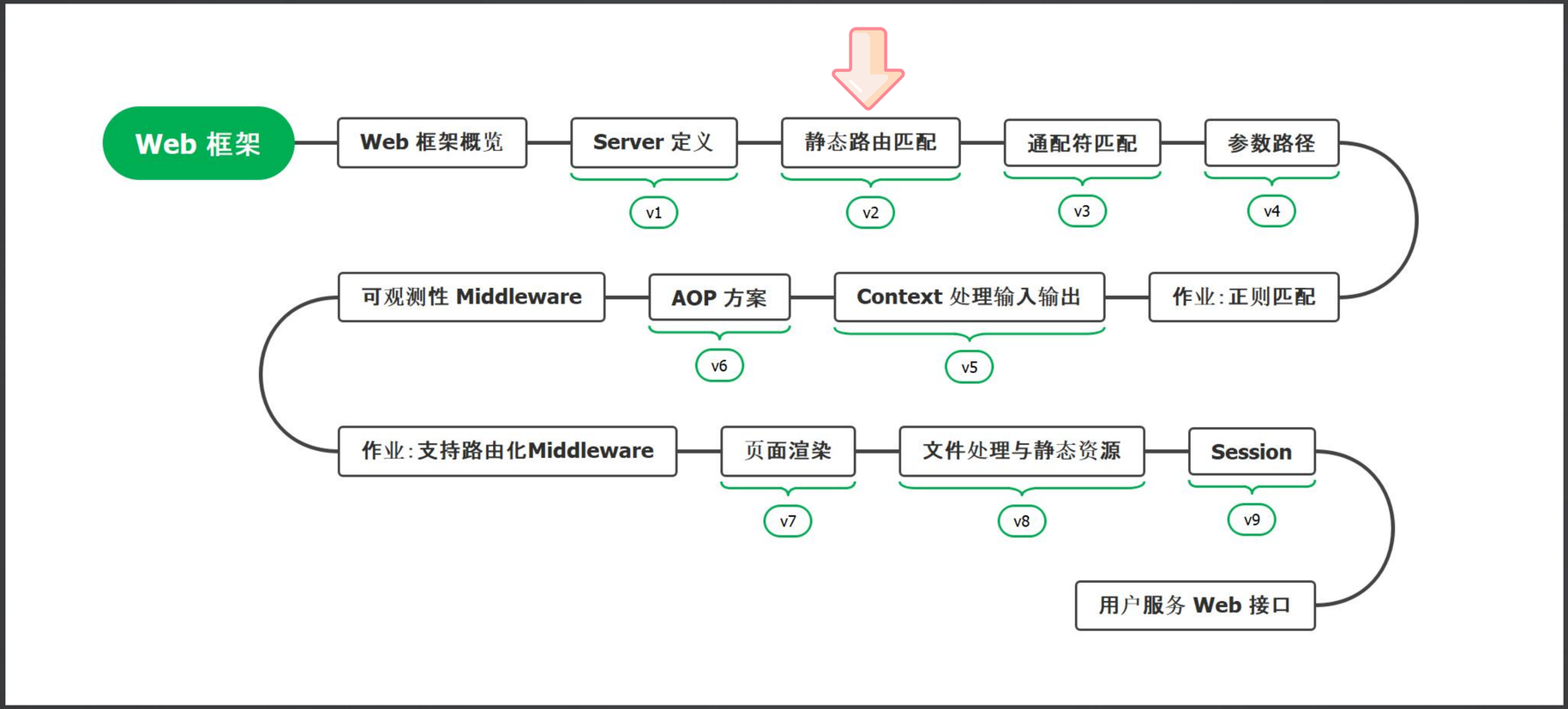
# Web 模块——路由树

大明

# 学习路线—— 静态路由匹配


Web 框架
Web 框架概览
Server 定义
静态路由匹配
通配符匹配
参数路径
v1
v2
v3
v4
可观测性 Middleware
AOP 方案
Context 处理输入输出
作业：正则匹配
v6
v5
作业：支持路由化Middleware
页面渲染
文件处理与静态资源
Session
v7
v8
v9
用户服务 Web 接口

# 路由树

我们分成以下步骤来实现一颗路由树：

- 全静态匹配
- 支持通配符匹配
- 支持参数路由

在这之前，我们需要再一次稍微考察一下各个框架的路由树是怎么实现的。

# 路由树 —— Beego 实现

Beego 的核心结构体是三个：

- ControllerRegister：类似于容器，放着所有的路由树
  - 路由树是按照 HTTP method 来组织的，例如 GET 方法会对应有一棵路由树
- Tree：它代表的就是路由树，在 Beego 里面，一棵路由树被看做是由子树组成的
- leafInfo：代表叶子节点

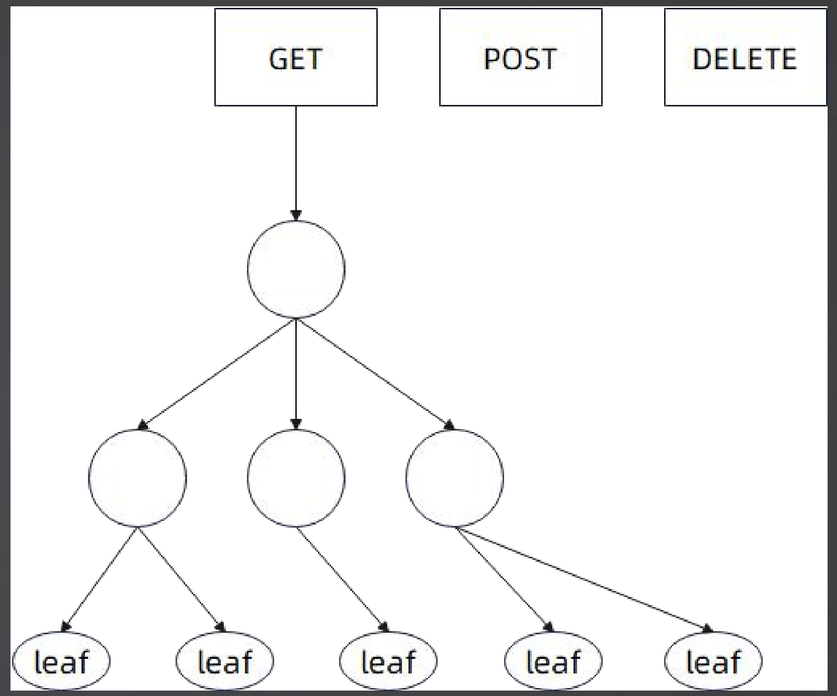
```go
// ControllerRegister containers regis
type ControllerRegister struct {
    routers      map[string]*Tree
    enablePolicy bool
    enableFilter bool
}
```

```go
// leaves store the endpoint information
type Tree struct {
    // prefix set for static router
    prefix string
    // search fix route first
    fixrouters []*Tree
    // if set, failure to match fixrouters searc
    wildcard *Tree
    // if set, failure to match wildcard search
    leaves []*leafInfo
}
```

```go
type leafInfo struct {
    // names of wildcards that
    wildcards []string

    // if the leaf is regexp
    regexps *regexp.Regexp

    runObject interface{}
}
```

