

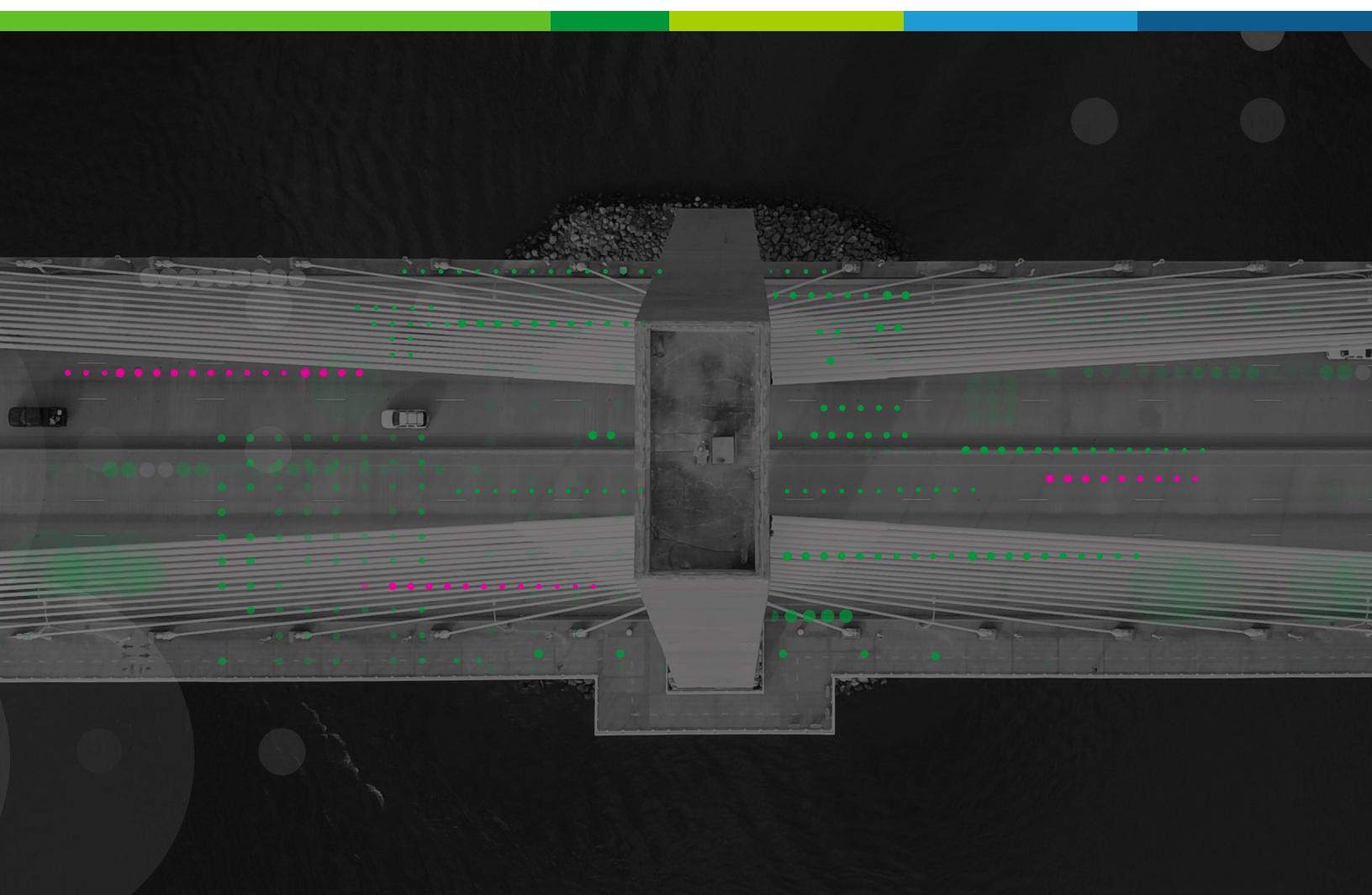


NGINX[®]
Part of F5

电子书

将 NGINX 部署为 API 网关

作者: Liam Crilly, F5 公司



目录

前言	3
NGINX 目前是如何用作 API 网关的?	4
为什么 NGINX 是赋能 API 网关的理想选择?	6
进一步增强 NGINX 解决方案	7
NGINX Management Suite 和 API 网关功能	7
1. 从 NGINX 和 API 网关开始.....	8
Warehouse API 简介	8
组织 NGINX 配置.....	9
定义顶层 API 网关	10
单体服务与微服务 API 的后端	12
定义 Warehouse API	13
响应错误	16
实施身份验证.....	18
总结	20
2. 保护后端服务	21
速率限制	21
执行特定的请求方法	24
应用细粒度的访问控制	25
控制请求大小	29
验证请求正文	30
总结	33
3. 发布 gRPC 服务	34
定义 gRPC 网关	35
运行示例 gRPC 服务	36
响应错误	40
使用 gRPC 元数据验证客户端	42
实施健康检查	42
应用速率限制和其他 API 网关控制	43
总结	44
附录:	
设置 gRPC 测试环境	45

一种更轻量、更灵活的方法应运而生，其中走在最前沿的是 NGINX

前言

应用开发和应用交付关系到当今大多数企业的业务发展大计，尤其是对各个领域的大型现有企业和创新初创公司来说。

为了应对层出不穷的应用开发和交付挑战，各种旨在解决特定问题的单点解决方案走上舞台，典型的例子有硬件负载均衡器，又称为应用交付控制器（ADC，比如 F5 BIG-IP 和 Citrix ADC），还有 Akamai 等内容分发网络（CDN）以及 Apigee 等 API 管理工具。

除了令人眼花缭乱的单点解决方案之外，还有一种更轻量、更灵活的方法，其中走在最前沿的是 NGINX。随着经济的通用服务器硬件性能的提升，企业开始使用简单、灵活的软件从单一端点轻松管理任务。

因此，如今有很多企业都使用 NGINX 负载均衡功能来补充甚至完全替代现有的硬件 ADC。他们使用 NGINX 实施多级缓存，甚至[开发自己的 CDN](#) 来补充或替代商用 CDN（大多数商用 CDN 的核心都是 NGINX）。

不断增长的通用硬件性能与 NGINX 稳步扩展的功能集强强联合，势不可挡，API 网关用例不得不开始做出让步。与 CDN 一样，许多现有的 API 管理工具都建立在 NGINX 之上。

在本电子书中，我们介绍了如何将现有的 NGINX 开源版或 F5 NGINX Plus 配置运用到 API 流量管理中。通过使用 NGINX 进行 API 管理，您可以亲身体验到 NGINX 家喻户晓的高性能、可靠性、强大的社区支持和专业的专家支持（面向 NGINX Plus 客户）。

有鉴于此，我们建议您尽量使用本书中描述的技术获取 NGINX 功能，然后借助补充解决方案添加其他特殊功能。

在这篇前言中，我们将先看一下 NGINX 是如何为现有 API 网关和 API 管理方法提供潜在的补充和替换方案的。然后再介绍电子书的其余部分，了解如何使用 NGINX 实现许多重要的 API 网关功能。

NGINX 目前是如何用作 API 网关的？

现在，企业可以通过三种不同的方式将 NGINX 部署为 API 网关：

- **NGINX 原生功能** —— 企业经常直接使用 NGINX 的功能管理 API 流量。他们认识到 API 流量采用的是 HTTP 或 gRPC 协议，因此利用 NGINX 配置对 API 请求进行接收、路由、速率限制和保护，以满足自己的 API 管理需求。
- **使用 Lua 扩展 NGINX** —— 基于 NGINX 的 OpenResty 软件将 Lua 解释器添加到了 NGINX 内核中，允许用户在 NGINX 之上构建丰富的功能。Lua 模块也可以进行编译并加载到 vanilla NGINX 开源版和 NGINX Plus build 版本中。[GitHub 上有几十种基于 NGINX 开源版的 API 网关实现](#)，其中很多都使用了 Lua 和 OpenResty。
- **第三方独立 API 网关** —— 独立的 API 网关是仅专注于处理 API 流量的单功能产品。大多数独立的 API 网关产品都在内核中使用 NGINX 和 Lua —— 无论它们是专门的开源解决方案还是商用产品。

大多数独立的 API 网关产品都在内核中使用 NGINX 和 Lua

对此，我们有两个建议：

- **关注性能和请求延迟** —— Lua 是扩展 NGINX 的有效方式，但是会牺牲一定的 NGINX 性能。我们的内部测试表明，简单的 Lua 脚本会导致 NGINX 性能下降 50% 到 90%。如果您选择的解决方案严重依赖 Lua，请务必核实它能否在不增加请求延迟的情况下满足您的峰值性能要求。
- **采用融合方法** —— NGINX 能够管理 API 流量以及常规的 Web 流量。API 网关只是 NGINX 的其中一项功能。（在某些情况下，API 网关产品包含 Lua 代码，能够复制 NGINX 中的功能，但潜在的可靠性和性能较差。）独立的单功能 API 网关会限制您的架构灵活性，因为使用单独的产品来管理 Web 和 API 流量会增加 DevOps、CI/CD、监控、安全保护以及其他应用开发和交付功能的复杂性。

融合方法在现代分布式应用环境中尤为重要。切记，NGINX 不仅可以充当 Web 和 API 流量的反向代理，而且还能用作缓存层、身份验证网关、Web 应用防火墙和应用网关。

