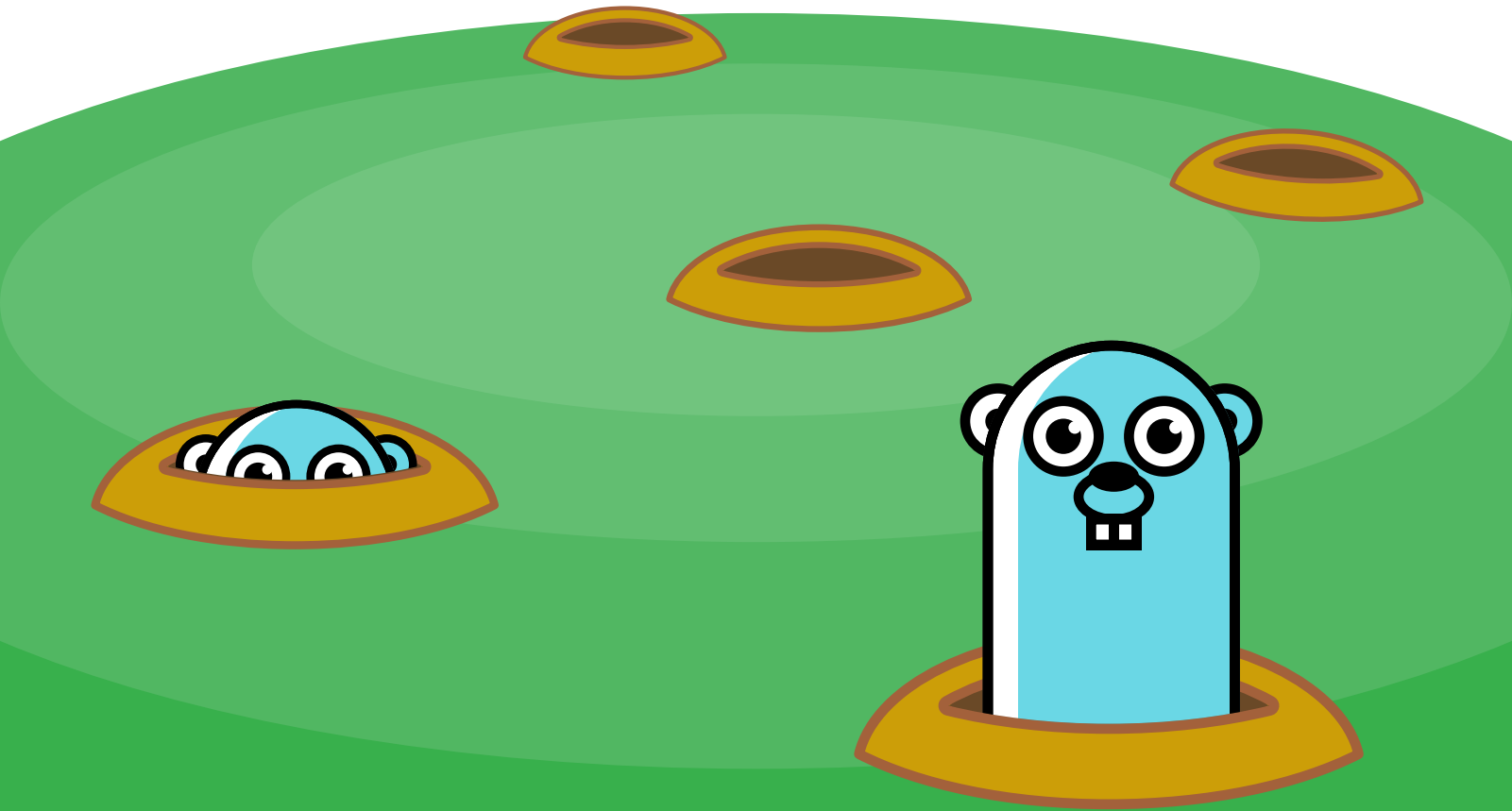
005

2016-03-31

架構 ARCHNOTES

高 可 用 架 构



LEARNING as we GO

Learning as we Go

如何用Go实现Web应用中的微服务

作者 / Jacob Martin

Jacob是一位波兰高中生，今年17岁。他自打记事开始就对IT技术很着迷，他最近的兴趣点集中在了用Scala和Go做开发上。他写博客的目的是为了给和他一样的人提供一个学习资源，他相信他和他的读者们都将是终身的学习者。

第一部分：设计

简介

“微服务”是最近经常出现的一个热词。你可以爱它，你也可以恨它，但你决不能无视它。在本文中，我们将用微服务架构创建一个Web应用。我们尽量不使用第三方工具和库。但是你需要知道，当你在生产环境中创建Web应用时，使用第三方库确实是个好办法（哪怕只是为了节省时间）。

我们会以基本形式创建各种组件。我们不会使用高级的caching或者database。我们会创建基本的健值存储（key-value store）以及一个简单的存储服务。整个过程我们都将使用Go来完成。

更新：本文只是为了展示一个微服务架构的可扩展工作框架。如果你只是想为照片增加滤镜，那就不要这样设计。杀鸡焉用宰牛刀。

如果进一步考虑，你确实需要这样设计架构。软件存活时间通常比我预想的要长，而这种设计会让我们得到一个可以轻松扩展的Web应用。

功能

首先我们需要决定我们的Web 应用能够做什么。我们将在本文中创建的应用可以从用户那里获取一张图片，然后返回一个独特的ID。图片将会被复杂且高度精细的算法所修改，比如交换红蓝通道，而用户可以使用ID查看照片是已经完成还是正在进行中。如果完成，他可以下载已经修改后的图片。

设计架构

我们想要架构成为微服务，所以我们也应该模仿微服务进行设计。我们必然需要一种面向用户的服务，该服务可以提供和应用通信的界面。这个服务还需要能够处理认证，并且把工作负载重新定向到正确的子服务上。（如果你想给应用集成更多功能的话，这么做是有用的。）

我们还需要一个能够处理我们所有图片的微服务。它可以获取图片，生成ID，存储和每个任务相关的信息，然后保存图片。为了处理高工作负载，我们可以使用主从（master-slave）系统来实现图片修改服务。处理图片的一方将是主（master），然后我们将创建从主站获得工作图片的从属（slave）微服务。

我们还需要一个用于各种配置的键值数据存储、一个用于保存修改前和修改后图片的存储系统，以及一个用于存放每个任务信息的类数据库服务。

这些应该足够我们开始了。

但是我还是要声明一下，如果需要的话，架构在这个过程中还是会改变。如果你觉得哪些地方可以提高也可以留言告诉我。

通信

我们需要定义服务之间的通信方式。在这个应用中我们将会在各个地方使用REST。你也可以使用消息总线（message BUS）或者远程过程调用（RPC），但是我这里就不具体说明了。

设计微服务API

另外一件重要的事就是设计你的微服务API。我们现在来逐个设计一下，顺便了解一下它们每个都是做什么用的。

键值存储

这个微服务主要是为了配置而构建的。它有一个简单的递交（post-get）界面：

- POST:
  - 参数：
    - 键
    - 值
  - 响应：
    - 成功/失败

- GET:
  - 参数:
    - 键
  - 响应:
    - 值 / 失败

存储

我们在这里存储图片，然后再次使用键值界面和参数说明来表明图片是修改前还是修改后。为了简洁起见，我们把图片保存在一个以图片状态（已完成 / 进行中）命名的文件夹中。

- POST:
  - 参数:
    - 键
    - 状态：修改前 / 后
    - 数据
  - 响应:
    - 成功 / 失败
- GET:
  - 参数:
    - 参数:
    - 键
    - 状态：修改前 / 后

- 响应：
  - 数据 / 失败

数据库

这个会保存我们的任务。如果它们等待启动、进行中，或已完成。

- POST:
  - 参数：
    - TaskId
    - 状态：尚未开始 / 进行中 / 已完成
  - 响应：
    - 成功 / 失败
- GET:
  - 参数：
    - TaskId
  - 响应：
    - 状态 / 失败
- GET:
  - 参数：
    - 尚未开始 / 进行中 / 已完成
  - 响应：
    - TaskId 列表

前端

前端的主要目的是提供不同服务和用户之间的通信方式。前端也可以用来认证和授权。

- POST:
  - 路径:
    - newImage
  - 参数:
    - 数据
  - 响应:
    - Id
- GET:
  - 路径:
    - 图片 /isReady
  - 参数:
    - Id
  - 响应:
    - 未找到 / 进行中 / 已完成
- GET:
  - 路径:
    - 图片 /get
  - 参数:
    - Id

- 响应：
  - 数据

图片主微服务

这个微服务将会从前端 / 用户获得新图片然后把图片发送到存储服务。它还会在数据库中创建新任务，协调可以请求任务的工人（worker）并且在任务完成后给予通知。

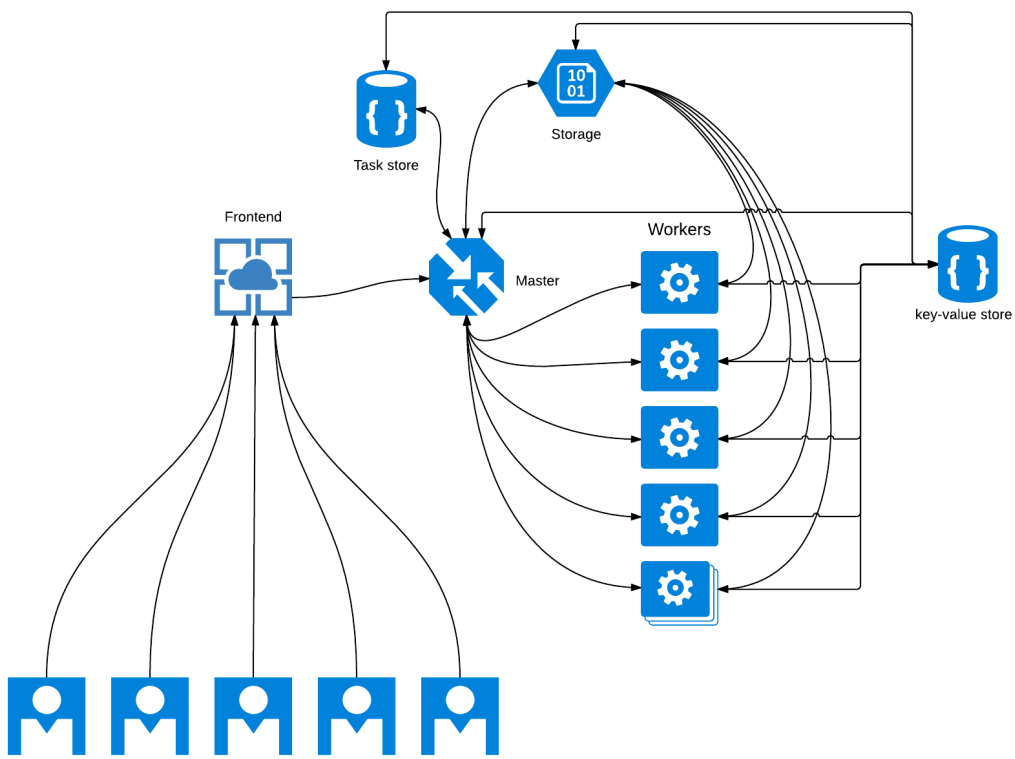
- 前端界面：
  - POST：
    - 路径：
      - newImage
    - 参数：
      - 数据
    - 响应：
      - Id
  - GET：
    - 路径：
      - isReady
    - 参数：
      - Id
    - 响应：
      - 未找到 / 进行中 / 已完成

    - GET:
        - 路径:
            - 获取
        - 参数:
            - Id
        - 响应:
            - 数据/失败
- 工人界面:
    - GET:
        - 路径:
            - getWork
        - 响应:
            - Id/noWorkToDo
    - POST:
        - 路径:
            - workFinished
        - 参数:
            - Id
        - 响应:
            - 成功/失败

图片工人微服务

这个微服务没有 API。它是主图片服务的客户端，它使用键值存储。该微服务从存储服务中获得需要处理的图片数据。

方案



第二部分：k/v存储和数据库

简介

在这部分中我们将实现Web应用所需要的一部分微服务。我们将要实现：

- 键值存储
- 数据库

接下来编码将占据很大部分，所以让我们集中注意力，一起来玩吧！

键值存储

设计

设计并没有改变太多。我们把键值对作为全局映射来保存，并且为并行存取创建一个全局互斥。我们还要增添为所有键值对列表的能力，为的是调试和分析。我们还需要添加删除已存在条目的能力。

首先，我们来创建一个结构：

```
package main

import (
    "net/http"
    "sync"
    "net/url"
    "fmt"
```

```
)

var keyValueStore map[string]string
var kVStoreMutex sync.RWMutex


func main() {
    keyValueStore = make(map[string]string)
    kVStoreMutex = sync.RWMutex{}
    http.HandleFunc("/get", get)
    http.HandleFunc("/set", set)
    http.HandleFunc("/remove", remove)
    http.HandleFunc("/list", list)
    http.ListenAndServe(":3000", nil)

}

func get(w http.ResponseWriter, r *http.Request) {
}

func set(w http.ResponseWriter, r *http.Request) {
}

func remove(w http.ResponseWriter, r *http.Request) {
}

func list(w http.ResponseWriter, r *http.Request) {
}
```

然后，我们再开始实现。首先，我们需要在get函数中增加参数解析然后核实关键参数是否正确。

```
func get(w http.ResponseWriter, r *http.Request) {
    if(r.Method == http.MethodGet) {
          values, err := url.ParseQuery(r.URL.RawQuery)
          if err != nil {
```

```
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, "Error:", err)
            return
        }
        if len(values.Get("key")) == 0 {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, "Error:","Wrong input key.")
            return
        }
    } else {
        w.WriteHeader(http.StatusBadRequest)
        fmt.Fprint(w, "Error: Only GET accepted.")
    }
}
```

key的长度不应该是0，所以需要长度检查。我们还需要检查方法是不是GET，如果不是的话，我们就需要输出它然后把状态码设置为错误请求（bad request）。在每条错误信息之前，我们用一个明确的Error作为回应，所以该信息不会被客户端误认为一个值。

现在，我们访问映射并返回一个回应：

```
if len(values.Get("key")) == 0 {
    w.WriteHeader(http.StatusBadRequest)
    fmt.Fprint(w, "Error:","Wrong input key.")
    return
    }

kVStoreMutex.RLock()
value := keyValueStore[string(values.Get("key"))] kVStoreMutex.RUnlock()
fmt.Fprint(w, value) 5.5
```

我们把值复制到变量中，所以我们在返回响应时不会阻塞映射。

现在我们来创建一个set函数，其实非常相似。

```
func set(w http.ResponseWriter, r *http.Request) {
    if(r.Method == http.MethodPost) {
       values, err := url.ParseQuery(r.URL.RawQuery)
       if err != nil {
           w.WriteHeader(http.StatusBadRequest)
           fmt.Fprint(w, "Error:", err)
           return
       }
       if len(values.Get("key")) == 0 {
           w.WriteHeader(http.StatusBadRequest)
           fmt.Fprint(w, "Error:", "Wrong input key.")
           return
       }
       if len(values.Get("value")) == 0 {
           w.WriteHeader(http.StatusBadRequest)
           fmt.Fprint(w, "Error:", "Wrong input value.")
           return
       }

       kVStoreMutex.Lock()
          keyValueStore[string(values.Get("key"))] = string(values.
Get("value"))
       kVStoreMutex.Unlock()

       fmt.Fprint(w, "success")
    } else {
       w.WriteHeader(http.StatusBadRequest)
       fmt.Fprint(w, "Error: Only POST accepted.")
                                                                    }
}
```

唯一的区别在于我们要检查是否存在正确的值参数，并且检查方法是否是POST。

现在我们可以添加列表函数的实现了，这步也很简单：

```
func list(w http.ResponseWriter, r *http.Request) {
    if(r.Method == http.MethodGet) {
        kVStoreMutex.RLock()
        for key, value := range keyValueStore {
            fmt.Fprintln(w, key, ":", value)
        }
        kVStoreMutex.RUnlock()
    } else {
        w.WriteHeader(http.StatusBadRequest)
        fmt.Fprint(w, "Error: Only GET accepted.")
    }
}
```

它会遍布映射并输出所有东西。简单而有效。

为了结束键值存储，我们需要实现remove函数：

```
func remove(w http.ResponseWriter, r *http.Request) {
    if(r.Method == http.MethodDelete) {
        values, err := url.ParseQuery(r.URL.RawQuery)
        if err != nil {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, "Error:", err)
            return
        }
        if len(values.Get("key")) == 0 {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, "Error:", "Wrong input key.")
            return
        }

        kVStoreMutex.Lock()
        delete(keyValueStore, values.Get("key"))
```

```
        kVStoreMutex.Unlock()

        fmt.Fprint(w, "success")
    } else {
        w.WriteHeader(http.StatusBadRequest)
        fmt.Fprint(w, "Error: Only DELETE accepted.")
    }
}
```

这和设置值是一样的，但这里进行的不是设定而是删除。

数据库

设计

在我考虑了整个设计之后，我认为如果数据库可以生成任务 Id 的话效果将会更好。这样做也有助于获取最后一个未完成任务并且生成连续的 Id。

运作方式：

- 它会保存新任务并且指定连续的 Id。
- 它将允许获取新任务。
- 它将允许通过 Id 获取任务。
- 它将允许通过 Id 设置任务。
- 状态将通过 int 来表示：
- 0 - 未开始
- 1 - 进行中
- 2 - 已完成

- 如果一个任务的in progress（进行中）状态过长，它将把状态改为not started（尚未开始）。（可能该任务已经被处理但是中途崩溃了。）
- 它将为了调试/分析需要，允许列出所有任务。

实现

首先，我们需要创建API，然后我们将像之前处理键值存储一样，增加函数的实现。我们还需要一个全局映射作为数据存储、一个指向最老的未开始任务的变量，以及用于存取数据存储和指针的互斥。

```
package main

import (
    "net/http"
    "net/url"
    "fmt"
)

type Task struct {
}

var datastore []Task
var datastoreMutex sync.RWMutex
var oldestNotFinishedTask int // remember to account for potential int
overflow in production. Use something bigger.
var oNFTMutex sync.RWMutex

func main() {

    datastore = make([]Task, 0)
    datastoreMutex = sync.RWMutex{}
    oldestNotFinishedTask = 0
    oNFTMutex = sync.RWMutex{}

    http.HandleFunc("/getById", getById)
    http.HandleFunc("/newTask", newTask)
    http.HandleFunc("/getNewTask", getNewTask)
    http.HandleFunc("/finishTask", finishTask)
    http.HandleFunc("/setById", setById)
```

```
    http.HandleFunc("/list", list)
    http.ListenAndServe(":3001", nil)
}

func getById(w http.ResponseWriter, r *http.Request) {
}

func newTask(w http.ResponseWriter, r *http.Request) {
}

func getNewTask(w http.ResponseWriter, r *http.Request) {
}

func finishTask(w http.ResponseWriter, r *http.Request) {
}

func setById(w http.ResponseWriter, r *http.Request) {
}

func list(w http.ResponseWriter, r *http.Request) {
}
```

我们也已经声明用于存储的Task类型。到目前为止还不错。现在我们就来实现所有这些函数吧!

