*Golang*创业三年的收获

个人经历简介

- 庾俊 17年北漂，目前深飘
- 2014-2017，北京火翼科技，权利之歌，末日世界项目
- 2011-2014年，美国 Kabam 北京公司，Thirst of night， Dragon of Atlantis 项目

Kabam

亚特兰蒂斯之龙：龙族崛起 Dragons of Atlantis：

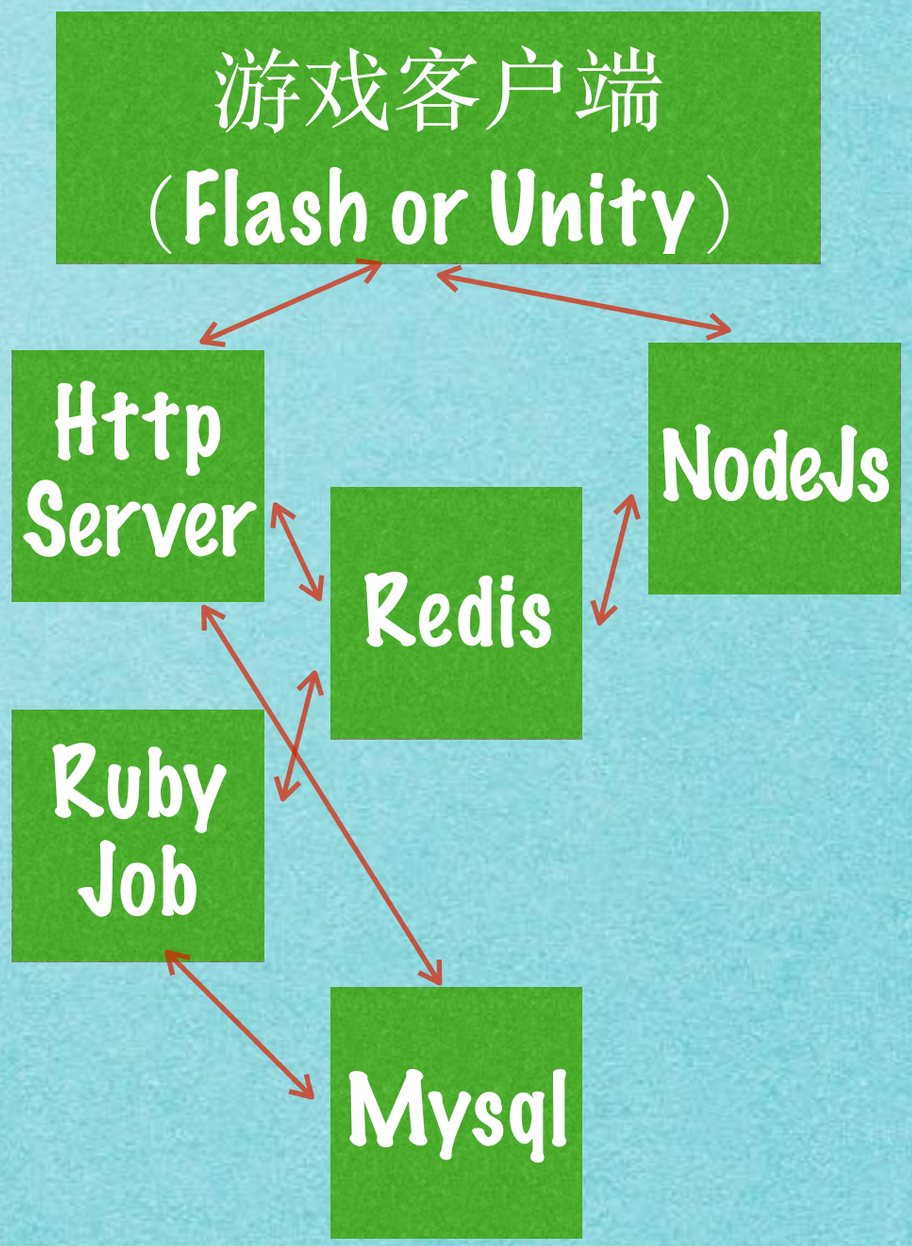
创业项目


权力之歌
已使用此时账户登入。为保存您在游戏内的进度，请前
往设备设置页中并使用 Google 账号登入。
正在加载存储数据.....


