



Fast HTTP implementation for Go.

## fasthttp might not be for you!

fasthttp was designed for some high performance edge cases. **Unless** your server/client needs to handle **thousands of small to medium requests per second** and needs a consistent low millisecond response time fasthttp might not be for you. **For most cases [net/http](#) is much better** as it's easier to use and can handle more cases. For most cases you won't even notice the performance difference.

## General info and links

Currently fasthttp is successfully used by VertaMedia in a production serving up to 200K rps from more than 1.5M concurrent keep-alive connections per physical server.

[TechEmpower Benchmark round 19 results](#)

[Server Benchmarks](#)

[Client Benchmarks](#)

[Install](#)

[Documentation](#)

[Examples from docs](#)

---

Code examples

Awesome fasthttp tools

Switching from net/http to fasthttp

Fasthttp best practices

Tricks with byte buffers

Related projects

FAQ

## HTTP server performance comparison with net/http

In short, fasthttp server is up to 10 times faster than net/http. Below are benchmark results.

*GOMAXPROCS=1*

net/http server:

```
1 $ GOMAXPROCS=1 go test -bench=NetHTTPServerGet -benchmem -benchtime=10s
2 BenchmarkNetHTTPServerGet1ReqPerConn      1000000
   12052 ns/op      2297 B/op      29 allocs/op
3 BenchmarkNetHTTPServerGet2ReqPerConn      1000000
   12278 ns/op      2327 B/op      24 allocs/op
4 BenchmarkNetHTTPServerGet10ReqPerConn     2000000
    8903 ns/op      2112 B/op      19 allocs/op
5 BenchmarkNetHTTPServerGet10KReqPerConn    2000000
    8451 ns/op      2058 B/op      18 allocs/op
6 BenchmarkNetHTTPServerGet1ReqPerConn10KClients 500000
   26733 ns/op      3229 B/op      29 allocs/op
7 BenchmarkNetHTTPServerGet2ReqPerConn10KClients 1000000
   23351 ns/op      3211 B/op      24 allocs/op
8 BenchmarkNetHTTPServerGet10ReqPerConn10KClients 1000000
   13390 ns/op      2483 B/op      19 allocs/op
9 BenchmarkNetHTTPServerGet100ReqPerConn10KClients 1000000
   13484 ns/op      2171 B/op      18 allocs/op
```

fasthttp server:

```
1 $ GOMAXPROCS=1 go test -bench=kServerGet -benchmem -benchtime=10s
2 BenchmarkServerGet1ReqPerConn      10000000
   1559 ns/op      0 B/op      0 allocs/op
3 BenchmarkServerGet2ReqPerConn      10000000
   1248 ns/op      0 B/op      0 allocs/op
4 BenchmarkServerGet10ReqPerConn     20000000
    797 ns/op      0 B/op      0 allocs/op
5 BenchmarkServerGet10KReqPerConn    20000000
    716 ns/op      0 B/op      0 allocs/op
```

---

6	BenchmarkServerGet1ReqPerConn10KClients	10000000
	1974 ns/op	0 B/op
7	BenchmarkServerGet2ReqPerConn10KClients	10000000
	1352 ns/op	0 B/op
8	BenchmarkServerGet10ReqPerConn10KClients	20000000
	789 ns/op	2 B/op
9	BenchmarkServerGet100ReqPerConn10KClients	20000000
	604 ns/op	0 B/op

GOMAXPROCS=4

net/http server:

1	\$ GOMAXPROCS=4 go test -bench=NetHTTPServerGet -benchmem -benchtime=10s	
2	BenchmarkNetHTTPServerGet1ReqPerConn-4	3000000
	4529 ns/op	2389 B/op
		29 allocs/op
3	BenchmarkNetHTTPServerGet2ReqPerConn-4	5000000
	3896 ns/op	2418 B/op
		24 allocs/op
4	BenchmarkNetHTTPServerGet10ReqPerConn-4	5000000
	3145 ns/op	2160 B/op
		19 allocs/op
5	BenchmarkNetHTTPServerGet10KReqPerConn-4	5000000
	3054 ns/op	2065 B/op
		18 allocs/op
6	BenchmarkNetHTTPServerGet1ReqPerConn10KClients-4	1000000
	10321 ns/op	3710 B/op
		30 allocs/op
7	BenchmarkNetHTTPServerGet2ReqPerConn10KClients-4	2000000
	7556 ns/op	3296 B/op
		24 allocs/op
8	BenchmarkNetHTTPServerGet10ReqPerConn10KClients-4	5000000
	3905 ns/op	2349 B/op
		19 allocs/op
9	BenchmarkNetHTTPServerGet100ReqPerConn10KClients-4	5000000
	3435 ns/op	2130 B/op
		18 allocs/op

fasthttp server:

1	\$ GOMAXPROCS=4 go test -bench=kServerGet -benchmem -benchtime=10s	
2	BenchmarkServerGet1ReqPerConn-4	10000000
	1141 ns/op	0 B/op
		0 allocs/op
3	BenchmarkServerGet2ReqPerConn-4	20000000
	707 ns/op	0 B/op
		0 allocs/op
4	BenchmarkServerGet10ReqPerConn-4	30000000
	341 ns/op	0 B/op
		0 allocs/op
5	BenchmarkServerGet10KReqPerConn-4	50000000
	310 ns/op	0 B/op
		0 allocs/op
6	BenchmarkServerGet1ReqPerConn10KClients-4	10000000
	1119 ns/op	0 B/op
		0 allocs/op
7	BenchmarkServerGet2ReqPerConn10KClients-4	20000000
	644 ns/op	0 B/op
		0 allocs/op
8	BenchmarkServerGet10ReqPerConn10KClients-4	30000000
	346 ns/op	0 B/op
		0 allocs/op
9	BenchmarkServerGet100ReqPerConn10KClients-4	50000000
	282 ns/op	0 B/op
		0 allocs/op

---

## HTTP client comparison with net/http

In short, fasthttp client is up to 10 times faster than net/http. Below are benchmark results.

GOMAXPROCS=1

net/http client:

```
1 $ GOMAXPROCS=1 go test -bench='HTTPClient(Do|GetEndToEnd)' -benchmem -
  benchtime=10s
2 BenchmarkNetHTTPClientDoFastServer                1000000
   12567 ns/op      2616 B/op      35 allocs/op
3 BenchmarkNetHTTPClientGetEndToEnd1TCP              200000
   67030 ns/op      5028 B/op     56 allocs/op
4 BenchmarkNetHTTPClientGetEndToEnd10TCP             300000
   51098 ns/op      5031 B/op     56 allocs/op
5 BenchmarkNetHTTPClientGetEndToEnd100TCP            300000
   45096 ns/op      5026 B/op     55 allocs/op
6 BenchmarkNetHTTPClientGetEndToEnd1Inmemory          500000
   24779 ns/op      5035 B/op     57 allocs/op
7 BenchmarkNetHTTPClientGetEndToEnd10Inmemory         1000000
   26425 ns/op      5035 B/op     57 allocs/op
8 BenchmarkNetHTTPClientGetEndToEnd100Inmemory        500000
   28515 ns/op      5045 B/op     57 allocs/op
9 BenchmarkNetHTTPClientGetEndToEnd1000Inmemory       500000
   39511 ns/op      5096 B/op     56 allocs/op
```

fasthttp client:

```
1 $ GOMAXPROCS=1 go test -bench='kClient(Do|GetEndToEnd)' -benchmem -
  benchtime=10s
2 BenchmarkClientDoFastServer                20000000
    865 ns/op      0 B/op      0 allocs/op
3 BenchmarkClientGetEndToEnd1TCP             1000000
   18711 ns/op      0 B/op      0 allocs/op
4 BenchmarkClientGetEndToEnd10TCP            1000000
   14664 ns/op      0 B/op      0 allocs/op
5 BenchmarkClientGetEndToEnd100TCP           1000000
   14043 ns/op      1 B/op      0 allocs/op
6 BenchmarkClientGetEndToEnd1Inmemory         5000000
    3965 ns/op      0 B/op      0 allocs/op
7 BenchmarkClientGetEndToEnd10Inmemory        3000000
    4060 ns/op      0 B/op      0 allocs/op
8 BenchmarkClientGetEndToEnd100Inmemory       5000000
    3396 ns/op      0 B/op      0 allocs/op
9 BenchmarkClientGetEndToEnd1000Inmemory     5000000
    3306 ns/op      2 B/op      0 allocs/op
```

GOMAXPROCS=4

net/http client:

---

```

1 $ GOMAXPROCS=4 go test -bench='HTTPClient(Do|GetEndToEnd)' -benchmem -
  benchtime=10s
2 BenchmarkNetHTTPClientDoFastServer-4                20000000
   8774 ns/op      2619 B/op      35 allocs/op
3 BenchmarkNetHTTPClientGetEndToEnd1TCP-4             5000000
  22951 ns/op      5047 B/op     56 allocs/op
4 BenchmarkNetHTTPClientGetEndToEnd10TCP-4            10000000
  19182 ns/op      5037 B/op     55 allocs/op
5 BenchmarkNetHTTPClientGetEndToEnd100TCP-4           10000000
  16535 ns/op      5031 B/op     55 allocs/op
6 BenchmarkNetHTTPClientGetEndToEnd1Inmemory-4        10000000
  14495 ns/op      5038 B/op     56 allocs/op
7 BenchmarkNetHTTPClientGetEndToEnd10Inmemory-4       10000000
  10237 ns/op      5034 B/op     56 allocs/op
8 BenchmarkNetHTTPClientGetEndToEnd100Inmemory-4      10000000
  10125 ns/op      5045 B/op     56 allocs/op
9 BenchmarkNetHTTPClientGetEndToEnd1000Inmemory-4     10000000
  11132 ns/op      5136 B/op     56 allocs/op

```

fasthttp client:

```

1 $ GOMAXPROCS=4 go test -bench='kClient(Do|GetEndToEnd)' -benchmem -
  benchtime=10s
2 BenchmarkClientDoFastServer-4                      500000000
   397 ns/op         0 B/op         0 allocs/op
3 BenchmarkClientGetEndToEnd1TCP-4                  20000000
  7388 ns/op         0 B/op         0 allocs/op
4 BenchmarkClientGetEndToEnd10TCP-4                 20000000
  6689 ns/op         0 B/op         0 allocs/op
5 BenchmarkClientGetEndToEnd100TCP-4                30000000
  4927 ns/op         1 B/op         0 allocs/op
6 BenchmarkClientGetEndToEnd1Inmemory-4             100000000
  1604 ns/op         0 B/op         0 allocs/op
7 BenchmarkClientGetEndToEnd10Inmemory-4            100000000
  1458 ns/op         0 B/op         0 allocs/op
8 BenchmarkClientGetEndToEnd100Inmemory-4           100000000
  1329 ns/op         0 B/op         0 allocs/op
9 BenchmarkClientGetEndToEnd1000Inmemory-4          100000000
  1316 ns/op         5 B/op         0 allocs/op

```

## Install

```

1 go get -u github.com/valyala/fasthttp

```

---

## Switching from net/http to fasthttp

Unfortunately, fasthttp doesn't provide API identical to net/http. See the FAQ for details. There is net/http -> fasthttp handler converter, but it is better to write fasthttp request handlers by hand in order to use all of the fasthttp advantages (especially high performance :)).

Important points:

- Fasthttp works with RequestHandler functions instead of objects implementing Handler interface. Fortunately, it is easy to pass bound struct methods to fasthttp:

```
1  type MyHandler struct {
2      foobar string
3  }
4
5  // request handler in net/http style, i.e. method bound to
   MyHandler struct.
6  func (h *MyHandler) HandleFastHTTP(ctx *fasthttp.RequestCtx) {
7      // notice that we may access MyHandler properties here - see h.
       foobar.
8      fmt.Fprintf(ctx, "Hello, world! Requested path is %q. Foobar is
       %q",
9          ctx.Path(), h.foobar)
10 }
11
12 // request handler in fasthttp style, i.e. just plain function.
13 func fastHTTPHandler(ctx *fasthttp.RequestCtx) {
14     fmt.Fprintf(ctx, "Hi there! RequestURI is %q", ctx.RequestURI())
15 }
16
17 // pass bound struct method to fasthttp
18 myHandler := &MyHandler{
19     foobar: "foobar",
20 }
21 fasthttp.ListenAndServe(":8080", myHandler.HandleFastHTTP)
22
23 // pass plain function to fasthttp
24 fasthttp.ListenAndServe(":8081", fastHTTPHandler)
```

- The RequestHandler accepts only one argument - RequestCtx. It contains all the functionality required for http request processing and response writing. Below is an example of a simple request handler conversion from net/http to fasthttp.

```
1  // net/http request handler
2  requestHandler := func(w http.ResponseWriter, r *http.Request) {
3      switch r.URL.Path {
4      case "/foo":
5          fooHandler(w, r)
6      case "/bar":
```

---

```

7     barHandler(w, r)
8     default:
9         http.Error(w, "Unsupported path", http.StatusNotFound)
10 }
11 }
```

```

1 // the corresponding fasthttp request handler
2 requestHandler := func(ctx *fasthttp.RequestCtx) {
3     switch string(ctx.Path()) {
4     case "/foo":
5         fooHandler(ctx)
6     case "/bar":
7         barHandler(ctx)
8     default:
9         ctx.Error("Unsupported path", fasthttp.StatusNotFound)
10 }
11 }
```

