API 性能测试和调优

本节核心内容

- 简单介绍 API 性能测试知识
- 介绍如何进行 API 性能测试
- 简单介绍如何进行 API 性能分析
- 给出 apiserver 的性能数据

> 本节最后会给出性能测试脚本：`wrktest.sh`，脚本见 `demo17/wrktest.sh`

性能测试

在 API 上线之前，我们需要知道 API 的性能，以便知道 API 服务器所能承载的最大请求量、性能瓶颈，再根据业务的需求量来对 API 进行性能调优或者扩缩容。通过这些可以使 API 稳定地对外提供服务，并且请求在合理的时间内返回。

API 性能测试指标

API 性能测试，大的方面包括 API 框架的性能和指定 API 的性能，因为指定 API 的性能跟该 API 具体的实现有关，比如有无数据库连接，有无复杂的逻辑处理等，脱离了具体实现来探讨单个 API 的性能是毫无意义的，所以本小节只探讨 API 框架的性能。

衡量 API 性能的指标主要有 3 个：

1. 并发数（Concurrent）

并发数是指某个时间范围内，同时正在使用系统的用户个数。

广义上的并发数是指同时使用系统的用户个数，这些用户可能调用不同的 API。严格意义上的并发数是指同时请求同一个 API 的用户个数。本小节所讨论的并发数是严格意义上的并发数。

2. 每秒查询数（QPS）

    每秒查询数 QPS 是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。

    QPS = 并发数 / 平均请求响应时间。

3. 请求响应时间（TTLB）

    请求响应时间指的是从客户端发出请求到得到响应的整个时间。这个过程从客户端发起的一个请求开始，到客户端收到服务器端的响应结束。在一些工具中，请求响应时间通常会被称为 TTLB（Time to last byte，意思是从发送一个请求开始，到客户端收到最后一个字节的响应为止所消费的时间）。请求响应时间的单位一般为"秒”或“毫秒”。

衡量 API 性能的最主要指标是 QPS，但是在说明 QPS 时，需要指明是多少并发数下的 QPS，否则毫无意义，因为不同并发数下的 QPS 是不同的。比如单用户 100 QPS 和 100 用户 100 QPS 是两个不同的概念，前者说明 API 可以在一秒内串行执行 100 个请求，而后者说明在并发数为 100 的情况下，API 可以在一秒内处理 100 个请求。当 QPS 相同时，并发数越大，说明 API 性能越好，并发处理能力越强。

在并发数设置过大时，API 同时要处理很多请求，会频繁切换进程，而真正用于处理请求的时间变少，使得 QPS 反而会降低。并发数设置过大时，请求响应时间也会变大。API 会有一个合适的并发数，在

该并发数下，API 的 QPS 可以达到最大，但该并发数不一定是最佳并发数，还要参考该并发数下的平均请求响应时间。

API 性能测试方法

Linux 下有很多 Web 性能测试工具，常用的有 Jmeter、AB、Webbench 和 Wrk。每个工具都有自己的特点，本小节用 Wrk 来对 API 进行性能测试。Wrk 非常简单，安装方便，测试结果也相对专业些，并且可以支持 Lua 脚本来创建更复杂的测试场景。

Wrk 安装

安装步骤如下（需要切换到 root 用户）：

1. Clone wrk repo

```
git clone https://github.com/wg/wrk
```

2. 执行 make和 `make install` 安装

```
make
cp ./wrk /usr/bin
```

Wrk 使用简介

Wrk 使用方法

Wrk 使用起来不复杂，执行 `wrk --help` 可以看到 wrk 的所有运行参数:

```
$ wrk --help
Usage: wrk <options> <url>
  Options:
    -c, --connections <N>  Connections to keep
open
    -d, --duration    <T>  Duration of test
    -t, --threads     <N>  Number of threads to
use

    -s, --script      <S>  Load Lua script file
    -H, --header      <H>  Add header to request
        --latency          Print latency
statistics
        --timeout     <T>  Socket/request timeout
    -v, --version          Print version details

  Numeric arguments may include a SI unit (1k,
1M, 1G)
  Time arguments may include a time unit (2s, 2m,
2h)
```

常用的参数为：

- -t: 线程数（线程数不要太多，是核数的 2 到 4 倍即可，多了反而会因为线程切换过多造成效率降低）
- -c: 并发数
- -d: 测试的持续时间，默认为 10s
- -T: 请求超时时间
- -H: 指定请求的 HTTP Header，有些 API 需要传入一些 Header，可通过 Wrk 的 -H 参数来传入
- --latency: 打印响应时间分布
- -s: 指定 Lua 脚本，Lua 脚本可以实现更复杂的请求

Wrk 结果解析

一个简单的测试如下：

```
$ wrk -t144 -c3000 -d30s -T30s --latency
http://127.0.0.1:8080/sd/health
Running 30s test @
http://127.0.0.1:8088/sd/health
  144 threads and 3000 connections
  Thread Stats   Avg      Stdev     Max   +/-
Stdev
    Latency    32.01ms   39.32ms 488.62ms
87.93%
    Req/Sec     1.00k   251.79     3.35k
69.00%
  Latency Distribution
     50%   25.05ms
     75%   55.36ms
     90%   78.45ms
     99%  166.76ms
  4329733 requests in 30.10s, 1.81GB read
  Socket errors: connect 0, read 5, write 0,
timeout 64
Requests/sec: 143850.26
Transfer/sec:     61.46MB
```

- 144 threads and 3000 connections: 用 144 个线程模拟 3000 个连接，分别对应 -t 和 -c 参数
- Thread Stats： 线程统计
  - Latency: 响应时间，有平均值、标准偏差、最大值、正负一个标准差占比
  - Req/Sec: 每个线程每秒完成的请求数，同样有平均值、标准偏差、最大值、正负一个标准差占比