我们先来实现getById函数。

```
func getById(w http.ResponseWriter, r *http.Request) {
    if r.Method == http.MethodGet {
        values, err := url.ParseQuery(r.URL.RawQuery)
        if err != nil {
            fmt.Fprint(w, err)
            return
        }
        if len(values.Get("id")) == 0 {
```

```
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, "Wrong input")
            return
        }

        id, err := strconv.Atoi(string(values.Get("id")))
        if err != nil {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, err)
            return
        }

        datastoreMutex.RLock()
          bIsInError := err != nil || id >= len(datastore) // Reading the
length of a slice must be done in a synchronized manner. That's why the
mutex is used.
        datastoreMutex.RUnlock()

        if bIsInError {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, "Wrong input")
            return
        }

        datastoreMutex.RLock()
        value := datastore[id]
        datastoreMutex.RUnlock()

        response, err := json.Marshal(value)

        if err != nil {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, err)
            return
        }

        fmt.Fprint(w, string(response))
```

```
    } else {
        w.WriteHeader(http.StatusBadRequest)
        fmt.Fprint(w, "Error: Only GET accepted")
    }
}
```

我们检查GET方法是否被使用过。然后我们解析id参数并检查其是否合适。接下来我们通过strconv.Atoi函数获得id作为int。然后我们需要确保我们的datastore没有出界，我们需要使用mutexes，因为我们接入的映射也可以被另一个线程访问。如果一切都没有问题，那么我们就又需要使用mutexes了，然后通过id获取任务。

在此之后，我们通过JSON库把我们的结构打包成JSON对象，如果完成得没有问题，我们就把这个JSON 对象发送到客户端。

同时，我们也该实现我们的Task结构了：

```
type Task struct {
    Id int `json:"id"`
    State int `json:"state"`
}
```

这就是所需的一切。我们还添加了JSON封送处理器（marshaller）所需的信息。

我们现在可以开始实现newTask函数了：

```
func newTask(w http.ResponseWriter, r *http.Request) {
    if r.Method == http.MethodPost {
```

```
        datastoreMutex.Lock()
        taskToAdd := Task{
            Id: len(datastore),
            State: 0,
        }
        datastore[taskToAdd.Id] = taskToAdd
        datastoreMutex.Unlock()

        fmt.Fprint(w, taskToAdd.Id)
    } else {
        w.WriteHeader(http.StatusBadRequest)
        fmt.Fprint(w, "Error: Only POST accepted")
    }
}
```

这个函数其实很小。用下一个ID创建一个新Task，并把其加入datastore。随后它将返回新的Tasks ID。 这就意味着我们可以继续实现可以列出所有Tasks的函数，因为这会帮助我们在接下来写代码的过程中进行调试。

基本上对于键值存储来说也是同样:

```
func list(w http.ResponseWriter, r *http.Request) {
    if r.Method == http.MethodGet {
        datastoreMutex.RLock()
        for key, value := range datastore {
              fmt.Fprintln(w, key, ": ", "id:", value.Id, " state:", value.
State)
        }
        datastoreMutex.RUnlock()
    } else {
        w.WriteHeader(http.StatusBadRequest)
        fmt.Fprint(w, "Error: Only GET accepted")
    }
}
```

好的，接下来我们开始实现能通过id设置Task的函数：

```
func setById(w http.ResponseWriter, r *http.Request) {
    if r.Method == http.MethodPost {
        taskToSet := Task{}

        data, err := ioutil.ReadAll(r.Body)
        if err != nil {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, err)
            return
        }
        err = json.Unmarshal([]byte(data), &taskToSet)
        if err != nil {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, err)
            return
        }

        bErrored := false
        datastoreMutex.Lock()
           if taskToSet.Id >= len(datastore) || taskToSet.State > 2 ||
taskToSet.State < 0 {
            bErrored = true
        } else {
            datastore[taskToSet.Id] = taskToSet
        }
        datastoreMutex.Unlock()

        if bErrored {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, "Error: Wrong input")
            return
        }

        fmt.Fprint(w, "success")
```

```
    } else {
        w.WriteHeader(http.StatusBadRequest)
        fmt.Fprint(w, "Error: Only POST accepted")
    }
}
```

没什么新鲜的。我们获取请求然后试图解包。如果成功了，我们就把它放进映射，检查其是否出界或者其状态是否无效。如果真是这样的话，我们输出一个错误，否则我们输出success。

如果前面都没有问题我们现在就来实现完成任务函数，因为这个工能非常简单：

```
func finishTask(w http.ResponseWriter, r *http.Request) {
    if r.Method == http.MethodPost {
        values, err := url.ParseQuery(r.URL.RawQuery)
        if err != nil {
            fmt.Fprint(w, err)
            return
        }
        if len(values.Get("id")) == 0 {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, "Wrong input")
        return
        }

    id, err := strconv.Atoi(string(values.Get("id")))

        if err != nil {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, err)
            return
        }
```

```
        updatedTask := Task{Id: id, State: 2}

        bErrored := false

        datastoreMutex.Lock()
        if datastore[id].State == 1 {
            datastore[id] = updatedTask
        } else {
            bErrored = true
        }
        datastoreMutex.Unlock()

        if bErrored {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, "Error: Wrong input")
            return
        }

        fmt.Fprint(w, "success")
    } else {
        w.WriteHeader(http.StatusBadRequest)
        fmt.Fprint(w, "Error: Only POST accepted")
    }
}
```

这和getById函数非常像。区别在于只有在处于in progress时我们才会在这里更新状态。

现在终于来到最有趣的函数了——getNewTask函数。它需要负责更新已知的最老已完成任务，还需要负责处理任务被接受后中途崩溃的情况。这种情况会导致永远处于进行状态中的鬼任务。这就是为什么我们必须添加这个函数，在启动一个任务的120秒之后，它会把任务设置回not started（未启动）：

```
func getNewTask(w http.ResponseWriter, r *http.Request) {
    if r.Method == http.MethodPost {

        bErrored := false

        datastoreMutex.RLock()
        if len(datastore) == 0 {
            bErrored = true
        }
        datastoreMutex.RUnlock()

        if bErrored {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, "Error: No non-started task.")
            return
        }

        taskToSend := Task{Id: -1, State: 0}

        oNFTMutex.Lock()
        datastoreMutex.Lock()
        for i := oldestNotFinishedTask; i < len(datastore); i++ {
            if datastore[i].State == 2 && i == oldestNotFinishedTask {
                oldestNotFinishedTask++
                continue
            }
            if datastore[i].State == 0 {
                datastore[i] = Task{Id: i, State: 1}
                taskToSend = datastore[i]
                break
            }
        }
        datastoreMutex.Unlock()
        oNFTMutex.Unlock()

        if taskToSend.Id == -1 {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, "Error: No non-started task.")
```

```
            return
        }

        myId := taskToSend.Id

        go func() {
            time.Sleep(time.Second * 120)
            datastoreMutex.Lock()
            if datastore[myId].State == 1 {
                datastore[myId] = Task{Id: myId, State: 0}
            }
            datastoreMutex.Unlock()
        }()

        response, err := json.Marshal(taskToSend)

        if err != nil {
            w.WriteHeader(http.StatusBadRequest)
            fmt.Fprint(w, err)
            return
        }

        fmt.Fprint(w, string(response))
    } else {
        w.WriteHeader(http.StatusBadRequest)
        fmt.Fprint(w, "Error: Only POST accepted")
    }
}
```

首先我们需要找到尚未启动的最老任务。顺便我们还需要更新oldestNotFinishedTask变量。如果一个任务已经完成了并且被变量指定，那么变量get就会增长。如果我们发现尚未开始的任务，我们就打破循环然后把其送回给用户并将其设定为in progress。但是当我们开启另一个线程上的函数时，如果该任务在120秒后仍然in progress，任务的状态就会被改回到not started。

现在还有最后一件事。当你不知道数据库在哪时，数据库就没有用！这就是为什么我们现在要实现一种机制，而数据库将会利用该机制在键值存储中注册自己。

```
func main() {

    if !registerInKVStore() {
        return
    }

    datastore = make(map[int]Task)
```

随后我们定义这个函数：

```
func registerInKVStore() bool {
    if len(os.Args) < 3 {
        fmt.Println("Error: Too few arguments.")
        return false
    }
    databaseAddress := os.Args[1] // The address of itself
    keyValueStoreAddress := os.Args[2]

     response, err := http.Post("http://" + keyValueStoreAddress + "/
set?key=databaseAddress&value=" + databaseAddress, "", nil)
    if err != nil {
        fmt.Println(err)
        return false
    }
    data, err := ioutil.ReadAll(response.Body)
    if err != nil {
        fmt.Println(err)
        return false
    }
    if response.StatusCode != http.StatusOK {
          fmt.Println("Error: Failure when contacting key-value store: ",
string(data))
```

```
        return false
    }
    return true
}
```

我们检查一下是否至少有三个参数。（第一个是可执行的。）我们从第二个参数中读取现在的数据库地址，并且从第三个参数中读取键值存储地址。利用这些信息我们可以制作POST请求并在其中把databaseAddress键加到k/v存储中，并把它的值设定为现在的数据库地址。如果响应的状态码没有OK，那么我们就知道事情进行得不顺利并且输出错误。在此之后，我们退出程序。

结论

我们现在已经完成了k/v存储和数据库。你现在甚至可以再REST客户端上测试这个部分。（我使用的是这个。）请记住，如果有必要的话你可以随时改变这些代码，但是我认为这样的情况不多。我希望你在这篇文章中获得了乐趣！

更新：我把sync.Mutex换成了 sync.RWMutex，在我们只读取数据的地方，我把mutex.Lock/Unlock换成了mutex.RLock/RUnlock。■

英文原文：Web app using Microservices in Go: Part 1 - Design Web app using Microservices in Go: Part 2 - k/v store and Database

Learning as we Go

我们用Go构建Teamwork Desk时犯下的菜鸟错误

作者 / Peter Kelly

Teamwork Desk高级工程师。

我们热爱Go语言。在过去的一年中我们为了构建Teamwork Desk的无数个服务，写下了将近200 000行Go代码。我们已经构建了该产品的十多个小型HTTP服务。为什么我们要使用Go？Go是一种快速（非常快）的静态类型编译语言，它有强大的并发模型、垃圾收集、优异的标准库、无继承、传奇作者、多核支持，以及非常不错的社区，更别说对于我们这种写Web应用的人，它的goroutine-per-request设置可以避免事件循环和回调地狱。在构建系统和服务器方面（尤其是微服务），Go语言已经成为了大热门。

正如使用任何新语言和技术一样，我们在早期的实践中经历了一段跌跌撞撞的过程。Go语言确实有自己的风格和语言特性，尤其当你原来使用的语言是OO语言比如Java或脚本语言比如Python时。所以我们确实犯了一些错误，而且我们愿意和大家分享这些错误以及我们从中得到的教训。如果你在生产环境中使用Go，你就能认出所有这些问题。如果你刚开始使用Go，那么希望你能从中找到一些对你有用的东西。

1. Revel对于我们来说不是一个好的选择

刚开始使用Go吗？要构建Web服务器？你需要一个框架吧？你可能会这么认为。使用MVC框架确实有一些优势，这些优势主要来自于“约定优于配置”方式给予项目的结构和约定，这种方式可以提供一致性并且降低跨项目开发的门槛。我们的结论是相比于约定的优势，我们更倾向于配置的力量，特别是用Go语言写Web应用毫不费力时，我们的很多Web应用都是小型服务。对我们来说，盖棺论定的一点在于，这种方法是不符合语言习惯的。Revel的基本观念在于努力向Go中引入类似于Play或Rails这样的框架，而不是使用Go和stdlib的力量并以此为基础向上构建。Revel作者是这样说的：

> 一开始这只是一个好玩的项目，我想看看是否可以复制神奇的Play！到了1.x版时我感觉到了不那么神奇的Go。

公正地说，在新语言中使用MVC框架对于当时的我们来说是很有道理的，因为这样做可以去除关于结构的争论，让新的团队可以以一种连贯的方式开始构建。在使用Go语言之前，几乎所有我写的Web应用都多少借助了一些MVC框架的方式。C#？ ASP.NET MVC。Java ？ SpringMVC。PHP ？ Symfony。Python ？ CherryPy。Ruby？ RoR。最后我们意识到我们不需要在Go中使用框架。标准库HTTP程序包已经拥有你需要的东西了，你通常只需要添加一个多路复用器（比如mux）用于路由选择，以及一个中

间件（比如negroni）的lib用于处理认证和登录等，这就是你需要的全部了。Go的HTTP程序包设计让一切都很简单。你还会意识到Go的一些能力就存在于Go的工具链和Go周围的工具中，这些工具会提供给你广泛而强大命令。但是在Revel中，因为项目结构的设置，并且因为其中不含有 `package main` 和 `func main() {}`条目（对于很多Go命令来说这是符合习惯而且必要的），你就不能使用这些工具。事实上Revel带有自己的命令包，它会镜像一些像 `run` 和 `build`和这样的命令。 使用Revel：

- 不能运行 `go build`
- 不能运行 `go install`
- 不能运行race探测器 (--race)
- 不能使用 `go-fuzz` 或其他任何需要可构建Go源的好工具
- 不能使用其他中间件或路由器
- 热重载虽然简洁，但很慢，Revel在源上使用反射，并且根据我们使用1.4版的经验，增加了约30%的编译时间。而且它还不使用go。
- 不能迁移到Go 1.5或更高版本，因为在Revel中的编译时间还要更慢。我们去掉了Revel并且把内核迁移到了1.6上。
- Revel把测试挪到了dir下面，违反了Go把`_test.go`文件和被测试的文件一起放进相同程序包的惯例。
- Revel测试如果要运行，就会启动你的服务器，从而进行集成测试。

我们发现Revel严重偏离了符合Go语言使用习惯的构建方式，而且我们失去了Go工具箱中的一些强大的力量。

2. 聪明地使用Panic

如果你原来是一位Java或C#开发者，你可能需要适应一下在Go中处理错误的方式。Go具有从函数返回的多次回迹，所以函数把其他信息和错误一起返回是一种非常普通的情况，当然如果没有问题的话就会返回nil（nil是Go中引用类型的默认值）。

```
func something() (thing string, err error) {
    s := db.GetSomething()
    if s == "" {
        return s, errors.New("Nothing Found")
    }
    return s, nil
}
```

我们最终无奈使用了panic，我们其实是想创建一个错误，并使其被调用栈的上级处理。

```
s, err := something()
    if err != nil {
    panic(err)
}
```

但是我们却真的惊慌（panic）了。一个错误？天呐，运行！但是在Go中，你需要认识到错误是值，它们是完全正常的，而且是调用函数和处理响应的惯用部分。panic会打垮你的应用，让它彻底死掉。就像一个运行时异常一样。如果只是一个函数返回了一个错误，你为什么要这样做？这是一个教训。在1.6之前，用于转储panic的堆栈转储（stack dump）都是运行Go例行程序的，所以想要找到原来的问题非常困难。你最终不得不费力做很多本不需要做的事。

