Think in Go

许式伟

2013-12-27

Go的优势

- 入门门槛低
  - 从任何语言转Go都不困难
- 心智负担低
  - 不容易出错，坑少
- 简洁
  - 语言的特性最少化，但有非常强大的表达能力

最适合Go的领域

- Go是通用语言，适用于绝大部分应用场景
- 但从库支持的角度，Go最合适的领域是
  - WS（Web Service）

WS是什么，谁需要?

- WS：Web Service，也叫 WS API

- 是不是只有云服务公司才有WS API?

  - No！所有有在线服务的公司都需要

    - 所有互联网公司都需要

互联网公司技术框架

- 基础设施（对大部分公司而言通常不适合自研）
    - 计算力：主机 / VM / AppEngine
    - 存储
        - 文件型（非结构化）：七牛云存储/开源存储/自研
        - 日志型（半结构化）：Hadoop
        - 关系型（结构化）：MongoDB/MySQL, Redis, TC/LevelDB, etc
        - 其他：MQ、Memcached
- 支撑
    - 运维：部署、升级、监控
    - 运营：用户行为分析、数据挖掘
    - 研发：开发、缺陷管理、测试

互联网公司技术框架

- WS API 层
    - 多终端时代到来，彰显 WS API 的重要性
    - 真正的业务逻辑层
        - 关注对公司业务逻辑的抽象
        - 关注安全性
        - 关注性能
- 多终端支持
    - Web：PC/Mobile Web
    - 手持(手机/平板)：Android / iOS / Windows Phone
    - PC: Mac / Windows / Linux
    - 其他：电视(盒) / 游戏机 / etc

WS的通用问题域

- 路由（route）
- 协议（marshal/unmarshal）：form/json/xml/etc
- 授权（authorization）
- ~~会话（session）~~
- 问题跟踪定位（trace/debug/log）
- 性能调优（prof）
- 审计（audit）
- 测试（test）
- 数值测量（measure）
- …

Go 能够帮你做什么?

- 降低高性能/高并发服务器的实现门槛
  - 绝大部分的Go服务器的都是高效的
  - 但不是全部:
    - 领域知识的门槛只能降低，但不会真正消除
    - 你还是可能会写出低效的服务器

性能瓶颈

- 带宽
  - 网络带宽
  - 磁盘IO
- 远程调用：rpc
- 近程通信（线程/goroutine通信）：channel/mutex
- …

有关于锁

- 锁不是唯一的性能杀手，也不是最大的性能杀手
- 变量不可变，其实不能够避开锁
  - 繆误：Erlang不需要锁，是因为变量不可变
- **本质上：锁是无法避开的，Why?**
  - 因为 WS 服务器本身是共享资源，无论什么技术都不能改变这个事实
    - Erlang不需要锁的本质，是因为WS服务器消息处理的串行化
      - 个人认为Erlang进入了视角误区

有关于锁

- 锁的问题在哪?
  - 最大问题：不易控制
    - 锁lock后忘记unlock的结果是灾难性的，因为服务器相当于挂了
  - 次要问题：性能杀手
    - 锁会导致代码串行化执行
    - 但是**别误会：锁并不特别慢**
      - 比近程（线程/goroutine）其他通信原语要快很多
        - **比锁快的东西：无锁、原子操作（并不比锁快很多）**
      - 网上有人用channel实现mutex，这很不正确

Go对锁的态度

- 避无可避，无需逃避!

锁的Go最佳实践

- 善用defer和滥用defer
  - 善用defer可以大大降低使用锁的心智负担
  - 滥用defer可能会导致锁粒度过大
- 控制锁粒度
  - 不要在锁里面执行费时操作
    - 会阻塞服务器，导致其他请求不能及时被响应

锁的Go最佳实践

- 读写锁（sync.RWMutex）
  - 如果一个资源（不一定是一个变量，可能是一组变量），有大量的读操作，少量的写操作，非常适合用读写锁。
- 锁数组（[]sync.Mutex）
  - 如果一个资源有很强的分区特征，各个分区的资源是独立的，很适合用锁数组。
    - 比如：一个资源分多用户，且不同用户间没有关联

```
var mutexs [N]sync.Mutex

mutex := &mutexs[uid % N] // 根据用户id选择锁
mutex.Lock()
defer mutex.Unlock()
...
```

其他近程通信

- channel 真的很好用
  - 用途：同步、收发消息
  - 留意 channel 的 bufsize
- channel 不是唯一的同步原语
  - sync.Group
    - 适合让很多人分工做某件事，等他们一起做完
  - sync.Cond
    - 非常适合实现生产者消费者模型（但真比channel要原始很多）
      - 当然不单单只用于这个场景

你还需要WS框架

- 框架的目标：解决通用问题域
  - 把你的精力解脱出来，专注于业务逻辑
- 内置的 net/http 解决了大部分的基础问题
- 最好还有一个专业点的WS框架，比如：beego
  - 七牛也许也会开源我们的WS框架

框架评价标准

- 业务代码的可测试性
  - 框架不应该提升模块的可测试复杂度
- 非侵入性
  - 框架是可选而并非必须
- 需求分解的正交性
  - 组件的单一职责

七牛的WS服务器

```
type Service struct { … }

type loginArgs struct {
    User string `json:"user"`
    Pwd string `json:"pwd"`
    ReturnUrl string `json:"return"`
    RememberMe int `json:"remember"`
}

/*
POST /login?user=$User&pwd=$Pwd&remember=$RememberMe&return=$ReturnUrl
*/
func (p *Service) WsLogin(args *loginArgs, env *session.Env) {
        if Matched(args.User, args.Pwd) {
                env.Session.Set("uid", args.User)
                http.Redirect(env.W, env.Req, args.ReturnUrl, 301)
        } else {
                http.Redirect(env.W, env.Req, loginUrl, 301)
        }
}
```

注：这不是 **WS API** 的形态，是常规 **Web** 的形态

七牛的WS服务器

```
type fooArgs struct {
    A string `json:”a”`
    B int `json:”b”`
}

type fooRet struct {
    C string `json:”c”`
    Uid uint `json:”uid”`
}

/*
POST /foo?a=$A&b=$B
Authorization: <Token>

200 OK
Content-Type: application/json
{c: $C, d: $D}
*/
func (p *Service) WsFoo(args *fooArgs, env *account.Env) (ret fooRet, err error) {
    if args.A == “” {
        err = httputil.NewError(400, “invalid argument ‘a’”)
        return
    }
    return fooRet{args.A, env.Uid}, nil
}
```

也可以是jsonrpc

```
/*
POST /double
Content-Type: application/json
123

200 OK
Content-Type: application/json
246
*/
func (p *Service) RpcDouble(v int) (int, error) {
   return v * 2, nil
}
```

注：通常 jsonrpc 仅限于内部服务

服务器的可测试性

- 多数业务逻辑可在不监听端口下完成测试
  - 大大降低服务器测试的难度

总结

- Go 的主战场：WS API 层
- 七牛的 WS 实战经验 & 框架

Q & A

@七牛云存储

@许式伟