- Fasthttp allows setting response headers and writing response body in an arbitrary order. There is no ‘headers first, then body’ restriction like in net/http. The following code is valid for fasthttp:

```

1 requestHandler := func(ctx *fasthttp.RequestCtx) {
2     // set some headers and status code first
3     ctx.SetContentType("foo/bar")
4     ctx.SetStatusCode(fasthttp.StatusOK)
5
6     // then write the first part of body
7     fmt.Fprintf(ctx, "this is the first part of body\n")
8
9     // then set more headers
10    ctx.Response.Header.Set("Foo-Bar", "baz")
11
12    // then write more body
13    fmt.Fprintf(ctx, "this is the second part of body\n")
14
15    // then override already written body
16    ctx.SetBody([]byte("this is completely new body contents"))
17
18    // then update status code
19    ctx.SetStatusCode(fasthttp.StatusNotFound)
20
21    // basically, anything may be updated many times before
22    // returning from RequestHandler.
23    //
24    // Unlike net/http fasthttp doesn't put response to the wire
25    // until
26    // returning from RequestHandler.
27 }
```

- Fasthttp doesn’t provide ServeMux, but there are more powerful third-party routers and web

---

frameworks with fasthttp support:

- fasthttp-routing
- router
- lu
- atreugo
- Fiber
- Gearbox

Net/http code with simple ServeMux is trivially converted to fasthttp code:

```
1 // net/http code
2
3 m := &http.ServeMux{}
4 m.HandleFunc("/foo", fooHandlerFunc)
5 m.HandleFunc("/bar", barHandlerFunc)
6 m.Handle("/baz", bazHandler)
7
8 http.ListenAndServe(":80", m)
```

```
1 // the corresponding fasthttp code
2 m := func(ctx *fasthttp.RequestCtx) {
3     switch string(ctx.Path()) {
4     case "/foo":
5         fooHandlerFunc(ctx)
6     case "/bar":
7         barHandlerFunc(ctx)
8     case "/baz":
9         bazHandler.HandlerFunc(ctx)
10    default:
11        ctx.Error("not found", fasthttp.StatusNotFound)
12    }
13 }
14
15 fasthttp.ListenAndServe(":80", m)
```

- Because creating a new channel for every request is just too expensive, so the channel returned by RequestCtx.Done() is only closed when the server is shutting down.

```
1 func main() {
2     fasthttp.ListenAndServe(":8080", fasthttp.TimeoutHandler(func(
3         ctx *fasthttp.RequestCtx) {
4             select {
5             case <-ctx.Done():
6                 // ctx.Done() is only closed when the server is shutting
7                 // down.
8                 log.Println("context cancelled")
9                 return
10            case <-time.After(10 * time.Second):
```



---

```

9         log.Println("process finished ok")
10    }
11    }, time.Second*2, "timeout"))
12 }

```

- net/http -> fasthttp conversion table:

- All the pseudocode below assumes w, r and ctx have these types:

```

1    var (
2        w http.ResponseWriter
3        r *http.Request
4        ctx *fasthttp.RequestCtx
5    )

```

- r.Body -> ctx.PostBody()
- r.URL.Path -> ctx.Path()
- r.URL -> ctx.URI()
- r.Method -> ctx.Method()
- r.Header -> ctx.Request.Header
- r.Header.Get() -> ctx.Request.Header.Peek()
- r.Host -> ctx.Host()
- r.Form -> ctx.QueryArgs() + ctx.PostArgs()
- r.PostForm -> ctx.PostArgs()
- r.FormValue() -> ctx.FormValue()
- r.FormFile() -> ctx.FormFile()
- r.MultipartForm -> ctx.MultipartForm()
- r.RemoteAddr -> ctx.RemoteAddr()
- r.RequestURI -> ctx.RequestURI()
- r.TLS -> ctx.IsTLS()
- r.Cookie() -> ctx.Request.Header.Cookie()
- r.Referer() -> ctx.Referer()
- r.UserAgent() -> ctx.UserAgent()
- w.Header() -> ctx.Response.Header
- w.Header().Set() -> ctx.Response.Header.Set()
- w.Header().Set("Content-Type") -> ctx.SetContentType()
- w.Header().Set("Set-Cookie") -> ctx.Response.Header.SetCookie()
- w.Write() -> ctx.Write(), ctx.SetBody(), ctx.SetBodyStream(), ctx.SetBodyStreamWriter()
- w.WriteHeader() -> ctx.SetStatusCode()
- w.(http.Hijacker).Hijack() -> ctx.Hijack()
- http.Error() -> ctx.Error()

