httprouter源码分析

结构

```
type Router struct {
  trees map[string]*node

  // Enables automatic redirection if the current route can't be matched but a
  // handler for the path with (without) the trailing slash exists.
  // For example if /foo/ is requested but a route only exists for /foo, the
  // client is redirected to /foo with http status code 301 for GET requests
  // and 307 for all other request methods.
  RedirectTrailingSlash bool

  // If enabled, the router tries to fix the current request path, if no
  // handle is registered for it.
  // First superfluous path elements like ../ or // are removed.
  // Afterwards the router does a case-insensitive lookup of the cleaned path.
  // If a handle can be found for this route, the router makes a redirection
  // to the corrected path with status code 301 for GET requests and 307 for
  // all other request methods.
  // For example /FOO and /..//Foo could be redirected to /foo.
  // RedirectTrailingSlash is independent of this option.
  RedirectFixedPath bool

  // If enabled, the router checks if another method is allowed for the
  // current route, if the current request can not be routed.
  // If this is the case, the request is answered with 'Method Not Allowed'
  // and HTTP status code 405.
  // If no other Method is allowed, the request is delegated to the NotFound
  // handler.
  HandleMethodNotAllowed bool

  // If enabled, the router automatically replies to OPTIONS requests.
  // Custom OPTIONS handlers take priority over automatic replies.
  HandleOPTIONS bool

  // An optional http.Handler that is called on automatic OPTIONS requests.
```

```
    // The handler is only called if HandleOPTIONS is true and no OPTIONS
    // handler for the specific path was set.
    // The "Allowed" header is set before calling the handler.
    GlobalOPTIONS http.Handler

    // Cached value of global (*) allowed methods
    globalAllowed string

    // Configurable http.Handler which is called when no matching route is
    // found. If it is not set, http.NotFound is used.
    NotFound http.Handler

    // Configurable http.Handler which is called when a request
    // cannot be routed and HandleMethodNotAllowed is true.
    // If it is not set, http.Error with http.StatusMethodNotAllowed is used.
    // The "Allow" header with allowed request methods is set before the handler
    // is called.
    MethodNotAllowed http.Handler

    // Function to handle panics recovered from http handlers.
    // It should be used to generate a error page and return the http error code
    // 500 (Internal Server Error).
    // The handler can be used to keep your server from crashing because of
    // unrecovered panics.
    PanicHandler func(http.ResponseWriter, *http.Request, interface{})
}
```

注意上面的`trees map[string]*node`，它是一个关键。

其中，`node`定义如下：

```
type node struct {
    path      string
    wildChild bool
    nType     nodeType
    maxParams uint8
    priority  uint32
    indices   string
    children  []*node
    handle    Handle
}
```

简单来说，`httprouter`的`router`维护了一个`trees(树)`，它是一个map。这个map的`key`就是各种HTTP请求方法，对应的值就是一个`node(节点)`。也就是路由器为每个请求方法管理一个单独的树。

httprouter路由原理

路由器依赖于大量使用通用前缀的树结构，它基本上是一个紧凑的前缀树(或只是基数树)。具有公共前缀的节点也共享一个公共的父节点。

下面来看一个简短的例子，我们定义了如下的路由信息：

```
router := httprouter.New()

router.GET("/search/", func1)
router.GET("/support/", func2)
router.GET("/blog/:post/", func3)
router.GET("/about-us/", func4)
router.GET("/about-us/team/", func5)
router.GET("/contact/", func6)
```

然后演示一下GET请求方法的路由树是什么样子的:

```
Priority   Path             Handle
9          \                *<1>
3          ├s               nil
2          |├earch\         *<2>
1          |└upport\        *<3>
2          ├blog\           *<4>
1          |    └:post      nil
1          |         └\     *<5>
2          ├about-us\       *<6>
1          |        └team\  *<7>
1          └contact\        *<8>
```

上面最右边那一列每个`*<num>`表示Handle处理函数(指针)的内存地址。如果你沿着树从根到叶的路径走，你会得到完整的路径。

例如：`blog/:post`其中`:post`只是实际文章名称的占位符(参数)。与`hash-maps`不同，这种树结构还允许我们使用像`:post`参数这种动态部分，因为我们实际上是根据路由模式进行匹配，而不仅仅是比较哈希值。

由于URL路径具有层次结构，并且只使用有限的一组字符(字节值)，所以很可能有许多常见的前缀。这使我们可以很容易地将路由简化为更小的问题。此外，**路由器为每个请求方法管理一个单独的树**。首先，它比在每个节点中保存一个方法(>句柄映射)更节省空间，它还允许我们在开始查找前缀树之前极大地减少路由问题。

为了获得更好的可伸缩性，每个树级别上的子节点都按`Priority(优先级)`排序，其中优先级（最左列）就是在子节点(子节点、子子节点等等)中注册的句柄的数量。这在两个方面有帮助:

