

# 极客时间Go初级工程师第三课 错误处理与简单路由树实现

大明



# 目录

1. 错误处理
2. AOP设计——责任链模式
3. sync 包
4. 路由树设计与实现

# Http Server —— Server 启动快速失败

```
func main() {  
    server := web.NewSdkHttpServer(name: "my-test-server")  
  
    // 注册路由  
    server.Route(method: "POST", pattern: "/user/create", demo.SignUp)  
  
    if err := server.Start(address: ":8080"); err != nil {  
        // 快速失败，因为服务器都没启动成功，啥也做不了  
        panic(err)  
    }  
  
    // 假设我们后面还有很多动作  
}
```



# 基础语法 —— 错误处理

- error: 一般用于表达可以被处理的错误
- error只是一个内置的接口
- panic: 一般用于表达非常严重不可恢复的错误

```
// The error built-in interface type is th  
// representing an error condition, with t  
type error interface {  
    Error() string  
}
```

```
type MyError struct {  
  
}  
  
func (m *MyError) Error() string {  
    return "Hello, it's my error"  
}
```

# 基础语法 —— errors 包

- New 创建一个新的 error
- Is 判断是不是特定的某个 error
- As 类型转换为特定的 error
- Unwrap 解除包装，返回被包装的 error

```
func ErrorsPkg() {  
    errors.  
}  
  
f errors.New(text string) errors  
f errors.Is(err error, target error) errors  
f errors.As(err error, target interface{}) error  
f errors.Unwrap(err error) error  
p  
par
```

# 基础语法 —— errors 包

- New 创建一个新的 error
- Is 判断是不是特定的某个 error
- As 类型转换为特定的 error
- Unwrap 解除包装，返回被包装的 error

```
func ErrorsPkg() {  
    err := &MyError{}  
    // 使用 %w 占位符，返回的是一个新错误  
    // wrappedErr 是一个新类型，fmt.wrapError  
    wrappedErr := fmt.Errorf(format: "this is an wrapped error %w", err)  
  
    // 再解出来  
    if err == errors.Unwrap(wrappedErr) {  
        fmt.Println(a...: "unwrapped")  
    }  
  
    if errors.Is(wrappedErr, err) {  
        // 虽然被包了一下，但是 Is 会逐层解除包装，判断是不是该错误  
        fmt.Println(a...: "wrapped is err")  
    }  
  
    copyErr := &MyError{}  
    // 这里尝试将 wrappedErr 转换为 MyError  
    // 注意我们使用了两次的取地址符号  
    if errors.As(wrappedErr, &copyErr) {  
        fmt.Println(a...: "convert error")  
    }  
}
```

# 基础语法 —— error 和 panic 选用哪个？

遇事不决选 `error`

当你怀疑可以用 `error` 的时候，就说明你  
不需要 `panic`

一般情况下，只有快速失败的过程，才会考虑  
`panic`



# 基础语法 —— 从 panic 中恢复

某些时候，你可能需要从 `panic` 中恢复过来：

比如某个库，发生 `panic` 的场景是你不希望发生的场景。

这时候，你需要我们的 `recover`

```
// The recover built-in function allows a program to manage behavior of a
// panicking goroutine. Executing a call to recover inside a deferred
// function (but not any function called by it) stops the panicking sequence
// by restoring normal execution and retrieves the error value passed to the
// call of panic. If recover is called outside the deferred function it will
// not stop a panicking sequence. In this case, or when the goroutine is not
// panicking, or if the argument supplied to panic was nil, recover returns
// nil. Thus the return value from recover reports whether the goroutine is
// panicking.
func recover() interface{}
```

Tip: 如果你自己panic了，然后又要恢复过来，那么应该考虑  
不要用panic了



# 基础语法 —— 从 panic 中恢复

```
func main() {  
    defer func() {  
        if data := recover(); data != nil {  
            fmt.Printf(format: "hello, panic: %v\n", data)  
        }  
        fmt.Println(a...: "恢复之后从这里继续执行")  
    }()  
  
    panic(v: "Boom")  
    fmt.Println(a...: "这里将不会执行下来")  
}
```

# 基础语法 —— defer

```
func main() {  
    defer func() {  
        if data := recover(); data != nil {  
            fmt.Printf(format: "hello, panic: %v\n", data)  
        }  
        fmt.Println(a...: "恢复之后从这里继续执行")  
    }()  
  
    panic(v: "Boom")  
    fmt.Println(a...: "这里将不会执行下来")  
}
```

# Golang 语法 —— defer

- 用于在方法返回之前执行某些动作

- 像栈一样，**先进后出**

defer 语义接近 java 的 finally 块

所以我们经常使用 defer 来释放资源，例如释放锁

```
func main() {  
    defer func() {  
        fmt.Println(a...: "aaa")  
    }()  
  
    defer func() {  
        fmt.Println(a...: "bbb")  
    }()  
  
    defer func() {  
        fmt.Println(a...: "ccc")  
    }()  
}
```



# Go1ang 语法 —— 闭包

- 函数闭包：匿名函数 + 定义它的上下文
- 它可以访问定义之外的变量
- Go 很强大的特性，很常用

```
func main() {  
    i := 13  
    a := func() {  
        fmt.Printf(format: "i is %d \n", i)  
    }  
    a()  
  
    fmt.Println(ReturnClosure(name: "Tom")())  
}  
  
func ReturnClosure(name string) func() string {  
    return func() string {  
        return "Hello, " + name  
    }  
}
```

# Golang 语法 —— 闭包延时绑定

- 闭包里面使用的闭包外的参数，其值是在最终调用的时候确定下来的

```
func Delay() {  
    fns := make([]func(), 0, 10)  
    for i := 0; i < 10; i++ {  
        fns = append(fns, func() {  
            fmt.Printf(format: "hello, this is : %d \n", i)  
        })  
    }  
  
    for _, fn := range fns {  
        fn()  
    }  
}
```

# 要点总结

1. `error` 其实就是一个内置的普通的接口。`error` 相关的操作在 `errors` 包里面
2. `panic` 强调的是无可挽回了。但是也可以用 `recover` 恢复过来
3. 闭包是很强大的特性，但是要小心延时绑定

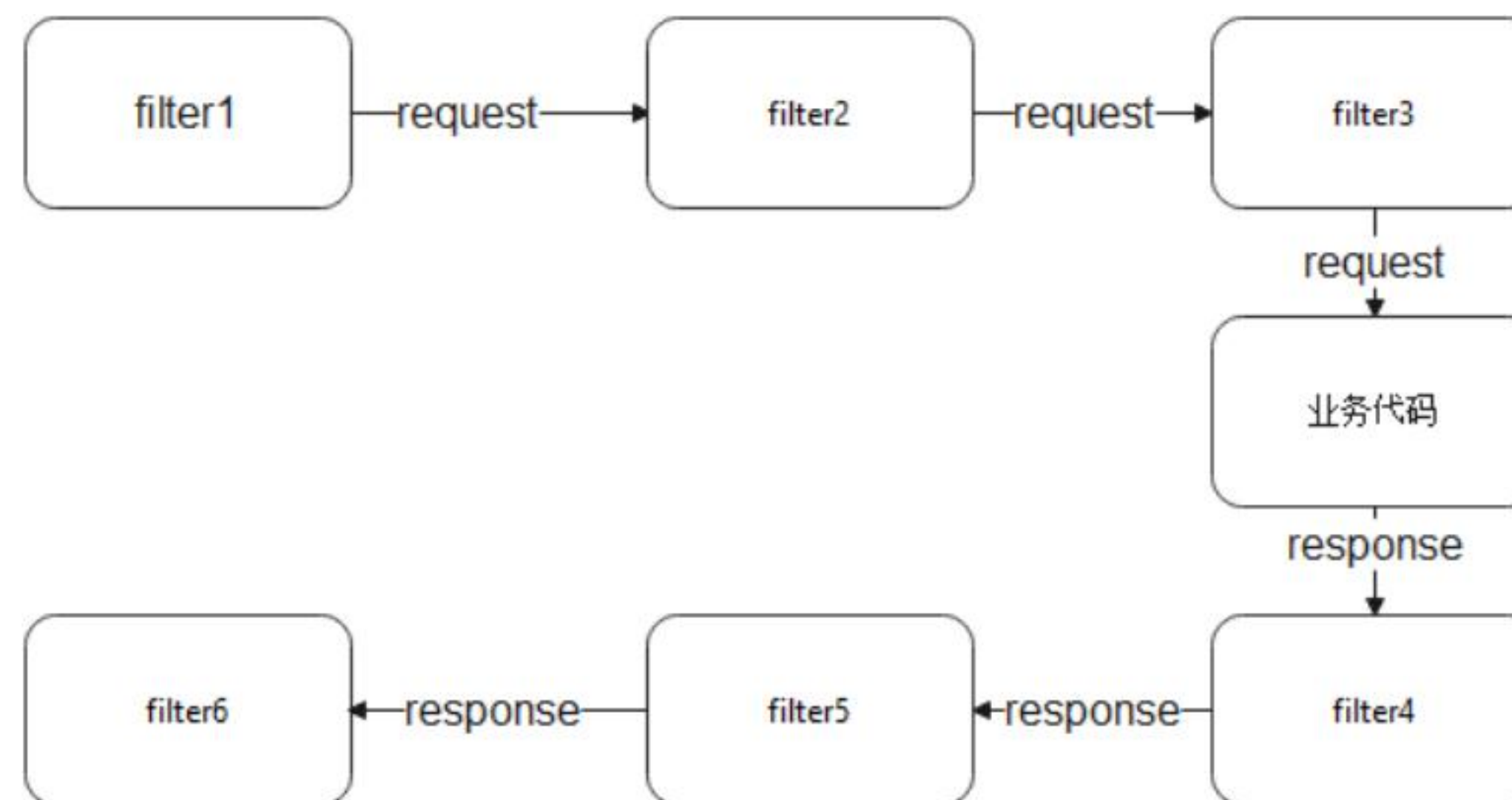


# 目录

1. 错误处理
2. AOP设计——责任链模式
3. sync 包
4. 路由树设计与实现

# Http Server —— AOP: 用闭包来实现责任链

- 为 server 支持一些 AOP 逻辑
- AOP: 横向关注点, 一般用于解决 Log, tracing, metric, 熔断, 限流等
- filter: 我们希望请求在真正被处理之前能够经过一大堆的 filter



# Http Server —— Filter 定义




```
type FilterBuilder func(next Filter) Filter
```

```
type Filter func(c *Context)
```



# Http Server —— Filter 定义



```
type FilterBuilder func(next Filter) Filter  
  
type Filter func(c *Context)
```

# Http Server —— metric filter

```
func MetricFilterBuilder(next Filter) Filter {  
    return func(c *Context) {  
        // 执行前的时间  
        startTime := time.Now().UnixNano()  
        next(c)  
        // 执行后的时间  
        endTime := time.Now().UnixNano()  
        fmt.Printf(format: "run time: %d \n", endTime-startTime)  
    }  
}
```

# Http Server —— 为什么这么定义 (Optional)

- 考虑我们 metric filter
- 在请求执行前，我记录时间戳
- 在请求执行后，我记录时间戳
- 两者相减就是执行时间
- 同时保证线程安全
- 是两个 filter 一前一后，那么要考虑线程安全问题
- 要考虑开始时间怎么传递给后一个 filter
- 如果是一个 filter，不采用这种方式，那么怎么把 filter 串起来？



# Http Server —— 集成 Filter

```
func NewSdkHttpServer(name string, builders ...FilterBuilder) Server {  
    handler := NewHandlerBasedOnMap()  
    // 因为我们是一个链，所以我们把最后的业务逻辑处理，也作为一环  
    var root Filter = func(c *Context) {  
        handler.ServeHTTP(c.W, c.R)  
    }  
    // 从后往前把filter串起来  
    for i := len(builders) - 1; i >= 0; i++ {  
        b := builders[i]  
        root = b(root)  
    }  
    res := &sdkHttpServer{  
        Name: name,  
        handler: handler,  
        root: root,  
    }  
    return res  
}
```

