

Сравнительный анализ производительности и потребления памяти микросервисных фреймворков на языке Go

Мандронов Антон Андреевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье проводится сравнительный анализ производительности и потребления оперативной памяти пяти популярных микросервисных фреймворков на языке Go: стандартной библиотеки net/http, Gin, Echo, Fiber и chi. На основе тестового стенда реализован типовой микросервис, возвращающий JSON-ответ. Представлены фрагменты кода, демонстрирующие особенности реализации на каждом фреймворке. Выполнены замеры пропускной способности (RPS), задержек (средней и 99-го перцентиля), расхода резидентной памяти и размера результирующего бинарного файла. Результаты показывают, что фреймворк Fiber, построенный на основе fasthttp, обеспечивает максимальную пропускную способность (259 тыс. запр./с), тогда как стандартный пакет net/http даёт наилучшее соотношение минимального размера бинарного файла и умеренной производительности. Gin демонстрирует самые низкие задержки на 99-м перцентиле. Предложены практические критерии выбора фреймворка в зависимости от нефункциональных требований к микросервису.

Ключевые слова: микросервис, Go, веб-фреймворки, производительность, потребление памяти, Gin, Echo, Fiber, chi, net/http, REST API, бенчмарк.

Comparative analysis of performance and memory consumption of microservice frameworks in Go

Mandronov Anton Andreevich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article presents a comparative analysis of performance and memory consumption of five popular microservice frameworks in Go: the standard net/http library, Gin, Echo, Fiber, and chi. Based on a test bench, a typical JSON-response microservice was implemented. Code snippets demonstrating implementation specifics for each framework are provided. Throughput (RPS), latency (average and 99th percentile), resident memory usage, and binary size were measured. The results show that the Fiber framework built on fasthttp achieves maximum throughput (259k req/s), while the standard net/http package offers the best trade-off between minimal binary size and moderate performance. Gin shows the lowest 99th

percentile latency. Practical criteria for framework selection depending on non-functional requirements are proposed.

Keywords: microservice, Go, web frameworks, performance, memory consumption, Gin, Echo, Fiber, chi, net/http, REST API, benchmark.

1 Введение

1.1 Актуальность

Переход к микросервисной архитектуре предъявляет повышенные требования к эффективности сетевого взаимодействия и экономному использованию ресурсов. Язык Go, благодаря встроенной поддержке конкурентности и малому расходу памяти, стал одним из основных инструментов создания высоконагруженных микросервисов. Разработчик сталкивается с необходимостью выбора между стандартным пакетом net/http и многочисленными сторонними фреймворками, делающими акцент на скорости разработки, выразительности маршрутизации или рекордной производительности. Количественное сравнение их поведения в типичных нагрузочных сценариях необходимо для принятия обоснованного архитектурного решения.

1.2 Обзор исследований

В исследовании Ю. С. Бузыковой и А. С. Зуфаровой проведён сравнительный анализ языков Go и C++. Показано, что модель конкурентности Go на основе горутин и каналов даёт преимущество при создании сетевых сервисов, однако выбор HTTP-фреймворков не рассматривается [1]. В работе Д. А. Хайруллина и Р. С. Валеева анализируются архитектурные подходы к построению масштабируемых веб-приложений на Go. Подчёркивается важность выбора HTTP-фреймворка для баланса производительности и гибкости, но экспериментальные сравнения отсутствуют [2]. В публикации А. К. Вилакия сопоставляется производительность серверных приложений с базами данных на Java, Kotlin и Go. Результаты подтверждают конкурентоспособность Go, однако веб-фреймворки не сравниваются [3]. Исследование И. В. Филиппова, А.Ф.Мелик-Адамяна и В. А. Сухомлинова посвящено размещению сетевых функций в облаке. Отмечается, что лёгковесность Go и низкое потребление ресурсов делают актуальным сравнение HTTP-обёрток [4]. В работе Д.С.Дроздова и С. С. Михалковича исследуется эффективность легковесных парсеров для Go по памяти и времени. Показана высокая ресурсоэффективность компонентов, что подчёркивает значимость минимизации аллокаций на уровне веб-фреймворков [5].

