Go标准库

xbZhong

2025-02-20

本页PDF

标准库

net/http

常见方法总览

服务端

- `http.ListenAndServe()`：全局方法，里面建立`Server`对象并嵌套执行了`Server.ListenAndServe()`方法
  - `Server.ListenAndServe()`：调用了`Server.Serve()`方法
    - `Server.Serve()`：服务器主协程工作原理
      - `conn.Serve()`：每个连接对象要执行的方法，为每个连接对象创建了一个协程
        - `conn.readRequest()`：获取每个连接的响应对象
        - `ServerHandler.ServeHTTP()`：选择最终要使用的`Handler`，由它来处理请求
          - `Handler.ServeHTTP()`：使用找到的处理器执行业务方法
          - `ServeMux.Handler()`：将请求体进行拆分，调用内部的`ServeMux.handler()`
            - `ServeMux.handler()`：执行真正的匹配逻辑
              - `ServeMux.match()`：进行路由、请求匹配
- `http.Handle()`：注册路由，传入实现了`Handler`接口的结构体
- `http.HandleFunc()`：注册路由，传入函数
- `Request.URL.Query()`：获取`url.Values`类型
  - `url.Values.Get()`：获取查询参数

客户端

- `http.Post()`：公开方法，使用默认的`DefaultClient`处理请求
- `Client.Post()`：构造请求参数，调用`Client.Do()`方法
  - `NewRequestWithContext()`：构造`Request`实例
  - `Client.Do()`：对请求参数进行校验等工作
    - `Client.do()`：重定向机制在此工作，获取超时时间
      - `Client.send()`：更新`CookieJar`中的`cookie`
        - `send()`：调用对外开放的`Transport.RoundTrip()`方法
          - `Transport.RoundTrip()`：调用私有的`Transport.roundTrip()`方法
            - `Transport.roundTrip()`：执行请求的发送

服务端

Server

- 整个http服务端模块都被封装在`Server`类中
- 常见组成部分
  - `Addr`：监听地址

- `Handler`：处理器，实现ServeHTTP方法的Handler接口
  - 为空的话就使用默认的`DefaultServeMux`
- `ReadTimeout`：读取整个请求的超时时间
- `WriteTimeout`：写入响应的超时时间
- `IdleTimeout`：空闲等待的最长时间

```go
type Server struct {
    Addr    string
    Handler Handler
    ReadTimeout    time.Duration
    WriteTimeout   time.Duration
    IdleTimeout    time.Duration
}
```

ServeMux

- 常见组成部分
  - `mu`：读写锁，**读共享写互斥**
  - `m`：精确路由表
  - `es`：排序的前缀路由表，按路由长度递减存储
  - `hosts`：路由里面是否包含主机名

```go
type ServeMux struct {
    mu    sync.RWMutex
    m     map[string]muxEntry
    es    []muxEntry
    hosts bool
}
```

- 实现路由注册：`ServeMux.Handle`
  - 将`path`和`handler`包装成一个`muxEntry`，以`path`为key注册到`ServeMux.m`中
  - 对于以`/`结尾的`path`，根据路由长度有顺序地插入到数组`ServeMux.es`中

```go
func (mux *ServeMux) Handle(pattern string,handler Handler){
    // 加写锁
    mux.mu.Lock()
    defer mux.mu.Unlock()
    // ...
    e := muxEntry{h:handler,pattern:pattern}
    mux.m[pattern] = e
    // 有序插入到数组中
    if pattern[len(pattern)-1] == '/' {
        mux.es = appendSorted(mux.es, e)
    }
}
```

muxEntry

- 常见组成部分
  - `h`：处理器
  - `pattern`：请求路径

```go
type muxEntry struct {
    h       Handler
    pattern string
}
```

Handler（接口）

- 任何结构体只要实现了 `ServeHTTP` 方法，他就是一个 `Handler`

```go
type Handler interface{
    ServeHTTP(ResponseWriter, *Request)
}

// 例子
type MyHandler struct{}

func (h MyHandler) ServeHTTP(w ResponseWriter, r *Request) {
    w.Write([]byte("Hello from MyHandler!"))
}
```

HandlerFunc（函数类型）

- 可以理解成是一个适配器，它把一个普通的函数变成了 `Handler`
- 它自身实现了 `ServeHTTP` 方法，调用函数自身，也说明**它实现了** `Handler` **接口**

```go
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP方法
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request){
    // 调用函数自身
    f(w,r)
}
```

无论使用 `Handler` 还是 `HandlerFunc` ，内部调用的还是 `ServeHTTP` 方法

http.Handle

- 使用**默认的多路复用器**注册路由
- 接受 `Handler` 接口

```go
func Handle(pattern string, handler Handler){
    // 注册路由
    DefaultServeMux.Handle(pater,handler)
}
```

http.HandleFunc

- 使用**默认的多路复用器**注册路由
- 接受函数
- 内部最终还是使用 `ServeMux.Handle` 方法注册路由，**使用** `HandlerFunc` **进行强转**
    - 因为HandlerFunc自身实现了ServeHTTP方法，所以它实现了Handler接口

```go
func HandleFunc(pattern string, handler func(ResponseWriter, *Request)){
    DefaultServeMux.HandleFunc(pattern, handler)
}


// ServeMux.HandleFunc
func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter,
*Request)) {
    // 关键行：这里调用了 Handle!
    mux.Handle(pattern, HandlerFunc(handler))
}
```