# 路由树 —— Beego 设计抽象

Beego 的树定义，并没有采用 children 式的定义，而是采用递归式的定义，即一棵树是由根节点 + 子树构成。



# 路由树 —— Gin 实现

Gin 的关键结构体更加直观：

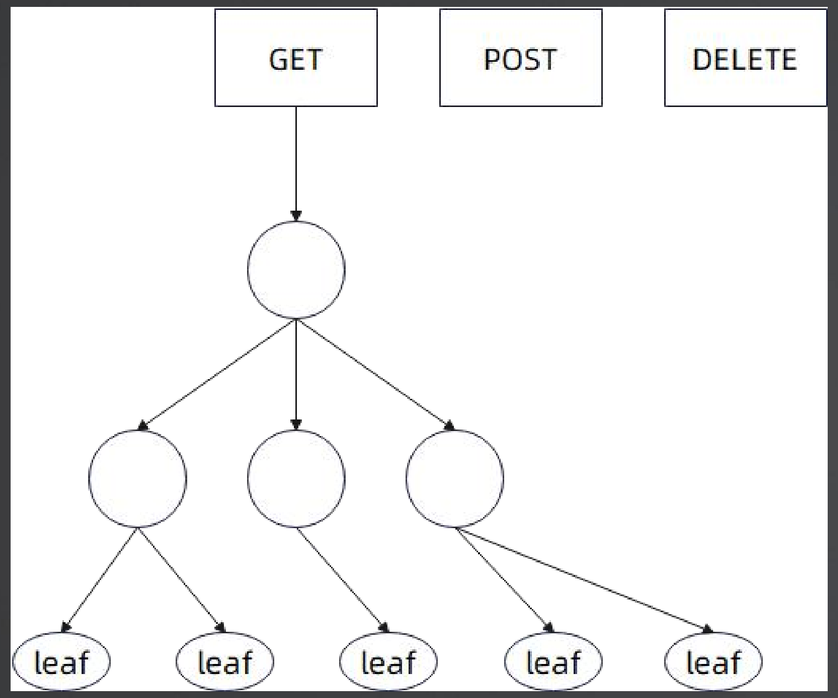
- methodTrees：也就是路由树也是按照 HTTP 方法组织的，例如 GET 会有一棵路由树
- methodTree：定义了单棵树。树在 Gin 里面采用的是 children 的定义方式，即树由节点构成（注意对比 Beego）
- node：代表树上的一个节点，里面维持住了 children，即子节点。同时有 nodeType 和 wildChild 来标记一些特殊节点

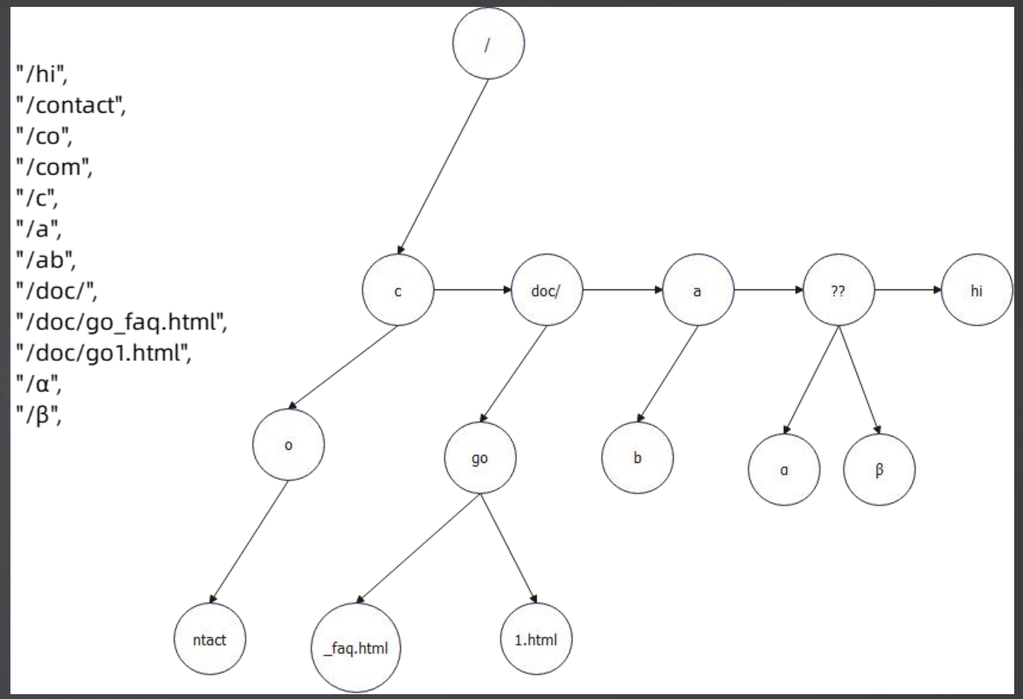
```
pool                    sync.Pool
trees                   methodTrees
maxParams               uint16
```

```
type methodTree struct {
    method string
    root   *node
}

type methodTrees []methodTree
```

```
type node struct {
    path      string
    indices   string
    wildChild bool
    nType     nodeType
    priority  uint32
    children  []*node // child n
    handlers  HandlersChain
    fullPath  string
}
```

# 路由树 —— Gin 设计抽象




Gin 是利用路由的公共前缀来构造路由树。

# 路由树 —— Echo 实现

Echo 的实现稍微有点复杂，并且概念之间不是很清晰：

- 在 Echo 结构体里面维护了 routers，但是 routers 并不是按照 HTTP 方法组织的，而是按照所谓的 Host 来组织的，Host 可以看做一个命名空间
- Router：代表路由注册中心，里面维护了路由树
- Route：代表具体的路由
- node：则是中规中矩的树节点设计，它内部依旧采用类型的数据

```
maxParam          *int
router            *Router
routers           map[string]*Router
```

```
// Router is the registry of a
// request matching and URL pa
Router struct {
    tree   *node
    routes map[string]*Route
    echo   *Echo
}
```

```
// Route contains a handler and informat
Route struct {
    Method string `json:"method"`
    Path   string `json:"path"`
    Name   string `json:"name"`
}
```