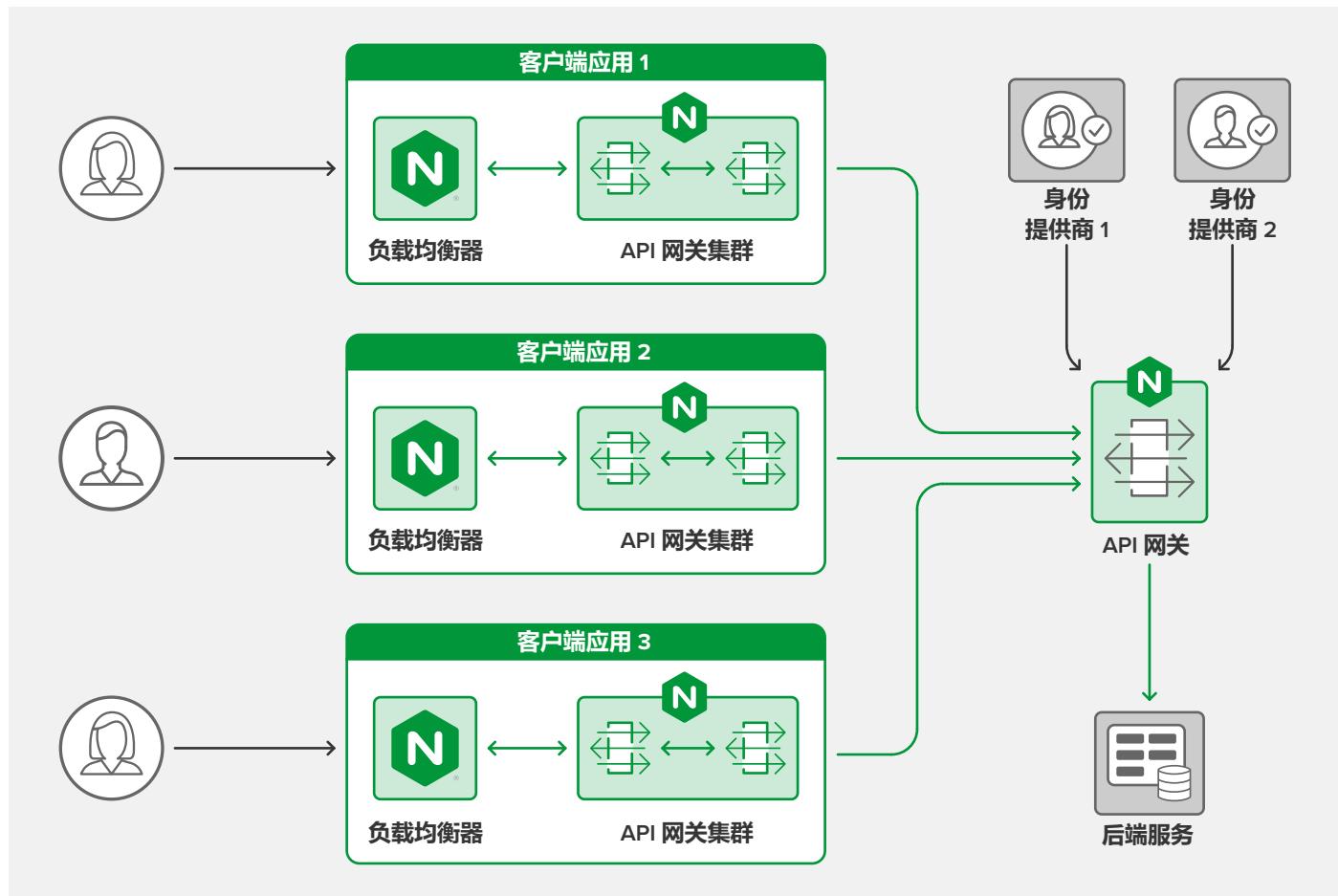


图 1：使用 NGINX 作为 API 网关时的应用架构示例

为什么 NGINX 是赋能 API 网关的理想选择？

下表分析了一些 API 网关用例，这些用例负责管理外部来源的 API 请求并将其路由到内部服务。

	API 网关用例	NGINX 反向代理
内核协议	REST (HTTPS) , gRPC	HTTP, HTTPS, HTTP/2, gRPC
其他协议	TCP-borne 消息队列	WebSocket, TCP, UDP
请求路由	根据服务（主机请求头）、API 方法 (HTTP URL) 和参数路由请求	可以根据主机请求头、URL 和请求消息的标头，非常灵活地进行请求路由
管理 API 生命周期	重写传统 API 请求，拒绝调用弃用的 API	全面的请求重写和丰富的决策引擎，能够直接路由或响应请求
保护易受攻击的应用	使用 API 和 API 方法速率限制	依据多个标准速率限制，包括源地址、请求参数、到后端服务的连接限制
卸载身份验证	对传入的请求查询身份验证令牌	多种身份验证方法，包括 JWT、API 密钥、OpenID Connect 以及其他外部身份验证服务
管理不断变化的应用拓扑	使用各种 API 接受配置变更并支持蓝绿工作流程	使用 API 和服务发现功能定位端点 (NGINX Plus)；对 API 进行编排，以支持蓝绿部署和其他用例

作为融合解决方案，NGINX 还可以轻松管理 Web 流量、在协议 (HTTP/2 和 HTTP、FastCGI、uwsgi) 之间转换并提供一致的配置和监控接口。NGINX 是一款十分轻量的软件，能够直接或者作为 sidecar 部署在容器环境中，并且占用的资源非常之少。

进一步增强 NGINX 解决方案

NGINX 一开始是为了充当 Web (HTTP) 流量网关开发的，配置时依据的 primitive 以 HTTP 请求的形式表示。这与您期望的 API 网关配置方式相似，但也不尽相同，因此 DevOps 工程师需要了解如何将 API 定义映射到 HTTP 请求。

对于简单的 API，这很好操作；对于复杂一些的情况，本书分三个章节描述了如何将复杂的 API 映射到 NGINX 中的服务，然后介绍了如何执行一些常见的任务，例如：

- 重写 API 请求
- 正确响应错误
- 管理 API 密钥，以支持访问控制
- 查询 JWT token，对用户进行身份验证
- 速率限制
- 执行特定的请求方法
- 应用细粒度的访问控制
- 控制请求大小
- 验证请求正文

NGINX Management Suite 和 API 网关功能

F5 NGINX Management Suite 是 NGINX 应用平台的控制平面，建立在 NGINX 和 NGINX Plus 的多功能特性之上，能够以简单且一致的方式提供强大的应用交付功能。

NGINX Management Suite 能够以对 API 友好的方式定义 API 管理策略，并将它们发布到跨应用平台部署的多功能 NGINX 网关上。如欲了解有关 NGINX Management Suite 的更多信息并获取免费试用版，请[访问我们的网站](#)。

1. 从 NGINX 和 API 网关开始

现代应用架构的核心是 HTTP API。HTTP 支持快速构建和轻松维护应用。HTTP API 提供了一个通用接口，因此不必考虑应用的规模大小，无论是单独用途的微服务还是大型综合应用。HTTP 不仅可以支持超大规模互联网，也可用于提供可靠和高性能的 API 交付。

要了解 API 网关对于微服务应用的重要性，请参阅我们的博文 [《构建微服务：使用 API 网关》](#)。

NGINX 拥有处理 API 流量所需的高级 HTTP 处理功能

作为领先的高性能、轻量级反向代理和负载均衡器，NGINX 拥有处理 API 流量所需的高级 HTTP 处理功能。这使得 NGINX 成为构建 API 网关的理想平台。本章描述了一些常见的 API 网关用例，并展示了如何以高效、可扩展和易于维护的方式配置 NGINX。我们描述了一套完整的配置，该配置可构成生产环境部署的基础。

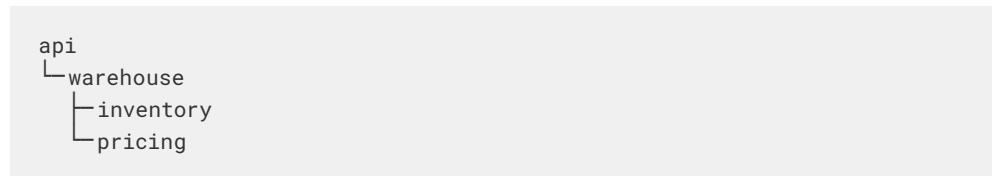
注：除非另有说明，否则本章中的所有信息都适用于 NGINX 开源版和 F5 NGINX Plus。

Warehouse API 简介

API 网关的主要功能是为多个 API（无论它们在后端如何实施或部署）提供统一的一致的入口点。并非所有 API 都是微服务应用。我们的 API 网关需要管理现有的 API、单体应用和正在过渡到微服务的应用。

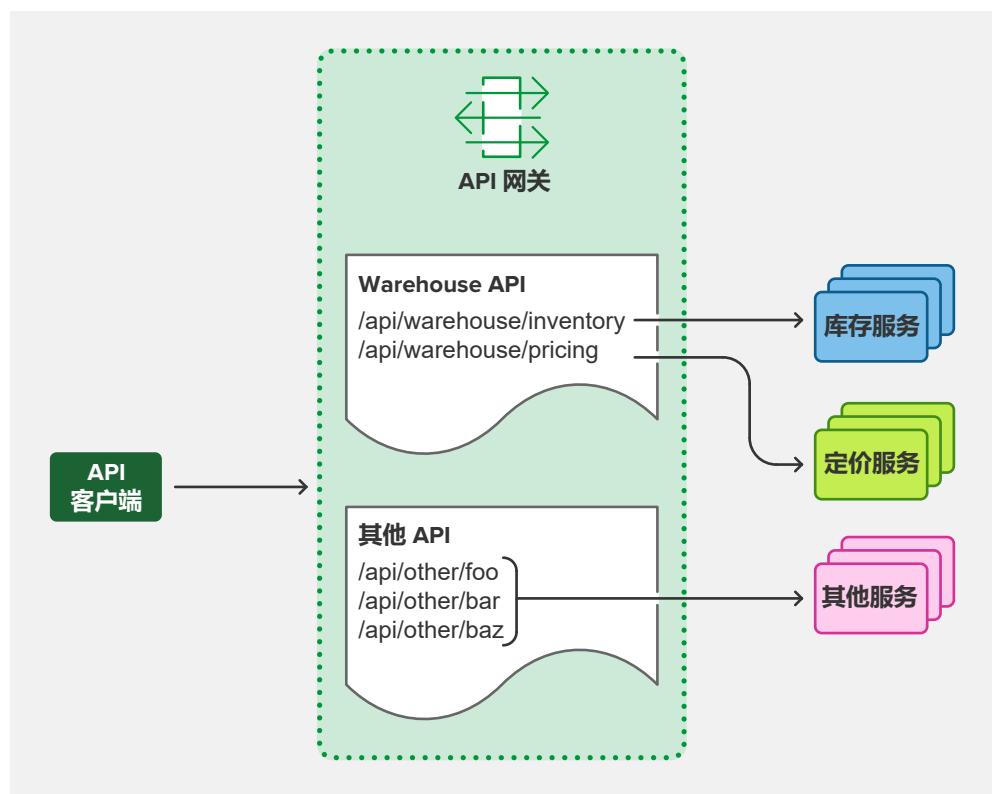
本章假定了一个用于库存管理的 API，名为“Warehouse API”。我们使用 NGINX 示例配置来说明不同的用例。Warehouse API 是一个 RESTful API，它接收 JSON 请求并且生成 JSON 响应。不过，在部署为 API 网关时，NGINX 并不限制只能使用 JSON，因为 NGINX 的部署与架构风格和 API 本身使用的数据格式是无关的。

Warehouse API 是由几个独立的微服务集合在一起后，作为单个 API 发布而实现的。库存和价格资源分别由不同的服务实现并部署到不同的后端。因此，API 的路径结构是：



举例来说，如要查询当前的仓库库存，客户端应用将向 `/api/warehouse/inventory` 发送 HTTP GET 请求。

图 2：面向多个应用的 API 网关架构



组织 NGINX 配置

如果 NGINX 已经是应用交付架构的一部分，那么通常不需要再部署一个独立的 API 网关

将 NGINX 用作 API 网关的一个优势是，它不仅可以很好地担任 API 网关这一角色，同时还可以充当现有 HTTP 流量的反向代理、负载均衡器和 Web 服务器。如果 NGINX 已经是应用交付架构的一部分，那么通常不需要再部署一个独立的 API 网关。然而，API 网关的一些默认行为与基于浏览器流量的行为有所不同。因此，我们将 API 网关配置与基于浏览器的流量的任何现有（或未来）配置分离。

为了实现这种分离，我们创建了一个支持多用途 NGINX 实例的配置，并提供了一个易于使用的结构，用于通过 CI/CD 流水线实现自动化配置部署。`/etc/nginx` 下的生成目录结构如下所示。

```
etc/
└── nginx/
    ├── api_conf.d/.....Subdirectory for per-API configuration
    │   └── warehouse_api.conf.....Definition and policy of the Warehouse API
    ├── api_backends.conf.....The backend services (upstreams)
    ├── api_gateway.conf.....Top-level configuration for the API gateway server
    ├── api_json_errors.conf.....HTTP error responses in JSON format
    ├── conf.d/
    │   └── ...
    │       └── existing_apps.conf
    └── nginx.conf
```

所有 API 网关配置的目录和文件名的前缀都是 `api_`。每个文件和目录支持一个不同的 API 网关特性或功能，下文将进行详细解释。`warehouse_api.conf` 文件是下文讨论的配置文件（这些文件以不同方式定义 Warehouse API）的通用“范例”。

定义顶层 API 网关

所有 NGINX 的配置都先从主配置文件 `nginx.conf` 开始。为了读取 API 网关配置，我们在 `nginx.conf` 的 `http` 块中定义了一个 `include` 指令，该指令引用包含网关配置的文件 `api_gateway.conf`（下面的第 28 行）。请注意，默认的 `nginx.conf` 文件使用 `include` 指令从 `conf.d` 子目录（第 29 行）中拉取基于浏览器的 HTTP 配置。本章使用了大量 `include` 指令来提高可读性及实现部分配置的自动化。

```
28 include /etc/nginx/api_gateway.conf;          # 所有 API 网关配置
29 include /etc/nginx/conf.d/*.conf;              # 常规 Web 流量
```

