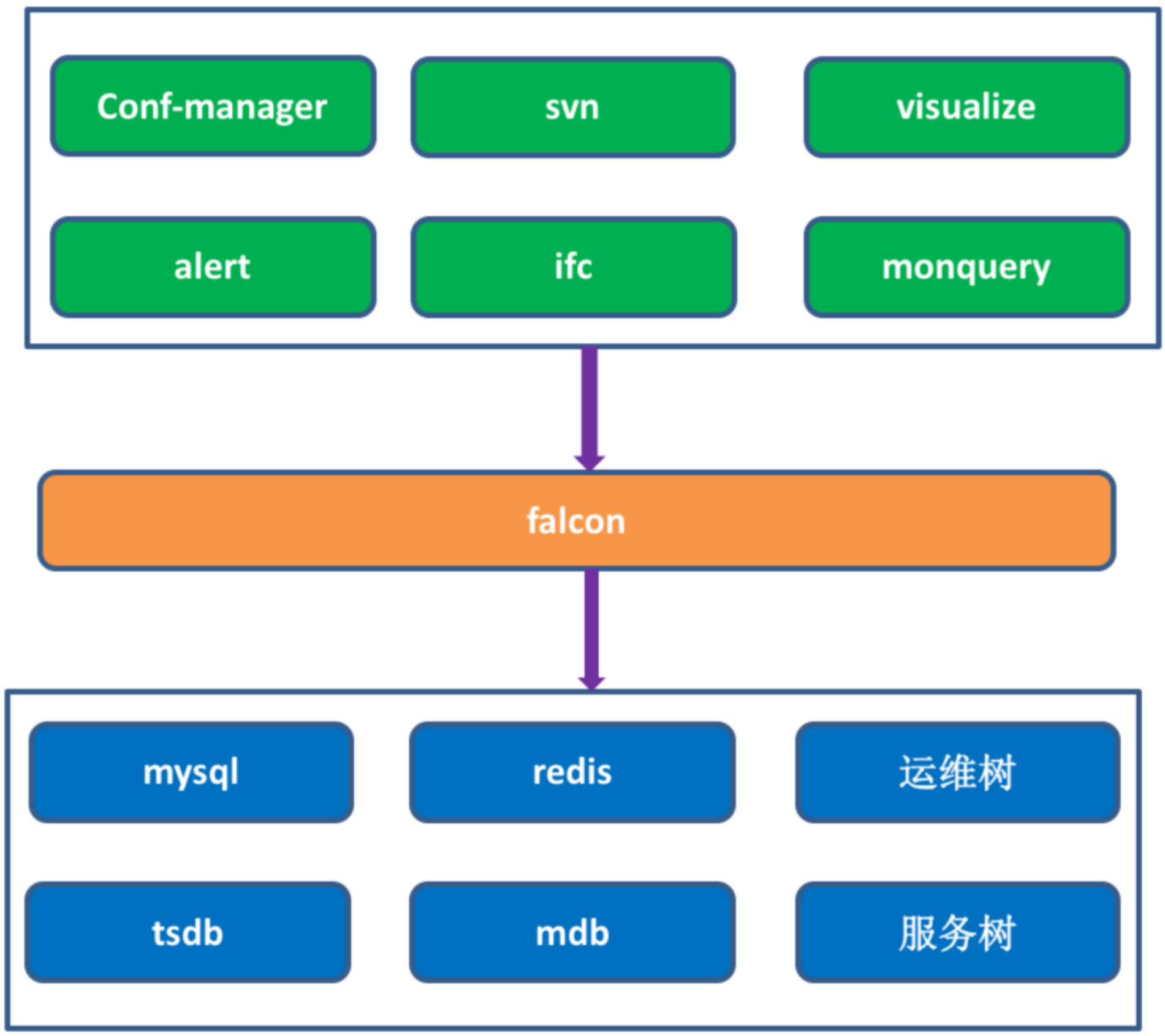
falcon golang GC 问题分析

目录

背景

在Argus中，falcon模块作为中间层为上层业务端提供了统一的查询接口，包括监控项、监控数据、权限、ans的关联关系数据缓存等。可参考下面的架构图：



近期有个别用户反馈argus的warningStatus接口偶尔出现时延过大的情况,例如：

问题分析

接口首先访问argus业务端的api模块，api模块访问falcon取得数据后返回结果。falcon会首先访问数据库之后访问 `warning-tracker` 获取数据。