不定参数

勘误 i ++ 应该是 i--

实现细节后边大家捋一捋

```
func (s *sdkHttpServer) Start(address string) error {  
    http.HandleFunc(pattern: "/", func(writer http.ResponseWriter,  
        request *http.Request) {  
        c := NewContext(writer, request)  
        s.root(c)  
    })  
    return http.ListenAndServe(address, handler: nil)  
}
```

# Http Server —— 集成 Filter

```
func NewSdkHttpServer(name string, builders ...FilterBuilder) Server {  
  
    handler := NewHandlerBasedOnMap()  
    // 因为我们是一个链，所以把最后的业务逻辑处理，也作为一环  
    var root Filter = handler.ServeHTTP  
    // 从后往前把filter串起来  
    for i := len(builders) - 1; i >= 0; i-- {  
        b := builders[i]  
        root = b(root)  
    }  
    res := &sdkHttpServer{  
        Name: name,  
        handler: handler,  
        root: root,  
    }  
    return res  
}
```

```
type Handler interface {  
    ServeHTTP(c *Context)  
    Routable  
}
```

重新修改我们的Handler 接口



# 要点总结

1. 责任链是很常见的用于解决 AOP 的一种方式。
2. 类似的也叫做 middleware, interceptor... 本质是一样的
3. Go 函数是一等公民，所以可以考虑用闭包来实现责任链
4. filter 很常见，比如说鉴权，日志，tracing，以及跨域等都可以用 filter 来实现

# 目录

1. 错误处理
2. AOP设计——责任链模式
3. sync 包
4. 路由树设计与实现



# 基础语法 —— 线程安全的 Map

```
func main() {
    server := web.NewSdkHttpServer(name: "my-test-server",
        web.MetricFilterBuilder)

    // 我们都希望用户在启动server之前全部注册完路由,
    // 但是总防不住有些用户会随便啥时候注册路由
    server.Route(method: "POST", pattern: "/user/create", demo.SignUp)

    if err := server.Start(address: ":8080"); err != nil {
        // 快速失败, 因为服务器都没启动成功, 啥也做不了
        panic(err)
    }

    // 假设我们后面还有很多动作
}
```

```
type HandlerBasedOnMap struct {
    handlers map[string]func(c *Context)
}
```

线程不安全的map会导致程序panic

# 基础语法 —— sync.Map

- key 和 value 类型都是 interface{}。意味着你要搞各种类型断言

```
func main() {  
    m := sync.Map{}  
    m.  
}
```

Load(key interface{}) → \*Map

Range(f func(key interface{}, value interface{}))

Store(key interface{}, value interface{})

Delete(key interface{}) → \*Map

LoadAndDelete(key interface{}) → \*Map

LoadOrStore(key interface{}, value interface{})

```
func main() {  
    m := sync.Map{}  
    m.Store(key: "cat", value: "Tom")  
    m.Store(key: "mouse", value: "Jerry")  
  
    // 这里重新读取出来的，就是  
    val, ok := m.Load(key: "cat")  
    if ok {  
        fmt.Println(len(val.(string)))  
    }  
}
```

# 基础语法 —— 类型断言

- 形式: `t, ok := x.(T)` 或者 `t := x.(T)`
- T 可以是结构体或者指针
- 如何理解?
  - 即x是不是T。
  - 类似Java `instanceOf` + 强制类型转换合体
- 如果 x 是 nil, 那么永远是 false
- 编译器不会帮你检查

```
func main() {  
    m := sync.Map{}  
    m.Store(key: "cat", value: "Tom")  
    m.Store(key: "mouse", value: "Jerry")  
  
    // 这里重新读取出来的, 就是  
    val, ok := m.Load(key: "cat")  
    if ok {  
        fmt.Println(len(val.(string)))  
    }  
}
```

# 基础语法 —— 类型转换

- 形式:  $y := T(x)$
- 如何理解? 记住数字类型转换, string 和 []byte 互相转
  - 类似Java强制类型转换
- 编译器会进行类型检查, 不能转换的会编译错误

```
a := 12.0
b := int(a)
fmt.Println(b)
```

```
str := "Hello"
bytes := ([]byte)(str)
```



# 基础语法 —— sync.Mutex 和 sync.RWMutex

- sync 包提供了基本的并发工具
  - sync.Map: 并发安全 map
  - sync.Mutex: 锁
  - sync.RWMutex: 读写锁
  - sync.Once: 只执行一次
  - sync.WaitGroup: goroutine 之间同步

```
var mutex sync.Mutex
var rwMutex sync.RWMutex

func Mutex() {
    mutex.Lock()
    defer mutex.Unlock()
    // 你的代码
}

func RWMutex() {
    // 加读锁
    rwMutex.RLock()
    defer rwMutex.RUnlock()

    // 也可以加写锁
    rwMutex.Lock()
    defer rwMutex.Unlock()
}
```



# 基础语法 —— mutex家族注意事项

- 尽量用 RWMutex
- 尽量用 defer 来释放锁，防止 panic 没有释放锁
- 不可重入：lock 之后，即便是同一个线程 (goroutine)，也无法再次加锁（写递归函数要小心）
- 不可升级：加了读锁之后，如果试图加写锁，锁不升级

不可重入和不可升级，和很多语言的实现都是不同的，因此要小心使用

```
// 不可重入例子
func Failed1() {
    mutex.Lock()
    defer mutex.Unlock()

    // 这一句会死锁
    // 但是如果你只有一个goroutine，那么这一个会导致程序崩溃
    mutex.Lock()
    defer mutex.Unlock()
}
```

```
// 不可升级
func Failed2() {
    rwMutex.RLock()
    defer rwMutex.RUnlock()

    // 这一句会死锁
    // 但是如果你只有一个goroutine，那么这一个会导致程序崩溃
    mutex.Lock()
    defer mutex.Unlock()
}
```

# 基础语法 —— sync.Once

sync 包提供了基本的并发工具

- sync.Map: 并发安全 map
- sync.Mutex: 锁
- sync.RWMutex: 读写锁
- sync.Once: 只执行一次
- sync.WaitGroup: goroutine 之间同步

```
var once sync.Once

// 这个方法，不管调用几次，只会输出一次
func PrintOnce() {
    once.Do(func() {
        fmt.Println(a...: "只输出一次")
    })
}
```

# 基础语法 —— sync.WaitGroup

sync 包提供了基本的并发工具

- sync.Map: 并发安全 map
- sync.Mutex: 锁
- sync.RWMutex: 读写锁
- sync.Once: 只执行一次
- sync.WaitGroup: goroutine 之间同步

```
func main() {  
    res := 0  
    wg := sync.WaitGroup{}  
    wg.Add(delta: 10)           设置总数  
    for i := 0; i < 10; i++ {  
        go func(val int) {  
            res += val  
            wg.Done()           goroutine内完成任务-1  
        }(i)  
    }  
    // 把这个注释掉你会发现，什么结果你都可能拿到  
    wg.Wait()                  这里会阻塞，直到计数归0  
    fmt.Println(res)  
}
```



# Http Server —— sync.Map改造

```
type HandlerBasedOnMap struct {
    handlers sync.Map
}

func (h *HandlerBasedOnMap) ServeHTTP(c *Context) {
    request := c.R
    key := h.key(request.Method, request.URL.Path)
    handler, ok := h.handlers.Load(key)
    if !ok {
        c.W.WriteHeader(http.StatusNotFound)
        _, _ = c.W.Write([]byte("not any router match"))
        return
    }

    handler.(func(c *Context))(c)
}
```

```
func (h *HandlerBasedOnMap) Route(method string, pattern string,
    handlerFunc func(c *Context)) {
    key := h.key(method, pattern)
    h.handlers.Store(key, handlerFunc)
}
```

# Http Server —— 定义一个handlerFunc简化代码

```
type handlerFunc func(c *Context)
```

```
// Routable 可路由的
type Routable interface {
    // Route 设定一个路由，命中该路由的会执行handlerFunc的代码
    Route(method string, pattern string, handlerFunc handlerFunc)
}
```



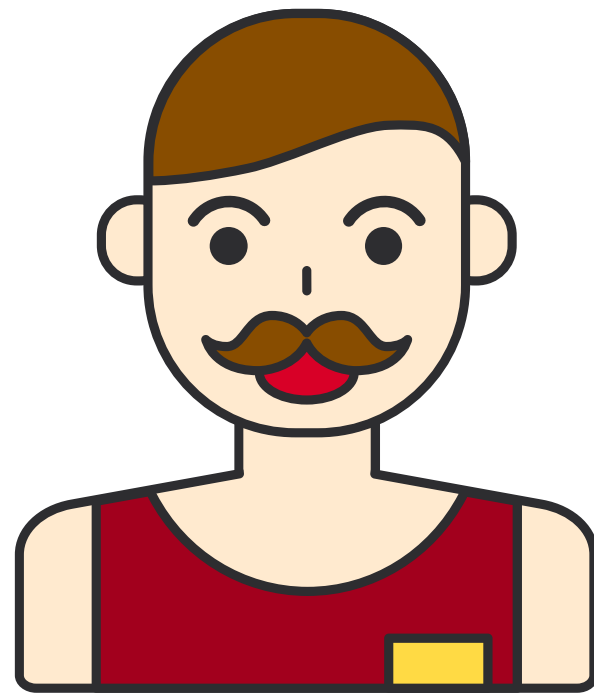
# 要点总结

1. 尽量用 `sync.RWMutex`
2. `sync.Once` 可以保证代码只会执行一次，一般用来解决一些初始化的需求
3. `sync.WaitGroup` 能用来在多个 `goroutine` 之间进行同步

