Golang中常见的坑与编码模式

刘奇
新浪微博 @goroutine

与go一起的那些年

- 2009 相识(只是因为在人群中多看了go一眼，再也没能忘掉go的容颜)
- 2010-2011 相知(红尘之中知己难求，沙尘中呢？)
- 2012 相爱(带刺的玫瑰)

惊鸿一瞥

```go
//Serving http://localhost:8080/
package main

import (
    "fmt"
    "http"
)

func handler(c *http.Conn, r *http.Request) {
    fmt.Fprintf(c, "Hello, %s.", r.URL.Path[1:])
}
func main() {
    http.ListenAndServe(":8080",
        http.HandlerFunc(handler))
}
```

为何偏偏爱上你

- 良好的并发支持
- 静态链接
- 简洁 直观
- 语言级的并发与自动化垃圾回收支持
- 卓越的跨平台支持

Go 语言的目标(理想是美好的)

- 同时具备静态语言的运行效率、动态语言的开发效率
- 类型安全、内存安全
- 优秀的并发与通信能力
- 高效、无延迟的自动化垃圾收集
- 高速编译与静态链接
- 丰富、高质量的包

现实是凑合的

- Go是一门很二，很二的语言(是表扬，大家hold住鸡蛋和西红柿)
- 成熟度不如erlang
- 速度不如c，目前和java还有一点差距
- 库比不上java，python, c, cpp, nodejs
- 简洁不如ruby, python

未来是美好的

- Golang正在快速进步
- 库越来越丰富
- 性能越来越高

Go的国学思维(中庸)

- 几乎每个方面都是第二阵营，所以玩铁人三项才能拿冠军(又一个小米手机？)

- 所以成了集大成者

那些年，那些坑

- 总觉着踩到坑里了才算是真爱过☺
- 有人说踩到的坑越多，爱得越深
- 也有人说情越深，分得越快

List的遍历删除

- 错误写法(简洁，漂亮，但是。。。):
- for e := l.Front(); e != nil; e = e.Next() {
- 	l.Remove(e)
- }

- 正确写法
- var next *Element
- for e := l.Front(); e != nil; e = next {
- 	next = e.Next()
- 	l.Remove(e)
- }

查找原因

- // 看看具体实现代码(container/list.go)
- func (l *List) remove(e *Element) *Element {
- 	e.prev.next = e.next
- 	e.next.prev = e.prev
- 	e.next = nil // avoid memory leaks
- 	e.prev = nil // avoid memory leaks
- 	e.list = nil
- 	l.len--
- 	return e
- }

Time Formatting / Parsing

- t := time.Now()
- fmt.Println(t.Format("2006-01-02T15:04:05Z07:00"))

- 亲，记住了，只能用2006 01 02 15 04，别问我为什么，一个如此讲究用户体验的好语言出现这个约束确实有点难以理解

查找原因

- 这里写死了(time/format.go)
- const (
- 	ANSIC = "Mon Jan _2 15:04:05 2006"
- 	UnixDate = "Mon Jan _2 15:04:05 MST 2006"
- 	RubyDate = "Mon Jan 02 15:04:05 -0700 2006"
- 	RFC822 = "02 Jan 06 15:04 MST"
- 	RFC3339 = "2006-01-02T15:04:05Z07:00"
- 	……
-)
- 详见nextStdChunk函数

range

- values := []string{"a", "b", "c"}
- for _, v := range values {
- 	go func() {
- 		fmt.Println(v)
- 	}()
- }
- 输出三个c，而不是顺序的a, b, c

查找原因

- values := []string{"a", "b", "c"}
- for _, v := range values { //(1)
- 	go func() { //(2)
- 		fmt.Println(v) //隐式用v的地址传递
- 	}()
- }
- (1)复用了临时变量，只有一个临时变量的空间
- (2) 只是goroutine放到调度队列，不是立刻运行，还要排队先，等排到自己的时候，黄花菜已经凉了

解决方案

- for _, v := range values {
- go func(u string) {
- fmt.Println(u)

  }(v) //明确值复制，作为栈变量
- }

类似的

- list := make(map[int]*Link)
  for _, lnk := range linktree {
  list[lnk.Code] = &lnk
  }

解决方案

- list := make(map[int]*Link)
  for _, lnk := range linktree {
  var lnk = linktree
  list[lnk.Code] = &lnk
  }

Build once, run everywhere?

- Build on ubuntu 12.04 (golang tip version)
- Run on centos 5.6 may panic !
- 未测试Golang 1.1 beta，目前还不稳定，建议暂时不在生产环境使用

原因

- Syscall not match?

