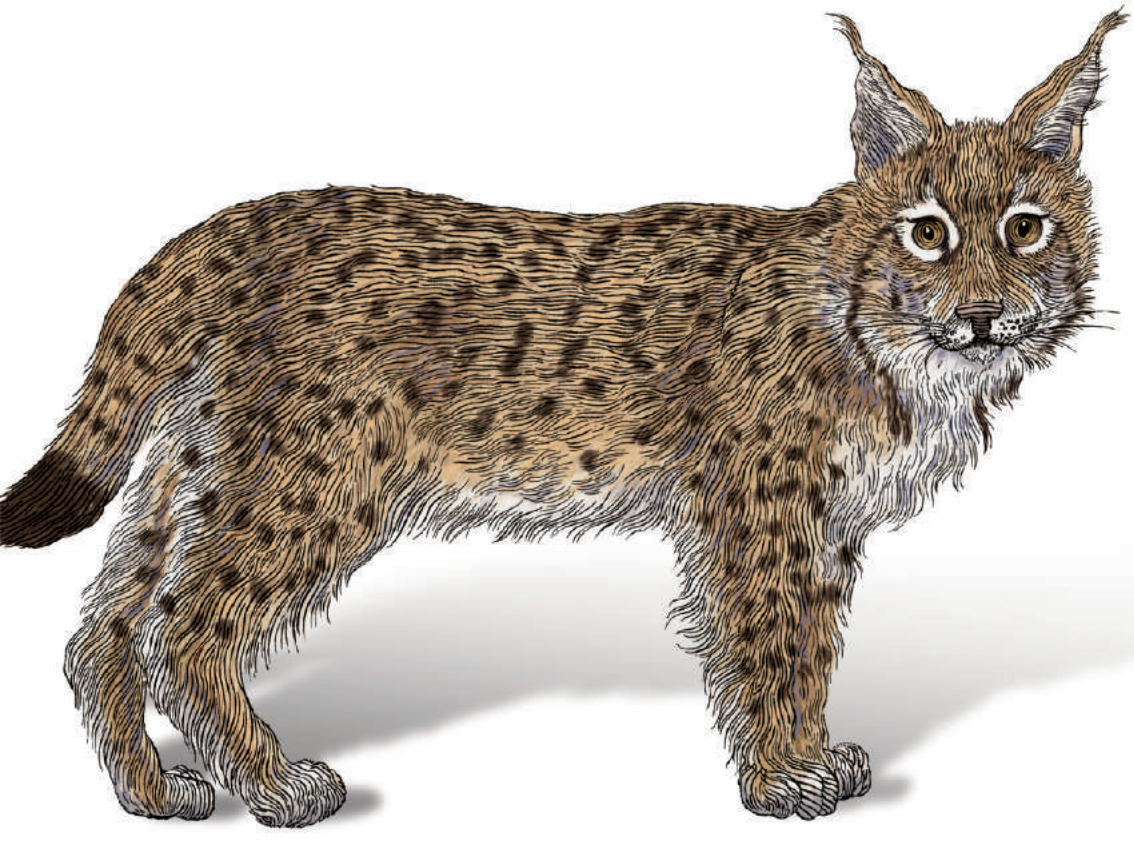
O'REILLY®

第二版

NGINX 完全指南

实现高性能负载均衡的进阶实操指南



Derek DeJonghe

O'REILLY®

NGINX 完全指南

NGINX 是当今使用最广泛的 Web 服务器之一，部分原因在于它可以用作 HTTP 和其他网络协议的负载均衡器和反向代理服务器。本修订版完全指南通过一些简单易懂的例子解析了应用交付中真实存在的问题。实用的实操指南可帮助您设置开源或商业产品，并利用它们解决各种用例中的问题。

对于了解现代 Web 架构（例如 n 层或微服务设计以及 TCP 和 HTTP 等常见 Web 协议）的专业人士来说，本书的这些实操指南为安全和软件负载均衡以及 NGINX 应用交付平台的监控和维护提供了经过验证的解决方案。您还将了解到免费的 NGINX 开源版以及许可授权版 NGINX Plus 的高级功能。

您将获取以下方面的实操指南：

- HTTP、TCP 和 UDP 高性能负载均衡
- 通过加密流量、安全链接、HTTP 身份验证子请求等保护访问
- 将 NGINX 部署到 Google、亚马逊云科技（AWS）和 Azure 云
- 设置和配置 NGINX Controller
- 安装和配置 NGINXApp Protect 模块
- 通过控制器 ADC 启用 WAF
- NGINX Instance Manager、NGINX Service Mesh 和 njs 模块

“NGINX 是当今市场上超级强大且全面的工具。本书为大型架构提供了终极配置指南，其中展示的用例几乎可以帮助任何人解决在微服务环境中运行时出现的任何问题，同时又不会失去对业务的关注。”

—— Gonzalo Spina

Brubank 软件工程师 Brubank

Derek DeJonghe 在 Web 开发、系统管理和网络方面的深厚背景和丰富经验让他对现代 Web 架构有着全面的了解。他擅长各种规模的云迁移和运维，领导着一支由云架构师和解决方案工程师组成的优秀团队，并为许多不同的应用开发可以自我修复、自动扩展的基础架构。

系统管理

Twitter：@oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

第二版

NGINX 完全指南

实现高性能负载均衡的进阶实操指南

Derek DeJonghe

北京 · 波士顿 · 法纳姆 · 塞巴斯托波尔 · 东京

O'REILLY®

NGINX 完全指南

作者：Derek DeJonghe

策划编辑： Melissa Duffield
开发编辑： Gary O'Brien
制作编辑： Christopher Faucher
文字编辑： Kim Cofer
英文版校对： JM Olejarz
索引： Potomac Indexing, LLC
内部设计： David Futato
封面设计： Karen Montgomery
插画： Kate Dullea
中文版校对： 熊平、林静、易久平、孔令子

目录

前言

《NGINX 完全指南》旨在通过一些简单易懂的例子解析应用交付中真实存在的问题。本书内容丰富全面，探讨了 NGINX 的大部分主要特性，可帮助您学习如何使用这些功能。

本书涉及到了免费的开源版 NGINX 软件以及 NGINX Inc. 的商用产品 NGINX Plus 和 NGINX Controller。对于那些仅可通过付费订阅 NGINX Plus 才能获得的功能和指令，本书将作出相关注释。由于 NGINX Plus 是一种*应用交付控制器（ADC）*，提供了许多高级功能，重点解释这些功能将有助于读者全面了解这一平台所提供的可能性。

本书将先解释 NGINX 和 NGINX Plus 的安装程序以及一些供 NGINX 新手借鉴的基础入门步骤。后面的内容将过渡到各种形式的负载均衡，并分章节提供了关于流量管理、缓存以及自动化的配套内容。第 6 章“身份验证”的涉及面很广，但同时也很重要，因为 NGINX 通常是 Web 流量进入应用的第一个入口点，也是应用层抵御 Web 攻击和漏洞的第一道防线。书中还有许多章节介绍了 HTTP/2、流媒体、云、service mesh（服务网格）和容器环境等前沿主题，最后又以一些更传统的运维主题收尾，例如监控、调试、性能和运维技巧等。本书最后将介绍 NGINX Instance Manager[1] 和 NGINX Controller[2]（一个以应用为中心的管理平台）。

我个人将 NGINX 用作一个多功能工具，相信本书也能帮您实现这一点。这是一款可靠、好用又值得信赖的软件。今日得幸于此畅谈，希望本书能给您带来些许启发，帮助您运用 NGINX 解决棘手的现实问题。

[1] NGINX Instance Manager 已于 2022 年正式合并入新产品 NGINX Management Suite，点击了解详情。

[2] NGINX Controller 已于 2022 年正式停售，点击了解相关新产品 NGINX Management Suite。

本书的使用惯例

本书使用了以下印刷惯例：

斜体

表示新术语、URL、电子邮件地址、文件名和文件扩展名。

此元素表示一般注释。

此元素表示警告或注意事项。

O'Reilly 在线学习平台

40 多年来，*O'Reilly Media* 一直致力于提供技术和商业培训、知识以及洞察，帮助企业取得成功。

我们拥有一个独特的专家和创新者网络，他们通过图书、文章和我们的在线学习平台分享知识和专业技能。O'Reilly 在线学习平台允许您按需访问实时培训课程、深度学习路径、交互式编码环境以及来自 O'Reilly 和其他 200 多家出版商的大量精选文本和视频。如欲了解更多信息，请访问：*http://oreilly.com*