启动与连接处理

公开方法内部会创建一个新的 `Server` 对象，嵌套执行 `Server.ListenAndServe()` 方法

```go
http.ListenAndServe(":8080",nil)

// 内部逻辑
func ListenAndServe(addr string, handler Handler) error {
    server := &Server{Addr: addr, Handler: handler}
    return server.ListenAndServe()
}
```

- `:8080`：冒号前是ip，留空表示监听所有网卡
- `nil`：处理器，填`nil`表示使用Go**默认的多路复用器**（`DefalutServeMux`）

`Server.ListenAndServe()`方法

- 根据用户传入的端口申请一个监听器`listener`调用`Server.Serve`方法

```go
func (srv *Server) ListenAndServe() error{
    // ...
    addr := srv.Adr
    if addr == ""{
        addr = ":http"
    }
    ln,err := net.Listen("tcp",addr)
    // ...
    return srv.Serve(ln)
}
```

`Server.Serve()`方法

- 体现http服务端运行架构，`for + listener.accept`**模式**，服务器主协程执行`for`循环
- 将`server`封装成一组kv对，塞入`context`中便于跨层传递信息
- `for`循环中，每轮循环调用`Listener.Accept()`方法**阻塞等待新连接到达**
- 每有一个**连接对象**到达，创建一个协程**异步执行**`conn.serve`方法
    - 这里是连接对象而不是请求，因为请求可以复用连接对象

```go
func (srv *Server) Serve(l net.Listener) error{
    // ...
    ctx := context.WithValue(baseCtx,ServerContextKey,srv)
    for{
        // 阻塞等待。调用os底层的accept方法
        rw,err := l.Accept()
        // ...
        connCtx := ctx
        // ...
        c := srv.newConn(rw)
        // ...
        go c.serve(connCtx)
    }
}
```

`conn.serve()` 方法

- 将 `TCP Reader` 包装成 `http Reader`
- 从 `conn` 中读取封装的 `response` 结构体以及请求参数 `http.Request`
    - 在下面的代码中就是 `w` ，它实现了 `http.ResponseWriter` 接口，因此可以作为 `ServeHTTP` 的参数
    - `w` ：此次请求的**响应对象**
    - `w.req` ：此次请求的**请求对象**
- 调用 `serveHandler.ServeHTTP` 方法，根据请求的 `path` 为其分配 `handler`
- 使用 `for` 循环实现**TCP**的连接复用和 `Keep-Alive`

```go
func (c *conn) serve(ctx context.Context){
    // ...

    // 包装成Http reader
    c.r = &connReader{conn: c}
    c.bufr = newBufioReader(c.r)
    c.bufw = newBufioWriterSize(checkConnErrorWriter{c}, 4<<10)

    for{
        w,err := c.readRequest(ctx)
        // ...
        serverHandler{c.server}.ServeHTTP(w,w.req)
        w.cancelCtx()
        // ...
    }
}
```

`serverHandler.ServeHTTP()` 方法

- 选出最终要用的 `Handler`
  - 如果 `Handler` 为空会使用 `DefaultServeMux`
- `serverHandler` 就只是把 `http.Server` 包装起来，然后调用它里面 `Handler` 的 `ServeHTTP` 方法

```
func (sh serverHandler) ServeHTTP(rw ResponseWriter, req *Request) {
    handler := sh.srv.Handler
    if handler == nil {
        handler = DefaultServeMux
    }
    // ...
    handler.ServeHTTP(rw, req)
}
```

`ServeMux.ServeHTTP()` 方法

- 调用了 `ServeMux.Handler()` 方法拿到处理器
- 使用处理器执行业务逻辑

```
func (mux *ServeMux) ServeHTTP(w ResponseWriter,r *Request){
    // ...
    h,_ = mux.Handler(r)
    h.ServeHTTP(w,r)
}
```

`ServeMux.Handler()` 方法

- 将 `r`，也就是请求结构体拆分成 `host` 和 `path`
- 调用 `ServeMux.handler()` 方法进行匹配

```
func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string){
    // ...
    return mux.handler(host,r.URL.Path)
}
```

`ServeMux.handler()` 方法

- 使用了读锁
- 进行路由匹配

```go
func (mux *ServeMux) handler(host, path string) (h Handler, pattern string) {
    mux.mu.RLock()
    defer mux.mu.RUnlock()

    // ...
    h, pattern = mux.match(path)
    // ...
    return
}
```

`ServeMux.match()` 方法

- 使用了路由字典 `ServeMux.m` 进行路由匹配
- 如果匹配不到则采用模糊匹配，在 `ServeMux.es` 中进行匹配
  - 找到一个与请求路径 `path` 前缀完全匹配且长度最长的 `pattern` ，其对应的 `handler` 会作为本次请求的处理函数

```go
func (mux *ServeMux) match(path string)(h Handler,pattern string){
    v,ok := mux.m[path]
    if ok{
        return v.h,v.pattern
    }
    // 在ServeMux.es中进行模糊匹配
    for _, e := range mux.es {
        if strings.HasPrefix(path, e.pattern) {
            return e.h, e.pattern
        }
    }
    return nil, ""
}
```

