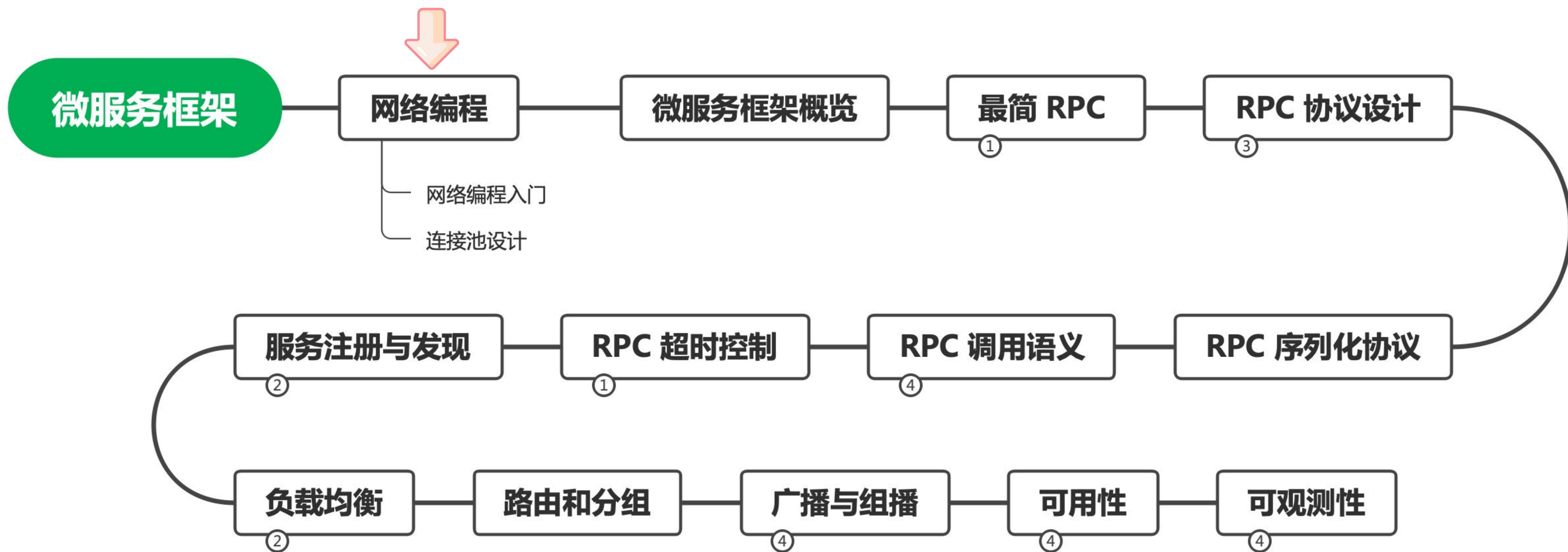


微服务框架——网络编程

大明

学习路线



网络编程：net 包

net 包是网络相关的核心包，里面包含了 http、rpc 等关键包。

在 net 里面，最重要的两个调用：

- **Listen**(network, addr string): 监听某个端口，等待客户端连接
- **Dial**(network, addr string): 拨号，其实也就是连上某个服务端

```
▼ net
  > http
  > internal
  > mail
  > netip
  > rpc
  > smtp
  > testdata
  > textproto
  > url
  🐼 addrselect.go
```