- Latency Distribution: 响应时间分布
    - 50%: 50% 的响应时间为：4.74ms
    - 75%: 75% 的响应时间为：23.42ms
    - 90%: 90% 的响应时间为：82.88ms
    - 99%: 99% 的响应时间为：236.39ms

- 19373531 requests in 30.10s, 1.35GB read: 30s 完成的总请求数（19373531）和数据读取量（1.35GB）
- Socket errors: connect 0, read 5, write 0, timeout 64: 错误统计
- Requests/sec: QPS
- Transfer/sec: TPS

apiserver 第一次性能测试

测试服务器配置：6 核 12G

在 apiserver 中，Gin middleware： Logging 会记录请求参数和返回参数，该 middleare 很消耗性能，为了测试框架的性能，这里暂时将该 middleware 禁掉，在 main.go 函数中将 middleware.Logging() 一行注释掉，如图：

```
    // Set gin mode.
    gin.SetMode(viper.GetString("runmode"))

    // Create the Gin engine.
    g := gin.New()

    // Routes.
    router.Load(
        // Cores.
        g,

        // Middlwares.
        //middleware.Logging(),
        middleware.RequestId(),
    )
```

编译并运行 apiserver

```
$ make
$ ./apiserver
```

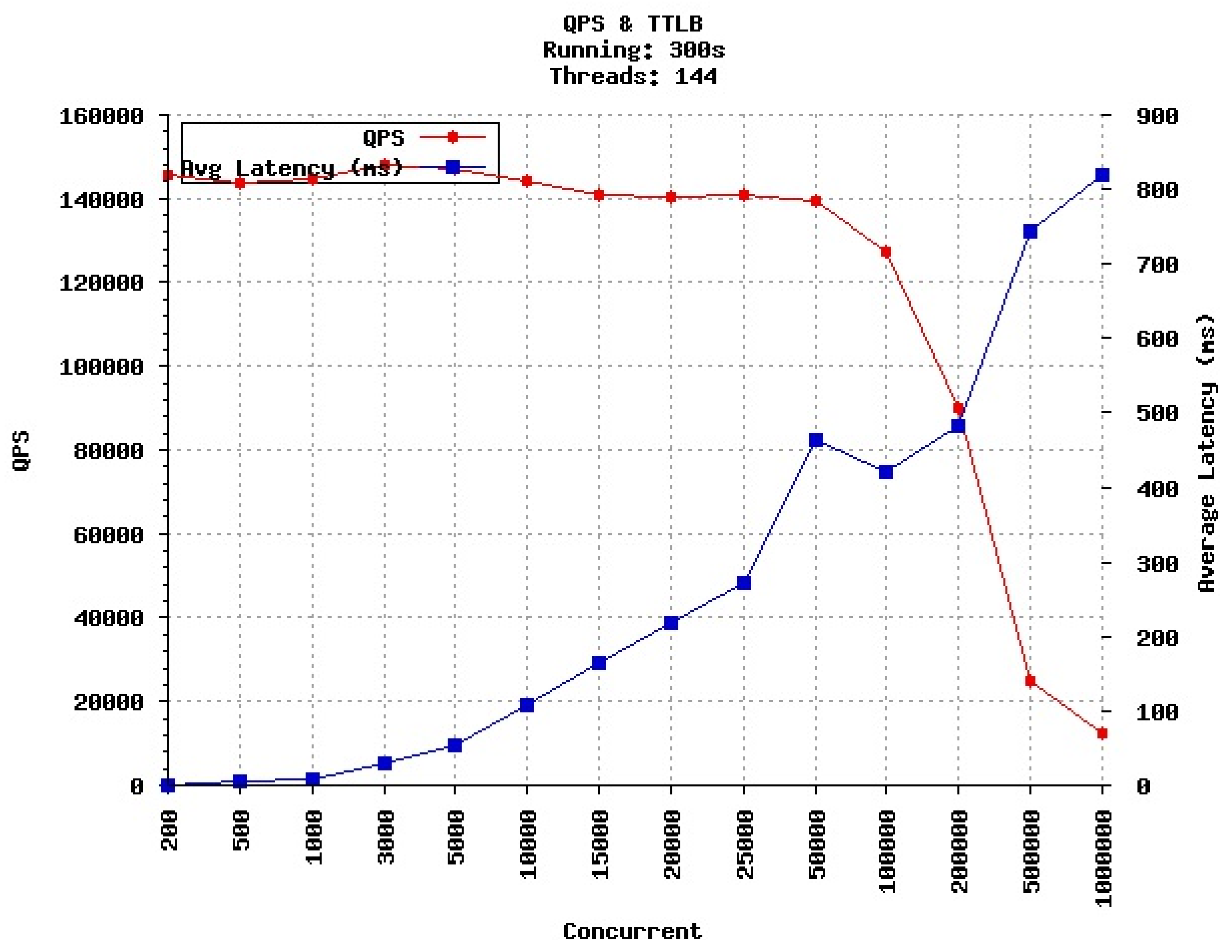
执行 wrk 命令测试 API 性能（分别测试多个并发数：200 500 1000 3000 5000 10000 15000 20000 25000 50000 100000 200000 500000 1000000）

```
$ wrk --latency -t144 -d60s -T300s
http://127.0.0.1:8080/sd/health -c 200
```