如何联系我们

如对本书有任何意见和问题，请发送给出版商：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938（美国或加拿大）

707-829-0515（国际或本地）

707-829-0104（传真）

我们为本书建立了一个网页，上面列出了勘误表、示例和任何其他信息，访问地址：*https://oreil.ly/NGINX-cookbook-2e*

如对本书有任何意见或技术问题，请发送电子邮件至：*bookquestions@oreilly.com*

有关我们的图书和课程的新闻和信息，请访问：*https://oreilly.com*

请关注我们的 LinkedIn：*https://linkedin.com/company/oreilly-media*

请关注我们的 Twitter：*https://twitter.com/oreillymedia*

请关注我们的 YouTube 视频：*https://youtube.com/oreillymedia*

致谢

感谢 Hussein Nasser 和 Gonzalo Josue Spina 对本书作出的详细而富有见地的指导，也感谢我的妻子在写作过程中对我的支持。

第 1 章
基础知识

1.0 简介

无论是 NGINX 开源版还是 NGINX Plus，您都要先在系统上安装它们并学习一些基础知识。在本章中，您将学习如何安装 NGINX、主要配置文件位于何处以及管理命令是什么，还将了解如何验证您的安装并向默认服务器发出请求。

1.1 在 Debian/Ubuntu 上安装 NGINX

问题

在 Debian 或 Ubuntu 机器上安装 NGINX 开源版。

解决方案

更新已配置源的软件包信息，并安装一些有助于配置官方 NGINX 软件包仓库的软件包：

```
apt-get update
apt install -y curl gnupg2 ca-certificates lsb-release \
    debian-archive-keyring
```

下载并保存 NGINX 签名密钥：

```
curl https://nginx.org/keys/nginx_signing.key | gpg --dearmor \
    | tee /usr/share/keyrings/nginx-archive-keyring.gpg >/dev/null
```

使用 lsb_release 设置定义操作系统和版本名称的变量，然后创建一个 *apt* 源文件：

```
OS=$(lsb_release -is | tr '[:upper:]' '[:lower:]')
RELEASE=$(lsb_release -cs)
echo "deb [signed-by=/usr/share/keyrings/nginx-archive-keyring.gpg] \
    http://nginx.org/packages/${OS} ${RELEASE} nginx" \
    | tee /etc/apt/sources.list.d/nginx.list
```

再次更新软件包信息，然后安装 NGINX：

```
apt-get update
apt-get install -y nginx
nginx
```

详解

本节提供的命令可指示高级打包工具（APT）软件包管理系统使用官方 NGINX 软件包仓库。NGINX GPG 软件包签名密钥已下载并保存至文件系统的某个位置，以供 APT 使用。为 APT 提供签名密钥有助于 APT 系统验证仓库中的软件包。lsb_release 命令用于自动确定操作系统和版本名称，用户可以在所有 Debian 或 Ubuntu 发布版本中使用这些指令。apt-get update 命令指示 APT 系统从其已知的仓库中刷新软件包列表，此后您便可以从官方 NGINX 仓库安装 NGINX 开源版。安装后，最后一个命令会启动 NGINX。

1.2 在 RedHat/CentOS 上安装 NGINX

问题

在 RedHat 或 CentOS 上安装 NGINX 开源版。

解决方案

创建一个名为 */etc/yum.repos.d/nginx.repo* 的文件，其中包含以下内容：

```
[nginx]
name=nginx repo
baseurl=http://nginx.org/packages/OS/OSRELEASE/$basearch/
gpgcheck=0
enabled=1
```

修改文件，将 URL 中间的 OS 替换为 rhel 或 centos，具体取决于您的发行版本。分别将版本 7.x 或 8.x 的 OSRELEASE 替换为 7 或 8。然后运行以下命令：

```
yum -y install nginx
systemctl enable nginx
systemctl start nginx
firewall-cmd --permanent --zone=public --add-port=80/tcp
firewall-cmd --reload
```

详解

您刚刚为此解决方案创建的文件将指示 YUM 软件包管理系统使用官方 NGINX 开源版软件包仓库。后面的命令将从官方仓库安装 NGINX 开源版，指示 systemd 在启动时启用 NGINX，并告知它立即启动。防火墙命令为 TCP 协议打开端口 80，这是 HTTP 的默认端口。最后一个命令重新加载防火墙，以提交更改。

1.3 安装 NGINX Plus

问题

安装 NGINX Plus。

解决方案

请访问 *http://cs.nginx.com/repo_setup*。从下拉菜单中选择您要将 NGINX Plus 安装到哪个操作系统，然后按照说明进行操作。其安装说明与开源解决方案类似，但是您需要安装一个证书，以便对 NGINX Plus 仓库进行身份验证。

详解

NGINX 会及时更新这个仓库安装指南和 NGINX Plus 安装说明。这些说明因您的操作系统和版本而略有不同，但都有一个共同点：您必须从 NGINX 门户获取证书和密钥并提供给您的系统，以便对 NGINX Plus 仓库进行身份验证。

1.4 验证安装

问题

验证 NGINX 是否安装成功并检查版本。

解决方案

您可以使用以下命令验证 NGINX 是否安装成功并检查其版本：

```
$ nginx -v
nginx version: nginx/1.21.3
```

如本例所示，响应显示了版本。

您可以使用以下命令确认 NGINX 是否正在运行：

```
$ ps -ef | grep nginx
root      1738     1  0 19:54 ?  00:00:00 nginx: master process
nginx     1739  1738  0 19:54 ?  00:00:00 nginx: worker process
```

ps 命令列出了正在运行的进程。通过将该命令导入到 grep 中，您可以在输出中搜索特定词。此示例使用 grep 搜索 nginx。结果显示有两个正在运行的进程：master 和 worker。如果 NGINX 正在运行中，您将始终可以看到一个 master 以及一个或多个 worker 进程。请注意，master 进程以 root 身份运行，因为 NGINX 只有使用最高权限才能正常运行。有关启动 NGINX 的说明，请参阅下一节。要了解如何将 NGINX 作为守护进程启动，请使用 init.d 或 systemd 方法。

要验证 NGINX 能否正确返回请求，请使用浏览器向您的机器发出请求或使用 curl。发送请求时，请使用机器的 IP 地址或主机名。如果安装在本地，您可以使用 localhost，如下所示：

```
curl localhost
```

您将看到 NGINX 欢迎页面默认的 HTML 站点。

详解

nginx 命令允许您与 NGINX 二进制文件交互，以便检查版本、列出已安装的模块、测试配置以及向 master 进程发送信号。NGINX 必须在运行时才能服务请求。ps 命令是一种确定 NGINX 是守护进程还是前台进程的可靠方法。NGINX 默认提供的配置在端口 80 上运行静态站点 HTTP 服务器。为了测试这一默认站点，您可以使用 localhost 以及主机的 IP 地址和主机名向机器发送 HTTP 请求。

1.5 关键文件、目录和命令

问题

了解重要的 NGINX 目录和命令。

解决方案

NGINX 文件和目录

以下文件、目录和命令对于 NGINX 新手来说十分重要。

/etc/nginx/

/etc/nginx/ 目录是 NGINX 服务器的默认配置根，您可以从中找到指示 NGINX 如何运行的配置文件。

/etc/nginx/nginx.conf

/etc/nginx/nginx.conf 文件是 NGINX 服务使用的默认配置入口点。此配置文件能够为 worker 进程、调优、日志记录、动态模块的加载以及对其他 NGINX 配置文件的引用设置全局设置。在默认配置中，*/etc/nginx/nginx.conf* 文件包括顶层 http 代码块，也就是上下文，它提供了下述目录中的所有配置文件。

/etc/nginx/conf.d/

/etc/nginx/conf.d/ 目录包含默认的 HTTP 服务器配置文件，其中以 *.conf* 结尾的文件都包含在 */etc/nginx/nginx.conf* 文件的顶层 http 代码块中。最佳实践是利用 include 语句并以这种方式组织配置，从而保持配置文件的简洁。在某些软件包仓库中，此文件夹被命名为 *sites-enabled*，配置文件链接到 *site-available* 文件夹；此惯例已不再使用。

/var/log/nginx/

/var/log/nginx/ 目录是 NGINX 的默认日志位置，您可以从中找到一个 *access.log* 文件和 *error.log* 文件。访问日志包含 NGINX 服务的每条请求的条目。如果启用了 debug 模块，则错误日志文件包含错误事件和调试信息。