- 
- `http.FileServer()` -> `fasthttp.FSHandler()`, `fasthttp.FS`
  - `http.ServeFile()` -> `fasthttp.ServeFile()`
  - `http.Redirect()` -> `ctx.Redirect()`
  - `http.NotFound()` -> `ctx.NotFound()`
  - `http.StripPrefix()` -> `fasthttp.PathRewriteFunc`
- **VERY IMPORTANT!** Fasthttp disallows holding references to `RequestCtx` or to its' members after returning from `RequestHandler`. Otherwise data races are inevitable. Carefully inspect all the `net/http` request handlers converted to fasthttp whether they retain references to `RequestCtx` or to its' members after returning. `RequestCtx` provides the following *band aids* for this case:
    - Wrap `RequestHandler` into `TimeoutHandler`.
    - Call `TimeoutError` before returning from `RequestHandler` if there are references to `RequestCtx` or to its' members. See the example for more details.

Use this brilliant tool - race detector - for detecting and eliminating data races in your program. If you detected data race related to fasthttp in your program, then there is high probability you forgot calling `TimeoutError` before returning from `RequestHandler`.

- Blind switching from `net/http` to fasthttp won't give you performance boost. While fasthttp is optimized for speed, its' performance may be easily saturated by slow `RequestHandler`. So profile and optimize your code after switching to fasthttp. For instance, use `quicktemplate` instead of `html/template`.
- See also `fasthttputil`, `fasthttpadaptor` and `expvarhandler`.

## Performance optimization tips for multi-core systems

- Use reuseport listener.
- Run a separate server instance per CPU core with `GOMAXPROCS=1`.
- Pin each server instance to a separate CPU core using `taskset`.
- Ensure the interrupts of multiqueue network card are evenly distributed between CPU cores. See this article for details.
- Use the latest version of Go as each version contains performance improvements.

## Fasthttp best practices

- Do not allocate objects and `[]byte` buffers - just reuse them as much as possible. Fasthttp API design encourages this.
- `sync.Pool` is your best friend.

- 
- Profile your program in production. `go tool pprof --alloc_objects your-program mem.pprof` usually gives better insights for optimization opportunities than `go tool pprof your-program cpu.pprof`.
  - Write tests and benchmarks for hot paths.
  - Avoid conversion between `[]byte` and `string`, since this may result in memory allocation+copy. Fasthttp API provides functions for both `[]byte` and `string` - use these functions instead of converting manually between `[]byte` and `string`. There are some exceptions - see this wiki page for more details.
  - Verify your tests and production code under race detector on a regular basis.
  - Prefer `quicktemplate` instead of `html/template` in your webserver.

## Tricks with `[]byte` buffers

The following tricks are used by fasthttp. Use them in your code too.

- Standard Go functions accept nil buffers

```
1 var (  
2     // both buffers are uninitialized  
3     dst []byte  
4     src []byte  
5 )  
6 dst = append(dst, src...) // is legal if dst is nil and/or src is nil  
7 copy(dst, src) // is legal if dst is nil and/or src is nil  
8 (string(src) == "") // is true if src is nil  
9 (len(src) == 0) // is true if src is nil  
10 src = src[:0] // works like a charm with nil src  
11  
12 // this for loop doesn't panic if src is nil  
13 for i, ch := range src {  
14     doSomething(i, ch)  
15 }
```

So throw away nil checks for `[]byte` buffers from you code. For example,

```
1 srcLen := 0  
2 if src != nil {  
3     srcLen = len(src)  
4 }
```

becomes

```
1 srcLen := len(src)
```

- String may be appended to `[]byte` buffer with `append`

---

```
1 dst = append(dst, "foobar"...)
```

- `[]byte` buffer may be extended to its' capacity.

```
1 buf := make([]byte, 100)
2 a := buf[:10] // len(a) == 10, cap(a) == 100.
3 b := a[:100]  // is valid, since cap(a) == 100.
```

- All fasthttp functions accept nil `[]byte` buffer

```
1 statusCode, body, err := fasthttp.Get(nil, "http://google.com/")
2 uintBuf := fasthttp.AppendUint(nil, 1234)
```

- String and `[]byte` buffers may converted without memory allocations

```
1 func b2s(b []byte) string {
2     return *(*string)(unsafe.Pointer(&b))
3 }
4
5 func s2b(s string) (b []byte) {
6     bh := (*reflect.SliceHeader)(unsafe.Pointer(&b))
7     sh := (*reflect.StringHeader)(unsafe.Pointer(&s))
8     bh.Data = sh.Data
9     bh.Cap = sh.Len
10    bh.Len = sh.Len
11    return b
12 }
```

### Warning:

This is an **unsafe** way, the result string and `[]byte` buffer share the same bytes.

