

Leapcell

Posted on Jan 27



Go's http.ServeMux Is All You Need

#go #webdev #http #performance



Go http.ServeMux



Optimization and Application Analysis of http.ServeMux in Go 1.22 Standard Library

In the field of Go web development, to achieve more efficient and flexible routing functions, many developers choose to introduce third - party libraries such as httprouter and gorilla/mux. However, in the Go 1.22 version, the official has significantly optimized the http.ServeMux in the standard library. This move is expected to reduce developers' reliance on third - party routing libraries.

I. Highlights of Go 1.22: Enhanced Pattern Matching Ability

The Go 1.22 version has implemented a much - anticipated proposal to enhance the pattern matching ability of the default HTTP service multiplexer in the net/http package of the standard library. The existing multiplexer (http.ServeMux) can only provide basic path matching functions, which are relatively limited. This has led to the emergence of a large number of third - party libraries to meet developers' needs for more powerful routing functions. The new multiplexer in Go 1.22 will significantly narrow the functional

gap with third - party libraries by introducing advanced matching capabilities. This article will briefly introduce the new multiplexer (mux), provide an example of a REST server, and compare the performance of the new standard library mux with that of gorilla/mux.

II. How to Use the New mux

For Go developers who have experience using third - party mux/router (such as gorilla/mux), using the new standard mux will be a simple and familiar task. It is recommended that developers first read its official documentation, which is concise and clear.

(I) Basic Usage Example

The following code demonstrates some new pattern matching functions of mux:

```
package main
import (
    "fmt"
    "net/http"
)
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /path/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprint(w, "got path\n")
    })
    mux.HandleFunc("/task/{id}/", func(w http.ResponseWriter, r *http.Request) {
        id := r.PathValue("id")
        fmt.Fprintf(w, "handling task with id=%v\n", id)
    })
    http.ListenAndServe("localhost:8090", mux)
}
```

Experienced Go programmers can immediately notice two new features:

1. In the first handler, the HTTP method (GET in this example) is explicitly used as part of the pattern. This means that this handler will only respond to GET requests for paths starting with /path/, and will not handle requests of other HTTP methods.
2. In the second handler, the second path component - {id} contains a wildcard, which was not supported in previous versions. This wildcard can match a single path

component, and the handler can obtain the matched value through the PathValue method of the request.

The following is an example of testing this server using the curl command:

```
$ gotip run sample.go
# Test in another terminal
$ curl localhost:8090/what/
404 page not found
$ curl localhost:8090/path/
got path
$ curl -X POST localhost:8090/path/
Method Not Allowed
$ curl localhost:8090/task/leapcell/
handling task with id=leapcell
```

From the test results, it can be seen that the server will reject POST requests to /path/ and only allow GET requests (curl uses GET requests by default). At the same time, when the request matches, the id wildcard will be assigned the corresponding value. It is recommended that developers refer to the documentation of the new ServeMux in detail to understand more functions, such as rules for trailing paths and wildcard matching with {id}, and strict matching of paths ending with {\$}.

(II) Pattern Conflict Handling


The proposal pays special attention to the possible conflict issues between different patterns. The following is an example:

```
mux := http.NewServeMux()
mux.HandleFunc("/task/{id}/status/", func(w http.ResponseWriter, r *http.Request) {
    id := r.PathValue("id")
    fmt.Fprintf(w, "handling task status with id=%v\n", id)
})
mux.HandleFunc("/task/0/{action}/", func(w http.ResponseWriter, r *http.Request) {
    action := r.PathValue("action")
    fmt.Fprintf(w, "handling task 0 with action=%v\n", action)
})
```

When the server receives a request for /task/0/status/, both handlers can match this request. The new ServeMux documentation describes in detail the pattern priority rules

and how to handle potential conflicts. If a conflict occurs, the registration process will trigger a panic. For the above example, the following error message will appear:

```
panic: pattern "/task/0/{action}/" (registered at sample - conflict.go:14) conflicts with  
/task/0/{action}/ and /task/{id}/status/ both match some paths, like "/task/0/status/".  
But neither is more specific than the other.  
/task/0/{action}/ matches "/task/0/action/", but /task/{id}/status/ doesn't.  
/task/{id}/status/ matches "/task/id/status/", but /task/0/{action}/ doesn't.
```



This error message is detailed and practical. In complex registration scenarios (especially when patterns are registered in multiple locations in the source code), these details can help developers quickly locate and solve conflict problems.