哪怕你真的有一个不可恢复的错误，或者你遇到了一个运行时panic，你大概也不想要毁掉你的整个Web服务器，你的服务器可能还在处理其他很多事情。(你的数据库也需要使用事务吧？)所以我们学会了如何处理这些事件，在Revel中添加过滤器，它可以恢复panic并且捕捉被打印到日志文件的堆栈跟踪，并发送到Sentry，于是我们就会马上在邮件和Teamwork Chat中获得提醒。API会向前端返回`500 Internal Server Error`。

```
// PanicFilter wraps the action invocation in a protective defer blanket
that
// recovers panics, logs everything, and returns 500.
func PanicFilter(rc *revel.Controller, fc []revel.Filter) {
    defer func() {
        if err := recover(); err != nil {
            handleInvocationPanic(rc, err) // stack trace, logging. alerting
        }
    }()
    fc[0](rc, fc[1:])
}
```

3. 仔细读取Request.Body不止一遍

在读取了`http.Request.Body`之后，`Body`就被排空了，而随后的读取就会返回`[]byte{}`——一个空的主文。这是因为当你在读取`http.Request.Body`的字节时，读取器处于这些字节的结尾，需要重置才能再次读取。但是，`http.Request.Body`是一个`io.ReadWriter`并且没有Peek或Seek这样可以帮上忙的方法。一个解决这个问题的方法是先把主文复制进存储空间，然后在读取之后把原来的回拨回去。如果你的请求都很庞大这么做的成本很高。这绝对是一个让人恍然大悟，而且时不时还会让人措手不及的程序。

下面是一个简短但是完整的展示程序：

```
package main

import (
    "bytes"
    "fmt"
    "io/ioutil"
    "net/http"
)

func main() {
    r := http.Request{}
     // Body is an io.ReadWriter so we wrap it up in a NopCloser to satisfy
that interface
    r.Body = ioutil.NopCloser(bytes.NewBuffer([]byte("test")))

    s, _ := ioutil.ReadAll(r.Body)
    fmt.Println(string(s)) // prints "test"

    s, _ = ioutil.ReadAll(r.Body)
    fmt.Println(string(s)) // prints empty string!
}
```

下面是复制并执行回拨的代码……如果你还记得的话

```
content, _ := ioutil.ReadAll(r.Body)
// Replace the body with a new io.ReadCloser that yields the same bytes
r.Body = ioutil.NopCloser(bytes.NewBuffer(content))
again, _ = ioutil.ReadAll(r.Body)
```

你可以创建一个小 util 函数：

```
func ReadNotDrain(r *http.Request) (content []byte, err error) {
    content, err = ioutil.ReadAll(r.Body)
```

```
    r.Body = ioutil.NopCloser(bytes.NewBuffer(content))
    return
}
```

然后调用它而不是使用像ioutil.ReadAll这样的命令。

```
content, err := ReadNotDrain(&r)
```

当然现在你已经用一个空操作替换了r.Body.Close()，当你在request.Body上调用Close时，这个空操作什么也不会做。这就是thehttputil.DumpRequest 的工作方式。

4. 有一些持续改善的库可以帮你写SQL

Teamwork Desk向顾客提供Web应用时需要完成的核心工作涉及很多MySQL。我们不使用存储过程，所以我们在Go中的数据层包含有一些很复杂的SQL……有些代码可能会因为其建立的复杂查询而赢得奥林匹克体操比赛的金牌。我们开始时使用Gorm和它的可链API来构建我们的SQL。你在Gorm中还是可以使用原始SQL，并且把结果打包进你的结构。（值得注意的是，我们发现我们执行这项操作的次数越来越多，这可能说明我们需要重新回到使用Gorm的真正方法，并且保证对其善加利用，否则我们就需要寻找其他替代品了——这种情况也没什么可怕的。）

对于有些人来说，ORM的确是一句脏话（当人们说你失去控制力、理解力，以及优化查询的可能性时）。我们真的只是把Gorm作为构建查询的封装器，我们理解它给我们的输出，我们并没有把

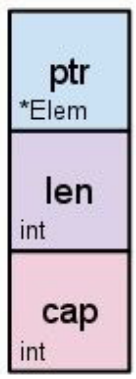
Gorm作为一个完整的ORM来使用。Grom允许你利用它的可链API（如下），并且把结果打包到结构中。Gorm的很多特性可以让你免受代码中的手工SQL折磨。它还支持Preloading，Limits，Grouping，Associations，原始SQL，Transactions以及更多。如果你正在Go中手写SQL的话，Gorm绝对值得你关注。

```
var customer Customer
   query = db.
   Joins("inner join tickets on tickets.customersId = customers.id").
   Where("tickets.id = ?", e.Id).
   Where("tickets.state = ?", "active").
   Where("customers.state = ?", "Cork").
   Where("customers.isPaid = ?", false).
   First(&customer)
```

5. 没有指向的指针是没有意义的

这真的只是针对slice来说的。把slice传到函数上？在Go里，阵列也是值，所以如果你有一个很大的阵列，你不想每次传递和分配它的时候都要复制一下吧？没错。到处传递阵列对于存储空间来说是一个昂贵的开销。但是在Go中，99%的时间你都在和slice打交道，而不是阵列。slice可以被看做用来描述阵列某些部分（经常是全部）的东西，它含有一个指向阵列开始元素的指针、slice的长度，以及slice的容量。

slice的每部分只需要8个字节，所以它永远都不会超过24个字节，无论其下的阵列有多少内容，有多大。



我们经常把指针传给一个函数的slice，并且误以为我们节省了存储空间。

```
t := getTickets() // e.g. returns []Tickets, a slice
ft := filterTickets(&t)

func filterTickets(t *[]Tickets) []Tickets {}
```

如果我们在 t 中有很多数据，我们以为通过把数据传给 filterTickets，就避免了存储空间中的大型数据拷贝。鉴于我们现在对slice的理解，我们可以愉快地把这个slice按照值来传递，而不用考虑存储空间问题。

```
t := getTickets() // []Tickets massive list of tickets, 20MB
ft := filterTickets(t)

func filterTickets(t []Tickets) []Tickets {} // 24 bytes passed by value
```

当然，不按引用传递也意味着避免了错误地改变指针指向的问题，因为slice 自身就是引用类型。

6. 裸返回随时可能造成伤害并且让你的代码难以理解（在更大的函数中）

“裸返回 (Naked returns)”这个名词描述的是在Go语言中，你从一个函数返回时不明确说明你返回的是什么。啥？在Go中，你可以有命名返回值，比如 `func add(a, b int) (total int) {}`。我可以只用`return` 就从函数中返回，而不需要使用`return total`。在小函数中使用裸返回也可以是简洁而有效的。

```
func findTickets() (tickets []Ticket, countActive int64, err error) {
    tickets, countActive = db.GetTickets()
    if tickets == 0 {
        err = errors.New("no tickets found!")
    }
    return
}
```

这里发生的情况显而易见。如果没有tickets，那么就会返回`0`，`0`，`error`。如果找到了tickets，那么类似 `120`，`80`，`nil`这样的东西就会被返回，这取决于ticket count等因素。这里的关键在于如果你在签名中有命名返回值，那么你可以使用`return`（裸返回），当调用`return` 时，它会在每个命名返回值的所处状态中为命名返回值返回数值。

但是……我们曾有一些（现在也有……）大型函数。过于大。甚至大得很蠢。函数中任何长到你需要滚动浏览的裸返回都是细微的漏洞，对于可读性来说也是灾难。特别是当你还有多个返回点时。不要这么做。两种做法都不可取——无论是裸返回还是大函数。以下是一个假设的例子：

```
func findTickets() (tickets []Ticket, countActive int64, err error) {
    tickets, countActive := db.GetTickets()
    if tickets == 0 {
        err = errors.New("no tickets found!")
    } else {
        tickets += addClosed()
        // return, hmmm...okay, I might know what this is
        return
    }
    .
    .
    .
    // lots more code
    .
    .
    .
    if countActive > 0 {
        countActive - closedToday()
        // have to scroll back up now just to be sure...
        return
    }
    .
    .
    .
    // Okay, by now I definitely can't remember what I was returning or what
values they might have
    return
}
```

7. 小心作用域和缩略声明

当你利用缩略法 := 用相同的名字在不同块中声明变量时（被称为 shadowing），你可能会因为 Go 中作用域的问题引入微小的漏洞。

```
func findTickets() (tickets []Ticket, countActive int64) {
    tickets, countActive := db.GetTickets() // 10 tickets returned, 3 active
```

```
    if countActive > 0 {
        // oops, tickets redeclared and used just in this block
          tickets, err := removeClosed() // 6 tickets left after removing
closed
        if err != nil {
            // Argh! We used the variables here for logging!, if we didn't
we would
               // have received a compile-time error at least for unused
variables.
            log.Printf("could not remove closed %s, ticket count %d", err.
Error(), len(tickets))
        }
    }
    return // this will return 10 tickets o_O
}
```

这里的问题存在于 := 缩略变量声明和分配之间。通常来说当你在左边声明新变量时，:=只会编译。但是如果左边有任何变量是新的话，它也会这样运行。在上面的例子中 err 是新的，所以你期待 tickets 被覆写，就像是已经在上面的函数返回参数中声明了一样。但是实际情况并非如此，原因在于块作用域——一个新的被声明的 ticket 变量被分配出去，并且一旦块完成之后就会丢失自己的作用域。要改变这一点，只要在块外声明变量 err，并且使用 = 而非 := 。（一个好的编辑器，比如 Emacs 或 Sublime 会解决这个 shadowing 问题。）

```
func findTickets() (tickets []Ticket, countActive int64) {
    var err error
    tickets, countActive := db.GetTickets() // 10 tickets returned, 3 active
    if countActive > 0 {
           tickets, err = removeClosed() // 6 tickets left after removing
closed
```

```
        if err != nil {
             log.Printf("could not remove closed %s, ticket count %d", err.
Error(), len(tickets))
        }
    }
    return // this will return 6 tickets
}
```

8. 映射和随机崩溃

用并发方式访问映射并不安全。我们有一个例子关于一个曾经为应用全程准备的包级别映射。这个映射用于回收应用中每个控制器的数据，当然在Go中，每个http请求都是它自身的goroutine。你能看出来将会发生什么——最终不同的goroutine会试图同时访问映射，无论是读还是写。这会造成panic，而我们的应用将会崩溃（当进程停止时，我们用Ubuntu上的upstart脚本来重生应用，至少保持让应用“不死”）。寻找这类panic原因的过程很笨重，有点像1.6版以前的情况，当堆栈转储把所有运行的goroutine都包括进来时，就会产生大量需要筛查的日志。

Go团队确实考虑过使映射在并发访问时更安全，但是最终决定放弃，因为这会为一般场景造成不必要的开支——这是一种让事情保持简单的实用的做法。参见golang.org FAQ。

“在漫长的讨论之后，我们决定，映射的一般用法并不需要来自多个goroutine的安全访问，在需要的场景中，映射可能处于某些已经被同步过的大型数据结构或计算中。所以，要求所有映射操作抓取互斥元会减慢大 多数程序，但是却只为很少的程序提供了安全性。

因为不可控制的映射访问会使程序崩溃，所以这并不是一个轻松的决定。”

我们的代码看起来有点像这个：

```
package stats

var Requests map[*revel.Controller]*RequestLog
var RequestLogs map[string]*PathLog
```

我们把它变成使用stdlib中的 sync包来嵌入结构中的读取器/写入器互斥锁，该结构还会封装我们的映射。我们向结构中添加了一些帮手：Add 和 Get方法。

```
var Requests ConcurrentRequestLogMap

// init is run for each package when the app first runs
func init() {
    Requests = ConcurrentRequestLogMap{items: make(map[interface{}]*Request
Log)}
}

type ConcurrentRequestLogMap struct {
    sync.RWMutex // We embed the sync primitive, a reader/writer Mutex
    items map[interface{}]*RequestLog
}

func (m *ConcurrentRequestLogMap) Add(k interface{}, v *RequestLog) {
    m.Lock() // Here we can take a write lock
    m.items[k] = v
    m.Unlock()
}

func (m *ConcurrentRequestLogMap) Get(k interface{}) (*RequestLog, bool) {
```

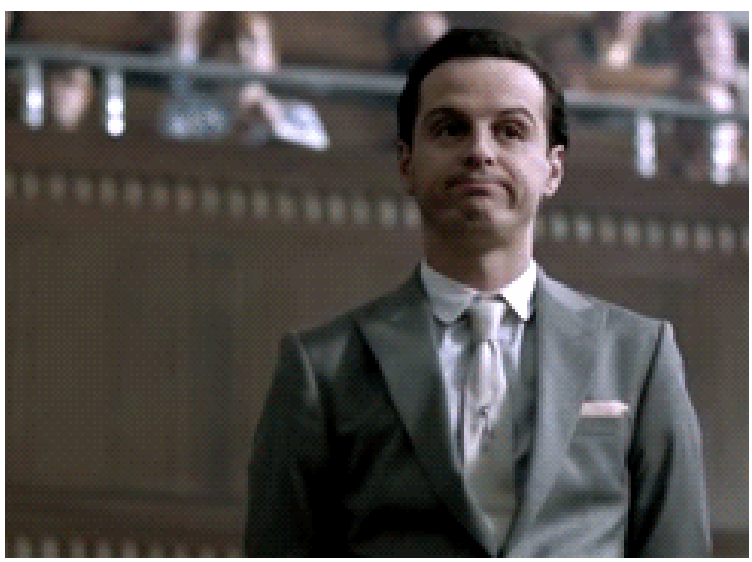
```
    m.RLock() // And here we can take a read lock
    v, ok := m.items[k]
    m.RUnlock()

    return v, ok
}
```

从此不再有崩溃了。

9. Vendor——宙斯的胡子

好吧，承认这件事有点不好意思。我们被抓住了，人赃俱获。罪名成立……哎。我们把代码部署到生产中时竟然没用vendor。



如果你不知道的话，接下来我将向你说明这件事为什么很糟糕。你通过从你项目的根中运行 `go get ./...` 获得依赖。这会把每个依赖从master上的HEAD拉走。很明显，这种情况非常糟糕，除非你

在服务器的$GOPATH上保存了完全相同版本的依赖，而且从来不更新（而且从来没有重建或启动新服务器），否则破坏性的改变不可避免，而你也无法控制在生产过程中运行的代码。在Go 1.4中我们使用了Godeps及其GOPATH方法。在1.5中我们使用了GO15VENDOREXPERIMENT环境变量。在1.6中，谢天谢地，项目根中的/vendor终于可以被识别为可以存放你的依赖的地方，不再需要工具了。你可以使用各种vendoring工具中的一种来追踪版本并且更轻松地添加/更新依赖（移除.git，更新清单等。）

学到了很多，但是还有更多需要学习

这只是我们早期犯下的一部分错误以及从中得到教训的小清单。我们是一个由5个开发者组成的构建Teamwork Desk的小团队，但是我们在去年一年的时间里学到了关于Go的大量知识，同时我们还以近乎危险的速度交付了大量的优秀功能。今年你会看到我们出席各种Go技术大会，包括在丹佛举行的GopherCon，我很快也将在科克本地的开发者聚会上分享关于Go的实践。我们将会继续关注Go开源工具发布，并且回馈已有的库。到目前为止，我们向一些小项目（如下）贡献了不算少的代码，我们提出的性能要求也被Stripe、Revel，以及其他开源Go项目所采纳。

- s3pp
- stripehooks
- tnef parser

我们永远都在寻找优秀的开发者！到 Teamwork.com 上来，加入我们吧！■

英文原文：Some rookie Go mistakes we made building Teamwork Desk, and what we learned from them

Learning as we Go

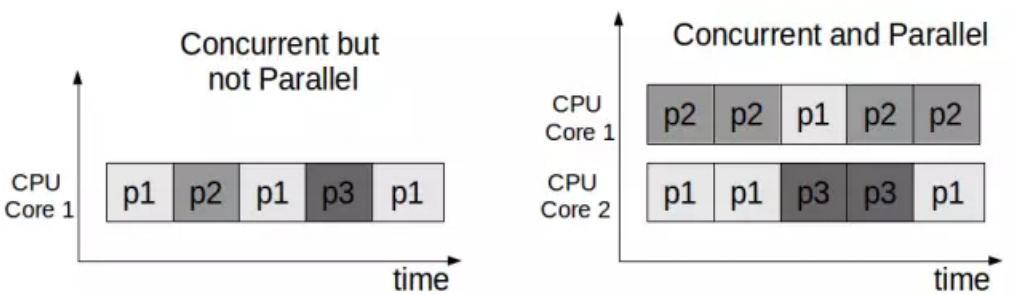
并发之痛：

Thread，Goroutine，Actor

作者／王渊命

团队协作 IM 服务 Grouk 联合创始人及 CTO，技术极客，曾任新浪微博架构师、微米技术总监。2014 年作为联合创始人创立团队协作 IM 服务 Grouk，长期关注团队协作基础工具和研发环境建设，Docker 深度实践者。