解决方案

- Update centos kernel

- Or build on centos 5.6

Channel的唤醒时序

- 当多个channel都处于就绪状态时，激活的channel是随机的
- **A Tour of Go:** A select blocks until one of its cases can run, then it executes that case. It chooses one at random if multiple are ready
- 切勿想当然的认为先来后到

queue

- Golang没有直接的queue
- 可以预期queue长度的情况下用channel
- 无法预期长度是用list当作队列用
- 也可以根据实际情况，将list作为channel的二级队列(sometimes you may need it)

解决方案

- 需要严格顺序处理的都放入一个channel

- 设计系统时需要仔细考虑时序的影响，特别是并发环境下

让人又爱又恨的GC

- 爱她的轻柔和美丽：代码看起来干净，整洁

- 恨她关键时刻停下来了：pause on marking and sweeping，比如网游和人pk的时候，或者实时系统，试想如果汽车刹车的时候刚好触发gc，后果很严重

原因

- 原始的标记清扫算法，一次完成整个标记清扫过程，没有分代(分生存期)。想想每天洗一件衣服和一周洗一次衣服的差别

- 程序产生大量的临时垃圾，引用关系复杂

解决方案

- Golang 1.1：已经支持并行gc，有较大改善
- 使用c模块来管理内存，c.malloc和c.free
- Object pool, buffer pool

Golang编码模式

惊艳的defer

- func FileOp() {
- 	f := os.Open(file)
- 	defer f.Close()
- 	// write data ...
- }
- 常用来做资源清理、关闭文件、解锁、记录执行时间等

Defer:来自标准库io.pipe的例子

- func (p *pipe) read(b []byte) (n int, err error) {
- 	// One reader at a time.
- 	p.rl.Lock()
- 	defer p.rl.Unlock()
-
- 	p.l.Lock()
- 	defer p.l.Unlock()
- 	……
- }

Defer:来自标准库的例子sql.go 连接池资源管理

- ci, err := db.conn()
- if err != nil {
- 	return nil, err
- }
- defer func() { //简洁，优雅
- 	db.putConn(ci, err)
- }()
- ……

Defer:一句话给函数计时

```
func  f() {
        defer timeoutCheck(“xx slow”, time.Now())
}

func timeoutCheck(tag string, start time.Time) {
        dis := time.Since(start).Seconds()
        if dis > 1 {
                log.Println(tag, dis, "s")
        }
}
```

Function design

- trying to return error if you can
- func (ns NullString) Value() (driver.Value, **error**) {
- 	if !ns.Valid {
- 		return nil, nil
- 	}
- 	return ns.String, nil
- }

让import更美观

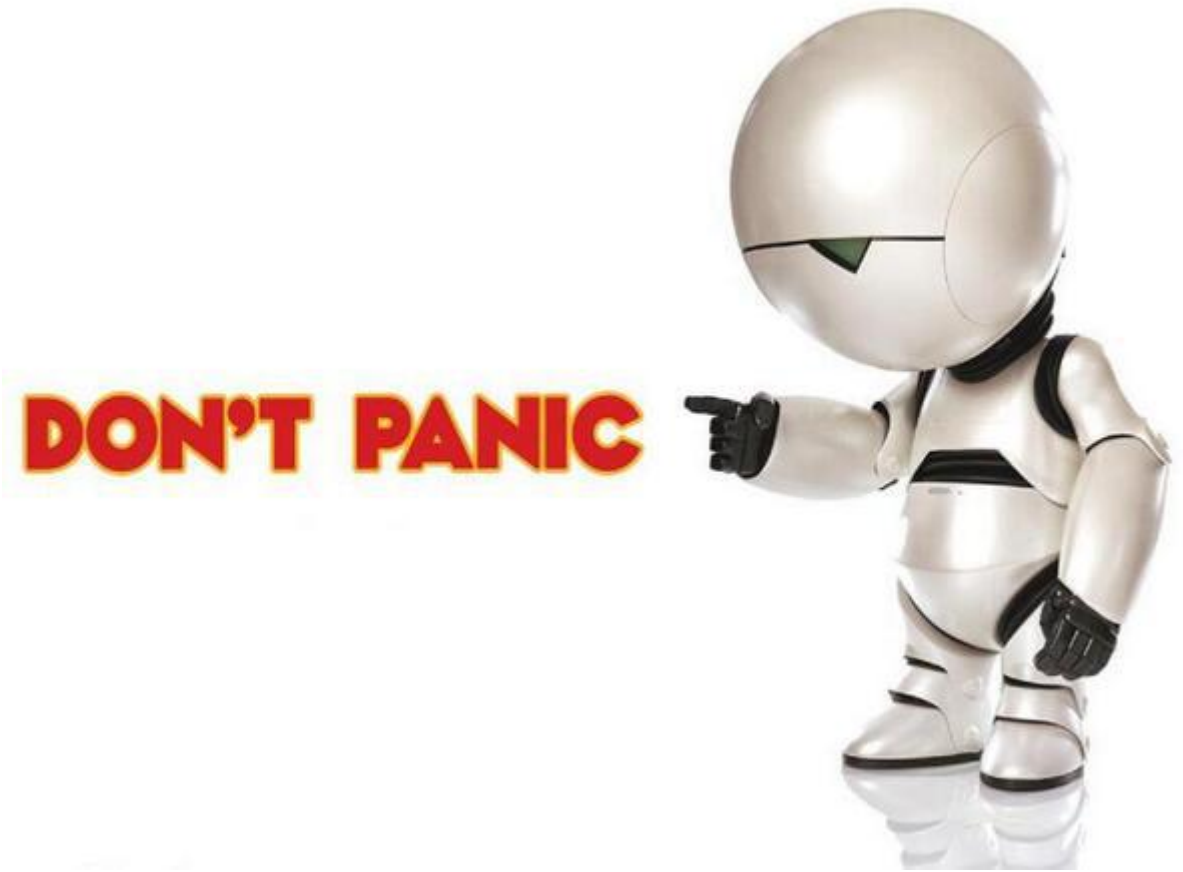
- import (
- 	"encoding/json"
- 	"fmt"
- 	"net"
-)
- 而不是
- import "encoding/json"
- import "fmt"
- import "net"