网络编程：通信基本流程

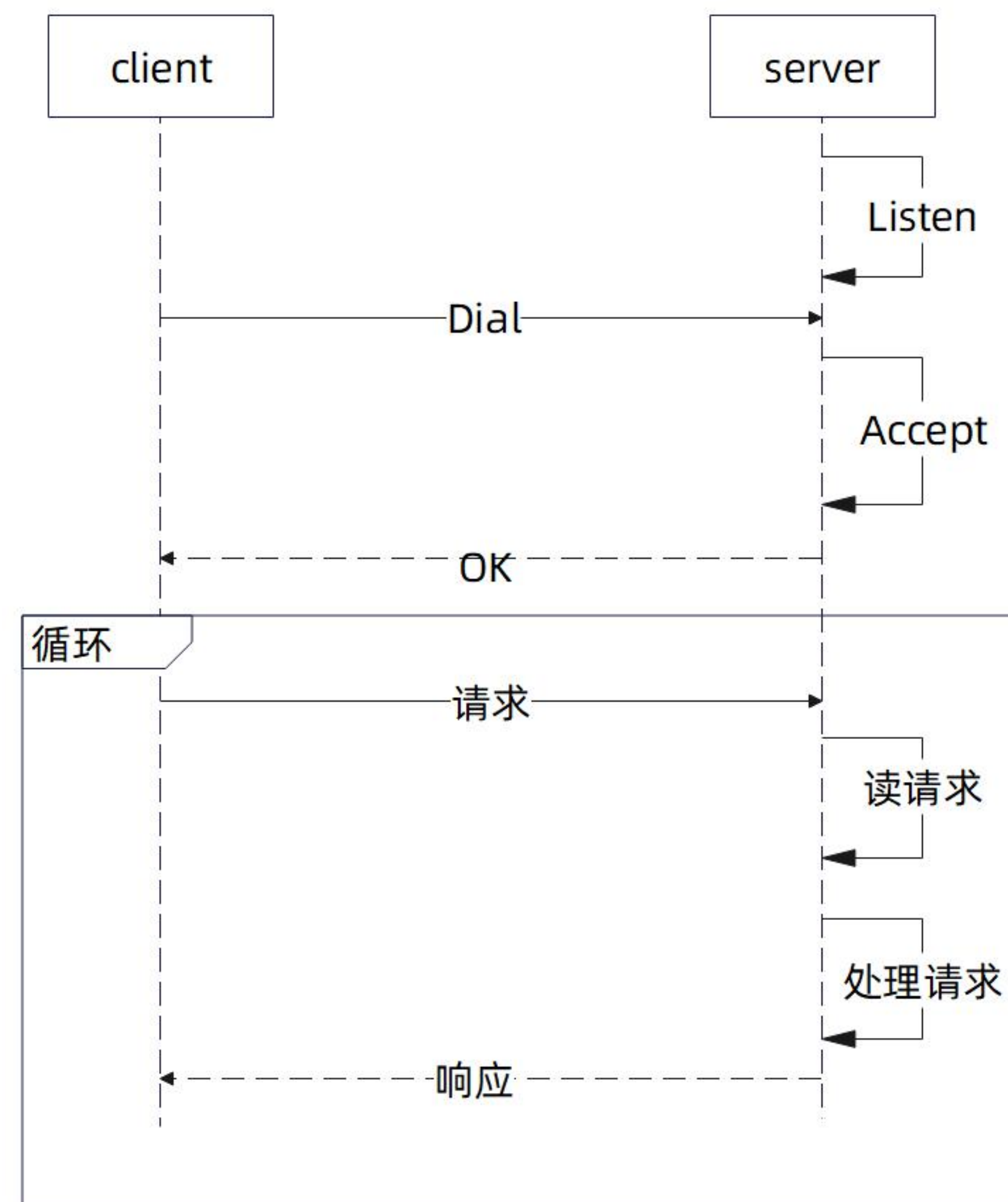
基本分成两个大阶段。

创建连接阶段：

- 服务端开始监听一个端口
- 客户端拨通服务端，两者协商创建连接 (TCP)

通信阶段：

- 客户端不断发送请求
- 服务端读取请求
- 服务端处理请求
- 服务端写回响应



网络编程：net.Listen

Listen 是监听一个端口，准备读取数据。它还有几个类似接口，可以直接使用：

- ListenTCP
- ListenUDP
- ListenIP
- ListenUnix

这几个方法都是返回 Listener 的具体类，如 TCPListener。**一般用 Listen 就可以**，除非你要依赖于具体的网络协议特性。

网络通信用 TCP 还是用 UDP 是一个影响巨大的事情，一般确认了就不会改。

```
func Serve(addr string) error {  
    listener, err := net.Listen(network: "tcp", addr)  
    if err != nil {  
        return err  
    }  
    for {  
        conn, err := listener.Accept()  
        if err != nil {  
            return err  
        }  
        go func() {  
            handleConn(conn)  
        }()  
    }  
}
```

代码模板就这样。**在handleConn里面读取数据--做点操作--写回响应。**

网络编程：处理连接

处理连接基本上就是在一个 for 循环内：

- **先读数据**：读数据要根据上层协议来决定怎么读。例如，简单的 RPC 协议一般是分成两段读，先读头部，根据头部得知 Body 有多长，再把剩下的数据读出来。
- **处理数据**
- **回写响应**：即便处理数据出错，也要返回一个错误给客户端，不然客户端不知道你处理出错了。

```
func handleConn(conn net.Conn) {
    for {
        // 读数据
        bs := make([]byte, 8)
        _, err := conn.Read(bs)
        if err == io.EOF || err == net.ErrClosed ||
            err == io.ErrUnexpectedEOF {
            // 一般关闭的错误比较懒得管
            // 也可以把关闭错误输出到日志
            _ = conn.Close()
            return
        }
        if err != nil {
            continue
        }
        res := handleMsg(bs)
        _, err = conn.Write(res)
        if err == io.EOF || err == net.ErrClosed ||
            err == io.ErrUnexpectedEOF {
            _ = conn.Close()
            return
        }
    }
}
```

网络编程： 错误处理

在读写的时候，都可能遇到错误，一般来说代表连接已经关掉的是这三个：
EOF、ErrUnexpectedEOF 和 ErrClosed。

但是，我建议只要是**出错了就直接关闭**，这样对客户端和服务端代码都简单。

```
func handleConnV1(conn net.Conn) {
    for {
        // 读数据
        bs := make([]byte, 8)
        _, err := conn.Read(bs)
        if err != nil {
            // 一般关闭的错误比较懒得管
            // 也可以把关闭错误输出到日志
            _ = conn.Close()
            return
        }
    }
}
```

```
func handleConn(conn net.Conn) {
    for {
        // 读数据
        bs := make([]byte, 8)
        _, err := conn.Read(bs)
        if err == io.EOF || err == net.ErrClosed ||
            err == io.ErrUnexpectedEOF {
            // 一般关闭的错误比较懒得管
            // 也可以把关闭错误输出到日志
            _ = conn.Close()
            return
        }
        if err != nil {
            continue
        }
        res := handleMsg(bs)
        _, err = conn.Write(res)
        if err == io.EOF || err == net.ErrClosed ||
            err == io.ErrUnexpectedEOF {
            _ = conn.Close()
            return
        }
    }
}
```

网络编程：net.Dial

net.Dial 是指创建一个连接，连上远端的服务器。

它也是有几个类似的方法：

- DialIP
- DialTCP
- DialUDP
- DialUnix
- DialTimeout

只有 DialTimeout 稍微特殊一点，它多了一个超时参数。

类似于 Listen，我也是建议大家直接使用

DialTimeout，因为设置超时可以避免一直阻塞。

```
func Connect(addr string) error {
    conn, err := net.DialTimeout(network: "tcp", addr, 3*time.Second)
    if err != nil {
        return err
    }
    defer func() {
        _ = conn.Close()
    }()
    for {
        _, err = conn.Write([]byte("hello"))
        if err != nil {
            return err
        }
        res := make([]byte, 128)
        _, err = conn.Read(res)
        if err != nil {
            return err
        }
    }
}
```