末日世界
末世战争
危机来袭

页游时代遗留下来的架构

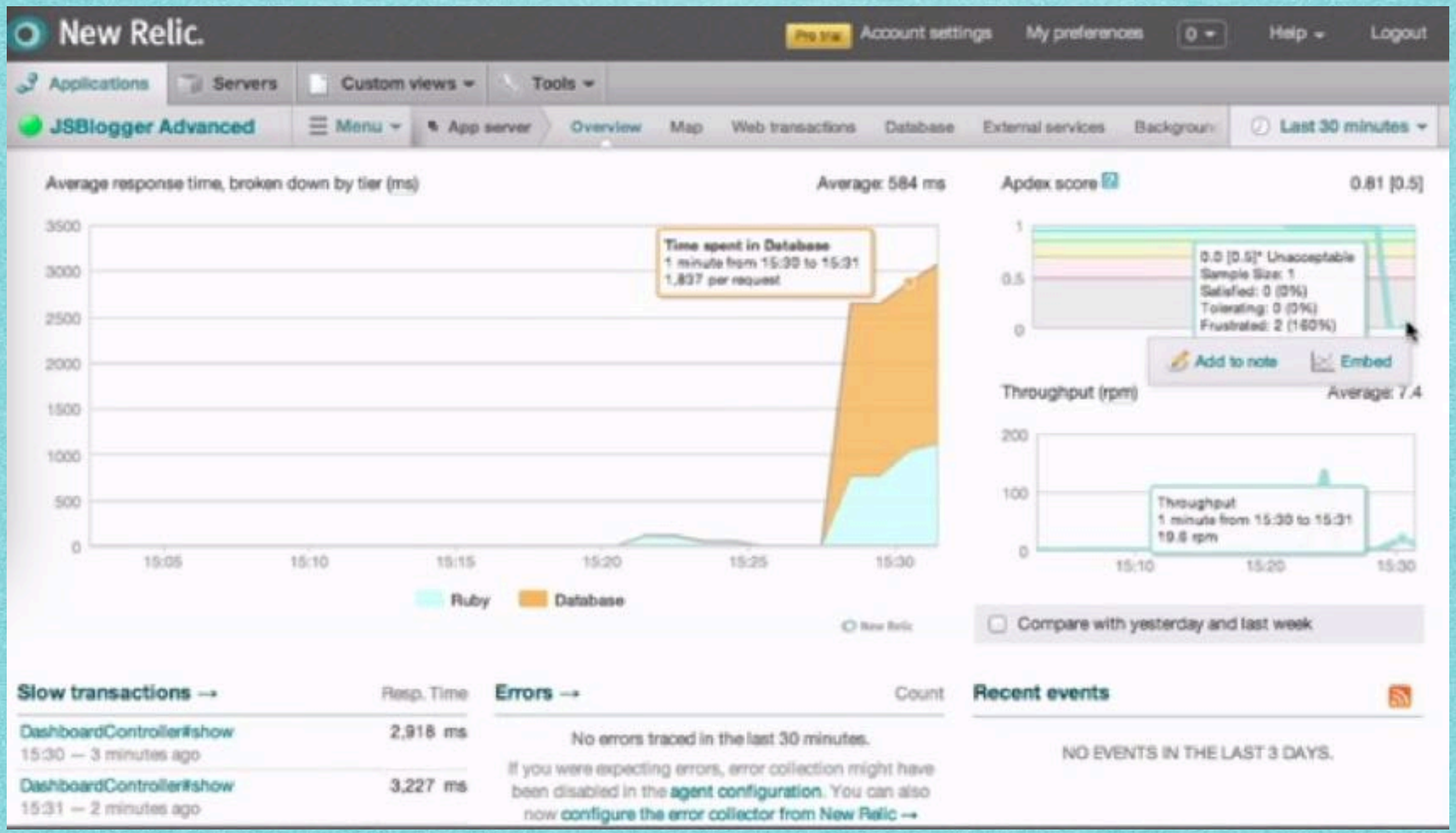
- Ruby On Rails
- 聊天服务器：Nodejs
- Job服务器: Ruby
- 缓存，消息转发：Redis



