

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 Web开发与数据库实战

- 1 HTTP编程
- 2 Client客户端
- 3 模板
- 4 Mysql
- 5 MySQL预处理
- 6 Go实现MySQL事务
- 7 sqlx使用

1 HTTP编程

- a. Go原生支持http, `import("net/http")`
- b. Go的http服务性能和nginx比较接近
- c. 几行代码就可以实现一个web服务



1.1 HTTP常见请求方法

5. http常见请求方法

1) **Get**请求

2) **Post**请求

3) Put请求

4) Delete请求

5) Head请求





1.2 http 常见状态码

http 常见状态码

`http.StatusContinue = 100`

`http.StatusOK = 200`

`http.StatusFound = 302`

`http.StatusBadRequest = 400`

`http.StatusUnauthorized = 401`

`http.StatusForbidden = 403`

`http.StatusNotFound = 404`

`http.StatusInternalServerError = 500`



2 Client客户端

http包提供了很多访问Web服务器的函数，比如http.Get()、http.Post()、http.Head()等，读到的响应报文数据被保存在 Response 结构体中。

Response结构体的定义

```
type Response struct {  
    Status string // e.g. "200 OK"  
    StatusCode int // e.g. 200  
    Proto string // e.g. "HTTP/1.0"  
    ProtoMajor int // e.g. 1  
    ProtoMinor int // e.g. 0  
    Header Header  
    Body io.ReadCloser  
    // ...  
}
```

服务器发送的响应包体被保存在Body中。可以使用它提供的Read方法来获取数据内容。保存至切片缓冲区中，拼接成一个完整的字符串来查看。

结束的时候，需要调用Body中的Close()方法关闭io。



2.1 基本的HTTP/HTTPS请求

Get、Head、Post和PostForm函数发出HTTP/HTTPS请求

```
resp, err := http.Get("http://example.com/")
...
resp, err := http.Post("http://example.com/upload", "image/jpeg", &buf)
...
resp, err := http.PostForm("http://example.com/form", url.Values{"key":
{"Value"}, "id": {"123"}})
```

使用完response后必须关闭回复的主体

```
resp, err := http.Get("http://example.com/")
if err != nil {
    // handle error
}
defer resp.Body.Close()
body, err := ioutil.ReadAll(resp.Body)
// ...
```



2.2.1 不带参数的Get方法示例

```
resp, err := http.Get("https://www.baidu.com/")
if err != nil {
    fmt.Println("get err:", err)
    return
}
defer resp.Body.Close()

data, err := ioutil.ReadAll(resp.Body)
if err != nil {
    fmt.Println("get data err:", err)
    return
}

fmt.Println(string(data))
```

2-2-1-http-get-client.go



2.2.2 带参数的Get方法示例

GET请求的参数需要使用Go语言内置的net/url这个标准库来处理

```
//1.处理请求参数
params := url.Values{}
params.Set("name", "darren")
params.Set("hobby", "足球")

//2.设置请求URL
rawUrl := "http://127.0.0.1:9000"
reqURL, err := url.ParseRequestURI(rawUrl)
if err != nil {
    fmt.Printf("url.ParseRequestURI()函数执行错误,错误为:%v\n", err)
    return
}

//3.整合请求URL和参数
//Encode方法将请求参数编码为url编码格式("bar=baz&foo=quux"), 编码时会以键进行排序。
reqURL.RawQuery = params.Encode()

//4.发送HTTP请求
//说明: reqURL.String() String将URL重构为一个合法URL字符串。
fmt.Println("Get url:", reqURL.String())
resp, err := http.Get(reqURL.String())
```

2-2-2-http-get-client.go

2-2-2-http-get-server.go



2.3.1 post方法

发送POST请求的示例代码

```
url := "http://127.0.0.1:9000/post"
contentType := "application/json"
data := `{"name":"darren","age":18}`
resp, err := http.Post(url, contentType, strings.NewReader(data))
if err != nil {
    fmt.Println("post failed, err:%v\n", err)
    return
}
defer resp.Body.Close()
b, err := ioutil.ReadAll(resp.Body)
if err != nil {
    fmt.Println("get resp failed, err:%v\n", err)
    return
}
fmt.Println(string(b))
```

2-3-1-http-post-client.go

2-3-1-http-post-server.go



2.4 head方法-client

HEAD请求常常被忽略，但是能提供很多有用的信息，特别是在有限的速度和带宽下。主要有以下特点：

- 1、只请求资源的首部；
- 2、检查超链接的有效性；
- 3、检查网页是否被修改；
- 4、多用于自动搜索机器人获取网页的标志信息，获取rss种子信息，或者传递安全认证信息等

```
url := "http://www.baidu.com"
c := http.Client{
    Transport: &http.Transport{
        Dial: func(network, addr string) (net.Conn, error) {
            timeout := time.Second * 2
            return net.DialTimeout(network, addr, timeout)
        },
    },
}
resp, err := c.Head(url)
if err != nil {
    fmt.Printf("head %s failed, err:%v\n", url, err)
} else {
    fmt.Printf("%s head succ, status:%v\n", url, resp.Status)
}
```