III. Implementing a Server with the New mux

The REST Servers in Go series implemented a simple server for a task/to - do list application in Go using multiple methods. The first part was implemented based on the standard library, and the second part re - implemented the same server using the gorilla/mux router. Now, it is of great significance to implement this server again with the enhanced mux of Go 1.22, and it is also interesting to compare it with the solution using gorilla/mux.

(I) Pattern Registration Example

The following is some representative pattern registration code:

```
mux := http.NewServeMux()  
server := NewTaskServer()  
mux.HandleFunc("POST /task/", server.createTaskHandler)  
mux.HandleFunc("GET /task/", server.getAllTasksHandler)  
mux.HandleFunc("DELETE /task/", server.deleteAllTasksHandler)  
mux.HandleFunc("GET /task/{id}/", server.getTaskHandler)  
mux.HandleFunc("DELETE /task/{id}/", server.deleteTaskHandler)  
mux.HandleFunc("GET /tag/{tag}/", server.tagHandler)  
mux.HandleFunc("GET /due/{year}/{month}/{day}/", server.dueHandler)
```

Similar to the gorilla/mux example, here requests with the same path are routed to different handlers using specific HTTP methods. When using the old http.ServeMux, such matchers would direct requests to the same handler, and then the handler would decide subsequent operations based on the request method.

(II) Handler Example

The following is a code example of a handler:

```
func (ts *taskServer) getTaskHandler(w http.ResponseWriter, req *http.Request) {
    log.Printf("handling get task at %s\n", req.URL.Path)
    id, err := strconv.Atoi(req.PathValue("id"))
    if err != nil {
        http.Error(w, "invalid id", http.StatusBadRequest)
        return
    }
    task, err := ts.store.GetTask(id)
    if err != nil {
        http.Error(w, err.Error(), http.StatusNotFound)
        return
    }
    renderJSON(w, task)
}
```

This handler extracts the ID value from `req.PathValue("id")`, which is similar to the Gorilla method. However, since the regular expression is not used to specify that {id} only matches integers, attention needs to be paid to the error returned by `strconv.Atoi`.

Overall, the final result is quite similar to the solution using `gorilla/mux`. Compared with the traditional standard library method, the new `mux` can perform more complex routing operations, reducing the need to leave routing decisions to the handler itself, and improving development efficiency and code maintainability.

IV. Conclusion

"Which router library should I choose?" has always been a common question faced by Go beginners. After the release of Go 1.22, the answer to this question may change. Many developers will find that the new standard library `mux` is sufficient to meet their needs, eliminating the need to rely on third - party packages.

Of course, some developers will continue to choose the familiar third - party libraries, which is also reasonable. Routers like `gorilla/mux` still have more functions than the standard library. In addition, many Go programmers will choose lightweight frameworks like `Gin` because it not only provides a router but also additional tools required to build a web backend.

In conclusion, the optimization of the standard library `http.ServeMux` in Go 1.22 is undoubtedly a positive change. Whether developers choose to use third - party packages or stick to the standard library, enhancing the functionality of the standard library is beneficial to the entire Go development community.

[Leapcell: The Best Serverless Platform for Go app Hosting, Async Tasks, and Redis](#)



Reliable Distributed Applications Platform.

Finally, recommend a platform that is most suitable for deploying Go services: [Leapcell](#)

1. Multi - Language Support

- Develop with JavaScript, Python, Go, or Rust.

2. Deploy unlimited projects for free

- pay only for usage — no requests, no charges.

3. Unbeatable Cost Efficiency

- Pay - as - you - go with no idle charges.
- Example: \$25 supports 6.94M requests at a 60ms average response time.

4. Streamlined Developer Experience

- Intuitive UI for effortless setup.
- Fully automated CI/CD pipelines and GitOps integration.
- Real - time metrics and logging for actionable insights.

5. Effortless Scalability and High Performance

- Auto - scaling to handle high concurrency with ease.
- Zero operational overhead — just focus on building.