首先确定是否为argus业务端的api模块的问题：api模块除访问falcon外没有其他处理逻辑，访问falcon的接口后发现问题复现，排除api模块问题。

确定是否为某台机器的问题：访问域名时，问题偶尔复现，当指定某台机器后发现，多次访问出现时延增高的问题稳定复现，且复现的频率大约为30s一次，卡顿一次后恢复正常。为确定是否是数据库或者warning-tracker出现问题，尝试打印日志分析接口的返回时延，返现各个对其他模块的访问时间正常，排除其他模块的问题。

对falcon自身进行排查后发现，falcon使用的revel框架，可以配置定时任务，而falcon的定时任务配置中有一项30s一次的 `gc` 定时任务（任务调用

为 `runtime.GC()` ），初步怀疑是定时的GC导致此问题。更改定时任务的时间间隔后发现时延出现问题频率随gc频率变化。定位为golang的gc问题。

问题解决

由于golang的gc一直是饱受诟病的（特别是早期版本），golang从1.5版本后对gc进行了大幅的改进（详情见下），1.6又对gc进行了改进。

golang gc历史

golang1.3 之前的gc比较简单，使用Mark&Sweep算法，当gc时，整个程序stop进行gc。

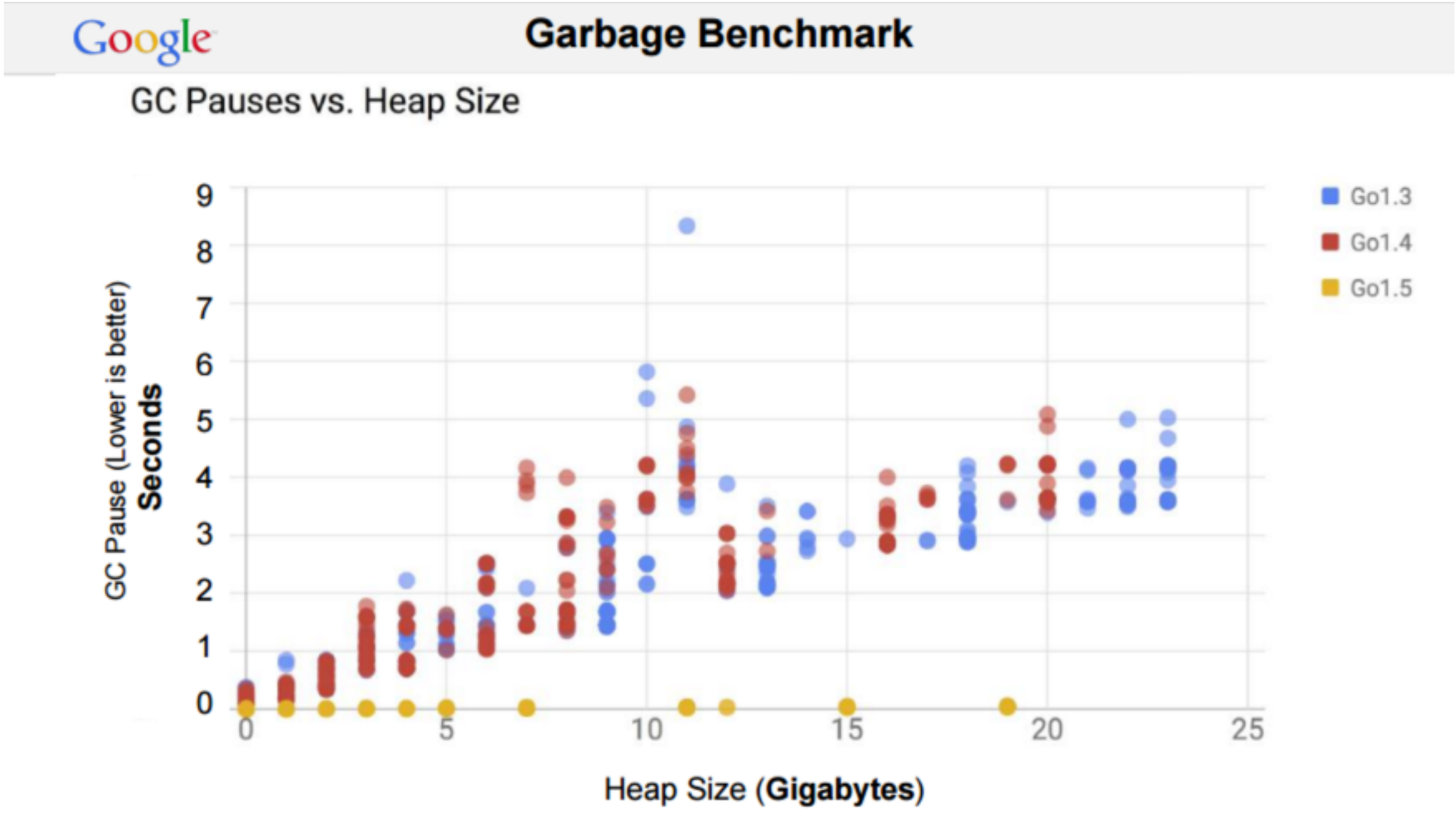
golang1.3 将Mark与Sweep的过程分开，可以在后台进行sweep操作。