2.5 表单处理

```
const form = `<html><body><form action="#" method="post" name="bar">
    <input type="text" name="in"/>
    <input type="text" name="in"/>
    <input type="submit" value="Submit"/>
</form></html></body>`
```

```
func FormServer(w http.ResponseWriter, request *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    switch request.Method {
    case "GET":
        io.WriteString(w, form)
    case "POST":
        request.ParseForm()
        fmt.Println("request.Form[in]:", request.Form["in"])
        io.WriteString(w, request.Form["in"][0])
        io.WriteString(w, "\n")
        io.WriteString(w, request.Form["in"][1])
    }
}
```

2-5-http-form-server.go



2.6 panic处理

```
http.HandleFunc("/test1", logPanics(SimpleServer))  
http.HandleFunc("/test2", logPanics(FormServer))
```

```
func logPanics(handle http.HandlerFunc) http.HandlerFunc {  
    return func(writer http.ResponseWriter, request *http.Request) {  
        defer func() {  
            if x := recover(); x != nil {  
                log.Printf("[%v] caught panic: %v", request.RemoteAddr, x)  
            }  
        }()  
        handle(writer, request)  
    }  
}
```

2-6-panic-server.go



3 模板

1) 替换 {{.字段名}}

```
var arr []Person
p := Person{Name: "Darren", Age: 18, Title: "我的个人网站"}
p1 := Person{Name: "King", Age: 19, Title: "我的个人网站"}
p2 := Person{Name: "柚子", Age: 20, Title: "我的个人网站"}
arr = append(arr, p)
arr = append(arr, p1)
arr = append(arr, p2)
```

| | | |
|--------|----|--------|
| Darren | 18 | 我的个人网站 |
| King | 19 | 我的个人网站 |
| 柚子 | 20 | 我的个人网站 |

3-1-template.go



3.1 模板-替换 {{. 字段名}}

```
var arr []Person
p := Person{Name: "Darren", Age: 18, Title: "我的个人网站"}
p1 := Person{Name: "King", Age: 19, Title: "我的个人网站"}
p2 := Person{Name: "柚子", Age: 20, Title: "我的个人网站"}
arr = append(arr, p)
arr = append(arr, p1)
arr = append(arr, p2)
```

| | | |
|--------|----|--------|
| Darren | 18 | 我的个人网站 |
| King | 19 | 我的个人网站 |
| 柚子 | 20 | 我的个人网站 |

3-1-template.go



4 Mysql

建库建表

在MySQL中创建一个名为go_test的数据库

```
CREATE DATABASE go_test;
```

进入该数据库:

```
use go_test;
```

创建一张用于测试的数据表:

```
CREATE TABLE `user` (  
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT,  
  `name` VARCHAR(20) DEFAULT '',  
  `age` INT(11) DEFAULT '0',  
  PRIMARY KEY(`id`)  
)ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT  
CHARSET=utf8mb4;
```



4.0 连接mysql

Open函数:

```
db, err := sql.Open("mysql", "用户名:密码@tcp(IP:端口)/数据库?charset=utf8")
```

例如: `db, err := sql.Open("mysql", "root:111111@tcp(127.0.0.1:3306)/test?charset=utf8")`



4-1 mysql插入数据

```
// 插入数据
func insertRowDemo(db *sql.DB) {
    sqlStr := "insert into user(name, age) values (?,?)"
    ret, err := db.Exec(sqlStr, "darren", 18)
    if err != nil {
        fmt.Printf("insert failed, err:%v\n", err)
        return
    }
    theID, err := ret.LastInsertId() // 新插入数据的id
    if err != nil {
        fmt.Printf("get lastinsert ID failed, err:%v\n", err)
        return
    }
    fmt.Printf("insert success, the id is %d.\n", theID)
}
```

4-1-mysql.go



4-2 mysql 查询-单行查询

单行查询

单行查询db.QueryRow()执行一次查询，并期望返回最多一行结果（即Row）。QueryRow总是返回非nil的值，直到返回值的Scan方法被调用时，才会返回被延迟的错误。（如：未找到结果）

```
func (db *DB) QueryRow(query string, args ...interface{}) *Row
```

```
// 查询单条数据示例
func queryRowDemo(db *sql.DB) {
    sqlStr := "select id, name, age from user where id=?"
    var u user
    // 非常重要：确保QueryRow之后调用Scan方法，否则持有的数据库链接不会被释放
    err := db.QueryRow(sqlStr, 1).Scan(&u.id, &u.name, &u.age)
    if err != nil {
        fmt.Printf("scan failed, err:%v\n", err)
        return
    }
    fmt.Printf("id:%d name:%s age:%d\n", u.id, u.name, u.age)
}
```