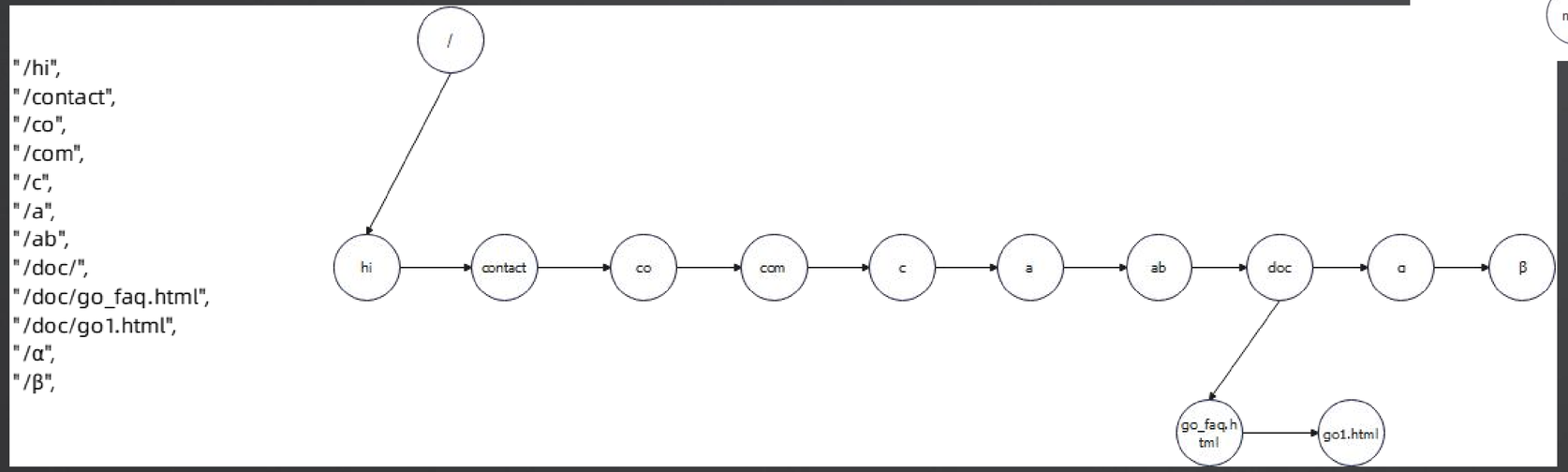
```
node struct {
    kind           kind
    label          byte
    prefix         string
    parent         *node
    staticChildren children
    ppath          string
    pnames         []string
    methodHandler  *methodHandler
    paramChild     *node
    anyChild       *node
    // isLeaf indicates that node doe
    isLeaf bool
    // isHandler indicates that node
    isHandler bool
```

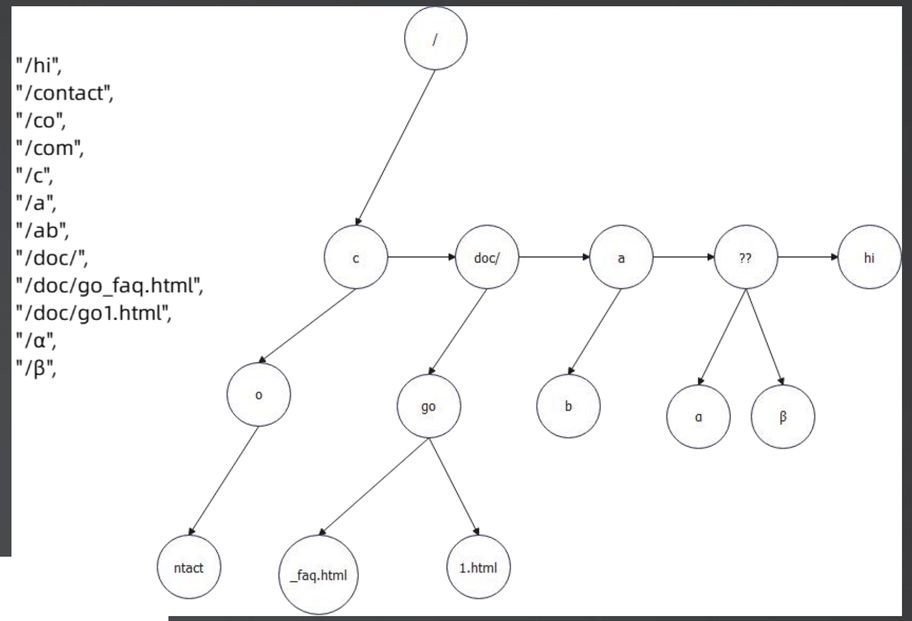
我觉得 Echo 的设计不如 Gin 的设计，也是因为 Echo 多引入了一些抽象，但是实际上它们功能都差不多。

开源，讲究的是一个以简为美；但是工作，讲究刷 KPI 唬人为要。因此开源要克制，而工作要泛滥。

# 路由树 —— 设计总结

- 归根结底就是设计一颗多叉树。
- 同时我们按照 HTTP 方法来组织路由树，每个 HTTP 方法一棵树
- 节点维持住自己的子节点
- 这两种形态的路由树组织，性能差异不大，但是下面的实现要简单很多



# 路由树 —— 全静态匹配

我们利用全静态匹配来构建路由树，后面再考虑重构路由树以支持通配符匹配、参数路由等复杂匹配。

所谓的静态匹配，就是路径的每一段都必须严格相等。

代码在 v2

# 全静态匹配 —— 接口设计

关键类型：

- router：维持住了所有的路由树，它是整个路由注册和查找的总入口。router 里面维护了一个 map，是按照 HTTP 方法来组织路由树的
- node：代表的是节点。它里面有一个 children 的 map 结构，使用 map 结构是为了快速查找到子节点

```go
type router struct {
    // trees 是按照 HTTP 方法来组织的
    // 如 GET => *node
    trees map[string]*node
}

func newRouter() *router {
    return &router{
        trees: map[string]*node{},
    }
}

func (r *router) AddRoute(method string, pa
    panic( v: "implement me")
}

type node struct {
    path string
    // children 子节点
    // 子节点的 path => node
    children map[string]*node
    // handler 命中路由之后执行的逻辑
    handler HandleFunc
```