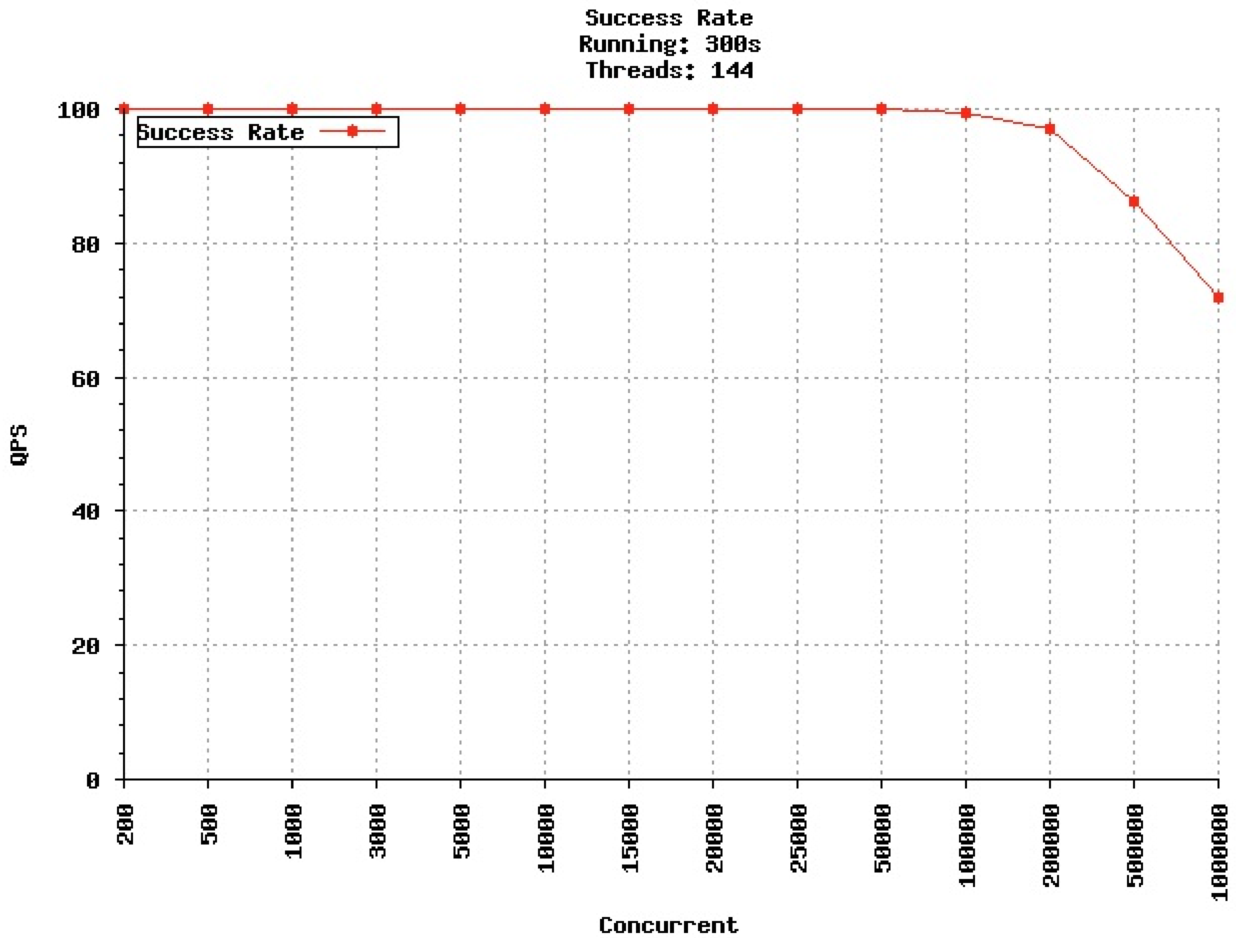
> 调用 apiserver 的健康检查接口：/sd/health

根据测试数据绘制出 QPS & TTLB 图和成功率图：

QPS & 平均响应时间:



成功率



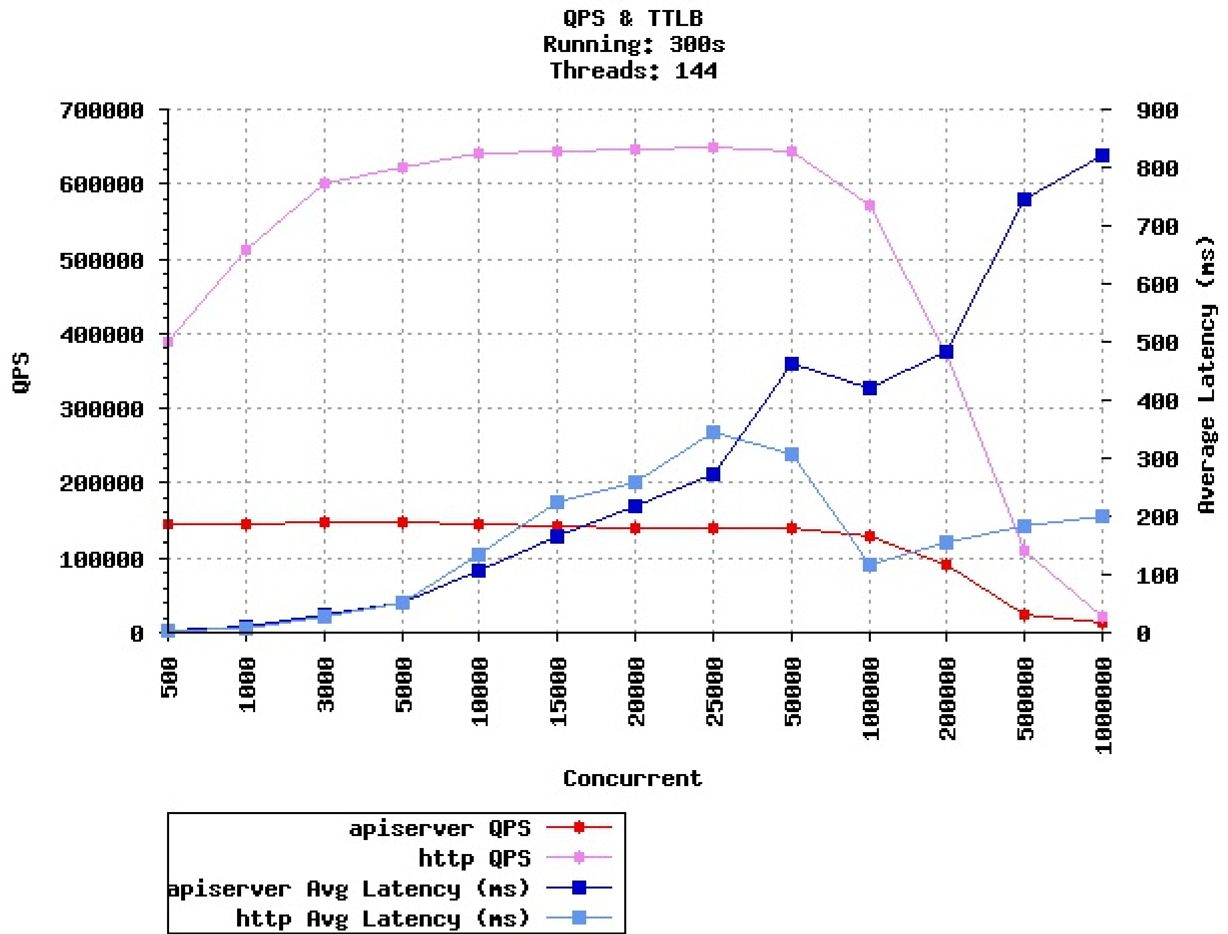
通过上面二图可以看到，apiserver 在并发数为 5000 时，QPS 最大，为 146953，平均响应时间为 52.75ms，在并发数达到 50000 时，成功率开始下降。

那么该 apiserver 的 QPS 处于什么水平呢？一方面可以根据自己的业务需要来对比，另一方面可以对比性能更好的 Web 框架。这里用 net/http 构建最简单的 HTTP 服务器，测试性能并作对比（相同的测试工具和测试服务器），HTTP 服务源码为：

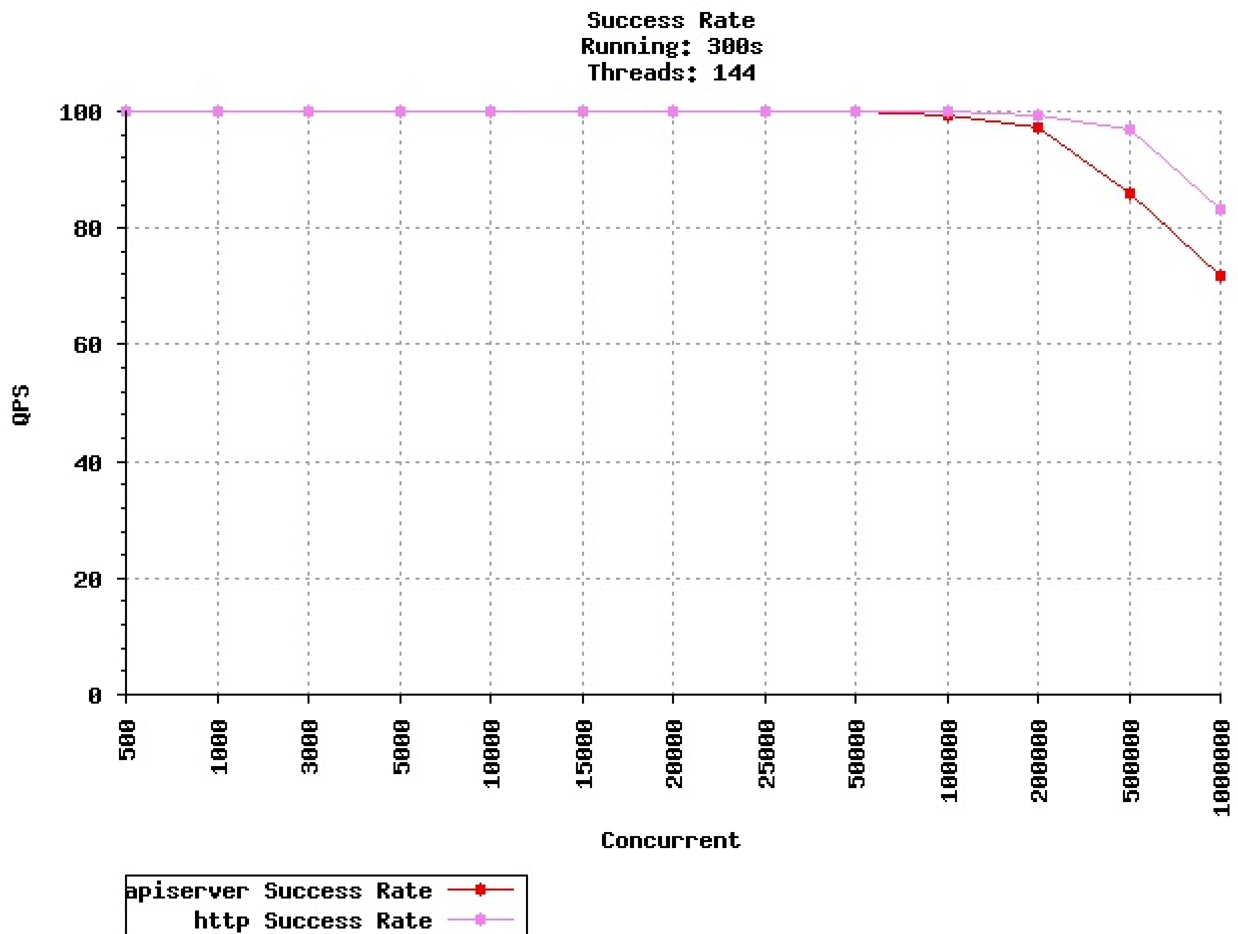
```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w
http.ResponseWriter, r *http.Request) {
        message := "OK"
        fmt.Fprintln(w, "\n"+message)
    })

    log.Fatal(http.ListenAndServe(":6667", nil))
}
```