[在 GitHub 上查看原始数据](#)

以下 `api_gateway.conf` 文件定义了将 NGINX 作为 API 网关暴露给客户端的 virtual server。此配置在单个入口点 `https://api.example.com/` (第 8 行) 暴露 API 网关发布的所有 API，这些 API 受第 12 行到第 16 行配置的 TLS 保护。请注意，此配置是纯 HTTPS —— 没有明文 HTTP 监听器。我们假定 API 客户端知道正确的入口点并默认建立的是 HTTPS 连接。

此配置是静态的 —— 各个 API 及其后端服务的详细配置在第 20 行 `include` 指令引用的文件中指定。第 23 行至第 26 行涉及错误处理，将在下面的 “[响应错误](#)” 部分进行讨论。

```
1  include api_backends.conf;
2  include api_keys.conf;
3
4  server {
5      access_log /var/log/nginx/api_access.log main;    # 每个 API 也可以
6                                # 记录到单独文件
7
8      listen 443 ssl;
9      server_name api.example.com;
10
11     # TLS config
12     ssl_certificate      /etc/ssl/certs/api.example.com.crt;
13     ssl_certificate_key  /etc/ssl/private/api.example.com.key;
14     ssl_session_cache    shared:SSL:10m;
15     ssl_session_timeout   5m;
16     ssl_ciphers          HIGH:!aNULL:!MD5;
17     ssl_protocols        TLSv1.2 TLSv1.3;
18
19     # API 定义，每个文件一个
20     include api_conf.d/*.conf;
21
22     # 错误响应
23     error_page 404 = @400;          # 将无效路径视为错误请求
24     proxy_intercept_errors on;     # 不向客户端发送后端错误
25     include api_json_errors.conf;  # API 客户端友好的 JSON 错误
26     default_type application/json; # 如果没有内容类型，则假定为 JSON
27 }
```

[在 GitHub 上查看原始数据](#)

单体服务与微服务 API 的后端

一些 API 由单个后端实现，尽管出于弹性或负载均衡方面的考虑，我们通常希望有多个后端。我们通过微服务 API 为每个 service 定义单独的后端；它们共同形成完整的 API 功能。此处，Warehouse API 被部署为两个独立的 service，每个 service 都有多个后端。

```
1 upstream warehouse_inventory {
2     zone inventory_service 64k;
3     server 10.0.0.1:80;
4     server 10.0.0.2:80;
5     server 10.0.0.3:80;
6 }
7
8 upstream warehouse_pricing {
9     zone pricing_service 64k;
10    server 10.0.0.7:80;
11    server 10.0.0.8:80;
12    server 10.0.0.9:80;
13 }
```

[在 GitHub 上查看原始数据](#)

NGINX Plus 用户还可以利用
动态 DNS 负载均衡功能

由 API 网关发布的所有后端 API 服务都在 `api_backends.conf` 中定义。此处，我们在每个 `upstream` 块中使用多个“IP 地址 - 端口”组合（也可以使用主机名）来指示 API 代码的部署位置。NGINX Plus 用户还可以利用动态 DNS 负载均衡功能将新的后端自动添加到运行时配置中。

定义 Warehouse API

Warehouse API 由嵌套配置中的一些 `location` 块定义，如下例所示。外部 `location` (`/api/warehouse`) 标识基本路径，嵌套位置在该路径下的 URI，指定路由到后端 API service。我们可以使用外部块定义适用于整个 API 的通用策略（在此示例中，为第 6 行的日志记录配置）。

```
1  # Warehouse API
2  #
3  location /api/warehouse/ {
4      # 这里的策略配置（身份验证、速率限制、日志记录……）
5      #
6      access_log /var/log/nginx/warehouse_api.log main;
7
8      # URI 路由
9      #
10     location /api/warehouse/inventory {
11         proxy_pass http://warehouse_inventory;
12     }
13
14     location /api/warehouse/pricing {
15         proxy_pass http://warehouse_pricing;
16     }
17
18     return 404; # Catch-all
19 }
```

[在 GitHub 上查看原始数据](#)

NGINX 拥有一个高效而又灵活的系统，用于匹配请求 URI

NGINX 拥有一个高效而又灵活的系统，用于将请求 URI 与配置的一部分相匹配。`location` 指令的顺序并不重要，系统会选择匹配度最高的指令。此处，第 10 行和第 14 行的嵌套位置定义了两个比外部 `location` 块更具体的 URI；每个嵌套块中的 `proxy_pass` 指令将请求路由到适当的 `upstream group`。除非需要为某些 URI 提供更具体的策略，否则策略配置从外部 `location` 继承。

任何与其中的一个嵌套位置不匹配的 URI 都由外部 `location` 处理，其中包括一个 `catch-all` 指令（第 18 行），该指令为所有无效 URI 返回响应 404 (Not Found)。

为 API 选择宽泛定义或精确定义

API 有两种定义方法——宽泛和精确。最合适的定义方法取决于 API 的安全要求，以及后端 service 是否需要处理无效 URI。

在上面的“[定义 Warehouse API](#)”中，我们对 Warehouse API 使用了宽泛定义方法，在第 10 行和第 14 行定义了 URI 前缀，这样以其中一个前缀开头的 URI 就会被代理到适当的后端 service。通过这种宽泛的、基于前缀的 location 匹配，对以下 URI 的 API 请求都是有效的：

```
/api/warehouse/inventory  
/api/warehouse/inventory/  
/api/warehouse/inventory/foo  
/api/warehouse/inventoryfoo  
/api/warehouse/inventoryfoo/bar/
```

如果只需考虑将每个请求代理到正确的后端 service，则宽泛的定义方法可提供最快的处理速度和最紧凑的配置。另一方面，更精确的方法可以显式定义每个可用 API 资源的 URI 路径，从而使 API 网关能够了解 API 完整的 URI 空间。通过采用精确定义方法，Warehouse API 中的以下 URI 路由配置可使用精确匹配 (=) 和正则表达式 (~) 组合来定义每个有效的 URI。

```
8  # URI 路由  
9  #  
10 location = /api/warehouse/inventory {           # 全部库存  
11     proxy_pass http://warehouse_inventory;  
12 }  
13  
14 location ~ ^/api/warehouse/inventory/shelf/[^\]+$ {  # 货架库存  
15     proxy_pass http://warehouse_inventory;  
16 }  
17  
18 location ~ ^/api/warehouse/inventory/shelf/[^\]+/box/[^\]+$ { # Box on shelf  
19     proxy_pass http://warehouse_inventory;  
20 }  
21  
22 location ~ ^/api/warehouse/pricing/[^\]+$ {      # 具体项目的价格  
23     proxy_pass http://warehouse_pricing;  
24 }
```

[在 GitHub 上查看原始数据](#)

这种配置较为冗长，但更准确地描述了后端 service 实现的资源。这样做的好处是可以保护后端 service 免受格式不正确的客户端请求的影响，而代价是产生少许额外的正则表达式匹配开销。有了此配置，NGINX 将接受一些 URI 并拒绝其他无效的 URI：

有效的 URI	无效的 URI
/api/warehouse/inventory	/api/warehouse/inventory/
/api/warehouse/inventory/shelf/foo	/api/warehouse/inventoryfoo
/api/warehouse/inventory/shelf/foo/box/bar	/api/warehouse/inventory/shelf
/api/warehouse/inventory/shelf/-/box/-	/api/warehouse/inventory/shelf/foo/bar
/api/warehouse/pricing/baz	/api/warehouse/pricing
	/api/warehouse/pricing/baz/pub

通过使用精确的 API 定义，现有的 API 归档格式可驱动 API 网关的配置。可以实现通过 [OpenAPI 规范](#)（以前称为 Swagger）自动定义 NGINX API。本章的 Gists 中提供了一个用于此目的的[示例脚本](#)。

重写客户端请求以处理重大变更

有时需要进行变更，会打破严格的向后兼容性

随着 API 的发展，有时需要进行变更，会打破严格的向后兼容性并要求更新客户端。例如重命名或移动某个 API 资源的时候，与 Web 浏览器不同，API 网关无法向客户端发送重定向（代码 301 (Moved Permanently)）来命名新位置。幸运的是，如果无法修改 API 客户端，我们可以动态地重写客户端请求。

在下面的示例中，我们使用与上文“[定义 Warehouse API](#)”相同的宽泛定义方法，但在本例中，配置替换了以前版本的 Warehouse API，其中定价 service 作为库存 service 的一部分实现。第 3 行的 `rewrite` 指令将对旧定价资源的请求转换为对新定价 service 的请求。

```
1 # 重写规则
2 #
3 rewrite ^/api/warehouse/inventory/item/price/(.*) /api/warehouse/pricing/$1;
4
5 # Warehouse API
6 #
7 location /api/warehouse/ {
8     # 这里的策略配置（身份验证、速率限制、日志记录……）
9     #
10    access_log /var/log/nginx/warehouse_api.log main;
11
12    # URI 路由
13    #
14    location /api/warehouse/inventory {
15        proxy_pass http://warehouse_inventory;
16    }
17
18    location /api/warehouse/pricing {
19        proxy_pass http://warehouse_pricing;
20    }
21
22    return 404; # Catch-all
23 }
```

[在 GitHub 上查看原始数据](#)

响应错误

HTTP API 和基于浏览器的流量之间的一个关键区别是如何将错误传递给客户端。当 NGINX 部署为 API 网关时，我们将其配置为以最适合 API 客户端的方式返回错误。
将错误传递给客户端

HTTP API 和基于浏览器的流量之间的一个关键区别是如何将错误传递给客户端。当 NGINX 部署为 API 网关时，我们将其配置为以最适合 API 客户端的方式返回错误。

顶层 API 网关配置包含了定义如何处理错误响应的部分。

```
22 # 错误响应
23 error_page 404 = @400;          # 将无效路径视为错误请求
24 proxy_intercept_errors on;      # 不向客户端发送后端错误
25 include api_json_errors.conf;   # API 客户端友好的 JSON 错误
26 default_type application/json; # 如果没有内容类型，则假定为 JSON
```

[在 GitHub 上查看原始数据](#)

未处理的响应异常可能包含堆
栈跟踪或其他敏感数据

第 23 行的 `error_page` 指令定义了当请求与任何 API 定义都不匹配时，NGINX 返回 400 (Bad Request) 错误，而不是默认的 404 (Not Found) 错误。此（可选）行为要求 API 客户端仅发出 API 文档中包含的有效 URI 的请求，并防止未经授权的客户端发现通过 API 网关发布的 API 的 URI 结构。

第 24 行涉及后端 service 本身产生的错误。未处理的后端 service 的响应异常可能包含堆栈跟踪或其他我们不想发送给客户端的敏感数据。此配置可向客户端发送标准化错误响应，进一步增加了防护级别。

标准化错误响应的完整列表在第 25 行的 `include` 指令引用的单独配置文件中定义，其中的前几行如下所示。如果首选是 JSON 以外的格式，则可以修改此文件，将 `api_gateway.conf` 第 26 行的 `default_type` 值更改为匹配值。您还可以在每个 API 的策略部分添加一个单独的 `include` 指令，以引用不同的错误响应文件，这些文件会覆盖全局响应。

```
1 error_page 400 = @400;
2 location @400 { return 400 '{"status":400,"message":"Bad request"}\n'; }
3
4 error_page 401 = @401;
5 location @401 { return 401 '{"status":401,"message":"Unauthorized"}\n'; }
6
7 error_page 403 = @403;
8 location @403 { return 403 '{"status":403,"message":"Forbidden"}\n'; }
9
10 error_page 404 = @404;
11 location @404 { return 404 '{"status":404,"message":"Resource not found"}\n'; }
```