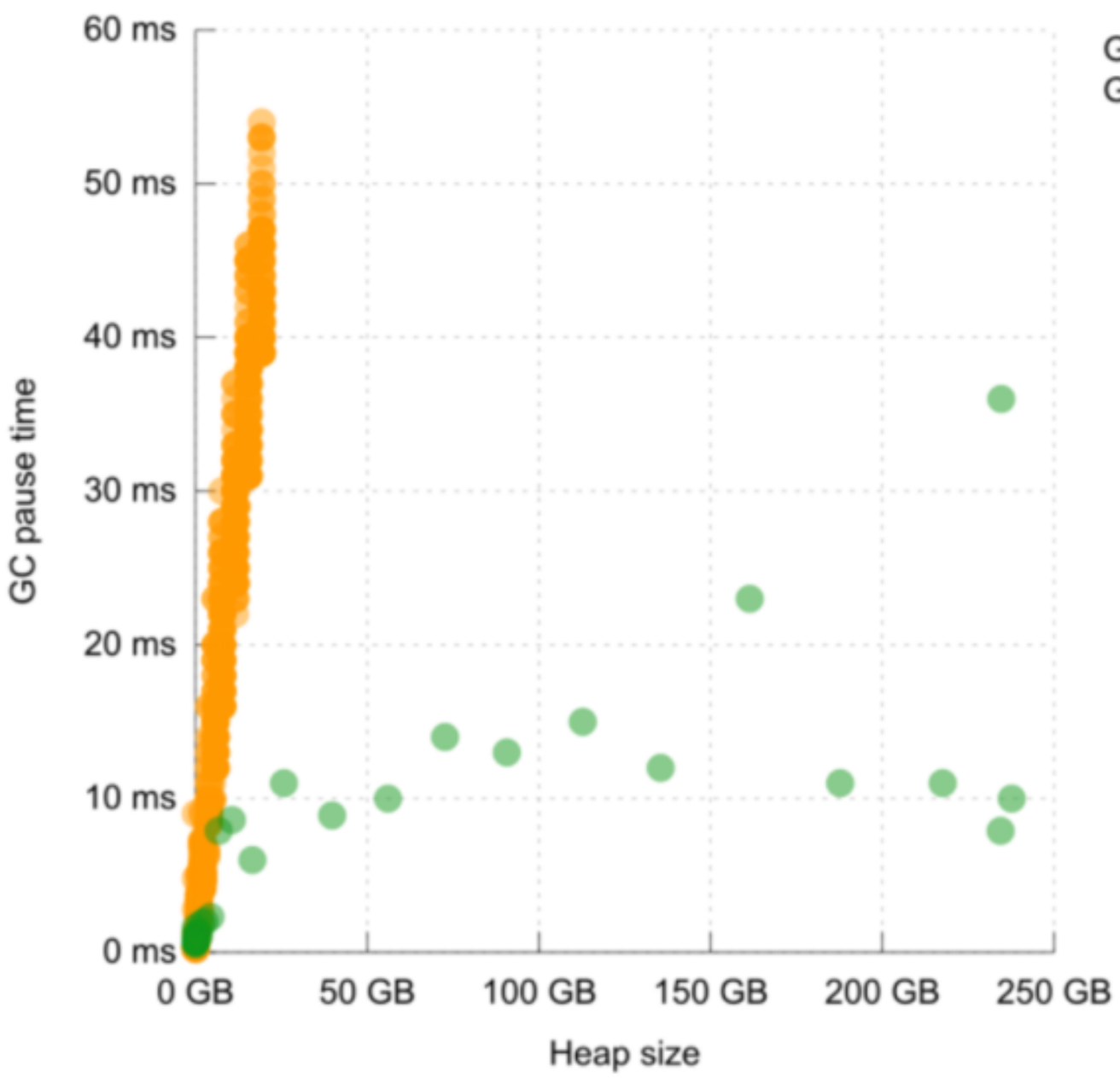
golang1.4 与golang1.3区别不大。

golang1.5 引入三色标记、并发标记。

golang1.6 没有大的算法改进,主要是对算法的优化。

下面是golang 各个版本的gc时间对比：





可以看到从1.4-1.6的gc停顿时间减小了很多。所以将Golang版本升级作为解决办法。

Golang升级为1.6.2版本后，发现接口的时延增大问题仍然很严重。所以对gc定时任务的执行时间做了统计，统计方法如下：

- gc定时任务会将执行时间打印到日志中
- 在机器中部署1.4版本的falcon，运行1h，期间不进行访问，统计gc定时任务的执行时间
- 在同一台机器中部署1.6版本的falcon，运行1h，期间不进行访问，统计gc定时任务的执行时间