客户端

Client

- 常见组成部分
  - `Transport` ： 负责http通信的核心部分
    - 负责建立TCP/TLS连接、连接池、代理等
  - `Jar` ： Cookie管理
  - `Timeout` ： **全局的超时时间**
- 不要new `Client` 和 `Transport` ，会导致复用失效

```go
type Client struct {
    // ...
    Transport RoundTripper
    // ...
    Jar CookieJar
    // ...
    Timeout time.Duration
}
```

RoundTripper

- 本质上是一个接口
- 需要实现方法 `RoundTrip()` ，通过传入请求 `Request` ，获得响应 `Response`

```go
type RoundTripper interface {
    RoundTrip(*Request) (*Response,error)
}
```

CookieJar

- 实现 `Cookie` 的存储与匹配策略
- 本质上是一个接口
- 发送前通过URL从 `CookieJar` 取cookie
- 获取响应后把 `Set-Cookie` 和 `URL` 存回 `CookieJar`

```go
type CookieJar interface {
    SetCookies(u *url.URL,cookies []*http.Cookie)
    Cookies(u *url.URL) []*http.Cookie
}
```

Transport

- 是 `RounTripper` 的实现类
- 核心字段
  - `idleConn` ： 存放当前可复用的空闲连接
    - 无法直接操作
    - 使用之后需**关闭响应体**，否则连接回不到 `idle` 池内
  - `DialContext()` ： 创建新的TCP连接的函数
  - `ResponseHeaderTimeout` ： 从写完请求到拿到响应的超时时间
  - `TLSHandshakeTimeout` ： TLS握手超时时间
  - `MaxIdleConns` ： 全局最大空闲连接数
  - `IdleConnTimeout` ： 空闲连接多久回收
  - `MaxIdleConnsPerHost` ： 单个 `host` 最大空闲连接数
  - `MaxConnsPerHost` ： 每个 `host` 的连接数上限

- `DisableCompression`：为真则禁用数据压缩
- `DisableKeepAlives`：为真则禁止连接复用

```go
type Transport struct {
    // ...
    idleConn map[connectMethodKey][]*persistConn

    TLSHandshakeTimeout   time.Duration

    DisableKeepAlives     bool

    DisableCompression    bool

    MaxIdleConns          int
    MaxIdleConnsPerHost   int
    MaxConnsPerHost       int
    IdleConnTimeout       time.Duration

    ResponseHeaderTimeout time.Duration
    DialContext func(ctx context.Context,network,addr string) (net.Conn,error)
    // ...
}
```

Request

- http请求参数结构体

```go
type Request struct {
    // 方法
    Method string
    // 请求路径
    URL *url.URL
    // 请求头
    Header Header
    // 请求参数内容
    Body io.ReadCloser
    // 服务器主机
    Host string
    // query请求参数
    Form url.Values
    // 响应参数
    Response *Response
    // 请求链路上下文
    ctx context.Context
    // ...
}
```

Response

- http响应参数结构体

```go
type Response struct {
    // 请求状态
    StatusCode int
    // http协议
    Proto string
    // 请求头
    Header Header
     // 响应参数内容
    Body io.ReadCloser
    // 指向请求参数
    Request *Request
    // ...
}
```

方法链路总览

- 构造HTTP请求参数
- 获取用于与服务端交互的TCP连接
- 通过TCP连接发送请求
- 通过TCP连接获取响应

`http.Post()` 方法

- 会使用默认的 `DefaultClient` 处理请求
- 需传入 `url` ，请求参数格式 `contentType` ，以及请求参数的 `io.Reader`

```
var DefaultClient = &Client{}

func Post(url,contentType string,body io.Reader) (resp *Response,err error) {
   return DefaultClient.Post(url, contentType, body)
}
```

`Client.Post()` 方法

- 根据用户入参构造完整的请求参数 `Request`
- 通过 `Client.Do()` 处理请求

```
func (c *Client) Post(url,contentType string,body io.Reader) (resp *http.Response
,err error){
    req,err := NewRequestWithContext(context.Background(),"POST",url,body)
    // ...
    req.Header.Set("Content-Type",contentType)
    return c.Do(req)
}
```

`NewRequestWithContext()` 方法

- 根据用户传入的信息构造了 `Request` 实例
- 注意
  - 这里的 `rc` 会被构建到 `http.Request` 里面，为了保证他能够被关闭，需要确保 `rc` 是 `io.ReadCloser` 类型（既可读也可关）
  - 如果 `body` 是 `nil` ，需要赋 `http.NoBody` 给它，否则后面关闭时会 `panic`
  - `io.NopCloser()` 可以把一个 `io.Reader` 包装成 `io.ReadCloser`
- 完整代码应该要先判断 `body` 是否为空，不为空则判断它是否是 `io.ReadCloser` 类型，不是这个类型调用 `io.NopCloser()` 方法将他转为 `io.ReadCloser` 类型

```go
func NewRequestWithContext(ctx context.Context,method,url string,body io.Reader)
(*Request, error) {
    // ...
    u, err := urlpkg.Parse(url)
    // ...
    rc, ok := body.(io.ReadCloser)
    // ...
    req := &Request{
        ctx:        ctx,
        Method:     method,
        URL:        u,
        // ...
        Header:     make(Header),
        Body:       rc,
        Host:       u.Host,
    }
    // ...
    return req, nil
}
```

`Client.Do()` 方法

- 负责对请求进行整理，参数校验

```go
func (c *Client) Do(req *http.Request) (*http.Response, error) {
    // ...
    return c.do(req)
}
```