NGINX 命令

`nginx -h`

显示 NGINX 帮助菜单。

`nginx -v`

显示 NGINX 版本。

`nginx -V`

显示 NGINX 版本、build 信息和配置参数，这些参数显示了 NGINX 二进制文件中内置的模块。

`nginx -t`
: 测试 NGINX 配置。

`nginx -T`
: 测试 NGINX 配置并将验证后的配置打印到屏幕上。此命令在寻求支持时很有用。

`nginx -s signal`
: -s 标记向 NGINX master 进程发送信号。您可以发送 stop、quit、reload 和 reopen 等信号。stop 信号可立即停止 NGINX 进程。quit 信号会在完成当前正在处理的请求后停止 NGINX 进程。reload 信号可重新加载配置。reopen 信号指示 NGINX 重新打开日志文件。

详解

在了解这些关键文件、目录和命令后，您就可以准备开始使用 NGINX 了。您可以运用这些知识更改默认配置文件，并使用 nginx -t 命令测试您的更改。如果测试成功，您还将了解到如何使用 nginx -s reload 命令指示 NGINX 重新加载配置。

1.6 提供静态内容

问题

使用 NGINX 提供静态内容。

解决方案

使用以下 NGINX 配置示例覆盖位于 */etc/nginx/conf.d/default.conf* 的默认 HTTP 服务器配置：

```
server {
    listen 80 default_server;
    server_name www.example.com;

    location / {
        root /usr/share/nginx/html;
        # alias /usr/share/nginx/html;
        index index.html index.htm;
    }
}
```

详解

此配置通过 HTTP 在端口 80 上从目录 */usr/share/nginx/html/* 提供静态文件。第一行配置定义了一个新的 server 代码块，这为 NGINX 定义了一个需要侦听的新上下文。第二行指示 NGINX 侦听端口 80，default_server 参数指示 NGINX 使用此服务器作为端口 80 的默认上下文。listen 指令也可以使用一系列端口。server_name 指令定义了主机名或应定向到此服务器的请求的名称。如果配置没有将此上下文定义为 default_server，那么只有当 HTTP 主机请求头与提供给 server_name 指令的值相匹配时，NGINX 才会将请求定向到这台服务器。如果您还没有要使用的域名，则可以通过设置 default_server 上下文省略 server_name 指令。

location 代码块根据 URL 中的路径定义配置。路径或域之后的部分 URL 被称为统一资源标识符（URI）。理想情况下，NGINX 会将请求的 URI 匹配到一个 location 代码块。示例使用了 / 匹配所有请求。root 指令向 NGINX 显示了为给定上下文提供内容时应在何处查找静态文件。在查找请求的文件时，请求的 URI 会附加到 root 指令的值。如果我们为 location 指令提供了 URI 前缀，那么除非我们使用 alias 指令（而非 root），否则该前缀将包含在附加路径中。location 指令能够匹配一系列广泛的表达式。有关更多信息，请访问下方“其他参考资料”中的链接。最后，index 指令为 NGINX 提供了一个默认文件或要检查的文件列表，以防 URI 中没有提供进一步的路径。

其他参考资料

NGINX HTTP location 指令文档

1.7 优雅重载

问题

在保证不丢包的情况下重新加载配置。

解决方案

通过使用 NGINX 的 reload 方法，您可以在不中止服务器的情况下有条不紊地重载配置：

```
nginx -s reload
```

此示例使用 NGINX 二进制文件向 master 进程发送信号，从而达到重载 NGINX 系统的目的。

详解

通过在不中止服务器的情况下重载 NGINX 配置，您将能够动态更改配置，同时又不丢失任何数据包。在正常运行时间较长的动态环境中，您需要在某个时间点更改负载均衡配置。NGINX 允许您在保持负载均衡器在线的同时执行此操作。此功能提供了无数可能性，例如在实时环境中重新运行配置管理，或者构建应用感知型和集群感知型模块来动态配置和重载 NGINX，从而满足环境需求。

第 2 章
高性能负载均衡

2.0 简介

如今的互联网用户体验离不开出色的性能和正常运行时间。为此，企业需要运行多个相同的副本，并将负载分散在整个系统集群上。随着负载的增加，集群内会引入新的副本。这种架构技术称为*水平扩展*。基于软件的基础架构因其灵活性而愈发受欢迎，并创造了更多的可能性。无论是仅有两个系统副本组成的高可用性方案，还是全球范围内成千上万个系统构成的大型集群，它们都需要一款像基础架构一样动态的负载均衡解决方案。NGINX 能够以多种方式满足这一需求，例如 HTTP、TCP 和用户数据报协议（UDP）负载均衡，本章将会对这些内容进行详细介绍。

在实施负载均衡时，必须要保证对客户体验产生完全正面的影响。许多现代 Web 架构采用无状态应用层，将状态存储在共享内存或数据库中。但是并非所有架构都是如此。在交互式应用中，会话状态不仅蕴含着重大价值，而且应用十分广泛。出于多种原因 —— 例如在需要处理大量数据的应用中，为了避免性能方面的高昂网络开销，这个状态可能就会存储到本地的应用服务器上。在这种情况下，后续的请求会被继续交付到同一台服务器，这对保障用户体验极其重要。与此同时，在会话完成之前，服务器不会得到释放。大规模使用有状态应用需要智能负载均衡器。NGINX Plus 提供了多种跟踪 cookie 或路由的办法来解决这一问题。本章介绍了会话保持，因为它与 NGINX 和 NGINX Plus 的负载均衡有关。

确保 NGINX 所服务的应用的健康状态十分重要。出于多种原因，上游（upstream）请求可能会失败，例如由于网络连接不佳、服务器故障或应用故障等等。代理和负载均衡器必须足够智能，以检测上游服务器（负载均衡器或代理后面的服务器）的故障，并停止向它们传输流量；否则客户端只能等待，最终超时。当服务器出现故障时，一种避免服务降级的方法是让代理检查上游服务器的健康状况。NGINX 提供了两种不同类型的健康检查方法：被动式（NGINX 开源版提供）和主动式

（仅 NGINX Plus 提供）。定期执行的主动健康检查会与上游服务器建立连接或向其发送请求，并且验证响应是否正确。被动健康检查能够在客户端发出请求或建立连接时监控上游服务器的连接或响应。您可能希望使用被动健康检查来减少上游服务器的负载，并使用主动健康检查确定上游服务器的故障，以免引发客户端服务失败。本章最后探讨了如何监控正在实施负载均衡的上游应用服务器的健康状况。

2.1 HTTP 负载均衡

问题

将负载分发到两台或多台 HTTP 服务器。

解决方案

在 NGINX 的 HTTP 模块内使用 upstream 代码块对 HTTP 服务器实施负载均衡：

```
upstream backend {
    server 10.10.12.45:80      weight=1;
    server app.example.com:80  weight=2;
    server spare.example.com:80  backup;
}
server {
    location / {
        proxy_pass http://backend;
    }
}
```

该配置对端口 80 的两台 HTTP 服务器实施负载均衡，然后再将另一台服务器定义为 backup，以便在两台主服务器不可用时发挥作用。weight 参数指示 NGINX 向第二台服务器传输两倍的请求，它的默认值为 1。

详解

HTTP 的 upstream 模块控制着 HTTP 负载均衡。该模块定义了一个目标池 —— 它可以是 Unix 套接字、IP 地址和 DNS（域名服务）记录的任意组合，也可以是它们的混合使用配置。upstream 模块还定义了如何将任一个请求分发给任何上游（upstream）服务器。

每个上游目标都通过 server 指令在上游池中进行定义。server 指令接收 Unix 套接字、IP 地址或 FQDN *（全限定域名）*以及一些可选的参数。可选参数能够增强对请求路由的控制。这包括均衡算法中服务器的 weight 参数（无论服务器处于待机模式、可用还是不可用），以及确定服务器是否不可用的参数。NGINX Plus 还提供了许多其他好用的参数，例如对服务器的连接限制、高级 DNS 解析控制以及在服务器启动后缓慢增加与服务器的连接等等。

2.2 TCP 负载均衡

问题

将负载分发到两台或多台 TCP 服务器。

解决方案

在 NGINX 的 stream 模块内使用 upstream 代码块对 TCP 服务器实施负载均衡：

```
stream {
    upstream mysql_read {
        server read1.example.com:3306 weight=5;
        server read2.example.com:3306;
        server 10.10.12.34:3306       backup;
    }

    server {
        listen 3306;
        proxy_pass mysql_read;
    }
}
```

此示例中的 server 代码块指示 NGINX 侦听 TCP 端口 3306，并对两个 MySQL 数据库读取副本实施负载均衡，同时将另一台服务器定义为 backup，以便在主服务器崩溃时传输流量。