4-2-mysql-query copy.go



4-2 mysql 查询-多行查询

多行查询db.Query()执行一次查询，返回多行结果（即Rows），一般用于执行select命令。参数args表示query中的占位参数。

```
func (db *DB) Query(query string, args ...interface{}) (*Rows, error)
```

```
// 查询多条数据示例
func queryMultiRowDemo(db *sql.DB) {
    sqlStr := "select id, name, age from user where id > ?"
    rows, err := db.Query(sqlStr, 0)
    if err != nil {
        fmt.Printf("query failed, err:%v\n", err)
        return
    }
    // 非常重要：关闭rows释放持有的数据库链接
    defer rows.Close()

    // 循环读取结果集中的数据
    for rows.Next() {
        var u user
        err := rows.Scan(&u.id, &u.name, &u.age)
        if err != nil {
            fmt.Printf("scan failed, err:%v\n", err)
            return
        }
        fmt.Printf("id:%d name:%s age:%d\n", u.id, u.name, u.age)
    }
}
```

4-2-mysql-multi-query.go



4-3 mysql更新

```
// 更新数据
func updateRowDemo(db *sql.DB) {
    sqlStr := "update user set age=? where id = ?"
    ret, err := db.Exec(sqlStr, 39, 1)
    if err != nil {
        fmt.Printf("update failed, err:%v\n", err)
        return
    }
    n, err := ret.RowsAffected() // 操作影响的行数
    if err != nil {
        fmt.Printf("get RowsAffected failed, err:%v\n", err)
        return
    }
    fmt.Printf("update success, affected rows:%d\n", n)
}
```

4-3-mysql-update.go



4-4 mysql 删除

```
// 删除数据
func deleteRowDemo(db *sql.DB) {
    sqlStr := "delete from user where id = ?"
    ret, err := db.Exec(sqlStr, 1)
    if err != nil {
        fmt.Printf("delete failed, err:%v\n", err)
        return
    }
    n, err := ret.RowsAffected() // 操作影响的行数
    if err != nil {
        fmt.Printf("get RowsAffected failed, err:%v\n", err)
        return
    }
    fmt.Printf("delete success, affected rows:%d\n", n)
}
```

4-4-mysql-delete.go



5 MySQL预处理

什么是预处理？

普通SQL语句执行过程：

- 1.客户端对SQL语句进行占位符替换得到完整的SQL语句。
- 2.客户端发送完整SQL语句到MySQL服务端
- 3.MySQL服务端执行完整的SQL语句并将结果返回给客户端。

预处理执行过程：

- 1.把SQL语句分成两部分，命令部分与数据部分。
- 2.先把命令部分发送给MySQL服务端，MySQL服务端进行SQL预处理。
- 3.然后把数据部分发送给MySQL服务端，MySQL服务端对SQL语句进行占位符替换。
- 4.MySQL服务端执行完整的SQL语句并将结果返回给客户端。

为什么要预处理？

- 1.优化MySQL服务器重复执行SQL的方法，可以提升服务器性能，提前让服务器编译，一次编译多次执行，节省后续编译的成本。
- 2.避免SQL注入问题。



5.1 Go实现MySQL预处理

`func (db *DB) Prepare(query string) (*Stmt, error)`

Prepare方法会先将sql语句发送给MySQL服务端，返回一个准备好的状态用于之后的查询和命令。
返回值可以同时执行多个查询和命令。

4-5-mysql-prepare.go

```
// 预处理查询示例
func prepareQueryDemo(db *sql.DB) {
    sqlStr := "select id, name, age from user where id > ?"
    stmt, err := db.Prepare(sqlStr)
    if err != nil {
        fmt.Printf("prepare failed, err:%v\n", err)
        return
    }
    defer stmt.Close()
    rows, err := stmt.Query(0)
    if err != nil {
        fmt.Printf("query failed, err:%v\n", err)
        return
    }
    defer rows.Close()
    // 循环读取结果集中的数据
    for rows.Next() {
        var u user
        err := rows.Scan(&u.id, &u.name, &u.age)
        if err != nil {
            fmt.Printf("scan failed, err:%v\n", err)
            return
        }
        fmt.Printf("id:%d name:%s age:%d\n", u.id, u.name, u.age)
    }
}
```

```
// 预处理插入示例
// 插入、更新和删除操作的预处理十分类似
func prepareInsertDemo(db *sql.DB) {
    sqlStr := "insert into user(name, age) values (?,?)"
    stmt, err := db.Prepare(sqlStr)
    if err != nil {
        fmt.Printf("prepare failed, err:%v\n", err)
        return
    }
    defer stmt.Close()
    _, err = stmt.Exec("darren", 18)
    if err != nil {
        fmt.Printf("insert failed, err:%v\n", err)
        return
    }
    _, err = stmt.Exec("柚子老师", 18)
    if err != nil {
        fmt.Printf("insert failed, err:%v\n", err)
        return
    }
    fmt.Println("insert success.")
}
```