`Client.do()` 方法

- 真正发送请求的函数
- 使用 `for` 循环进行重定向
- 使用 `Clinet.deadline()` 方法获取超时时间
- 使用 `Client.send()` 方法发送请求
- 服务端返回的响应是 `resp`
  - 需要关闭它返回的 `body`，即 `resp.Body`，否则连接无法复用，资源容易耗尽
  - 一般使用 `defer resp.Body.Close()`

```go
func (c *Client) do(req *http.Request) (retres *http.Response,reterr error){
    var(
        deadline = c.deadline()
        resp *http.Response
    )

    for {
        // ...
        var err error
        if resp, didTimeout, err = c.send(req, deadline); err != nil {
            // ...
        }
        // ...
    }
}
```

`Client.send()` 方法

- 将 `request` 交给 `Transport.RoundTrip()` 发出去
- 需要注入 `RoundTripper` 模块，默认使用全局单例 `DefaultTransport` 进行注入，通过 `Client.transport()` 方法实现
- 返回的 `didTimeout()` 是一个闭包，用来告诉上层是否是因为触发了 `Client.Timeout` 导致的错误
- 流程
  - 从 `CookieJar` 里的 `cookie` 塞进请求头
  - 发送一次请求
  - 将响应里的 `Set-Cookie` 写回 `CookieJar`

```go
func(c *Client) send(req *http.Request,deadline time.Time) (resp
*http.Response,didTimeout func() bool,err error){
    // 设置 cookie 到请求头
    if c.Jar != nil{
        for _,cookie := range c.Jar.Cookies(req.URL){
            req.AddCookie(cookie)
        }
    }
    // 发送请求
    resp,didTimeout,err = send(req,c.transport(),deadline)
    // 更新 resp 的 cookie 到请求头中
    if c.Jar != nil{
        if rc := resp.Cookies();len(rc) > 0{
            c.Jar.SetCookies(req.URL,rc)
        }
    }
    return resp,nil,nil
}
```

`Client.transport()` 方法

- 选择最后要使用的 `Transport`
- 如果没有配置使用 `Transport`，使用全局单例 `DefaultTransport`

```
// 全局单例
var DefaultTransport RoundTripper = &Transport{
    // ...
    DialContext: defaultTransportDialContext(&net.Dialer{
        Timeout:   30 * time.Second,
        KeepAlive: 30 * time.Second,
    }),
    // ...
}

func (c *Client) transport() RoundTripper {
    if c.Transport != nil {
        return c.Transport
    }
    return DefaultTransport
}
```

`send()` 方法

- 是全局私有方法
- 调用 `RoundTrip()` 方法发送请求

```
func send(ireq *http.Request,rt RoundTripper,deadline time.Time) (resp
*http.Response,didTimeout func() bool, err error) {
    // ...
    resp, err = rt.RoundTrip(req)
    // ...
    return resp, nil, nil
}
```

`RoundTripper.RoundTrip()` 方法

- 里面调用了 `RoundTripper.roundTrip()` 方法

```
func (t *Transport) RoundTrip(req *http.Request) (*http.Response, error) {
    return t.roundTrip(req)
}
```

`RoundTripper.roundTrip()` 方法

- 将 `http.Request` 包装成 `transportRequest`
- 通过复用或新建拿到一个可用连接
- 执行真正的请求和响应
- 使用 `for` 循环进行重试，和 `Client.do()` 的重定向机制不一样

```
func (t *Transport) roundTrip(req *http.Request) (*http.Response,error) {
    // ...
    for{
        // ...
        // 包装结构体

treq := &transportRequest{Request: req, trace: trace, cancelKey: cancelKey}
        // ...
        // 拿到可用连接
        pconn,err := t.getConn(treq,cm)
        // ...
        // 发送请求
        resp, err = pconn.roundTrip(treq)
        // ...
    }
}
```

获取输入

获取path

```
path := r.URL.Path
```

获取请求方法

```
method := r.Method
```

读URL参数

```
// 获取查询参数
query := r.URL.Query()
keyword := query.Get("q")
```

读路径参数

- 需要预先在路由进行定义

```
id := r.PathValue("id")
```

获取Header

```
ua := r.Header.Get("User-Agent")
auth := r.Header.Get("Authorization")
ct := r.Header.Get("Content-Type")
```

读JSON

- `r.Body`：需要确保读完之后关闭，否则会影响连接复用和资源

```
type Req struct{
    // ...
}

var req Req
// 创建JSON解码器
dec := json.NewDecoder(r.Body)
if err := dec.Decode(&req); err != nil{
     // ...
}
```

发送输出

这里有严格的**顺序要求**

1. 设置 `Header`
2. 设置 `Status Code`
3. 写入 `Body`

```
// 1. 先设 Header
w.Header().Set("Content-Type", "application/json")
// 2. 再设状态码
w.WriteHeader(http.StatusCreated) // 201
// 3. 最后写数据
json.NewEncoder(w).Encode(data)
```

返回错误

- 需要传入 `http.ResponseWriter`、状态码

```
http.Error(w, "bad request", http.StatusBadRequest)
```

重定向

- 需要传入 `http.ResponseWriter` 、重定向网址、状态码

```
http.Redirect(w, r, "https://example.com", http.StatusFound)
```