此配置不会被添加到 *conf.d* 文件夹中，因为该文件夹包含在 http 代码块中；您应该另行创建名为 *stream.conf.d* 的文件夹，打开 *nginx.conf* 文件中的 stream 代码块，添加新文件夹以支持 stream 配置。示例见下。

在 */etc/nginx/nginx.conf* 配置文件中：

```
user nginx;
worker_processes auto;
pid /run/nginx.pid;

stream {
    include /etc/nginx/stream.conf.d/*.conf;
}
```

名为 */etc/nginx/stream.conf.d/mysql_reads.conf* 的文件可能包含以下配置：

```
upstream mysql_read {
    server read1.example.com:3306 weight=5;
    server read2.example.com:3306;
    server 10.10.12.34:3306       backup;
}

server {
    listen 3306;
    proxy_pass mysql_read;
}
```

详解

http 和 stream 上下文之间的主要区别在于它们在 OSI 模型的不同层上运行。http 上下文在应用层（七层）运行，stream 在传输层（四层）运行。这并不意味着 stream 上下文不能通过一些巧妙的脚本获得应用感知能力，而是说 http 上下文是专门为了完全理解 HTTP 协议而设计的，stream 上下文默认情况下只能对数据包进行路由和负载均衡。

TCP 负载均衡由 NGINX 的 stream 模块定义。stream 模块与 HTTP 模块一样，允许您定义上游（upstream）服务器池并配置侦听服务器。在配置服务器侦听给定端口时，您必须定义待侦听的端口或者地址加端口（可选）。然后您必须配置目标服务，无论这是连接另一个地址的直接反向代理还是上游资源池。

配置中有许多选项可以改变 TCP 连接反向代理的属性，包括 SSL/TLS 验证限制、超时和 keepalive 等。这些代理选项的一些值可以是（或者包含）变量，例如下载速率、验证 SSL/TLS 证书时使用的名称等。

TCP 与 HTTP 负载均衡中的 upstream 指令非常相似，它们均将上游资源定义为服务器，配置格式同样为 Unix 套接字、IP 或 FQDN，此外服务器 weight 参数、最大连接数、DNS 解析器、连接数缓增期以及判断服务器是激活状态、故障状态还是备用模式的参数也都相似。

NGINX Plus 甚至提供了更多 TCP 负载均衡特性，这些高级特性贯穿整份指南。本章稍后将介绍所有负载均衡的健康检查。

2.3 UDP 负载均衡

问题

将负载分发到两台或多台 UDP 服务器。

解决方案

在 NGINX 的 stream 模块内使用 upstream 代码块（定义为 udp）对 UDP 服务器实施负载均衡：

```
stream {
    upstream ntp {
        server ntp1.example.com:123 weight=2;
        server ntp2.example.com:123;
    }

    server {
        listen 123 udp;
        proxy_pass ntp;
    }
}
```

这部分配置对使用 UDP 协议的两台上游（upstream）网络时间协议（NTP）服务器实施了负载均衡。UDP 负载均衡的指定非常简单，只需使用 listen 指令中的 udp 参数便可。

如果进行负载均衡的服务需要在客户端和服务器之间来回发送多个数据包，则可以指定 reuseport 参数。例如，OpenVPN、互联网语音协议（VoIP）、虚拟桌面解决方案和数据报传输层安全（DTLS）都是这种类型的服务。下面举例说明了如何使用 NGINX 处理 OpenVPN 连接并将其代理到本地运行的 OpenVPN 服务：

```
stream {
    server {
        listen 1195 udp reuseport;
        proxy_pass 127.0.0.1:1194;
    }
}
```

详解

您可能会问：“我的 DNS A 或服务记录（SRV 记录）中已经有多个主机了，又何必再要负载均衡器呢？”原因是我们不仅有备选的负载均衡算法，我们还可以对 DNS 服务器本身实施负载均衡。UDP 服务构成了我们在网络系统中依赖的许多服务，例如 DNS、NTP、QUIC、HTTP/3 和 VoIP。UDP 负载均衡可能对某些企业来说不太常见，但在大型环境中十分有用。

与 TCP 类似，您可以在 stream 模块中找到 UDP 负载平衡，并以同样的方式完成大部分配置。两者的主要区别在于，UDP 负载均衡的 listen 指令指定打开的套接字用于处理数据报。此外，在处理数据报时，UDP 负载均衡还提供了 TCP 所没有的一些其他指令，例如 proxy_response 指令，它向 NGINX 指定了可以从上游服务器发送多少预期的响应。默认情况下，除非达到 proxy_timeout 限制，否则这一数量是无限的。proxy_timeout 指令设置了在连接关闭之前，客户端或代理服务器连接连续进行两次读取或写入操作之间的时间。

reuseport 参数指示 NGINX 为每个 worker 进程创建一个单独的侦听套接字。这允许内核在 worker 进程之间分发传入的连接，以处理在客户端和服务器之间发送的多个数据包。reuseport 功能仅适用于 Linux kernels 3.9 及更高版本、DragonFly BSD、FreeBSD 12 及更高版本。

2.4 负载均衡方式

问题

轮询负载均衡不适合您的用例，因为您有异构工作负载或服务器池。

解决方案

使用 NGINX 的负载均衡方法之一，例如最少连接、最短时间、通用哈希、随机算法或 IP 哈希。此示例将后端上游（upstream）池的负载均衡算法设为了最少连接：

```
upstream backend {
    least_conn;
    server backend.example.com;
    server backend1.example.com;
}
```

除了通用哈希、随机算法和最短时间外，所有其他负载均衡算法都是独立的指令，如上例所示。下面的“详解”一节解释了这些指令的参数。

详解

并非所有请求或数据包都有相同的权重。有鉴于此，轮询甚至上例使用的加权轮询都将无法满足所有应用或流量的需求。NGINX 提供了多种负载均衡算法，您可以根据特定的用例进行选择，也可以对您选择的算法进行配置。以下负载均衡方法可用于上游 HTTP、TCP 和 UDP 池：

轮询

轮询是默认的负载均衡方法，它按照上游池中服务器列表的顺序分发请求。当上游服务器的容量变化时，您还可以考虑使用加权轮询。权重的整数值越高，服务器在轮询中的优势就越大。权重背后的算法只是加权平均值的统计概率。

最少连接

此方法通过将当前请求代理到打开连接数最少的上游服务器实现负载均衡。与轮询一样，在决定将连接发送到哪台服务器时，最少连接也会考虑权重。其指令名称是 least_conn。

最短时间

该算法仅在 NGINX Plus 中提供，与最少连接算法类似，它将请求代理到当前连接数最少的上游服务器，但首选平均响应时间最短的服务器。此方法是最复杂的负载均衡算法之一，能够满足高性能 Web 应用的需求。最短时间在最少连接的基础上进行了优化，因为少量连接并不一定意味着最快的响应。使用此算法时，切记要考虑服务请求时间的统计差异。有些请求可能本身就需要更多的处理，请求时间也就更长，因而拉宽了统计的范围。请求时间长并不一定意味着服务器性能欠佳或超负荷工作。但是，需要进行更多处理的请求可以考虑使用异步工作流。用户必须为此指令指定 header 或 last_byte 参数。当指定 header 时，使用接收响应头的时间；当指定 last_byte 时，使用接收完整响应的时间。其指令名称是 least_time。

通用哈希

管理员使用请求或运行时给定的文本、变量或两者的组合定义哈希值。NGINX 能够为当前请求生成哈希值并将其放在上游服务器上，从而在这些服务器之间分发负载。当您希望更好地控制请求的发送位置或确定哪台上游服务器最有可能缓存数据

时，此方法就会派上用场。请注意，当从池中添加或删除服务器时，将重新分发哈希请求。此算法有一个可选的参数：consistent，它能够将重新分发带来的影响最小化。其指令名称是 hash。

随机

该方法用于指示 NGINX 从组中随机选择一台服务器，同时考虑服务器的权重。可选的 two [method] 参数指示 NGINX 随机选择两台服务器，然后使用提供的负载均衡方法对两者均匀地分发请求。默认情况下，如果传输的参数只有 two，没有 method，则使用 least_conn 方法。随机负载均衡的指令名称是 random。

IP 哈希

此方法仅适用于 HTTP。IP 哈希算法使用客户端 IP 地址作为哈希。IP 哈希与通用哈希存在细微的不同，前者使用 IPv4 地址的前三个八进制位或整个 IPv6 地址，而后者使用的是远程变量。当会话状态十分重要，但又无法通过应用的共享内存进行处理时，此方法可确保客户端始终被代理到同一上游服务器（只要服务器可用）。此方法在分发哈希值时也考虑了 weight 参数。其指令名称是 ip_hash。