聊这个话题之前，先梳理下两个概念，几乎所有讲并发的文章都要先讲这两个概念：



- 并发（concurrency）并发的关注点在于任务切分。举例来说，你是一个创业公司的CEO，开始只有你一个人，你一人分饰多角，一会做产品规划，一会写代码，一会见客户，虽然你不能见客户的同时写代码，但由于你切分了任务，分配了时间片，表现出来好像是多个任务一起在执行。
- 并行（parallelism）并行的关注点在于同时执行。还是上面的例子，你发现你自己太忙了，时间分配不过来，于是请了工程师，产品经理，市场总监，各司一职，这时候多个任务可以同时执行了。

所以总结下，并发并不要求必须并行，可以用时间片切分的方式模拟，比如单核 CPU 上的多任务系统，并发的要求是任务能切分成独立执行的片段。而并行关注的是同时执行，必须是多（核）CPU，要能并行的程序必须是支持并发的。本文大多数情况下不会严格区分这两个概念，默认并发就是指并行机制下的并发。

为什么并发程序这么难？

> We believe that writing correct concurrent, fault-tolerant and scalable applications is too hard. Most of the time it ' s because we are using the wrong tools and the wrong level of abstraction. —— Akka

Akka 官方文档开篇这句话说的好，之所以写**正确的并发，容错，可扩展的程序如此之难，是因为我们用了错误的工具和错误的抽象。**（当然该文档本来的意思是 Akka 是正确的工具，但我们可以独立的看待这句话）。

那我们从最开始梳理下程序的抽象。开始我们的程序是面向过程的，数据结构 + func。后来有了面向对象，对象组合了数结构和 func，我们想用模拟现实世界的方式，抽象出对象，有状态和行为。但无论是面向过程的 func 还是面向对象的 func，本质上都是代码块的组织单元，本身并没有包含代码块的并发策略的定义。于是为了解决并发的需求，引入了 Thread（线程）的概念。

线程（Thread）

1.系统内核态，更轻量的进程

2.由系统内核进行调度

3.同一进程的多个线程可共享资源

线程的出现解决了两个问题，一个是 GUI 出现后急切需要并发机制来保证用户界面的响应。第二是互联网发展后带来的多用户问题。最早的 CGI 程序很简单，将通过脚本将原来单机版的程序包装在一个进程里，来一个用户就启动一个进程。但明显这样承载不了多少用户，并且如果进程间需要共享资源还得通过进程间的通信机制，线程的出现缓解了这个问题。

线程的使用比较简单，如果你觉得这块代码需要并发，就把它放在单独的线程里执行，由系统负责调度，具体什么时候使用线程，要用多少个线程，由调用方决定，但定义方并不清楚调用方会如何使用自己的代码，很多并发问题都是因为误用导致的，比如 Go 中的 map 以及 Java 的 HashMap 都不是并发安全的，误用在多线程环境就会导致问题。另外也带来复杂度：

1. 竞态条件（race conditions）如果每个任务都是独立的，不需要共享任何资源，那线程也就非常简单。但世界往往是复杂的，总有一些资源需要共享，比如前面的例子，开发人员和市场人员同时需要和 CEO 商量一个方案，这时候 CEO 就成了竞态条件。

2. 依赖关系以及执行顺序 如果线程之间的任务有依赖关系，需要等待以及通知机制来进行协调。比如前面的例子，如果产品和 CEO 讨论的方案依赖于市场和 CEO 讨论的方案，这时候就需要协调机制保证顺序。

为了解决上述问题，我们引入了许多复杂机制来保证:

- Mutex (Lock) (Go 里 的 sync 包 , Java 的 concurrent 包) 通过互斥量来保护数据，但有了锁，明显就降低了并发度。
- Semaphore 通过信号量来控制并发度或者作为线程间信号 (signal) 通知。
- Volatile Java 专门引入了 volatile 关键词来，来降低只读情况下的锁的使用。
- Compare-and-swap 通过硬件提供的 CAS 机制保证原子性 (atomic)，也是降低锁的成本的机制。

如果说上面两个问题只是增加了复杂度，我们通过深入学习，严谨的 Code Review，全面的并发测试 (比如 Go 语言中单元测试的时候加上 -race 参数)，一定程度上能解决 (当然这个也是有争议的，有论文认为当前的大多数并发程序没出问题只是并发度不够，如果 CPU 核数继续增加，程序运行的时间更长，很难保证不出问题)。但最让人头痛的还是下面这个问题:

系统里到底需要多少线程?

这个问题我们先从硬件资源入手，考虑下线程的成本:

- **内存（线程的栈空间）**

  每个线程都需要一个栈（Stack）空间来保存挂起（suspending）时的状态。Java 的栈空间（64 位 VM）默认是 1024k，不算别的内存，只是栈空间，启动 1024 个线程就要 1G 内存。虽然可以用 -Xss 参数控制，但由于线程是本质上也是进程，系统假定是要长期运行的，栈空间太小会导致稍复杂的递归调用（比如复杂点的正则表达式匹配）导致栈溢出。所以调整参数治标不治本。

- **调度成本（context-switch）**

  我在个人电脑上做的一个非严格测试，模拟两个线程互相唤醒轮流挂起，线程切换成本大约 6000 纳秒 / 次。这个还没考虑栈空间大小的影响。国外一篇论文专门分析线程切换的成本，基本上得出的结论是切换成本和栈空间使用大小直接相关。

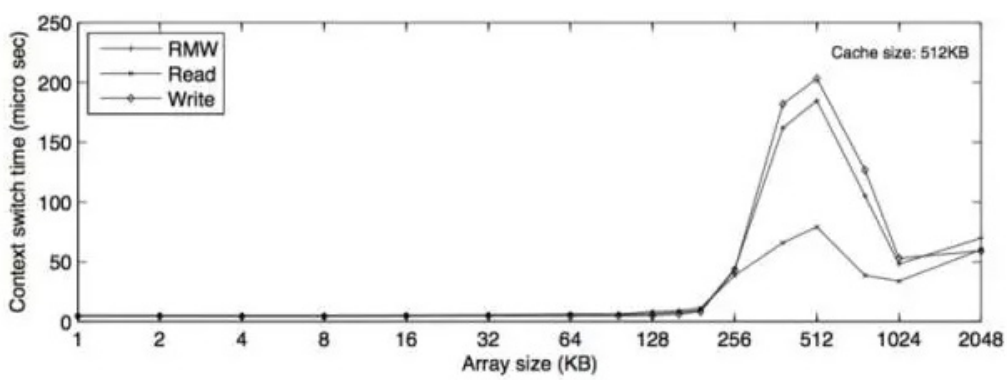


Figure 1: The effect of data size on the cost of the context switch

- **CPU使用率**

  我们搞并发最主要的一个目标就是我们有了多核，想提高CPU利用率，最大限度的压榨硬件资源，从这个角度考虑，我们应该用多少线程呢?

100毫秒的时间片，该线程实际使用运算15毫秒，等待网络80毫秒，解析和处理结果返回5毫秒，我们有4核CPU，应该多少线程合适?

这个我们可以通过一个公式计算出来，100 / (15 + 5) * 4 = 20，用 20 个线程最合适。但一方面网络的时间不是固定的，另外一方面，如果考虑到其他瓶颈资源呢？比如锁，比如数据库连接池，就会更复杂。

作为一个 1 岁多孩子的父亲，认为这个问题的难度好比你要写个给孩子喂饭的程序，需要考虑『给孩子喂多少饭合适？』，这个问题有以下回答以及策略:

- 孩子不吃了就好了（但孩子贪玩，不吃了可能是想去玩了）
- 孩子吃饱了就好了（废话，你怎么知道孩子吃饱了？孩子又不会说话）

- 逐渐增量，长期观察，然后计算一个平均值（这可能是我们调整线程常用的策略，但增量增加到多少合适呢？）
- 孩子吃吐了就别喂了（如果用逐渐增量的模式，通过外部观察，可能会到达这个边界条件。系统性能如果因为线程的增加倒退了，就别增加线程了）
- 没控制好边界，把孩子给给撑坏了（这熊爸爸也太恐怖了。但调整线程的时候往往不小心可能就把系统搞挂了）

通过这个例子我们可以看出，从外部系统来观察，或者以经验的方式进行计算，都是非常困难的。于是结论是：

让孩子会说话，吃饱了自己说，自己学会吃饭，自管理是最佳方案。

然并卵，计算机不会自己说话，如何自管理？

但我们从以上的讨论可以得出一个结论：

- 线程的成本较高（内存，调度）不可能大规模创建
- 应该由语言或者框架动态解决这个问题

线程池方案

Java 1.5 后，Doug Lea 的 Executor 系列被包含在默认的 JDK 内，是典型的线程池方案。

线程池一定程度上控制了线程的数量，实现了线程复用，降低了线程的使用成本。但还是没有解决数量的问题，线程池初始化的时候还是要设置一个最小和最大线程数，以及任务队列的长度，自管理只是在设定范围内的动态调整。另外不同的任务可能有不同的并发需求，为了避免互相影响可能需要多个线程池，最后导致的结果就是 Java 的系统里充斥了大量的线程池。

新的思路

从前面的分析我们可以看出，如果线程是一直处于运行状态，我们只需设置和 CPU 核数相等的线程数即可，这样就可以最大化的利用 CPU，并且降低切换成本以及内存使用。但如何做到这一点呢？

陈力就列，不能者止

这句话是说，能干活的代码片段就放在线程里，如果干不了活（需要等待，被阻塞等），就摘下来。通俗的说就是不要占着茅坑不拉屎，如果拉不出来，需要酝酿下，先把茅坑让出来，因为茅坑是稀缺资源。

要做到这点一般有两种方案：

1. **异步回调方案**　典型如 NodeJS，遇到阻塞的情况，比如网络调用，则注册一个回调方法（其实还包括了一些上下文数据对象）给 IO 调度器（Linux 下是 Libev，调度器在另外的线程里），当前线

程就被释放了，去干别的事情了。等数据准备好，调度器会将结果传递给回调方法然后执行，执行其实不在原来发起请求的线程里了，但对用户来说无感知。但这种方式的问题就是很容易遇到 callback hell，因为所有的阻塞操作都必须异步，否则系统就卡死了。还有就是异步的方式有点违反人类思维习惯，人类还是习惯同步的方式。

2. **GreenThread/Coroutine/Fiber 方案** 这种方案其实和上面的方案本质上区别不大，关键在于回调上下文的保存以及执行机制。为了解决回调方法带来的难题，这种方案的思路是写代码的时候还是按顺序写，但遇到 IO 等阻塞调用时，将当前的代码片段暂停，保存上下文，让出当前线程。等 IO 事件回来，然后再找个线程让当前代码片段恢复上下文继续执行，写代码的时候感觉好像是同步的，仿佛在同一个线程完成的，但实际上系统可能切换了线程，但对程序无感。

GreenThread

- 用户空间 首先是在用户空间，避免内核态和用户态的切换导致的成本。
- 由语言或者框架层调度。
- 更小的栈空间允许创建大量实例（百万级别）。

几个概念

- Continuation 这个概念不熟悉 FP 编程的人可能不太熟悉，不过这里可以简单的顾名思义，可以理解为让我们的程序可以暂停，然后下次调用继续（contine）从上次暂停的地方开始的一种机制。相当于程序调用多了一种入口。
- Coroutine 是 Continuation 的一种实现，一般表现为语言层面的组件或者类库。主要提供 yield，resume 机制。
- Fiber 和 Coroutine 其实是一体两面的，主要是从系统层面描述，可以理解成 Coroutine 运行之后的东西就是 Fiber。

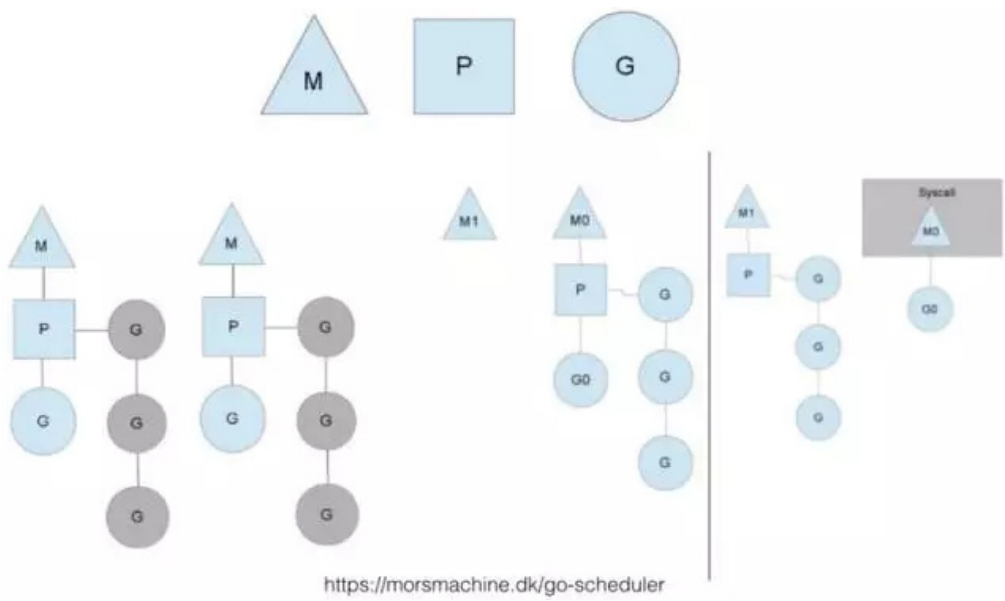
Goroutine

Goroutine 其实就是前面 GreenThread 系列解决方案的一种演进和实现。

- 首先，它内置了 Coroutine 机制。因为要用户态的调度，必须有可以让代码片段可以暂停/继续的机制。
- 其次，它内置了一个调度器，实现了 Coroutine 的多线程并行调度，同时通过对网络等库的封装，对用户屏蔽了调度细节。

- 最后，提供了 Channel 机制，用于 Goroutine 之间通信，实现 CSP 并发模型 (Communicating Sequential Processes)。因为 Go 的 Channel 是通过语法关键词提供的，对用户屏蔽了许多细节。其实 Go 的 Channel 和 Java 中的 SynchronousQueue 是一样的机制，如果有 buffer 其实就是 ArrayBlockQueue。

Goroutine 调度器



这个图一般讲 Goroutine 调度器的地方都会引用，想要仔细了解的可以看看原博客 (小编：点击阅读原文获取)。这里只说明几点：

1. M 代表系统线程，P 代表处理器 (核)，G 代表 Goroutine。Go 实现了 M : N 的调度，也就是说线程和 Goroutine 之间

是多对多的关系。这点在许多GreenThread / Coroutine 的调度器并没有实现。比如 Java 1.1 版本之前的线程其实是 GreenThread（这个词就来源于 Java），但由于没实现多对多的调度，也就是没有真正实现并行，发挥不了多核的优势，所以后来改成基于系统内核的 Thread 实现了。

2. 某个系统线程如果被阻塞，排列在该线程上的 Goroutine 会被迁移。当然还有其他机制，比如 M 空闲了，如果全局队列没有任务，可能会从其他 M 偷任务执行，相当于一种 rebalance 机制。这里不再细说，有需要看专门的分析文章。

3. 具体的实现策略和我们前面分析的机制类似。系统启动时，会启动一个独立的后台线程（不在 Goroutine 的调度线程池里），启动 netpoll 的轮询。当有 Goroutine 发起网络请求时，网络库会将 fd（文件描述符）和 pollDesc（用于描述 netpoll 的结构体，包含因为读 / 写这个 fd 而阻塞的 Goroutine）关联起来，然后调用 runtime.gopark 方法，挂起当前的 Goroutine。当后台的 netpoll 轮询获取到 epoll（Linux 环境下）的 event，会将 event 中的 pollDesc 取出来，找到关联的阻塞 Goroutine，并进行恢复。

Goroutine 是银弹么?

Goroutine 很大程度上降低了并发的开发成本，是不是我们所有需要并发的地方直接 go func 就搞定了呢?