# 路由树 —— TDD 起步

在有了类型定义之后，我们就可以考虑按照 TDD 的思路，用测试来驱动我们的实现。

注册路由的测试如右图。

我们使用简化版的 TDD，即：

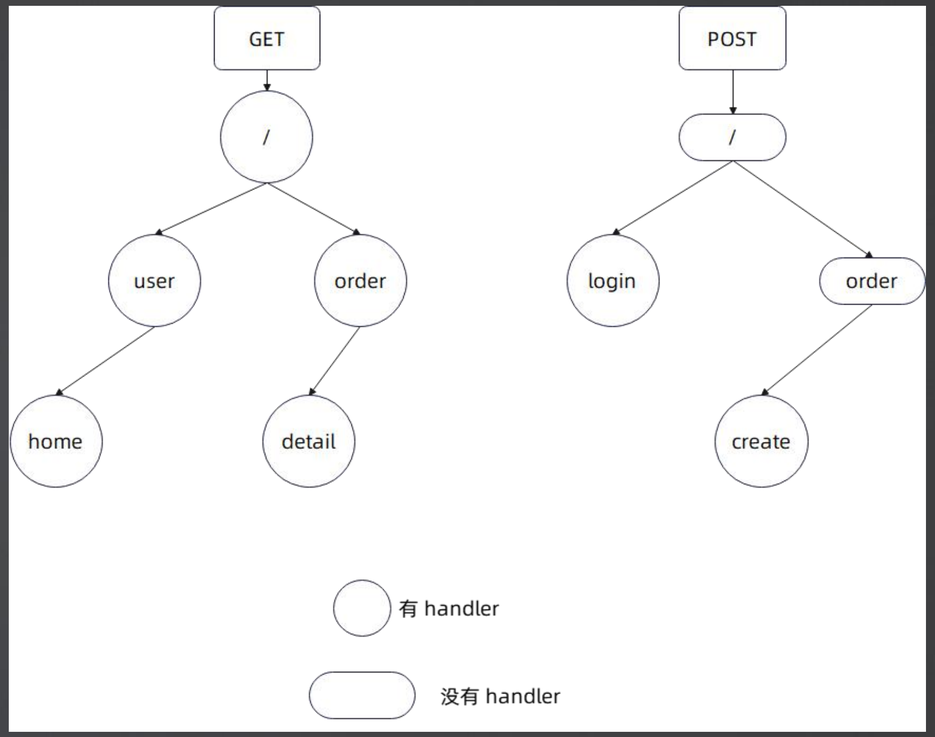
1. 定义 API
2. 定义测试
3. 添加测试用例
4. 实现，并且确保实现能够通过测试用例
5. 重复 3-4 直到考虑了所有的场景
6. 重复步骤 1-5

```go
func Test_router_AddRoute(t *testing.T) {
    testRoutes := []struct{
        method string
        path string
    } {
        {
            method: http.MethodGet,
            path: "/user/home",
        },
    }

    mockHandler := func(ctx *Context) {}
    r := newRouter()
    for _, tr := range testRoutes {
        r.AddRoute(tr.method, tr.path, mockHandler)
    }
```

# 全静态匹配 —— 测试用例

```
{
    method: http.MethodGet,
    path: "/",
},
{
    method: http.MethodGet,
    path: "/user",
},
{
    method: http.MethodGet,
    path: "/user/home",
},
{
    method: http.MethodGet,
    path: "/order/detail",
},
```

```
{
    method: http.MethodPost,
    path: "/order/create",
},
{
    method: http.MethodPost,
    path: "/login",
},
```



# 全静态匹配 —— 非法用例

```
// 空字符串
assert.PanicsWithValue(t, expected: "web: 路由是空字符串", func() {
    r.AddRoute(http.MethodGet, path: "", mockHandler)
})

// 前导没有 /
assert.PanicsWithValue(t, expected: "web: 路由必须以 / 开头", func() {
    r.AddRoute(http.MethodGet, path: "a/b/c", mockHandler)
})

// 后缀有 /
assert.PanicsWithValue(t, expected: "web: 路由不能以 / 结尾", func() {
    r.AddRoute(http.MethodGet, path: "/a/b/c/", mockHandler)
})
```

```
// 根节点重复注册
r.AddRoute(http.MethodGet, path: "/", mockHandler)
assert.PanicsWithValue(t, expected: "web: 路由冲突[/]", func() {
    r.AddRoute(http.MethodGet, path: "/", mockHandler)
})
// 普通节点重复注册
r.AddRoute(http.MethodGet, path: "/a/b/c", mockHandler)
assert.PanicsWithValue(t, expected: "web: 路由冲突[/a/b/c]", func()
    r.AddRoute(http.MethodGet, path: "/a/b/c", mockHandler)
})

// 多个 /
assert.PanicsWithValue(t, expected: "web: 非法路由。不允许使用 //a/b,
    r.AddRoute(http.MethodGet, path: "/a//b", mockHandler)
})
assert.PanicsWithValue(t, expected: "web: 非法路由。不允许使用 //a/b,
    r.AddRoute(http.MethodGet, path: "//a/b", mockHandler)
})
```

# 全静态匹配 —— 实现代码

```go
// AddRoute 注册路由。
// method 是 HTTP 方法
// path 必须以 / 开始并且结尾不能有 /，中间也不允许有连续的 /
func (r *router) AddRoute(method string, path string, handler HandleFunc) {
    if path == "" {
        panic( v: "web: 路由是空字符串")
    }
    if path[0] != '/' {
        panic( v: "web: 路由必须以 / 开头")
    }

    if path != "/" && path[len(path) - 1] == '/' {
        panic( v: "web: 路由不能以 / 结尾")
    }
```

对 path 进行校验

```go
    root, ok := r.trees[method]
    // 这是一个全新的 HTTP 方法，创建根节点
    if !ok {
        // 创建根节点
        root = &node{path: "/"}
        r.trees[method] = root
    }
    if path == "/" {
        if root.handler != nil {
            panic( v: "web: 路由冲突[/]")
        }
        root.handler = handler
        return
    }
```

处理根节点

# 全静态匹配 —— 实现代码

```
segs := strings.Split(path[1:], sep: "/")
// 开始一段段处理
for _, s := range segs {
    if s == "" {
        panic(fmt.Sprintf( format: "web: 非法路由。不允许使用
    }
    root = root.childOrCreate(s)
}
if root.handler != nil {
    panic(fmt.Sprintf( format: "web: 路由冲突[%s]", path))
}
root.handler = handler
```