从过去项目学到的一些经验

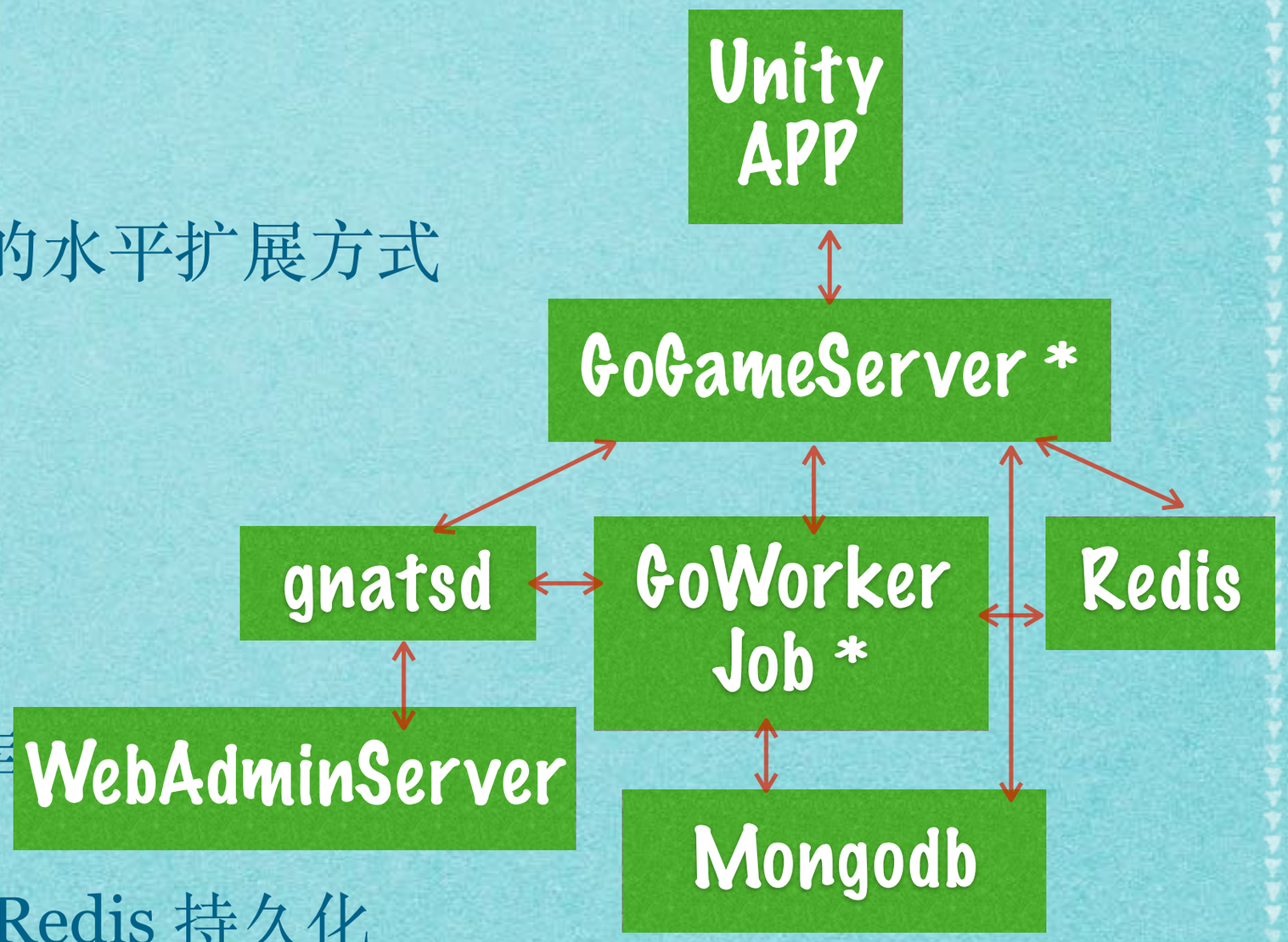
- 动态语言的大规模重构危险，需要有全面自动化测试的基础
- HTTP 已经跟不上 SLG 手游发展的趋势
  - 数据包太大
  - 即时性差
  - 响应相对慢
- 可水平扩展的游戏架构
- 早期 Redis 的 AOF 持久化的 rewrite 会阻塞 IO
- 在线游戏的 Mysql 数据结构的变更维护有点痛苦