可以看到该 HTTP 服务器很简单，只是利用 net/http 包最原生的功能，在 Go 中几乎所有的 Web 框架都是基于 net/http 包封装的，既然是封装，相对于原生的性能肯定有所不及，所以这里拿 net/http 直接启动的 HTTP 服务器来做对比，对比结果如下：

QPS & 平均响应时间对比



成功率对比



通过上面两个对比图可以看出，apiserver 在 QPS、响应时间和成功率上都不如原生的 HTTP Server，特别是 QPS，最大 QPS 只有原生 HTTP Server 最大 QPS 的 22%，性能需要调优。

> 看到需要绘图，是不是觉得有点麻烦，不慌，笔者最后会奉上自动化测试脚本，该脚本会自动解析 wrk 结果并生成需要的图表。

apiserver 性能分析

API 性能分析涉及的范围比较广，本小节不是专门教读者如何进行详细的性能分析的教程，这里仅仅展示性能分析的步骤和基本思路。

在执行性能测试的过程中，运行 go tool pprof http://127.0.0.1:8080/debug/pprof/profile，采集 30s 的性能数据并查看耗时比较久的 20 个函数：

```
(pprof) top20
Showing nodes accounting for 1.57mins, 52.96% of 2.97mins total
Dropped 510 nodes (cum <= 0.01mins)
Showing top 20 nodes out of 199
      flat  flat%   sum%        cum   cum%
  0.37mins 12.35% 12.35%   0.37mins 12.35%  runtime.futex
  0.28mins  9.49% 21.84%   0.30mins 10.02%  syscall.Syscall
  0.12mins  4.10% 25.95%   0.30mins 10.27%  runtime.lock
  0.12mins  4.09% 30.04%   0.30mins 10.05%  runtime.mallocgc
  0.12mins  3.98% 34.02%   0.12mins  3.98%  runtime.epollwait
  0.09mins  2.94% 36.96%   0.09mins  2.94%  runtime.usleep
  0.08mins  2.74% 39.70%   0.96mins 32.43%  runtime.findrunnable
  0.06mins  2.01% 41.71%   0.15mins  4.94%  runtime.runqgrab
  0.04mins  1.31% 43.03%   0.04mins  1.31%  runtime.memmove
```

```
  0.03mins   1.14% 44.17%   0.03mins   1.14%
runtime.heapBitsSetType
  0.03mins   1.12% 45.29%   0.07mins   2.20%
runtime.scanobject
  0.03mins   1.07% 46.36%   0.03mins   1.07%
runtime.procyield
  0.03mins   1.03% 47.39%   0.03mins   1.03%
crypto/sha256.block
  0.03mins   0.98% 48.38%   0.26mins   8.66%
runtime.unlock
  0.03mins   0.88% 49.26%   0.03mins   0.88%
runtime.greyobject
  0.02mins   0.82% 50.08%   0.02mins   0.82%
runtime.osyield
  0.02mins   0.76% 50.84%   0.02mins   0.76%
runtime.runqempty
  0.02mins   0.72% 51.56%   0.03mins   0.92%
runtime.step
  0.02mins   0.71% 52.27%   0.05mins   1.54%
runtime.mapassign_faststr
  0.02mins   0.69% 52.96%   0.14mins   4.73%
runtime.netpoll
```

在 go tool pprof (https://github.com/hyper0x/go_command_tutorial/blob/ma 文章中，我们知道，在默认情况下，top 命令输出的列表中只包含本地取样计数最大的前十个函数，统计的是这些函数本身运行的执行时间，实际上我们还需要知道，函数中有没有调用耗时的函数以及执行其它函数所耗费的时间，这种情况下我们需要按累积取样计数来排序，这在 pprof 中需要加上 -cum 参数：

```
(pprof) top -cum
Showing nodes accounting for 6.38s, 3.58% of
178.08s total
Dropped 510 nodes (cum <= 0.89s)
Showing top 10 nodes out of 199
      flat  flat%   sum%          cum   cum%
     0.31s  0.17%  0.17%    104.25s 58.54%
net/http.(*conn).serve
     0.08s 0.045%  0.22%     63.81s 35.83%
net/http.serverHandler.ServeHTTP
     0.09s 0.051%  0.27%     63.73s 35.79%
vendor/github.com/gin-gonic/gin.
(*Engine).ServeHTTP
     0.15s 0.084%  0.35%     63.08s 35.42%
vendor/github.com/gin-gonic/gin.
(*Engine).handleHTTPRequest
     0.09s 0.051%   0.4%     60.25s 33.83%
runtime.mcall
     0.22s  0.12%  0.53%     59.60s 33.47%
vendor/github.com/gin-gonic/gin.(*Context).Next
     0.04s 0.022%  0.55%     59.58s 33.46%
vendor/github.com/gin-
gonic/gin.RecoveryWithWriter.func1
     0.50s  0.28%  0.83%     59.20s 33.24%
runtime.schedule
     0.02s 0.011%  0.84%     59.16s 33.22%
apiserver/router/middleware.NoCache
     4.88s  2.74%  3.58%     57.76s 32.43%
runtime.findrunnable
    ....
```