发送请求

接收响应

这个模板和服务端处理请求的模板也很像。

代码演示：创建简单的 TCP 服务器

在前面的代码里面，我们创建的接收数据的字节数组都是固定长度的，那么问题在于，在真实的环境下，长度应该是不确定的。

比如说发送字符串“Hello”和发送字符串“Hello, world”，这长度就不太一样了，怎么办？

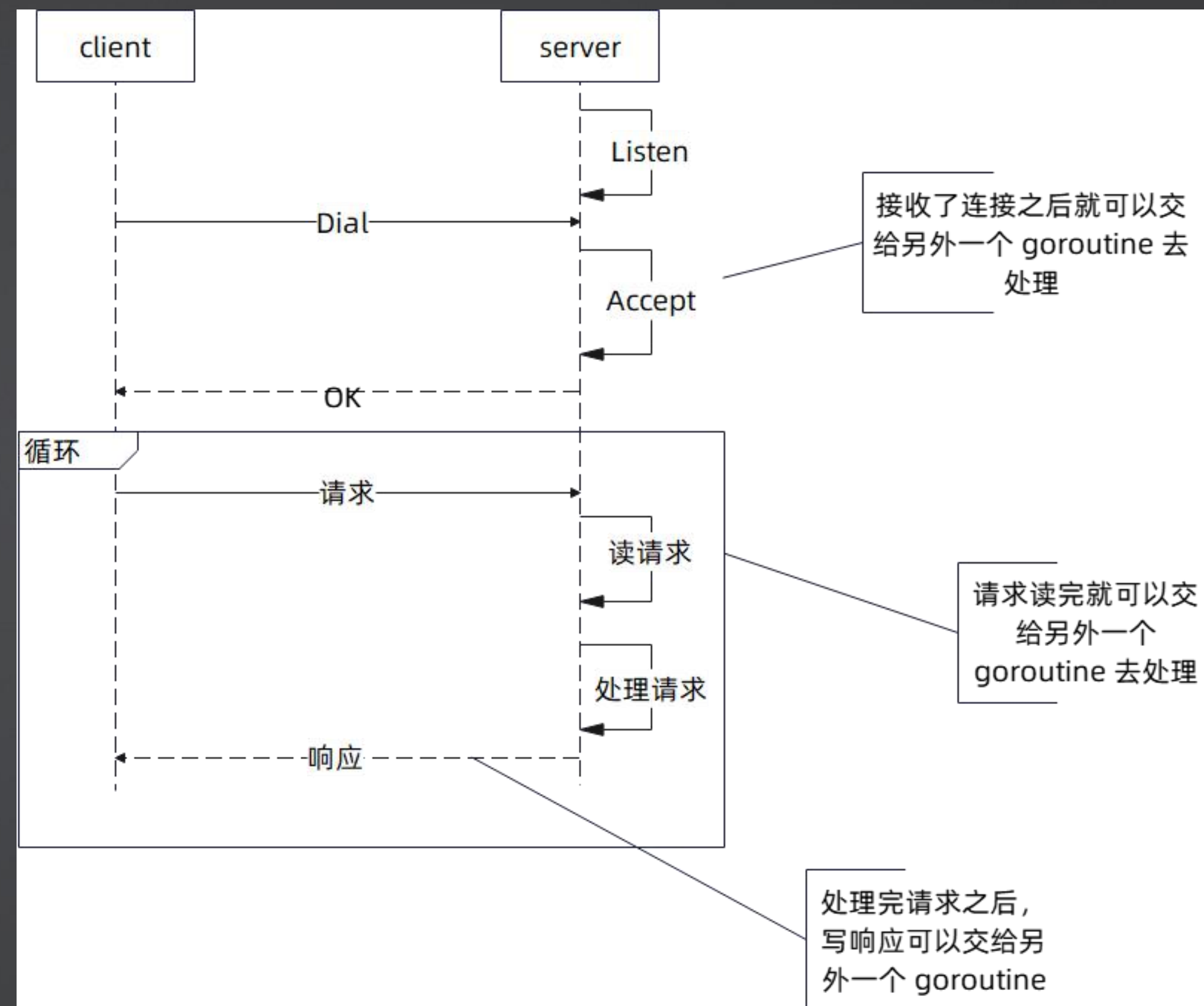
```
// 读数据
bs := make([]byte, 8)
_, err := conn.Read(bs)
if err == io.EOF || err == net.ErrClosed ||
    err == io.ErrUnexpectedEOF {
    // 一般关闭的错误比较懒得管
```

```
    return err
}
res := make([]byte, 8)
_, err = conn.Read(res)
if err != nil {
    return err
}
```

