C/C++Linux服务器开发

高级架构师课程

三年课程沉淀 | 五次精益升级 | 十年行业积累 | 百个实战项目 | 十万内容受众

讲师:darren/326873713

扫一扫 升职加薪

班主任:柚子/2690491738

讲师介绍--专业来自专注和实力

King老师

系统架构师，曾供职著名创业公司系统架构师，微软亚洲研究院、创维集团全球研发中心。国内第一代商业Paas平台开发者。著有多个软件专利，参与多个开源软件维护。在全球化，高可用的物联网云平台架构与智能硬件设计方面有丰富的研发与实战经验。

Darren老师

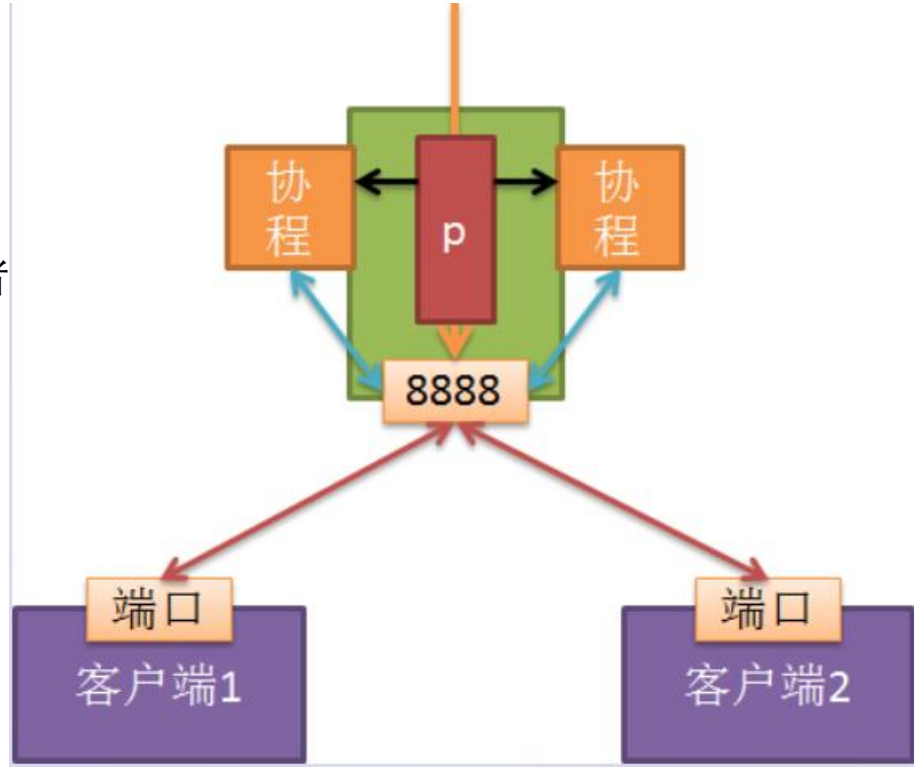
曾供职于国内知名半导体公司（珠海扬智/深圳联发科），曾在某互联网公司担任音视频通话项目经理。主要从事音视频驱动、多媒体中间件、流媒体服务器的开发，开发过即时通讯+音视频通话的大型项目，在音视频、C/C++/GO Linux服务器领域有丰富的实战经验。

Go语言网络编程和常用库使用

1. 网络编程
2. Redis库redigo
3. 临时对象池sync.Pool
4. 配置文件读取goconfig
5. 解析命令行flag
6. uuid生成方案

1 网络编程

目前主流服务器一般均采用的都是"Non-Block + I/O多路复用"（有的也结合了多线程、多进程）。不过I/O多路复用也给使用者带来了不小的复杂度，以至于后续出现了许多高性能的I/O多路复用框架， 比如libevent、libev、libuv等，以帮助开发者简化开发复杂性，降低心智负担。不过Go的设计者似乎认为I/O多路复用的这种通过回调机制割裂控制流 的方式依旧复杂，且有悖于"一般逻辑"设计，为此**Go语言将该"复杂性"隐藏在Runtime中**了：Go开发者无需关注socket是否是 non-block的，也无需亲自注册文件描述符的回调，**只需在每个连接对应的goroutine中以"block I/O"的方式对待socket处理即可**




```go
func main() {
    fmt.Println("服务器开始监听...")
    //协议、端口
    listen, err := net.Listen("tcp", "0.0.0.0:8888")
    if err != nil {
        fmt.Println("监听失败,err=", err)
        return
    }
    //延时关闭
    defer listen.Close()
    for {
        //循环等待客户端连接
        fmt.Println("等待客户端连接...")
        conn, err := listen.Accept()
        if err != nil {
            fmt.Println("Accept() err=", err)
        } else {
            fmt.Printf("Accept() suc con=%v,客户端Ip=%v\n", conn, conn.RemoteAddr().String())
        }
        //这里准备起个协程为客户端服务
        go process(conn)
    }
    //fmt.Printf("监听成功, suv=%v\n", listen)
}
```