[在 GitHub 上查看原始数据](#)

有了此配置，对无效 URI 的客户端请求将收到以下响应。

```
$ curl -i https://api.example.com/foo
HTTP/1.1 400 Bad Request
Server: nginx/1.21.3
Content-Type: application/json
Content-Length: 39
Connection: keep-alive

{"status":400,"message":"Bad request"}
```

NGINX 提供了多种方法来保护 API 和认证 API 客户端

实施身份验证

不通过某种形式的身份验证就发布 API 的情况较为罕见。NGINX 提供了多种方法来保护 API 和认证 API 客户端。要了解同样适用于常规 HTTP 请求的方法，请参阅[基于 IP 地址的访问控制列表（ACL）](#)、[数字证书身份认证](#)和[HTTP Basic 认证](#)的文档。此处，我们重点介绍适用 API 的身份验证方法。

API 密钥是客户端知道的共享密钥

API 密钥身份验证

API 密钥是客户端和 API 网关的共享密钥。API 密钥本质上是一个作为长期凭证发给 API 客户端的长而复杂的密码。创建 API 密钥很简单——只需像本例中那样编码产生一个随机数。

```
$ openssl rand -base64 18  
7B5zIqmRGXmrJTFmKa99vcit
```

在顶层 API 网关配置文件 `api_gateway.conf` (如“[定义 API 网关](#)”中所示) 的第 2 行，我们使用 `include` 指令添加了以下名为 `api_keys.conf` 的文件，其中包含每个 API 客户端的 API 密钥，并由客户端名称或其他描述加以标识。以下是该文件的内容：

```
1 map $http_apikey $api_client_name {  
2     default "";  
3  
4     "7B5zIqmRGXmrJTFmKa99vcit" "client_one";  
5     "QzV6y1EmQFbbx0fRCwyJs35" "client_two";  
6     "mGcjH8Fv6U9y3BVF9H3Ypb9T" "client_six";  
7 }
```

[在 GitHub 上查看原始数据](#)

API 密钥在 `map` 块中定义。`Map` 指令有两个参数。第一个参数定义在何处查找 API 密钥，本例中是在客户端请求的 `apikey` HTTP 包头中，该包头于 `$http_apikey` 变量中捕获。第二个参数创建一个新变量 (`$api_client_name`)，并将其设置为第一个参数与密钥匹配行的第二个参数的值。

例如，当客户端请求中带有 API 密钥 `7B5zIqmRGXmrJTFmKa99vcit` 时，`$api_client_name` 变量设置为 `client_one`。此变量可用于检查经过验证的客户端并包含在日志条目中以进行更详细的审核。`Map` 块的格式非常简单，容易集成到从已有凭证存储生成 `api_keys.conf` 文件的自动化工作流中。

此处，我们通过修改 “[定义 Warehouse API](#)” 中的“宽泛”配置，在策略部分添加一个 `auth_request` 指令（将身份验证决策委托给指定 location），从而启用 API 密钥身份验证。

```
1  # Warehouse API
2  #
3  location /api/warehouse/ {
4      # 这里的策略配置（身份验证、速率限制、日志记录……）
5      #
6      access_log /var/log/nginx/warehouse_api.log main;
7      auth_request /_validate_apikey;
8
9      # URI 路由
10     #
11     location /api/warehouse/inventory {
12         proxy_pass http://warehouse_inventory;
13     }
14
15     location /api/warehouse/pricing {
16         proxy_pass http://warehouse_pricing;
17     }
18
19     return 404; # Catch-all
20 }
```

[在 GitHub 上查看原始数据](#)

例如，通过 `auth_request` 指令（第 7 行），我们可以让外部身份认证服务器（例如 [OAuth 2.0 token introspection](#)）处理身份验证。在此示例中，我们将验证 API 密钥的逻辑添加到顶层 API 网关配置文件中，其形式为以下名为 `/_validate_apikey` 的 location 块。

```
28      # API 密钥验证
29      location = /_validate_apikey {
30          internal;
31
32          if ($http_apikey = "") {
33              return 401; # Unauthorized
34          }
35          if ($api_client_name = "") {
36              return 403; # 禁止
37          }
38
39          return 204; # 成功（无内容）
40      }
```

[在 GitHub 上查看原始数据](#)

第 30 行的 `internal` 指令意味着外部客户端不能直接访问此位置（只能由 `auth_request` 访问）。客户端应在 `apikey` HTTP 包头中显示其 API 密钥。如果此标头丢失或为空（第 32 行），我们将发送 401 (Unauthorized) 响应，告知客户端需要进行身份验证。第 35 行处理 API 密钥与 `map` 块中的任何密钥都不匹配的情况——在这种情况下，`api_keys.conf` 中第 2 行的 `default` 参数将 `$api_client_name` 设置为空字符串，我们将发送 403 (Forbidden) 响应，告诉客户端身份验证失败。如果这些条件都不匹配，则 API 密钥有效并且该 `location` 返回 204 (No Content) 响应。

有了此配置，Warehouse API 现在实现了 API 密钥身份验证。

```
$ curl https://api.example.com/api/warehouse/pricing/item001
{"status":401,"message":"Unauthorized"}

$ curl -H "apikey: thisIsInvalid" https://api.example.com/api/
warehouse/pricing/item001
{"status":403,"message":"Forbidden"}

$ curl -H "apikey: 7B5zIqmRGXmrJTFmKa99vcit" https://api.example.com/
api/warehouse/pricing/item001
{"sku":"item001","price":179.99}
```

JWT 身份验证

JSON Web Token (JWT) 被越来越多地用于 API 身份验证。原生 JWT 支持是 NGINX Plus 的独有功能，支持验证 JWT，详见我们的博文《借助 JWT 和 NGINX Plus 验证 API 客户端》。有关示例实现，请参阅第 2 章中的“控制对特定方法的访问”。

总结

本章详细介绍了将 NGINX 部署为 API 网关的完整解决方案。您可前往我们的 [GitHub Gist](#) repo 查看和下载本章中提及的完整文件集。

2. 保护后端服务

本章对第 1 章中介绍的用例进行了扩展，探讨了一系列可用于保护生产环境中后端 API 服务的安全措施：

- 速率限制
- 执行特定的请求方法
- 应用细粒度的访问控制
- 控制请求大小
- 验证请求正文

注：除非另有说明，否则本章中的所有信息都适用于 NGINX 开源版和 NGINX Plus。

速率限制

单个 API 客户端就能够给您的
API 造成巨大的负载

与基于浏览器的客户端不同，单个 API 客户端就能够给您的 API 造成巨大的负载，甚至会消耗大量的系统资源，以致其他 API 客户端因此被“排挤”。不仅恶意客户端会构成这种威胁，行为异常或存在缺陷的 API 客户端也可能会反复压垮后端。为了防止出现这种情况，我们用速率限制来确保每个客户端合理使用 API 并保护后端服务的资源。

NGINX 可以根据请求的任何属性应用速率限制。通常使用客户端 IP 地址，但如果为 API 启用身份验证，则经过身份验证的客户端 ID 将是更为可靠和准确的属性。

速率限制本身在顶层 API 网关配置文件中定义，并且可以全局、按每个 API 甚至每个 URI 来应用。

```
1  include api_backends.conf;
2  include api_keys.conf;
3
4  limit_req_zone $binary_remote_addr zone=client_ip_10rs:1m rate=1r/s;
5  limit_req_zone $http_apikey          zone=apikey_200rs:1m    rate=200r/s;
6
7  server {
8      access_log /var/log/nginx/api_access.log main; # 每个 API 也可以
9                      # 记录到单独文件
10
11     listen 443 ssl;
12     server_name api.example.com;
13
14     # TLS config
15     ssl_certificate      /etc/ssl/certs/api.example.com.crt;
16     ssl_certificate_key  /etc/ssl/private/api.example.com.key;
17     ssl_session_cache    shared:SSL:10m;
18     ssl_session_timeout  5m;
19     ssl_ciphers          HIGH:!aNULL:!MD5;
20     ssl_protocols        TLSv1.2 TLSv1.3;
21
22     # API 定义，每个文件一个
23     include api_conf.d/*.conf;
24
25     # 错误响应
26     error_page 404 = @400;           # 将无效路径视为错误请求
27     proxy_intercept_errors on;       # 不向客户端发送后端错误
28     include api_json_errors.conf;   # API 客户端友好的 JSON 错误
29     default_type application/json; # 如果没有内容类型，则假定为 JSON
49 }
```

[在 GitHub 上查看原始数据](#)

在此示例中，第 4 行的 `limit_req_zone` 指令为每个客户端 IP 地址 (`$binary_remote_addr`) 定义每秒 10 个请求的速率限制，第 5 行的 `limit_req_zone` 指令为每个经过身份验证的客户端 ID (`$http_apikey`) 定义每秒 200 个请求的速率限制。该示例说明了我们可以定义多个速率限制，而不受它们所应用位置的约束。一个 API 可以同时应用多个速率限制，或者对不同的资源应用不同的速率限制。

在下面的配置段中，我们使用 `limit_req` 指令来应用第 1 章中描述的“Warehouse API”策略部分中的第一个速率限制。默认情况下，当超过速率限制阈值时，NGINX 会发送 503 (Service Unavailable) 响应。然而，让 API 客户端明确地知道自己已超过速率限制阈值，有助于它们调整自己的行为。为此，我们使用 `limit_req_status` 指令来发送 429 (Too Many Requests) 响应。

```
1  # Warehouse API
2  #
3  location /api/warehouse/ {
4      # 这里的策略配置 (身份验证、速率限制、日志记录.....)
5      #
6      access_log /var/log/nginx/warehouse_api.log main;
7      limit_req zone=client_ip_10rs;
8      limit_req_status 429;
9
10     # URI 路由
11     #
12     location /api/warehouse/inventory {
13         limit_except GET {
14             deny all;
15         }
16         error_page 403 = @405;    # 从'403 (Forbidden)'
17                                     # 到405的转换拒绝响应 (方法不被允许)'
18         proxy_pass http://warehouse_inventory;
19     }
20
21     location /api/warehouse/pricing {
22         limit_except GET PATCH {
23             deny all;
24         }
25         error_page 403 = @405;
26         proxy_pass http://warehouse_pricing;
27     }
28
29     return 404;      # Catch-all
30 }
```

[在 GitHub 上查看原始数据](#)

您可以使用 `limit_req` 指令的附加参数来微调 NGINX 执行速率限制的方式。例如，当超过速率限制阈值时，可以让请求排队而不是直接拒绝它们，从而使请求速率有时间降至定义的限制之下。有关微调速率限制阈值的更多信息，请参阅我们的博文《[使用 NGINX 和 NGINX Plus 实现速率限制](#)》。

对于 RESTful API, HTTP 方法是每个 API 调用的重要组成部分

部分

执行特定的请求方法

对于 RESTful API, HTTP 方法是每个 API 调用的重要组成部分, 对 API 定义非常重要。以 Warehouse API 的定价服务 service 为例:

- GET /api/warehouse/pricing/item001 returns the price of item001
- PATCH /api/warehouse/pricing/item001 changes the price of item001

我们可以更新 Warehouse API 中的 URI 路由定义, 以便在对定价 service 的请求中只接受这两个 HTTP 方法 (并且在对库存 service 的请求中只接受 GET 方法)。

```
10      # URI 路由
11      #
12      location /api/warehouse/inventory {
13          limit_except GET {
14              deny all;
15          }
16          error_page 403 = @405; # 从 '403 (Forbidden)'
17          # 到 '405 的转换拒绝请求 (方法不被允许)'
18          proxy_pass http://warehouse_inventory;
19      }
20
21      location /api/warehouse/pricing {
22          limit_except GET PATCH {
23              deny all;
24          }
25          error_page 403 = @405; # 从 '403 (Forbidden)'
26          # 到 '405 的转换拒绝请求 (方法不被允许)'
27          proxy_pass http://warehouse_pricing;
28      }
29
30
31 }
```