按照上面的方法统计后，得到结果如下：

```
go 1.4: [end avgTime = 5.86666 ],times= 120
go 1.6: [end avgTime = 6.58423 ],times= 120
```

可以看到，在运行了120次定时gc期间，平均的程序停顿时间基本没有变化，甚至1.6要长一些。

升级golang版本没有达到预期效果

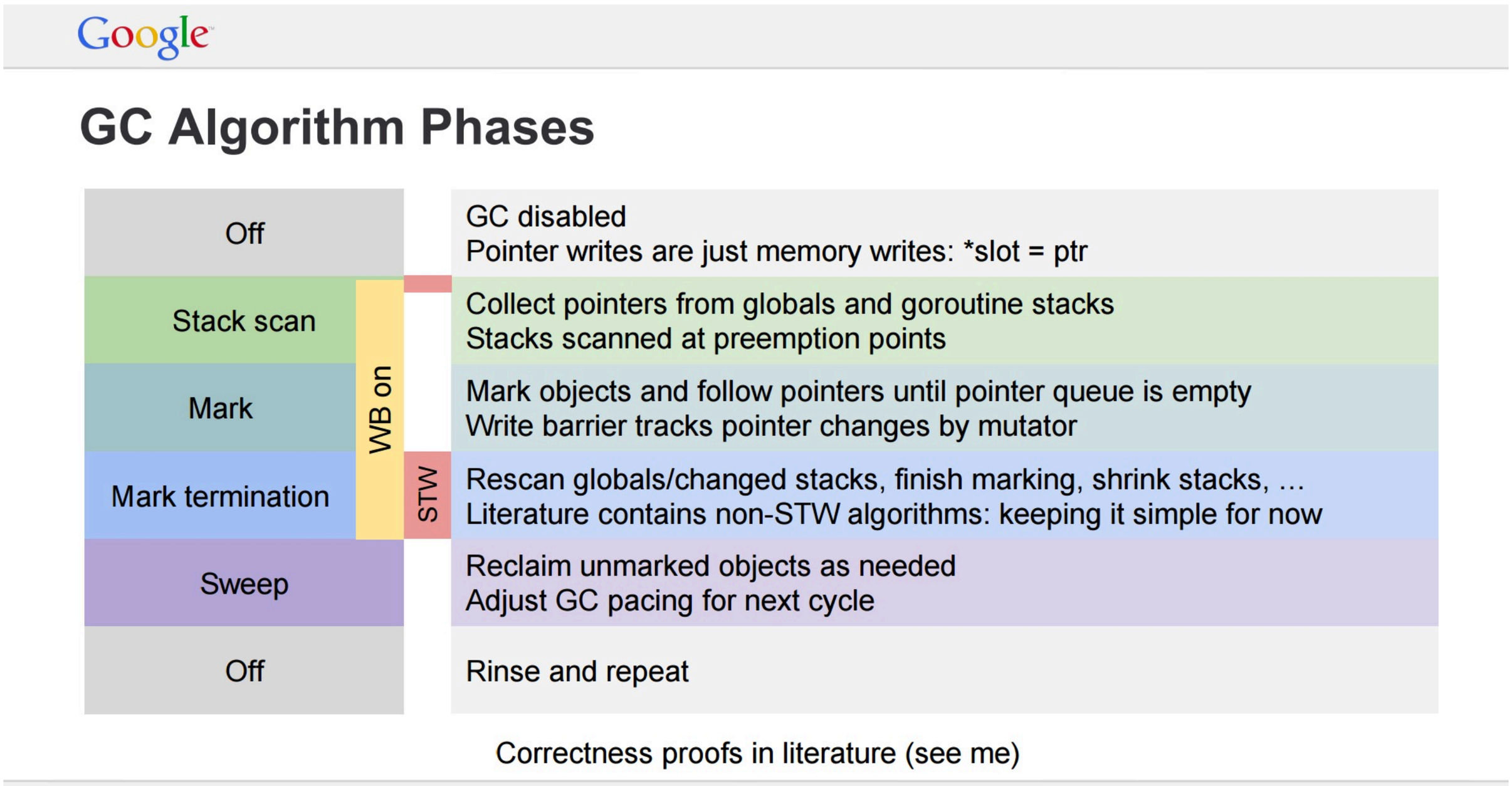
对配置进行修改

由于升级版本没有达到预期效果，怀疑是否是 `runtime.GC` 的全阶段STW导致的。

gc算法简介

go语言垃圾回收总体采用的是经典的mark and sweep算法。go15开始引进基于三色标记的、并行、mark-sweep算法。

golang 1.6的GC流程如下：



其中mark阶段是并行进行的，所以较1.4的gc相比提升很大。但是当手动使用 `runtime.GC` 调用时，并不会进行并行的标记，源码流程如下：

```Go
if mode == gcBackgroundMode { // Do as much work concurrently as possible
    setGCPhase(_GCmark)

    // markrootSpans uses work.spans, so make sure
    // it is up to date.
    gcCopySpans()

    gcBgMarkPrepare() // Must happen before assist enable.
    gcMarkRootPrepare()
} else {
        t := nanotime()
        work.tMark, work.tMarkTerm = t, t
        work.heapGoal = work.heap0

        // Perform mark termination. This will restart the world.
        gcMarkTermination()
}
```

GC执行的时机

- 在当前堆大小与上次回收结束的堆大小的比率达到 `$GOGC` 时,会触发GC
- 同时在监控服务sysmon中，每隔2分钟会检查一次垃圾回收状态，如果两分钟内未执行GC，则会强制进行一次GC。这种定期检查的原因主要是防止只设置GC比例

时，如果上次gc时有大量的内存分配，会将下次的GC阈值提升到一个很高的比例，导致垃圾回收久久无法被触发，造成隐性的内存泄漏。
- 通过 `runtime.GC` 手动强制触发GC

猜想验证

为了验证这个猜想进行了如下实验：

> 将30s的定时gc任务去掉后，golang本身每2min会检测期间是否gc，如果没有gc，则会强制进行一次gc。与 `runtime.GC` 不同的是，此时的gc并不会全阶段STW。 实验后发现，仍会出现2min一次的接口时延增高问题，猜想不成立。

打印GC日志

从上面的实验得知，应该是GC中STW的阶段花费时间过长导致此问题。首先对Golang的gc过程进行分析。分析gc过程可以根据Golang GC中整理的方法打印日志，即