中间件

HTTP请求多路复用器

`HTTP Request Multiplexer`

自定义多路复用器

- `DefaultServeMux` 是一个**全局变量**，如果第三方库偷偷给这个默认的多路复用器加了**路由**，排查起来会很头疼
- 因此最好自定义多路复用器

```
mux := http.NewServeMux() // 自己创建一个全新的、干净的 Mux
mux.HandleFunc("/hello", handler)
http.ListenAndServe(":8080", mux) // 把它传进去，而不是 nil
```

特性

- `Mux` **自身也是一个Handler**
- 它会对不干净的URL进行清洗，并发出**重定向响应**
- 遵循**最长匹配规则**，优先匹配**范围最小、最具体**的那个模式

Response.Body

这里的 `Response` 的类型是 `http.Response`

- `Body` 的类型是 `io.ReadCloser` （既可读也可关）
- 如果是 `io.Reader` ，需要使用 `io.NopCloser()` 方法包转成 `io.ReadCloser`
- 如果 `Body` 为空，需要传递 `http.NoBody` 给它，否则关闭这个 `Body` 的时候会 `panic`

context

主要作用：实现**并发协调**以及对**goroutine的生命周期控制**

- **并发安全**，内部通过加锁实现并发安全

核心数据结构

`context.Context`

- **本质上是一个接口**

- `Deadline`：返回`context`的过期时间
- `Done`：返回`context`中的`channel`，是**只读通道**
- `Err`：返回错误
- `Value`：返回`context`中`key`对应的`value`值

```go
type Context interface{
    Deadline() (deadline time.Time,ok bool)
    Done() <- chan struct{}
    Err() error
    Value(key any) any
}
```

两种`error`类型

- `Canceled`：`context`被`cancel`时报此错误
- `DeadlineExceeded`：`context`超时时会报此错误
  - `Error()`：实现`error`接口
  - `Timeout()`：返回值为真表示这个错误是因为超时导致的
  - `Temporary()`：返回值为真表示这个错误是临时错误，提醒业务可重试

```go
// Canceled错误
var Canceled = errors.New("context canceled")

// 超时错误
var deadlineExceeded error = struct{}

func (deadlineExceeded) Error() string {
    return "context deadline exceeded"
}

func (deadlineExceeded) Timeout() bool {
    // ...
}

func (deadlineExceededError) Temporary() bool{
    // ...
}
```

空上下文

`emptyCtx`

- 空上下文，私有、无超时、不可取消、无数据
- 本质上为一个整型

- `Deadline()` 方法返回公元元年时间以及 `false` 的 `flag`
- `Done()` 方法返回一个 `nil` 值
- `Err()` 和 `Value()` 方法返回 `nil`

```go
type emptyCtx int

// 几个方法的实现
func (*emptyCtx) Deadline() (deadline time.Time, ok bool) {
    return
}

func (*emptyCtx) Done() <-chan struct{} {
    return nil
}

func (*emptyCtx) Err() error {
    return nil
}

func (*emptyCtx) Value(key any) any {
    return
}
```

`context.Background()` 与 `context.TODO()`

- 它们均是 `emptyCtx` **类型**

```go
var (
    background = new(emptyCtx)
    todo = new(emptyCtx)
)

func Background() Context {
    return background
}

func TODO() Context {
    return todo
}
```

可取消的上下文

`cancelCtx`

- 内嵌了一个 `context` 作为其父 `context`，因此 `cancelCtx` 必然为某个 `context` 的子 `context`

- 含有一把锁，用以协调并发场景下的资源获取
- `done`：存放的实际类型为 `chan struct{}`
- `children`：一个 `set`，指向 `cancelCtx` 的所有子上下文
- `err`：记录了当前 `cancelCtx` 的错误
- 并未直接实现 `Deadline()` 方法，直接使用会报错

```
type cancelCtx struct{
    Context

    mu sync.Mutex
    done atomic.Value
    children map[cancleler]struct{}
    err error
}
```

`canceler`

- 本质是一个接口
- 只有具备**可取消能力**的 `ctx` 才需要实现它

```
type canceler interface {
    cancel(removeFromParent bool, err error)
    Done() <-chan struct{}
}
```

`cancelCtx.Done()` 方法

- 如果创建过 `chan`，直接通过 `atomic` 获取
- 否则加锁进行创建，创建完之后还需要再检查 `chan` 是否被其它协程创建了

```
func (c *cancelCtx) Done() <-chan struct{} {
    d := c.done.Load()
    if d != nil {
        return d.(chan struct{})
    }
    c.mu.Lock()
    defer c.mu.Unlock()
    d = c.done.Load()
    if d == nil {
        d = make(chan struct{})
        c.done.Store(d)
    }
    return d.(chan struct{})
}
```

`cancelCtx.Err()` 方法

- 加锁
- 读取 `cancelCtx.err`
- 解锁
- 返回结果

```
func (c *cancelCtx) Err() error{
    c.mu.Lock()
    err := c.err
    c.mu.Unlock()
    return err
}
```

`cancelCtx.Value()` 方法

- 如果 `key` 是 `&cancelCtxKey` ，返回 `cancelCtx` 自身指针
- 否则按照 `valueCtx` 的思路返回：从父节点依次向上查找

```
func (c *cancelCtx) Value(key any) any {
    if key == &cancelCtxKey {
        return c
    }
    return value(c.Context, key)
}
```