Go 通过 Goroutine 的调度解决了 CPU 利用率的问题。但遇到其他的瓶颈资源如何处理？比如带锁的共享资源，比如数据库连接等。互联网在线应用场景下，如果每个请求都扔到一个 Goroutine 里，当资源出现瓶颈的时候，会导致大量的 Goroutine 阻塞，最后用户请求超时。这时候就需要用 Goroutine 池来进行控流，同时问题又来了：池子里设置多少个 Goroutine 合适？

所以这个问题还是没有从更本上解决。

Actor 模型

Actor 对没接触过这个概念的人可能不太好理解，Actor 的概念其实和 OO 里的对象类似，是一种抽象。面对对象编程对现实的抽象是对象 = 属性 + 行为（method），但当使用方调用对象行为（method）的时候，其实占用的是调用方的 CPU 时间片，是否并发也是由调用方决定的。这个抽象其实和现实世界是有差异的。现实世界更像 Actor 的抽象，互相都是通过异步消息通信的。比如你对一个美女 say hi，美女是否回应，如何回应是由美女自己决定的，运行在美女自己的大脑里，并不会占用发送者的大脑。

所以 Actor 有以下特征：

- Processing - actor 可以做计算的，不需要占用调用方的 CPU 时间片，并发策略也是由自己决定。
- Storage - actor 可以保存状态。

- Communication - actor 之间可以通过发送消息通讯。

Actor 遵循以下规则：

- 发送消息给其他的 Actor
- 创建其他的 Actor
- 接受并处理消息，修改自己的状态

Actor 的目标：

- Actor 可独立更新，实现热升级。因为 Actor 互相之间没有直接的耦合，是相对独立的实体，可能实现热升级。
- 无缝弥合本地和远程调用 因为 Actor 使用基于消息的通讯机制，无论是和本地的 Actor，还是远程 Actor 交互，都是通过消息，这样就弥合了本地和远程的差异。
- 容错 Actor 之间的通信是异步的，发送方只管发送，不关心超时以及错误，这些都由框架层和独立的错误处理机制接管。
- 易扩展，天然分布式 因为 Actor 的通信机制弥合了本地和远程调用，本地 Actor 处理不过来的时候，可以在远程节点上启动 Actor 然后转发消息过去。

Actor 的实现：

- Erlang/OTP Actor 模型的标杆，其他的实现基本上都一定程度参照了 Erlang 的模式。实现了热升级以及分布式。

- Akka (Scala,Java) 基于线程和异步回调模式实现。由于 Java 中没有 Fiber，所以是基于线程的。为了避免线程被阻塞，Akka 中所有的阻塞操作都需要异步化。要么是 Akka 提供的异步框架，要么通过 Future-callback 机制，转换成回调模式。实现了分布式，但还不支持热升级。
- Quasar (Java) 为了解决 Akka 的阻塞回调问题，Quasar 通过字节码增强的方式，在 Java 中实现了 Coroutine/Fiber。同时通过 ClassLoader 的机制实现了热升级。缺点是系统启动的时候要通过 javaagent 机制进行字节码增强。

Golang CSP VS Actor

二者的格言都是：

> Don ' t communicate by sharing memory, share memory by communicating.

通过消息通信的机制来避免竞态条件，但具体的抽象和实现上有些差异。

- **CSP 模型里消息和 Channel 是主体，处理器是匿名的。**也就是说发送方需要关心自己的消息类型以及应该写到哪个 Channel，但不需要关心谁消费了它，以及有多少个消费者。Channel 一般都是类型绑定的，一个 Channel 只写同一种类

型的消息，所以 CSP 需要支持 alt/select 机制，同时监听多个 Channel。Channel 是同步的模式 (Golang 的 Channel 支持 buffer，支持一定数量的异步)，背后的逻辑是发送方非常关心消息是否被处理，CSP 要保证每个消息都被正常处理了，没被处理就阻塞着。

- **Actor 模型里 Actor 是主体，Mailbox (类似于 CSP 的 Channel) 是透明的。** 也就是说它假定发送方会关心消息发给谁消费了，但不关心消息类型以及通道。所以 Mailbox 是异步模式，发送者不能假定发送的消息一定被收到和处理。Actor 模型必须支持强大的模式匹配机制，因为无论什么类型的消息都会通过同一个通道发送过来，需要通过模式匹配机制做分发。它背后的逻辑是现实世界本来就是异步的，不确定 (non-deterministic) 的，所以程序也要适应面对不确定的机制编程。自从有了并行之后，原来的确定编程思维模式已经受到了挑战，而 Actor 直接在模式中蕴含了这点。

从这样看来，CSP 的模式比较适合 Boss-Worker 模式的任务分发机制，它的侵入性没那么强，可以在现有的系统中通过 CSP 解决某个具体的问题。它并不试图解决通信的超时容错问题，这个还是需要发起方进行处理。同时由于 Channel 是显式的，虽然可以通过 netchan (原来 Go 提供的 netchan 机制由于过于复杂，被废弃，在讨论新的 netchan) 实现远程 Channel，但很难做到对使用方透明。而 Actor 则是一种全新的抽象，使用 Actor 要面临整个应用架构机制和思维方式的变更。它试图要解决的问题要更广

一些，比如容错，比如分布式。但 Actor 的问题在于以当前的调度效率，哪怕是用 Goroutine 这样的机制，也很难达到直接方法调用的效率。当前要像 OO 的『一切皆对象』一样实现一个『一切皆 Actor』的语言，效率上肯定有问题。所以折中的方式是在 OO 的基础上，将系统的某个层面的组件抽象为 Actor。

再扯一下 Rust

Rust 解决并发问题的思路是首先承认现实世界的资源总是有限的，想彻底避免资源共享是很难的，不试图完全避免资源共享，它认为并发的问题不在于资源共享，而在于错误的使用资源共享。比如我们前面提到的，大多数语言定义类型的时候，并不能限制调用方如何使用，只能通过文档或者标记的方式（比如 Java 中的 @ThreadSafe, @NotThreadSafe annotation）说明是否并发安全，但也只能仅仅做到提示的作用，不能阻止调用方误用。虽然 Go 提供了 -race 机制，可以通过运行单元测试的时候带上这个参数来检测竞态条件，但如果你的单元测试并发度不够，覆盖面不到也检测不出来。所以 Rust 的解决方案就是：

- 定义类型的时候要明确指定该类型是否是并发安全的。
- 引入了变量的所有权（Ownership）概念 非并发安全的数据结构在多个线程间转移，也不一定就会导致问题，导致问题的是多个线程同时操作，也就是说是因为这个变量的所有权不明确导致的。有了所有权的概念后，变量只能由拥有所有权的作用域代码操作，

而变量传递会导致所有权变更，从语言层面限制了竞态条件出现的情况。

有了这机制，Rust 可以在编译期而不是运行期对竞态条件做检查和限制。虽然开发的时候增加了心智成本，但降低了调用方以及排查并发问题的心智成本，也是一种有特色的解决方案。

结论

革命尚未成功 同志任需努力。

本文带大家一起回顾了并发的问题，和各种解决方案。虽然各家有各家的优势以及使用场景，但并发带来的问题还远远没到解决的程度。所以还需努力，大家也有机会。

最后抛个砖，构想：在 Goroutine 上实现 Actor？

- **分布式**　解决了单机效率问题，是不是可以尝试解决下分布式效率问题？
- **和容器集群融合**　当前的自动伸缩方案基本上都是通过监控服务器或者 LoadBalancer，设置一个阀值来实现的。类似于我前面提到的喂饭的例子，是基于经验的方案，但如果系统内和外部集群结合，这个事情就可以做的更细致和智能。

- **自管理**　前面的两点最终的目标都是实现一个可以自管理的系统。做过系统运维的同学都知道，我们照顾系统就像照顾孩子一样，时刻要监控系统的各种状态，接受系统的各种报警，然后排查问题，进行紧急处理。孩子有长大的一天，那能不能让系统也自己成长，做到自管理呢？虽然这个目标现在看来还比较远，但我觉得是可以期待的。

想写好并发编程的工程师，建议扫码订阅王渊命个人公众号「午夜咖啡」，其定位是「工具・架构・成长・思考」，不但有深度的技术文章，也有对社会事件分析及个人成长的思考。■

Learning as we Go

Golang并发编程总结

作者/王渊命

网名Jolestar，曾任新浪微博架构师，微米技术总监。当前在创业中，做一款团队协作通讯工具Grouk。云与容器的深度实践者。喜欢尝试和折腾各种工具，从研发，测试，运维，到协作。希望能通过工具提高工作效率，降低研发以及协作成本，让工作和生活的节奏更自然。

趁着元旦放假，整理了一下最近学习Golang时，『翻译』的一个Golang的通用对象池，放到github (https://github.com/jolestar/go-commons-pool) 开源出来。之所以叫做『翻译』，是因为这个库的核心算法以及逻辑都是基于 Apache Commons Pool 的，只是把原来的Java『翻译』成了Golang。

前一段时间阅读kubernetes源码的时候，整体上学习了下Golang，但语言这种东西，学了不用，几个星期就忘差不多了。一次Golang实践群里聊天，有人问到Golang是否有通用的对象池，搜索了下，貌似没有比较完备的。当前Golang的pool有以下解决方案：

- sync.Pool
  sync.Pool 使用很简单，只需要传递一个创建对象的func即可。

```
var objPool = sync.Pool{
New: func() interface{} {
return NewObject()}}
p := objPool.Get().(*Object)
```

但sync.Pool只解决对象复用的问题，pool中的对象生命周期是两次gc之间，gc后pool中的对象会被回收，使用方不能控制对象的生命周期，所以不适合用在连接池等场景。

- 通过container/list来实现自定义的pool，比如redigo 就使用这种办法。但这些自定义的pool大多都不是通用的，功能也不完备。比如redigo当前没有获取连接池的超时机制，参看这个issue Blocking with timeout when Get PooledConn

而Java中的commons pool，功能比较完备，算法和逻辑也经过验证，使用也比较广泛，所以就直接『翻译』过来，顺便练习Golang的语法。

作为一个通用的对象池，需要包含以下主要功能：

1. 对象的生命周期可以精确控制 Pool提供机制允许使用方自定义对象的创建/销毁/校验逻辑。
2. 对象的存活数量可以精确控制 Pool提供设置存活数量以及时长的配置。
3. 获取对象有超时机制避免死锁，方便使用方实现failover 以前也遇到过许多线上故障，就是因为连接池的设置或者实现机制有缺陷导致的。

Apache Commons Pool的核心是基于LinkedBlockingDeque，idle对象都放在deque中。之所以是deque，而不是queue，是

因为它支持LIFO(last in, first out) /FIFO(first in, first out) 两种策略获取对象。然后有个包含所有对象的Map，key是用户自定义对象，value是PooledObject，用于校验Return Object的合法性，后台定时abandoned时遍历，计算活跃对象数等。超时是通过Java锁的wait timeout机制实现的。

下面总结下将Java翻译成Golang的时候遇到的多线程问题。

递归锁或者叫可重入锁(Recursive Lock)

Java中的synchronized关键词以及LinkedBlockingDequeu中用到的ReentrantLock，都是可重入的。而Golang中的sync.Mutex是不可重入的。表现出来就是：

```
ReentrantLock lock;

public void a(){
   lock.lock();
   //do some thing
   lock.unlock();
}

public void b(){
   lock.lock();
   //do some thing
   lock.unlock();
}

public void all(){
   lock.lock();
   //do some thing
```

```
    a();
    //do some thing
    b();
    //do some thing
    lock.unlock();
}
```

上例all方法中嵌套调用a方法，虽然调用a方法的时候也需要锁，但因为all已经申请锁，并且该锁可重入，所以不会导致死锁。而同样的代码在Golang中是会导致死锁的：

```
var lock sync.Mutex

func a() {
    lock.Lock()
    //do some thing
    lock.Unlock()
}

func b() {
    lock.Lock()
    //do some thing
    lock.Unlock()
}

func all() {
    lock.Lock()
    //do some thing
    a()
    //do some thing
    b()
    //do some thing
    lock.Unlock()
}
```

只能重构为下面这样的(命名不规范请忽略，只是demo)：

```
var lock sync.Mutex

func a() {
   lock.Lock()
   a1()
   lock.Unlock()
}

func a1() {
   //do some thing
}

func b() {
   lock.Lock()
   b1()
   lock.Unlock()
}

func b1() {
   //do some thing
}

func all() {
   lock.Lock()
   //do some thing
   a1()
   //do some thing
   b1()
   //do some thing
   lock.Unlock()
}
```

Golang的核心开发者认为可重入锁是不好的设计，所以不提供，参看Recursive (aka reentrant) mutexes are a bad idea。于是我们使用锁的时候就需要多注意嵌套以及递归调用。

锁等待超时机制

Golang的 sync.Cond 只有Wait，没有如Java中的Condition的超时等待方法await(long time, TimeUnit unit)。这样就没法实现LinkBlockingDeque的 pollFirst(long timeout, TimeUnit unit) 这样的方法。有人提了issue，但被拒绝了 sync: add WaitTimeout method to Cond。 所以只能通过channel的机制模拟了一个超时等待的Cond。完整源码参看 go-commons-pool/concurrent/cond.go。

```
type TimeoutCond struct {
  L       sync.Locker
  signal chan int
}

func NewTimeoutCond(l sync.Locker) *TimeoutCond {
  cond := TimeoutCond{L: l, signal: make(chan int, 0)}
  return &cond
}

/**
return remain wait time, and is interrupt
*/
func (this *TimeoutCond) WaitWithTimeout(timeout time.Duration) (time.
Duration, bool) {
  //wait should unlock mutex,  if not will cause deadlock
  this.L.Unlock()
  defer this.L.Lock()
  begin := time.Now().Nanosecond()
  select {
  case _, ok := <-this.signal:
        end := time.Now().Nanosecond()
```

```
        return time.Duration(end - begin), !ok
    case <-time.After(timeout):
        return 0, false
    }
}
```

Map机制的问题

这个问题严格的说不属于多线程的问题。虽然Golang的map不是线程安全的，但通过mutex封装一下也很容易实现。关键问题在于我们前面提到的，pool中用于维护全部对象的map，key是用户自定义对象，value是PooledObject。而Golang对map的key的约束是：go-spec#Map_types

> The comparison operators == and != must be fully defined for operands of the key type; thus the key type must not be a function, map, or slice. If the key type is an interface type, these comparison operators must be defined for the dynamic key values; failure will cause a run-time panic.

也就是说key中不能包含不可比较的值，比如 slice, map, and function。而我们的key是用户自定义的对象，没办法进行约束。于是借鉴Java的IdentityHashMap的思路，将key转换成对象的指针地址，实际上map中保存的是key对象的指针地址。

```
type SyncIdentityMap struct {
    sync.RWMutex
```

```
    m map[uintptr]interface{}
}

func (this *SyncIdentityMap) Get(key interface{}) interface{} {
    this.RLock()
    keyPtr := genKey(key)
    value := this.m[keyPtr]
    this.RUnlock()
    return value
}

func genKey(key interface{}) uintptr {
    keyValue := reflect.ValueOf(key)
    return keyValue.Pointer()
}
```

同时，这样做的缺点是Pool中存的对象必须是指针，不能是值对象。比如string，int等对象是不能保存到Pool中的。

其他的关于多线程的题外话

Golang的test -race 参数非常好用，通过这个参数，发现了几个data race的bug，参看commit fix data race test error。■

如何用Go实现支持数亿用户的长连消息系统

作者 / 周洋

360手机助手技术经理及架构师，负责360长连接消息系统，360手机助手架构的开发与维护。

360消息系统介绍

360消息系统更确切的说是长连接push系统，目前服务于360内部多个产品，开发平台数千款app，也支持部分聊天业务场景，单通道多app复用，支持上行数据，提供接入方不同粒度的上行数据和用户状态回调服务。

目前整个系统按不同业务分成9个功能完整的集群，部署在多个idc上（每个集群覆盖不同的idc），实时在线数亿量级。通常情况下，pc，手机，甚至是智能硬件上的360产品的push消息，基本上是从我们系统发出的。

关于push系统对比与性能指标的讨论

很多同行比较关心go语言在实现push系统上的性能问题，单机性能究竟如何，能否和其他语言实现的类似系统做对比么？甚至问如果是创业，第三方云推送平台，推荐哪个？

其实各大厂都有类似的push系统，市场上也有类似功能的云服务。包括我们公司早期也有erlang，nodejs实现的类似系统，也一度被公司要求做类似的对比测试。我感觉在讨论对比数据的时候，很难保证大家环境和需求的统一，我只能说下我这里的体会，数据是有的，但这个数据前面估计会有很多定语~

第一个重要指标：单机的连接数指标

做过长连接的同行，应该有体会，如果在稳定连接情况下，连接数这个指标，在没有网络吞吐情况下对比，其实意义往往不大，维持连接消耗cpu资源很小，每条连接tcp协议栈会占约4k的内存开销，系统参数调整后，我们单机测试数据，最高也是可以达到单实例300w长连接。但做更高的测试，我个人感觉意义不大。

因为实际网络环境下，单实例300w长连接，从理论上算压力就很大：实际弱网络环境下，移动客户端的断线率很高，假设每秒有1000分之一的用户断线重连。300w长连接，每秒新建连接达到3w，这同时连入的3w用户，要进行注册，加载离线存储等对内rpc调用，另外300w长连接的用户心跳需要维持，假设心跳300s一次，心跳包每秒需要1w tps。单播和多播数据的转发，广播数据的转发，本身也要响应内部的rpc调用，300w长连接情况下，gc带来的压力，内部接口的响应延迟能否稳定保障。这些集中在一个实例中，可用性是一个挑战。所以线上单实例不会hold很高的长连接，实际情况也要根据接入客户端网络状况来决定。

第二个重要指标:消息系统的内存使用量指标

这一点上，使用go语言情况下，由于协程的原因，会有一部分额外开销。但是要做两个推送系统的对比，也有些需要确定问题。比如系统从设计上是否需要全双工（即读写是否需要同时进行）如果半双工，理论上对一个用户的连接只需要使用一个协程即可（这种情况下，对用户的断线检测可能会有延时），如果是全双工，那读/写各一个协程。两种场景内存开销是有区别的。

