我的go实战经验

李 霞
https://github.com/zimulala
@紫沐夏_Stubborn
gopher

摘要

- 常见的编程问题
- 测试
- 性能优化问题
- Go1.4新特性

常见的编程问题

- - 语言简洁

```
func CopyFile (dst, src string) (w int64, err error) {
   srcFile, err := os.Open (src)
   defer srcFile.Close ()
   if err != nil {
      return
   }
     //using srcFile to do sth

    return
}
```

- 函数可以返回多个值
- 推荐将error作为最后一个返回值
- Defer，常用来做资源清理、记录执行时间等

常见的编程问题

- - go range

```
values := []string{"a", "b", "c"}
for _, v := range values {
    go func() {
        fmt.Println(v)
    }()
}
```

常见的编程问题

--goroutine 通信

- 消息机制基于通信来共享。
- Go中goroutine之间是通过chan通讯的，chan的处理少不了用到select。
- 当多个receiver channel都处于就绪状态时，激活的channel是随机的。

```go
runtime.GOMAXPROCS(runtime.NumCPU())
ch := make(chan int, 1024)

go func(ch chan int) {
    for {
        val := <-ch
        fmt.Println("val:", val)
    }
}(ch)

tick := time.NewTicker(1 * time.Second)
for i := 0; i < 30; i++ {
    select {
    case ch <- i:
    case <-tick.C:
        fmt.Println("1 second")
    }
    time.Sleep(200 * time.Millisecond)
}

close(ch)
tick.Stop()
```

常见的编程问题

--goroutine 通信

- 往chan中放数据时，如果缓冲区已经满那么将block
- 以下方式可以试探往chan放数据
- func putSignal(ch chan struct{}, sign struct{}) (ok bool) {
- 	select {
- 	case ch <- sign:
- 		ok = true
- 	default://省略了会block
- 	}
-
- 	return
- }

常见的编程问题

--goroutine 通信

- cpu消耗问题

select滥用引发

```go
func GetSignal(ch chan bool) (ok bool) {
    select {
    case <-ch:
        ok = true
    default:
    }

    return
}

func (s *DataNode) handleReqs(msgH *MsgHandler) {
    for {
        select {
        case <-msgH.handleCh:
            //do sth
        case <-msgH.exitCh:
            //do sth
            return
        default:
            if GetSignal(s.shutdownCh) {
                //do sth
                return
            }
        }
    }
}
```

常见的编程问题

--goroutine 通信

- 某个chan block
- 某个处理 block

常见的编程问题

--goroutine 通信

- chan close，select可收到此信号，且如果有多个goroutine中有此chan，也都能收到。
- 对已经close的chan，不能重复close，只可以读取，但是不能写，报panic
- 在close（chan）并将chan里面的值全部读出来后，

  v，ok ： = <- ch

  此时v为zero值，ok为false。
- 对chan用range读取值会产生block，因为对于chan来说将其中的数据读完后会一直处于读等待，除非此时close。

测试

--重要性

- 写代码的人，可以用test来测试代码的准确性和高效性，现在比较多的说法是测试驱动开发。
- 看代码的人，可以通过test得知代码的使用；也可以通过test了解此代码的性能情况等。

测试

--单元测试

- 常用参数

-v 会记录所有单元测试运行的结果和其中的测试日志

-test.run pattern 只跑哪些单元测试用例

-test.timeout 超时限制

-race 竞态测试

go test -help 可以知道具体参数

测试

--单元测试

- 常用参数列表

```
-test.bench="": regular expression to select benchmarks to run
-test.benchmem=false: print memory allocations for benchmarks
-test.benchtime=1s: approximate run time for each benchmark
-test.blockprofile="": write a goroutine blocking profile to the named file after execution
-test.blockprofilerate=1: if >= 0, calls runtime.SetBlockProfileRate()
-test.coverprofile="": write a coverage profile to the named file after execution
-test.cpu="": comma-separated list of number of CPUs to use for each test
-test.cpuprofile="": write a cpu profile to the named file during execution
-test.memprofile="": write a memory profile to the named file after execution
-test.memprofilerate=0: if >=0, sets runtime.MemProfileRate
-test.outputdir="": directory in which to write profiles
-test.parallel=1: maximum test parallelism
-test.run="": regular expression to select tests and examples to run
-test.short=false: run smaller test suite to save time
-test.timeout=0: if positive, sets an aggregate time limit for all tests
-test.v=false: verbose: print additional output
```

测试

--单元测试

- 覆盖率

go test -coverprofile=coverage.out，可以看到测试的覆盖率。

```
PASS
coverage: 48.6% of statements
ok      dev/blockstore/datanode 40.510s
```

go tool cover -html=coverage.out，可以在网页上看到哪些代码测试到了，哪些没测试

测试

--基准测试

- 基准测试用例必须遵循如下格式，其中XXX可以是任意字母数字的组合，但是首字母不能是小写字母

func BenchmarkXXX(b *testing.B) { ... }

go test不会默认执行基准测试的函数，如果要执行基准测试需要带上参数-test.bench

- 例子:

```
PASS
BenchmarkBufPool-4       1000000                 1431 ns/op
ok      dev/blockstore/datanode 1.460s
```