New Relic


New Relic.
Account settings
My preferences
Help
Logout
Applications
Servers
Custom views
Tools
JSBlogger Advanced
Menu
App server
Overview
Map
Web transactions
Database
External services
Last 30 minutes
Average response time, broken down by tier (ms)
Average: 584 ms
Time spent in Database
1 minute from 15:30 to 15:31
1,837 per request
3500
3000
2500
2000
1500
1000
500
0
15:05
15:10
15:15
15:20
15:25
15:30
Ruby
Database
Apdex score
0.81 [0.5]
0.0 [0.5]* Unacceptable
Sample Size: 1
Satisfied: 0 (0%)
Tolerating: 0 (0%)
Frustrated: 2 (100%)
Add to note
Embed
Throughput (rpm)
Average: 7.4
Throughput
1 minute from 15:30 to 15:31
19.8 rpm
200
100
0
15:10
15:20
15:30
Compare with yesterday and last week
Slow transactions →
Resp. Time
DashboardController#show
15:30 — 3 minutes ago
2,918 ms
DashboardController#show
15:31 — 2 minutes ago
3,227 ms
Errors →
Count
No errors traced in the last 30 minutes.
If you were expecting errors, error collection might have been disabled in the agent configuration. You can also now configure the error collector from New Relic →
Recent events
NO EVENTS IN THE LAST 3 DAYS.

新架构设计时的一些选择

- 静态语言 Golang
- 和过去架构类似的水平扩展方式
- Tcp 长连接
- Protobuf 数据包
- MongoDB 数据库
- 尽可能不再使用 Redis 持久化



分析性能的预备

type	name	min	max	total	count
LPUB_MSV	MsGetCurrentOnlinePlayersData	27.970628	27.97062	27.97062	1
LPUB_MSV	lpub	27.970628	27.97062	27.97062	1
MGO	bi_current_online_players:Upsert	2.530017	2.530017	2.530017	1
MGO	bi_current_online_players:Upsert	2.454645	2.454645	2.454645	1
LPUB_MSV	lpub	22.315607	22.3156(	22.31560	1
LPUB_MSV	MsGetCurrentOnlinePlayersData	22.315607	22.3156(	22.31560	1
MGO	ban_to_post:Find_{}	0.224007	0.224007	0.224007	1
MGO	version:Find_{}	0.220204	0.220204	0.220204	1
REDIS	GET_realms:_id	0.119695	21.8607	118.9297	320
REDIS	H_CREATE_USER:GET	0.119695	21.8607	118.9297	320
Requests	requests	0.216959	79.4197	3394.341	1734
Requests	H_GET_CLIENT_RANDOM_KEY	0.216959	23.7484	437.7675	640
Requests	H_DEVICE_REG	2.142529	79.4197	2270.031	574
Requests	H_CREATE_USER	0.672335	25.1115	457.5279	320

保存panic信息

- 所有 panic 信息会被保存到数据库里面，将信息的MD5作为主键，按日期记录次数

- 所有客户端的异常也会被收集上报，同样保存到数据库里

创业初期踩的坑

- 自备梯子
- package import cycle not allowed
- 给每个go routine 加个 recover
- 早期第三方的组件会有 bug，或者功能不全
- 有些API文档描述的比较简略，必须阅读源码才能

