

服务端开发那些事

许式伟

2015-10-17

服务端开发领域知识

- 网络协议
 - HTTP 协议
 - TCP/IP 协议
- 操作系统原理
 - 线程间通讯：互斥、同步、消息
 - 进程间通讯
- 存储系统原理
 - 数据库/缓存
- 模块设计
 - 架构哲学
 - 模块的最佳实践
- 服务器设计
 - 服务器的测试方法
 - 服务器的可维护性

1. 网络协议

- 网络协议
 - HTTP 协议

HTTP 协议

- 看请求包
 - nc -l 9999
 - chrome 浏览器输入 http://localhost:9999/hello

GET /hello HTTP/1.1

Host: localhost:9999

Connection: keep-alive

Cache-Control: max-age=0

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

User-Agent: Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.17 (KHTML, like Gecko) Chrome/24.0.1312.52 Safari/537.17

Accept-Encoding: gzip,deflate,sdch

Accept-Language: zh-CN,zh;q=0.8

Accept-Charset: GBK,utf-8;q=0.7,*;q=0.3

HTTP 协议

- 看请求包
 - nc -l 9999
 - curl -d 'a=1&b=2' http://localhost:9999/hello

POST /hello HTTP/1.1

User-Agent: curl/7.19.7 (i486-pc-linux-gnu) libcurl/7.19.7 OpenSSL/0.9.8k zlib/
1.2.3.3 libidn/1.15

Host: localhost:9999

Accept: */*

Content-Length: 7

Content-Type: application/x-www-form-urlencoded

a=1&b=2

HTTP 协议

- 看返回包

- `curl -i -d 'user=foo&pwd=bar' http://localhost:9999/login`

HTTP/1.1 200 OK

Content-Length: 0

Content-Type: application/json

Set-Cookie:

 session=PnVW8KJsHU6jIVbl7fHE_SKoQmyAjJBvPkveI9zPQfEI2iOTCETkHOLywt3t
 T2Y; Path=/

Date: Thu, 10 Oct 2013 07:21:13 GMT

HTTP 协议

- 看返回包
 - curl -i -b '_session=PnVW8KJsHU6jIVbl7fHE_SKoQmyAjJBrvPkveI9zPQfEI2iOTCETkHOLywt3tT2Y' http://localhost:9999/hello

HTTP/1.1 200 OK

Content-Type: text/plain; charset=utf-8

Content-Length: 10

Date: Thu, 10 Oct 2013 07:28:04 GMT

Hello foo

2.操作系统原理

- 操作系统原理
 - 线程间通讯：互斥、同步、消息

线程间通讯：互斥

- 互斥 ==> 锁
 - 有共享变量，就需要锁！
 - Golang 服务器里，共享变量几乎是必然的
 - Why?
 - 因为服务器本身同时在响应很多请求。
 - 服务器本身就是共享资源（里面定义的变量被很多线程同时访问）。
 - Erlang 为什么没有锁？
 - 不是因为 Erlang 是 FP 语言，Erlang 没有“变量”。
 - » Why? 为什么我说即便变量不可变，仍然无法避免锁？
 - 正确的原因是：Erlang 强制让所有请求串行化处理，Erlang 的服务器并不真正并行！
 - 如果某个 Golang 的服务器框架让所有请求串行化处理，那么也一样是不需要锁的。
 - Golang 服务器 `runtime.GOMAXPROCS(1)` 将程序设为单线程后，是否可以不需要锁？
 - » 不，仍然需要锁！Why?（请理解下goroutine工作原理）
 - » 单线程 != 所有请求串行化处理

线程间通讯：互斥

- 锁的问题在哪？
 - 最大问题：不易控制
 - 锁 lock 但忘记 unlock 的结果是灾难性的，因为服务器相当于挂了（所有和该锁有关的代码都不能被执行）！
 - 次要问题：性能杀手
 - 锁会导致代码串行化执行
 - 但**别误会：锁并不特别慢**
 - 比线程间通讯其他原语（同步、收发消息）要快很多！
 - » **比锁快的东西：无锁、原子操作（比锁并不快太多）**
 - 网上有人用 golang 的 channel 来实现锁，这很不正确

锁的最佳编程实践

- 善用 defer 和滥用 defer
 - 善用 defer 可以大大降低用锁的心智负担
 - 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

锁的最佳编程实践

- 读写锁: `sync.RWMutex`
 - 如果一个共享资源(不一定是变量,可能是一组变量),绝大部分情况下是读操作,偶然有写操作,则非常适合用读写锁。
- 锁数组: `[]sync.Mutex`
 - 如果一个共享资源,有很强的分区特征,则非常适合用锁数组
 - 比如一个网盘服务,网盘不同用户之间的资源彼此完全不相干

```
var mutexs [N]sync.Mutex
```

```
mutex := &mutexs[uid % N] // 根据用户id选择锁
```

```
mutex.Lock()
```

```
defer mutex.Unlock()
```

```
...
```

线程间通讯：同步、消息

- go lang 的 channel 很好用
 - 可同步、也可收发消息
 - 留意 channel 的缓冲区大小
 - `make(chan Type, N)`
- channel 不是唯一的同步原语
 - sync.Group
 - 适合让很多人一起干做某件事情，然后等他们干完。
 - sync.Cond
 - 非常容易用它实现生产者/消费者模型（比如channel）。
 - 比 channel 原始得多（但应该了解）。