# 目录

1. 错误处理
2. AOP设计——责任链模式
3. sync 包
4. 路由树设计与实现

# Http Server —— 路由树



我读书少，你别骗我。谁  
用 `map` 来搞路由？

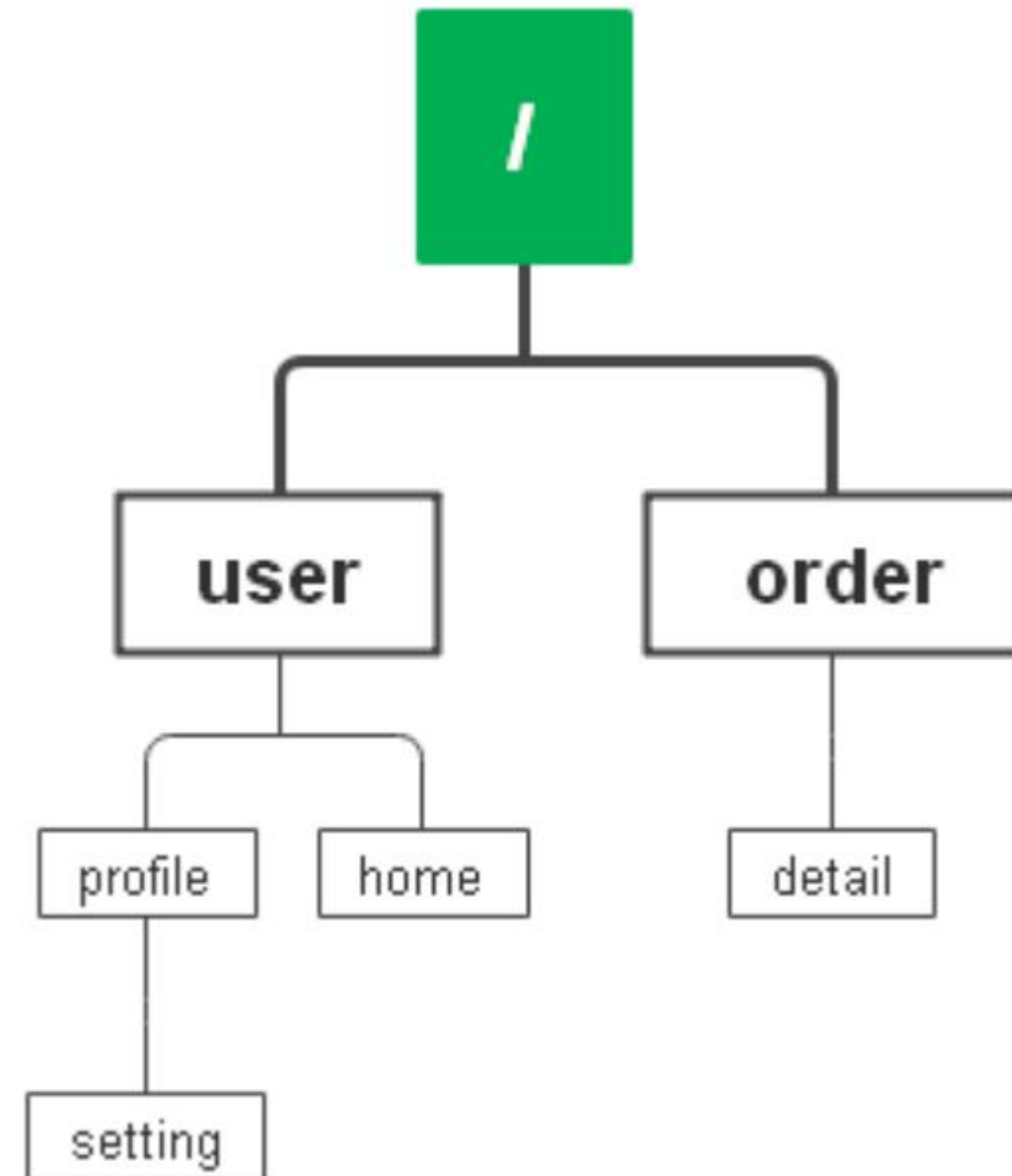
# Http Server —— 路由树

/user/profile/setting

/user/home

/order

/order/detail



# Http Server —— 路由树设计

- v1: 极简路由实现
- v2: 支持 \* 匹配
- v3: 支持复杂匹配



# Http Server —— 最简单的路由树

- 不考虑路径参数问题：例如/user/:id 等，只支持静态路由
- 不考虑 http method：例如暂时区分 GET /user 还是 POST /user
- 不考虑性能问题：这是为了排除干扰，减少代码的复杂度
- 不支持路由冲突

Tip: 任何框架的实现，最开始要抓住最核心的点，留出接口，后边再根据需要扩展各种接口。如果是自己练习，那么接口的向后兼容性都可以不考虑，这样就能专注在功能的核心支持上

# Http Server —— 树的定义

```
type HandlerBasedOnTree struct {  
    root *node  
}  
  
type node struct {  
    path string  
    children []*node  
  
    // 如果这是叶子节点,  
    // 那么匹配上之后就可以调用该方法  
    handler handlerFunc  
}
```

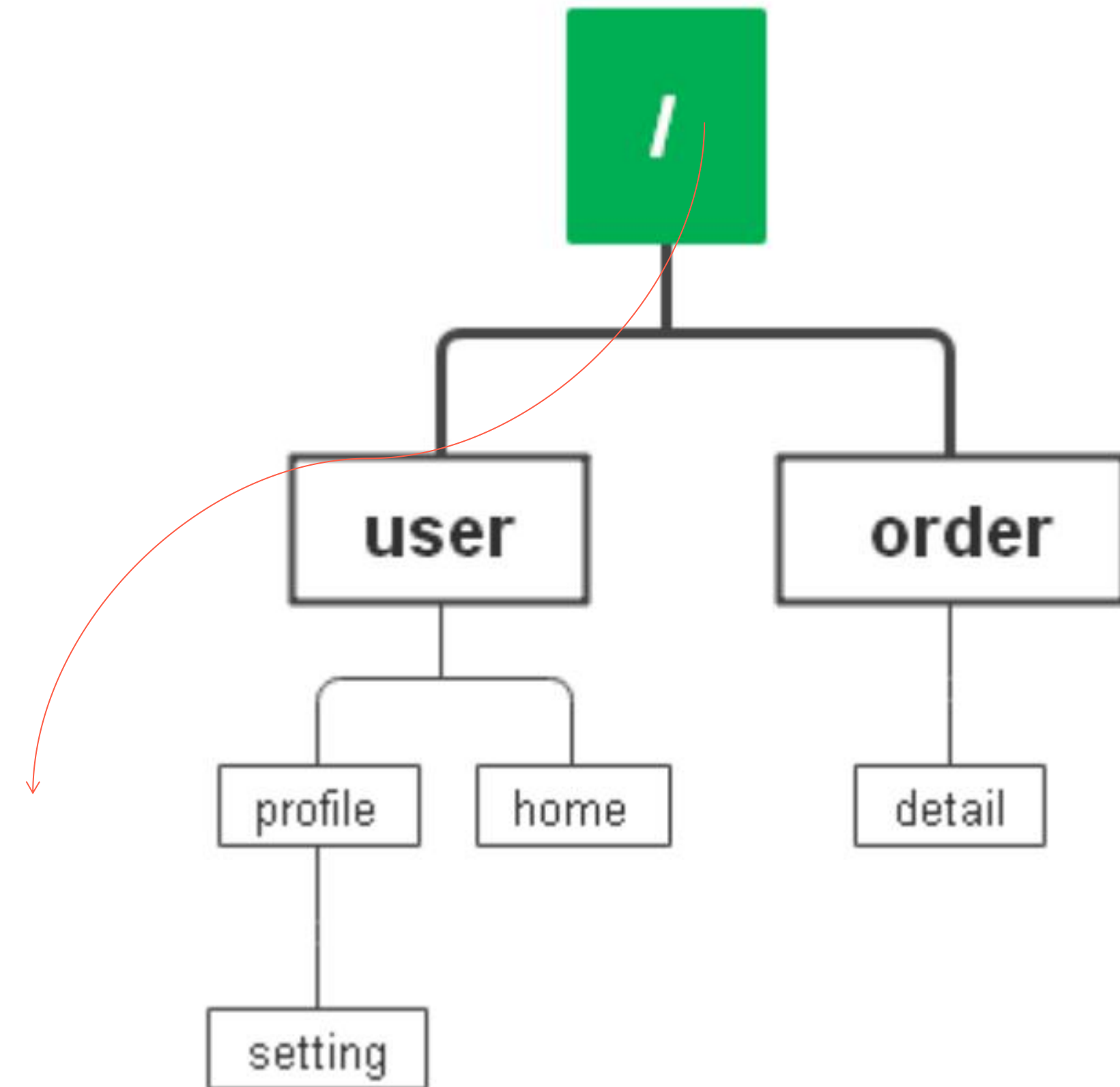
```
// ServeHTTP 就是从树里面找节点  
// 找到了就执行  
func (h *HandlerBasedOnTree) ServeHTTP(c *Context) {  
    panic(v: "implement me")  
}  
  
// Route 就相当于往树里面插入节点  
func (h *HandlerBasedOnTree) Route(method string, pattern string,  
    handlerFunc handlerFunc) {  
    panic(v: "implement me")  
}
```

# Http Server —— 增加新的路由

新增一条： `/user/friends`

步骤：

1. 从根节点出发，作为当前节点
2. 查找命中的子节点
3. 将子节点作为当前节点，重复2
4. 如果当前节点的子节点没有匹配下一段的，为下一段路径创建子节点
5. 如果路径还没结束，重复4
6. 新增成功

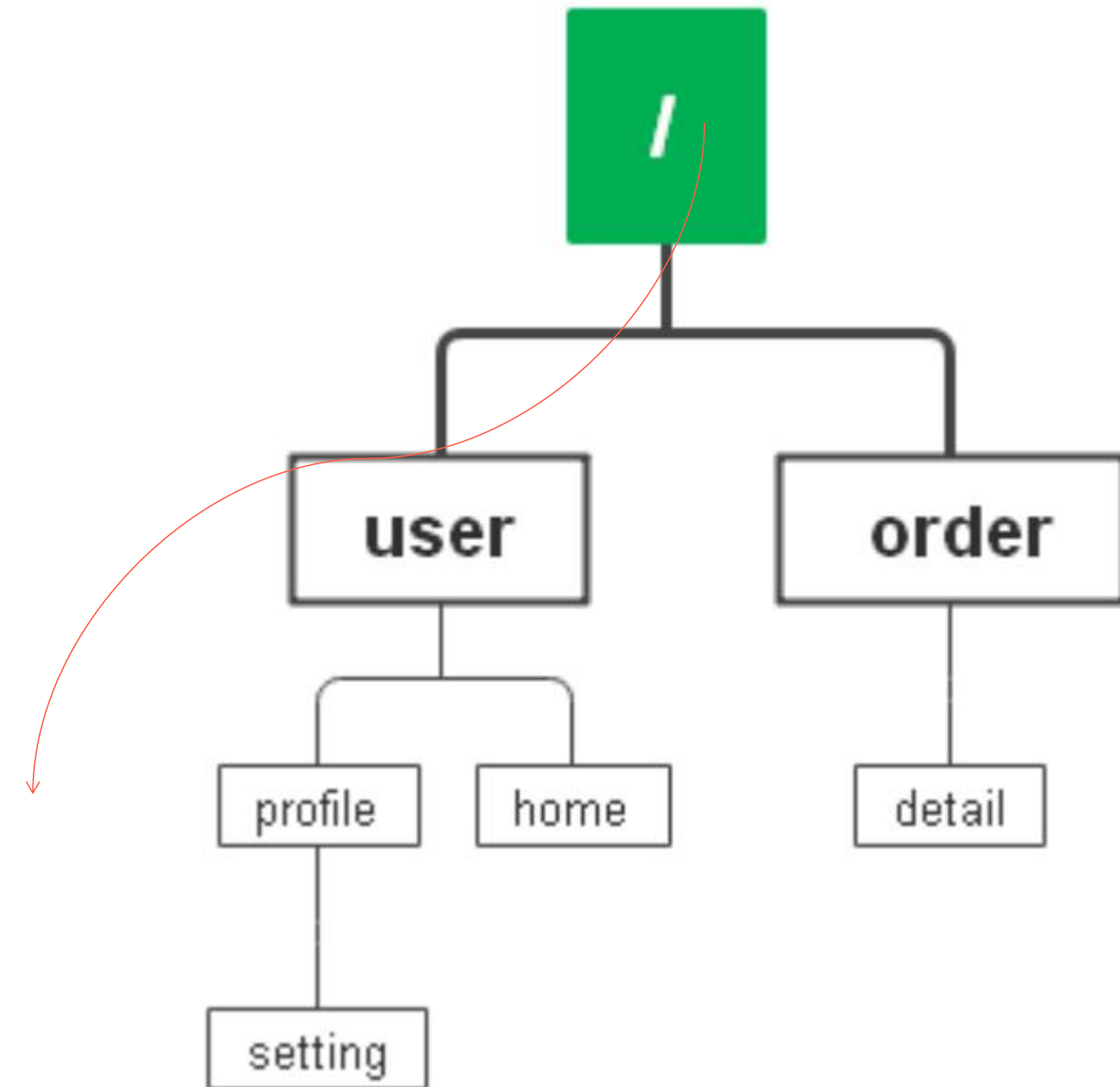


# Http Server —— 增加新的路由

新增一条: `/user/friends`

步骤:

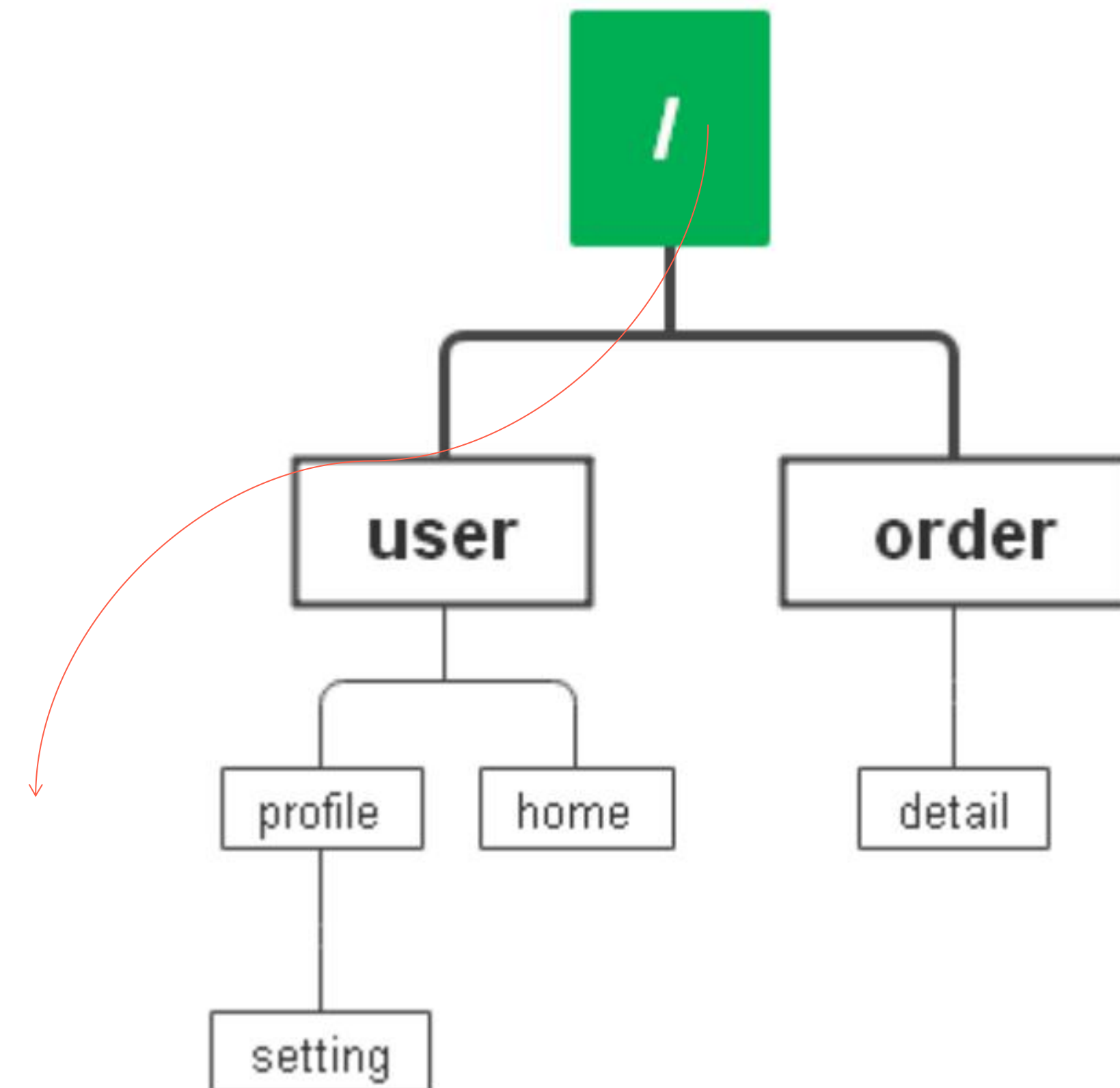
1. 根节点出发, 找到第一段命中的子节点, `user`
2. `user` 作为当前节点, 找子节点中命中 `friends` 的, 没找到
3. 在 `user` 之下创建子节点 `friends`
4. 该路由已经全部创建完成, 返回





# Http Server —— 增加新的路由

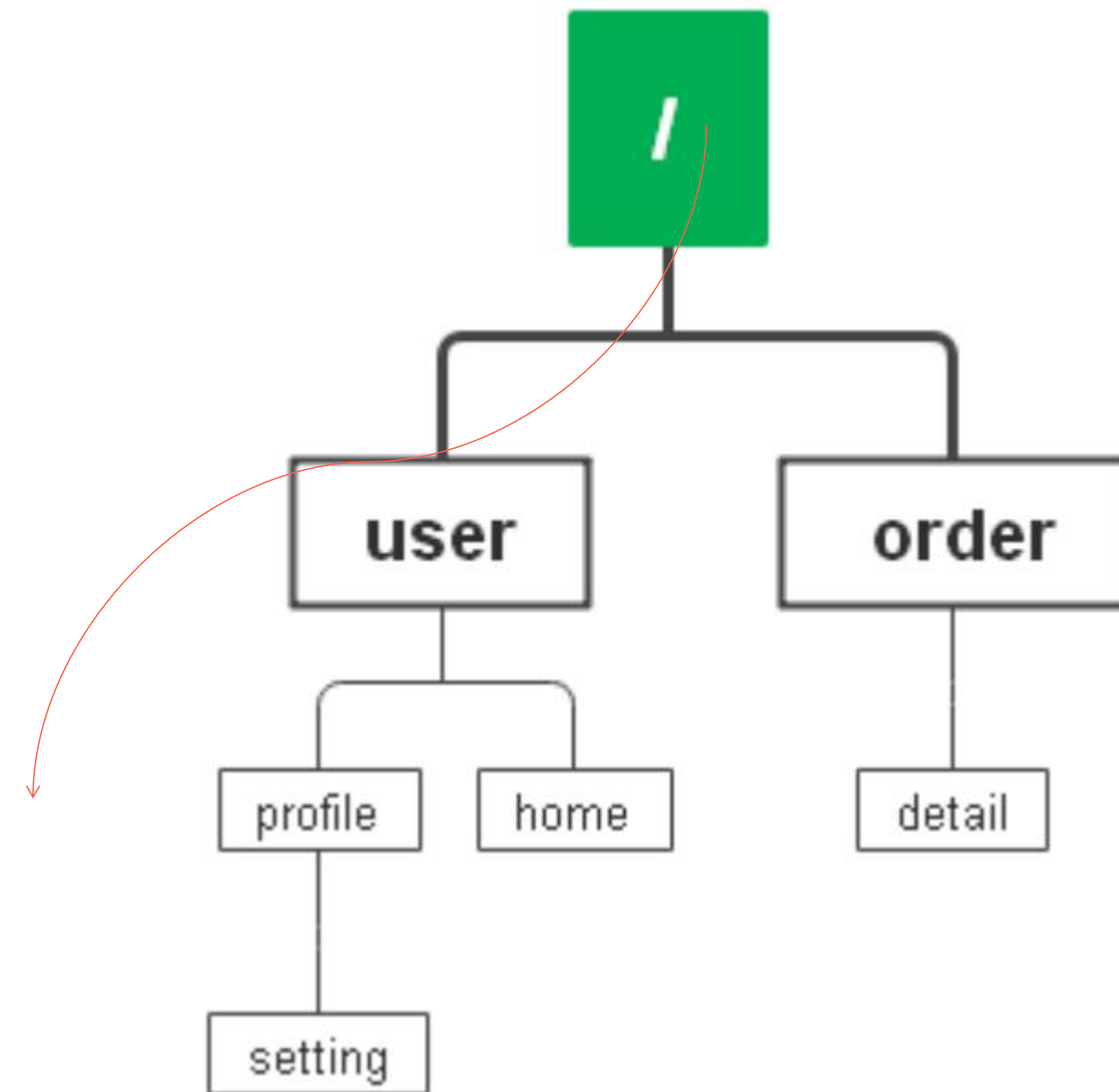
```
// Route 就相当于往树里面插入节点
func (h *HandlerBasedOnTree) Route(method string, pattern string,
    handlerFunc handlerFunc) {
    // 将pattern按照URL的分隔符切割
    // 例如, /user/friends 将变成 [user, friends]
    // 将前后的/去掉, 统一格式
    pattern = strings.Trim(pattern, cutset: "/")
    paths := strings.Split(pattern, sep: "/")
    // 当前指向根节点
    cur := h.root
    for index, path := range paths {
        // 从子节点里边找一个匹配到了当前 path 的节点
        matchChild, found := h.findMatchChild(cur, path)
        if found {
            cur = matchChild
        } else {
            // 为当前节点根据
            h.createSubTree(cur, paths[index:], handlerFunc)
            break
        }
    }
    // 离开了循环, 说明我们加入的是短路径,
    // 比如说我们先加入了 /order/detail
    // 再加入/order, 那么会走到这里
    cur.handler = handlerFunc
}
```



# Http Server —— 查找路由

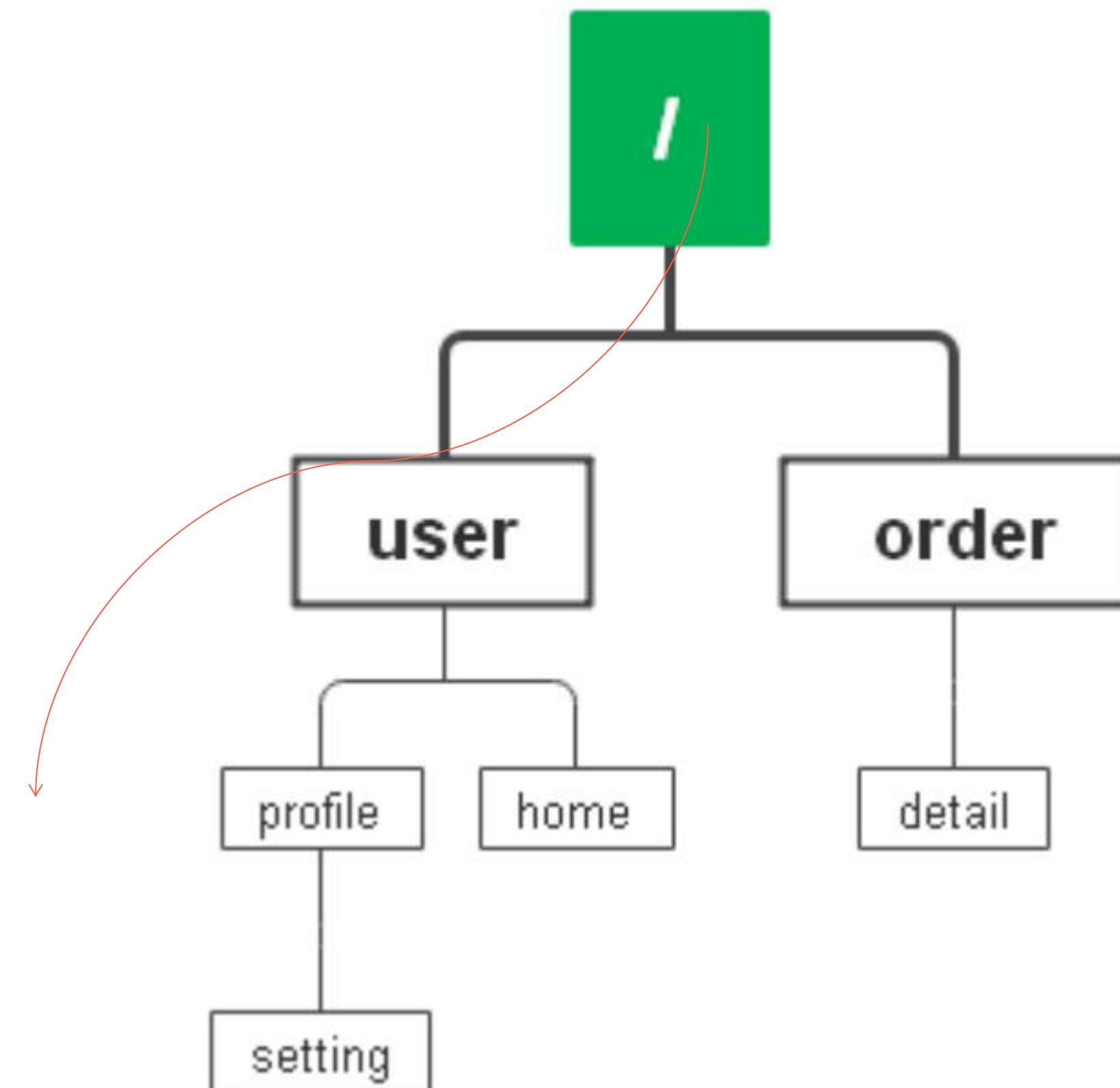
步骤:

1. 从根节点出发，作为当前节点
2. 查找子节点命中的那个节点
3. 将命中的子节点作为当前节点，重复2
4. 如果当前节点是叶子节点，并且路径已经完全匹配（不多不少），那么该节点就是找到的最终路由节点
5. 否则，没有匹配到路由



# Http Server —— 查找路由

```
func (h *HandlerBasedOnTree) ServeHTTP(c *Context) {  
    url := strings.Trim(c.R.URL.Path, cutset: "/")  
    paths := strings.Split(url, sep: "/")  
    cur := h.root  
    for _, path := range paths {  
        // 从子节点里边找一个匹配到了当前 path 的节点  
        matchChild, found := h.findMatchChild(cur, path)  
        if !found {  
            // 找不到匹配的路径, 直接返回  
            c.W.WriteHeader(statusCode: 404)  
            _, _ = c.W.Write([]byte("Not Found"))  
            return  
        }  
        cur = matchChild  
    }  
    // 到这里, 应该是找完了  
    if cur.handler == nil {  
        // 到达这里是因为这种场景  
        // 比如说你注册了 /user/profile  
        // 然后你访问 /user  
        c.W.WriteHeader(statusCode: 404)  
        _, _ = c.W.Write([]byte("Not Found"))  
        return  
    }  
    cur.handler(c)  
}
```



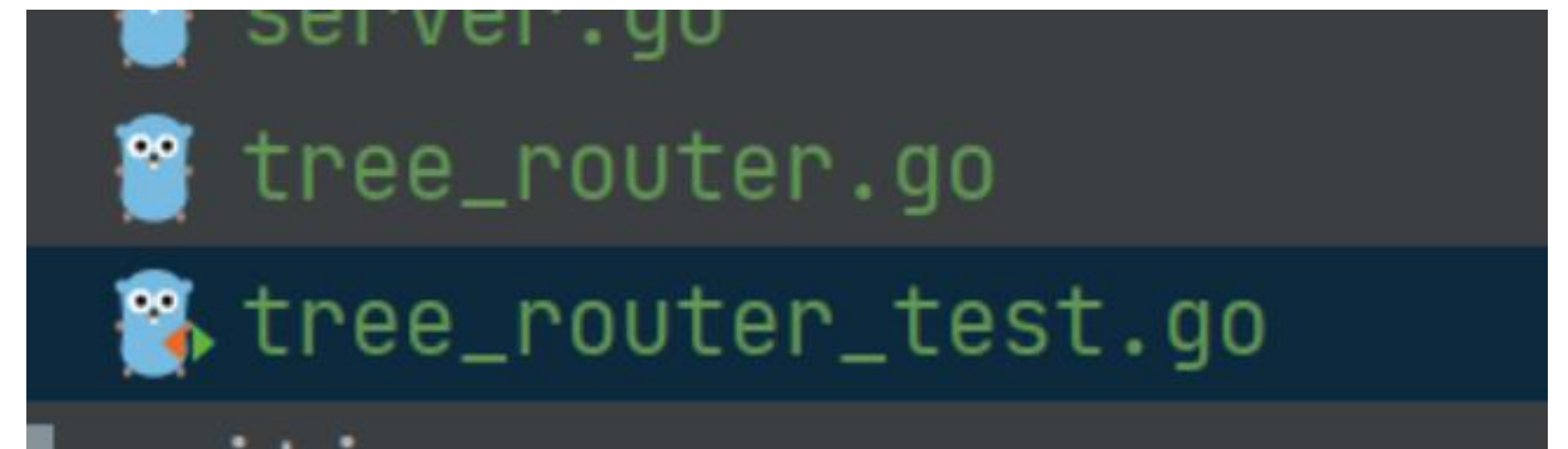


# 基础语法 —— 如何写单元测试

写了一堆代码之后，我们需要测试来确保没BUG

golang 单元测试规范：

1. 文件用xxx\_test.go结尾
2. 方法形式 TestXXXX(t \*testing.T)



```
import "testing"

func TestHandlerBasedOnTree_Route(t *testing.T) {
    // ...
}
```



# Http Server —— 测试增加路由

一般来说，单元测试都要：

1. 正常执行
2. 异常例子
3. 正常与异常的边界例子
4. 追求分支覆盖
5. 追求代码覆盖

```
func TestHandlerBasedOnTree_Route(t *testing.T) {  
    handler := NewHandlerBasedOnTree().(*HandlerBasedOnTree)  
    assert.NotNil(t, handler.root)  
  
    handler.Route(http.MethodPost, pattern: "/user", func(c *Context) {})  
  
    // 开始做断言，这个时候我们应该确认，在根节点之下只有一个user节点  
    assert.Equal(t, expected: 1, len(handler.root.children))  
  
    n := handler.root.children[0]  
    assert.NotNil(t, n) // assert 是第三方库  
    assert.Equal(t, expected: "user", n.path)  
    assert.NotNil(t, n.handler)  
    assert.Empty(t, n.children)  
}
```

Tip: 注意这不是理论上正统的单元测试写法，我们这个还是比较粗糙的

# Http Server —— 写好测试的代码

为什么这个代码不好测：

和http.Request, http.ResponseWriter 耦合起来了

1. 难以构造 http.Request
2. 难以对 http.ResponseWriter 做断言

```
func (h *HandlerBasedOnTree) ServeHTTP(c *Context) {
    url := strings.Trim(c.R.URL.Path, cutset: "/")
    paths := strings.Split(url, sep: "/")
    cur := h.root
    for _, path := range paths {
        // 从子节点里边找一个匹配到了当前 path 的节点
        matchChild, found := h.findMatchChild(cur, path)
        if !found {
            // 找不到匹配的路径，直接返回
            c.W.WriteHeader(statusCode: 404)
            _, _ = c.W.Write([]byte("Not Found"))
            return
        }
        cur = matchChild
    }
    // 到这里，应该是找完了
    if cur.handler == nil {
        // 到达这里是因为这种场景
        // 比如说你注册了 /user/profile
        // 然后你访问 /user
        c.W.WriteHeader(statusCode: 404)
        _, _ = c.W.Write([]byte("Not Found"))
        return
    }
    cur.handler(c)
}
```



# Http Server —— 写好测试的代码

抽取一个和 `http.Request`,  
`http.ResponseWriter` 没有关系的方法。

```
func (h *HandlerBasedOnTree) findRouter(path string) (handlerFunc, bool) {  
    // 去除头尾可能有的/, 然后按照/切割成段  
    paths := strings.Split(strings.Trim(path, "/"), "/")  
    cur := h.root  
    for _, p := range paths {  
        // 从子节点里边找一个匹配到了当前 path 的节点  
        matchChild, found := h.findMatchChild(cur, p)  
        if !found {  
            return nil, false  
        }  
        cur = matchChild  
    }  
    // 到这里, 应该是找完了  
    if cur.handler == nil {  
        // 到达这里是因为这种场景  
        // 比如说你注册了 /user/profile  
        // 然后你访问 /user  
        return nil, false  
    }  
    return cur.handler, true  
}
```

# Http Server —— 改造一下Server

```
func NewSdkHttpServer(name string, builders ...FilterBuilder) Server {  
    // 改用我们的树  
    handler := NewHandlerBasedOnTree()  
    // handler := NewHandlerBasedOnMap()  
    // 因为我们是一个链，所以把最后的业务逻辑处理，也作为一环  
    var root Filter = handler.ServeHTTP  
    // 从后往前把filter串起来  
    for i := len(builders) - 1; i >= 0; i-- {  
        b := builders[i]  
        root = b(root)  
    }  
    res := &SdkHttpServer{  
        name: name,  
        handler: root,  
    }
```

Add a description

POST localhost:8080/user/create

Authorization Headers (1) Body Pre-request Script Tests

Type No Auth

Body Cookies Headers (3) Test Results

Pretty Raw Preview Text

1 {"biz\_code":0,"msg":"","data":123}



# 基础语法 —— go mod 管理依赖

- 增加 go mod
- go mod init 初始化模块
- go get 添加依赖



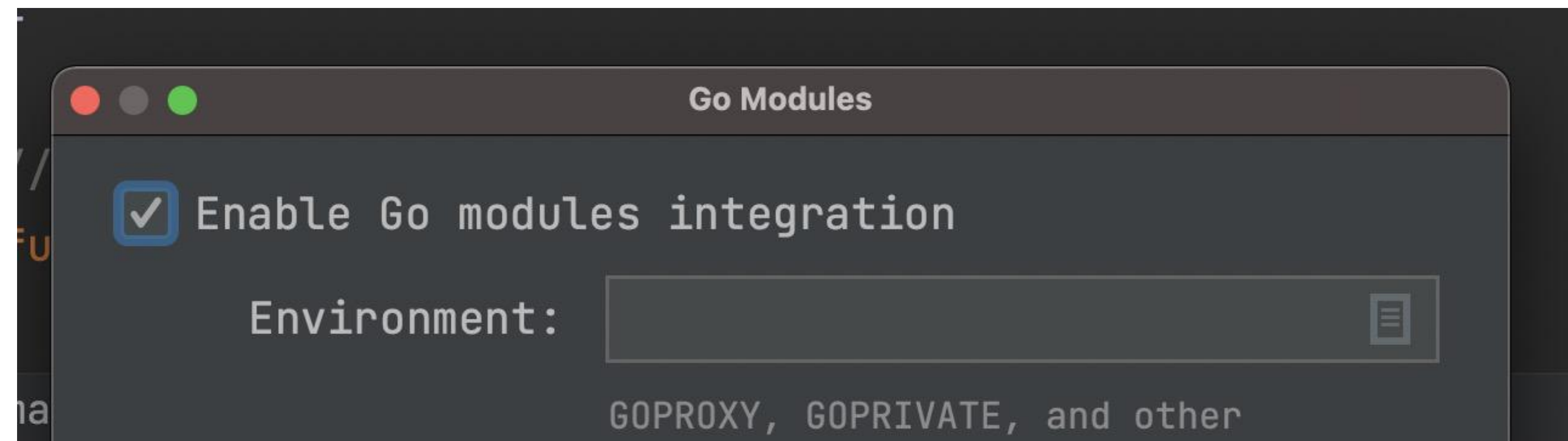
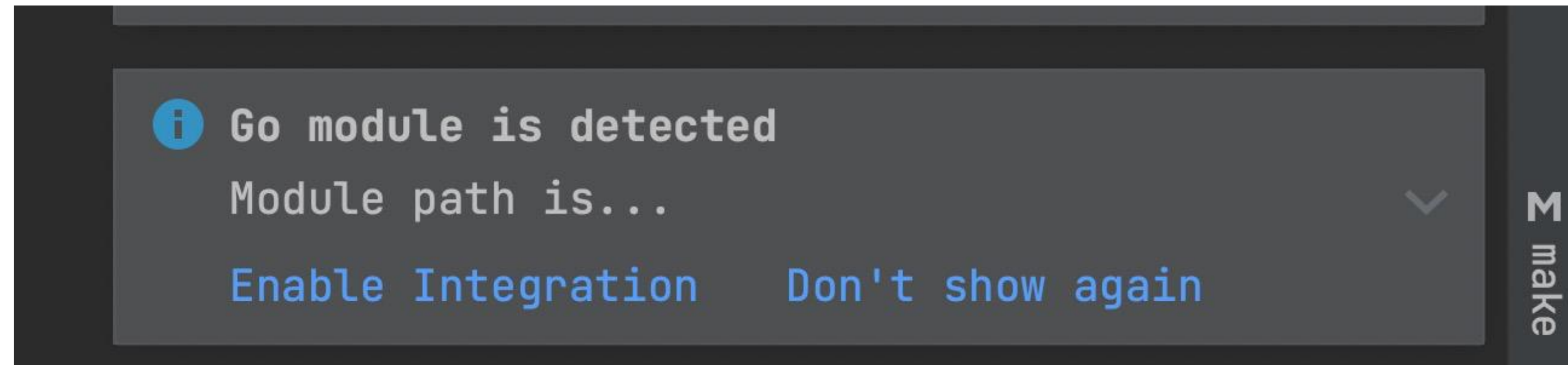
```
D:\workspace\go\src\geektime\toy-web>go mod init
go: creating new go.mod: module geektime/toy-web
go: to add module requirements and sums:
    go mod tidy
```

```
D:\workspace\go\src\geektime\toy-web>go mod tidy
```

```
D:\workspace\go\src\geektime\toy-web>go get github.com/stretchr/testify
go get: added github.com/stretchr/testify v1.7.0
```