2.5 NGINX Plus 之 Sticky Cookie

问题

使用 NGINX Plus 将下游（downstream）客户端绑定到上游（upstream）服务器。

解决方案

使用 sticky cookie 指令指示 NGINX Plus 创建和跟踪 cookie：

```
upstream backend {
    server backend1.example.com;
    server backend2.example.com;
    sticky cookie
            affinity
            expires=1h
            domain=.example.com
            httponly
            secure
            path=/;
}
```

此配置创建并跟踪了一个 cookie，该 cookie 可以将下游客户端绑定到上游服务器。在此示例中，cookie 被命名为 affinity，为 *example.com* 设置，有效期为一个小时，无法在客户端使用，只能通过 HTTPS 发送，并且对所有路径有效。

详解

在 sticky 指令中使用 cookie 参数，为第一个请求创建一个包含上游服务器信息的 cookie。NGINX Plus 将跟踪此 cookie，允许它继续将后续请求定向到同一台服务器。cookie 参数的第一个位置参数是待创建和跟踪的 cookie 的名称。其他参数提供了额外的控制，为浏览器提供了相应的使用信息 —— 如有效期、域、路径，以及 cookie 是否可以在客户端使用，或者 cookie 是否可以通过不安全的协议传递。

2.6 NGINX Plus 之 Sticky Learn

问题

使用 NGINX Plus 的现有 cookie 将下游（downstream）客户端绑定到上游（upstream）服务器。

解决方案

使用 sticky learn 指令发现和跟踪上游应用创建的 cookie：

```
upstream backend {
    server backend1.example.com:8080;
    server backend2.example.com:8081;

    sticky learn
            create=$upstream_cookie_cookiename
            lookup=$cookie_cookiename
            zone=client_sessions:2m;
}
```

此示例指示 NGINX 通过在响应头中查找名为 COOKIENAME 的 cookie 来查找和跟踪会话，并通过在请求头中查找相同的 cookie 来查找现有会话。此会话的 affinity 被存储在一个 2MB 的共享内存区中，后者可以跟踪大约 16,000 个会话。cookie 的名称始终是应用专有的。常用的 cookie 名称（如 jsessionid 或 phpsessionid）通常是应用或应用服务器配置中的默认设置。

详解

当应用创建自己的会话状态 cookie 时，NGINX Plus 可以在请求响应中发现它们并跟踪它们。当在 sticky 指令中使用 learn 参数时，就会执行这种类型的 cookie 跟踪。跟踪 cookie 的共享内存由 zone 参数指定，包括名称和大小。NGINX Plus 通过指定 create 参数来从上游服务器的响应中查找 cookie，并使用 lookup 参数搜索之前注册的服务器 affinity。这些参数的值是 HTTP 模块暴露的变量。

2.7 NGINX Plus 之 Sticky Routing

问题

使用 NGINX Plus 对持久会话路由到上游（upstream）服务器的方式进行细粒度控制。

解决方案

通过带 route 参数的 sticky 指令使用关于请求的变量进行路由：

```
map $cookie_jsessionid $route_cookie {
    ~.+\.(?P<route>\w+)$ $route;
}

map $request_uri $route_uri {
    ~jsessionid=.+\.(?P<route>\w+)$ $route;
}
upstream backend {
    server backend1.example.com route=a;
    server backend2.example.com route=b;

    sticky route $route_cookie $route_uri;
}
```

此示例尝试提取 Java 会话 ID，首先通过将 Java 会话 ID cookie 映射到第一个 map 代码块的变量来提取，然后查找请求 URI 中名为 jsessionid 的参数，将值映射到使用第二

个 map 代码块的变量。带有 route 参数的 sticky 指令可以传输任意数量的变量。第一个非零或非空值用于路由。如果使用 jsessionId cookie，请求将被路由到 backend1；如果使用 URI 参数，请求将被路由到 backend2。尽管此示例使用的是 Java 通用会话 ID，但这也同样适用于其他会话技术（如 phpsessionid）或者您的应用为会话 ID 生成的任何有保证的唯一标识符。

详解

有时，您可能希望利用更细粒度的控制，将流量定向到特定的服务器。sticky 指令的 route 参数就是为了实现这个目标而构建的。与通用哈希负载均衡算法相比，sticky route 能够为您提供更好的控制、实际跟踪能力和粘性。该算法会先根据指定的路线将客户端路由到上游服务器，随后的请求则会在 cookie 或 URI 中携带路由信息。Sticky route 采用了许多要计算的位置参数。第一个非空变量用于路由到服务器。map 代码块可以选择性地解析变量，并将它们保存为要在路由中使用的其他变量。本质上，sticky route 指令会在 NGINX Plus 共享内存区内创建一个会话，用于跟踪为上游服务器指定的任何客户端会话标识符，从而始终一致地将具有此会话标识符的请求传输到与原请求相同的上游服务器。

2.8 NGINX Plus 之连接清空

问题

出于维护或其他原因，您需要使用 NGINX Plus 优雅地移除服务器，同时仍然提供会话。

解决方案

借助 NGINX Plus API（详见第 5 章），使用 drain 参数指示 NGINX 停止发送未被跟踪的新连接：

```
$ curl -X POST -d '{"drain":true}' \
    'http://nginx.local/api/3/http/upstreams/backend/servers/0'

{
    "id":0,
    "server":"172.17.0.3:80",
```

```
        "weight":1,
        "max_conns":0,
        "max_fails":1,
        "fail_timeout":
        "10s","slow_start":
        "0s",
        "route":"",
        "backup":false,
        "down":false,
        "drain":true
    }
```

详解

当会话状态被存储到本地服务器时，必须要先清空连接和持久会话，然后再从池中删除服务器。连接清空是指在从上游（upstream）池中删除服务器之前，先让服务器会话在本地失效的过程。您可以通过将 drain 参数添加到 server 指令来配置特定服务器的清空。如果设置了 drain 参数，NGINX Plus 会停止向该服务器发送新会话，但允许在当前会话长度内继续提供当前会话。您还可以通过将 drain 参数添加到上游服务器指令来切换此配置，然后重新加载 NGINX Plus 配置。

2.9 被动健康检查

问题

被动检查上游（upstream）服务器的健康状况。

解决方案

通过 NGINX 健康检查和负载均衡确保只使用健康的上游服务器：

```
upstream backend {
    server backend1.example.com:1234 max_fails=3 fail_timeout=3s;
    server backend2.example.com:1234 max_fails=3 fail_timeout=3s;
}
```

此配置能够监控定向到上游服务器的客户端请求的响应，从而被动监控上游服务器的健康状况。该示例将 max_fails 指令设置为 3，将 fail_timeout 设置为 3 秒。这些指令参数在 stream 和 HTTP 服务器中的工作原理相同。

详解

NGINX 开源版提供了被动健康检查功能，并且使用了相同的 server 参数来实施 HTTP、TCP 和 UDP 负载均衡。当客户端发出请求时，被动监控功能可以监测通过 NGINX 的失效或超时连接。默认情况下启用被动健康检查；此处提到的参数允许您调整它们的行为。max_fails 的默认值为 1，fail_timeout 的默认值为 10s。健康监控在所有类型的负载均衡中都很重要，这不仅是为了保障用户体验，也是为了实现业务连续性。NGINX 能够被动监控上游 HTTP、TCP 和 UDP 服务器，确保它们健康、高效地运行。

其他参考资料

HTTP 健康检查管理指南
TCP 健康检查管理指南
UDP 健康检查管理指南

2.10 NGINX Plus 之主动健康检查

问题

使用 NGINX Plus 主动检查上游（upstream）服务器的健康状况。

解决方案

对于 HTTP，请使用 location 代码块中的 health_check 指令：

```
http {
    server {
        # ...
        location / {
            proxy_pass http://backend;
            health_check interval=2s
                fails=2
                passes=5
                uri=/
                match=welcome;
        }
    }
    # 状态码是 200，内容类型是 "text/html",
    # 正文包含 "Welcome to nginx!"
    match welcome {
        status 200;
        header Content-Type = text/html;
        body ~ "Welcome to nginx!";
    }
}
```