```
export GODEBUG=gctrace=2
./go_binary 2>gc.log
```

设置后，go在gc后会打印gc信息到stderr，然后将stderr重定向到gc.log。

go 1.4 日志

程序运行稳定后，截取日志如下：

```
gc130(12): 1+55+3087108+206 us, 917 -> 917 MB, 14053746 (129167607-115113861) objects, 384 goroutines, 133293/0/128769 sweeps, 37280(1970
```

可以看到，时间在3s左右，对象数目在1400W左右，但是此处的时间与gc定时任务的总时间不对应，且相差较多，此处暂无法解释。

go 1.6 日志

```
gc 25 @625.450s 8%: 0.11+4392+7828 ms clock, 0.47+388/4391/8973+31314 ms cpu, 1726->1733->817 MB, 1853 MB goal, 4 P

gc 13 @203.441s 80%: 0.005+0+216472 ms clock, 0.021+18297/21367/42665+865889 ms cpu, 4352->4352->4125 MB, 4352 MB goal, 4 P (forced)
```

1.6 gc日志的时间与gc定时任务日志的时间相差不大，可以看到 `mark-termination` 的时间较长。

日志排查总结

通过日志没有分析到太多有用的信息，能够看到的现象有：

1. go 1.4 的日志与gc同步任务的时间不同，无法解释
2. 对象数目庞大，在1400W左右
3. go 1.6 的 `mark-termination` 时间很长，而这部分是一定会STW的

初步怀疑是程序的代码中除了问题，如现象2、3中看到的，可能是某处代码频繁的创建对象，或者是频繁的丢弃对象（ `mark-termination` 是并发标记期间对发生变化的对象重新标记的阶段，会STW）。

下一步对程序的代码进行分析。

使用pprof分析

分析程序的代码，可以打印函数调用的关系图进行分析。

使用原生的pprof包进行分析（revel也有pprof模块，是对pprof进行了一层包装，此处没有使用），具体步骤如下：

在init.go中，加入以下内容：

```Go
package app
import (
    "argus/app/filters"
    _ "argus/app/models/cron"
    "github.com/robfig/revel"
    "math/rand"
    "runtime"
    "time"
    "code.google.com/p/glog"

    //add
    "net/http"
    _ "net/http/pprof"
    //end add
)
func init() {
    go func() {
        http.ListenAndServe("0.0.0.0:9090", nil)
    }()
}