另外测试数据的大小往往决定我们对连接上设置的读写buffer是多大，是全局复用的，还是每个连接上独享的，还是动态申请的。另外是否全双工也决定buffer怎么开。不同的策略，可能在不同情况的测试中表现不一样。

第三个重要指标：每秒消息下发量

这一点上，也要看我们对消息到达的QoS级别(回复ack策略区别)，另外看架构策略，每种策略有其更适用的场景，是纯粹推？还是推拉结合？甚至是否开启了消息日志？日志库的实现机制、以及缓冲开多大？flush策略……这些都影响整个系统的吞吐量。

另外为了HA，增加了内部通信成本，为了避免一些小概率事件，提供闪断补偿策略，这些都要考虑进去。如果所有的都去掉，那就是比较基础库的性能了。

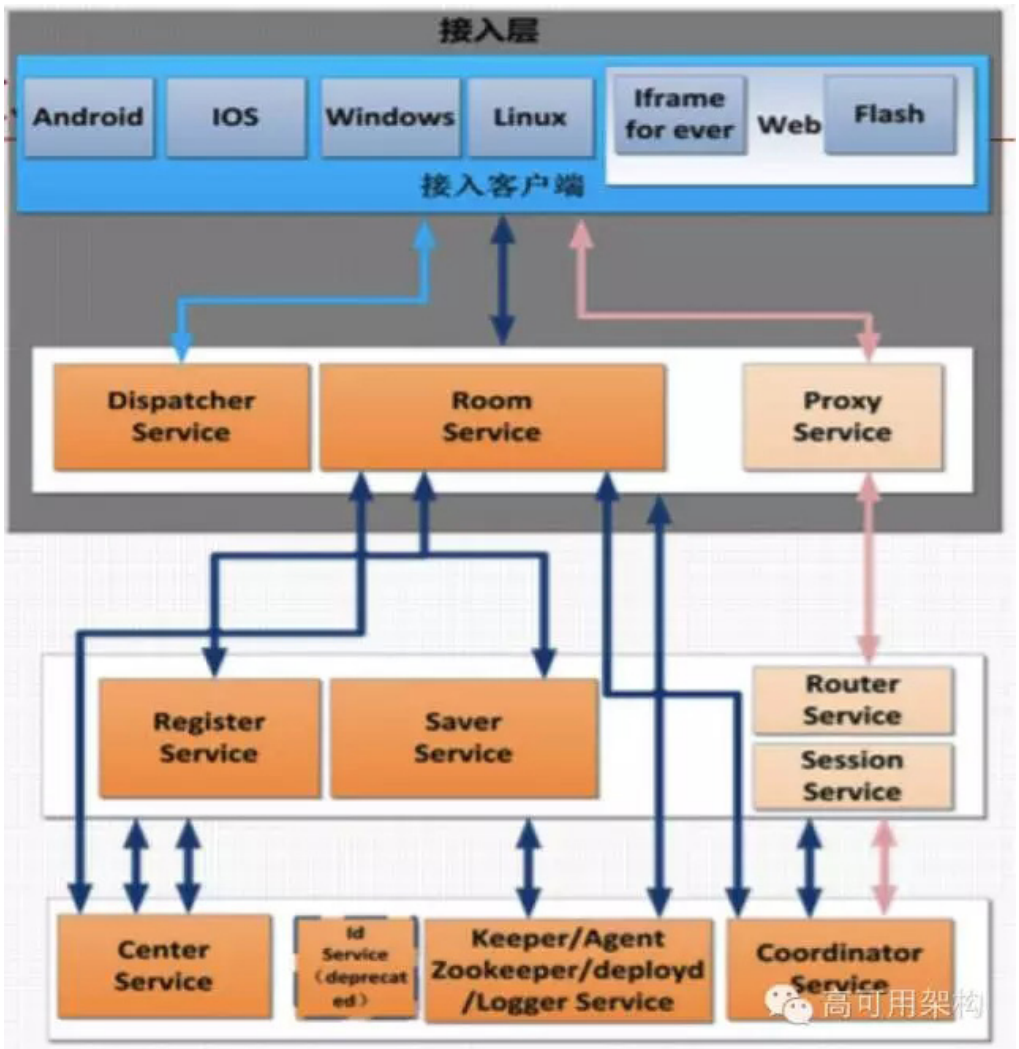
所以我只能给出大概数据，24核，64G的服务器上，在QoS为message at least，纯粹推，消息体256B~1kB情况下，单个实例100w实际用户（200w+）协程，峰值可以达到2~5w的QPS...内存可以稳定在25G左右，gc时间在200~800ms左右（还有优化空间）。

我们正常线上单实例用户控制在80w以内，单机最多两个实例。事实上，整个系统在推送的需求上，对高峰的输出不是提速，往往是进行限速，以防push系统瞬时的高吞吐量，转化成对接入方业务服务器的ddos攻击所以对于性能上，我感觉大家可以放心使用，至少在我们这个量级上，经受过考验，go1.5到来后，确实有之前投资又增值了的感觉。

消息系统架构介绍

下面是对消息系统的大概介绍，之前一些同学可能在gopher china上可以看到分享，这里简单讲解下架构和各个组件功能，额外补充一些当时遗漏的信息：

架构图如下，所有的service都 written by golang。



几个大概重要组件介绍如下：

- dispatcher service 根据客户端请求信息，将应网络和区域的长连接服务器的，一组IP传送给客户端。客户端根据返回的IP，建立长连接，连接Room service。

- room Service，长连接网关，hold用户连接，并将用户注册进register service，本身也做一些接入安全策略、白名单、IP限制等。
- register service 是我们全局session存储组件，存储和索引用户的相关信息，以供获取和查询。
- coordinator service 用来转发用户的上行数据，包括接入方订阅的用户状态信息的回调，另外做需要协调各个组件的异步操作，比如kick用户操作，需要从register拿出其他用户做异步操作.
- saver service是存储访问层，承担了对redis和mysql的操作，另外也提供部分业务逻辑相关的内存缓存，比如广播信息的加载可以在saver中进行缓存。另外一些策略，比如客户端sdk由于被恶意或者意外修改，每次加载了消息，不回复ack，那服务端就不会删除消息，消息就会被反复加载，形成死循环，可以通过在saver中做策略和判断。(客户端总是不可信的)。
- center service 提供给接入方的内部api服务器，比如单播或者广播接口，状态查询接口等一系列api,包括运维和管理的api。
- deployd/agent service 用于部署管理各个进程，收集各组件的状态和信息,zookeeper和keeper用于整个系统的配置文件管理和简单调度

举两个常见例子，了解工作机制：比如发一条单播给一个用户，center先请求Register获取这个用户之前注册的连接通道标识、room实例地址，通过room service下发给长连接Center

Service比较重的工作如全网广播，需要把所有的任务分解成一系列的子任务，分发给所有center，然后在所有的子任务里，分别获取在线和离线的所有用户，再批量推到Room Service。通常整个集群在那一瞬间压力很大。

推送的服务端架构

常见的推送模型有长轮训拉取，服务端直接推送（360消息系统目前主要是这种），推拉结合（推送只发通知，推送后根据通知去拉取消息）。拉取的方式不说了，现在并不常用了，早期很多是nginx+lua+redis，长轮训，主要问题是开销比较大，时效性也不好，能做的优化策略不多。

直接推送的系统，目前就是360消息系统这种，消息类型是消耗型的，并且对于同一个用户并不允许重复消耗，如果需要多终端重复消耗，需要抽象成不同用户。

推的好处是实时性好，开销小，直接将消息下发给客户端，不需要客户端走从接入层到存储层主动拉取.但纯推送模型，有个很大问题，由于系统是异步的，他的时序性无法精确保证。这对于push需求来说是够用的，但如果复用推送系统做im类型通信，可能并不合适。

对于严格要求时序性，消息可以重复消耗的系统，目前也都是走推拉结合的模型，就是只使用我们的推送系统发通知，并附带id等给客户端做拉取的判断策略，客户端根据推送的key，主动从业务服务器拉取消息。并且当主从同步延迟的时候，跟进推送的key做延

迟拉取策略。同时也可以通过消息本身的QoS，做纯粹的推送策略，比如一些“正在打字的”低优先级消息，不需要主动拉取了，通过推送直接消耗掉。

哪些因素决定推送系统的效果？

首先是sdk的完善程度，sdk策略和细节完善度，往往决定了弱网络环境下最终推送质量。

SDK选路策略，最基本的一些策略如下：有些开源服务可能会针对用户hash一个该接入区域的固定ip，实际上在国内环境下不可行，最好分配器（dispatcher）是返回散列的一组，而且端口也要参开，必要时候，客户端告知是retry多组都连不上，返回不同idc的服务器。因为我们会经常检测到一些case，同一地区的不同用户，可能对同一idc内的不同ip连通性都不一样，也出现过同一ip不同端口连通性不同，所以用户的选路策略一定要灵活，策略要足够完善.另外在选路过程中，客户端要对不同网络情况下的长连接ip做缓存，当网络环境切换时候（wifi、2G、3G），重新请求分配器，缓存不同网络环境的长连接ip。

客户端对于数据心跳和读写超时设置，完善断线检测重连机制。

针对不同网络环境，或者客户端本身消息的活跃程度，心跳要自适应的进行调整并与服务端协商，来保证链路的连通性。并且在弱网络环境下，除了网络切换（wifi切3G）或者读写出错情况，什么时候重新建立链路也是一个问题。客户端发出的ping包，不同网络下，

多久没有得到响应，认为网络出现问题，重新建立链路需要有个权衡。另外对于不同网络环境下，读取不同的消息长度，也要有不同的容忍时间，不能一刀切。好的心跳和读写超时设置，可以让客户端最快的检测到网络问题，重新建立链路，同时在网络抖动情况下也能完成大数据传输。

结合服务端做策略。

另外系统可能结合服务端做一些特殊的策略，比如我们在选路时候，我们会将同一个用户尽量映射到同一个room service实例上。断线时，客户端尽量对上次连接成功的地址进行重试。主要是方便服务端做闪断情况下策略，会暂存用户闪断时实例上的信息，重新连入的 时候，做单实例内的迁移，减少延时与加载开销.

客户端保活策略。

很多创业公司愿意重新搭建一套push系统，确实不难实现，其实在协议完备情况下（最简单就是客户端不回ack不清数据），服务端会保证消息是不丢的。但问题是为什么在消息有效期内,到达率上不去？往往因为自己app的push service存活能力不高。选用云平台或者大厂的，往往sdk会做一些保活策略，比如和其他app共生，互相唤醒，这也是云平台的push service更有保障原因。我相信很多云平台旗下的sdk，多个使用同样sdk的app，为了实现服务存活，是可以互相唤醒和保证活跃的。另外现在push sdk本身是单连接，多app复用的，这为sdk实现，增加了新的挑战。

综上，对我来说，选择推送平台，优先会考虑客户端sdk的完善程度。对于服务端，选择条件稍微简单，要求部署接入点（IDC）越要多，配合精细的选路策略，效果越有保证，至于想知道哪些云服务有多少点，这个群里来自各地的小伙伴们，可以合伙测测。

go语言开发问题与解决方案

下面讲下，go开发过程中遇到挑战和优化策略，给大家看下当年的一张图，在第一版优化方案上线前一天截图：

可以看到，内存最高占用69G，GC时间单实例最高时候高达3~6s.这种情况下，试想一次悲剧的请求，经过了几个正在执行gc的组件，后果必然是超时... gc照成的接入方重试，又加重了系统的负担。遇到这种情况当时整个系统最差情况每隔2，3天就需要重启一次。

当时出现问题，现在总结起来，大概以下几点：

1.散落在协程里的I/O，Buffer和对象不复用

当时（12年）由于对go的gc效率理解有限，比较奔放，程序里大量short live的协程，对内通信的很多io操作，由于不想阻塞主循环逻辑或者需要及时响应的逻辑，通过单独go协程来实现异步。这回会gc带来很多负担。 针对这个问题，应尽量控制协程创建，对于长连接这种应用，本身已经有几百万并发协程情况下，很多情况没必要在各个并发协程内部做异步io，因为程序的并行度是有限，理论上做协程内做阻塞操作是没问题。 如果有些需要异步执行，比如如果不异步执行，影响对用户心跳或者等待response无法响应，最好通过一个任务池，和一组常驻协程，来消耗，处理结果，通过channel再传回调用方。使用任务池还有额外的好处，可以对请求进行打包处理，提高吞吐量，并且可以加入控量策略.

2.网络环境不好引起激增

go协程相比较以往高并发程序，如果做不好流控，会引起协程数量激增。早期的时候也会发现，时不时有部分主机内存会远远大于其他服务器，但发现时候，所有主要profiling参数都正常了。

后来发现，通信较多系统中，网络抖动阻塞是不可免的（即使是内网），对外不停accept接受新请求，但执行过程中，由于对内通信阻塞，大量协程被创建，业务协程等待通信结果没有释放，往往瞬时会迎来协程暴涨。但这些内存在系统稳定后，virt和res都并没能彻底释放，下降后，维持高位。

处理这种情况，需要增加一些流控策略，流控策略可以选择在rpc库来做，或者上面说的任务池来做，其实我感觉放在任务池里做更合理些，毕竟rpc通信库可以做读写数据的限流，但它并不清楚具体的限流策略，到底是重试还是日志还是缓存到指定队列。任务池本身就是业务逻辑相关的，它清楚针对不同的接口需要的流控限制策略。

3. 低效和开销大的rpc框架

早期rpc通信框架比较简单，对内通信时候使用的也是短连接。这本来短连接开销和性能瓶颈超出我们预期，短连接io效率是低一些，但端口资源够，本身吞吐可以满足需要，用是没问题的，很多分层的系统，也有http短连接对内进行请求的。但早期go版本，这样写程序，在一定量级情况，是支撑不住的。短连接大量临时对象和临时buffer创建，在本已经百万协程的程序中，是无法承受的。所以后续我们对我们的rpc框架作了两次调整。

第二版的rpc框架，使用了连接池，通过长连接对内进行通信（复用的资源包括client和server的：编解码Buffer、Request/response），大大改善了性能。

但这种在一次request和response还是占用连接的，如果网络状况ok情况下，这不是问题，足够满足需要了，但试想一个room实例要与后面的数百个的register，coordinator，saver，center，keeper实例进行通信，需要建立大量的常驻连接，每个目标机几十个连接，也有数千个连接被占用。

非持续抖动时候（持续逗开多少无解），或者有延迟较高的请求时候，如果针对目标ip连接开少了，会有瞬时大量请求阻塞，连接无法得到充分利用。第三版增加了Pipeline操作，Pipeline会带来一些额外的开销，利用tcp的全双特性，以尽量少的连接完成对各个服务集群的rpc调用。

4. Gc时间过长

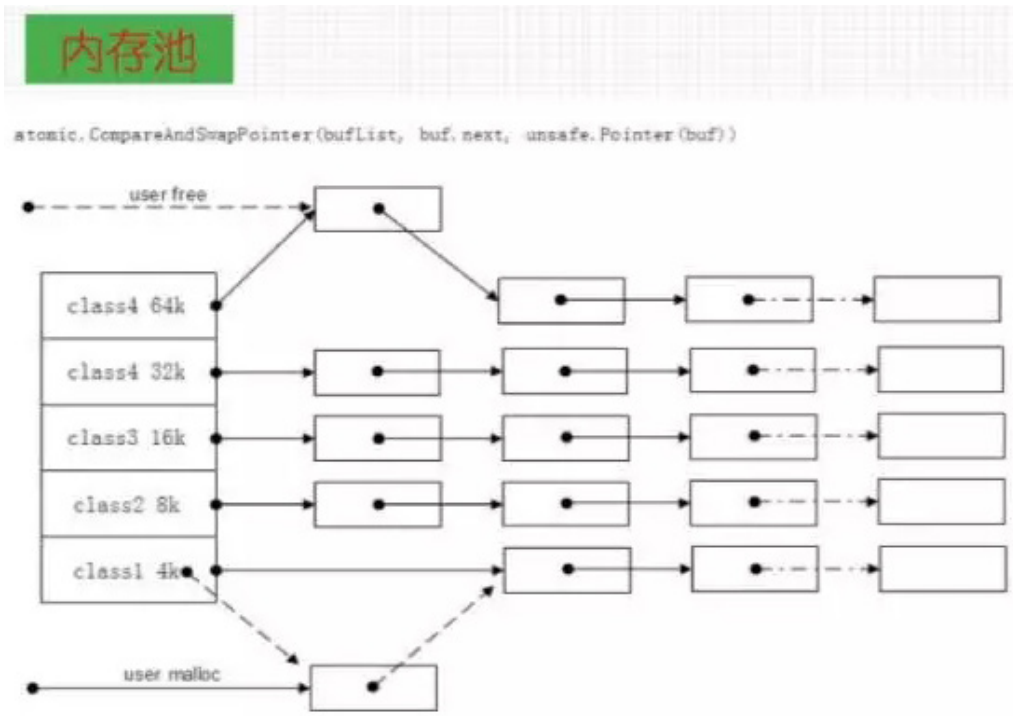
Go的Gc仍旧在持续改善中，大量对象和buffer创建，仍旧会给gc带来很大负担，尤其一个占用了25G左右的程序。之前go team的大咖邮件也告知我们，未来会让使用协程的成本更低，理论上不需要在应用层做更多的策略来缓解gc。