1-1-client.go

1-1-server.go

1.0 TCP socket api

•Read(): 从连接上读取数据。
•Write(): 向连接上写入数据。
•Close(): 关闭连接。
•LocalAddr(): 返回本地网络地址。
•RemoteAddr(): 返回远程网络地址。
•SetDeadline(): 设置连接相关的读写最后期限。等价于同时调用SetReadDeadline()和SetWriteDeadline()。
•SetReadDeadline(): 设置将来的读调用和当前阻塞的读调用的超时最后期限。
•SetWriteDeadline(): 设置将来写调用以及当前阻塞的写调用的超时最后期限。

1.1 TCP连接的建立

服务端是一个标准的Listen + Accept的结构(可参考上面的代码)，而在客户端Go语言使用**net.Dial()**或**net.DialTimeout()**进行连接建立。

服务端

参考上一页

客户端

阻塞Dial:

```
conn, err := net.Dial("tcp", "www.baidu.com:80")
if err != nil {
    //handle error
}
//read or write on conn
```

超时机制的Dial:

```
conn, err := net.DialTimeout("tcp", "www.baidu.com:80", 2*time.Second)
if err != nil {
    //handle error
}
//read or write on conn
```

1.2 客户端连接异常情况分析

1、网络不可达或对方服务未启动

2、对方服务的listen backlog满

3、网络延迟较大，Dial阻塞并超时

1.2.1 客户端连接异常–网络不可达或对方服务未启动

如果传给Dial的Addr是可以立即判断出网络不可达，或者Addr中端口对应的服务没有启动，端口未被监听，Dial会几乎立即返回错误，比如：

```go
func main() {
    log.Println("begin dial...")
    conn, err := net.Dial("tcp", ":8888")
    if err != nil {
        log.Println("dial error:", err)
        return
    }
    defer conn.Close()
    log.Println("dial ok")
}
```

1-2-1-client.go

1.2.2 客户端连接异常-对方服务的listen backlog满

对方服务器很忙，瞬间有大量client端连接尝试向server建立，server端的listen backlog队列满，server accept不及时((即便不accept，那么在backlog数量范畴里面，connect都会是成功的，因为new conn已经加入到server side的listen queue中了，accept只是从queue中取出一个conn而已)，这将导致client端Dial阻塞。

```
connectex: No connection could be made because the target machine actively refused it.
```

1-2-2-client.go

1.2.3 客户端连接异常-网络延迟较大，Dial阻塞并超时

如果网络延迟较大，TCP握手过程将更加艰难坎坷（各种丢包），时间消耗的自然也会更长。Dial这时会阻塞，如果长时间依旧无法建立连接，则Dial也会返回“ getsockopt: operation timed out”错误
在连接建立阶段，多数情况下，Dial是可以满足需求的，即便阻塞一小会儿。但对于某些程序而言，需要有严格的连接时间限定，如果一定时间内没能成功建立连接，程序可能会需要执行一段“异常”处理逻辑，为此我们就需要DialTimeout了。

执行结果如下，需要模拟一个网络延迟大的环境

1-2-3-client.go

1.3 Socket读写

Dial成功后，方法返回一个net.Conn接口类型变量值，这个接口变量的动态类型为一个*TCPConn:

1-2-panic.go

1. 3. 1 conn. Read的行为特点

1 Socket中无数据

连接建立后，如果对方未发送数据到socket，接收方(Server)会阻塞在Read操作上，这和前面提到的“模型”原理是一致的。执行该Read操作的goroutine也会被挂起。runtime会监视该socket，直到其有数据才会重新调度该socket对应的Goroutine完成read。