网络编程：goroutine 问题

前面的模板，我们是在创建了连接之后，就交给另外一个 goroutine 去处理，除了这个位置，还有两个位置：

- 在读取了请求之后，交给别的 goroutine 处理，当前的 goroutine 继续读请求
- 写响应的时候，交给别的 goroutine 去写



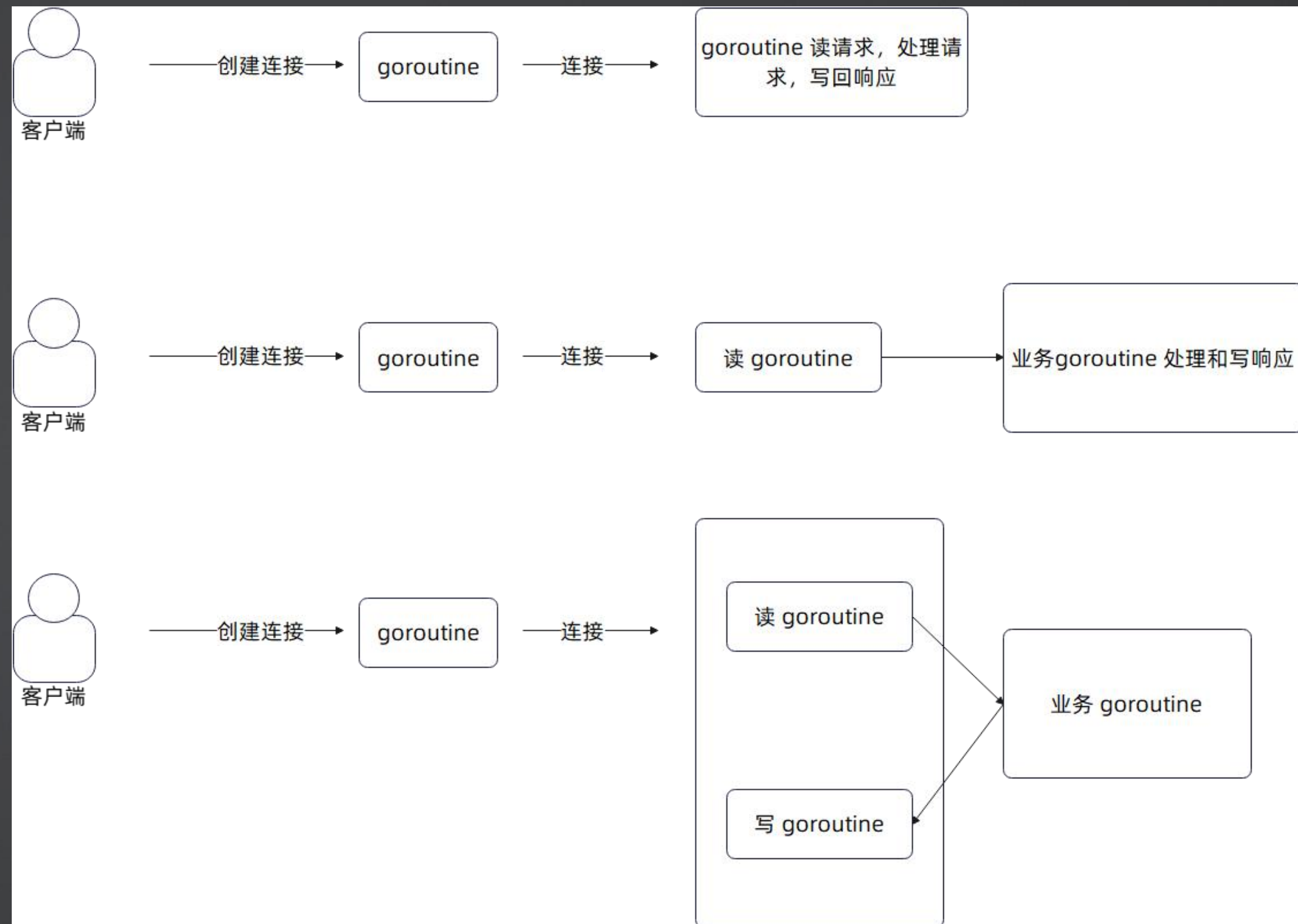
网络编程：goroutine 问题

由上至下：

- TCP 通信效率提高
- 系统复杂度提高

因为 goroutine 非常轻量，所以即便使用模式一，对于绝大多数应用来说，性能也可以满足。

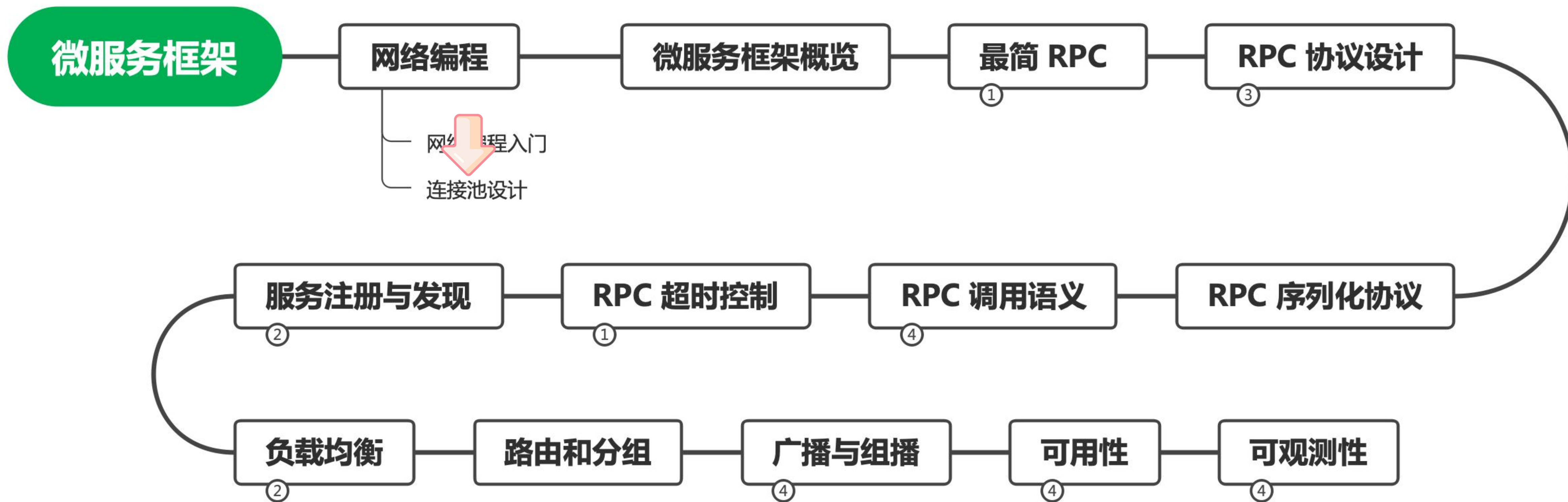
准确说，虽然很多人尝试开发新的 net 库来取代 Go 自带的，但是实际上，这些库普遍存在的问题就是 BUG 多，性能提升有限，但是编程模型及其复杂。不到逼不得已不要使用这一类的库。



面试要点

- 网络的基础知识，包含 TCP 和 UDP 的基础知识。
 - 三次握手和四次挥手
- 如何利用 Go 写一个简单的 TCP 服务器。直接面 net 里面的 API 是很少见的，但是如果有编程题环节，那么可能会让你直接写一个简单的 TCP 服务器。
- 记住 goroutine 和连接的关系，可以在不同的环节使用不同的 goroutine，以充分利用 TCP 的全双工通信。