沿着子节点层层深入下去

```
// childOrCreate 查找子节点，如果子节点不存在就创建一个
// 并且将子节点放回去了 children 中
func (n *node) childOrCreate(path string) *node {
    if n.children == nil {
        n.children = make(map[string]*node)
    }
    child, ok := n.children[path]
    if !ok {
        child = &node{path: path}
        n.children[path] = child
    }
    return child
}
```

查找或者创建一个子节点

# 全静态匹配 —— method 需要校验吗

我们对 path 进行了校验，但是却没有对 method 进行校验。理论上 method 只能是合法的 HTTP 方法。

```go
// Common HTTP methods.
//
// Unless otherwise noted, these are defin
const (
    MethodGet     = "GET"
    MethodHead    = "HEAD"
    MethodPost    = "POST"
    MethodPut     = "PUT"
    MethodPatch   = "PATCH" // RFC 5789
    MethodDelete  = "DELETE"
    MethodConnect = "CONNECT"
    MethodOptions = "OPTIONS"
    MethodTrace   = "TRACE"
)
```

```go
// AddRoute 注册路由。
// method 是 HTTP 方法
// path 必须以 / 开始并且结尾不能有 /，中间也不允许有连续的 /
func (r *router) AddRoute(method string, path string, handler HandleFunc) {
    if path == "" {
        panic( v: "web: 路由是空字符串")
    }
    if path[0] != '/' {
        panic( v: "web: 路由必须以 / 开头")
    }

    if path != "/" && path[len(path) - 1] == '/' {
        panic( v: "web: 路由不能以 / 结尾")
    }
```

# 全静态匹配 —— 将 AddRoute 改成私有的

定义为私有的 addRoute：

- 用户只能通过 Get 或者 Post 方法来注册，那么可以确保 method 参数永远都是对的
- addRoute 在接口里面是私有的，限制了用户将无法实现 Server。实际上如果用户想要实现 Server，就约等于自己实现一个 Web 框架了

path 之所以会有那么强的约束，是因为我希望用户写出来的代码风格要一致，就是注册的路由有些人喜欢加 /，有些人不喜欢加 /。

```go
type Server interface {
    http.Handler
    // Start 启动服务器
    // addr 是监听地址。如果只指定端口，可以使用 ":8081"
    // 或者 "localhost:8082"
    Start(addr string) error

    // addRoute 注册一个路由
    // method 是 HTTP 方法
    // path 是路径，必须以 / 为开头
    addRoute(method string, path string, handler HandleFunc)
    // 我们并不采取这种设计方案
```

```go
func (s *HTTPServer) Post(path string, handler HandleFunc) {
    s.addRoute(http.MethodPost, path, handler)
}

func (s *HTTPServer) Get(path string, handler HandleFunc) {
    s.addRoute(http.MethodGet, path, handler)
}
```

# 全静态匹配 —— 路由查找

```
testRoutes := []struct{
    method string
    path string
} {
    {
        method: http.MethodGet,
        path: "/",
    },
    {
        method: http.MethodGet,
        path: "/user",
    },
    {
        method: http.MethodPost,
        path: "/order/create",
    },
}
```

```
testCases := []struct {
    name string
    method string
    path string
    found bool
    wantNode *node
}{
    {
        name: "method not found",
        method: http.MethodHead,
    },
    {
        name: "path not found",
        method: http.MethodGet,
        path: "/abc",
    },
```

```
    {
        name: "root",
        method: http.MethodGet,
        path: "/",
        found: true,
        wantNode: &node{
            path: "/",
            handler: mockHandler,
        },
    },
    {
        name: "user",
        method: http.MethodGet,
        path: "/user",
        found: true,
        wantNode: &node{
            path: "user",
            handler: mockHandler,
        },
    },
```

```
    {
        name: "no handler",
        method: http.MethodPost,
        path: "/order",
        found: true,
        wantNode: &node{
            path: "order",
        },
    },
    {
        name: "two layer",
        method: http.MethodPost,
        path: "/order/create",
        found: true,
        wantNode: &node{
            path: "create",
            handler: mockHandler,
        },
    },
```

# 全静态匹配 —— 实现代码

```go
// findRoute 查找对应的节点
// 注意，返回的 node 内部 HandleFunc 不为 nil 才算是注册了路由
func (r *router) findRoute(method string, path string) (*node, bool) {
    root, ok := r.trees[method]
    if !ok {
        return nil, false
    }

    if path == "/" {
        return root, true
    }

    segs := strings.Split(strings.Trim(path, cutset: "/"), sep: "/")
    for _, s := range segs {
        root, ok = root.childOf(s)
        if !ok {
            return nil, false
        }
    }
    return root, true
}
```

```go
func (n *node) childOf(path string) (*node, bool) {
    if n.children == nil {
        return nil, false
    }
    res, ok := n.children[path]
    return res, ok
}
```