Start deploying in Leapcell



[Explore more in the documentation!](#)

Leapcell Twitter: <https://x.com/LeapcellHQ>



Checkly

PROMOTED



[5 Playwright CLI Flags That Will Transform Your Testing Workflow](#)

- [0:56](#) --last-failed
- [2:34](#) --only-changed
- [4:27](#) --reporter

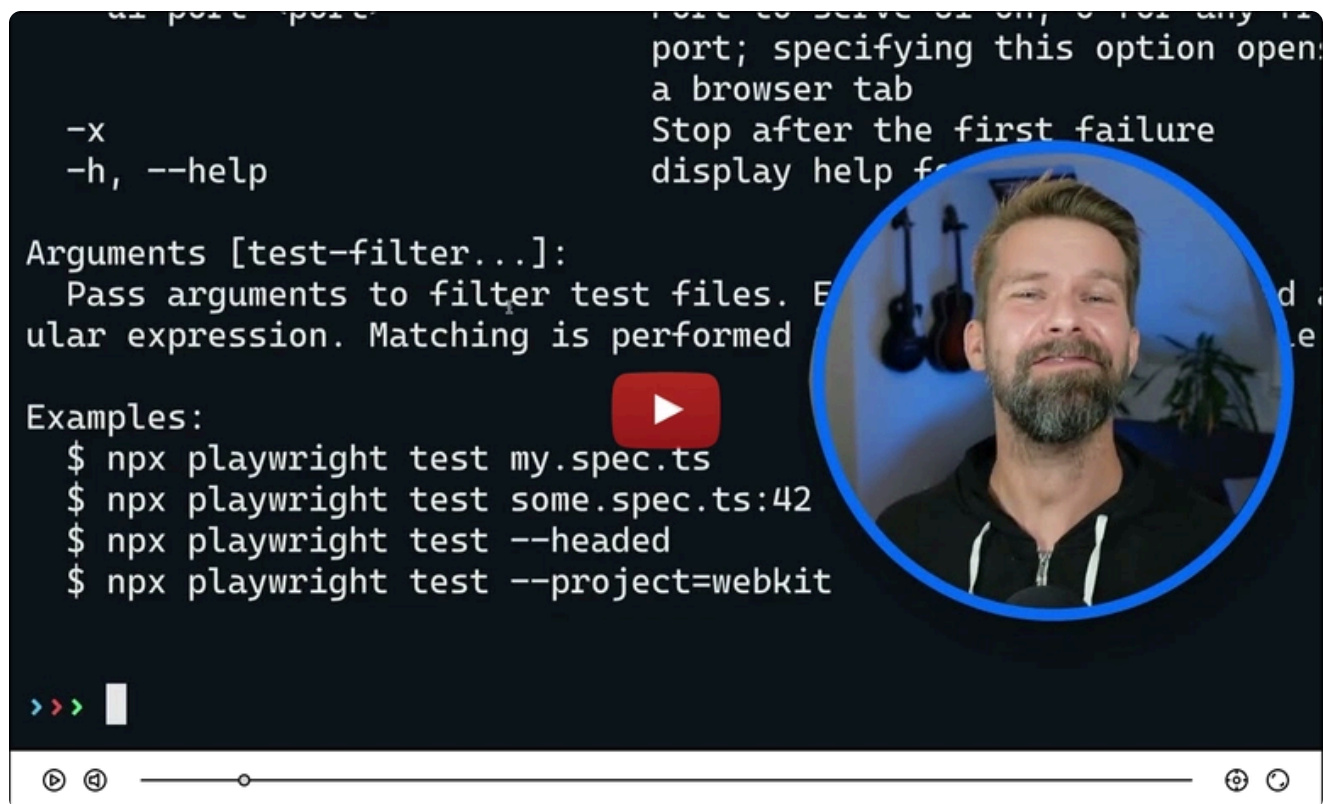
[Read More](#)

Top comments (0)

[Code of Conduct](#) • [Report abuse](#)

 Checkly PROMOTED

...



[5 Playwright CLI Flags That Will Transform Your Testing Workflow](#)

[Watch Full Video](#) 



Leapcell

leapcell.io: serverless web hosting / async task / redis

LOCATION

California

JOINED

Jul 30, 2024

More from **Leapcell**

Traits in Rust Explained: From Usage to Internal Mechanics

[#webdev](#) [#programming](#) [#backend](#) [#rust](#)

Understanding Traits and Trait Bounds in Rust

[#webdev](#) [#programming](#) [#backend](#) [#rust](#)

Reimplementing the Gin Web Framework from Scratch in Go

[#go](#) [#gin](#) [#http](#) [#webdev](#)



Sentry

PROMOTED





[The best way to debug slow web pages](#)

Tools like Page Speed Insights and Google Lighthouse are great for providing advice for front end performance issues. But what these tools can't do, is evaluate performance across your entire stack of distributed services and applications.

[Watch video](#)