2 Socket中有部分数据

如果socket中有部分数据，且长度小于一次Read操作所期望读出的数据长度，那么Read将会成功读出这部分数据并返回，而不是等待所有期望数据全部读取后再返回。

3 Socket中有足够数据

如果socket中有数据，且长度大于等于一次Read操作所期望读出的数据长度，那么Read将会成功读出这部分数据并返回。这个情景是最符合我们对Read的期待的了：Read将用Socket中的数据将我们传入的slice填满后返回：n = 10, err = nil

4 Socket关闭

有数据关闭是指在client关闭时，socket中还有server端未读取的数据。当client端close socket退出后，server依旧没有开始Read，10s后第一次Read成功读出了所有的数据，当第二次Read时，由于client端 socket关闭，Read返回EOF error
无数据关闭情形下的结果，那就是Read直接返回**EOF error**

5 读取操作超时

有些场合对Read的阻塞时间有严格限制，在这种情况下，Read的行为到底是什么样的呢？在返回超时错误时，是否也同时Read了一部分数据了呢？

不会出现“读出部分数据且返回超时错误”的情况

1.3.2 conn.Write的行为特点

1 成功写

前面例子着重于Read，client端在Write时并未判断Write的返回值。所谓“成功写”指的就是Write调用返回的n与预期要写入的数据长度相等，且error = nil。这是我们在调用Write时遇到的最常见的情形，这里不再举例了

2 写阻塞

TCP连接通信两端的OS都会为该连接保留数据缓冲，一端调用Write后，实际上数据是写入到OS的协议栈的数据缓冲的。TCP是全双工通信，因此每个方向都有独立的数据缓冲。当发送方将对方的接收缓冲区以及自身的发送缓冲区写满后，Write就会阻塞

3 写入部分数据

Write操作存在写入部分数据的情况。没有按照预期的写入所有数据。这时候循环写入便是

综上例子，虽然Go给我们提供了阻塞I/O的便利，但在调用Read和Write时依旧要综合需要方法返回的n和err的结果，以做出正确处理。net.conn实现了io.Reader和io.Writer接口，因此可以试用一些wrapper包进行socket读写，比如bufio包下面的Writer和Reader、io/ioutil下的函数等

1.4 Goroutine safe

基于goroutine的网络架构模型，存在在不同goroutine间共享conn的情况，那么conn的读写是否是goroutine safe的呢。

Write
Read 内部是goroutine安全的，内部都有Lock保护

1.5 Socket属性

SetKeepAlive
SetKeepAlivePeriod
SetLinger
SetNoDelay （默认no delay）
SetWriteBuffer
SetReadBuffer

要使用上面的Method的，需要type assertion

```
tcpConn, ok := conn.(*TCPConn)
if !ok { //error handle }
    tcpConn.SetNoDelay(true)
```

1.6 关闭连接

socket是全双工的，client和server端在己方已关闭的socket和对方关闭的socket上操作的结果有不同。

1-7 读写超时

SetDeadline(t time.Time) error 设置读写超时

SetReadDeadline(t time.Time) error 设置读超时

SetWriteDeadline(t time.Time) error 设置写超时

2 redis

参考文档: github.com/garyburd/redigo/redis
https://pkg.go.dev/github.com/garyburd/redigo/redis#pkg-index

使用第三方开源的redis库: github.com/gomodule/redigo/redis

import(
“go get github.com/gomodule/redigo/redis”
)

go get github.com/gomodule/redigo/redis

2.1 连接redis

```go
func main() {
    c, err := redis.Dial("tcp", "192.168.2.132:6379")
    if err != nil {
        fmt.Println("conn redis failed,", err)
        return
    }

    defer c.Close()
}
```

2-1-redis-conn.go

2.2 redis set操作

```go
_, err = c.Do("Set", "abc", 100)
if err != nil {
    fmt.Println(err)
    return
}

r, err := redis.Int(c.Do("Get", "abc"))
if err != nil {
    fmt.Println("get abc failed,", err)
    return
}

fmt.Println(r)
```

2.3 redis Hash操作

```go
_, err = c.Do("HSet", "books", "abc", 100)
if err != nil {
    fmt.Println(err)
    return
}

r, err := redis.Int(c.Do("HGet", "books", "abc"))
if err != nil {
    fmt.Println("get abc failed,", err)
    return
}
```

2-3-redis-hash.go

2.4 redis mset操作

```go
_, err = c.Do("MSet", "abc", 100, "efg", 300)
if err != nil {
    fmt.Println(err)
    return
}

r, err := redis.Ints(c.Do("MGet", "abc", "efg"))
if err != nil {
    fmt.Println("get abc failed,", err)
    return
}

for _, v := range r {
    fmt.Println(v)
}
```