# 基础语法 —— go mod 管理依赖

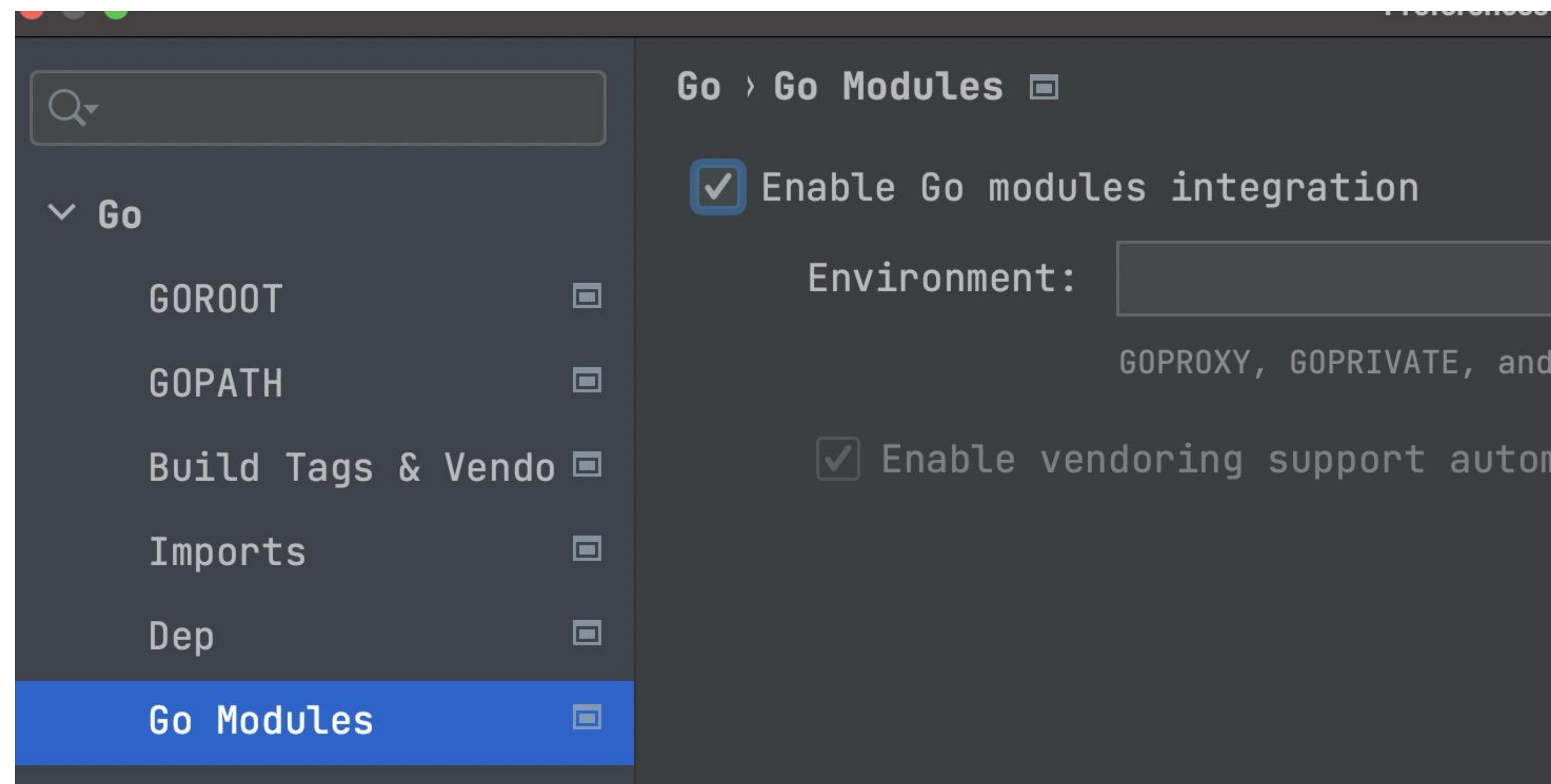
- 配置 Goland



Tip: 使用Goland来创建文件夹，它会自动加上package

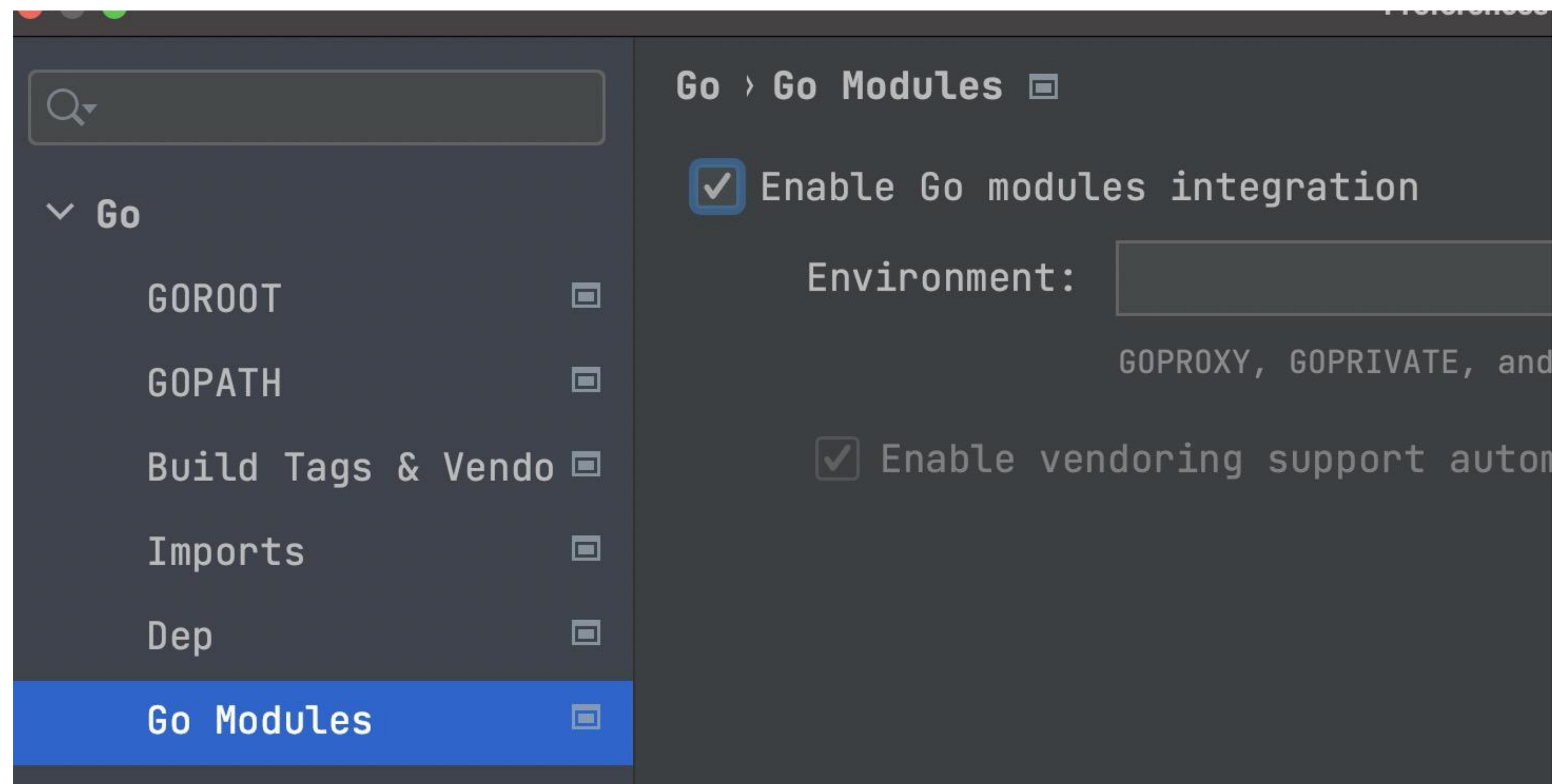
# 基础语法 —— go mod 管理依赖

- 直接开发设置配置也可以



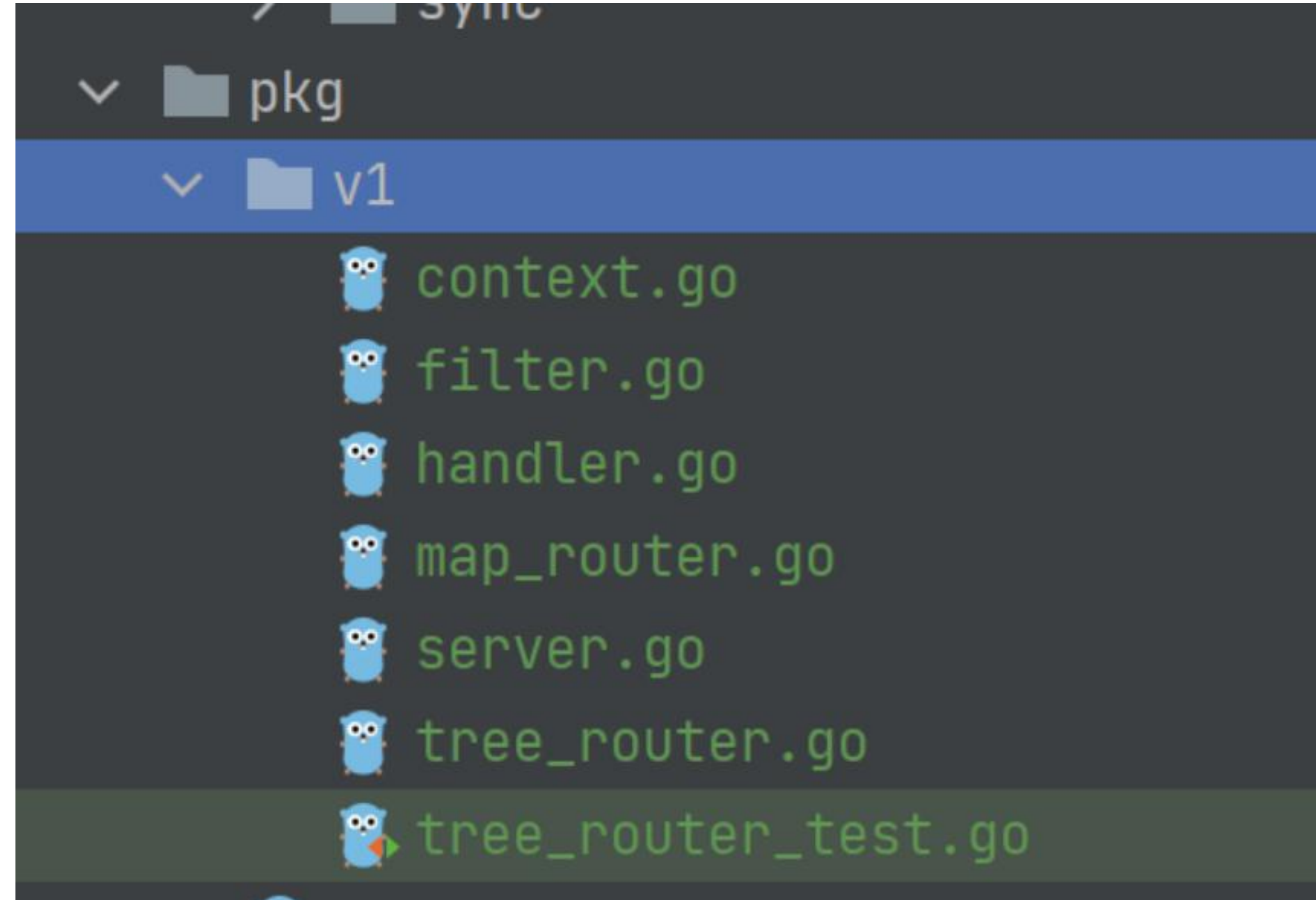
# 基础语法 —— go mod 管理依赖

- 直接开发设置配置也可以





# Http Server —— V1 完成了



# Http Server —— 路由树设计

- v1: 极简路由实现
- v2: 支持 \* 匹配
- v3: 支持复杂匹配

## Http Server —— v2 支持 \* 匹配路由

刚才我们的v1实在是太死板了，只能所有的路径都完全匹配。

比如说注册 `/order` 的路由，那么 `/order/order_sn` 就不匹配了。

我们尝试扩展一下，支持 `*`，例如注册路由 `/order/*`

那么 `/order/order_sn` 就匹配上。

# Http Server —— 匹配原则

问题来了：

1. 用户注册了一个路由 `/order/*`
2. 用户又注册了一个路由 `/order/order_sn`

请求 `/order/order_sn` 该命中哪一个？

答案：我们遵循最“详细”原则。按照一般的说法，应该是最左最长匹配。所以应该命中 `/order/order_sn` 而不是 `/order/*`



# Http Server —— 支持 \*

需要审视一下我们的代码：

修改的点会在哪里？

```
func (h *HandlerBasedOnTree) findMatchChild(root *node, path string) (*node, bool) {  
    for _, child := range root.children {  
        if child.path == path {  
            return child, true  
        }  
    }  
    return nil, false  
}
```

# Http Server —— 支持 \*

需要审视一下我们的代码：

似乎可以了？

这个忽略了冲突的问题

这一段的逻辑按照顺序找，  
而我们期望的是，**优先“挑选详细的”**

```
func (h *HandlerBasedOnTree) findMatchChild(root *node, path string) (*node, bool) {  
    var wildcardNode *node  
    for _, child := range root.children {  
        // 并不是 * 的节点命中了，直接返回  
        // != * 是为了防止用户乱输入  
        if child.path == path &&  
            child.path != "*" {  
            return child, true  
        }  
        // 命中了通配符的，我们看看后面还有没有更加详细的  
        if child.path == "*" {  
            wildcardNode = child  
        }  
    }  
    return wildcardNode, wildcardNode != nil  
}
```

# Http Server —— 支持 \*

接下来又遇到一个问题：

我注册了 /order/\*

又注册了 /order/\*/checkout/confirm

这时候，我一个请求：

/order/123/checkout/cancel

应该命中 /order/\* 吗？

```
func (h *HandlerBasedOnTree) findMatchChild(root *node, path string) (*node, bool) {
    var wildcardNode *node
    for _, child := range root.children {
        // 并不是 * 的节点命中了，直接返回
        // != * 是为了防止用户乱输入
        if child.path == path &&
            child.path != "*" {
            return child, true
        }
        // 命中了通配符的，我们看看后面还有没有更加详细的
        if child.path == "*" {
            wildcardNode = child
        }
    }
    return wildcardNode, wildcardNode != nil
}
```

# Http Server —— 支持 \*

答案是：犯不着命中。

为什么？不是做不到，而是会让代码唰一下变复杂，投入大产出小。

为什么代码会复杂？因为你需要回溯了，当你发现匹配不到任何的时候，你要回溯你的路由树，看看最近有没有一个通配符节点

```
func (h *HandlerBasedOnTree) findMatchChild(root *node, path string) (*node, bool) {
    var wildcardNode *node
    for _, child := range root.children {
        // 并不是 * 的节点命中了，直接返回
        // != * 是为了防止用户乱输入
        if child.path == path &&
            child.path != "*" {
            return child, true
        }
        // 命中了通配符的，我们看看后面还有没有更加详细的
        if child.path == "*" {
            wildcardNode = child
        }
    }
    return wildcardNode, wildcardNode != nil
}
```



# Http Server —— 支持 \*

注册的时候，加上校验，如果出现了\*，它就必须是最后一个。并且我们要求，它前面必须是 /。即最终结尾应该是 /\*

```
guard error
var ErrorInvalidRouterPattern = errors.New(text: "invalid router pattern")
```

Tip: 很多框架会预定义一些错误，这往往意味着你需要对这些错误特殊处理，或者多加关注，典型的就是 ErrNoRow

```
func (h *HandlerBasedOnTree) validatePattern(pattern string) error {
    // 校验 *, 如果存在, 必须在最后一个, 并且它前面必须是 /
    // 即我们只接受 /* 的存在, abc*这种是非法

    pos := strings.Index(pattern, substr: "*")
    // 找到了 *

    if pos > 0 {
        // 必须是最后一个
        if pos != len(pattern) - 1 {
            return ErrorInvalidRouterPattern
        }
        if pattern[pos-1] != '/' {
            return ErrorInvalidRouterPattern
        }
    }

    return nil
}
```

# Http Server —— v2 完结

这里还有一个小问题：

