

# Golang Server Design Pattern

傅理  
@Tap4Fun

# Classical IPC pattern

- mutex
- semaphore
- pipe
- socket
- ...

一点小的改动,会导致不可预期的结果  
问题难于排查.

# What is CSP?

CSP :=

Communication Sequential Processing

is a formal language for describing patterns of interaction in concurrent systems. (一种用于定义并行系统的交互模式的形式语言)

CSP was first described in a 1978 paper by C. A. R. Hoare, -- [cspbook.pdf](#)

# CSP in brief:

1. send/receive only
2. no shared variable

CSP := 独立任务+同步消息传递

CSP message-passing fundamentally involves a rendezvous between the processes involved in sending and receiving the message, i.e. **the sender cannot transmit a message until the receiver is ready to accept it**

CSP语义保证收发同时成功

# Example:

```
func main() {  
    ch :=make(chan int, 10)  
    ch <- 1  
  
    select {  
    case v := <-ch:  
        println(v)  
    }  
}
```

不符合CSP语义, 异步消息

```
func main() {  
    ch :=make(chan int)  
    ch <- 1  
  
    select {  
    case v := <-ch:  
        println(v)  
    }  
}
```

符合CSP语义, 同步消息

# CSP 模式的优点

1. It avoids many of the traditional problems of parallelism in programming—interference, mutual exclusion, interrupts, multithreading, semaphores, etc. (避免了传统的锁等问题)
2. It provides a secure mathematical foundation for avoidance of errors and for achievement of **provable correctness** in the design. (有坚实的数学基础, 复杂系统设计上的正确性可被证明)

**Chan -- the CSP of golang**



创建一个Chan非常简单:

```
ch := make(chan interface{})
```

Chan 收发也很容易:

```
ch <- data
```

```
data := <- ch
```

并且,Chan 是 first-class value.....

Question: 收/发chan的时候golang做了什么?

**Answer:** 本质是带锁的FIFO操作

LOCK  
ENQUEUE  
UNLOCK  
&&&  
LOCK  
DEQUEUE  
UNLOCK

LOCK is implemented with futex()

# Question: futex 是个神马东西?

Answer:

1. Fast Userspace Mutex
2.  $\text{futex} := \text{CAS} + \text{mutex}()$
3. 一种优化的mutex实现, 通过延迟进入kernel  $\text{mutex}()$ , 本质上是atomic ops的应用.

# C++ perspective of Golang

从C++的角度来看golang

# Namespace+Class = path

一个目录路径就是一个namespace/class

同一目录中的所有变量,函数,互相可见.

小写字母开头的为private函数/变量  
func test()

大写字母开头的为public函数/变量  
func Test()

# Inheritance = Embedding

```
type A struct {  
    a int  
}
```

```
type B struct {  
    b int  
}
```

```
type AB struct {  
    A  
    B  
}
```

# Class method ~= receivers

```
type ByteSlice []byte
```

```
func (slice ByteSlice) Append(data []byte) []  
byte {  
    // Body exactly the same as above  
}
```

没有完整意义上的对象方法，没有隐含的this.

# Constructor $\sim$ `init()`

```
var names = map[string]int
```

```
func init() {  
    names = make(map[string]int)  
}
```

一个目录/一个文件中可以有若干个**init**函数  
**init**函数是全局的, 程序启动一次性的执行.  
类似于 `pthread_once()`



# 类型转换

`void * ~= interface{}`

`interface{}` 类似于C中的void指针.

`value.(typeName)`

类似于C++中

`dynamic_cast<typeName*>(ptr)`

PS: `interface{}`通常用于 reflect

**exceptions ~ = panic/recover**

panic()触发异常, recover()捕获异常  
等于  
try() -> catch()

# Rules of organizing a go project

< Go工程的组织方式 >

# Rules:

Rule #1: 一个目录下所有.go文件共同构成一个功能模块, 为一个目的服务.

Example:

/src/agent/

agent.go    buffer.go    proxy.go

session\_work.go

# Rules:

Rule #2: 用层次化(目录)的方式组织工程.