[在 GitHub 上查看原始数据](#)

使用此配置后, 未使用第 22 行所列方法向定价 service 发出的请求 (以及未使用第 13 行所列方法对库存 service 进行请求) 将被拒绝, 并且不会传递到后端 service。NGINX 发送 405 (Method Not Allowed) 响应, 以通知 API 客户端确切的错误类型, 如以下控制台跟踪所示。在需要遵循“最小披露”的安全策略时, 可使用 `error_page` 指令将此响应转换为信息量较少的错误, 例如 400 (Bad Request)。

```
$ curl https://api.example.com/api/warehouse/pricing/item001
{"sku":"item001","price":179.99}
$ curl -X DELETE https://api.example.com/api/warehouse/pricing/item001
{"status":405,"message":"Method not allowed"}
```

应用细粒度的访问控制

第 1 章介绍了如何通过启用身份验证选项（例如 API 密钥和 JSON Web Tokens (JWT)）保护 API 免受未经授权的访问。我们可以使用经过身份验证的 ID 或经过身份验证的 ID 的属性来执行细粒度的访问控制。

我们在此处提供了两个相关示例：

- 第一个示例为控制对特定 API 资源的访问，扩展了第 1 章中介绍的配置，并使用 API 密钥身份验证方式验证给定 API 客户端是否在允许名单 (allowlist) 中。
- 第二个示例为限定客户端使用哪些 HTTP 方法。它实现了第 1 章中提到的 JWT 身份验证方式，使用自定义声明来识别符合条件的 API 客户端。（请注意，JWT 支持是 NGINX Plus 的独有功能。）

当然，其他身份验证方式也适用于这些示例中的用例，例如 HTTP Basic 认证和 OAuth 2.0 令牌自省。

控制对特定资源的访问

假设我们只允许“基础设施客户端”访问 Warehouse API 库存 service 的 audit 资源。启用 API 密钥身份验证方式后，我们使用 map 块创建基础设施客户端名称的允许名单，以便在使用相应的 API Key 时变量 \$is_infrastructure 的结果为 1。

```
11 map $api_client_name $is_infrastructure {  
12     default          0;  
13  
14     "client_one"    1;  
15     "client_six"    1;  
16 }
```

[在 GitHub 上查看原始数据](#)

在 Warehouse API 的定义中，我们为库存 `audit` 资源添加了一个 `location` 块（第 15-20 行）。`if` 块可确保只有基础设施客户端可以访问该资源。

```
1  # Warehouse API
2  #
3  location /api/warehouse/ {
4      # 这里的策略配置（身份验证、速率限制、日志记录……）
5      #
6      access_log /var/log/nginx/warehouse_api.log main;
7      auth_request /_validate_apikey;
8
9      # URI 路由
10     #
11     location /api/warehouse/inventory {
12         proxy_pass http://warehouse_inventory;
13     }
14
15     location = /api/warehouse/inventory/audit {
16         if ($is_infrastructure = 0) {
17             return 403;      # 禁止（无基础设施）
18         }
19         proxy_pass http://warehouse_inventory;
20     }
21
22     location /api/warehouse/pricing {
23         proxy_pass http://warehouse_pricing;
24     }
25
26     return 404;          # Catch-all
27 }
```

[在 GitHub 上查看原始数据](#)

请注意，第 15 行的 `location` 指令使用 `=`（等号）修饰符与 `audit` 资源进行精确匹配。精确匹配优先于用于其他资源的默认路径前缀定义。以下跟踪显示了在使用此配置的情况下，不在允许名单上的客户端无法访问库存 `audit` 资源。所示 API Key 属于 `client_two`（如第 1 章中所定义）。

```
$ curl -H "apikey: QzVV6y1EmQFbbxOfRCwyJs35"
https://api.example.com/api/warehouse/inventory/audit
{"status":403,"message":"Forbidden"}
```

为启用 JWT 身份验证后，每个客户端的权限都被编码为自定义声明

控制对特定方法的访问

如上所述，定价 service 接受 GET 和 PATCH 方法，分别支持客户端获取和修改特定物品的价格（我们还可以选择允许 POST 和 DELETE 方法，以提供定价数据的全生命周期管理）。在本部分，我们对该用例进行扩展，控制特定用户可以发出哪些方法。为 Warehouse API 启用 JWT 身份验证后，每个客户端的权限都被编码为自定义声明。发给授权更改定价数据的管理员的 JWT 包含声明 "admin":true。现在，我们扩展了访问控制逻辑，以便只有管理员才能进行更改。

```
60 # 对写入操作的访问由 JWT 声明 'admin' 进行赋值
61 map $request_method $admin_permitted_method {
62     "GET"      true;
63     "HEAD"     true;
64     "OPTIONS"  true;
65     default    $jwt_claim_admin;
66 }
```

[在 GitHub 上查看原始数据](#)

此 map 块（被添加到“速率限制”中所述配置文件的底部）将请求方法（\$request_method）作为输入并生成一个新变量 \$admin_permitted_method。只读方法始终允许（第 62-64 行），但对写入操作的访问取决于 JWT 中 admin 声明的值（第 65 行）。我们现在扩展了 Warehouse API 配置，以确保只有管理员才能更改定价。

```
1 # Warehouse API
2 #
3 location /api/warehouse/ {
4     # 这里的策略配置 (身份验证、速率限制、日志记录.....)
5     #
6     access_log /var/log/nginx/warehouse_api.log main;
7     auth_jwt "Warehouse API";
8     auth_jwt_key_file /etc/nginx/idp_jwks.json;
9
10    # URI 路由
11    #
12    location /api/warehouse/inventory {
13        limit_except GET {
14            deny all;
15        }
16        error_page 403 = @403; # 从 '403 (Forbidden) '
17                                # 到 '405 的转换拒绝响应 (方法不被允许) '
18        proxy_pass http://warehouse_inventory;
19    }
20
21    location /api/warehouse/pricing {
22        limit_except GET PATCH {
23            deny all;
24        }
25        if ($admin_permitted_method != "true") {
26            return 403;
27        }
28        error_page 403 = @405; # 从 '403 (Forbidden) '
29                                # 到 '405 的转换拒绝响应 (方法不被允许) '
30        proxy_pass http://warehouse_pricing;
31    }
32
33    return 404; # Catch-all
34 }
```

[在 GitHub 上查看原始数据](#)

Warehouse API 要求所有客户端都提供有效的 JWT（第 7 行）。我们还通过评估 \$admin_permitted_method 变量（第 25 行）来检查是否允许写入操作。再次提醒，JWT 身份验证是 NGINX Plus 的独有功能。

请求正文可能会构成后端 API service 的攻击向量

控制请求大小

HTTP API 通常使用请求正文来包含后端 API service 要处理的指令和数据。XML/SOAP API 以及 JSON/REST API 也是如此。因此，请求正文可能会构成后端 API service 的攻击向量，当后端 API service 处理超大的请求正文时，可能容易受到缓冲区溢出攻击。

默认情况下，NGINX 拒绝正文大于 1MB 的请求。对于专门处理大型负载（例如图像处理）的 API，此值可以增加，但对于大多数 API，我们会设置一个较低的值。

```
1 # Warehouse API
2 #
3 location /api/warehouse/ {
4     # 这里的策略配置（身份验证、速率限制、日志记录……）
5     #
6     access_log /var/log/nginx/warehouse_api.log main;
7     client_max_body_size 16k;
```

[在 GitHub 上查看原始数据](#)

第 7 行的 `client_max_body_size` 指令限制了请求正文的大小。有了此配置，我们就可以比较 API 网关在接收到两个不同的 PATCH 定价 service 请求时的行为。第一个 curl 命令发送一小段 JSON 数据，而第二个命令则尝试发送一个大文件 (`/etc/services`) 的内容。

```
$ curl -iX PATCH -d '{"price":199.99}'
https://api.example.com/api/warehouse/pricing/item001
HTTP/1.1 204 No Content
Server: nginx/1.21.3
Connection: keep-alive

$ curl -iX PATCH -d@/etc/services
https://api.example.com/api/warehouse/pricing/item001
HTTP/1.1 413 Request Entity Too Large
Server: nginx/1.21.3
Content-Type: application/json
Content-Length: 45
Connection: close

{"status":413,"message":"Payload too large"}
```

验证请求正文

后端 API service 容易受到包含无效或意外数据的正文的影响

除了容易受到大型请求正文的缓冲区溢出攻击之外，后端 API service 还容易受到包含无效或意外数据的正文的影响。对于需要请求正文具有正确格式的 JSON 的应用，我们可以在将 JSON 数据代理到后端 API service 之前，使用 **NGINX JavaScript 模块** 验证其解析是否正确。

安装 **JavaScript 模块** 后，我们使用 `js_import` 指令来引用包含 JSON 数据验证函数的 JavaScript 代码的文件。

```
51 js_import json_validation.js;
52 js_set $json_validated json_validation.parseRequestBody;
```

[在 GitHub 上查看原始数据](#)

`js_set` 指令定义了一个新变量 `$json_validated`，通过调用 `parseRequestBody` 函数对其进行赋值。

```
1 export default { parseRequestBody };
2
3 function parseRequestBody(r) {
4     try {
5         if (r.variables.request_body) {
6             JSON.parse(r.variables.request_body);
7         }
8         return r.variables.upstream;
9     } catch (e) {
10         r.error('JSON.parse exception');
11         return '127.0.0.1:10415'; // Address for error response
12     }
13 }
```

[在 GitHub 上查看原始数据](#)

`parseRequestBody` 函数尝试使用 `JSON.parse` 方法（第 6 行）解析请求正文。如果解析成功，则返回此请求所需上游 group 的名称（第 8 行）。如果无法解析请求正文（导致异常），则返回本地服务器地址（第 11 行）。`return` 指令将填充 `$json_validated` 变量，以便我们可以使用它来确定将请求发送到何处。

```
8      # URI 路由
9      #
10     location /api/warehouse/inventory {
11         proxy_pass http://warehouse_inventory;
12     }
13
14     location /api/warehouse/pricing {
15         set $upstream warehouse_pricing;
16         mirror /_get_request_body;          # 强制提前读取
17         client_body_in_single_buffer on;   # 尽量在请求正文上
18                                         # 减少内存复制操作
19         client_body_buffer_size        16k; # 将最大的正文保存到内存中
20                                         # (在写入文件之前)
21         client_max_body_size          16k;
22         proxy_pass http://$json_validated$request_uri;
23     }
```

[在 GitHub 上查看原始数据](#)

在 Warehouse API 的 URI 路由部分，我们在第 22 行修改了 `proxy_pass` 指令。它将请求传递给后端 API service，如前面部分中讨论的 Warehouse API 配置一样（例如在“[控制对特定方法的访问](#)”中 `warehouse_api_methods.conf` 中的第 18 和 30 行），但是现在使用 `$json_validated` 变量作为目标地址。如果客户端正文被成功解析为 JSON，那么我们将代理到第 15 行定义的上游 group。但是，如果出现异常，我们将使用返回值 127.0.0.1:10415 向客户端发送错误响应。

```
54 server {
55     listen 127.0.0.1:10415;
56     return 415; # 不支持的介质类型
57     include api_json_errors.conf;
58 }
```

[在 GitHub 上查看原始数据](#)