2-4-redis-mset.go

2.5 redis expire操作

```go
_, err = c.Do("expire", "abc", 10)
if err != nil {
    fmt.Println(err)
    return
}
```

2-5-redis-expire.go

2.6 redis list操作

```go
_, err = c.Do("lpush", "book_list", "abc", "ceg", 300)
if err != nil {
    fmt.Println(err)
    return
}

r, err := redis.String(c.Do("lpop", "book_list"))
if err != nil {
    fmt.Println("get abc failed,", err)
    return
}
```

2-6-list.go

2-7 subpub操作

2-7-subpub.go

2-8 连接池

MaxIdle：最大的空闲连接数，表示即使没有redis连接时依然可以保持N个空闲的连接，而不被清除，随时处于待命状态。
MaxActive：最大的连接数，表示同时最多有N个连接。0表示不限制。
IdleTimeout：最大的空闲连接等待时间，超过此时间后，空闲连接将被关闭。如果设置成0，空闲连接将不会被关闭。应该设置一个比redis服务端超时时间更短的时间。
DialConnectTimeout：连接Redis超时时间。
DialReadTimeout：从Redis读取数据超时时间。
DialWriteTimeout：向Redis写入数据超时时间。
连接流程大概是这样的
1.尝试从空闲列表MaxIdle中，获得一个可用连接;如果成功直接返回，失败则尝试步骤2
2.如果当前的MaxIdle < 连接数 < MaxActive，则尝试创建一个新连接，失败则尝试步骤3
3. 如果连接数 > MaxActive就等待,直到满足步骤2的条件，重复步骤2

2-8 redis连接池的坑

遇到过的问题

目前为止，连接池的问题只遇到过一次问题，而且是在测试环境的，当时的配置是

```
DialConnectTimeout: time.Duration(200)*time.Millisecond
DialReadTimeout: time.Duration(200)*time.Millisecond
DialWriteTimeout: time.Duration(200)*time.Millisecond
```

配置的都是200毫秒。有一次使用hgetall的时候，就一直报错，大概类似下面的提示

```
read tcp 127.0.0.1:6379: i/o timeout
```

字面意思就是 read tcp 超时，可能某些写入大点数据的时候也会报，write tcp timeout。

后来将读写超时时间都改为1000毫秒，就再也没有出现过类似的报错。

想了解更多的Redis使用，可以看下官方的文档：

github.com/gomodule/redigo/redis

3 临时对象池

sync.Pool类型值作为存放临时值的容器。此类容器是自动伸缩的，高效的，同时也是并发安全的。

sync.Pool类型只有两个方法：

- Put，用于在当前的池中存放临时对象，它接受一个空接口类型的值
- Get，用于从当前的池中获取临时对象，它返回一个空接口类型的值

New字段

sync.Pool类型的New字段是一个创建临时对象的函数。它的类型是没有参数但是会返回一个空接口类型的函数。即：func() interface{}。
这个函数是Get方法最后的获取到临时对象的手段。函数的结果不会被存入当前的临时对象池中，**而是直接返回给Get方法的调用方。**
这里的New字段的实际值需要在初始化临时对象池的时候就给定。否则，在Get方法调用它的时候就会得到nil。

```
if x == nil && p.New != nil { // 未找到可用的元素，调用New生成
    x = p.New()
}
return x
```

3.1 Get

Pool 会为每个 P 维护一个本地池，P 的本地池分为 私有池 private和共享池 shared。私有池中的元素只能本地 P 使用，共享池中的元素可能会被其他 P 偷走，所以使用私有池 private 时不用加锁，而使用共享池 shared 时需加锁。

Get 会优先查找本地 private，再查找本地 shared，最后查找其他 P 的shared，如果以上全部没有可用元素，最后会调用 New 函数获取新元素。

3. 2 Put

Put 优先把元素放在 private 池中；如果 private 不为空，则放在 shared 池中。有趣的是，在入池之前，该元素有 1/4 可能被丢掉。

```go
func (p *Pool) Put(x interface{}) {
    if x == nil {
        return
    }
    if race.Enabled {
        if fastrand()%4 == 0 {
            // Randomly drop x on floor. 随机把元素丢掉，入池之前有1/4的概率被扔掉
            return
        }
        race.ReleaseMerge(poolRaceAddr(x))
        race.Disable()
    }
```