findRoute 只是返回节点，但是并没有进一步判断究竟有没有 HandleFunc，所以调用者需要进一步检查。

# Server 集成 router

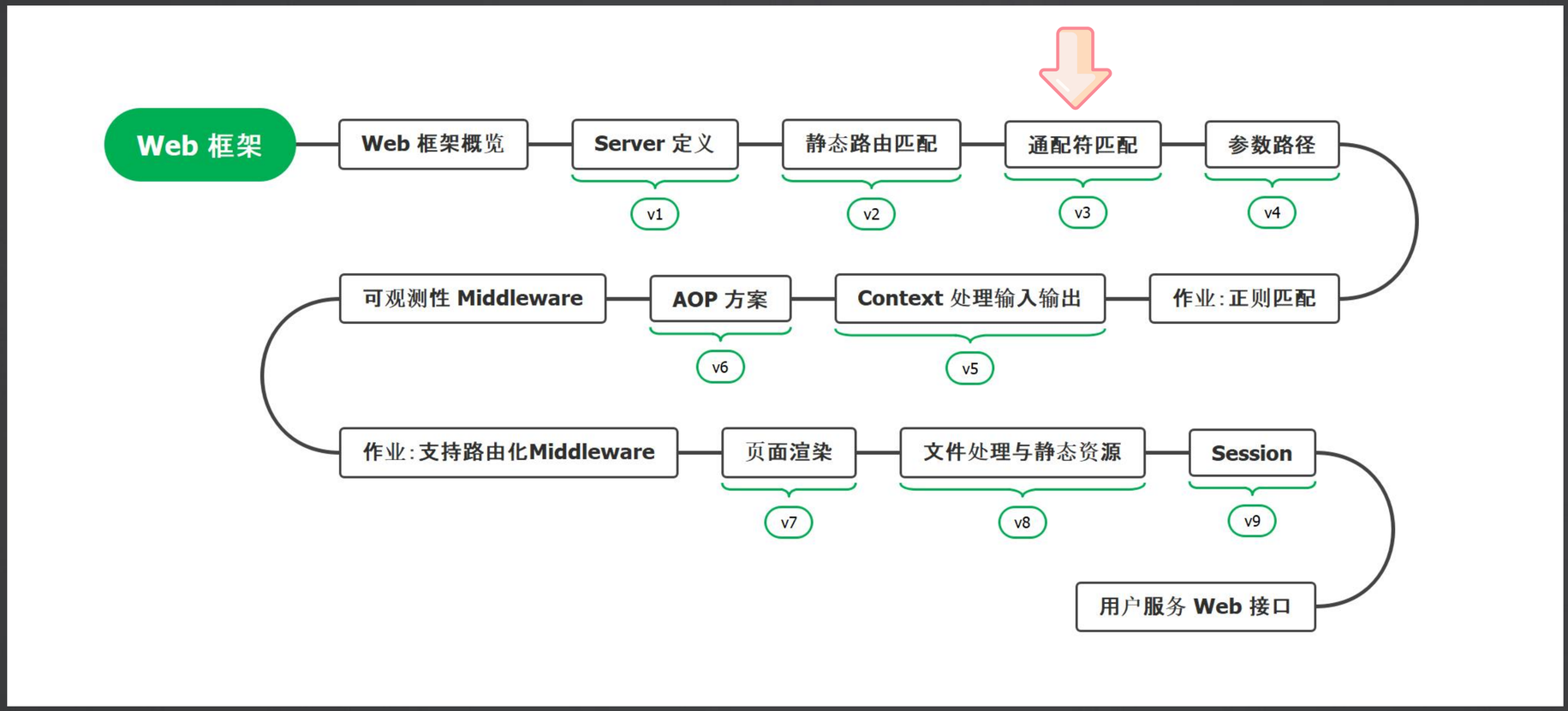
```go
type HTTPServer struct {
    router
}

func NewHTTPServer() *HTTPServer {
    return &HTTPServer{
        router: newRouter(),
    }
}
```

```go
func (s *HTTPServer) serve(ctx *Context) {
    n, ok := s.findRoute(ctx.Req.Method, ctx.Req.URL.Path)
    if !ok || n.handler == nil {
        ctx.Resp.WriteHeader(statusCode: 404)
        ctx.Resp.Write([]byte("Not Found"))
        return
    }
    n.handler(ctx)
}
```

在这种情况下，用户只能使用 NewHTTPServer 来创建服务器实例。

如果考虑到用户可能自己 s := &HTTPServer 引起 panic，那么可以将 HTTPServer 做成私有的，即改名为 httpServer。

# 学习路线—— 通配符匹配


Web 框架
Web 框架概览
Server 定义
静态路由匹配
通配符匹配
参数路径
v1
v2
v3
v4
可观测性 Middleware
AOP 方案
Context 处理输入输出
作业：正则匹配
v6
v5
作业：支持路由化Middleware
页面渲染
文件处理与静态资源
Session
v7
v8
v9
用户服务 Web 接口

# 路由树——通配符匹配

所谓通配符匹配，是指用 * 号来表达匹配任何路径。要考虑几个问题：

- 如果路径是 /a/b/c 能不能命中 /a/* 路由？
- 如果注册了两个路由 /user/123/home 和 /user/*/*。那么输入路径 /user/123/detail 能不能命中 /user/*/*？

这两个都是理论上可以，但是不应该命中。

- 从实现的角度来说，其实并不难。
- 从用户的角度来说，他们不应该设计这种路由。给用户自由，但是也要限制不良实践。
- 后者要求的是一种可回溯的路由匹配，即发现 /user/123/home 匹配不上之后要回溯回去 /user/* 进一步查找，典型的投入大产出低的特性。

# 通配符匹配 —— node 变更

```go
type node struct {
    path string
    // children 子节点
    // 子节点的 path => node
    children map[string]*node
    // handler 命中路由之后执行的逻辑
    handler HandleFunc
}
```

```go
// node 代表路由树的节点
// 路由树的匹配顺序是:
// 1. 静态完全匹配
// 2. 通配符匹配
// 这是不回溯匹配
type node struct {
    path string
    // children 子节点
    // 子节点的 path => node
    children map[string]*node
    // handler 命中路由之后执行的逻辑
    handler HandleFunc

    // 通配符 * 表达的节点，任意匹配
    starChild *node
}
```

# 通配符匹配 —— childOf 变更

```go
func (n *node) childOf(path string) (*node, bool) {
    if n.children == nil {
        return nil, false
    }
    res, ok := n.children[path]
    return res, ok
}
```

```go
func (n *node) childOf(path string) (*node, bool) {
    if n.children == nil {
        return n.starChild, n.starChild != nil
    }
    res, ok := n.children[path]
    if !ok {
        return n.starChild, n.starChild != nil
    }
    return res, ok
}
```

# 通配符匹配 —— childOrCreate 变更

```go
// childOrCreate 查找子节点，如果子节点不存在就创建一个
// 并且将子节点放回去了 children 中
func (n *node) childOrCreate(path string) *node {
    if n.children == nil {
        n.children = make(map[string]*node)
    }
    child, ok := n.children[path]
    if !ok {
        child = &node{path: path}
        n.children[path] = child
    }
    return child
}
```

```go
// childOrCreate 查找子节点，如果子节点不存在就创建一个
// 并且将子节点放回去了 children 中
func (n *node) childOrCreate(path string) *node {
    if path == "*" {
        if n.starChild == nil {
            n.starChild = &node{path: "*"}
        }
        return n.starChild
    }
    if n.children == nil {
        n.children = make(map[string]*node)
    }
    child, ok := n.children[path]
    if !ok {
        child = &node{path: path}
        n.children[path] = child
    }
    return child
}
```