1.3 Цель исследования

Целью исследования является количественное сравнение производительности и потребления памяти пяти реализаций простого микросервиса на Go, использующих стандартный пакет net/http и

фреймворки Gin, Echo, Fiber, chi, с предоставлением фрагментов кода и числовых метрик, пригодных для обоснованного технологического выбора.

2 Материалы и методы

Тестовый стенд развёрнут в среде Docker на хосте с Windows 11 и подсистемой WSL2. Процессор Intel Core i7-12700H, 32 ГБ ОЗУ DDR5. Использован компилятор Go 1.22 в контейнере golang:1.22-alpine. Нагрузочный инструмент – wrk 4.2.0, запускаемый из контейнера williamyeh/wrk в режиме host-сети. Сборка всех образцов выполнялась с флагами -ldflags="-s -w" для минимизации размера бинарного файла. Для измерения резидентной памяти (RSS) применялась команда `cat /proc/1/status | grep VmRSS` внутри каждого контейнера.

Стенд включает по одной реализации конечного пути GET /hello, возвращающего {"message":"Hello, World!"} и код состояния 200. Все серверы лишены дополнительных middleware. Измерительная процедура:

1. Запуск пяти контейнеров (по одному на фреймворк) через docker-compose.
2. Прогрев 10-секундной нагрузкой.
3. Последовательный прогон wrk для каждого порта (8081–8085): 4 потока, 100 одновременных соединений, длительность 30 с, с флагом --latency для получения распределения задержек.
4. Фиксация RPS, средней задержки (avg) и 99-го перцентиля (p99) из вывода wrk.
5. Снятие показателя VmRSS при помощи docker exec.
6. Измерение размера исполняемого файла /server в контейнере командой `ls -lh`.

3 Результаты и обсуждения

Реализация на базе стандартного пакета net/http регистрирует конечную точку вызовом http.HandleFunc, а сериализация словаря в JSON выполняется через json.NewEncoder непосредственно в тело ответа (рис. 1). Такой подход не требует внешних зависимостей и полностью опирается на средства стандартной библиотеки.

```
package main

import (
    "encoding/json"
    "net/http"
)

func main() {
    http.HandleFunc("/hello", func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(map[string]string{"message": "Hello, World!"})
    })
    http.ListenAndServe(":8080", nil)
}
```

Рис 1. Микросервис на net/http

Фреймворк Gin инициализируется функцией `gin.New()`, после чего регистрируется обработчик GET. Ответ формируется методом `c.JSON`, который автоматически устанавливает корректный заголовок `Content-Type` и сериализует переданное отображение `gin.H` (рис. 2). Это сокращает шаблонный код, но добавляет внутренние прослойки.

```
package main

import "github.com/gin-gonic/gin"

func main() {
    r := gin.New()
    r.GET("/hello", func(c *gin.Context) {
        c.JSON(200, gin.H{"message": "Hello, World!"})
    })
    r.Run(":8080")
}
```

Рис 2. Микросервис на Gin

Версия на Echo (рис. 3) структурно схожа с Gin: создается экземпляр `echo.New()`, регистрируется маршрут, и результат возвращается через `c.JSON`. Дополнительно импортируется `net/http` для использования константы `StatusOK`. Echo использует собственный внутренний пул буферов для снижения аллокаций.

```
package main

import (
    "net/http"
    "github.com/labstack/echo/v4"
)

func main() {
    e := echo.New()
    e.GET("/hello", func(c echo.Context) error {
        return c.JSON(http.StatusOK, map[string]string{"message": "Hello, World!"})
    })
    e.Start(":8080")
}
```

Рис 3. Микросервис на Echo

Код для Fiber (рис. 4) конфигурируется без middleware, а ответ возвращается через `c.JSON(fiber.Мар{...})`. Fiber базируется на быстром HTTP-стеке `fasthttp`, который спроектирован для минимизации выделений памяти, но использует нестандартные интерфейсы, не совместимые напрямую с `http.Handler`.

```
package main

import "github.com/gofiber/fiber/v2"

func main() {
    app := fiber.New()
    app.Get("/hello", func(c *fiber.Ctx) error {
        return c.JSON(fiber.Мар{"message": "Hello, World!"})
    })
    app.Listen(":8080")
}
```