Example:

db/

    user\_tbl/

misc/packet

misc/alg/

    /alg/rbtree/

    /alg/dijkstra/

# Rules:

Rule #3:尽可能多的使用goroutine处理并行  
goroutine是很廉价的。

PS: goroutine定义: runtime.h:

```
struct G{  
...  
byte*stacko;  
...  
}
```

# Rules

## Rule #4:

用模块 + 接口的方式去构建系统

而不是OOP的方式

# Design of gonet

gonet的设计



# gonet网络模型

一个goroutine对应一个 connection, 类似于线程模型.

ps : 底层golang用 epoll 实现

# Example:

```
listener, err := net.ListenTCP("tcp", tcpAddr)
for {
    conn, err := listener.Accept()

    if err != nil {
        continue
    }
    go handleClient(conn)
}
```

That's all for a C10K server....

# 主消息循环

```
for {  
    select {  
        case msg, ok := <-in:           // 来自网络  
        case msg, ok := <-sess.MQ:      // 内部IPC Send()  
        case msg, ok := <-sess.CALL:    // 内部IPC Call()  
        case msg, ok := <-sess.OUT:     // 服务器发起  
        case _ = <-timer_ch_session:    // 定时器  
    }  
}
```

一个Session中包含 同步,异步,服务器发起三类MQ

# IPC/服务器端玩家通信.

一个全局的用户中心 src/hub/names

func Register(sess \*Session, id int32)

func Unregister(id int32)

func Query(id int32) \*Session

建立一个 ID -> Session 对应关系

# Send()的实现

```
peer := names.Query(id)
```

```
req := &RequestType{Code: tos}  
req.Params = params
```

```
peer.MQ <- req
```

# Call()的实现

```
peer := names.Query(id)
```

```
req := &RequestType{Code: tos}  
req.CH = make(chan interface{})  
req.Params = params
```

```
peer.CALL <- req  
ret, ok := <-req.CH
```

临时创建一个Chan,将这个Chan发送给peer,并在这个Chan上等结果.  
CH和CALL都为CSP

# 全局的信息访问

src/hub/online // 在线用户注册中心

src/hub/ranklist // 排名中心

设计一个接口来屏蔽锁操作, (sync包)  
模块内部对锁正确性负责.

大部分操作是读多写少,因此:

尽量用读写锁~~~ sync.RWMutex

尽量用原子操作~~~ atomic.AddXXX

# Testing



# Unit Testing

package naming

```
import "testing"
```

```
func TestXXXX (t *testing.T) {
```

```
    if .....
```

```
        t.Error
```

```
}
```

```
#go test -v naming
```

# Binary Protocol Testing

```
echo "000D 0001 0001 41 000000001  
01 0001 42" | xxd -r -ps  
| nc 127.0.0.1 8888 -q 10  
| hexdump -C
```

用xxd把hexstring转换为binary  
再用hexdump把返回结果显示

# 数据库同步

# 同步原则

Rule #1. Read尽可能在内存中进行, 尽可能延迟访问db

Rule #2. Write必须立即Sync到DB

Rule #3. 在内存中完成逻辑/一致性/完整性保证.

( 周期性sync到db,无法保证一致性和完整性 )

# gosched() internal

调度细节

# 多任务的模式

1. Cooperative multitasking/time-sharing  
协作式
1.  
2. Preemptive multitasking/time-sharing  
抢占式

golang is the former

# gosched()发生时期

1. channel send/recv/select/close (收、发、选、关)
2. map access/assign/iterate (读、写、遍历)
3. malloc (内部内存分配)
4. garbage collection
5. goroutine sleep
6. syscalls (所有系统调用)

# References:

<http://www.usingcsp.com/cspbook.pdf>

<http://www.akkadia.org/drepper/futex.pdf>

<http://bullshitlie.blogspot.jp/2013/04/golang.html>

源码:

<https://github.com/xtaci/gonet/tree/develop/src/>