学习路线



连接池

在前面的示例代码里面，我们客户端创建的连接都是一次性使用。然而，创建一个连接是非常昂贵的：

- 要发起系统调用
- TCP 要完成三次握手
- 高并发的情况，可能耗尽文件描述符

连接池就是为了复用这些创建好的连接。

开源实例：silenceper/pool

Github 地址：<https://github.com/silenceper/pool>

- **InitialCap**: 这种参数是在初始化的时候直接创建好的连接数量。过小，启动的时候可能大部分请求都需要创建连接；过大，则浪费。
- **MaxIdle**: 最大空闲连接数，过大浪费，过小无法应付突发流量。
- **MaxCap**: 最大连接数。

```
//factory 创建连接的方法
factory := func() (interface{}, error) { return net.Dial("tcp", "127.0.0.1:4000") }

//close 关闭连接的方法
close := func(v interface{}) error { return v.(net.Conn).Close() }

//ping 检测连接的方法
//ping := func(v interface{}) error { return nil }

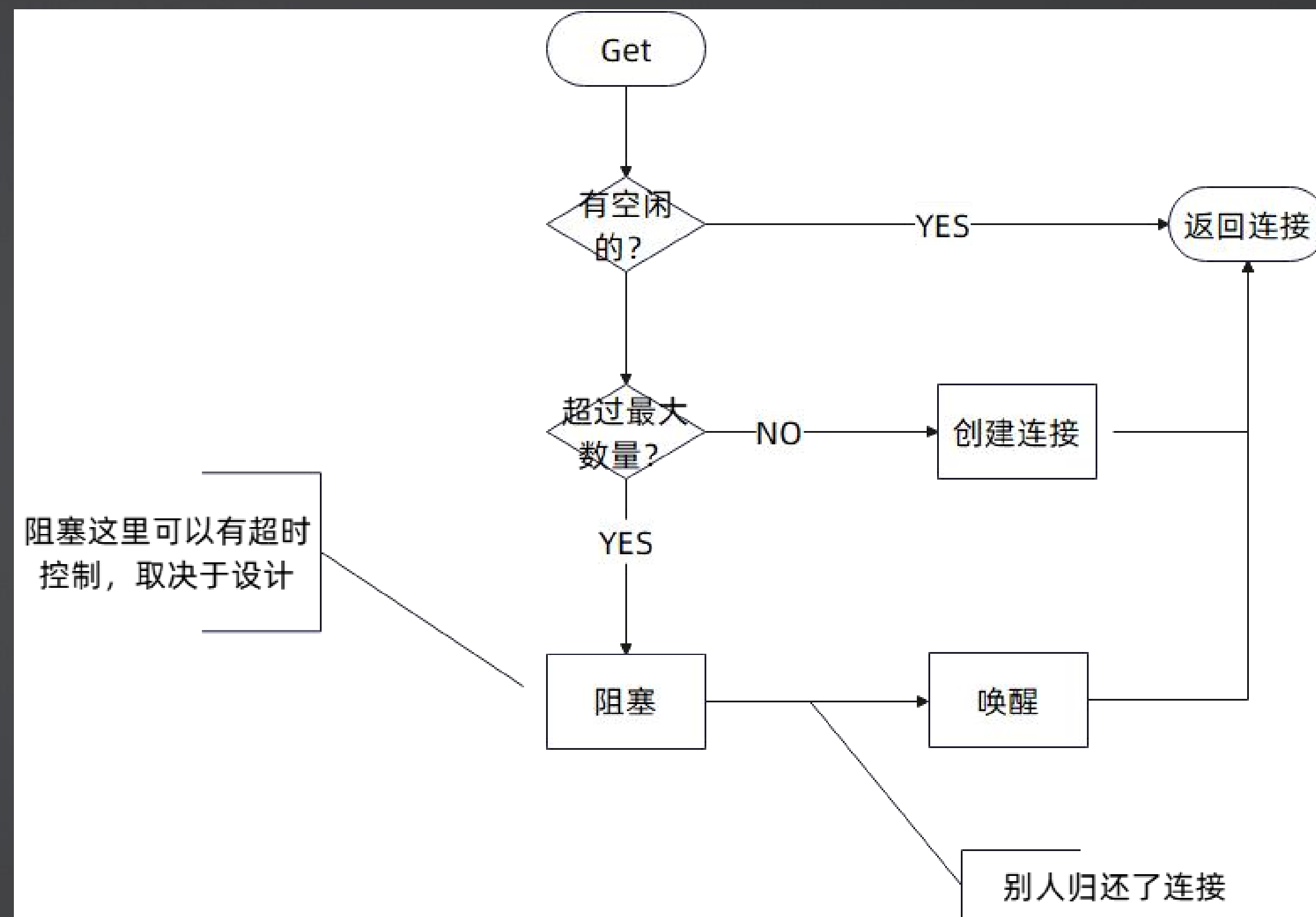
//创建一个连接池： 初始化5，最大空闲连接是20，最大并发连接30
poolConfig := &pool.Config{
    InitialCap: 5, //资源池初始连接数
    MaxIdle:    20, //最大空闲连接数
    MaxCap:     30, //最大并发连接数
    Factory:    factory,
    Close:      close,
    //Ping:      ping,
    //连接最大空闲时间，超过该时间的连接 将会关闭，可避免空闲时连接EOF，自动失效的问题
    IdleTimeout: 15 * time.Second,
}

p, err := pool.NewChannelPool(poolConfig)
if err != nil {
    fmt.Println("err=", err)
}
```

基本上连接池都会设计这几个参数

开源实例：一般连接池 Get 处理流程

- 阻塞的地方可以有超时控制，例如最多阻塞1s。
- 从空闲处取出来的连接，可能需要进一步检查这个连接有没有超时（就是很久没用了）。



开源实例：silenceper/pool Get 方法

```
// channelPool 存放连接信息
type channelPool struct {
    mu                sync.RWMutex
    conns             chan *idleConn 空闲的
    factory           func() (interface{}, error)
    close             func(interface{}) error
    ping             func(interface{}) error
    idleTimeout, waitTimeOut time.Duration
    maxActive         int
    openingConns      int
    connReqs          []chan connReq 被阻塞的Get请求
}
```

```
// Get 从pool中取一个连接
func (c *channelPool) Get() (interface{}, error) {
    conns := c.getConns()
    if conns == nil {
        return nil, ErrClosed
    }
    for {
        select {
        case wrapConn := <-conns:
            if wrapConn == nil {
                return nil, ErrClosed
            }
        }
    }
}
```

防止并发修改，主要是防止 Release 方法

开源实例：silenceper/pool Get 方法

```
for {
    select {
    case wrapConn := <-conns: 这是有空闲链接的
        if wrapConn == nil {
            return nil, ErrClosed
        }
        //判断是否超时，超时则丢弃 检查超时
        if timeout := c.idleTimeout; timeout > 0 {
            if wrapConn.t.Add(timeout).Before(time.Now()) {
                //丢弃并关闭该连接
                c.Close(wrapConn.conn)
                continue
            }
        }
        //判断是否失效，失效则丢弃，如果用户没有设定 ping 方法，就不检查
        if c.ping != nil { 应该是检查连通性
            if err := c.Ping(wrapConn.conn); err != nil {
                c.Close(wrapConn.conn)
                continue
            }
        }
    }
    return wrapConn.conn, nil
}
```

开源实例：silenceper/pool Get 方法

```
default: // 没有空闲链接
    c.mu.Lock()
    log.Debugf("openConn %v %v", c.openingConns, c.maxActive)
    if c.openingConns >= c.maxActive {
        req := make(chan connReq, 1) // 连接池满了
        c.connReqs = append(c.connReqs, req)
        c.mu.Unlock()
        ret, ok := <-req // 阻塞在这里，直到有人放回链接
        if !ok {
            return nil, ErrMaxActiveConnReached
        }
        if timeout := c.idleTimeout; timeout > 0 { // 这里只检查了超时
            if ret.idleConn.t.Add(timeout).Before(time.Now()) {
                // 丢弃并关闭该连接
                c.Close(ret.idleConn.conn)
                continue
            } // 其实也可以考虑继续检查连通性
        }
    }
    return ret.idleConn.conn, nil
}
```