**Please make sure not to modify the bytes in the `[]byte` buffer if the string still survives!**

### Related projects

- fasthttp - various useful helpers for projects based on fasthttp.
- fasthttp-routing - fast and powerful routing package for fasthttp servers.
- http2 - HTTP/2 implementation for fasthttp.
- router - a high performance fasthttp request router that scales well.
- fastws - Bloatless WebSocket package made for fasthttp to handle Read/Write operations concurrently.
- framework - a web framework made by one of fasthttp maintainers

- 
- `lu` - a high performance go middleware web framework which is based on `fasthttp`.
  - `websocket` - Gorilla-based websocket implementation for `fasthttp`.
  - `websocket` - Event-based high-performance WebSocket library for zero-allocation websocket servers and clients.
  - `fasthttpsession` - a fast and powerful session package for `fasthttp` servers.
  - `atreugo` - High performance and extensible micro web framework with zero memory allocations in hot paths.
  - `kratgo` - Simple, lightweight and ultra-fast HTTP Cache to speed up your websites.
  - `kit-plugins` - go-kit transport implementation for `fasthttp`.
  - `Fiber` - An Expressjs inspired web framework running on `Fasthttp`
  - `Gearbox` - :gear: gearbox is a web framework written in Go with a focus on high performance and memory optimization
  - `http2curl` - A tool to convert `fasthttp` requests to curl command line

## FAQ

- *Why creating yet another http package instead of optimizing net/http?*

Because `net/http` API limits many optimization opportunities. For example:

- `net/http` Request object lifetime isn't limited by request handler execution time. So the server must create a new request object per each request instead of reusing existing objects like `fasthttp` does.
- `net/http` headers are stored in a `map[string][]string`. So the server must parse all the headers, convert them from `[]byte` to `string` and put them into the map before calling user-provided request handler. This all requires unnecessary memory allocations avoided by `fasthttp`.
- `net/http` client API requires creating a new response object per each request.

- *Why fasthttp API is incompatible with net/http?*

Because `net/http` API limits many optimization opportunities. See the answer above for more details. Also certain `net/http` API parts are suboptimal for use:

- Compare `net/http` connection hijacking to `fasthttp` connection hijacking.
- Compare `net/http` Request.Body reading to `fasthttp` request body reading.

- *Why fasthttp doesn't support HTTP/2.0 and WebSockets?*

HTTP/2.0 support is in progress. WebSockets has been done already. Third parties also may use `RequestCtx.Hijack` for implementing these goodies.

- 
- *Are there known net/http advantages comparing to fasthttp?*

Yes:

- net/http supports HTTP/2.0 starting from go1.6.
- net/http API is stable, while fasthttp API constantly evolves.
- net/http handles more HTTP corner cases.
- net/http can stream both request and response bodies
- net/http can handle bigger bodies as it doesn't read the whole body into memory
- net/http should contain less bugs, since it is used and tested by much wider audience.

- *Why fasthttp API prefers returning []byte instead of string?*

Because []byte to string conversion isn't free - it requires memory allocation and copy. Feel free wrapping returned []byte result into string() if you prefer working with strings instead of byte slices. But be aware that this has non-zero overhead.

- *Which GO versions are supported by fasthttp?*

Go 1.18.x. Older versions won't be supported.

- *Please provide real benchmark data and server information*

See this issue.

- *Are there plans to add request routing to fasthttp?*

There are no plans to add request routing into fasthttp. Use third-party routers and web frameworks with fasthttp support:

- fasthttp-routing
- router
- framework
- lu
- atreugo
- Fiber
- Gearbox

See also this issue for more info.

- *I detected data race in fasthttp!*

Cool! File a bug. But before doing this check the following in your code:

- Make sure there are no references to RequestCtx or to its' members after returning from RequestHandler.

- 
- Make sure you call `TimeoutError` before returning from `RequestHandler` if there are references to `RequestCtx` or to its' members, which may be accessed by other goroutines.

- *I didn't find an answer for my question here*

Try exploring these questions.