当请求被代理到这个虚拟服务器时，NGINX 将向客户端发送 415 (Unsupported Media Type) 响应。

有了这个完整的配置，NGINX 将只在请求具有正确格式的 JSON 正文时才将其代理到后端 API service。

```
$ curl -iX POST -d '{"sku":"item002","price":85.00}'  
https://api.example.com/api/warehouse/pricing  
HTTP/1.1 201 Created  
Server: nginx/1.21.3  
Location: /api/warehouse/pricing/item002  
  
$ curl -X POST -d 'item002=85.00' https://api.example.com/api/  
warehouse/pricing  
{"status":415,"message":"Unsupported media type"}
```

关于 \$request_body 变量的说明

JavaScript 函数 `parseRequestBody` 使用 `$request_body` 变量来执行 JSON 解析。但是，NGINX 默认不赋值此变量，只是将请求正文流式传输到后端而不创建中间副本。我们通过在 URI 路由部分（第 16 行）使用 `mirror` 指令，创建客户端请求的副本，并赋值 `$request_body` 变量。

```
14     location /api/warehouse/pricing {  
15         set $upstream warehouse_pricing;  
16         mirror /_get_request_body;          # 强制提前读取  
17         client_body_in_single_buffer on;    # 尽量在请求正文上  
18                                         # 减少内存复制操作  
19         client_body_buffer_size      16k;  # 将最大的正文保存到内存中  
20                                         # (在写入文件之前)  
21         client_max_body_size        16k;  
22         proxy_pass http://$json_validated$request_uri;  
23     }
```

[在 GitHub 上查看原始数据](#)

第 17 行和第 19 行的指令控制 NGINX 如何在内部处理请求正文。我们将 `client_body_buffer_size` 设置为与 `client_max_body_size` 相同的大小，这样请求正文就不会写入磁盘。这样做有助于最大限度地减少磁盘 I/O 操作，从而提高整体性能，但代价是内存利用率会有所增加。对于大多数请求正文较小的 API 网关用例，这是一个不错的折衷方案。

如前所述，`mirror` 指令会创建客户端请求的副本。除了赋值 `$request_body` 之外，我们不需要此副本，因此我们将其发送到我们在顶层 API 网关配置的 `server` 块中定义的“死胡同（dead end）”位置（`@get_request_body`）。

```
45      # 用来填充 $request_body 以进行 JSON 身份验证的虚拟位置
46      location /_get_request_body {
47          return 204;
48      }
```

[在 GitHub 上查看原始数据](#)

此位置只发送 204 (No Content) 响应。此响应与镜像请求相关，因此被忽略，对原始客户端请求的处理所增加的开销也可以忽略不计。

总结

在本章中，我们重点讨论了如何保护生产环境中的后端 API service 免受恶意和行为异常的客户端的影响。NGINX 所使用的 API 流量管理技术同样被用于支持和保护[当今互联网上最繁忙的站点](#)。

您可前往我们的 [GitHub Gist repo](#) 查看本章中用到的完整文件集。

3. 发布 gRPC 服务

本章解释了如何将 NGINX 部署为 gRPC 服务的 API 网关。

注：除非另有说明，否则本章中的所有信息都适用于 NGINX 开源版和 NGINX Plus。

近年来，介绍微服务应用架构的概念和优势的文章非常多，其中以 [NGINX 博文](#) 居首。微服务应用的核心是 HTTP API，第 1 章和第 2 章使用了一个假设的 REST API 来说明 NGINX 如何处理此类应用。

尽管基于 JSON 消息格式的 REST API 在现代应用中非常流行，但它并不是所有场景或所有企业的理想之选。最常见的挑战是：

- **文档标准** —— 如果没有良好的开发者制度或强制性的文档要求，最后很容易产生大量缺乏准确定义的 REST API。[Open API 规范](#)已成为 REST API 的通用接口描述语言，但其使用却不是强制性的，需要开发组织内部的有力治理。
- **事件和长连接** —— REST API 以及它们使用 HTTP 传输几乎决定了所有 API 调用都是请求 - 响应模式。当客户端应用需要服务器反馈消息时，使用 [HTTP 长轮询](#) 和 [WebSocket](#) 等解决方案会有所帮助，但使用此类解决方案最终都需要构建一个单独、相邻的 API。
- **复杂事务** —— REST API 是围绕唯一资源的概念构建的，每个资源都由一个 URI 表示。当应用需要调用多个资源更新时，要么需要多个 API 调用（效率低下），要么必须在后端实现复杂的事务（与 REST 的核心原则相悖）。

[近年来，gRPC 已发展成为构建分布式应用的替代方法](#)

近年来，gRPC 已发展成为构建分布式应用，尤其是微服务应用的替代方法。gRPC 最初由 Google 开发，并于 2015 年开源，现已成为云原生计算基金会的一个[项目](#)。值得注意的是，gRPC 使用 HTTP/2 作为传输机制，并利用其二进制数据格式和多路复用流功能。

gRPC 的主要优势包括：

- 紧密耦合的接口定义语言 ([协议缓冲区](#))
- 对流数据的原生支持 (双向)
- 高效的二进制数据格式
- 自动生成多语言的代码，支持真正的多语言开发环境，且不会产生互操作性问题

定义 gRPC 网关

第 1 章和第 2 章描述了如何通过单个入口点（例如 <https://api.example.com>）交付多个 API。当 NGINX 部署为 gRPC 网关时，gRPC 流量的默认行为和特征促使 NGINX 也要采用这种方法。虽然 NGINX 可以在同一主机名和端口上共享 HTTP 和 gRPC 流量，但最好还是将它们分开，主要有以下原因有：

- REST 和 gRPC 应用的 API 客户端需要不同格式的错误响应
- REST 和 gRPC 访问日志的相关字段有所不同
- 因为 gRPC 不涉及旧版 Web 浏览器，因此它可以实施更严格的 TLS 策略

为了实现这种分离，我们需要修改 gRPC 网关主配置文件 `grpc_gateway.conf` 的 `server{}` 块，它位于 `/etc/nginx/conf.d` 目录。

```
1 log_format grpc_json escape=json '{"timestamp": "$time_iso8601",'
2           '"client": "$remote_addr", "uri": "$uri", "http-status": $status,'
3           '"grpc-status": $grpc_status, "upstream": "$upstream_addr"'
4           '"rx-bytes": $request_length, "tx-bytes": $bytes_sent}';
5
6 map $upstream_trailer_grpc_status $grpc_status {
7     default $upstream_trailer_grpc_status; # grpc-status 通常是一个消息头
8     '' $sent_http_grpc_status; # 否则就使用请求头，无论其来源如何
9 }
10
11 server {
12     listen 50051 http2; # 在生产环境中，注释掉以禁用明文端口
13     listen 443 http2 ssl;
14     server_name grpc.example.com;
15     access_log /var/log/nginx/grpc_log.json grpc_json;
16
17     # TLS config
18     ssl_certificate      /etc/ssl/certs/grpc.example.com.crt;
19     ssl_certificate_key  /etc/ssl/private/grpc.example.com.key;
20     ssl_session_cache    shared:SSL:10m;
21     ssl_session_timeout  5m;
22     ssl_ciphers          HIGH:!aNULL:!MD5;
23     ssl_protocols        TLSv1.2 TLSv1.3;
24 }
25
26
27
28
29
30
31
32
33
34
35
36 }
```

[在 GitHub 上查看原始数据](#)

我们首先定义 gRPC 流量访问日志中的条目格式（第 1-4 行）。在本例中，我们使用 JSON 格式从每个请求中捕获最相关的数据。请注意，HTTP method 不包括在内，因为所有 gRPC 请求都使用 POST。我们还记录了 gRPC 状态代码和 HTTP 状态代码。然而，gRPC 状态代码可通过不同的方式生成。在正常情况下，`grpc-status` 从后端返回 HTTP/2 消息头，但在一些错误情况下，它可能会被后端或 NGINX 自己返回 HTTP/2 消息头。为了简化访问日志，我们使用 `map` 块（第 6-9 行）来赋值新变量 `$grpc_status` 并从产生该变量的地方获取 gRPC 状态。

此配置包含两个 `listen` 指令（第 12 行和第 13 行），所以我们可以测试明文（端口 50051）和受 TLS 保护的（端口 443）流量。`http2` 参数将 NGINX 配置为接受 HTTP/2 连接——请注意，这与 `ssl` 参数无关。另请注意，端口 50051 是 gRPC 的常规明文端口，但不推荐在生产环境中使用。

TLS 配置是常规配置，但 `ssl_protocols` 指令（第 23 行）除外，该指令将 TLS 1.2 指定为最弱的可接受协议。HTTP/2 规范要求使用 TLS 1.2（或更高版本），以保证所有客户端都支持对 TLS 的 SNI（`Server Name Indication`）扩展。这意味着 gRPC 网关可以与其他 `server{}` 块中定义的虚拟服务器共享端口 443。

运行示例 gRPC 服务

为了了解 NGINX 的 gRPC 功能，我们使用了一个简单的测试环境，该环境代表了 gRPC 网关的关键组件，并部署了多个 gRPC 服务。我们使用[官方 gRPC 指南](#)中的两个示例应用：`helloworld`（用 Go 编写）和 `RouteGuide`（用 Python 编写）。`RouteGuide` 应用特别有用，因为它包含了四种 gRPC 服务方法：

- 简单 RPC（单一请求 - 响应）
- 响应流 RPC
- 请求流 RPC
- 双向流 RPC

所有 gRPC 服务都作为 Docker 容器安装在我们的 NGINX 主机上。有关构建该测试环境的完整说明，请参阅[附录](#)。

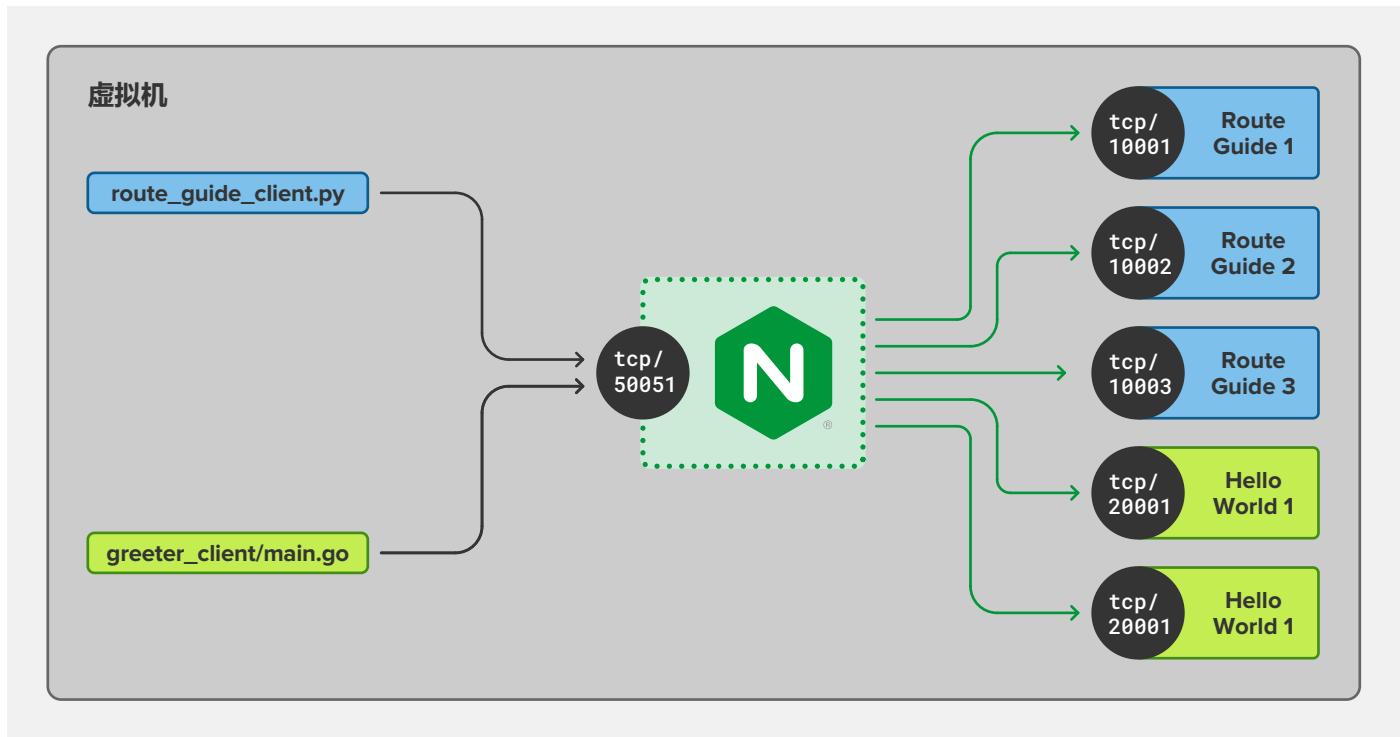


图 3: NGINX 作为 gRPC 网关的测试环境

我们配置 NGINX 以了解 RouteGuide 和 helloworld service，以及可用容器的地址。

```

40 upstream routeguide_service {
41     zone routeguide_service 64k;
42     server 127.0.0.1:10001;
43     server 127.0.0.1:10002;
44     server 127.0.0.1:10003;
45 }
46
47 upstream helloworld_service {
48     zone helloworld_service 64k;
49     server 127.0.0.1:20001;
50     server 127.0.0.1:20002;
51 }