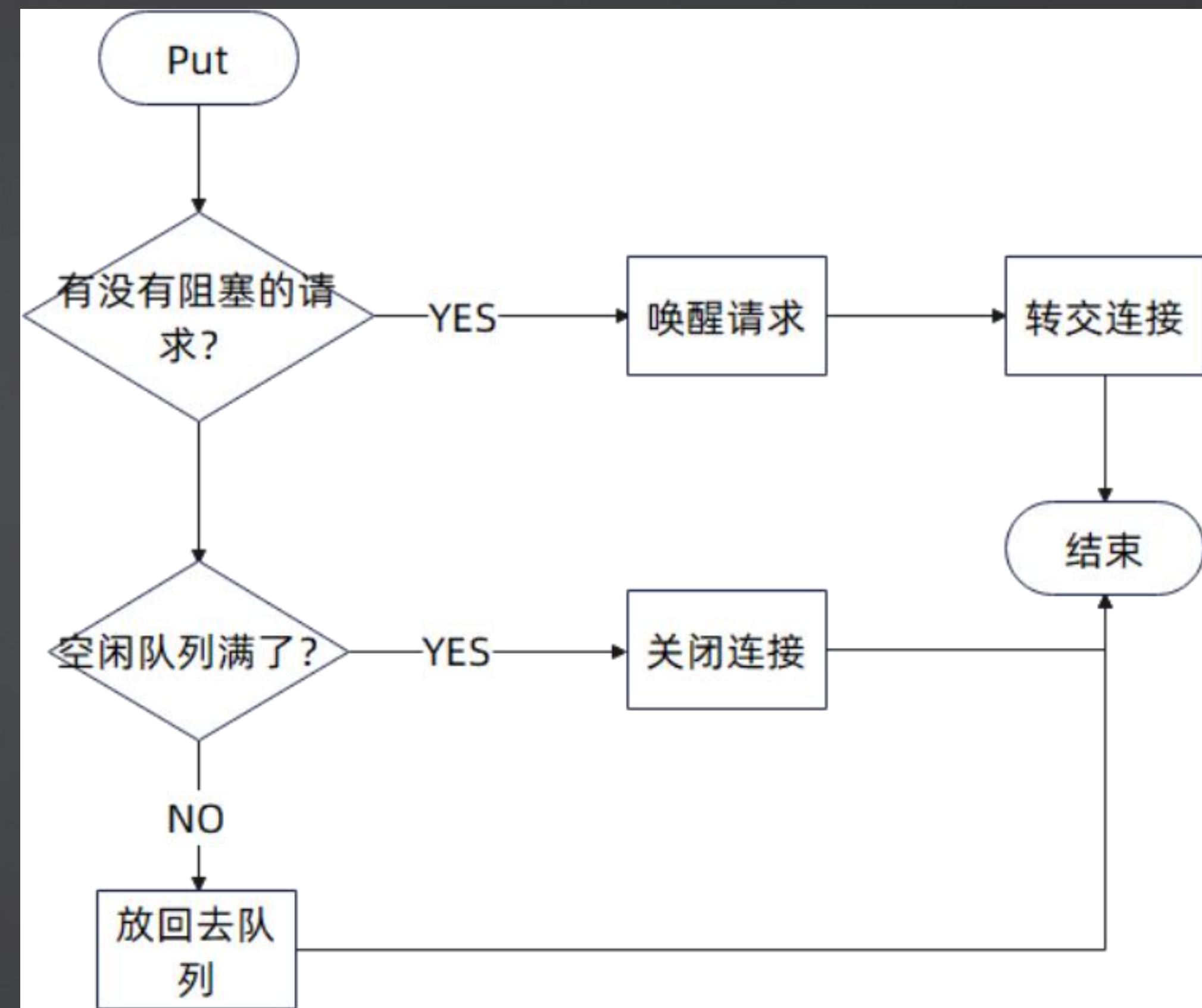
开源实例：silenceper/pool Get 方法

```
if c.factory == nil {  
    c.mu.Unlock()  
    return nil, ErrClosed  
}  
conn, err := c.factory()  
if err != nil {  
    c.mu.Unlock()  
    return nil, err  
}  
c.openingConns++  
c.mu.Unlock()  
return conn, nil  
}
```

虽然没有空闲链接，但是链接池还没满，直接创建新的

开源实例：一般连接池 Put 处理流程

- Put 会先看有没有阻塞的 goroutine（线程），有就直接转交。
- 如果空闲队列满了，又没有人需要连接，那么需要关闭这个连接。



开源实例：silenceper/pool Put 方法

```
func (c *channelPool) Put(conn interface{}) error {
    if conn == nil { errors.New("connection is nil. rejecting") }

    c.mu.Lock()

    if c.conns == nil {...}

    if l := len(c.connReqs); l > 0 {
        req := c.connReqs[0]
        copy(c.connReqs, c.connReqs[1:])
        c.connReqs = c.connReqs[:l-1]
        req <- connReq{
            idleConn: &idleConn{conn: conn, t: time.Now()},
        }
        c.mu.Unlock()
        return nil
    } else {
```

有阻塞的 Get 请求，把连接丢过去，先到先得

```
default:
    c.mu.Lock()
    log.Debugf("openConn %v %v", c.openingConns, c.maxActive)
    if c.openingConns >= c.maxActive {
        req := make(chan connReq, 1)
        c.connReqs = append(c.connReqs, req)
        c.mu.Unlock()
        ret, ok := <-req
        if !ok {
            return nil, ErrMaxActiveConnReached
        }
        if timeout := c.idleTimeout; timeout > 0 {
            if ret.idleConn.t.Add(timeout).Before(time.Now()) {
                // 丢弃并关闭该连接
                c.Close(ret.idleConn.conn)
                continue
            }
        }
        return ret.idleConn.conn, nil
    }
```

没有空闲链接

连接池满了

阻塞在这里，直到有人放回链接

这里只检查了超时

其实也可以考虑继续检查连通性

开源实例：silenceper/pool Put 方法

```
    } else {  
        select {  
        case c.conns <- &idleConn{conn: conn, t: time.Now()}:  
            c.mu.Unlock()    可以放到空闲连接  
            return nil  
        default:  
            c.mu.Unlock()    满了直接关  
            //连接池已满，直接关闭该连接  
            return c.Close(conn)  
        }  
    }  
}
```