1. 首先计算大部分路由路径中的节点。这有助于使尽可能多的路由尽可能快地到达。
2. 这是一种成本补偿。最长可达路径(最高成本)总是可以先求值。下面的方案显示了树结构。从上到下，从左到右计算节点。

注册路由

```
// addRoute adds a node with the given handle to the path.
// Not concurrency-safe!
func (n *node) addRoute(path string, handle Handle) {
  fullPath := path
  n.priority++
  numParams := countParams(path)

  // non-empty tree
  if len(n.path) > 0 || len(n.children) > 0 {
  walk:
    for {
      // Update maxParams of the current node
      if numParams > n.maxParams {
        n.maxParams = numParams
      }

      // Find the longest common prefix.
      // This also implies that the common prefix contains no ':' or '*'
      // since the existing key can't contain those chars.
      i := 0
      max := min(len(path), len(n.path))
      for i < max && path[i] == n.path[i] {
        i++
      }

      // Split edge
      if i < len(n.path) {
        child := node{
          path:      n.path[i:],
          wildChild: n.wildChild,
          nType:     static,
          indices:   n.indices,
          children:  n.children,
          handle:    n.handle,
          priority:  n.priority - 1,
        }

        // Update maxParams (max of all children)
        for i := range child.children {
          if child.children[i].maxParams > child.maxParams {
            child.maxParams = child.children[i].maxParams
          }
        }

        n.children = []*node{&child}
        // []byte for proper unicode char conversion, see #65
```

```
        n.indices = string([]byte{n.path[i]})
        n.path = path[:i]
        n.handle = nil
        n.wildChild = false
      }

      // Make new node a child of this node
      if i < len(path) {
        path = path[i:]

        if n.wildChild {
          n = n.children[0]
          n.priority++

          // Update maxParams of the child node
          if numParams > n.maxParams {
            n.maxParams = numParams
          }
          numParams--

          // Check if the wildcard matches
          if len(path) >= len(n.path) && n.path == path[:len(n.path)] &&
            // Adding a child to a catchAll is not possible
            n.nType != catchAll &&
            // Check for longer wildcard, e.g. :name and :names
            (len(n.path) >= len(path) || path[len(n.path)] == '/') {
            continue walk
          } else {
            // Wildcard conflict
            var pathSeg string
            if n.nType == catchAll {
              pathSeg = path
            } else {
              pathSeg = strings.SplitN(path, "/", 2)[0]
            }
            prefix := fullPath[:strings.Index(fullPath, pathSeg)] + n.path
            panic("'" + pathSeg +
              "' in new path '" + fullPath +
              "' conflicts with existing wildcard '" + n.path +
              "' in existing prefix '" + prefix +
              "'")
          }
        }

        c := path[0]

        // slash after param
        if n.nType == param && c == '/' && len(n.children) == 1 {
          n = n.children[0]
```

```
            n.priority++
            continue walk
          }

          // Check if a child with the next path byte exists
          for i := 0; i < len(n.indices); i++ {
            if c == n.indices[i] {
              i = n.incrementChildPrio(i)
              n = n.children[i]
              continue walk
            }
          }

          // Otherwise insert it
          if c != ':' && c != '*' {
            // []byte for proper unicode char conversion, see #65
            n.indices += string([]byte{c})
            child := &node{
              maxParams: numParams,
            }
            n.children = append(n.children, child)
            n.incrementChildPrio(len(n.indices) - 1)
            n = child
          }
          n.insertChild(numParams, path, fullPath, handle)
          return

        } else if i == len(path) { // Make node a (in-path) leaf
          if n.handle != nil {
            panic("a handle is already registered for path '" + fullPath + "'")
          }
          n.handle = handle
        }
        return
      }
    } else { // Empty tree
      n.insertChild(numParams, path, fullPath, handle)
      n.nType = root
    }
}
```

```
func (n *node) insertChild(numParams uint8, path, fullPath string, handle
Handle) {
  var offset int // already handled bytes of the path

  // find prefix until first wildcard (beginning with ':'' or '*'')
  for i, max := 0, len(path); numParams > 0; i++ {
```

```
    c := path[i]
    if c != ':' && c != '*' {
      continue
    }

    // find wildcard end (either '/' or path end)
    end := i + 1
    for end < max && path[end] != '/' {
      switch path[end] {
      // the wildcard name must not contain ':' and '*'
      case ':', '*':
        panic("only one wildcard per path segment is allowed, has: '" +
          path[i:] + "' in path '" + fullPath + "'")
      default:
        end++
      }
    }

    // check if this Node existing children which would be
    // unreachable if we insert the wildcard here
    if len(n.children) > 0 {
      panic("wildcard route '" + path[i:end] +
        "' conflicts with existing children in path '" + fullPath + "'")
    }

    // check if the wildcard has a name
    if end-i < 2 {
      panic("wildcards must be named with a non-empty name in path '" +
fullPath + "'")
    }

    if c == ':' { // param
      // split path at the beginning of the wildcard
      if i > 0 {
        n.path = path[offset:i]
        offset = i
      }

      child := &node{
        nType:     param,
        maxParams: numParams,
      }
      n.children = []*node{child}
      n.wildChild = true
      n = child
      n.priority++
      numParams--

      // if the path doesn't end with the wildcard, then there
```

```
      // will be another non-wildcard subpath starting with '/'
      if end < max {
        n.path = path[offset:end]
        offset = end

        child := &node{
          maxParams: numParams,
          priority:  1,
        }
        n.children = []*node{child}
        n = child
      }

    } else { // catchAll
      if end != max || numParams > 1 {
        panic("catch-all routes are only allowed at the end of the path in path
'" + fullPath + "'")
      }

      if len(n.path) > 0 && n.path[len(n.path)-1] == '/' {
        panic("catch-all conflicts with existing handle for the path segment
root in path '" + fullPath + "'")
      }

      // currently fixed width 1 for '/'
      i--
      if path[i] != '/' {
        panic("no / before catch-all in path '" + fullPath + "'")
      }

      n.path = path[offset:i]

      // first node: catchAll node with empty path
      child := &node{
        wildChild: true,
        nType:     catchAll,
        maxParams: 1,
      }
      // update maxParams of the parent node
      if n.maxParams < 1 {
        n.maxParams = 1
      }
      n.children = []*node{child}
      n.indices = string(path[i])
      n = child
      n.priority++

      // second node: node holding the variable
      child = &node{
```

```
        path:      path[i:],
        nType:     catchAll,
        maxParams: 1,
        handle:    handle,
        priority:  1,
      }
      n.children = []*node{child}

      return
    }
  }

  // insert remaining path part and handle to the leaf
  n.path = path[offset:]
  n.handle = handle
}
```

