:

https://zhuanlan.zhihu.com/p/67188414

https://blog.csdn.net/qq_21057881/article/details/75946735

https://cloud.tencent.com/developer/article/1068458

7.总结

1.性能优化思路

1.0 确定性能优化的目标。

If you can't measure it, you can't improve it!。借助公司同事参加字节跳动AB分享会的心得，我们处理一个问题，首先都要有选择1个目标，同时要有一个对应的度量是否达到该目标的方法。然后再去达成目标。这主要是为了时刻提醒自己注意性能优化的投入产出比。

结合业务，我们选择普罗米休斯监控的 商品搜索接口的p85作为性能优化的衡量指标，目标是p85<=100ms。

1.1 从应用程序本身的角度

a. 从业务出发。加缓存(redis缓存或内存缓存)。

b.如果是gc语言, 减少创建对象的数量，减轻gc压力。 也可以是空间换时间，比如split→map。

1.2 从三方库的角度

一方面要及时的升级三方库，另一方面要多熟悉和多尝试三方库的API，了解其原理。比如go版本对gc和timer的优化，gorm v2 引入的预编译等。

2.核心第三方库的大版本升级时的发布日志中的任意一个小点都不要忽略，最好都试一下，可能的话深入了解下这些点的原理，毕竟值得写入发布日志都不是小东西。

附录

区分 提升gc能力 和 通过减少分配的对象进而减少gc次数 是两种不同的思路。

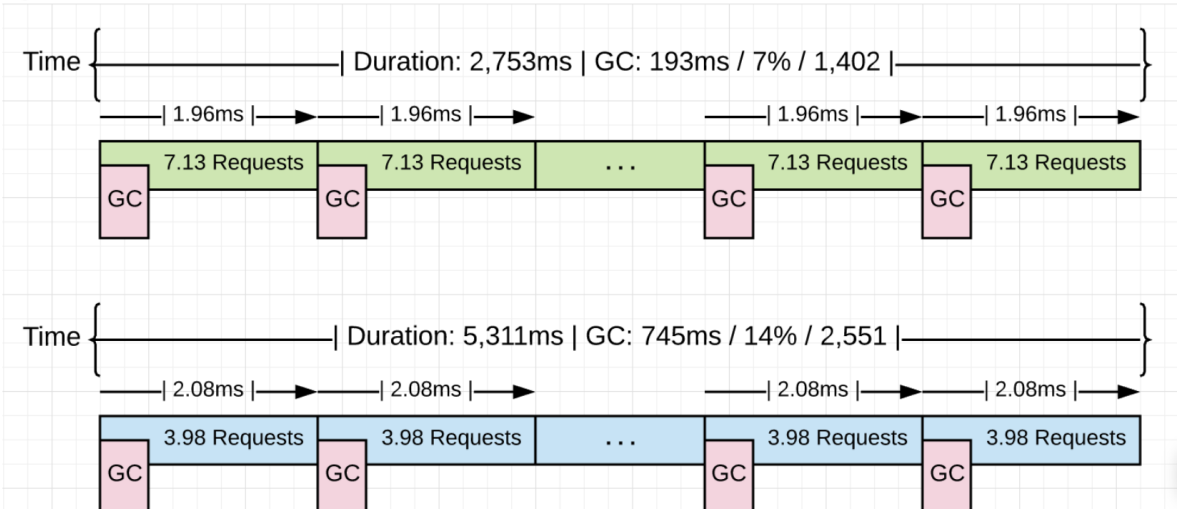
一般而言只有升级go版本才能提升gc的能力。

10w次请求，产生垃圾w。

优化后。

20w次请求，才产生垃圾w。

Figure 5



https://www.ardanlabs.com/blog/2018/12/garbage-collection-in-go-part1-semantics.html

类比法理解上图。

单个操作对比	服务处理请求能力	餐馆接客能力
	单位时间的请求处理数量	单位时间内接客数量
	gc能力	吃完后收拾餐桌的能力
	gc时间	吃完后收拾餐桌的时间
	stw	餐桌没收拾完，客人得等着
场景综合对比	减少产生垃圾的速度，进而减少gc触发的次数，减少gc导致的cpu消耗和stw,提高程序的吞吐量。 注意，这种方法并没有提高gc的能力。	减少产生垃圾的速度(吃饭垃圾吐盘子里面), 进而减少gc触发的次数(收拾餐桌的步骤精简为收盘子，收餐者的体力 减少gc导致的cpu消耗和stw, 提高程序的吞吐量(接客数量)，节约出来的时间累计多了就可以凑一桌多了。
	提高gc能力。 优化gc算法，使得回收同等数量的垃圾耗时更少。	换了个速度更快的收餐者，或者更好的收餐步骤，使得回收同等数量的

无标签