```

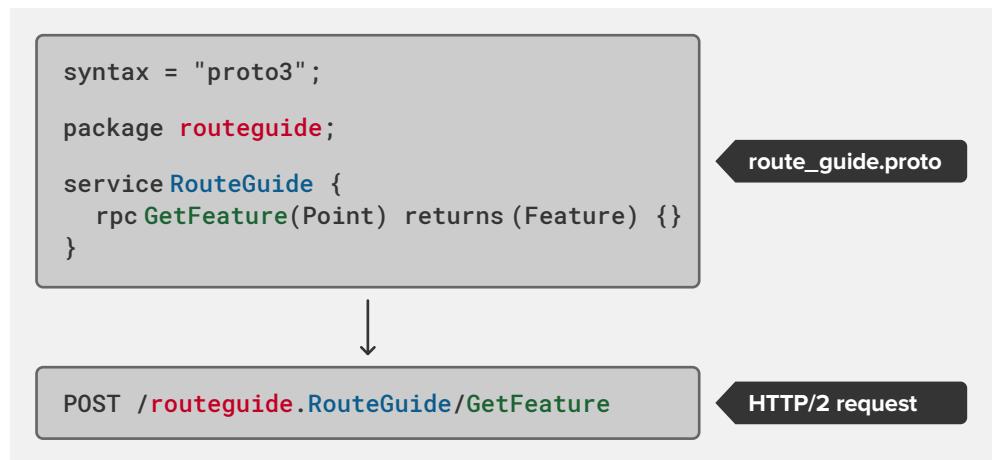
[在 GitHub 上查看原始数据](#)

我们为每个 gRPC 服务添加一个 `upstream` 块（第 40-45 和 47-51 行），并使用运行 gRPC 服务器代码的各个容器的地址填充它们。

路由 gRPC 请求

通过 NGINX 监听 gRPC 的常规明文端口（50051），我们将路由信息添加到配置中，以便客户端请求能够到达正确的后端 service。但首先我们需要了解 gRPC method 调用如何表示为 HTTP/2 请求。下图为 RouteGuide service 的 `route_guide.proto` 文件的缩略版，说明了 package、service 和 RPC method 如何形成 URI，如 NGINX 所见。

图 4：协议缓冲区 RPC method 如何转换为 HTTP/2 请求



因此，HTTP/2 请求中携带的信息只需匹配包名（此处为 `routeguide` 或 `helloworld`）即可用于路由。

```
25     # 路由
26     location /routeguide. {
27         grpc_pass grpc://routeguide_service;
28     }
29     location /helloworld. {
30         grpc_pass grpc://helloworld_service;
31     }
```

[在 GitHub 上查看原始数据](#)

第一个 `location` 块（第 26 行），不包含任何修饰符，定义了一个前缀匹配，以便 `/routeguide.` 匹配该包对应的 `.proto` 文件中定义的所有 service 和 RPC method。因此，`grpc_pass` 指令（第 27 行）将来自 RouteGuide 客户端的所有请求传递给上游 group `routeguide_service`。该配置（以及第 29 行和第 30 行的 helloworld 服务的并行配置）提供了 gRPC 包与其后端 service 之间的简单映射。

请注意，`grpc_pass` 指令的参数以 `grpc://` 方式请求，该请求方式使用明文 gRPC 连接代理请求。如果后端配置了 TLS，我们可以使用 `grpcs://` 通过端到端加密来保护 gRPC 连接。

运行 RouteGuide 客户端后，我们可以通过查看日志文件来确认路由行为。此处，我们看到 RouteChat RPC method 被路由到在端口 10002 上运行的容器。

```
$ python route_guide_client.py
...
$ tail -1 /var/log/nginx/grpc_log.json | jq
{
    "timestamp": "2021-10-20T12:17:56+01:00",
    "client": "127.0.0.1",
    "uri": "/routeguide.RouteGuide/RouteChat",
    "http-status": 200,
    "grpc-status": 0,
    "upstream": "127.0.0.1:10002",
    "rx-bytes": 161,
    "tx-bytes": 212
}
```

精确路由

如上所示，将多个 gRPC 服务简单、高效的路由到不同后端，只需要少数几行配置。然而，生产环境中的路由要求可能更加复杂，需要基于 URI 中的其他元素（gRPC 服务甚至单个 RPC method）进行路由。

以下配置片段扩展了前面的示例，以便将双向流 RPC method RouteChat 路由到同一个后端，而将其他所有 RouteGuide 方法路由到不同的后端。

```
1 # 服务级别目标
2 location /routeguide.RouteGuide/ {
3     grpc_pass grpc://routeguide_service_default;
4 }
5
6 # 方法级别路由
7 location = /routeguide.RouteGuide/RouteChat {
8     grpc_pass grpc://routeguide_service_streaming;
9 }
```

[在 GitHub 上查看原始数据](#)

第二个 location 指令（第 7 行）使用 =（等号）来表示这是 RouteChat RPC method 的 URI 上的精确匹配。精确匹配在前缀匹配之前进行处理，这意味着 RouteChat URI 不会考虑其他 location 块。

响应错误

gRPC 错误与传统 HTTP 流量的错误有些不同。客户端期望错误条件表示为 gRPC 响应，这使得当 NGINX 配置为 gRPC 网关时，默认的 NGINX 错误页面集（HTML 格式）将不适合使用。我们的解决方法是为 gRPC 客户端指定一组自定义的错误响应。

```
33      # 错误响应
34      include conf.d/errors.grpc_conf;          # 符合 gRPC 的错误响应
35      default_type application/grpc;           # 确保所有错误响应符合 gRPC
```

[在 GitHub 上查看原始数据](#)

完整的 gRPC 错误响应集是一个相对较长且大部分是静态响应的配置，因此我们将它们保存在一个单独的文件 `errors.grpc_conf` 中，并使用 `include` 指令（第 34 行）引用它们。与 HTTP/REST 客户端不同，gRPC 客户端应用不需要处理大量的 HTTP 状态代码。[gRPC 文档](#)指定了 NGINX 等中间代理必须如何将 HTTP 错误代码转换为 gRPC 状态代码，以便客户端始终能够接收到合适的响应。我们使用 `error_page` 指令来执行这个映射。

```
1  # 标准的 HTTP-to-gRPC 状态代码映射
2  # 参考: https://github.com/grpc/grpc/blob/master/doc/http-grpc-status-mapping.md
3  #
4  error_page 400 = @grpc_internal;
5  error_page 401 = @grpc_unauthenticated;
6  error_page 403 = @grpc_permission_denied;
7  error_page 404 = @grpc_unimplemented;
8  error_page 429 = @grpc_unavailable;
9  error_page 502 = @grpc_unavailable;
10 error_page 503 = @grpc_unavailable;
11 error_page 504 = @grpc_unavailable;#
```

[在 GitHub 上查看原始数据](#)

每个标准 HTTP 状态代码都使用 @ 前缀传递到指定 location，这样就可以生成符合 gRPC 要求的响应。例如，HTTP 404 响应在内部被重定向到 @grpc_unimplemented location，该 location 文件定义如下：

```
49 location @grpc_unimplemented {  
50     add_header grpc-status 12;  
51     add_header grpc-message unimplemented;  
52     return 204;  
53 }
```

[在 GitHub 上查看原始数据](#)

@grpc_unimplemented 命名 location 仅可用于内部 NGINX 处理——由于没有可路由的 URI，客户端无法直接请求该 location。在 location 中，我们填充强制性 gRPC 标头并使用 HTTP 状态代码 204 (No Content) 发送它们（不包含响应正文），从而构造 gRPC 响应。

我们可以使用 `curl(1)` 命令模拟一个行为不端的 gRPC 客户端请求不存在的 gRPC method。但是请注意，由于协议缓冲区使用二进制数据格式，curl 通常不适合作为 gRPC 测试客户端。要在命令行上测试 gRPC，可考虑使用 `grpc_cli`。

```
$ curl -i --http2 -H "Content-Type: application/grpc" -H "TE: trailers" -X POST  
https://grpc.example.com/does.Not/Exist  
HTTP/2 204  
server: nginx/1.21.3  
date: Wed, 20 Oct 2021 15:03:41 GMT  
grpc-status: 12  
grpc-message: unimplemented
```

上面引用的 `grpc_errors.conf` 文件还包含 NGINX 可能生成的其他错误响应的 HTTP 到 gRPC 状态代码映射，例如超时和客户端证书错误。

使用 gRPC 元数据验证客户端

gRPC 元数据允许客户端在 RPC method 调用的同时发送附加信息，而无需将这些数据作为协议缓冲区规范文件 (.proto 文件) 的一部分。元数据是一个简单的键值对 (key-value) 列表，每个键值对都作为单独的 HTTP/2 标头传输。因此，NGINX 访问元数据非常容易。

在元数据的众多用例中，客户端身份验证对 gRPC API 网关来说是最常见的。以下配置片段显示了 NGINX Plus 如何使用 gRPC 元数据执行 [JWT 身份验证](#) (JWT 身份验证是 NGINX Plus 的独有功能)。在此示例中，JWT 在 auth-token 元数据中发送。

```
1 location /routeguide. {
2     auth_jwt realm=routeguide token=$http_auth_token;
3     auth_jwt_key_file my_idp.jwk;
4     grpc_pass grps://routeguide_service;
5 }
```

[在 GitHub 上查看原始数据](#)

对 NGINX Plus 来说，每个 HTTP 请求标头都可作为一个名为 \$http_header 的变量来使用。标头名称中的连字符 (-) 转换为变量名称中的下划线 (_)，因此 JWT 可用作 \$http_auth_token (第 2 行)。

如果 API 密钥用于身份验证（可能是现有的 HTTP/REST API），那么这些密钥也可以在 gRPC 元数据中携带，并由 NGINX 验证。第 1 章介绍了 [API 密钥身份验证](#) 的配置。

实施健康检查

当对多个后台服务器进行负载均衡时，一定要避免将请求发送到已关闭或不可用的后台服务器。借助 NGINX Plus，我们可以使用[主动健康检查](#)主动向后台服务器发送带外请求，并在它们未按预期响应健康检查时将其从负载均衡轮换中移除。通过这种方式，我们可以确保客户端请求永远不会被传输到停止服务的后台服务器。