> 为了能够查看到所有函数的耗时排名，你需要列出更多的函数（本小节列出了 top100）。

如果你对代码很熟悉，通过最后一列的函数名，你应该可以定位到程序中所调用函数的位置，并进行优化。因为 top100 内容过多，这里筛选程序中所调用的函数（顺序不变）。

```
Showing nodes accounting for 67.31s, 37.80% of
178.08s total
Dropped 510 nodes (cum <= 0.89s)
Showing top 50 nodes out of 199
      flat  flat%   sum%        cum   cum%
     0.31s  0.17%  0.17%    104.25s 58.54%
net/http.(*conn).serve
     0.08s 0.045%  0.22%     63.81s 35.83%
net/http.serverHandler.ServeHTTP
     0.09s 0.051%  0.27%     63.73s 35.79%
vendor/github.com/gin-gonic/gin.
(*Engine).ServeHTTP
     0.15s 0.084%  0.35%     63.08s 35.42%
vendor/github.com/gin-gonic/gin.
(*Engine).handleHTTPRequest
     0.09s 0.051%   0.4%     60.25s 33.83%
runtime.mcall
     0.22s  0.12%  0.53%     59.60s 33.47%
vendor/github.com/gin-gonic/gin.(*Context).Next
     0.04s 0.022%  0.55%     59.58s 33.46%
vendor/github.com/gin-
gonic/gin.RecoveryWithWriter.func1
     0.50s  0.28%  0.83%     59.20s 33.24%
runtime.schedule
     0.02s 0.011%  0.84%     59.16s 33.22%
```

```
apiserver/router/middleware.NoCache
     4.88s  2.74%  3.58%     57.76s 32.43%
runtime.findrunnable
     0.05s 0.028%  3.61%     56.76s 31.87%
runtime.park_m
     0.05s 0.028%  3.64%     56.72s 31.85%
apiserver/router/middleware.Options
     0.06s 0.034%  3.67%     55.32s 31.06%
apiserver/router/middleware.RequestId.func1
     0.07s 0.039%  3.71%     51.34s 28.83%
apiserver/router/middleware.AuthMiddleware.func1
     0.11s 0.062%  3.77%     51.17s 28.73%
apiserver/pkg/token.ParseRequest
     0.05s 0.028%  3.80%     26.59s 14.93%
apiserver/pkg/token.Parse
     0.07s 0.039%  3.84%     26.12s 14.67%
vendor/github.com/dgrijalva/jwt-go.Parse
     0.10s 0.056%  3.90%     26.05s 14.63%
vendor/github.com/dgrijalva/jwt-go.
(*Parser).Parse
     0.29s  0.16%  4.06%     25.84s 14.51%
vendor/github.com/dgrijalva/jwt-go.
(*Parser).ParseWithClaims
     ...
     0.05s 0.028% 36.28%     11.08s  6.22%
vendor/github.com/spf13/viper.GetString
     0.02s 0.011% 36.29%     11.03s  6.19%
vendor/github.com/spf13/viper.(*Viper).GetString
     0.12s 0.067% 36.36%     10.92s  6.13%
vendor/github.com/spf13/viper.(*Viper).Get
```

从上面，我们可以知道 apiserver 中函数耗时排名如下：

1. apiserver/router/middleware.NoCache: Gin middleware，强制浏览器不使用缓存
2. apiserver/router/middleware.Options: Gin middleware，跨域设置
3. apiserver/router/middleware.RequestId.func1: Gin middleware，记录 RequestId
4. apiserver/router/middleware.AuthMiddleware.fun Gin middleware，JWT 认证
5. apiserver/pkg/token.ParseRequest: Token 功能，JWT 认证相关
6. apiserver/pkg/token.Parse: Token 功能，JWT 认证相关
7. vendor/github.com/dgrijalva/jwt-go.Parse: Token 功能，JW 认证相关
8. vendor/github.com/dgrijalva/jwt-go.(*Parser).Parse: Token 功能，JWT 认证相关
9. vendor/github.com/dgrijalva/jwt-go.(*Parser).ParseWithClaims: Token 功能，JWT 认证相关
10. vendor/github.com/spf13/viper.GetString: pkg/token/token.go 中获取 jwt_secret 的值
11. vendor/github.com/spf13/viper.(*Viper).GetString: pkg/token/token.go 中获取 jwt_secret 的值
12. vendor/github.com/spf13/viper.(*Viper).Get: pkg/token/token.go中获取 jwt_secret 的值

上面的列表中可以看到有 ServeHTTP 字样的函数，这些函数是 gin/http 自带的函数，需要的函数，无法进行优化，所以上述列表没有列出。可以看到主要是 Gin middleware 耗时较久，这里处理方法是删除不需要的 Gin middleware，删除 Middleware 如下：