匹配路由

```
// Returns the handle registered with the given path (key). The values of
// wildcards are saved to a map.
// If no handle can be found, a TSR (trailing slash redirect) recommendation is
// made if a handle exists with an extra (without the) trailing slash for the
// given path.
func (n *node) getValue(path string) (handle Handle, p Params, tsr bool) {
walk: // outer loop for walking the tree
  for {
    if len(path) > len(n.path) {
      if path[:len(n.path)] == n.path {
        path = path[len(n.path):]
        // If this node does not have a wildcard (param or catchAll)
        // child,  we can just look up the next child node and continue
        // to walk down the tree
        if !n.wildChild {
          c := path[0]
          for i := 0; i < len(n.indices); i++ {
            if c == n.indices[i] {
              n = n.children[i]
              continue walk
            }
          }

          // Nothing found.
          // We can recommend to redirect to the same URL without a
          // trailing slash if a leaf exists for that path.
          tsr = (path == "/" && n.handle != nil)
```

```
            return

        }

        // handle wildcard child
        n = n.children[0]
        switch n.nType {
        case param:
            // find param end (either '/' or path end)
            end := 0
            for end < len(path) && path[end] != '/' {
                end++
            }

            // save param value
            if p == nil {
                // lazy allocation
                p = make(Params, 0, n.maxParams)
            }
            i := len(p)
            p = p[:i+1] // expand slice within preallocated capacity
            p[i].Key = n.path[1:]
            p[i].Value = path[:end]

            // we need to go deeper!
            if end < len(path) {
                if len(n.children) > 0 {
                    path = path[end:]
                    n = n.children[0]
                    continue walk
                }

                // ... but we can't
                tsr = (len(path) == end+1)
                return
            }

            if handle = n.handle; handle != nil {
                return
            } else if len(n.children) == 1 {
                // No handle found. Check if a handle for this path + a
                // trailing slash exists for TSR recommendation
                n = n.children[0]
                tsr = (n.path == "/" && n.handle != nil)
            }

            return

        case catchAll:
```

```
        // save param value
        if p == nil {
          // lazy allocation
          p = make(Params, 0, n.maxParams)
        }
        i := len(p)
        p = p[:i+1] // expand slice within preallocated capacity
        p[i].Key = n.path[2:]
        p[i].Value = path

        handle = n.handle
        return

      default:
        panic("invalid node type")
      }
    }
  } else if path == n.path {
    // We should have reached the node containing the handle.
    // Check if this node has a handle registered.
    if handle = n.handle; handle != nil {
      return
    }

    if path == "/" && n.wildChild && n.nType != root {
      tsr = true
      return
    }

    // No handle found. Check if a handle for this path + a
    // trailing slash exists for trailing slash recommendation
    for i := 0; i < len(n.indices); i++ {
      if n.indices[i] == '/' {
        n = n.children[i]
        tsr = (len(n.path) == 1 && n.handle != nil) ||
          (n.nType == catchAll && n.children[0].handle != nil)
        return
      }
    }

    return
  }

  // Nothing found. We can recommend to redirect to the same URL with an
  // extra trailing slash if a leaf exists for that path
  tsr = (path == "/") ||
    (len(n.path) == len(path)+1 && n.path[len(path)] == '/' &&
      path == n.path[:len(n.path)-1] && n.handle != nil)
  return
```

```
    }
}
```

参考链接