以下配置片段为 RouteGuide 和 helloworld gRPC service 启用了主动健康检查；为了突出显示相关配置，该片段省略了一些指令，这些指令包含在前面几节中使用的 `grpc_gateway.conf` 文件中。

```
11 server {
12     listen 50051 http2;      # Plaintext
13
14     # 路由
15     location /routeguide. {
16         grpc_pass grpc://routeguide_service;
17         health_check type=grpc grpc_status=12;          # 12=unimplemented
18     }
19     location /helloworld. {
20         grpc_pass grpc://helloworld_service;
21         health_check type=grpc grpc_status=12;          # 12=unimplemented
22     }
23 }
```

[在 GitHub 上查看原始数据](#)

对于每个路由，我们现在还指定 `health_check` 指令（第 17 和 21 行）。正如 `type=grpc` 参数所指定的，NGINX Plus 使用 [gRPC 健康检查协议](#)向上游 group 中的每个服务器发送健康检查。但是，我们简单的 gRPC 服务没有实现 gRPC 健康检查协议，因此我们希望它们使用表示“unimplemented” (`grpc_status=12`) 的状态代码进行响应。当它们使用这种状态代码进行响应时，就足以表明我们正在与一个活动的 gRPC 服务进行通信。

有了这个配置，我们可以关闭任何后端容器，且 gRPC 客户端不会出现延迟或超时。主动健康检查是 NGINX Plus 的独有功能；有关 [gRPC 健康检查](#)的更多信息，请阅读我们的博客。

应用速率限制和其他 API 网关控制

`grpc_gateway.conf` 中的示例配置适合生产环境使用，其中对 TLS 进行了一些小的修改。基于 package、service 或 RPC method 路由 gRPC 请求的能力表明现有的 NGINX 功能可以以 HTTP/REST API 或常规 Web 流量完全相同的方式应用于 gRPC 流量。在每种情况下，相关的 `location` 块都可以通过进一步的配置（例如速率限制或带宽控制）进行扩展。

总结

本章重点介绍了将 gRPC 作为构建微服务应用的云原生技术。我们展示了 NGINX 如何能够像交付 HTTP/REST API 一样有效地交付 gRPC 应用，以及如何通过 NGINX 作为多用途 API 网关发布这两种 API。

有关本章使用的测试环境的说明位于下面的[附录](#)中，您可以从我们的 [GitHub Gist repo](#) 中下载所有文件。

附录

设置 gRPC 测试环境

以下说明将第 3 章中使用的 gRPC 测试环境安装在一个虚拟机上，以便隔离和重复使用。当然也如果有条件也可以安装在物理服务器上。

为了简化测试环境，我们使用 Docker 容器来运行 gRPC 服务。这么做的好处是我们不需要在测试环境中使用多个主机，但仍然可以像在生产环境中一样，让 NGINX 通过网络调用建立代理连接。

Docker 还支持我们在不同的端口上运行每个 gRPC 服务的多个实例，而无需修改代码。每个 gRPC 服务监听容器内的端口 50051，该端口映射到虚拟机上唯一的 localhost 端口。这反过来释放了端口 50051，NGINX 可以将其用作监听端口。因此，当测试客户端使用其预配置的端口 50051 连接时，它们会连接到 NGINX。

安装 NGINX 开源版或 NGINX Plus

1. 根据 NGINX Plus 管理员指南中的说明安装 [NGINX 开源版或 NGINX Plus](#)。

2. 将以下文件从 [GitHub Gist repo](#) 复制到 `/etc/nginx/conf.d`:

- `grpc_gateway.conf`
- `errors.grpc_conf`

注意：如果未使用 TLS，则注释掉 `grpc_gateway.conf` 中的 `ssl_*` 指令。

3. 启动 NGINX 开源版或 NGINX Plus。

```
$ sudo nginx
```

安装 Docker

对于 Debian 和 Ubuntu，运行:

```
$ sudo apt-get install docker.io
```

对于 CentOS、RHEL 和 Oracle Linux，运行:

```
$ sudo yum install docker
```

安装 RouteGuide 服务容器

- 通过以下 Dockerfile 为 RouteGuide 容器构建 Docker 镜像。

```
1 # 该 Dockerfile 从以下网址运行 RouteGuide 服务器
2 # https://grpc.io/docs/tutorials/basic/python.html
3
4 FROM python
5 RUN pip install grpcio-tools
6 RUN git clone -b v1.14.x https://github.com/grpc/grpc
7 WORKDIR grpc/examples/python/route_guide
8
9 EXPOSE 50051
10 CMD ["python", "route_guide_server.py"]
```

[在 GitHub 上查看原始数据](#)

您可以在构建之前将 Dockerfile 复制到本地子目录，也可以将 Dockerfile 的 Gist 的 URL 指定为 docker build 命令的参数：

```
$ sudo docker build -t routeguide
https://gist.githubusercontent.com/username/gists/
87ed942d4ee9f7e7ebb2ccf757ed90be/raw/ce090f92f3bbcb5a94bbf8de
d4d597cd47b43cbe/routeguide.Dockerfile
```

下载和构建镜像可能需要几分钟时间。出现消息 Successfully built 和一个十六进制字符串 (image ID) 即表示构建完成。

- 确认镜像是通过运行 docker images 构建的。

```
$ sudo docker images
REPOSITORY      TAG          IMAGE ID      CREATED       SIZE
routeguide     latest        63058a1cf8ca  1 minute ago  1.31 GB
python          latest        825141134528  9 days ago   923 MB
```

- 启动 RouteGuide 容器

```
$ sudo docker run --name rg1 -p 10001:50051 -d routeguide
$ sudo docker run --name rg2 -p 10002:50051 -d routeguide
$ sudo docker run --name rg3 -p 10003:50051 -d routeguide
```

每个命令执行成功时，都会出现一个长的十六进制字符串，代表正在运行的容器。

4. 运行 `docker ps`, 检查三个容器是否都已启动。 (为了便于阅读, 我们将示例输出拆分成了多行。)

```
$ sudo docker ps
CONTAINER ID   IMAGE      COMMAND           STATUS    ...
d0cdaaedd0f   routeguide "python route_g..." Up 2 seconds ...
c04996ca3469  routeguide "python route_g..." Up 9 seconds ...
2170ddb62898  routeguide "python route_g..." Up 1 minute ...

...  PORTS          NAMES
...  0.0.0.0:10003->50051/tcp  rg3
...  0.0.0.0:10002->50051/tcp  rg2
...  0.0.0.0:10001->50051/tcp  rg1
```

输出中的 `PORTS` 列显示了每个容器如何将不同的本地端口映射到容器内的端口 50051。

安装 helloworld Service 容器

1. 通过以下 Dockerfile 为 helloworld 容器构建 Docker 镜像。

```
1  # 该 Dockerfile 从以下网址运行 helloworld 服务器
2  # https://grpc.io/docs/quickstart/go.html
3
4  FROM golang
5  RUN go get -u google.golang.org/grpc
6  WORKDIR $GOPATH/src/google.golang.org/grpc/examples/helloworld
7
8  EXPOSE 50051
9  CMD ["go", "run", "greeter_server/main.go"]
```

[在 GitHub 上查看原始数据](#)

您可以在构建之前将 Dockerfile 复制到本地子目录, 也可以将 Dockerfile 的 Gist 的 URL 指定为 `docker build` 命令的参数:

```
$ sudo docker build -t helloworld
https://gist.githubusercontent.com/username/87ed942d4ee9f7e7ebb2ccf757ed90be/raw/ce090f92f3bbcb5a94bbf8de
d4d597cd47b43cbe/helloworld.Dockerfile
```

下载和构建镜像可能需要几分钟时间。出现消息 `Successfully built` 和一个十六进制字符串 (image ID) 即表示构建完成。

2. 确认镜像是通过运行 docker images 构建的。

```
$ sudo docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
helloworld      latest   e5832dc0884a  10 seconds ago  926MB
routeguide     latest   170761fa3f03  4 minutes ago  1.31GB
python          latest   825141134528  9 days ago    923MB
golang          latest   d0e7a411e3da  3 weeks ago   794MB
```

3. 启动 helloworld 容器。

```
$ sudo docker run --name hw1 -p 20001:50051 -d helloworld
$ sudo docker run --name hw2 -p 20002:50051 -d helloworld
```

每个命令执行成功时，都会出现一个长的十六进制字符串，代表正在运行的容器。

4. 运行 docker ps，检查两个 helloworld 容器是否都已启动。

```
$ sudo docker ps
CONTAINER ID  IMAGE      COMMAND      STATUS      ...
e0d204ae860a  helloworld "go run greeter..." Up 5 seconds ...
66f21d89be78  helloworld "go run greeter..." Up 9 seconds ...
d0cdaaeddff0f  routeguide  "python route_g..." Up 4 minutes ...
c04996ca3469  routeguide  "python route_g..." Up 4 minutes ...
2170ddb62898  routeguide  "python route_g..." Up 5 minutes ...

...  PORTS      NAMES
...  0.0.0.0:20002->50051/tcp  hw2
...  0.0.0.0:20001->50051/tcp  hw1
...  0.0.0.0:10003->50051/tcp  rg3
...  0.0.0.0:10002->50051/tcp  rg2
...  0.0.0.0:10001->50051/tcp  rg1
```

安装 gRPC 客户端应用

1. 安装编程语言的先决条件，其中一些可能已安装在测试环境中。

- 对于 Ubuntu 和 Debian，运行：

```
$ sudo apt-get install golang-go python3 python-pip git
```

- 对于 CentOS、RHEL 和 Oracle Linux，运行：

```
$ sudo yum install golang python python-pip git
```

请注意，`python-pip` 需要启用 EPEL 存储库（根据需要先运行 `sudo yum install epel-release`）。

2. 下载 helloworld 应用：

```
$ go get google.golang.org/grpc
```

3. 下载 RouteGuide 应用：

```
$ git clone -b v1.14.1 https://github.com/grpc/grpc
$ pip install grpcio-tools
```

测试设置

1. 运行 helloworld 客户端：

```
$ go run  
go/src/google.golang.org/grpc/examples/helloworld/  
greeter_client/main.go
```

2. 运行 RouteGuide 客户端：

```
$ cd grpc/examples/python/route_guide  
$ python route_guide_client.py
```

3. 检查 NGINX 日志，确认测试环境可正常运行：

```
$ tail /var/log/nginx/grpc_log.json
```

F5 NGINX 市场销售热线: 400 991 8366
F5 NGINX 售后支持电话: 400 815 5595, 010-5643 8123
与我们在线沟通: contactme_nginxapac@f5.com



**NGINX 官方社区
微信公众号**
(产品信息、解决方案、活动资源)

**NGINX 开源社区
微信公众号**
(技术资料、活动信息、社区动态)

**立即加入
NGINX 社区微信群**
(答疑互动、行业交流、活动提醒)

F5 NGINX 北京办公室
地址: 北京市朝阳区建国路 81 号华贸中心 1 号
写字楼 21 层 05-09 室
邮编: 100025
电话: (+86) 10 5643 8000
传真: (+86) 10 5643 8100
<https://www.nginx-cn.net/>

F5 NGINX 上海办公室
地址: 上海市黄浦区湖滨路 222 号企业天地
1 号楼 1119 室
邮编: 200021
电话: (+86) 21 6113 2588
传真: (+86) 21 6113 2599
<https://www.nginx-cn.net/>

F5 NGINX 广州办公室
地址: 广州市天河区珠江新城华夏路 10 号
富力中心写字楼 1108 室
邮编: 510623
电话: (+86) 20 3892 7557
传真: (+86) 20 3892 7547
<https://www.nginx-cn.net/>