改善方式，一种是多实例的拆分，如果公司没有端口限制，可以很快部署大量实例，减少gc时长，最直接方法。不过对于360来说，外网通常只能使用80和433。因此常规上只能开启两个实例。当然很多人给我建议能否使用SO_REUSEPORT，不过我们内核版本确实比较低，并没有实践过。

另外能否模仿nginx，fork多个进程监控同样端口，至少我们目前没有这样做，主要对于我们目前进程管理上，还是独立的运行的，对外监听不同端口程序，还有配套的内部通信和管理端口，实例管理和升级上要做调整。

解决gc的另两个手段，是内存池和对象池，不过最好做仔细评估和测试，内存池、对象池使用，也需要对于代码可读性与整体效率进行权衡。

这种程序一定情况下会降低并行度，因为用池内资源一定要加互斥锁或者原子操作做CAS，通常原子操作实测要更快一些。CAS可以理解为可操作的更细行为粒度的锁（可以做更多CAS策略，放弃运行，防止忙等）。这种方式带来的问题是，程序的可读性会越来越像C语言，每次要malloc，各地方用完后要free，对于对象池free之前要reset，我曾经在应用层尝试做了一个分层次结构的“无锁队列”。



上图左边的数组实际上是一个列表，这个列表按大小将内存分块，然后使用atomic操作进行CAS。但实际要看测试数据了，池技术可以明显减少临时对象和内存的申请和释放，gc时间会减少，但加锁带来的并行度的降低，是否能给一段时间内的整体吞吐量带来提升，要做测试和权衡…

在我们消息系统，实际上后续去除了部分这种黑科技，试想在百万个协程里面做自旋操作申请复用的buffer和对象，开销会很大，尤其在协程对线程多对多模型情况下，更依赖于golang本身调度策略，除非我对池增加更多的策略处理，减少忙等，感觉是在把runtime做的事情，在应用层非常不优雅的实现。普遍使用开销理论就大于收益。

但对于rpc库或者codec库，任务池内部，这些开定量协程，集中处理数据的区域，可以尝试改造。

对于有些固定对象复用，比如固定的心跳包什么的，可以考虑使用全局一些对象，进行复用，针对应用层数据，具体设计对象池，在部分环节去复用，可能比这种无差别的设计一个通用池更能进行效果评估。

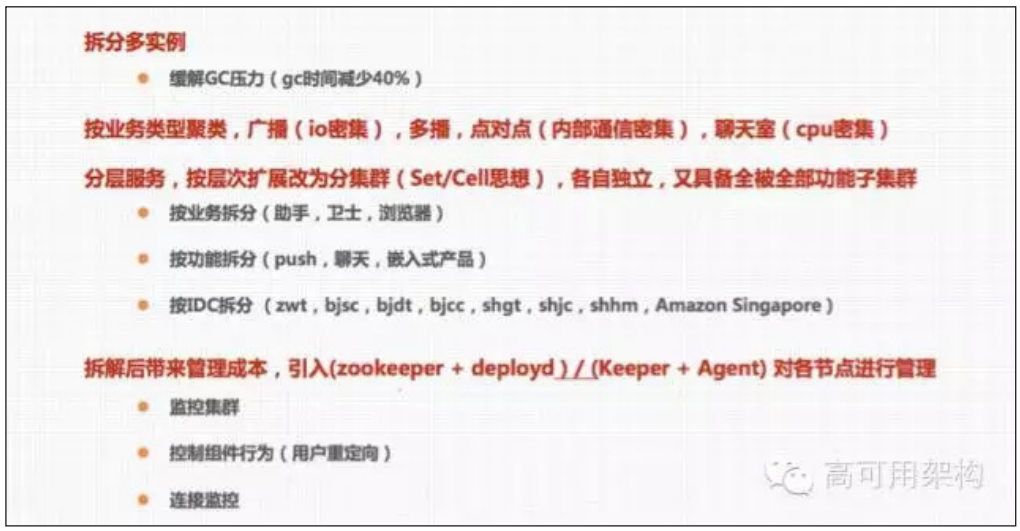
消息系统的运维及测试

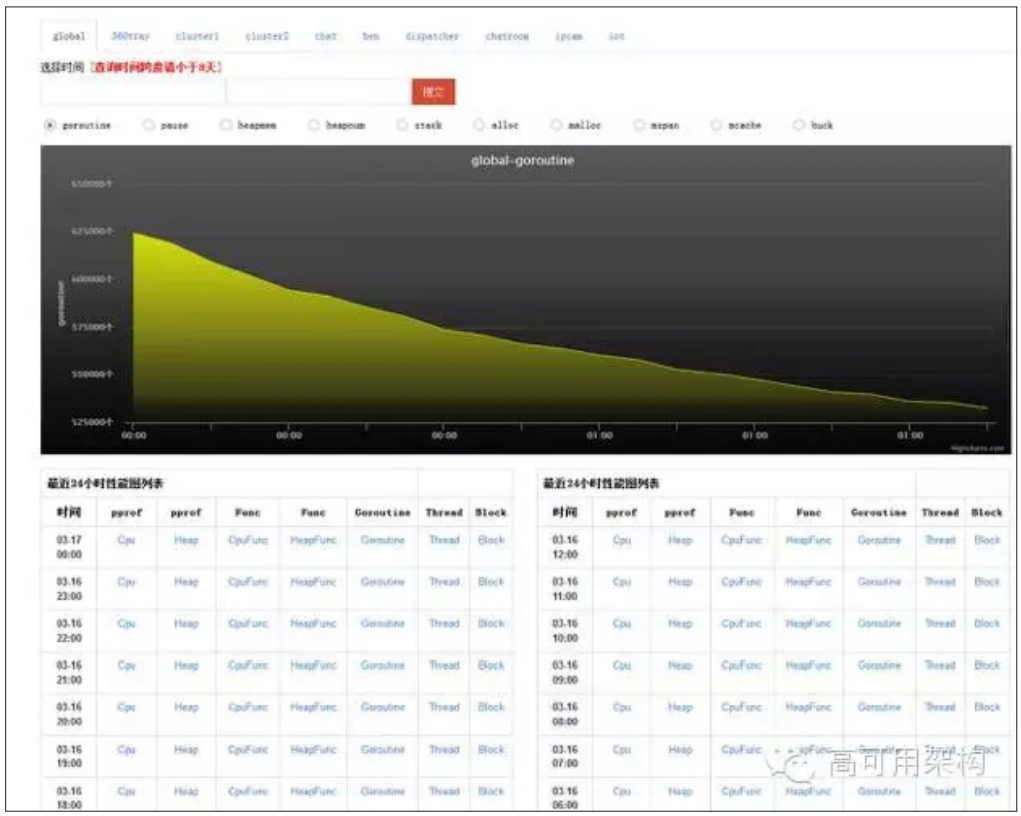
下面介绍消息系统的架构迭代和一些迭代经验，由于之前在其他地方有过分享，后面的会给出相关链接，下面实际做个简单介绍，感兴趣可以去链接里面看。

架构迭代~根据业务和集群的拆分，能解决部分灰度部署上线测试，减少点对点通信和广播通信不同产品的相互影响，针对特定的功能做独立的优化。

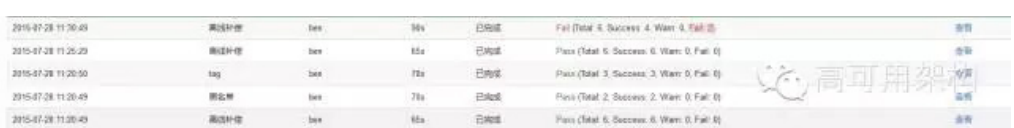
消息系统架构和集群拆分，最基本的是拆分多实例，其次是按照业

务类型对资源占用情况分类，按用户接入网络和对idc布点要求分类（目前没有条件，所有的产品都部署到全部idc）。





系统的测试go语言在并发测试上有独特优势。



对于压力测试，目前主要针对指定的服务器，选定线上空闲的服务器做长连接压测。然后结合可视化，分析压测过程中的系统状态。但压测早期用的比较多，但实现的统计报表功能和我理想有一定差距。我觉得最近出的golang开源产品都符合这种场景，go写网络并发程序给大家带来的便利，让大家把以往为了降低复杂度，拆解或者分层协作的组件，又组合在了一起。■

Q&A

Q1:协议栈大小，超时时间定制原则?

移动网络下超时时间按产品需求通常2g，3G情况下是5分钟，wifi情况下5~8分钟。但对于个别场景，要求响应非常迅速的场景，如果连接idle超过1分钟，都会有ping，pong，来校验是否断线检测，尽快做到重新连接。

Q2:消息是否持久化?

消息持久化，通常是先存后发，存储用的redis，但落地用的mysql。mysql只做故障恢复使用。

Q3:协议栈是基于tcp吗?

是否有协议拓展功能？协议栈是tcp，整个系统tcp长连接，没有考虑扩展其功能~如果有好的经验，可以分享。

Q4: 这个系统是接收上行数据的吧，系统接收上行数据后是转发给相应系统做处理么，是怎么转发呢，如果需要给客户端返回调用结果又是怎么处理呢?

系统上行数据是根据协议头进行转发，协议头里面标记了产品和转发类型，在coordinator里面跟进产品和转发类型，回调用户，如果用户需要阻塞等待回复才能后续操作，那通过再发送消息，路由回用户。因为整个系统是全异步的。

Q5: 生产系统的profiling是一直打开的么?

不是一直打开，每个集群都有采样，但需要开启哪个可以后台控制。这个profling是通过接口调用。

Q6: 面前系统中的消息消费者可不可以分组? 类似于Kafka。

客户端可以订阅不同产品的消息，接受不同的分组。接入的时候进行bind或者unbind操作。

Q7: 流控问题有排查过网卡配置导致的idle问题吗?

流控是业务级别的流控，我们上线前对于内网的极限通信量做了测试，后续将请求在rpc库内，控制在小于内部通信开销的上限以下.在到达上限前作流控。

Q8: 服务的协调调度为什么选择zk有考虑过raft实现吗？golang的raft实现很多啊，比如Consul和ectd之类的。

3年前，还没有后两者或者后两者没听过应该。zk当时公司内部成熟方案，不过目前来看，我们不准备用zk作结合系统的定制开发，准备用自己写的keeper代替zk，完成配置文件自动转数据结构，数据结构自动同步指定进程，同时里面可以完成很多自定义的发现和控制策略，客户端包含keeper的sdk就可以实现以上的所有监控数据，profling数据收集，配置文件更新，启动关闭等回调。完全抽象成语keeper通信sdk，keeper之间考虑用raft。

Q9: 负载策略是否同时在服务侧与CLIENT侧同时做的 (DISPATCHER 会返回一组IP)？ 另外，ROOM SERVER/REGISTER SERVER连接状态的一致性 | 可用性如何保证？服务侧保活有无特别关注的地方？安全性方面是基于TLS再加上应用层加密？

会在server端做，比如重启操作前，会下发指令类型消息，让客户端进行主动行为。部分消息使用了加密策略，自定义的rsa+des，另外满足我们安全公司的需要，也定制开发很多安全加密策略。一致性是通过冷备解决的，早期考虑双写，但实时状态双写同步代价太高而且容易有脏数据，比如register挂了，调用所有room，通过重新刷入指定register来解决。

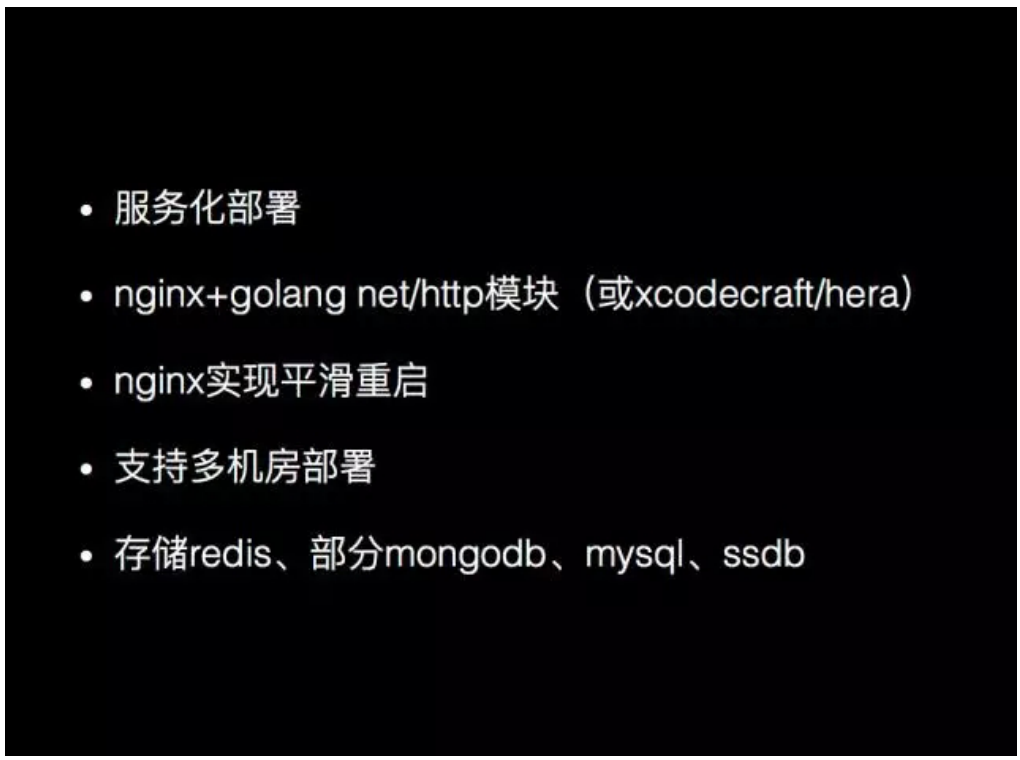
Golang在视频直播平台的高性能实践

作者／杨武明

熊猫 TV 首席架构师，曾担任奇虎 360 PC 网游技术架构负责人，前新浪微博平台资深后端开发、技术专家。对大型互联网架构有丰富的实践经验，擅长后端基础服务与组件开发，尤其高性能、高并发、大数据业务场景。

熊猫 TV 是一家视频直播平台，先介绍下我们系统运行的环境，下面这 6 大服务只是我们几十个服务中的一部分，由于并发量与重要性比较高，所以成为 golang 小试牛刀的首批高性能高并发服务。

把大服务拆细，然后服务化独立部署，更容易简化部署，也容易单点细节优化与升级。多数服务的能力是通用的，如平滑重启、多机房部署等。



关于在线服务，如果 1000 万用户在线，即使每 30 秒一次在线 http 发送打点，那么也有每秒 30 万+的接口 qps。

Golang 提供的 http server 性能非常好，一个普通工程师如果做到单虚机支撑 1w - 2w 每秒的请求量，那么只需要 10 - 20 台后台服务器了。

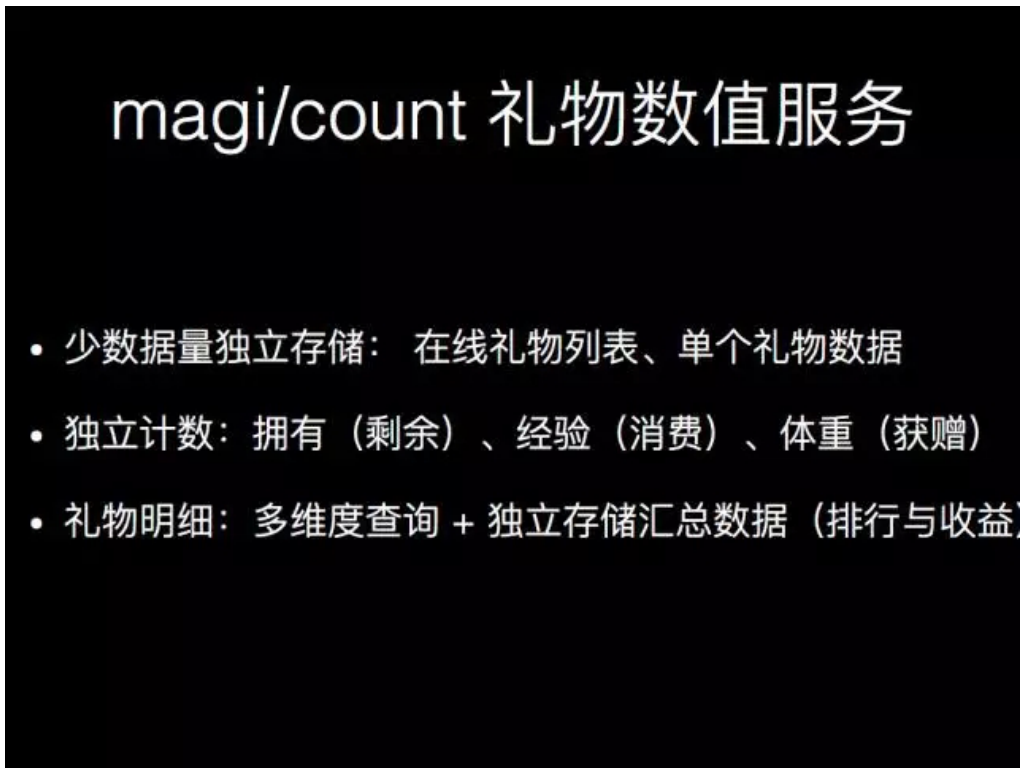
用其他非高性能语言，需要的机器能需要多翻好几倍；C++ 性能足够，但对工程师要求很高。