```

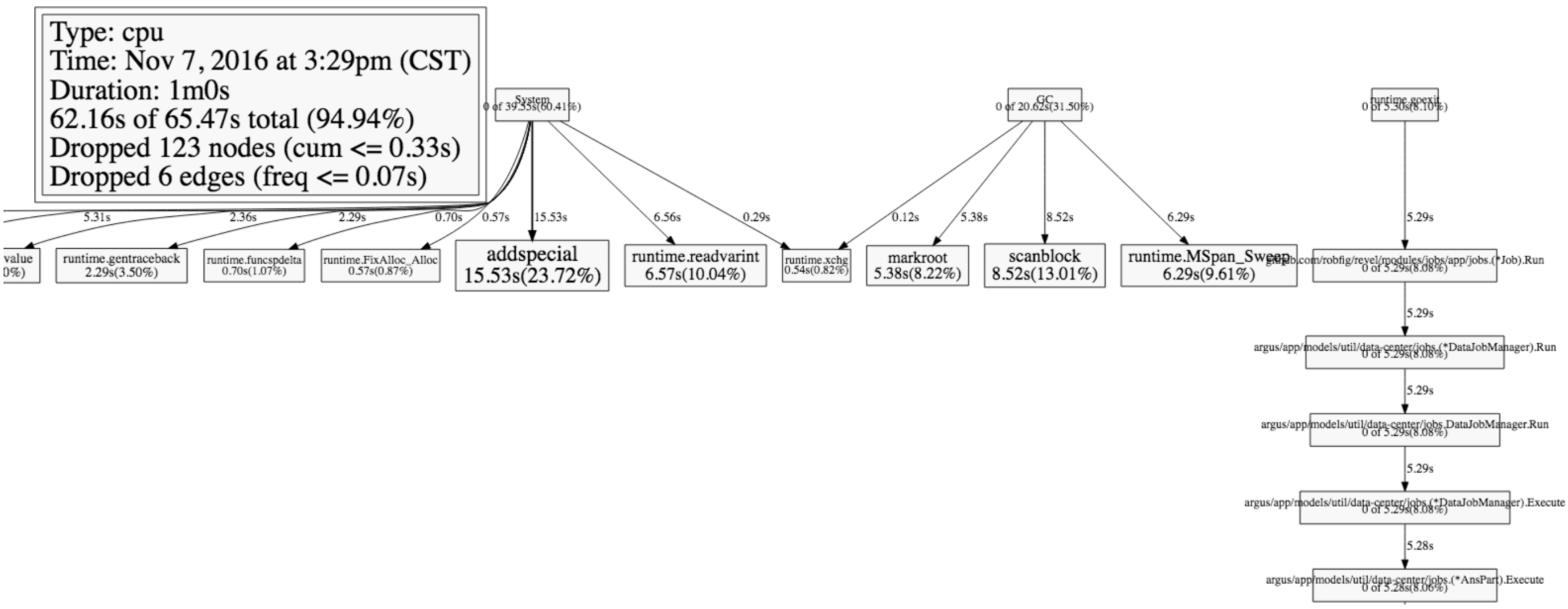
启动程序后，使用

```
go tool pprof localhost:9090/debug/pprof/profile
```