1. middleware.RequestId（main.go 文件中）

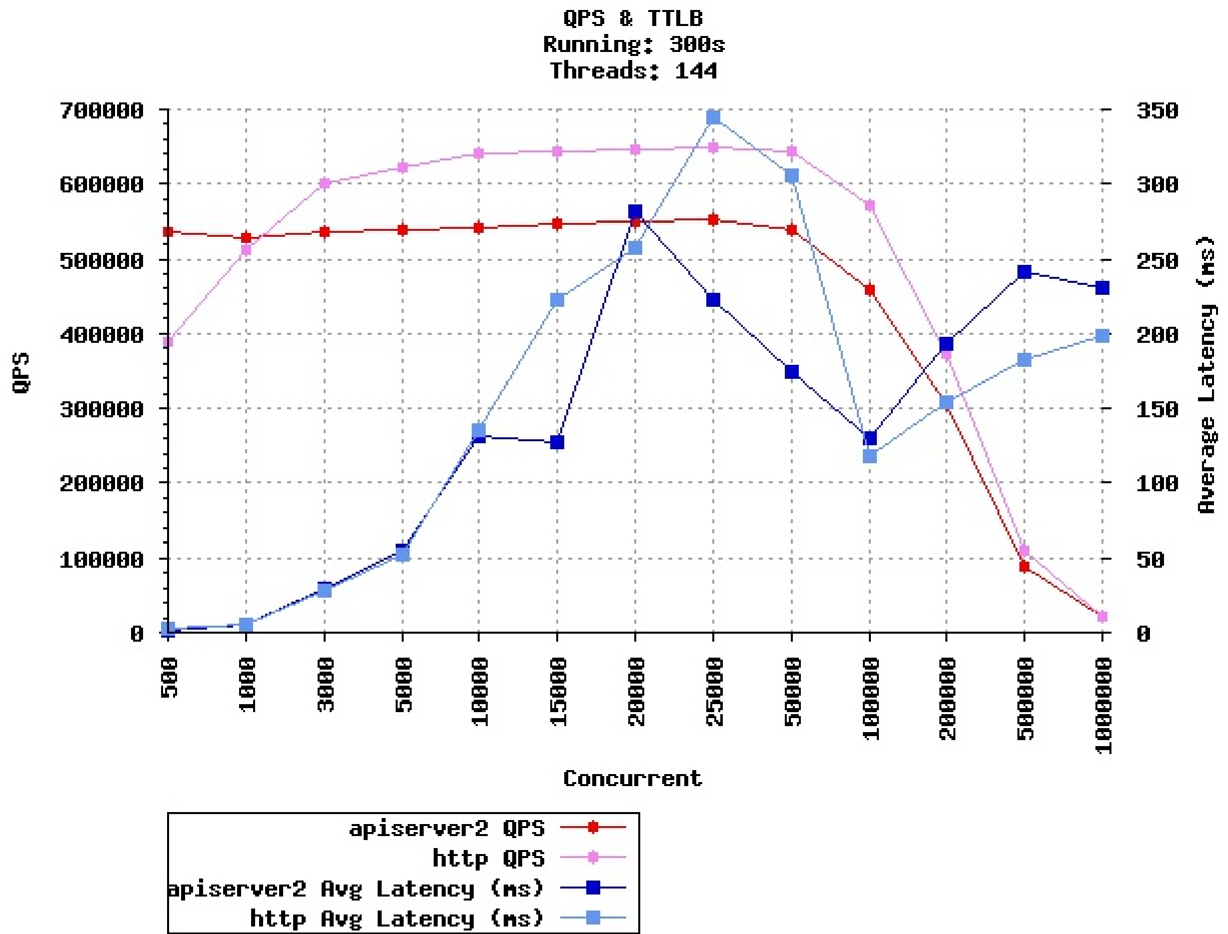
```
    // Routes.
    router.Load(
        // Cores.
        g,

        // Middlwares.
        //middleware.Logging(),
        //middleware.RequestId(),
    )
```

2. middleware.NoCache 和 middleware.Options（router/router.go 文件中）

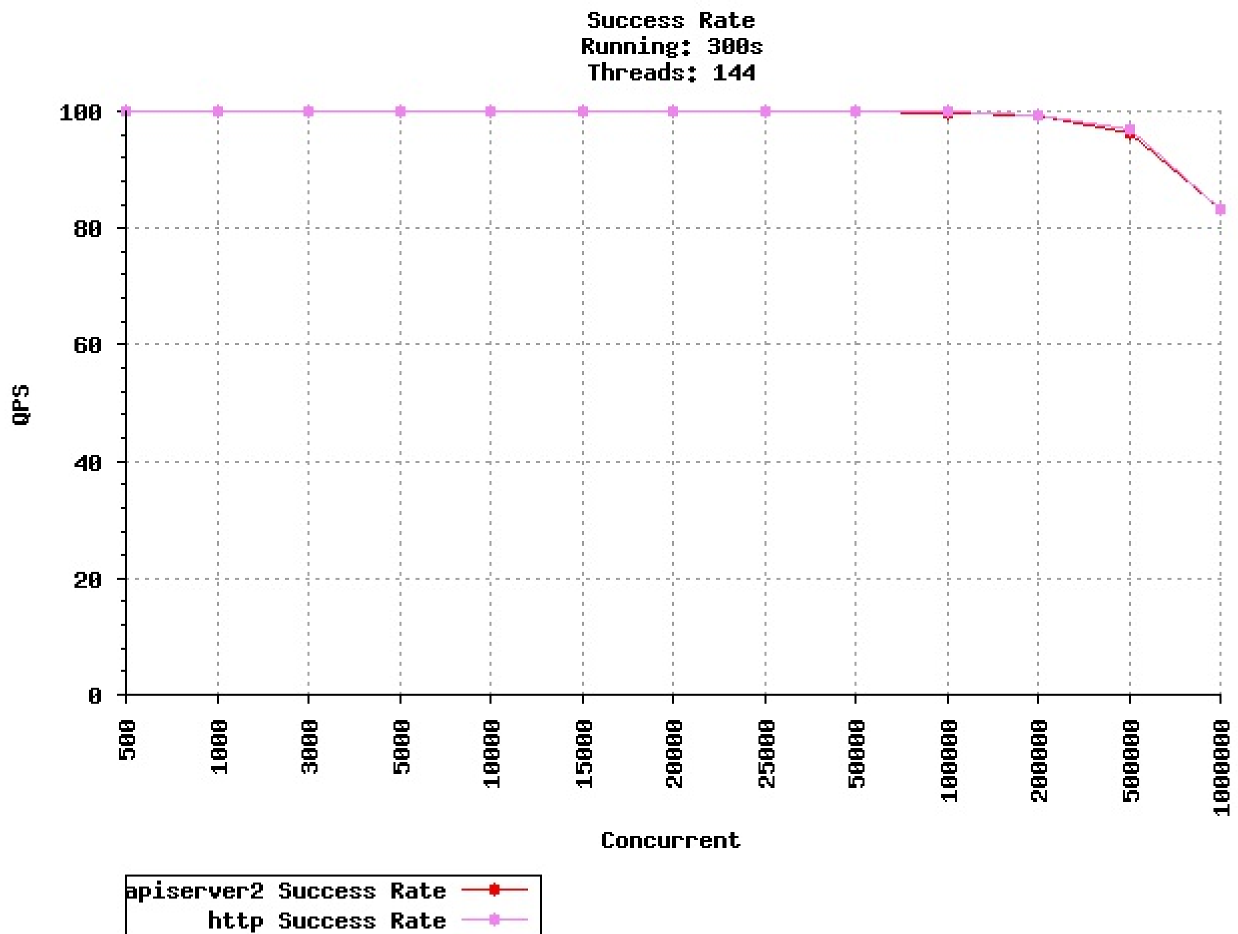
```
// Load loads the middlewares, routes, handlers.
func Load(g *gin.Engine, mw ...gin.HandlerFunc) *gin.Engine {
    // Middlewares.
    g.Use(gin.Recovery())
    //g.Use(middleware.NoCache)
    //g.Use(middleware.Options)
    g.Use(middleware.Secure)
    g.Use(mw...)
    // 404 Handler.
    g.NoRoute(func(c *gin.Context) {
        c.String(http.StatusNotFound, "The incorrect API route.")
    })
```