For range 里的相同指针

```go
func main() {
    jackie := Dog{
        Name: "Jackie",
        Age:  19,
    }

    fmt.Printf("Jackie Addr: %p\n", &jackie)

    sammy := Dog{
        Name: "Sammy",
        Age:  10,
    }

    fmt.Printf("Sammy Addr: %p\n", &sammy)

    dogs := []Dog{jackie, sammy}

    fmt.Println("")

    for _, dog := range dogs {
        fmt.Printf("Name: %s Age: %d\n", dog.Name, dog.Age)
        fmt.Printf("Addr: %p\n", &dog)

        fmt.Println("")
    }
}
```

For range 里的相同指针

```
$ go run forRangePointer.go
Jackie Addr: 0xc42000a060
Sammy Addr: 0xc42000a080

Name: Jackie Age: 19
Addr: 0xc42000a0a0

Name: Sammy Age: 10
Addr: 0xc42000a0a0
```

误解了 Select 的 Timeout

```
c1 := make(chan string, 1)
go func() {
    c1 <- "result 1"
}()

select {
case <-c1:
    time.Sleep(30 * time.Second)
    fmt.Println("done")
case <-time.After(time.Second * 1):
    fmt.Println("never timeout")
}
```

习惯了脚本语言带来的坑

```
var a uint8 = 1
c := -a
fmt.Println(a)
fmt.Println(reflect.TypeOf(c))
fmt.Println(c)
```

```
1
uint8
255
```

压力测试

- 查询性能还行，但是光靠Mongodb还是扛不住
- MongoDB的早期2.6版本写入性能比较差
- Sharding有风险，关键是坏了也不知道咋修

做的一些优化

- Mongodb分库
- 数据库中预创建用户
- 基于数据主键的Redis Cache
- 基于请求的热数据进程内缓存

权利游戏公测

- 跑满了 CPU，紧急加一倍的机器抗过压力
- Golang的静态编译使得运维轻松很多
- 被优化时加的一个正则匹配坑了

Benchmark

```
$ go test -bench=. -benchtime=20s
goos: darwin
goarch: amd64
pkg: goBenchmark
Benchmark_matchTimestamp-8                  5000000              5318 ns/op
Benchmark_fixedMatchTimestamp-8           100000000               244 ns/op
PASS
ok      goBenchmark     56.778s
```

优化留下的坑

```go
func matchTimestamp() {
    regMatchTimestamp, _ := regexp.Compile("^[0-9]{10}$")
    regMatchTimestamp.MatchString("100230200203302")

}

var fixedRegMatchTimestamp *regexp.Regexp

func init() {
    fixedRegMatchTimestamp, _ = regexp.Compile("^[0-9]{10}$")

}
func fixedMatchTimestamp() {
    fixedRegMatchTimestamp.MatchString("100230200203302")
}
```

末日世界公测

- 超了云主机的Mongodb主机的连接数限制
- CPU 还是占用高
- 游戏新手任务里的 for 循环

完全没想到

```
$ go test -v -bench=.
goos: darwin
goarch: amd64
pkg: goBenchmark/loopMapBenchmark
Benchmark_checkQuestsA-8                3000                506788 ns/op
Benchmark_checkQuestsB-8              100000                 16261 ns/op
PASS
ok      goBenchmark/loopMapBenchmark    3.396s
```

for 循环的性能差异

```go
var quests map[string]uint32
var keys []string

func init() {
    quests = make(map[string]uint32)
    keys = []string{}
    for i := 0; i < 30000; i++ {
        k := fmt.Sprintf("quests-%d", uint32(i))
        quests[k] = uint32(i)
        keys = append(keys, k)
    }
}

func checkQuestsA() {
    for k, _ := range quests {
        if k == "" {
        }
    }
}
func checkQuestsB() {

    for _, k := range keys {
        if k == "" {
        }
    }
}
```

总结

- Golang易学易用，但是如果要自己设计实现框架，把框架做稳定还是需要趟不少坑
- 虽然性能比动态语言要好，但是还是要经常做Benchmark测试