此处的 HTTP 服务器健康检查配置通过每 2 秒向 URI “/” 发送 HTTP GET 请求来检查上游服务器的健康状况。我们无法为健康检查定义 HTTP 方法，只能执行 GET 请求，因为其他方法可能会更改后端系统的状态。上游服务器只有连续通过五次健康检查才能被认为是健康的。如果它们连续两次未通过检查，则被认定为不健康。上游服务器的响应必须匹配定义的 match 代码块，后者将状态码定义为 200，将请求头的 Content-Type 值定义为 ‘text/html’ ，同时定义了响应正文中的字符串 “Welcome to nginx!” 。HTTP match 代码块具有三个指令：status、header 和 body。这三个指令均带有比较标记。

TCP/UDP 服务的 stream 健康检查非常相似：

```
stream {
    # ...
    server {
        listen 1234;
        proxy_pass stream_backend;
        health_check interval=10s
            passes=2
            fails=3;
        health_check_timeout 5s;
    }
    # ...
}
```

在此示例中，TCP 服务器配置为侦听端口 1234，并代理到一组上游服务器，它将主动检查这些服务器的健康状况。除了 uri 之外，stream health_check 指令与 HTTP 中的其他参数都相同，并且 stream 版本有一个可以将检查协议切换到 udp 的参数。在此示例中，间隔时间设为 10 秒，规定通过两次被视为健康，失败三次被视为不健康。主动 stream 健康检查也能验证来自上游服务器的响应。但是 stream 服务器的 match 代码块只有两个指令：send 和 expect。send 指令是要发送的原始数据，expect 是确切的响应或要匹配的正则表达式。

详解

NGINX Plus 允许使用被动或主动健康检查来监控源服务器。这些健康检查可以测量的不仅仅是响应代码。在 NGINX Plus 中，主动 HTTP 健康检查根据一系列有关上游服务器响应的接受标准实施监控。您可以对主动健康检查的监控进行配置，包括上游服务器的检查频率、服务器必须通过多少次检查才能被视为健康、失败多少次会被视为不健康以及预期的结果应该是什么。对于更复杂的逻辑，match 代码块的 require 指令

允许使用值不能为空或零的变量。match 参数指向了定义响应接受标准的 match 代码块。当在 TCP/UDP 的 stream 上下文中使用时，match 代码块还定义了要发送到上游服务器的数据。这些特性使得 NGINX 能够确保上游服务器始终处于健康状态。

其他参考资料

HTTP 健康检查管理指南
TCP 健康检查管理指南
UDP 健康检查管理指南

2.11 NGINX Plus 之慢启动

问题

您的应用需要缓慢增加流量，直至承担全部生产负载。

解决方案

当上游（upstream）负载均衡池重新引入服务器时，使用 server 指令中的 slow_start 参数在指定的时间内逐渐增加连接数：

```
upstream {
    zone backend 64k;

    server server1.example.com slow_start=20s;
    server server2.example.com slow_start=15s;
}
```

server 指令配置将在上游池重新引入服务器后缓慢增加流量。server1 和 server2 将分别在 20 秒和 15 秒内缓慢增加连接数量。

详解

*慢启动*是指在一段时间内缓慢增加代理到服务器的请求数量的概念。慢启动允许应用通过填充缓存和初始化数据库连接来“热身”，而不至于一启动就被“汹涌而来”的连接压垮。当健康检查失败的服务器开始再次通过检查并重新进入负载均衡池时，该功能就会派上用场，并且仅在 NGINX Plus 中提供。慢启动不能与 hash、ip_hash 或 random 负载均衡方法一起使用。

第 3 章
流量管理

3.0 简介

NGINX 和 NGINX Plus 也被归类于 Web 流量控制器。您可以使用 NGINX 智能地路由流量，并根据多个属性控制流量。本章介绍了 NGINX 的多项功能，包括：按比例分割客户端请求；利用客户端的地理位置；通过速率、连接和带宽限制控制流量。阅读时请注意，您可以混合搭配这些功能来解锁更多可能。

3.1 A/B 测试

问题

在文件或应用的两个或多个版本之间分割客户端流量，以测试接受度或参与度。

解决方案

使用 split_clients 模块将一定比例的客户端流量定向到一个不同的上游（upstream）池：

```
split_clients "${remote_addr}AAA" $variant {
    20.0%    "backendv2";
    *        "backendv1";
}
```

split_clients 指令对作为第一个参数提供的字符串进行哈希处理，并用该哈希值除以提供的百分比，以映射作为第二个参数提供的变量的值。在第一个参数中添加“AAA”是为了证明这是一个可以包含多个变量的串联字符串，如通用哈希负载均衡算法中所述。第三个参数是一个包含键值（key-value）对的对象，其中键是百分比权重，值是要分配的值。键可以是百分比或星号。星号表示取完所有百分比后的剩余部分。对

于 $variant 变量的值，backendv2 为客户端 IP 地址流量的 20%，backendv1 为其余 80%。

在此示例中，backendv1 和 backendv2 表示上游服务器池，可以与 proxy_pass 指令一起使用，如下所示：

```
location / {
    proxy_pass http://$variant
}
```

使用变量 $variant 时，我们的流量将会分配给两个不同的应用服务器池。

为了理解 split_clients 的广泛用途，我们通过一个例子来看看两个静态站点版本之间的流量分割：

```
http {
    split_clients "${remote_addr}" $site_root_folder {
        33.3%    "/var/www/sitev2/";
        *        "/var/www/sitev1/";
    }
    server {
        listen 80 _;
        root $site_root_folder;
        location / {
            index index.html;
        }
    }
}
```

详解

在测试电子商务网站上不同类型的营销和前端功能的转化率时，这种类型的 A/B 测试非常有用。应用通常通过“*灰度发布*”进行部署。在这种类型的部署中，用户可以逐步增加路由到新版本的用户百分比，进而将流量慢慢切换到新版本。在推出新版本的代码时，在不同的应用版本之间分割客户端流量非常实用，这可以降低发生错误后带来的影响。更常见的部署模式是蓝绿部署，也就说在将用户切换到新版本的同时，仍然保持旧版本在验证部署期间的可用性。无论出于何种原因在两组不同的应用之间分割客户端流量，NGINX 的 split_client 模块都可以帮您轻松搞定。

其他参考资料

split_clients 模块文档

3.2 使用 GeoIP 模块和数据库

问题

您需要安装 GeoIP 数据库并启用它在 NGINX 中的嵌入式变量，以利用 NGINX 日志、被代理请求或请求路由中客户端的物理位置。

解决方案

安装 NGINX 时，官方 NGINX 开源版软件包仓库（配置方式见第 2 章）提供了一个名为 nginx-module-geoip 的软件包。当使用 NGINX Plus 软件包仓库时，该软件包被命名为 nginx-plus-module-geoip。以下示例展示了如何安装动态 NGINX GeoIP 模块软件包，以及如何下载 GeoIP 国家/地区和城市数据库：

RHEL/CentOS NGINX 开源版：

```
# yum install nginx-module-geoip
```

Debian/Ubuntu NGINX 开源版：

```
# apt-get install nginx-module-geoip
```

RHEL/CentOS NGINX Plus：

```
# yum install nginx-plus-module-geoip
```

Debian/Ubuntu NGINX Plus：

```
# apt-get install nginx-plus-module-geoip
```

下载 GeoIP 国家/地区和城市数据库并解压：

```
# mkdir /etc/nginx/geoip
# cd /etc/nginx/geoip
# wget "http://geolite.maxmind.com/\
download/geoip/database/GeoLiteCountry/GeoIP.dat.gz"
# gunzip GeoIP.dat.gz
# wget "http://geolite.maxmind.com/\
download/geoip/database/GeoLiteCity.dat.gz"
# gunzip GeoLiteCity.dat.gz
```

这组命令在 */etc/nginx* 目录下创建了一个 *geoip* 目录，请移动到这个新目录，下载并解压软件包。

将 GeoIP 国家/地区和城市数据库保存到本地磁盘后，就可以指示 NGINX GeoIP 模块使用它们根据客户端 IP 地址公开嵌入式变量了：

```
load_module "/usr/lib64/nginx/modules/ngx_http_geoip_module.so";

http {
    geoip_country /etc/nginx/geoip/GeoIP.dat;
    geoip_city /etc/nginx/geoip/GeoLiteCity.dat;
# ...
}
```

load_module 指令从它在文件系统中的路径动态加载模块。load_module 指令仅在 main 上下文中有效。geoip_country 指令获取了 *GeoIP.dat* 文件的路径，该文件包含将 IP 地址映射到国家/地区代码的数据库，并且仅在 http 上下文中有效。