开源实例：silenceper/pool 总结

总结：

- **Get 要考虑：**
 - 有空闲连接，直接返回
 - 否则，没超过最大连接数，直接创建新的
 - 否则，阻塞调用方
- **Put 要考虑：**
 - 有 Get 请求被阻塞，把连接丢过去
 - 否则，没超过最大空闲连接数，放到空闲列表
 - 否则，直接关闭



连接池运作图解：起步

刚开始啥都没有，**直接创建一个新的。**



连接池运作图解：超过上限

假如说我们不断请求连接，直到超过了十个连接。

请求被阻塞。



连接池运作图解：放回去，有阻塞请求

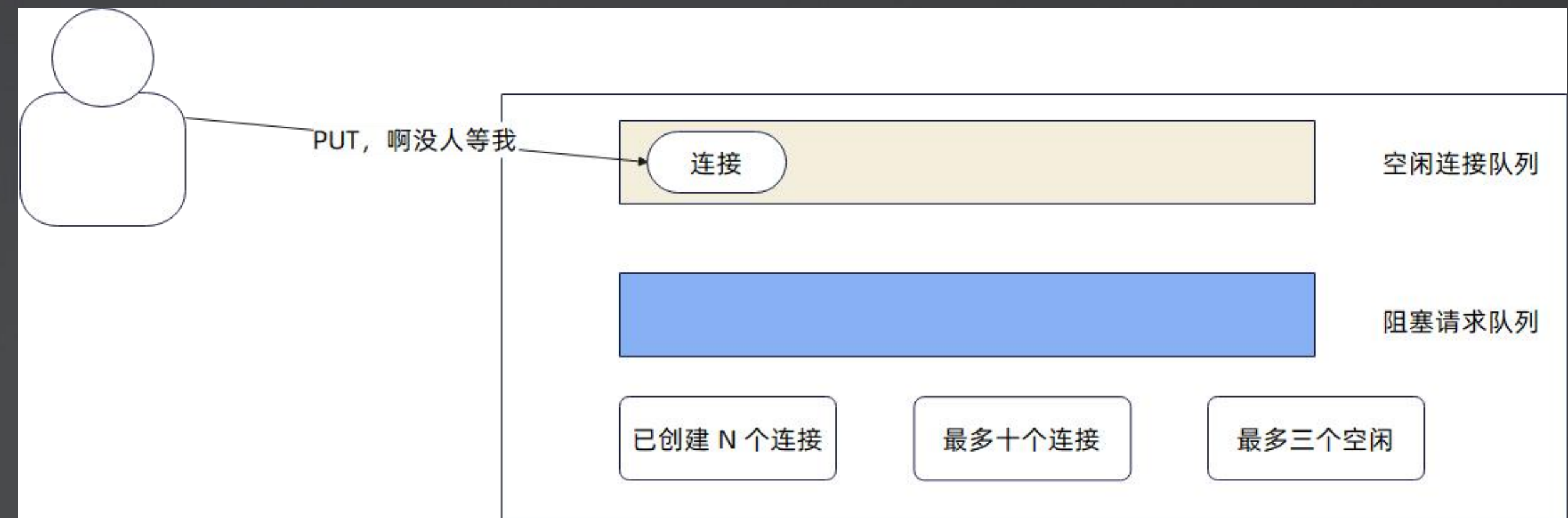
假如说这时候有人用完了连接，就放回来了。

唤醒一个请求，然后将连接交过去。



连接池运作图解：放回去空闲连接队列

如果这个时候没有阻塞请求，并且此时空闲连接队列还没有满，那么就**放回去空闲连接队列**。



连接池运作图解：空闲连接队列满了

空闲队列都满了，只能关掉这个连接了。



连接池运作图解：从空闲连接队列 GET

空闲队列有可用连接，直接拿。



开源实例：sql.DB 中连接池管理

它也基本遵循前面总结的：

- 利用 channel 来管理空闲连接
- 利用一个队列来阻塞请求

sql.DB 有很多细节，这里我们只是看它怎么管连接的。

```
mu          sync.Mutex    // protects following fields
freeConn    []*driverConn // free connections ordered by returnedAt oldest
connRequests map[uint64]chan connRequest
nextRequest uint64        // Next key to use in connRequests.
numOpen     int           // number of opened and pending open connections
// Used to signal the need for new connections
```

```
// result, rows)
type driverConn struct {
    db          *DB
    createdAt   time.Time

    sync.Mutex // guards following
    ci         driver.Conn
    needReset  bool // The connection
    closed     bool
    finalClosed bool // ci.Close has been called
    openStmt   map[*driverStmt]bool

    // guarded by db.mu
    inUse      bool
    returnedAt time.Time // Time the connection was
    onPut      []func() // code (with db.mu) to run when connection is put back
    dbmuClosed bool      // same as closed, but for the database mutex
}
```

开源实例：sql.DB conn

获取一个连接在 sql.DB 里面是 conn 方法，基本过程和我们前面讲的差不多。

```
// conn returns a newly-opened or cached *driverConn.
func (db *DB) conn(ctx context.Context, strategy connReuseStrategy) (*driverConn, error) {
    db.mu.Lock()
    if db.closed {                检测连接池有没有被关闭
        db.mu.Unlock()
        return nil, errDBClosed
    }
    // Check if the context is expired.
    select {
    default:                检测 ctx 有没有超时
    case <-ctx.Done():
        db.mu.Unlock()
        return nil, ctx.Err()
    }
}
```

开源实例：sql.DB conn

sql.DB 是从队尾开始拿空闲连接的，为什么？

因为队首的空闲连接更可能已经超过了最大空闲时间

```
// Prefer a free connection, if possible.
last := len(db.freeConn) - 1
if strategy == cachedOrNewConn && last >= 0 {
    // Reuse the lowest idle time connection so we can close
    // connections which remain idle as soon as possible.
    conn := db.freeConn[last]          尝试从空闲连接里面拿一个
    db.freeConn = db.freeConn[:last]
    conn.inUse = true
    if conn.expired(lifetime) {
        db.maxLifetimeClosed++          检测空闲时间
        db.mu.Unlock()
        conn.Close()
        return nil, driver.ErrBadConn
    }
    db.mu.Unlock()

    // Reset the session if required. 重置会话，不用管
    if err := conn.resetSession(ctx); errors.Is(err, driver.ErrBadConn) {
        conn.Close()
        return nil, err
    }

    return conn, nil
}
```

