[toc]

demo

```
// demo
pingHandler := func(w http.ResponseWriter, r *http.Request, _
httprouter.Params) {
    fmt.Fprint(w, "pong")
}

router := httprouter.New()
router.GET("/ping", pingHandler)
log.Fatal(http.ListenAndServe(":8080", router))
```

httprouter

httprouter相当于net/http的ServeMux，基于radix tree/prefix tree提升了路由性能，提供注册、路由方法即可

struct

路由注册与分发

```
// Router is a http.Handler which can be used to dispatch requests to
different
// handler functions via configurable routes
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

        // 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{})
}
```

sign

```
// Handle is a function that can be registered to a route to handle HTTP
// requests. Like http.HandlerFunc, but has a third parameter for the
```

```
values of
// wildcards (variables).
type Handle func(http.ResponseWriter, *http.Request, Params)
```

named参数

```
// Param is a single URL parameter, consisting of a key and a value.
type Param struct {
        Key   string
        Value string
}

// Params is a Param-slice, as returned by the router.
// The slice is ordered, the first URL parameter is also the first slice
value.
// It is therefore safe to read values by the index.
type Params []Param

// ByName returns the value of the first Param which key matches the given
name.
// If no matching Param is found, an empty string is returned.
func (ps Params) ByName(name string) string {
        for i := range ps {
                if ps[i].Key == name {
                        return ps[i].Value
                }
        }
        return ""
}
```

register

```
// GET is a shortcut for router.Handle("GET", path, handle)
func (r *Router) GET(path string, handle Handle) {
        r.Handle("GET", path, handle)
}
```

```
// Handle registers a new request handle with the given path and method.
//
// For GET, POST, PUT, PATCH and DELETE requests the respective shortcut
// functions can be used.
//
// This function is intended for bulk loading and to allow the usage of
less
// frequently used, non-standardized or custom methods (e.g. for internal
// communication with a proxy).
```

```
func (r *Router) Handle(method, path string, handle Handle) {
        if path[0] != '/' {
                panic("path must begin with '/' in path '" + path + "'")
        }

        if r.trees == nil {
                r.trees = make(map[string]*node)
        }

        root := r.trees[method]
        if root == nil {
                root = new(node)
                r.trees[method] = root
        }

        root.addRoute(path, handle)
}

// 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)] &&
                                                // 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,
                        }
                        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
}
```

兼容http.HandlerFunc和http.Handler，封装一下然后构造Handle，把参数放到ParamsKey里面了

```
// HandlerFunc is an adapter which allows the usage of an http.HandlerFunc as a
// request handle.
func (r *Router) HandlerFunc(method, path string, handler http.HandlerFunc) {
        r.Handler(method, path, handler)
}

// Handler is an adapter which allows the usage of an http.Handler as a
// request handle. With go 1.7+, the Params will be available in the
// request context under ParamsKey.
func (r *Router) Handler(method, path string, handler http.Handler) {
        r.Handle(method, path,
                func(w http.ResponseWriter, req *http.Request, p Params) {
                        ctx := req.Context()
                        ctx = context.WithValue(ctx, ParamsKey, p)
                        req = req.WithContext(ctx)
                        handler.ServeHTTP(w, req)
                },
        )
}
```