`context.WithCancel()` 方法

- 创建一个可以取消的 `ctx`
- 逻辑
  - 校验父 `ctx` 非空
  - 注入父 `ctx` ，构造一个新的 `cancelCtx`
  - 启动一个守护协程，保证父 `ctx` 终止时， `cancelCtx` 也会被终止
  - 返回 `cancelCtx` 和一个可用于终止的闭包函数

```
func WithCancel(parent Context) (ctx,context,cancel CanecelFunc){
    if parent == nil{
        panic("cannot create context from nil parent")
    }
    c := newCancelCtx(parent)
    // 启动守护协程
    propagateCancel(parent,&c)
    return c.func(){c.cancel(true,Canceld)}
}
```

`newCancelCtx()` 方法

- 直接进行构造，只声明了父 `ctx`

```
func newCancelCtx(parent Context) cancelCtx {
    return cancelCtx{Context: parent}
}
```

`propagateCancel()` 方法

- 用于传递父 `context` 和 `context` 之间的 `cancel` 事件
- 老版本
  - 先判断父 `context` 是否为空，再判断父 `context` 是否已经死亡，接着分两种情况讨论
    - 父 `context` 是标准库的 `cancelCtx` ，直接把子 `context` 加入到父 `context` 的内部map中，无需开启新的协程
    - 父 `context` 是自定义的 `context` ， `Go` 无法访问第三方 `context` 的map，因此会开启一个协程监听父 `context` 的 `Done()` ，当父节点死亡，协程立刻执行子节点的 `cancel()` 方法

```go
func propagateCancel(parent Context,child canceler){
    done := parent.Done()
    if done == nil{
        return
    }
    select{
    case <- done:
        // 取消子ctx
        child.cancel(false,parent.Err())
        return

    default:
    }

    if p,ok := parentCancelCtx(parent);ok{
        p.mu.Lock()
        if p.err != nil{
            // 父节点已死
            child.cancel(false,p.err)
        }else{
            if p.children == nil{
                p.children = make(map[canceler]struct{})
            }
            p.childeren[child] = struct{}{}
        }
        p.mu.Unlock()
    }else{
        // 内部原子计数器
        atomic.AddInt32(&goroutines, +1)
        // 开一个协程监听父节点的Done()
        go func() {
            select {
            case <-parent.Done():
                child.cancel(false, parent.Err())
            case <-child.Done():
            }
        }()
    }
}
```

- 新版本
    - 无需开启新的协程监听父节点的`Done()`，通过`AfterFunc`在子节点注册一个回调函数，实现父节点的**死后回调**，自动把子节点也取消掉
    - 如果子节点先被取消，使用`stopCtx`的`stop()`函数**注销**刚才在父节点那注册的回调，防止内存泄漏

- 无需启动协程，只需要注册回调函数和建立 `stopCtx` 结构体

```
func (c *cancelCtx) propagateCancel(parent Context, child canceler) {
    c.Context = parent

    done := parent.Done()
    if done == nil {
        return // parent is never canceled
    }

    select {
    case <-done:
        // parent is already canceled
        child.cancel(false, parent.Err(), Cause(parent))
        return
    default:
    }

    if p, ok := parentCancelCtx(parent); ok {
        // parent is a *cancelCtx, or derives from one.
        p.mu.Lock()
        if err := p.err.Load(); err != nil {
            // parent has already been canceled
            child.cancel(false, err.(error), p.cause)
        } else {
            if p.children == nil {
                p.children = make(map[canceler]struct{})
            }
            p.children[child] = struct{}{}
        }
        p.mu.Unlock()
        return
    }

    if a, ok := parent.(afterFuncer); ok {
        // parent implements an AfterFunc method.
        c.mu.Lock()
        stop := a.AfterFunc(func() {
            child.cancel(false, parent.Err(), Cause(parent))
        })
        c.Context = stopCtx{
            Context: parent,
            stop:    stop,
        }
        c.mu.Unlock()
        return
    }

```

```
    goroutines.Add(1)
    go func() {
        select {
        case <-parent.Done():
            child.cancel(false, parent.Err(), Cause(parent))
        case <-child.Done():
        }
    }()
}
```

`parentCancelCtx()` 方法

- 找到离当前 `ctx` **最近的、真正可取消的父** `ctx` ，并获取父 `ctx` 和它对应的 `key`
- 方便形成取消传播链
- 工作流程
  - 获取父 `ctx` 的channel，判断父 `ctx` 的 `chan` 是否已经被取消
  - 获取父 `ctx` 并进行类型断言
  - 比较 `cancelCtx` 自己维护的 `chan` 和 `parent` 维护的 `chan` 是否一致

```
func parentCance;Ctx(parent Context) (*cancelCtx,bool) {
    done := parent.Done()
    if done == closedchan || done == nil {
        return nil,false
    }
    // 获取父ctx并进行类型断言
    p,ok := parent.Value(&cancelCtxKey).(*cancelCtx)
    if !ok {
        return nil,false
    }
    pdone, _ := p.done.Load().(chan struct{})
    if pdone != done {
        return nil, false
    }
    return p, true
}
```