`/order/*` 究竟能不能匹配 `/order` ？

这个问题的看法，这是一个典型的设计决策问题。因为无论是否匹配，都能找到一些“冠冕堂皇”的理由。

这里我没有支持，仅仅是因为我觉得，当你希望匹配 `/order` 的时候，你完全可以自己注册一下。当你注册 `/order/*` 我觉得你是期望 `/order` 之后肯定有东西的。不然你注册 `/order` 就可以了。

我也不觉得，正常的业务下 `/order` 和 `/order/*` 对应的 handler 会是一样的。

这就是一个很有意思的话题：**中间件开发者试图教你如何写代码。**

如果你发现有些框架，不能这么写，不能那么写。有时候是功能确实不好支持，有些时候就是作者在教你写代码，他通过不支持某些东西，来避免用户写出一些他完全无法接受的代码。所谓的无法接受的代码，你可以稍微思考一下，这个确实是不良实践，还是说这就是作者的个人偏好。

**Tip：不想教用户如何写代码的框架开发者是没有原则的**

# Http Server —— 路由树设计

- v1: 极简路由实现
- v2: 支持 \* 匹配
- v3: 支持复杂匹配

# Http Server —— v3 支持路径参数

直到现在我们都没有解决一个问题，所谓的路径参数，在 RESTFu1 里面很常见。

例如 github.com 上的典型网址

github.com/flycash

其实命中的是

github.com/:username

username 就是我们的路径参数



# Http Server —— 思路是类似 \* 的



所以大明你只会教我们复制粘  
贴和 if-else 吗？

# Http Server —— 抽象思考一下路由匹配

目前为止：

1. 我们支持静态路由，就是严格匹配，节点的path和传入的path一模一样才匹配
2. 通配符 \*，可以在末尾匹配任何的路由

我们计划加上支持路径参数，语法是 :paramName;

我们还没支持 http method

我们未来还可以支持正则路由，即路径匹配正则表达式；

我们也可以设计自己的特有语法，设计特有的匹配算法……

```
func (h *HandlerBasedOnTree) findMatchChild(root *node, path string) (*node, bool) {  
    var wildcardNode *node  
    for _, child := range root.children {  
        // 并不是 * 的节点命中了，直接返回  
        // != * 是为了防止用户乱输入  
        if child.path == path &&  
            child.path != "*" {  
            return child, true  
        }  
        // 命中了通配符的，我们看看后面还有没有更加详细的  
        if child.path == "*" {  
            wildcardNode = child  
        }  
    }  
    return wildcardNode, wildcardNode != nil  
}
```

# Http Server —— 抽象思考一下路由匹配

总结如下：

1. 一个节点怎么才算是匹配上了
2. 同一层，多个节点都匹配上了，优先挑哪个？
3. 要不要考虑朝前看朝后看的问题？

# Http Server —— 路由树节点抽象

一个节点怎么才算是匹配上了？

让节点自己决定

```
type Node interface {  
    // Match 会尝试在匹配上了之后，注入路径参数到 Context之中  
    Match(method string, path string, c *Context) bool  
}
```

```
type baseNode struct {  
    children []Node  
    handler handlerFunc  
}
```

```
type staticNode struct {  
    baseNode  
    path string  
}
```

```
func (s *staticNode) Match(method string,  
    path string, c *Context) bool {  
    return path == s.path  
}
```



# Http Server —— 节点定义

```
// 通配符 * 节点
func newAnyNode() *node {
    return &node{
        // 因为我们不允许 * 后面还有节点，所以这里可以不用初始化
        // children: make([]*node, 0, 2),
        matchFunc: func(p string, c *Context) bool {
            return true
        },
        nodeType: nodeTypeAny,
        pattern: any,
    }
}
```

```
// 路径参数节点
func newParamNode(path string) *node {
    paramName := path[1:]
    return &node{
        children: make([]*node, 0, 2),
        matchFunc: func(p string, c *Context) bool {
            if c != nil {
                c.PathParams[paramName] = p
            }
            // 如果自身是一个参数路由，
            // 然后又来一个通配符，我们认为是不匹配的
            return p != any
        },
        nodeType: nodeTypeParam,
        pattern: path,
    }
}
```

# Http Server —— 修改查找子节点方法

这里的候选者挑出来之后，究竟走哪个，就取决于自己的需要了

```
func (h *HandlerBasedOnTree) findMatchChild(root *node,
path string, c *Context) (*node, bool) {
    candidates := make([]*node, 0, 2)
    for _, child := range root.children {
        if child.matchFunc(path, c) {
            candidates = append(candidates, child)
        }
    }

    if len(candidates) == 0 {
        return nil, false
    }

    // type 也决定了它们的优先级 偷懒的做法，工业产品应该把这个做成策略模式
    sort.Slice(candidates, func(i, j int) bool {
        return candidates[i].nodeType < candidates[j].nodeType
    })
    return candidates[len(candidates) - 1], true
}
```

# Http Server —— 自己尝试扩展正则节点 (optional)

你需要:

- 定义正则路由的表达方式。例如说 `/user/:reg(正则表达式)`
- 定义路由节点的匹配方式
- 决定正则路由节点与其它路由节点是否兼容。例如当我注册了 `/user/:reg([1,9]+)` 之后, 我还能否注册 `/user/*` 或者 `/user/:id`