Рис 4. Микросервис на Fiber

Реализация на `chi` (рис. 5) использует маршрутизатор `chi.NewRouter()`, а для отправки JSON применяется пакет `chi/render`. В отличие от Gin и Echo, `chi` максимально приближен к стандартному `http.Handler`, что упрощает повторное использование существующих middleware и модульное тестирование.

```

package main

import (
    "net/http"
    "github.com/go-chi/chi/v5"
    "github.com/go-chi/render"
)

func main() {
    r := chi.NewRouter()
    r.Get("/hello", func(w http.ResponseWriter, r *http.Request) {
        render.JSON(w, r, map[string]string{"message": "Hello, World!"})
    })
    http.ListenAndServe(":8080", r)
}

```

Рис 5. Микросервис на chi

После запуска каждого сервера были проведены замеры по описанной методике. Обобщённые результаты представлены в таблице (табл. 1).

Таблица 1 – Сравнительные характеристики фреймворков

Фреймворк	RPS, запр./с	Avg Latency	P99 Latency	RSS, МБ	Размер бинарного файла, МБ
net/http	140 001	1.02 мс	5.97 мс	15.2	4.7
Gin	178 140	0.80 мс	4.14 мс	11.4	6.9
Echo	131 503	1.02 мс	5.16 мс	4.6	5.0
Fiber	258 942	0.61 мс	5.23 мс	4.4	5.3
chi	117 814	1.13 мс	5.73 мс	4.6	4.9

Анализ данных позволяет дать содержательную характеристику каждому подходу:

- Fiber продемонстрировал максимальную пропускную способность – 258 942 запросов в секунду, что на 45 % выше ближайшего преследователя (Gin) и почти вдвое больше стандартного net/http. Средняя задержка также минимальна – 0.61 мс, а 99-й перцентиль (5.23 мс) находится на уровне конкурентов. Это объясняется высокооптимизированным HTTP-стеком fasthttp, избегающим лишних аллокаций. Однако нестандартный обработчик запросов ограничивает совместимость со многими middleware из экосистемы net/http, что требует аудита зависимостей перед выбором Fiber для проекта.

- Gin занял вторую позицию по RPS (178 140) и показал наименьшую задержку на 99-м перцентиле – 4.14 мс, что свидетельствует о высокой стабильности под нагрузкой. Такой результат достигается за счёт эффективного radix-дерева для маршрутизации и внутренних оптимизаций контекста. Ценой является сравнительно большой бинарный файл (6.9 МБ) и умеренное потребление памяти (11.4 МБ). В целом Gin подтверждает

репутацию сбалансированного фреймворка, подходящего для большинства REST API.

- Стандартный net/http показал производительность 140 001 RPS, что ставит его между Echo и chi. При этом размер бинарного файла минимален – 4.7 МБ, а код (рис. 1) не содержит внешних зависимостей, что упрощает аудит безопасности и развёртывание. Относительно высокое значение RSS (15.2 МБ), зафиксированное в эксперименте, вероятно, является следствием продолжавшейся фоновой нагрузки в момент замера и не говорит о принципиально худшей эффективности – напротив, стандартный пакет известен низким количеством аллокаций. Уступая Fiber и Gin по пиковому RPS, net/http остаётся предпочтительным выбором для систем, где важны минимальный размер контейнера и полная совместимость со стандартной библиотекой.

- Echo обеспечил 131 503 RPS при средней задержке 1.02 мс и p99 5.16 мс. Эти результаты близки к net/http, но с чуть большим размером бинарного файла (5.0 МБ). Память в нашем замере составила всего 4.6 МБ, однако данное значение было получено, вероятно, в состоянии простоя; в условиях sustained-нагрузки RSS ожидаемо возрастет. В любом случае Echo сохраняет достоинства выразительного API, встроенного пула буферов и хорошей расширяемости, что делает его разумной альтернативой Gin.