详解

要使用此功能，您必须先安装 NGINX GeoIP 模块以及本地 GeoIP 国家/地区和城市数据库，本节演示了相关安装和检索教程。

geoip_country 和 geoip_city 指令暴露了该模块中可用的多个嵌入式变量。geoip_country 指令允许使用一些变量来区分客户端的来源地/国。这些变量包括 $geoip_country_code、$geoip_country_code3 和 $geoip_country_name。country code 变量返回由两个字母组成的国家/地区代码，以 3 结尾的变量返回由三个字母组成的国家/地区代码。country name 变量返回国家/地区的全名。

geoip_city 指令启用了相当一部分变量。geoip_city 指令启用了与 geoip_country 指令相同的所有变量，只不过名称不一样，例如 $geoip_city_country_code、$geoip_city_country_code3 和 $geoip_city_country_name。其他变量包括 $geoip_city、$geoip_lati tude、$geoip_longitude、$geoip_city_continent_code 和 $geoip_postal_code，所有这些都是它们返回的值的描述性变量。$geoip_region 和 $geoip_region_name 描述了地区、领地、州、省、联邦土地等。Region 是由两个字母组成的代码，而 region name 是全名。$geoip_area_code（仅在美国有效）返回三位电话区号。

您可以利用这些变量记录您的客户端信息。您可以选择性地将此信息作为请求头或变量传输给应用，也可以使用 NGINX 以特定的方式路由流量。

其他参考资料

geoip 模块文档
GeoIP Update GitHub

3.3 基于国家/地区的访问限制

问题

根据合同或应用要求限制来自特定国家/地区的访问。

解决方案

将您要阻止或允许的国家代码映射到变量：

```
load_module
    "/usr/lib64/nginx/modules/ngx_http_geoip_module.so";

http {
    map $geoip_country_code $country_access {
        "US"    0;
        default 1;
    }
    # ...
}
```

此映射会将新变量 $country_access 设置为 1 或 0。如果客户端 IP 地址来自美国，该变量将设置为 0；如果客户端 IP 地址来自其他国家/地区，该变量将设置为 1。

现在，在我们的 server 代码块中，我们将使用 if 语句来拒绝来自美国以外的任何人的访问：

```
server {
    if($country_access = '1') {
        return 403;
    }
    # ...
}
```

如果 $country_access 变量设置为 1，则这个 if 语句的计算结果为 True。如果为 True，服务器将返回 403 unauthorized，否则服务器就正常运行。因此，这个 if 代码块只是用来拒绝非美国地区的用户。

详解

这个例子简明扼要地说明了如何仅允许几个国家/地区的访问，您可以根据自己的需求在此基础之上进行延伸。您可以借鉴这一做法，根据 GeoIP 模块提供的任何嵌入式变量允许或阻止访问。

3.4 查找原始客户端

问题

NGINX 服务器前面设置了代理，您需要查找原始客户端 IP 地址。

解决方案

使用 geoip_proxy 指令定义代理 IP 地址范围，并使用 geoip_proxy_recursive 指令查找原始 IP：

```
load_module "/usr/lib64/nginx/modules/ngx_http_geoip_module.so";

http {
    geoip_country /etc/nginx/geoip/GeoIP.dat;
    geoip_city /etc/nginx/geoip/GeoLiteCity.dat;
    geoip_proxy 10.0.16.0/26;
    geoip_proxy_recursive on;
# ...
}
```

geoip_proxy 指令定义了一个*无类域间路由*（CIDR）范围，我们的代理服务器就在这个范围内，并指示 NGINX 利用 X-Forwarded-For 请求头查找客户端 IP 地址。geoip_proxy_recursive 指令指示 NGINX 递归性地从 X-Forwarded-For 请求头中查找已知的最后一个客户端 IP。

Forwarded 请求头已经成为为被代理请求添加代理信息的标准请求头。NGINX GeoIP 模块使用的请求头是 X-Forwarded-For，否则就无法在写入时进行配置。虽然 X-Forwarded-For 不是官方标准，但仍然是被大多数代理广为使用、接受和设置的请求头。

详解

您可能会发现，如果您在 NGINX 前面使用代理，NGINX 将获取代理（而非客户端）的 IP 地址。对此，当在给定范围内打开连接时，您可以使用 geoip_proxy 指令指示 NGINX 使用 X-Forwarded-For 请求头。geoip_proxy 指令使用一个地址或 CIDR 范围。当 NGINX 前面有多个传输流量的代理时，您可以使用 geoip_proxy_recursive 指令递归性地从 X-Forwarded-For 地址中搜索和查找源客户端。当在 NGINX 前面使用 Amazon Web Services Elastic Load Balancing（AWS ELB）、谷歌负载均衡器或 Microsoft Azure 负载均衡器等负载均衡器时，这种方法比较有用。

3.5 限制连接数

问题

根据预定义的键（例如客户端的 IP 地址）限制连接数。

解决方案

构造一个共享内存区来保存连接指标，并使用 limit_conn 指令限制打开的连接数：

```
http {
    limit_conn_zone $binary_remote_addr zone=limitbyaddr:10m;
    limit_conn_status 429;
    # ...
    server {
        # ...
            limit_conn limitbyaddr 40;
        # ...
    }
}
```

此配置创建了一个名为 limitbyaddr 的共享内存区。使用的预定义键是二进制形式的客户端 IP 地址。共享内存区的大小设置为 10 MB。limit_conn 指令使用两个参数：limit_conn_zone 名称和允许的连接数量。limit_conn_status 定义了连接状态被限制为 429 时的响应，此时表示响应过多。limit_conn 和 limit_conn_status 指令在 http、server 和 location 上下文中有效。

详解

通过使用键来限制连接数量，您不仅能够防止滥用，而且还能在所有客户端之间公平共享资源。请务必谨慎使用预定义键。正如我们在前面的示例中所示，如果许多用户位于源自同一 IP 地址的同一网络上，例如当在网络地址转换（NAT）后面时，使用 IP 地址是不合理的，这会使整个客户端组受到限制。limit_conn_zone 指令仅在 http 上下文中有效。您可以利用 NGINX 在 http 上下文中任意数量的可用变量来构建一个限制字符串。有一种更简洁的方法是，根据具体的用例利用一个变量（例如会话 cookie）识别应用层的用户。limit_conn_status 默认值为 503 时，代表服务不可用。由于服务可用，我们最好使用 429，而 500 级的响应码表示服务器错误，400 级的响应码表示客户端错误。

测试限制可能是个很棘手的问题，测试方案通常很难在替代环境中模拟实时流量。在这种情况下，您可以将 limit_req_dry_run 指令设置为 on，然后使用访问日志中的变量 $limit_req_status。$limit_req_sta tus 变量将计算为 PASSED、DELAYED、REJECTED、DELAYED_DRY_RUN 或 REJECTED_DRY_RUN。启用 dry run 后，您将能够分析实时流量日志，并在真正实施限制前根据需要调整限制，从而确保您的限制配置正确。

3.6 限制速率

问题

通过预定义的键（例如客户端的 IP 地址）来限制请求的速率。

解决方案

利用限速模块限制请求速率：

```
http {
    limit_req_zone $binary_remote_addr
        zone=limitbyaddr:10m rate=3r/s;
    limit_req_status 429;
    # ...
    server {
        # ...
            limit_req zone=limitbyaddr;
        # ...
    }
}
```

此示例配置创建了一个名为 limitbyaddr 的共享内存区。使用的预定义键是二进制形式的客户端 IP 地址。共享内存区的大小设置为 10 MB。该区域使用关键字参数设置速率。limit_req 指令使用了一个必不可少的关键字参数：zone。zone 指示了要使用哪个共享内存请求限制区的指令。根据 limit_req_status 指令的定义，超过明示速率的请求将返回 429 HTTP 代码。建议设置一个 400 级范围的状态码，因为默认值是 503，这代表服务器有问题，而实际问题是出在客户端方面。

使用 limit_req 指令的可选关键字参数来启用两级速率限制：

```
server {
    location / {
        limit_req zone=limitbyaddr burst=12 delay=9;
    }
}
```

在某些情况下，客户端需要同时发出许多请求，此后先在一段时间内降低速率，然后再发出更多请求。您可以使用关键字参数 burst 允许客户端超过其速率限制但不拒绝其请求。超出速率的请求将延迟处理，以将速率限制匹配到配置的值。有一组关键字参数可以改变这种行为，即 delay 和 nodelay。nodelay 参数不带值，只允许客户端一次性消耗所有流量突发值；但是必须要先等待足够的时间，直到满足速率限制要求为止，否则所有请求都会被拒绝。在此示例中，如果我们使用 nodelay，客户端可以在第一秒消耗 12 个请求，但是必须要在初始请求之后等待 4 秒才能发出另一个请求。delay 关键字参数定义了在不限流的情况下可以预先发出多少请求。在这种情况下，客户端可以毫无延迟地预先发出 9 个请求，接下来的 3 个将受到限制，此后 4 秒内的任何请求都将被拒绝。