3. 存储系统原理

- 存储系统原理
 - 数据库/缓存

存储系统原理

- 存储是什么？
 - 存储是状态的维持者
 - 存储本来不是问题，有服务器以后才成为问题
 - Why? 因为服务器必须逻辑上不宕机的
 - 物理服务器可以宕，但是状态必须仍然被维持
 - 维持一个逻辑不宕机的状态相当麻烦，所以存储就产生了
 - 存储就是数据结构
 - 维持状态这个任务从具体业务中剥离出来的结果
 - 存储非常多样化，它通常表现为某种大家熟知的数据结构（数据模型）
 - 比如：字典、表、文件系统（树）、队列（MQ）等等
- 详细参考
 - <http://open.qiniudn.com/golang-and-cloud-storage.pdf>

存储系统原理

- 靠谱服务的构建方式
 - **Fail Fast**（速错，故障即停）思想
 - 速错理念的基础是靠谱的存储
 - 参考：[http://open.qiniudn.com/\[Joe-Armstrong\]\[CN\]Making-reliable-distributed-systems-in-the-presence-of-software-errors.pdf](http://open.qiniudn.com/[Joe-Armstrong][CN]Making-reliable-distributed-systems-in-the-presence-of-software-errors.pdf)

存储为什么难？

- 别人都可以 Fail Fast
- 但存储秉承的理念是
 - 怎么错都应该有正确的（或合理的）结果
 - 软件问题？
 - 网络错？
 - 磁盘错？
 - 服务器断电？
 - 内存故障？
 - IDC 断线？
 - ...
- 详细参考
 - <http://open.qiniudn.com/cloud-storage-tech.pdf>

现实中的存储系统

- 七牛云存储（必须精通）
 - 七牛如何基于七牛构建七牛
- MongoDB/MySQL
- Hadoop/HBase
- Redis/Memcache
- ...

4. 模块设计

- 程序设计
 - 架构哲学
 - 模块的最佳实践

架构哲学

- 唯有理解需求才有架构
 - 架构的目标是为了更好地满足需求
- 架构的关键不是设计框架，而是需求的正交分解
 - 大需求(应用程序)被切分为小需求(模块)，小需求继续分解(类/组件/函数)
 - 本质上 app/service/module/package/class/func 是同一类东西，只是粒度问题。后文我们统一叫“模块”。
 - 我的“反框架”观
 - 重要的是模块，因为模块的需求是稳定的。
 - 大部分框架代码都是模块的连接方式(耦合方式)。框架通常是异变的、不稳定的（框架需要演进）。
 - 不稳定的东西通常是最不重要的东西。
 - 所以框架从实现角度可以依赖，但从架构角度最不重要。

模块设计

- 模块最重要的是什么？
 - 使用界面 (interface, 接口)
 - 对 app 来说, interface 就是用户交互
 - 对 service 来说, interface 就是网络协议 (api)
 - 对 package 来说, 就是包的导出函数/类/...
 - 对 class 来说, 就是公开方法/成员
 - 对 func 来说, 就是函数原型
 - 模块的使用界面体现了模块的需求

模块的最佳实践

- 模块的需求：单一职责
- 模块的使用界面：体现需求
- 模块的可测试性：最少的环境、最低耦合

模块的最佳实践

- 模块的**使用界面应该符合需求**，而不是符合某种框架的需要
- 模块应该是可完成的
 - 即：需求是稳定的、可预期的。
 - 模块的目标应该单一，只做一件事情。
 - 反例很多
 - C++: Boost / MFC / QT / ...
 - C++ 社区不是唯一
- 模块应该是可测试的
 - 越容易测试，说明模块的耦合(对环境的依赖)越小，使用界面越合理。

5. 服务器设计

- 服务器设计
 - 服务器的测试方法
 - 服务器的可维护性

需求的正交分解

- **Web 服务器**有哪些需求？
 - 路由(route)
 - 协议(marshal/unmarshal): form/json/etc
 - 授权(authorization)
 - 会话(session)
 - 问题跟踪/定位(debug/log/xlog)
 - 审计(auditlog)
 - 性能调优(prof)
 - 测试(test)
 - 监控(mon)
 - ...

服务器的测试方法

- 基于 **mockhttp** 进行测试
 - 不用监听物理的端口，没有端口冲突等问题，心智负担略低，执行速度快
- 基于七牛 **httptest** 进行测试
 - 不用写 **client** 端 **sdk**，可直接写测试案例，简单快捷

服务器的可维护性

- 服务器开发与运维不可分割
 - 服务器设计需要为运维做好充分的准备
- 对于经常发生的情况，服务器设计要有免运维能力
 - 异常情况的包容性：没有单点
 - 扩容的友好性
 - ...

服务器的可维护性

- 性能
 - 发现慢请求、定位瓶颈
 - 异常情况预警
- 故障发现与处理
 - 发现服务故障
 - 通知运维，或者自我恢复
 - 快速定位故障发生源
- 用户问题排查
 - 如何及时定位用户反馈的问题根源

Q & A