4 配置文件解析器goconfig的使用

ini配置文件读写

4-1-config-ini.go

5 命令行解析Go flag

cmd -flag　　// 只支持bool类型
cmd -flag=xxx
cmd -flag xxx　// 只支持非bool类型

1. 定义flag参数

参数有三个：第一个为 参数名称，第二个为 默认值，第三个是 使用说明

（1）通过 flag.String()，Bool()，Int() 等 flag.Xxx() 方法，该种方式返回一个相应的指针

```
var ip = flag.Int("flagname", 1234, "help message for flagname")
```

（2）通过 flag.XxxVar() 方法将 flag 绑定到一个变量，该种方式返回 值类型

```
var flagvar int
flag.IntVar(&flagvar, "flagname", 1234, "help message for flagname")
```

（3）通过 flag.Var() 绑定自定义类型，自定义类型需要实现 Value 接口 (Receiver 必须为指针)

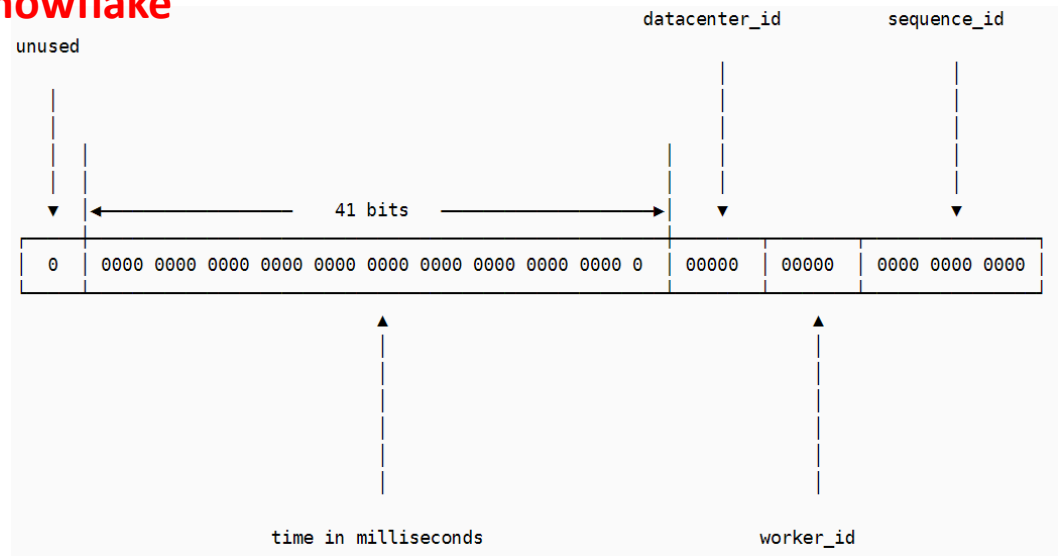
```
fmt.Println("flagvar has value ", flagvar)
```

5-1-cli-flag.go
5-2-self-flag.go

6 uuid

雪花算法 Snowflake & Sonyflake https://www.cnblogs.com/li-peng/p/12124249.html

snowflake



snowflake算法使用的一个64 bit的整型数据，被划分为四部分。

1.不含开头的第一个bit，因为是符号位；
2.41bit来表示收到请求时的时间戳，精确到1毫秒；
3.5bit表示数据中心的id， 5bit表示机器实例id
4.共计10bit的机器位，因此能部署在1024台机器节点上生成ID；
5.12bit循环自增序列号，增至最大后归0，1毫秒最大生成唯一ID的数量是4096个。

sonyflake

```
+--------------------------------------------------------------------------+
| 1 Bit Unused | 39 Bit Timestamp |  8 Bit Sequence ID  |  16 Bit Machine ID |
+--------------------------------------------------------------------------+
```

这里时间戳用39位精确到10ms，所以可以达到174年，比snowflake的长很久。

8bit 做为序列号，每10毫最大生成256个，1秒最多生成25600个，比原生的Snowflake少好多，如果感觉不够用，目前的解决方案是跑多个实例生成同一业务的ID来弥补。

16bit 做为机器号，默认的是当前机器的私有IP的最后两位。

6-1-1-uuid.go 常用uuid性能测试

sonyflake对于snowflake的改进是**用空间换时间**，时间戳位数减少，以从69年升至174年。但是1秒最多生成的ID从409.6w降至2.56w条。