6 Go实现MySQL事务

事务相关方法 Go语言中使用以下三个方法实现MySQL中的事务操作。

开始事务: `func (db *DB) Begin() (*Tx, error)`

提交事务: `func (tx *Tx) Commit() error`

回滚事务: `func (tx *Tx) Rollback() error`

6-mysql-transaction.go



7 sqlx使用

第三方库sqlx能够简化操作，提高开发效率。

安装

[go get github.com/jmoiron/sqlx](https://github.com/jmoiron/sqlx)

7-mysql-sqlx.go

7-mysql-sqlx-2.go



8 gin + mysql rest full api

gin_restful



8.1 gin + mysql rest full api - 增

http://192.168.2.132:8806/api/v1/users/add

POST

http://192.168.2.132:8806/api/v1/users/add?name=darren&telephone=18570368134

Params

Authorization

Headers (10)

Body

Pre-request Script

Tests

Settings

Query Params

| | KEY | VALUE | DESCRIPTION |
|-------------------------------------|-----------|-------------|-------------|
| <input checked="" type="checkbox"/> | name | darren | |
| <input checked="" type="checkbox"/> | telephone | 18570368134 | |
| | Key | Value | Description |

Body

Cookies

Headers (3)

Test Results

Status: 200 OK

Time: 223 ms

Pretty

Raw

Preview

Visualize

JSON

```
1 {
2   "msg": "insert successful 13"
3 }
```



8.2 gin + mysql rest full api - 改

http://192.168.2.132:8806/api/v1/users/update

POST http://192.168.2.132:8806/api/v1/users/update?id=13&telephone=18888888888

Params ● Authorization Headers (10) Body ● Pre-request Script Tests Settings

Query Params

| | KEY | VALUE | |
|-------------------------------------|-----------|-------------|--|
| <input checked="" type="checkbox"/> | id | 13 | |
| <input checked="" type="checkbox"/> | telephone | 18888888888 | |
| | Key | Value | |

Body Cookies Headers (3) Test Results

Status: 200 OK

Pretty Raw Preview Visualize JSON

```
1 {  
2   "msg": "updated successful 1"  
3 }
```



8.3 gin + mysql rest full api - 查

http://192.168.2.132:8806/api/v1/users/get/5

The screenshot shows a REST client interface with a GET request to `http://192.168.2.132:8806/api/v1/users/get/14`. The 'Params' tab is selected, showing a single query parameter: 'Key' with value 'Value'. Below this, the 'Body' tab is selected, displaying the JSON response in 'Pretty' format. The response is a nested object with a 'result' field containing user details.

| KEY | VALUE |
|-----|-------|
| Key | Value |

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "result": {
3     "id": 14,
4     "name": "king",
5     "telephone": "123456778"
6   }
7 }
```



8.4 gin + mysql rest full api - 获取所有

http://192.168.2.132:8806/api/v1/users

GET

http://192.168.2.132:8806/api/v1/users

Params

Authorization

Headers (10)

Body ●

Pre-request Script

Tests

Settings

Query Params

| KEY | VALUE |
|-----|-------|
| Key | Value |

Body

Cookies

Headers (3)

Test Results

⌚

Status:

Pretty

Raw

Preview

Visualize

JSON

⌵

```
1 {
2   "list": [
3     {
4       "id": 13,
5       "name": "darren",
6       "telephone": "18888888888"
7     },
8     {
9       "id": 14,
10      "name": "king",
11      "telephone": "123456778"
12    }
13  ]
14 }
```



8.5 gin + mysql rest full api - 删除


POST http://192.168.2.132:8806/api/v1/users/del?id=14

Params ● Authorization Headers (10) Body ● Pre-request Script Test

Query Params

| | KEY | VALUE |
|-------------------------------------|-----|-------|
| <input checked="" type="checkbox"/> | id | 14 |
| | Key | Value |

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize JSON 

```
1 {  
2   "msg": "delete successful 1"  
3 }
```