`cancelCtx.cancel()` 方法

- `removeFromParent` 表示当前 `ctx` 是否需要从父 `ctx` 的 `children set` 中删除
- 工作流程
  - 确定 `err`
  - 加锁保证并发安全
  - 关闭 `Done()` 通道，有两种情况
    - 这个 `ctx` 的 `Done()` 没有协程去触发，是空的，因此存 `closedchan`
    - `Done()` 不为空，关闭这个通道
  - `cancel` 掉所有子 `ctx`

```go
func (c *cancelCtx) cancel(removeFromParent bool,err error) {
    if err == nil{
        panic("context: internal error: missing cancel error")
    }
    c.mu.Lock()
    if c.err != nil {
        c.mu.Unlock()
        return // already canceled
    }
    d, _ := c.done.Load().(chan struct{})
    if d == nil {
        c.done.Store(closedchan)
    }else {
        close(d)
    }
    for child := range c.children {
        child.cancel(false,err)
    }
    c.children = nil
    c.mu.Unlock()
    if removeFromParent {
        removeChild(c.Context, c)
    }
}
```

`removeChild()` 方法

- 将当前 `ctx` 从 `parent` 的 `children set` 中移除

```go
func removeChild(parent Context, child canceler) {
    p, ok := parentCancelCtx(parent)
    if !ok {
        return
    }
    p.mu.Lock()
    if p.children != nil {
        delete(p.children, child)
    }
    p.mu.Unlock()
}
```

超时上下文

`timerCtx`

- 在 `cancelCtx` 基础上做了一层封装
- 新增 `timer` 用于定时终止 `context`
- 新增 `deadline` 用于表示过期时间

```
type timerCtx struct {
    cancelCtx
    timer *time.Timer // Under cancelCtx.mu.

    deadline time.Time
}
```

`timerCtx.Deadline()` 方法

- 用于展示过期时间

```
func (c *timerCtx) Deadline() (deadline time.Time ok bool){
    return c.deadline,true
}
```

`timerCtx.cancel()` 方法

- 复用了 `cancelCtx` 的 `cancel` 能力

携带值的上下文

`valueCtx`

- 一个 `valueCtx` 中仅有**一组kv对**

```
type valueCtx struct {
    Context
    key, val any
}
```

`valueCtx.Value()` 方法

- 传入的 `key` 是用于传入的 `key` ，直接返回 `value`
- 否则从父 `context` 依次向上查找
- 不适合存大量的 `kv` 数据
  - 不支持基于 `key` 去重
  - 基于 `key` 找 `val` 的过程是 `O(N)` 的时间复杂度

```go
func (c *valueCtx) Value(key any) any {
    if c.key == key {
        return c.val
    }
    return value(c.Context, key)
}
```

`context.WithValue()` 方法

- 创建一个可以携带 `kv` 值的上下文

```go
func WithValue(parent Context,key,val any) Context{
    if parent == nil{
        panic("cannot create context from nil parent")
    }
    if key == nil{
        panic("nil key")
    }
    // 若key的类型无法比较
    if !reflectlite.TypeOf(key).Comparable(){
        panic("key is not comparable")
    }
    return &valuectx{parent,key,val}
}
```

sync

`sync.Mutex`

- **互斥锁**
- 有两种模式，如果一个协程等锁**超过1ms还未拿到锁**，就会从正常模式切换到饥饿模式
  - **正常模式**（默认）：释放锁时，唤醒的协程和新来的协程一起抢锁，**不公平**
  - **饥饿模式**：锁直接交给等待队列最前面的那个协程，**公平**
- **同一个协程不能重复加锁**，否则会死锁
- 传递的时候需要传递**指针**

常见用法

- `Mutex.Lock()`：加锁
- `Mutex.Unlock()`：解锁

```go
var mu sync.Mutex
var count int
func add(){
    mu.Lock()
    defer mu.Unlock()
    count ++
}
```

`sync.RWMutex`

- **读写锁**
- 适用于读多写少的场景
- 写操作的优先级高，当有协程申请写锁，后续新来的读请求就**会被阻塞**直到写操作完成

常见用法

- `RWMutex.RLock()`：加读锁
- `RWMutex.RUnlock()`：解读锁
- `RWMutex.Lock()`：加写锁
- `RWMutex.Unlock()`：解写锁

```go
var rw sync.RWMutex

func read(){
    rw.RLock()
    defer rw.RUnlock()
    // ...
}

func write(){
    rw.Lock()
    defer rw.Unlock()
}
```

`sync.Once`

- **单例模式，只执行一次**
- 常用于初始化

常见用法

- `Once.Do(func)`：只执行一次 `func`

```go
var once sync.Once
func Init(){
    once.Do(func{
        // ...
    })
}
```

原理

- `Once` 字段
  - 一个 `done` 标志位：表示是否已经执行过
  - 一把互斥锁：保证只有一个协程真正去执行函数

```go
type Once struct{
    done uint32 // 原子标志：0=没执行过，1=执行过
    m Mutex     // 互斥锁
}
```

- `Once.Do(f)` 方法
  - 使用 `atomic.Load()` 方法检查 `done` ，如果 `done` 为1，返回
  - 否则看能不能拿锁
  - 拿到锁之后再检查一次 `done` ，如果其为1，返回
  - 否则执行函数并使用 `atomic.Store()` 方法把 `done` 设置为1

`sync.WaitGtoup`

- **任务编排**，主协程等待一组子协程结束
- 不能在子协程里面使用 `Add()` 方法，否则主协程可能跑到 `Wait()` 的时候子协程还未启动，**直接结束任务**
- 调用 `Done()` 方法的次数不能超过在 `Add()` 设定的值，否则会 `panic`