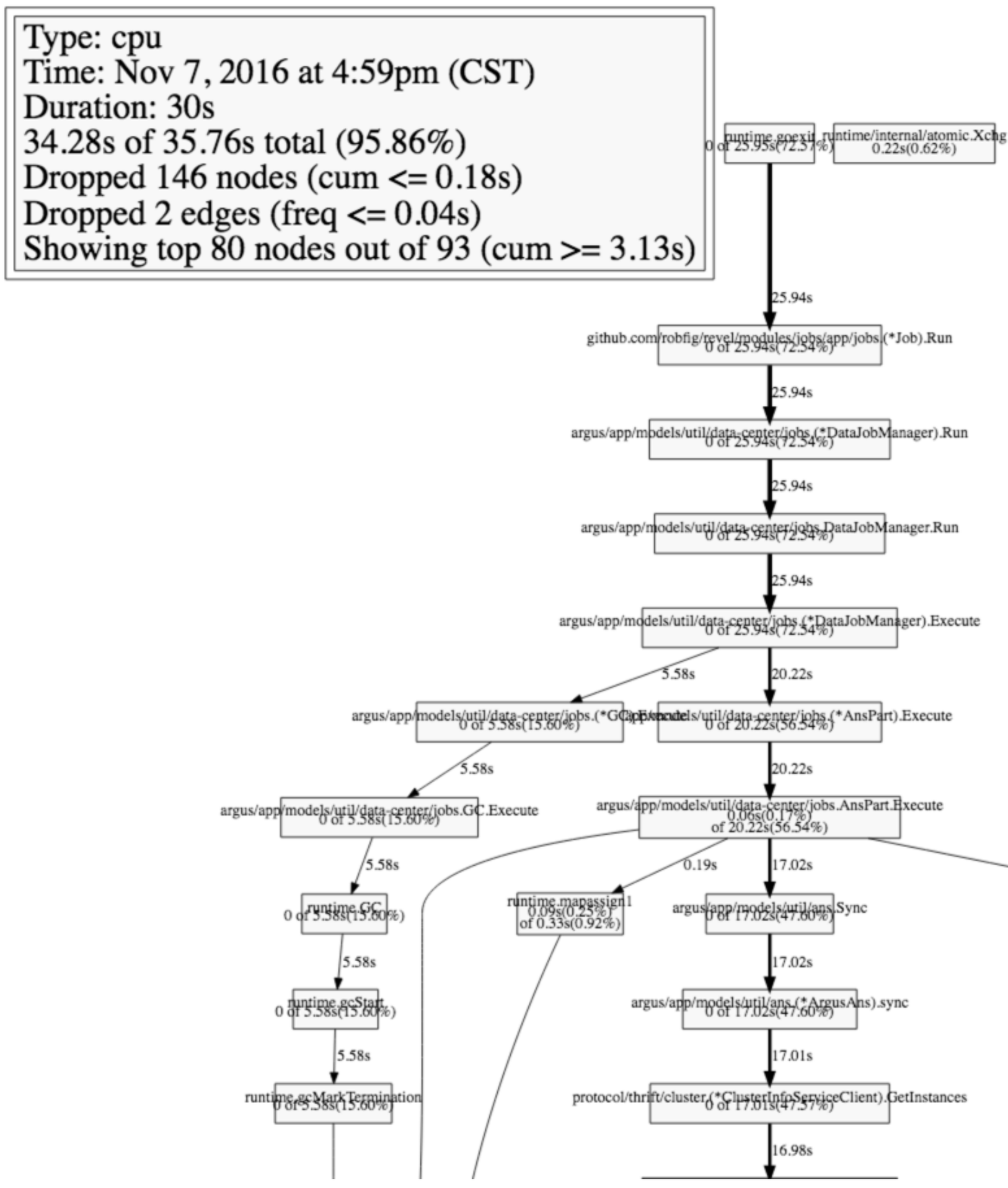
进行分析,执行命令后，会等待30s采集数据。如果进行自定义时间的采集的话，可以指定 `-seconds=N` 参数。

执行完后，可以使用 `web` 命令画出函数间的调用关系图。

1.4的调用关系图如下（大小有限，图片不完整：



1.6的调用关系图如下（大小有限，图片不完整）：



> 说明：pprof的原理为golang以1s中100次的速率采样 第一行为local time，第二行为cum time，第一行为采样点落在该函数中的时间，第二行为该函数的时间及被他调用的函数时间之和

可以看出，GC占了相当的时间，其次是Ans的同步任务占用了较多时间，（addspecial是mheap的函数，但是没有查到哪些操作会调用这个函数）。1.6比1.4的gc时间略

微短一些。

由于ans同步任务一直在执行，怀疑是否是ans缓存导致的gc缓慢。

下面先介绍下ans缓存的基本逻辑：

falcon会每8s从ans中进行一次增量同步，每1h进行一次全量的同步。同步的内容包括：cluster/service/instance/host之间的关联关系。

例如：存储service关联关系时，会用到如下的数据结构：

```
type ServiceInfo struct {
    Service string
    Timestamp time.Time
    Instances compare.Items
    Hosts compare.Items
    Clusters compare.Items
}
```

结合Golang 1.4 gc的日志中的对象数

```
14053746 (129167607-115113861) objects
```

初步判定gc时间过长的原因为：ans缓存导致对象数量太多，对象总量为1400W左右。

结合pprof中的ans同步任务，将anspart同步任务的日志做了一次统计。

> 取线上任意一台机器，进行取样，统计平均时间

得到结果如下：

```
[end avgTime = 252.94] times=947
```

947次取样，anspart定时任务平均执行时间253s，而falcon在revel中配置了8s一次anspart任务，导致anspart一直执行，会频繁产生垃圾对象。