# 通配符匹配 —— addRoute 测试用例

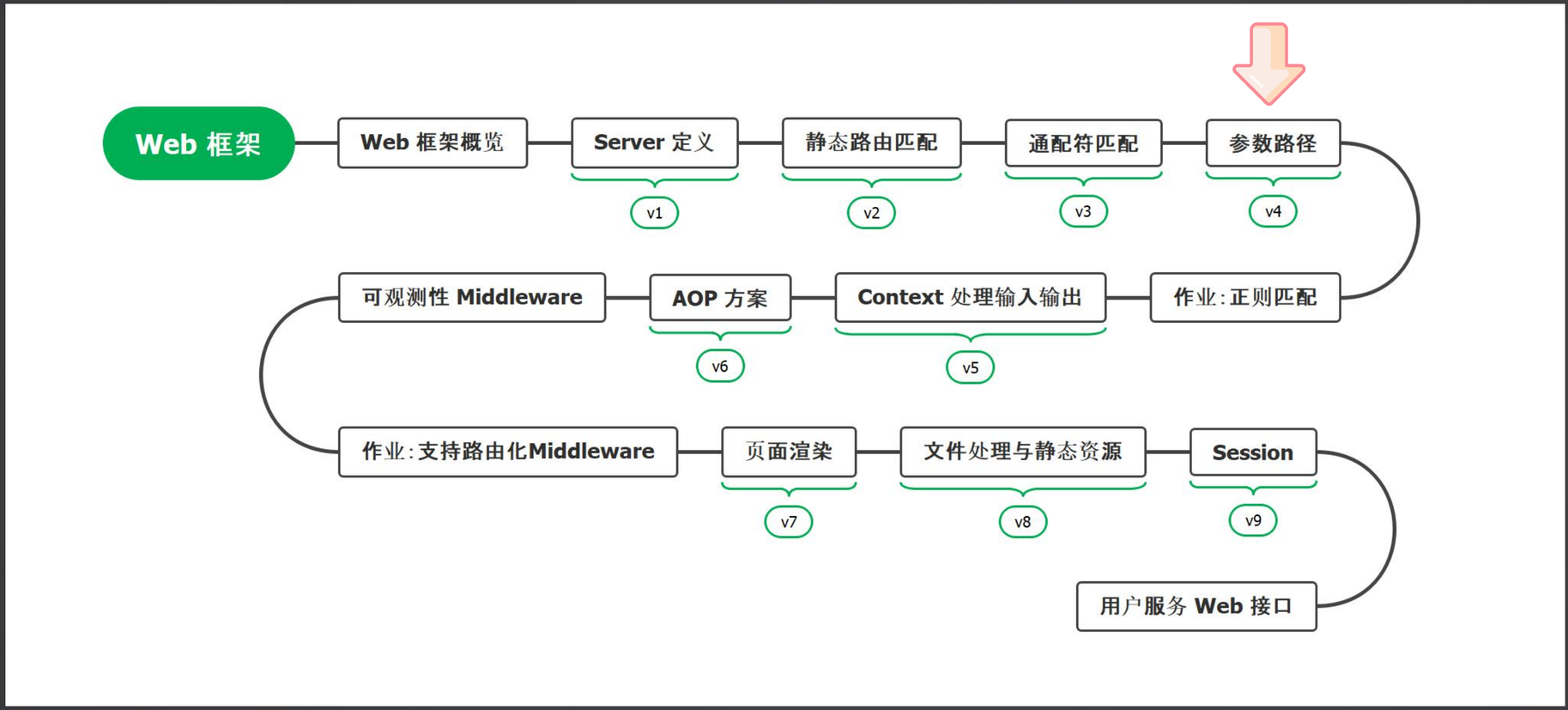
```
// 通配符测试用例
{
    method: http.MethodGet,
    path: "/order/*",
},
{
    method: http.MethodGet,
    path: "/*",
},
{
    method: http.MethodGet,
    path: "/*/*",
},
{
    method: http.MethodGet,
    path: "/*/abc",
},
{
    method: http.MethodGet,
    path: "/*/abc/*",
},
```

# 通配符匹配 —— findRoute 测试用例

```
{
    // 命中/order/*
    name: "star match",
    method: http.MethodPost,
    path: "/order/delete",
    found: true,
    wantNode: &node{
        path: "*",
        handler: mockHandler,
    },
},
{
    // 命中通配符在中间的
    // /user/*/home
    name: "star in middle",
    method: http.MethodGet,
    path: "/user/Tom/home",
    found: true,
    wantNode: &node{
        path: "home",
        handler: mockHandler,
    },
```

```
{
    // 比 /order/* 多了一段
    name: "overflow",
    method: http.MethodPost,
    path: "/order/delete/123",
},
```

# 学习路线—— 参数路径


Web 框架
Web 框架概览
Server 定义
静态路由匹配
通配符匹配
参数路径
v1
v2
v3
v4
可观测性 Middleware
AOP 方案
Context 处理输入输出
作业：正则匹配
v6
v5
作业：支持路由化Middleware
页面渲染
文件处理与静态资源
Session
v7
v8
v9
用户服务 Web 接口

# 路由树 —— 参数路径

所谓参数路径，就是指在路径中带上参数，同时这些参数对应的值可以被业务取出来使用。

例如：/user/:id ，如果输入路径 /user/123，那么会命中这个路由，并且 id = 123。

那么要考虑：

- 允不允许同样的参数路径和通配符匹配一起注册？例如同时注册 /user/* 和 /user/:id
  - 可以，但是没必要，用户也不应该设计这种路由

# 参数路径 —— node 变更

```
type node struct {
    path string
    // children 子节点
    // 子节点的 path => node
    children map[string]*node
    // handler 命中路由之后执行的逻辑
    handler HandleFunc
}
```

```
// node 代表路由树的节点
// 路由树的匹配顺序是：
// 1. 静态完全匹配
// 2. 通配符匹配
// 这是不回溯匹配
type node struct {
    path string
    // children 子节点
    // 子节点的 path => node
    children map[string]*node
    // handler 命中路由之后执行的逻辑
    handler HandleFunc

    // 通配符 * 表达的节点，任意匹配
    starChild *node
}
```

# 参数路径 —— childOf 变更

```go
func (n *node) childOf(path string) (*node, bool) {
    if n.children == nil {
        return n.starChild, n.starChild != nil
    }
    res, ok := n.children[path]
    if !ok {
        return n.starChild, n.starChild != nil
    }
    return res, ok
}
```

```go
func (n *node) childOf(path string) (*node, bool) {
    if n.children == nil {
        if n.paramChild != nil {
            return n.paramChild, true
        }
        return n.starChild, n.starChild != nil
    }
    res, ok := n.children[path]
    if !ok {
        if n.paramChild != nil {
            return n.paramChild, true
        }
        return n.starChild, n.starChild != nil
    }
    return res, ok
}
```