常见用法

- `WaitGroup.Add(n)` ：等待n个子协程执行完毕，将未完成计数加n
- `WaitGroup.Done()` ：将计数减一
- `WaitGroup.Wait()` ：阻塞等待直到计数变为0

```go
var wg sync.WaitGroup

for i := 0; i < 3; i++ {
    wg.Add(1)
    go func(id int) {
        defer wg.Done() // 任务完成，计数减 1
        fmt.Println("Job", id)
    }(i)
}

wg.Wait() // 阻塞，直到计数归零
fmt.Println("All done")
```

`sync.Pool`

- 独立**保存和恢复临时对象**的集合，复用对象，减少内存重分配，降低GC压力

常见用法

- `New func() any`：生成全新对象
- `Get() any`：从池子获取一个对象
- `Put(x any)`：将使用完毕的对象 放回池子中，供后续复用

原理

- 为GMP的P分配一个专属本地池 `poolLocal`，可分为
  - `private`：私有对象，每个P只能存一个私有对象，当前P访问自己的 `private` 是绝对无锁的
  - `shared`：共享队列，一个无锁的双向环形链表，**当前P可以把多余对象塞进这里，其它P也可以从这里获取对象**

`Get() any`

- 将当前协程绑定在当前P上，防止被调度器强行切到别的P上
- 从 `private` 拿，无锁返回，速度极快
- 若**私有对象**为空，从本地共享对象拿
- 若**本地共享对象**为空，从其它P中拿对象
- 若其它P的共享对象也为空，从上一轮GC中幸存下来的受害者缓存（**victim**）找
- 最后没办法了，只能现场调用 `New` 编造新对象

`Put(x any)`

- 当前协程绑定当前P
- 放入 `private`
- 若 `private` 已满，放入 `shared`
- 解绑

受害者机制

- `Pool` 池子内部由 `local` 和 `victim` 组成
- 当进行GC的时候，把 `local` 的对象全部转移到 `victim`，并清空 `local`
- 进行下一轮GC时，**此时** `victim` **没被复活的对象就会被回收**

`sync.Map`

- 原生的Map并发读写会panic
- **并发Map**，适用于**写少读多**的场景，通用场景优先使用 `Mutex + map`

常见用法

- 存储：`Map.Store(key)`
- 查找：`Map.Load(key)`
- 删除：`Map.Delete(key)`
- 遍历：`Map.Range(func)`
  - 当 `func` 返回值为真则继续遍历
  - **不是强一致性快照**，遍历过程中可能被别的协程存储或者删除
- 原子性的加载或存储：`Map.LoadOrStore(key,value)`
  - 先尝试加载 `key` 对应的值，如果 `key` 不存在，则存储给定的 `value` ，然后返回实际的值
  - 操作是**原子性的**，保证在多个协程同时调用的情况下只有一个会成功**创建**

原理

- **结构**
  - `read` ：一个只读 `map` ，读操作查这个
  - `dirty` ：一个需要加锁的 `map` ，写操作在这里进行
  - `misses` ：读不到时的**未命中计数**，用来决定何时把 `dirty` 升级成 `read`
- **查找的流程**
  - 先去 `read` 查
  - 如果没找到，可能在 `dirty` 中，加锁去 `dirty` 查的同时 `misses` ++
  - 当 `misses` 太多，说明大量 `key` 都在 `dirty` ，每次查询都要加锁，速率太慢，此时触发升级，把 `dirty` 整体升级成新的 `read`
- **存储的流程**
  - 加锁更新 `dirty`
  - 写太多会对 `dirty` 进行升级
- **删除的流程**
  - 主要在 `dirty/read` 的条目上做标记或删除

---

atomic

只保证单变量原子性，主要操作 `int32` 、 `int64` 、 `uint32` 、 `uint64` 、 `uintptr` 、 `Pointer` 类型

常见API

`atomic.AddInt64(AddInt32)(&num,n)`

- 原子地给 `num` 加 `n`

`atomic.StoreInt32/StoreInt64(&status,n)`

- 原子地把变量 `status` 设置为 `n`

`atomic.LoadInt32/LoadInt64(&cnt)`

- 原子地读取变量值

`atomic.SwapInt32/SwapInt64(&x,new)`

- 写入新值 `new` 到 `x` 中，同时返回 `x` 的旧值

`atomic.CompareAndSwapInt32/Int64(&x,old,new)`

- **CAS：原子比较并交换**
- 只有当 `x` 等于 `old` 时，才把它改成 `new` ，整个过程原子性完成
- 返回 `bool`
  - `true` ： 交换成功
  - `false` ： 交换失败
- 单次调用可能失败，要么失败就返回，要么死循环直到成功

`atomic.Value`

- 可以放任意类型的盒子，协程可以在不加锁的情况下随时读到最新放进去的值
- 两个方法
  - `Store(v any)` ： 放一个值进去
  - `Load() any` ： 取出当前值
- 注意
  - 第一次用之前要先 `Store`
  - `Store` 的具体类型**自始至终要保持一致**
  - 要修改里面存储的值时，**绝对不能修改**读出来的对象。必须**深拷贝**一份新的，修改新的，然后 Store 进去

框架

Gin