测试

- - 异常测试

- cfg可能需要被修改多份

json串建议是小写加下划线（src_addr），结构体变量名可以加类似于`json:“src_addr”`

- kill()　//exec.Command函数模拟
- 异常测试的结尾，其实测试代码都需要收场

```
func teardown(c net.Conn, dir string) {
    os.Remove(dir)
    c.Close()
}
```

测试

- - 异常测试

```
const (
    masterPort01  = "9992"
    standbyPort01 = "9993"
)

var (
    master    *Server
    standby   *Server
    testDir01 string
)

func init() {
    testDir01 = path.Join(path, "testdir01")
    if err := os.Mkdir(testDir01, os.ModePerm); err != nil {
        log.Fatal("mkdir err:", err)
        return
    }

    master = initChunkServer(testDir01, masterPort01, true)
    standby = initChunkServer(testDir01, standbyPort01, false)
}
```

测试

初始化函数:

- 每个源文件都可以定义一个或多个初始化函数。
- 编译器不保证多个初始化函数执行次序。
- 初始化函数在单一线程被调用，仅执行一次。
- 初始化函数在包所有全局变量初始化后执行。
- 在所有初始化函数结束后才执行main.main。
- 无法调用初始化函数。

性能优化

--查看程序运行情况

1、通过http接口监控程序：

```
import _ “net/http/pprof”

go func() {
      http.ListenAndServe(“:6060", nil)
 }()
```

随时观察程序状态(goroutine, heap, thread)

http://localhost:6060/debug/pprof

/debug/pprof/

profiles:
0 block
8 goroutine
2 heap
6 threadcreate

full goroutine stack dump

性能优化

--查看序运行情况

2、随时可以查看程序运行时profile、heap等情况：

#go tool pprof http://localhost:6060/debug/pprof/profile　　//cup的占用情况

#top -cum

#web（需要安装graphviz）

```
(pprof) top
Total: 4138 samples
     732  17.7%  17.7%      795  19.2% syscall.Syscall
     612  14.8%  32.5%      612  14.8% hash/crc32.update
     437  10.6%  43.0%      437  10.6% runtime.futex
     263   6.4%  49.4%      265   6.4% syscall.Syscall6
     189   4.6%  54.0%      189   4.6% runtime.epollwait
     113   2.7%  56.7%      113   2.7% runtime.findfunc
      79   1.9%  58.6%       79   1.9% findrunnable
      78   1.9%  60.5%      147   3.6% runtime.mallocgc
      75   1.8%  62.3%       75   1.8% runtime.usleep
      54   1.3%  63.6%       64   1.5% fmt.(*fmt).integer
```

性能优化

--查看序运行情况

3、**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

http://localhost:8888/debug/vars

性能优化

```
type Cache struct {
    mu    sync.Mutex
    name  string
    saved []interface{}
    new   func() interface{}
}

func (c *Cache) Put(x interface{}) {
    c.mu.Lock()
    if len(c.saved) < cap(c.saved) {
        c.saved = append(c.saved, x)
    }
    c.mu.Unlock()
}

func (c *Cache) Get() interface{} {
    c.mu.Lock()
    n := len(c.saved)
    if n == 0 {
        c.mu.Unlock()
        return c.new()
    }
    x := c.saved[n-1]
    c.saved = c.saved[0 : n-1]
    c.mu.Unlock()

    return x
}

func NewCache(name string, cap int, f func() interface{}) *Cache {
    return &Cache{name: name, saved: make([]interface{}, 0, cap), new: f}
}
```

性能优化

--内存池

- 如果interface是一个[]byte类型，可能存在slice的边界问题，如果取出来的buf=buf[:len(buf)-3]，再把buf放进内存池，之后这个buf的使用就可能变成隐患。不过也可以通过

```
func (c *Cache) Put(buf []byte) {
    if len(buf) != c.bufSize {
        return
    }

    //releasing buf
}
```

性能优化

- - 连接池

- 具体问题具体分析
- 例如redis的连接池new的时候可以ping一下
- MySQL的第三方库自带了连接池

注意点：推荐Open操作后，判断err=db.Ping()中，err是否为nil来确定连接已经打开

//因为Open中的go db.connectionOpener()才是真的在建立连接

性能优化

--对象池

- 抽象化
- 比如，vitess里的对象池

性能优化

--小细节

- 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
- io操作用buffer系列来处理，性能更好些
- 避免[]byte和string的反复来回转换
- 不建议在被大量调用或者频繁调用的函数中使用defer，他会影响gc和性能。
- 性能要求非常严格的地方，可以手动管理资源
- 可以考虑调用c模块

Go1.4新特性

- 语言变化，for range slice｛｝
- runtime很多代码被替换成了用Go自身实现
- 重写使得垃圾回收器变得更加精确
- goroutine的初始栈大小从8k变到2k
- 源码布局变化：src/pkg/fmt，到src/fmt
- 官方说：性能大部分程序会略有提升，具体很难准确预测

Q & A

- **Thanks !**