删除无用的 Gin middleware 重新编译 apiserver，启动 apiserver，测试性能后，再跟原生的 HTTP Server 对比，结果如下。

QPS & 平均响应时间对比



成功率对比



可以看到删除无用 Gin middleware 后，apiserver 的性能有了很大的提升，并发数为 25000 时，QPS 最大，为 553335（实际上并发数为 50000 时依然能达到很高的 QPS: 538144），响应时间为 222.91ms，QPS 很高，是原生 HTTP Server 的 85.34%。成功率基本跟原生的 HTTP Server 一致。优化后的 API 服务器可以支持很高的并发，在 20w+ 的并发下，API 服务器请求成功率可以达到 99.16%。这些性能远远好于企业级 API 服务器的要求。

性能测试自动化

本小节的性能测试脚本请参考最终源码目录下的 `wrktest.sh` 脚本。脚本大致流程是：先执行 wrk 测试，收集测试数据，格式化测试数据，最后调用 gnuplot 生成图表。

确保系统安装了 gnuplot，如果没有安装，CentOS 系统中可通过如下命令安装：

```
yum -y install gnuplot
```

附件：API 性能测试数据

原生的 HTTP Server 性能数据

```
并发数 QPS 平均响应时间(毫秒) 成功率

200 107975.55 2.18 100.00
500 387894.92 2.52 100.00
1000 512223.67 5.89 100.00
3000 599781.96 27.75 100.00
5000 623458.30 52.72 100.00
10000 640701.55 134.92 100.00
15000 644269.17 222.44 100.00
20000 646675.63 257.28 100.00
25000 648380.78 344.17 100.00
50000 642420.16 305.24 99.99
100000 572197.78 118.41 99.84
200000 372247.81 154.68 99.31
500000 110261.20 181.90 96.90
1000000 20954.71 198.87 83.15
```

优化前 apiserver 性能数据

```
并发数 QPS 平均响应时间(毫秒) 成功率

200 145651.44 1.28 100.00
500 143562.75 4.30 100.00
1000 144860.93 8.79 100.00
3000 147833.70 30.91 100.00
5000 146953.40 52.75 100.00
10000 144015.46 108.19 100.00
15000 140960.03 164.76 100.00
20000 140586.00 218.57 100.00
25000 140783.38 272.12 100.00
50000 139312.92 462.26 99.93
100000 127629.61 419.98 99.28
200000 90035.14 483.31 97.14
500000 25118.75 743.60 86.08
1000000 12304.60 819.44 71.78
```

优化后 apiserver 性能数据

```
并发数 QPS 平均响应时间(毫秒) 成功率

200 540539.65 0.44213 100.00
500 536362.78 1.89 100.00
1000 529081.92 5.61 100.00
3000 535506.99 30.00 100.00
5000 539251.92 55.10 100.00
10000 541375.64 131.69 100.00
15000 547164.96 127.14 100.00
20000 550434.18 282.16 100.00
25000 553335.38 222.91 100.00
50000 538144.69 174.27 99.98
100000 457695.17 130.28 99.80
200000 305915.69 193.37 99.16
500000 87672.39 241.65 96.14
1000000 20351.67 231.12 82.99
```

总结

本小节介绍了如何进行 API 的性能测试，并给出了本小册 apiserver 的性能数据，最后笔者附上了自己测试用的自动化测试脚本。

本小节介绍的是框架的性能，具体到某个接口的性能，因为影响因素比较多，需要读者自己去优化，这里给出 HTTP 接口性能要求，供读者在优化时参考。

指标名称	要求	优先级
响应时间	500 ms	1
请求成功率	99%	2
QPS	在满足预期要求的情况下服务器状态稳定，单台服务器 QPS 要求在 1000+	3