开源实例：sql.DB conn

```
// Out of free connections or we were asked not to use one. If we're not
// allowed to open any more connections, make a request and wait.
if db.maxOpen > 0 && db.numOpen >= db.maxOpen {
    // Make the connRequest channel. It's buffered so that the
    // connectionOpener doesn't block while waiting for the req to be read.
    req := make(chan connRequest, 1)
    reqKey := db.nextRequestKeyLocked()
    db.connRequests[reqKey] = req
    db.waitCount++
    db.mu.Unlock()

    waitStart := nowFunc()

    // Timeout the connection request with the context.
    select {
    case <-ctx.Done():...
    case ret, ok := <-req:...
    }
}
```

超过了最大连接数，阻塞在这里等别人归还

训练营

```
// Timeout the connection request with the context.
select {
case <-ctx.Done():
    // Remove the connection request and ensure no value has been sent
    // on it after removing.
    db.mu.Lock()
    delete(db.connRequests, reqKey)
    db.mu.Unlock()

    atomic.AddInt64(&db.waitDuration, int64(time.Since(waitStart)))
```

超时了

```
select {
default:
    // 因为你的请求还在，所以超时之后你要转发连接
case ret, ok := <-req:
    if ok && ret.conn != nil {
        db.putConn(ret.conn, ret.err, resetSession: false)
    }
}
return nil, ctx.Err()
```

```
case ret, ok := <-req:
    atomic.AddInt64(&db.waitDuration, int64(time.Since(waitStart)))
    // 等到了别人归还的请求

    if !ok : nil, errDBClosed ↗
    // Only check if the connection is expired if the strategy is cached
    // If we require a new connection, just re-use the connection without
    // at the expiry time. If it is expired, it will be checked when it
    // back into the connection pool.
    // This prioritizes giving a valid connection to a client over the
    // lifetime, which could expire exactly at this point anyway.
    if strategy == cachedOrNewConn && ret.err == nil && ret.conn.expired
    if ret.conn == nil : nil, ret.err ↗

    // Reset the session if required.
    if err := ret.conn.resetSession(ctx); errors.Is(err, driver.ErrBadConn) {
        ret.conn.Close()
        return nil, err
    }
    return ret.conn, ret.err
```

执行各种检测

开源实例：sql.DB conn

```
db.numOpen++ // optimistically
db.mu.Unlock()
ci, err := db.connector.Connect(ctx)
if err != nil {                                新建一个连接
    db.mu.Lock()
    db.numOpen-- // correct for earlier optimism
    db.maybeOpenNewConnections()
    db.mu.Unlock()
    return nil, err
}
db.mu.Lock()
dc := &driverConn{
    db:      db,
    createdAt: nowFunc(),
    returnedAt: nowFunc(),
    ci:      ci,
    inUse:    true,
}
db.addDepLocked(dc, dc)
db.mu.Unlock()
return dc, nil
```

开源实例：sql.DB putConn

因为本身 DB 比较复杂，所以在 putConn 的时候要**做很多校验**，维持好整体状态：

- 处理 ErrBadConn 的情况
- 确保 dc 并没有任何人在使用
- 处理超时

```
// putConn adds a connection to the db's free pool.
// err is optionally the last error that occurred on this connection.
func (db *DB) putConn(dc *driverConn, err error, resetSession bool) {
    if !errors.Is(err, driver.ErrBadConn) {
        if !dc.validateConnection(resetSession) {
            err = driver.ErrBadConn
        }
    }
    db.mu.Lock()
    if !dc.inUse {
        db.mu.Unlock()
        if debugGetPut {
            fmt.Printf("putConn({dc}) DUPLICATE was: #{stack()}\n\nPRE
        }
        panic(v: "sql: connection returned that was never out")
    }
}
```

可以简单理解为关闭的原因

```
if !errors.Is(err, driver.ErrBadConn) && dc.expired(db.maxLifetime) {
    db.maxLifetimeClosed++
    err = driver.ErrBadConn
}
if debugGetPut {
    db.lastPut[dc] = stack()
}
dc.inUse = false
dc.returnedAt = nowFunc()

for _, fn := range dc.onPut {
    fn()
}
dc.onPut = nil
```

这一堆代码不用特别去管，因为你设计的连接池可能压根不需要维持那么多的状态

开源实例：sql.DB putConn

```
if errors.Is(err, driver.ErrBadConn) {  
    // Don't reuse bad connections.  
    // Since the conn is considered bad and i  
    // as closed. Don't decrement the open co  
    // take care of that.  
    db.maybeOpenNewConnections()  
    db.mu.Unlock()  
    dc.Close()  
    return  
}  
if putConnHook != nil {  
    putConnHook(db, dc)  
}  
added := db.putConnDBLocked(dc, err: nil)  
db.mu.Unlock()  
  
if !added {  
    dc.Close()  
    return  
}
```

这个也不具备参考价值

关键步骤就在这里

开源实例：sql.DB putConn

putConnDBLocked 步骤和 silenceper/pool 的 Put 流程几乎一致。

```
func (db *DB) putConnDBLocked(dc *driverConn, err error) bool {
    if db.closed {
        return false
    }
    if db.maxOpen > 0 && db.numOpen > db.maxOpen {
        return false // 超出上限, 直接关闭
    }
    if c := len(db.connRequests); c > 0 {
        var req chan connRequest
        var reqKey uint64
        for reqKey, req = range db.connRequests {
            break
        }
        delete(db.connRequests, reqKey) // Remove from pending
        if err == nil {
            dc.inUse = true
        }
        req <- connRequest{
            conn: dc,
            err:   err,
        }
        return true
    } else if err == nil && !db.closed {
```

```
        } else if err == nil && !db.closed {
            if db.maxIdleConnsLocked() > len(db.freeConn) {
                db.freeConn = append(db.freeConn, dc)
                db.startCleanerLocked() // 放到空闲链表
                return true
            }
            db.maxIdleClosed++
        }
    }
}
```

面试要点

- 几个参数的含义：初始连接、最大空闲连接、最大连接数，最大空闲时间
- 连接池的运作原理：拿连接会发生什么，放回去又会发生什么，记住我们的流程图
- sql.DB 解决过期连接的懒惰策略，可以类比其它如本地缓存的
- 实现一个简单的连接池：那种注重考察代码能力的公司可能会让你手写代码。

Q & A

THANKS