# 参数路径 —— childOrCreate 变更

```go
// childOrCreate 查找子节点，如果子节点不存在就创建一个
// 并且将子节点放回去了 children 中
func (n *node) childOrCreate(path string) *node {
    if path == "*" {
        if n.starChild == nil {
            n.starChild = &node{path: "*"}
        }
        return n.starChild
    }
    if n.children == nil {
        n.children = make(map[string]*node)
    }
    child, ok := n.children[path]
    if !ok {
        child = &node{path: path}
        n.children[path] = child
    }
    return child
}
```

```go
// 如果没有找到，那么会创建一个新的节点，并且保存在 node 里面
func (n *node) childOrCreate(path string) *node {
    if path == "*" {
        if n.paramChild != nil {
            panic(fmt.Sprintf( format: "web: 非法路由，已有
        }
        if n.starChild == nil {
            n.starChild = &node{path: path}
        }
        return n.starChild
    }

    // 以 : 开头，我们认为是参数路由
    if path[0] == ':' {
        if n.starChild != nil {
            panic(fmt.Sprintf( format: "web: 非法路由，已有
        }
        if n.paramChild != nil {
            if n.paramChild.path != path {
                panic(fmt.Sprintf( format: "web: 路由冲突,
            }
        } else {
            n.paramChild = &node{path: path}
        }
```

# 参数路径 —— 获取参数值

在路径匹配上来的时候，要把参数对应的值给带出来。

例如 /user/:id，输入路径 /user/123，那么应该把 id = 123 这个信息带出来。

```go
func (r *router) findRoute(method string, path string) (*matchInfo, bool) {
    root, ok := r.trees[method]
    if !ok {
```

```go
type matchInfo struct {
    n *node
    pathParams map[string]string
}
```

```go
type Context struct {
    Req  *http.Request
    Resp http.ResponseWriter
    PathParams map[string]string
}
```

# 参数路径 —— findRoute

例如/user/:id，输入路径 /user/123，那么就相当于把 id = 123 这样一个键值对放进去了 mi (matchInfo) 里面。

```go
func (r *router) findRoute(method string, path string) (*matchInfo, bool) {
    root, ok := r.trees[method]
    if !ok { return nil, false }

    if path == "/" { return &matchInfo{n: root}, true }

    segs := strings.Split(strings.Trim(path, cutset: "/"), sep: "/")
    mi := &matchInfo{}
    for _, s := range segs {
        var matchParam bool
        root, matchParam, ok = root.childOf(s)
        if !ok { return nil, false }
        if matchParam {
            mi.addValue(root.path[1:], s)
        }
    }
    mi.n = root
    return mi, true
}
```

# 路由树总结 —— 注册路由的注意事项

- 已经注册了的路由，无法被覆盖，例如 /user/home 注册两次，会冲突
- path 必须以 / 开始并且结尾不能有 /，中间也不允许有连续的 /
- 不能在同一个位置注册不同的参数路由，例如 /user/:id 和 /user/:name 冲突
- 不能在同一个位置同时注册通配符路由和参数路由，例如 /user/:id 和 /user/* 冲突
- 同名路径参数，在路由匹配的时候，值会被覆盖，例如 /user/:id/abc/:id，那么 /user/123/abc/456 最终 id = 456

```
// addRoute 注册路由。
// method 是 HTTP 方法
// - 已经注册了的路由，无法被覆盖。例如 /user/home
// - path 必须以 / 开始并且结尾不能有 /，中间也不允
// - 不能在同一个位置注册不同的参数路由，例如 /user/
// - 不能在同一个位置同时注册通配符路由和参数路由，例
// - 同名路径参数，在路由匹配的时候，值会被覆盖。例如
func (r *router) addRoute(method string, path
    if path == "" {
```

把这个放在 addRoute 方法上，那么但凡用户发现 panic 的时候看一眼这个注释，都知道出了什么问题。

最后一条是可以考虑在注册路由的时候强制 panic 的。

# 路由树总结 —— 为什么在注册路由用 panic

俗话说，遇事不决用 error。为什么注册路由的过程我们有一大堆 panic?

这个地方确实可以考虑返回 error。例如 Get 方法，但是这要求用户必须处理返回的 error。

从另外一个角度来说，用户必须要注册完路由，才能启动 HTTPServer。那么我们就可以采用 panic，因为启动之前就代表应用还没运行。

```
func (s *HTTPServer) Get(path string, handler HandleFunc) error {
    return s.addRoute(http.MethodGet, path, handler)
}
```

```
    if path == "" {
        panic( v: "web: 路由是空字符串")
    }
    if path[0] != '/' : "web: 路由必须以 / 开头" *

    if path != "/" && path[len(path) - 1] == '/' {
        panic( v: "web: 路由不能以 / 结尾")
    }
```

```
 if s == "" {
     panic(fmt.Sprintf( format: "web: 非法路由。不允许使用 /,
 }
 root = root.childOrCreate(s)

root.handler != nil {
 panic(fmt.Sprintf( format: "web: 路由冲突[%s]", path))
```

# 路由树总结 —— 路由树是线程安全的吗？

显然不是线程安全的。

我们要求用户必须要注册完路由才能启动 HTTPServer。而正常的用法都是在启动之前依次注册路由，不存在并发场景。

至于运行期间动态注册路由，没必要支持。这是典型的为了解决 1% 的问题，引入 99% 的代码。

# 面试要点（一）

- 路由树算法？核心就是前缀树。前缀的意思就是，两个节点共同的前缀，将会被抽取出来作为父亲节点。在我们的实现里面，是按照 / 来切割，每一段作为一个节点。
- 路由匹配的优先级？本质上这是和 Web 框架相关的。在我们的设计里面是静态匹配 > 路径参数 > 通配符匹配。
- 路由查找会回溯吗？这也是和 Web 框架相关的，我们在课程上是不支持的。在这里可以简单描述可回溯和不可回溯之间的区别，可以是用课程例子 /user/123/home 和 /user/*/* 。我这里不支持是因为这个特性非常鸡肋。
- Web 框架是怎么组织路由树的？一个 HTTP 方法一颗路由树，也可以考虑一颗路由树，每个节点标记自己支持的 HTTP 方法。在课程中可以看到，前者是比较主流的。

# 面试要点（二）

- 路由查找的性能受什么影响？或者说怎么评估路由查找的性能？核心是看路由树的高度，次要因素是路由树的宽度（想想我们的 children 字段）。
- 路由树是线程安全的吗？严格来说也是跟 Web 框架相关的。大多数都不是线程安全的，这是为了性能。所以才要求大家一定要先注册路由，后启动 Web 服务器。如果你有运行期间动态添加路由的需求，只需要利用装饰器模式，就可以将一个线程不安全的封装为线程安全的路由树。
- 具体匹配方式的实现原理。课程上我们讨论了静态匹配、通配符匹配和路径匹配，作业里面要求大家照着实现一个正则匹配。其实核心就是划定优先级，然后一种种匹配方式挨个匹配过去。

# Q & A

# THANKS