run

同net/http

router

net/http最终会调用Router.ServeHTTP，作用同ServeMux.ServeHTTP，查找到handler，然后执行

```
// ServeHTTP makes the router implement the http.Handler interface.
func (r *Router) ServeHTTP(w http.ResponseWriter, req *http.Request) {
        if r.PanicHandler != nil {
                defer r.recv(w, req)
        }

        path := req.URL.Path

        if root := r.trees[req.Method]; root != nil {
                if handle, ps, tsr := root.getValue(path); handle != nil {
                        handle(w, req, ps)
                        return
                } else if req.Method != "CONNECT" && path != "/" {
                        code := 301 // Permanent redirect, request with GET method
                        if req.Method != "GET" {
                                // Temporary redirect, request with same method
                                // As of Go 1.3, Go does not support status code 308.
                                code = 307
                        }

                        if tsr && r.RedirectTrailingSlash {
                                if len(path) > 1 && path[len(path)-1] == '/' {
                                        req.URL.Path = path[:len(path)-1]
                                } else {
                                        req.URL.Path = path + "/"
                                }
                                http.Redirect(w, req, req.URL.String(), code)
                                return
                        }

                        // Try to fix the request path
                        if r.RedirectFixedPath {
                                fixedPath, found := root.findCaseInsensitivePath(
                                        CleanPath(path),
					r.RedirectTrailingSlash,
				)
				if found {
					req.URL.Path = string(fixedPath)
					http.Redirect(w, req,
req.URL.String(), code)
					return
				}
			}
		}
	}

	if req.Method == "OPTIONS" && r.HandleOPTIONS {
		// Handle OPTIONS requests
		if allow := r.allowed(path, req.Method); len(allow) > 0 {
			w.Header().Set("Allow", allow)
			return
		}
	} else {
		// Handle 405
		if r.HandleMethodNotAllowed {
			if allow := r.allowed(path, req.Method);
len(allow) > 0 {
				w.Header().Set("Allow", allow)
				if r.MethodNotAllowed != nil {
					r.MethodNotAllowed.ServeHTTP(w,
req)
				} else {
					http.Error(w,

http.StatusText(http.StatusMethodNotAllowed),

http.StatusMethodNotAllowed,
					)
				}
				return
			}
		}
	}

	// Handle 404
	if r.NotFound != nil {
		r.NotFound.ServeHTTP(w, req)
	} else {
		http.NotFound(w, req)
	}
}
```

```
// handler sign
type Handle func(http.ResponseWriter, *http.Request, Params)
```

```
// register
func (r *Router) GET(path string, handle Handle) {
        r.Handle("GET", path, handle)
}

// http.HandlerFunc
func (r *Router) HandlerFunc(method, path string, handler
http.HandlerFunc) {
        r.Handler(method, path, handler)
}
func (r *Router) Handler(method, path string, handler http.Handler) {
        r.Handle(method, path,
                func(w http.ResponseWriter, req *http.Request, p Params) {
                        ctx := req.Context()
                        ctx = context.WithValue(ctx, ParamsKey, p)
                        req = req.WithContext(ctx)
                        handler.ServeHTTP(w, req)
                },
        )
}

func (r *Router) Handle(method, path string, handle Handle) {
        if path[0] != '/' {
                panic("path must begin with '/' in path '" + path + "'")
        }

        if r.trees == nil {
                r.trees = make(map[string]*node)
        }

        root := r.trees[method]
        if root == nil {
                root = new(node)
                r.trees[method] = root
        }

        root.addRoute(path, handle)
}
```

```
// router
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)] &&
                                                // 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,
                        }
                        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
}
```

```
// serve
func (r *Router) ServeHTTP(w http.ResponseWriter, req *http.Request) {
	if r.PanicHandler != nil {
		defer r.recv(w, req)
	}

	path := req.URL.Path

	if root := r.trees[req.Method]; root != nil {
		if handle, ps, tsr := root.getValue(path); handle != nil {
			handle(w, req, ps)
			return
		} else if req.Method != "CONNECT" && path != "/" {
			code := 301 // Permanent redirect, request with GET method
			if req.Method != "GET" {
				// Temporary redirect, request with same method
				// As of Go 1.3, Go does not support status code 308.
				code = 307
			}

			if tsr && r.RedirectTrailingSlash {
				if len(path) > 1 && path[len(path)-1] == '/' {
					req.URL.Path = path[:len(path)-1]
				} else {
					req.URL.Path = path + "/"
				}
				http.Redirect(w, req, req.URL.String(), code)
				return
			}

			// Try to fix the request path
			if r.RedirectFixedPath {
				fixedPath, found := root.findCaseInsensitivePath(
					CleanPath(path),
					r.RedirectTrailingSlash,
				)
				if found {
					req.URL.Path = string(fixedPath)
					http.Redirect(w, req, req.URL.String(), code)
					return
				}
			}
		}
	}

	if req.Method == "OPTIONS" && r.HandleOPTIONS {
```

```
                // Handle OPTIONS requests
                if allow := r.allowed(path, req.Method); len(allow) > 0 {
                        w.Header().Set("Allow", allow)
                        return
                }
        } else {
                // Handle 405
                if r.HandleMethodNotAllowed {
                        if allow := r.allowed(path, req.Method);
len(allow) > 0 {
                                w.Header().Set("Allow", allow)
                                if r.MethodNotAllowed != nil {
                                        r.MethodNotAllowed.ServeHTTP(w,
req)
                                } else {
                                        http.Error(w,

http.StatusText(http.StatusMethodNotAllowed),

http.StatusMethodNotAllowed,
                                        )
                                }
                                return
                        }
                }
        }

        // Handle 404
        if r.NotFound != nil {
                r.NotFound.ServeHTTP(w, req)
        } else {
                http.NotFound(w, req)
        }
}
```