为了验证是否是ans的问题，手动去掉ans缓存，GC时间在ms级，一切正常（GC正常，但是falcon无法查询到关联关系）

（对象数目对gc时间的验证在扩展中可以看到）

结论

ans缓存了大量的关联关系，这些关联关系导致生成了大量的对象，并且由于ans 同步任务每（8s 增量，1h全量）执行，产生了大量的临时对象，导致GC压力过大。

runtime.GC 与Golang 的普通gc不同，会全阶段 STW。

当业务量不大，缓存对象不多时，不需要过多关注GC。

扩展

根据这段时间的问题分析经验及搜索的结果，下面分享几个实验及常用的优化方法，以减少gc的停顿时间。

主要包括：

1. 减少对象数目
2. 减少临时对象的创建
3. 重用对象
4. 手动管理内存

实验

下面看一个实验：

数据定义：

```go
const TOTAL = 2000000;
type basicInfo struct {
    name string
    age int
}
type interests struct {
    interest int
    next *interests
}
type user struct {
    userBasicInfo *basicInfo
    userInterests *interests
}
type userUpdate struct {
    userBasicInfo *basicInfo
    userInterests *interestsUpdate
}
type interestsUpdate struct {
    interests []int
}
```

代码1：

```go
func run() {
    users := []user{}
    for seq := 0; seq < 100; seq++ {
        userBasicInfo := basicInfo{"name", 10}
        linkedInterest := &interests{1, nil}
        firstInterest := &interests{1, linkedInterest}
        for i := 0; i < TOTAL; i++ {
            temp := &interests{1, nil}
            linkedInterest.next = temp
            linkedInterest = temp
        }
        someone := user{&userBasicInfo, firstInterest}
        users = append(users, someone)
    }
    fmt.Print("alloc finished\n")
    start := time.Now().Unix()
    runtime.GC()
    end := time.Now().Unix()
    fmt.Print(end - start)
    fmt.Printf(strconv.Itoa(users[0].userBasicInfo.age))
}
```

代码片段2：

```go
func run() {
    users := []userUpdate{}
    for seq := 0; seq < 100; seq++ {
        userBasicInfo := basicInfo{"name", 10}
        firstInterest := &interestsUpdate{make([]int, TOTAL+10)}
        for i := 0; i < TOTAL; i++ {
            firstInterest.interests[i] = i;
        }
        someone := userUpdate{&userBasicInfo, firstInterest}
        users = append(users, someone)
    }
    fmt.Print("alloc finished\n")
    start := time.Now().Unix()
    runtime.GC()
    end := time.Now().Unix()
    fmt.Print(end - start)
    fmt.Printf(strconv.Itoa(users[0].userBasicInfo.age))
}
```

最后的 `runtime.GC()` 时间对比：

1. `gc 11 @19.457s 65%: 0.002+0+11198 ms clock, 0.009+1251/2613/5078+44795 ms cpu, 3051->3051->3051 MB, 3051 MB goal, 4 P (force`
2. `gc 10 @1.510s 3%: 0.002+0+0.18 ms clock, 0.011+0/0.076/0.10+0.74 ms cpu, 1526->1526->1526 MB, 1526 MB goal, 4 P (forced)`

结论：

- 对象数量过多时，会显著影响gc的时间。
- 尽量少的创建对象,使用slice减少对象的数量（存储在slice中的结构体被认为是一个对象，slice中需要存储结构体，而不是结构体指针，同时结构体中不能包含指针），对象数目与GC停顿时间关系很大。

减少对象分配的几个技巧

所谓减少对象的分配，实际上是尽量做到，对象的重用。 比如像如下的两个函数定义，实现从Reader中读取数据，

```
func(r*Reader)Read()([]byte,error)
func(r*Reader)Read(buf[]byte)(int,error)
```

第一个函数没有形参，每次调用的时候返回一个[]byte，第二个函数在每次调用的时候，形参是一个buf []byte 类型的对象，之后返回读入的byte的数目。

第一个函数在每次调用的时候都会分配一段空间，这会给gc造成额外的压力。第二个函数在每次迪调用的时候，会重用形参声明。

> http://wangzhezhe.github.io/blog/2016/04/30/golang-gc/

string用slice拼接

```
s = strings.Join([]string{s, "[", v, "]"}, "")
```

append

```
b := make([]int, 1024)
fmt.Printf("%p\n", b);
fmt.Println("len:", len(b), "cap:", cap(b))
b = append(b, 99)
fmt.Printf("%p\n", b);
fmt.Println("len:", len(b), "cap:", cap(b))
```

结果：

```
0xc820072000
len: 1024 cap: 1024
0xc820076000
len: 1025 cap: 1312
```

append无论如何都会向slice的尾部追加数据

在使用了append操作之后，数组的空间由1024增长到了1312，所以如果能提前知道数组的长度的话，最好在最初分配空间的时候就做好空间规划操作，会增加一些代码管理的成本，同时也会降低gc的压力，提升代码的效率。

sync.Pool

重用对象

cgo

也可以使用cgo的方法自己管理内存。