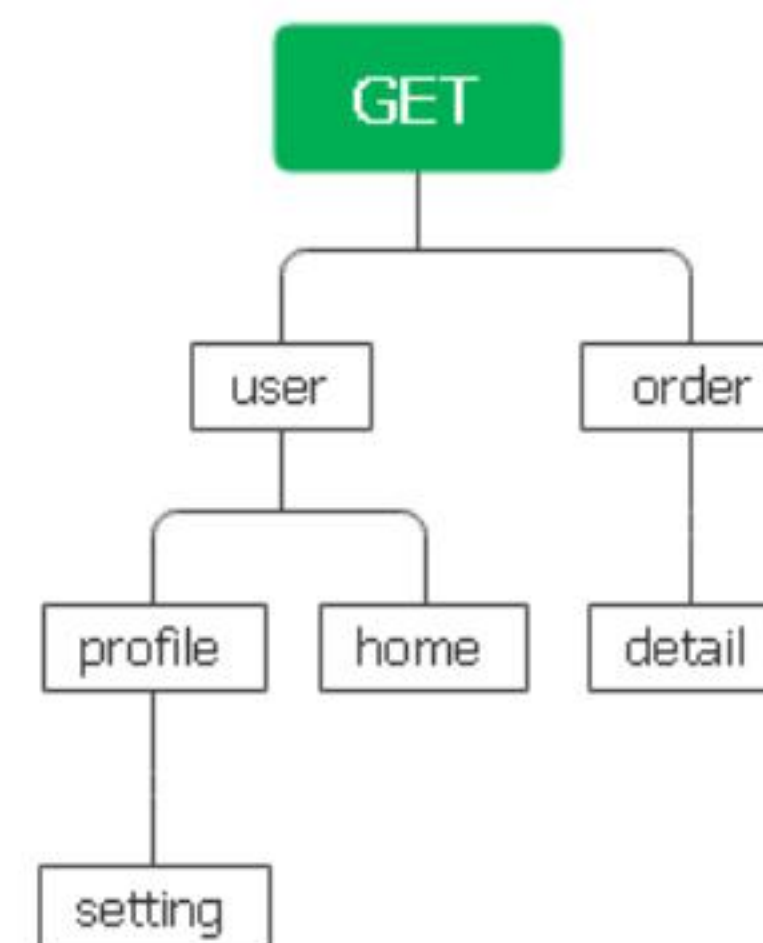
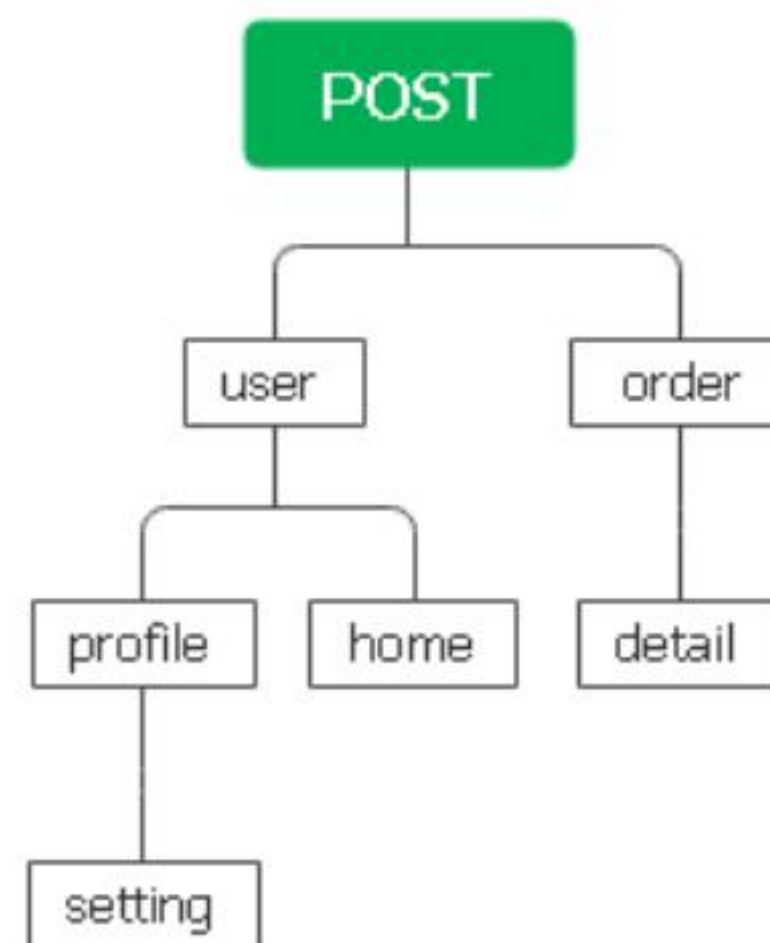
```
// 正则节点
func newRegNode(path string) *node {
    // 依据你的规则拿到正则表达式
    return &node{
        children: make([]*node, 0, 2),
        matchFunc: func(p string, c *Context) bool {
            // 怎么写?
        },
        nodeType: nodeTypeParam,
        pattern: path,
    }
}
```



# Http Server —— Http Method 支持

目前我们的路由不支持区分 http method, 思路有:

1. 不同的方法构建不同的树, 例如 POST 一个, GET 一个;
2. 节点里面加上方法作为区别。例如最开始我们的静态节点 /user 只创建为 user。现在可以加上方法名, 比如说 POST user 是一个节点, GET user 是一个节点

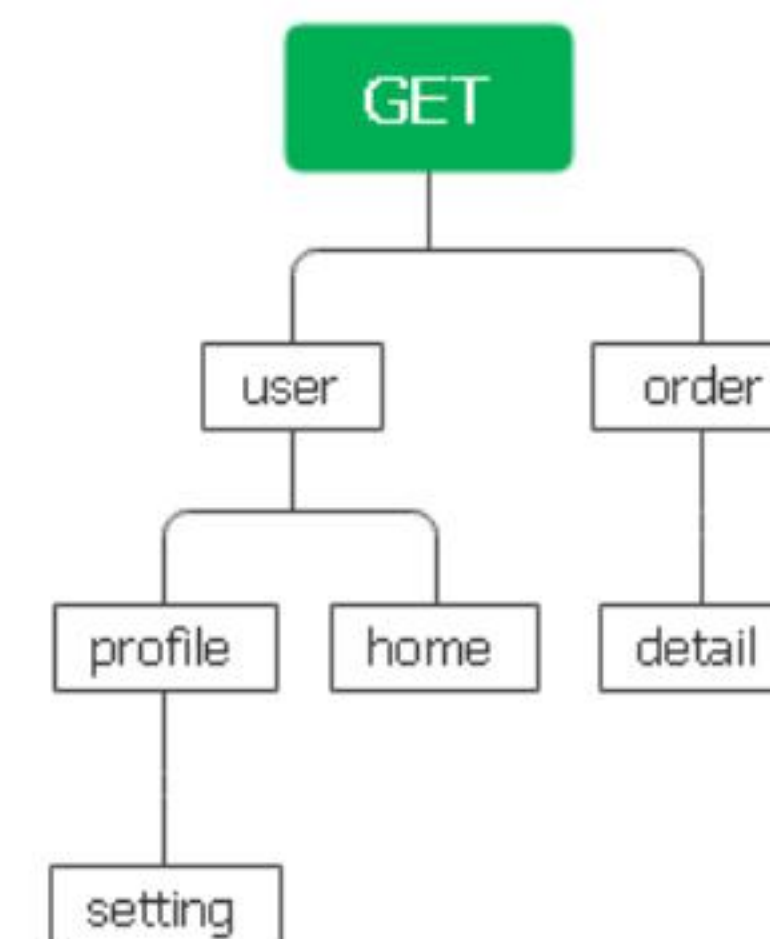
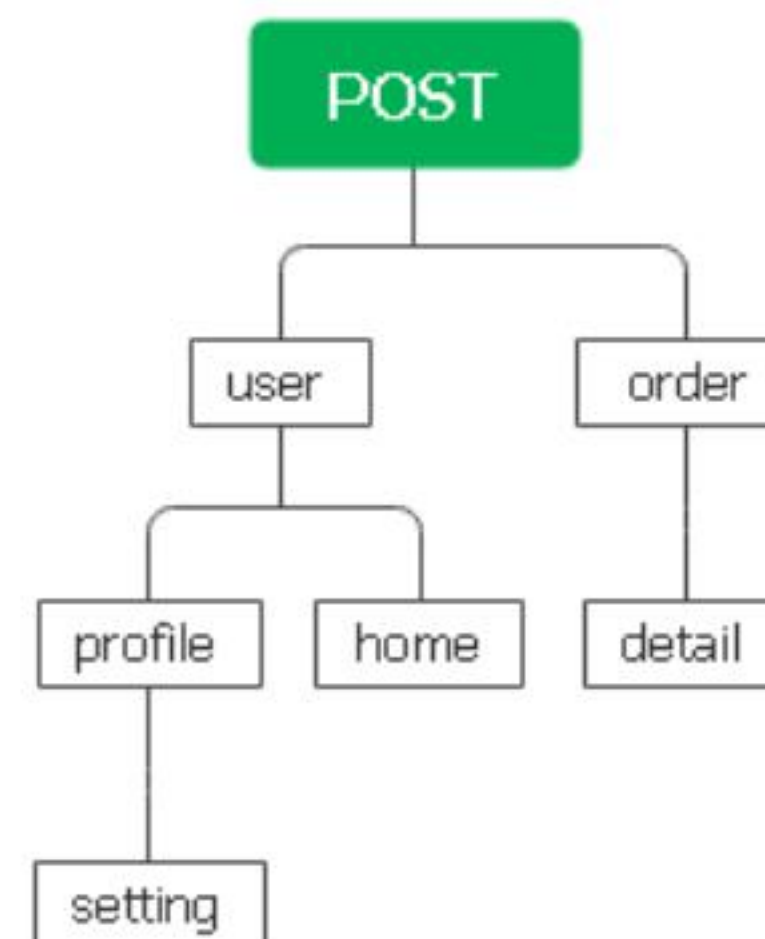




# Http Server —— Http Method 支持

2 虽然理论上可行，但是我们往往不这么搞。因为一种方法一棵树，我们只需要处理http method一次，即挑树。而 2 的设计思路意味着，每一个节点都要处理一下 http method。实现起来要复杂

```
type HandlerBasedOnTree struct {  
    forest map[string]*node  
}
```



Tip: 其实你可以设计为一棵树。根节点的子节点就是http method 节点。在 toy-web 里面很容易改造

# Http Server —— 改进点 (optional)

- 将 Node 抽象为接口
- 将 children 进一步抽象，快速定位可能匹配的子节点。比如说直接 map，或者首字母组成数组
- 支持输出完整路径。比如说我们现在命中了/order/:id，我怎么输出这个东西？很多监控场景下，我们是要拿到命中的路由的。（可以考虑在叶子节点维护整个路由规则，也可以在树向下遍历的时候记录路径）
- 扩展接口，允许用户无侵入式扩展自己的路由规则。假如说我是一个用户，我有一个特殊的语法，比如说/user/:daming:uint(userid)。daming 是我设计的路由规则标识，后面的uint(userid)是指这一段被解析为userid，并且路径参数要帮我转化为 uint 类型
- 用正则节点来实现其它节点。这不是改进，从性能上来说反而更差，但是这能够有助于理解“统一抽象”

Tip: 后两点考验的是设计能力，所谓的先有一致的抽象，后面才有不同的实现

# 要点总结

- 从简单到复杂一步步实现，不要一开始就想搞出来满足所有功能的设计
- 设计是一个迭代的过程

# 课后练习

- 沿着课上的思路实现一遍路有树。或者直接从最后的版本出发
- 尝试实现 `error` 接口
- 实现任何一个 `filter` 并且集成进去 `server` 里面
- 尝试自己设计一下 `filter` 接口（非函数式的设计）
- 预习 `channel` 和 `select`



# THANKS

 极客时间 | 训练营