避免大对象的拷贝

- 如果map的value较大，通常应该使用指针来存储，以避免性能问题，类似的还有channel，slice等

- 避免[]byte和string的反复来回转换

About panic



About panic

- Panic if and only if you can’t handle it

Work pool

- func worker(jobs <-chan int, results chan<- int) {
- 	for j := range jobs {
- 	 results <- xx
- 	}
- }

- for w := 1; w <= 100; w++ {
- 	go worker(w, jobs, results)
- }

Object pool & buffer pool

```
// TODO(bradfitz): use a sync.Cache when available
var textprotoReaderCache = make(chan *textproto.Reader, 4)

func newTextprotoReader(br *bufio.Reader) *textproto.Reader {
        select {
        case r := <-textprotoReaderCache:
                r.R = br
                return r
        default:
                return textproto.NewReader(br)
        }
}

func putTextprotoReader(r *textproto.Reader) {
        r.R = nil
        select {
        case textprotoReaderCache <- r:
        default:
        }
}
```

一个简单的Connection pool

```
func (p *ConnectionPool) InitPool(size int, f FactoryMethod) {
    p.conn = make(chan interface{}, size)
    for i := 0; i < size; i++ {
        p.conn <-  f()
  }
  p.size = size
}

func (p *ConnectionPool) Get() interface{} {
  return <-p.conn
}

func (p *ConnectionPool) Put(conn interface{}) {
  p.conn <- conn
}
```

Connection pool

- 一个更加完善的例子 https://code.google.com/p/vitess/source/browse/go/pools/roundrobin.go

- 来自youtube的开源项目vtocc

goroutine

- Goroutine is cheap，not free(每个goroutine约占用6k的内存，实际代码很容易达到15-30K)
- 设计系统时要能预见并控制goroutine的总数，必要时可以发放ticket来控制资源
- 避免滥用goroutine，不合理的并行会带来更多的bug，也让问题难以复现和调试

及时，批处理模式

```
arr := make([]int, 8192)
for {
        select {
        case x:= <-ch:          //wait for first job
                count := 1
                arr[0] = x
        L:
                for ; count < MAX; count++ {    //check if we can handle more
                        select {
                        case e := <-ch:
                                arr[count] = e
                        default:
                                break L
                        }
                }
                ……
        }
}
```

Go fmt your code

通过http接口监控程序

- import _ "net/http/pprof“
- go func() {
- 　　http.ListenAndServe("localhost:　6060", nil)
- }()
- 随时观察程序状态(goroutine, heap, thread)
- 随时可以profile程序(go tool pprof http://localhost:6060/debug/pprof)

来个素颜照

/debug/pprof/

profiles:

0 block

1142 goroutine

157 heap

14 threadcreate

full goroutine stack dump

通过expvar导出关键变量

- import _ "expvar“

- 导出变量为json格式，便于分析和绘图，有利于编写自动化监控程序

通过http查看rpc的调用信息

- 用浏览器打开网址http://ip:port/debug/rpc 可以看到rpc各个函数的调用次数

牢记

- Don’t communicate by shared memory. Instead, share memory by communicating. —— Rob Pike

- Golang不是万能的，不排斥其它语言和其它工具，世界需要丰富多彩

Q & A

THANKS