房间服务是主要服务，需要核心保障，所以对房间服务里面的版块，进行了再细粒度的拆分。

对房间信息变更频繁的内容也采用独立存储，如人数计数等字段，这样的缺点是每次去房间信息会多带来一次额外的访问人数服务的成本。这也是架构设计需要权衡点之一。

礼物系统是多数视频直播平台的标配，国内有礼物消费习惯的土豪不少，高峰期送礼物的并发量很大，尤其很多土豪对刷的时候。

礼物系统对一致性要求略高，所以存一份数据建多条索引也是一种选择，也可以降低对事务的依赖。



弹幕交互方式是一个很不错的体验，更偏年轻化，大量用户喜欢通过这种方式与主播互动。

国内网络状况比较复杂，最好根据用户位置选择就近对应运营商的单线机房接入弹幕消息服务，让弹幕更及时。也可以用 BGP 机房，但网络带宽价格会比单线贵不少。

对于大房间，弹幕消息量特别大，主播与用户都看不过来，在产品策略层面可以做一些体验上的优化。

视频直播体验是整体平台最关键点，视频 CDN 是一种让用户就近获取所需流媒体的技术，且解决延时、卡顿等问题。 从技术角度不要只依赖某一 CDN 提供商或线路，业内很多视频直播平台都做了拉流线路互备，推流后视频（转码、转发）集群也是一个可优化的点，做互推是冗余了部分推流资源，但对优化直播流的体验与高可用性显而易见。

包含自由业务逻辑的接口代理服务，是很多公司都有的一个公共服务，可以把公司内网服务低成本的提供来外网来使用。不过需要考虑好外网安全策略，包括授权认证、服务关系映射、频率限制、业务资源隔离等。

能看到最后肯定是对我们技术及业务感兴趣的朋友，熊猫 TV 技术团队正在招聘 Golang 工程师，全部配备高配电脑与 DELL 双屏大屏幕显示器，并提供有挑战性的技术环境。感兴趣朋友请扫码进入（同时也有大量 PHP、前端、运维、DBA 等技术岗位）。■

点击下载PPT

Learning as we Go

杨武明：

从3000元月薪码农到首席架构师

杨武明，熊猫 TV 首席架构师，曾担任奇虎 360 PC 网游技术架构负责人，前新浪微博平台资深后端开发、技术专家。对大型互联网架构有丰富的实践经验，擅长后端基础服务与组件开发，尤其高性能、高并发、大数据业务场景。本文是杨武明在高可用架构群新年聚会暨架构开源研讨会上的演讲。

我在 2007 年毕业，和很多同行一样，也是从小公司起步，曾做过 ERP，信息安全，互联网社交平台、游戏平台等，现在工作是直播娱乐平台。写过几年的语言有 Java、C/C++、PHP、Golang，偶尔写点 Python、Lua、NodeJS 等。花在开源上的时间不多，但也开源一些个人代码，如分布式发号器。我现在负责熊猫 TV 整体基础架构工作，是一个快 9 年工龄的码农。

在我讲我个人成长经历前，我先推荐一本书《联盟》，LinkedIn 领英创始人里德 · 霍夫曼的作品。在《联盟》中，提供了一种使雇主与员工之间从商业交易转变为互惠关系的框架，创建了一种鼓励公司和个人相互投资的工作模式。它提出打造任期制，将非终身雇用的员工变为公司的长期人脉，并吸收员工的高效人脉情报。

只有雇主与雇员结为强大的联盟，共同拥有持续的创新与丰富的智慧宝库，员工、团队、企业，乃至整个经济才能繁荣发展。

每个人对于价值观有不同的理解，我个人对于人生幸福理解很简单：年轻时有人生阅历丰富的人（下面我姑且称之为长者）指导，跟随有理想的长者去学习及改变世界；到自己成为长者时，也同样能将相同的价值观及做事方法影响一批人，聚拢一批有志青年来一起做有意义的事情。我说的长者不是指年龄，还是指在人生阅历及行业领域有深入见解的人。在这个时代，单枪匹马很难成功，我更看重团体的力量及跟优秀的人一起做事情。

长者能通过集市模式聚到一批与同样兴趣与尿点的人，这些人每天

共同生活做事：吃饭、做事；吃饭、讨论；吃饭、学习打豆豆、睡觉，找谁是豆豆。这个尿点这重要，有同样尿点也意味着有更多一起接触的机会。有些码农喜欢白天写代码，有些码农喜欢晚上写代码，这个尿点不一致，可能就少了很多交流的机会。一起生活只是一个比喻，重点是通过长者大家一起找到共同做事的节奏。Linux 操作系统就是在长者号召的背景下，产生于这种貌似混乱、无序的集市模式。

我的观察国内缺少这种长者，大部分团队都以商业为重。在国外，硅谷科技在几十年前通过创新走在世界前列，很大程度依赖这种土壤；几百年前，欧洲的文艺复兴也依附于这份土壤。有机会我要去佛罗伦萨圣母百花大教堂与硅谷湾区看看，去感受那些当时能影响一个时代的人物。

和优秀的人在一起

刚毕业时，放弃了父母在家乡的安排及强烈反对来到了北京，以码农的身份及 3,000 元月薪加入到电讯盈科北京分公司的企业方案部门做 ERP，盈科是一家香港电信公司，可以理解成香港的中国移动。第二份工作是在神州泰岳做信息安全方面开发，在神州泰岳的经历如果说收获较多的，可能是信息安全的项目对代码质量要求严格，自己打好了较好的编程习惯与基础。

转眼就毕业三年过去，到了 2010 年时候开始有些郁郁寡欢，觉得自己往上进一步成长非常慢，也看不到未来的出路，感觉企业信

息安全领域开发不是我长期想要的技术生活。但在另外一方面，我也观察到国内的互联网行业及技术都发展非常蓬勃，觉得它应该是技术人员未来的方向。因此也憧憬着去尝试海量用户规模互联网平台的技术挑战，希望能做一些更大的、能影响整个互联网用户的事情。

在迷茫的路口，我开始关注互联网技术，也阅读了很多互联网技术大牛的技术书籍及文章，包括有幸全盘拜读 Tim 的博客。一次偶然的机会，花了 100 元（谁说技术人员一毛不拔）报名了 CSDN TUP 技术沙龙，并聆听了 Tim 当时在会议上的演讲。会议结束后我在电梯口联系 Tim 表达了求职意向，Tim 当时给了一份考试题让我回去试试。

跟很多做毕业后一直从事企业开发的同行一样，当打算转入互联网行业时候，普遍碰到经验不足及资历不够的问题。我幸好有之前几年打好的扎实编程基础，因此提交的考题代码还不错，获得了微博面试的机会。在自己资历还比较普通的情况下，果断对自己身价清仓出血甩卖，也同样出于对团队的向往，面试时说只要能加入给多少钱都行（但码农通常也都很现实，面试完回家后就对自己提出的工资后悔了）。就在这不计较个人工资多少的情况下（也是个人为数不多的一次跳槽不大幅度涨薪），果断加入了刚处在风口不久的微博技术团队。总的来说，这次转换于我个人这是一次全新的开始，我可以开始做自己喜欢且擅长的事：网络服务器与高并发系统。

加入微博平台团队后，开始适应互联网团队在我看来全新的开发模式。一上岗发现已俨然进入摆好台的手术室，Tim 是教授院长，提出了

技术前进的整体方向。田大师就是专家主刀医生，主控一个大的模块的架构设计，剩下就等着我这个护士递手术刀 (写代码)。一开始我参与了 firehose-stream 项目的开发，这是一个管理微博所有内部数据的实时数据流服务，每秒实时推送数万条数据，包括微博、评论、私信等消息及事件。当时也是紧张而又刺激，在业务飞速发展及访问量剧增的背景下解决了上线后很多问题。

经过这个项目适应后，我开始接触千万级用户访问的平台，如每秒几万、甚至十万以上的全站提醒、通知、导航、邀请等系统。每个系统除了自己摸索，以及工作有田大师指教外，团队也提供了非常好的交流氛围。每周都有固定的时间，大家在一起讨论及交流技术。Tim 也每周拿出一些在架构领域有代表性的场景跟团队小伙伴一起探讨解决方案。除了平台有这么大的数据规模及用户访问量的环境给大家历练之外，我觉得平台的技术学习氛围这也是我当时成长较快的原因。

在微博期间做了很多项目，包括也有幸参与到核心 feed 系统的开发。几年的历练下来。慢慢从一个积极能动性型码农，成长为在高性能高并发领域略有心得的技术专家。

快乐时光总是过得快，转眼到微博又快 3 年。有一天突然发现自己又出现了原先郁郁寡欢的心慌，隐约感觉自己是希望下一步有机会再次做一些不同的事情。事后回想也许是自己在一定程度想从跟随者到召集者角色的转变。在微博平台的团队中，跑在前面的都是从技术到思想都非常优秀的人，在短时间内我不太可能有机会超越这些前辈转变到召集者的角色。

当时的心慌也可以用一句古话来描述，人无远虑必有近忧，这是不是说明我还是远虑者（现场鼓掌）。考虑到自己未来期望的转变，决定离开微博去创业，去主导及影响一个小范围领域的事情。走的时候心情也非常纠结，当时国内有氛围且有挑战，能收容有技术情节码农的技术团队不多，有些东西一旦失去可能再也回不去了。

不舍不只是微博平台本身，更多是有幸结识了一帮趣味相投资深码农，离开以后很难再有这么一个团队来成长了。包括有钢琴艺术气息的田大师、有时间洗千亿级数据（小编注：由于架构升级进行的迁移数据）而没时间洗澡的小军、算法、棋艺、运动与于一身的小麦、长胡子艺术家气息的老王、帅气的一乐、少年班福林、低调的朱总、呵护多年新同学成长的校长、国际范 James Wei、德州赌神刀刀、托马斯海涛、兵乓球及架构高手姚老板、烤鸭老板方圆等，这些好朋友今天大多数也来到了现场。包括还在微博战斗以及散落在各大互联网公司担任要职的好朋友，我就不一一点名了。想想都是幸运，成天跟一群这么优秀的人在一起，想变差都不容易。

但自己也需要跳出这个舒适区，走的时候回望奋斗过三年的理想国际大厦时，决定自己将来也要打造出一支有技术范与战斗力，同时能服务于社会并带来商业价值的工程团队，同时实现财务自由。

走出舒适区

但创业的现实很骨感。创业失败看来是必然，里面有些隐私的因素就不在这里细说。创业公司虽然失败，但这过程中我经历了角色转

换拐点，由成熟大公司的技术专家变为创业小公司的产品技术负责人，不再只执着技术细节点，同时还需要更多关注技术带来的商业价值。在创业阶段也经历了快速搭建一个技术团队并最后散伙的心情。

创业失败后，如果继续回到技术专家的岗位，我可以很快找到新的工作。但我内心仍然希望继续担当技术组织者的角色，这时候可能选择大平台里面成长快的业务、以及对技术带头人有强烈需求的团队更为合适，这也许意味自己需要更长的时间去寻找及物色。

2013 年，我加入奇虎 360 PC 网游团队，负责技术架构工作，由一个更专注基础技术架构专家逐渐接触更多商业化项目，更多思考技术的商业价值。也逐渐在引进技术人才与管理技术资源方面变得娴熟。

到 2015 年，外界的条件对我感触很大，“大众创业，万众创新”，O2O 与共享经济等方向带来的互联网创业热潮，瞬间感觉技术人员的春天来了，鸭子也可卖上好价钱。由于很多团队都出现“只差一个技术负责人”的场面，开始发现自己经常能跟这个行业里面大佬对上话，里面不乏有一堆跑车的上市公司 VP，或一出生坐拥数亿资产的创业新秀。当时想如果跟这些大佬们接触，即使暂时不愿加入，也可以帮他们出谋划策或物色技术高手，至少可以在一定层面发挥价值。

这是个最坏的时代，也是个最好时代。最坏时代是到了 2015 下半年，转眼资本寒冬已来到，很多在创业的小伙伴感受到阵阵凉意。

最好的时代是说互联网依旧是国内经济发展巨大的引擎，互联网的发展离不开技术，因此技术人有比过去多得多的机会。尤其是那些有实力与口碑的技术人，通常会被各路行业大佬直接抢夺。

为了让职业生涯更好，除了技术硬实力外，还需要有高度的软实力。俗话说两手抓两手都要硬。 提升自身价值与商业价值的匹配度，让自己做得事情满足老板同时贴切商业价值，也就是这个时代技术人员的风口，这是我对当前这个互联网时代技术人成长的思考。

这也是为什么我在 2015 年中选择加入了熊猫 TV，并承担了公司最重要的基础架构的职责。

开源是让很多资源 free 的一个手段，但这个 free 不单指免费，更多是指提供廉价、自由、方便、开放、平等的资源，可以供平民来使用，这些资源不再那么昂贵，尤其技术资源。让屌丝创业公司，也能够站在相同的起跑线与巨头比武。技术基础建设者七牛云、青云等是我敬佩的公司，我所在的公司也大量使用了这些业界提供的基础服务。

感谢这个时代通过技术人的努力，让我们闻到 free 的味道，并让其开始生根发芽，好戏开始了，让我们见证万物复苏的开端，由技术驱动互联网创新的大戏才刚开始。■

高可用架构部分分享讲师名单（按姓名首字母排序）

- 陈飞，新浪微博技术经理
- 常雷博士，Pivotal 中国研发中心研发总监，HAWQ 并行 Hadoop SQL 引擎创始人，Pivotal HAWQ 团队负责人
- 陈宗志，奇虎 360 基础架构组高级存储研发工程师
- 杜传赢，Google 研发工程师
- 董西成，Hulu 网高级研发工程师，dongxicheng.org 博主
- 付海军，时趣互动技术总监
- 冯磊，新浪微博技术保障架构师
- 高磊，雪球运维架构师
- 郭斯杰，Twitter 高级工程师
- 郭伟，腾讯安全架构师
- 高永超，宜信大数据创新中心云平台运维专家
- 黄东旭，Ping CAP CTO，开源项目 Codis co-author
- 霍泰稳，InfoQ、极客邦科技创始人兼 CEO
- 蒋海滔，阿里巴巴国际事业部 高级技术专家
- 金自翔，百度资深研发工程师
- 吕毅，前百度资深研发工程师
- 马利超，小米科技的系统研发与大数据工程师

- 马涛，前迅雷网络 CDN 系统研发工程师，前 EMC/Pivotal Hawq 研发工程师
- 彭哲夫，芒果 TV 平台部核心技术团队负责人
- 秦迪，新浪微博研发中心技术专家
- 沈剑，58 到家技术总监 / 技术委员会负责人
- 孙其瑞，得图技术总监
- 孙宇聪，Coding.net CTO，前 Google SRE
- 孙子荀，腾讯手机 QQ 公众号后台技术负责人
- 谭政，Hulu 网大数据基础平台研发工程师
- 唐福林，雪球首席架构师
- 田琪，京东云数据库技术负责人
- 王富平，1 号店搜索与精准化部门架构师
- 王劲，酷狗音乐大数据架构师
- 王晶昱，花名沈询，阿里资深技术专家
- 王康，奇虎 360 基础架构组资深工程师
- 王新春，大众点评网数据平台资深工程师
- 王晓伟，麦图科技
- 王渊命，Grouk 联合创始人及 CTO
- 王卫华，百姓网资深开发工程师及架构师
- 温铭，奇虎 360 企业安全服务端架构师，OpenResty 社区咨询委员会成员

- 萧少聪，阿里云 RDS for PostgreSQL/PPAS 云数据库产品经理
- 许志雄，腾讯云产品运营负责人
- 颜国平，腾讯云天御系统研发负责人
- 闫国旗，京东资深架构师，京东架构技术委员会成员
- 杨保华，IBM 研究院高级研究员
- 杨尚刚，美图公司数据库高级 DBA
- 尤勇，大众点评网资深工程师，开源监控系统 CAT 开发者
- 张开涛，京东高级工程师
- 赵磊，Uber 高级工程师
- 张亮，当当网架构师、当当技术委员会成员、消息中间件组负责人
- 张虔熙，Hulu 网 HBase contributor
- 赵星宇，新浪微博 Android 高级研发工程师
- 周洋，奇虎 360 手机助手技术经理及架构师

扫描二维码关注微信公众号
或搜索 [ArchNotes] 高可用架构

出品人　　杨卫华

编辑/整理　　李　盼　　刘　伟　　四　正

设计　　大　胖