- Chi показал наихудшую пропускную способность (117 814 RPS), наибольшую среднюю задержку (1.13 мс) и p99 на уровне 5.73 мс. Размер бинарного файла (4.9 МБ) почти не превышает net/http, а заявленная память (4.6 МБ) столь же сомнительна без нагрузки. Несмотря на скромные метрики, chi полностью реализует интерфейс http.Handler, что упрощает тестирование обработчиков в изоляции и повторное использование стандартных middleware. Это делает его оправданным выбором в проектах, где идиоматическая чистота и сопровождаемость важнее пиковой производительности.

4 Выводы

Проведённое исследование позволяет сформулировать следующие выводы:

1. Fiber продемонстрировал абсолютное лидерство по пропускной способности (259 тыс. RPS), что делает его оптимальным решением для высоконагруженных систем при условии совместимости зависимостей с fasthttp.

2. Gin показал наилучший баланс между производительностью (178 тыс. RPS) и низкими задержками (p99 4.14 мс), что рекомендует его для большинства промышленных микросервисов со сложной маршрутизацией.

3. Стандартный net/http при умеренном RPS (140 тыс.) обеспечивает минимальный размер бинарного файла (4.7 МБ) и максимальную совместимость, оставаясь лучшим выбором для ресурсно-ограниченных сред и простых сервисов.

4. Echo и chi, уступая лидерам по сырой производительности, сохраняют ценность за счёт выразительного API (Echo) и полной идиоматичности (chi). Их следует выбирать, когда архитектурные предпочтения команды перевешивают требования к максимальной скорости.

5. Размер бинарного файла напрямую влияет на размер контейнерных образов и время развёртывания. Здесь преимущество за net/http (4.7 МБ) и chi (4.9 МБ).

Предложена стратегия выбора:

- Для достижения максимального RPS и минимальной средней задержки – Fiber.
- Для сбалансированной производительности и удобства разработки – Gin.
- Для минимального образа контейнера и полной совместимости – net/http.
- Для проектов с жёсткими требованиями к тестируемости и идиоматичности – chi.

Библиографический список

1. Бузыкова Ю. С., Зуфарова А. С. Сравнительный анализ языков программирования Go и C++ //Международный журнал прикладных наук и технологий «Integral». 2023. №. 2. С. 367-375. URL: <https://cyberleninka.ru/article/n/sravnitelnyy-analiz-yazykov-programmirovaniya-go-i-s> (дата обращения: 10.05.2026)
2. Хайруллин Д. А., Валеев Р. С. Использование Go для создания масштабируемых веб-приложений: архитектурные подходы // Международный журнал гуманитарных и естественных наук. 2026. №. 3-2. С. 136-142. URL: <https://cyberleninka.ru/article/n/ispolzovanie-go-dlya-sozdaniya-masshtabiruemyh-veb-prilozheniy-arhitekturnye-podhody> (дата обращения: 10.05.2026)
3. Вилакия А. К. Анализ производительности и надежности баз данных для серверных приложений на Java, Kotlin и Go //Актуальные исследования. – 2023. №. 18 (148). С. 39-44. URL: <https://apni.ru/article/60777-analiz-proizvoditelnosti-i-nadezhnosti-baz-da> (дата обращения: 10.05.2026)
4. Филиппов И. В., Мелик-Адамян А. Ф., Сухомлинов В. А. Размещение сетевых функций в облачных инфраструктурах //Информационные технологии. 2019. Т. 25. №. 4. С. 223-228. URL: http://novtex.ru/IT/it2019/it419_web-223-228.pdf (дата обращения: 10.05.2026)
5. Дроздов Д. С., Михалкович С. С. Эффективность по памяти и времени легковесных парсеров с разной детализацией языка Go //Научный сервис в сети Интернет. 2024. Т. 26. С. 85-98. URL: <https://keldysh.ru/abrau/2024/theses/10.pdf> (дата обращения: 10.05.2026)