详解

限速模块非常强大，可以防止滥用快速请求，同时仍然为每个人提供优质服务。限制请求速率的原因有很多，安全性就是其中之一。您可以通过严格限制登录页面的速率来防御暴力破解攻击。您可以对所有请求设置合理的限制，从而防止恶意用户试图对您的应用拒绝服务或浪费资源的不轨行为。限速模块的配置很像实操指南 3.5 中描述的连接限制模块，并且存在许多相同的问题。您可以按照每秒限速，也可以按照每分钟限速。当达到速率限制时，日志就会记录事件。此外，还有一条指令没有在示例中给出：limit_req_log_level，它的默认值为 error，您也可以把它设置为 info、notice 或 warn。在 NGINX Plus 中，速率限制目前具有集群感知能力（请参见实操指南 12.5 中的“区域同步”示例）。

测试限制可能是个很棘手的问题，测试方案通常很难在替代环境中模拟实时流量。在这种情况下，您可以将 limit_req_dry_run 指令设置为 on，然后使用访问日志中的变量 $limit_req_status。$limit_req_status 变量将计算为 PASSED、REJECTED 或 REJECTED_DRY_RUN。启用 dry run 后，您将能够分析实时流量日志，并在真正实施限制前根据需要调整限制，从而确保您的限制配置正确。

3.7 限制带宽

问题

按客户端限制资产的下载带宽。

解决方案

使用 NGINX 的 limit_rate 和 limit_rate_after 指令限制响应客户端的带宽：

```
location /download/ {
    limit_rate_after 10m;
    limit_rate 1m;
}
```

这个 location 代码块的配置指定，对于前缀为 *download* 的 URI，向客户端提供响应的速率将在 10 MB 之后被限制为每秒 1 MB。带宽限制是针对每个连接的，因此您可能希望在适用的情况下配合使用连接限制和带宽限制。

详解

通过限制特定连接的带宽，NGINX 能够以您指定的方式在所有客户端上共享其上传带宽，并且只需 limit_rate_after 和 limit_rate 这两个指令就可以做到这一点。limit_rate_after 指令几乎可以在任何上下文中进行设置，包括 http、server、location 以及 location 代码块内的 if。limit_rate 指令的适用上下文与 limit_rate_after 相同，但是它还可以通过一个名为 $limit_rate 的变量进行设置。

limit_rate_after 指令规定，在传输指定数量的数据之前不得限制连接的速率。limit_rate 指令指定了给定上下文中的速率限制，默认单位是每秒字节数，但是您也可以设置为 m（兆字节）或 g（千兆字节）。这两条指令的默认值都是 0，表示不对下载速率进行任何限制。此模块允许您以编程方式更改客户端的速率限制。

第 4 章

大规模可扩展的内容缓存

4.0 简介

缓存能够存储请求的响应结果，以供未来再次使用，进而加速内容的提供。内容缓存可以缓存完整的响应，减少上游服务器的负载，避免了每次都为相同的请求重新运行计算和查询的麻烦。缓存可以提高性能并减少负载，这意味着您可以用更少的资源更快地提供服务。缓存服务器在战略位置上的扩展和分布会对用户体验产生巨大影响。最好将内容托管在靠近消费者的地方，这有助于释放最高性能。您还可以在靠近用户的地方缓存内容，内容交付网络（CDN）就采用了这种模式。NGINX 允许您在放置 NGINX 服务器的任何地方缓存内容，从而有效创建自己的 CDN。借助 NGINX 缓存，您还可以在上游（upstream）发生故障时被动地缓存并提供缓存的响应。缓存功能仅在 http 上下文中可用。

4.1 缓存区

问题

缓存内容并定义缓存的存储位置。

解决方案

使用 proxy_cache_path 指令定义共享内存缓存区和内容的位置：

```
proxy_cache_path /var/nginx/cache
                 keys_zone=CACHE:60m
                 levels=1:2
```

```
                    inactive=3h
                    max_size=20g;
    proxy_cache CACHE;
```

上述缓存定义示例在文件系统 */var/nginx/cache* 中为缓存响应创建了一个目录，以及一个名为 CACHE、大小为 60M 的共享内存空间。此示例设置了目录结构级别，定义了如果缓存响应在 3 小时内未被请求就被释放，同时定义了最大缓存大小为 20GB。proxy_cache 指令通知特定上下文使用缓存区。proxy_cache_path 在 http 上下文中有效，proxy_cache 指令在 http、server 和 location 上下文中有效。

详解

要在 NGINX 中配置缓存，就必须声明要使用的路径和区域。您可以使用 proxy_cache_path 指令创建 NGINX 中的缓存区。proxy_cache_path 指定了存储缓存信息的位置，以及存储活动键和响应元数据的共享内存空间。该指令的可选参数对维护和访问缓存的方式提供了更多控制。levels 参数定义了文件结构的创建方式。该值用冒号分隔，声明了子目录名称的长度，最多支持三个级别。NGINX 缓存以缓存键（一个哈希值）为基础，它将结果存储在提供的文件结构中，使用缓存键作为文件路径并根据 levels 值分解目录。inactive 参数能够控制缓存项在最后一次使用后驻留的时间长度。您也可以使用 max_size 参数配置缓存的大小。其他参数与缓存加载进程有关，这个进程可以将缓存键从磁盘的缓存文件加载到共享内存区。

4.2 缓存锁定

问题

您不希望 NGINX 将当前正在写入缓存的请求代理到上游（upstream）服务器。

解决方案

使用 proxy_cache_lock 指令确保一次只能将一个请求写入缓存，随后的请求将等待响应被写入：

```
proxy_cache_lock on;
proxy_cache_lock_age 10s;
proxy_cache_lock_timeout 3s;
```

详解

proxy_cache_lock 指令指示 NGINX 保存当前正在填充的请求，这些请求最终将成为被缓存的元素。根据 proxy_cache_lock_age 指令的定义，代理请求填充缓存的时间是有限制的，默认为 5 秒，此后才允许另一个请求填充元素。NGINX 还允许将已经等待指定时间（同样默认为 5 秒）的请求传递到代理服务器，该请求不会尝试通过使用 proxy_cache_lock_timeout 指令来填充缓存。您可以将 proxy_cache_lock_age 理解成“你花的时间太长了，我来为你填充缓存”，将 proxy_cache_lock_timeout 理解成“你让我等的时间太长了，你先慢慢填充，我去干点别的”。

4.3 缓存哈希键

问题

控制内容的缓存和检索方式。

解决方案

使用 proxy_cache_key 指令和变量定义缓存的命中或未命中：

```
proxy_cache_key "$host$request_uri $cookie_user";
```

上述缓存哈希键将根据被请求的主机和 URI、定义用户的 cookie 指示 NGINX 缓存页面。这样您就可以缓存动态页面，而且不用提供为其他用户生成的内容。

详解

proxy_cache_key 的默认配置是“$scheme$proxy_host$request_uri”，该配置适用于大多数用例。使用的变量包括 scheme、HTTP 或 HTTPS、proxy_host（发送请求）以及请求 URI。总之，这反映了 NGINX 将请求代理到哪个 URL。您可能会发现，还有许多其他因素定义了每个应用的唯一请求，比如请求参数、请求头、会话标识符等等，您希望为这些因素创建自己的哈希键。[3]

[3] 暴露给 NGINX 的文本或变量的任何组合都可以用来形成一个缓存键。NGINX 提供了这些变量的列表。

选择一个好的哈希键非常重要，其中离不开对应用的了解。为静态内容选择缓存键通常非常简单，只需使用主机名和 URI 便可。但如果是为相当动态的内容（例如仪表盘应用的页面）选择缓存键，就要求对用户与应用的交互方式以及用户体验之间的差异程度有着更深刻的认识。出于安全考虑，您可能不希望在没有完全了解上下文的情况下就将用户的缓存数据展示给其他用户。proxy_cache_key 指令将字符串配置为缓存键的哈希值，该指令可以在 http、server 和 location 代码块中进行设置，从而灵活控制请求的缓存方式。

4.4 绕过缓存

问题

绕过缓存。

解决方案

使用具有非空或非零值的 proxy_cache_bypass 指令。动态执行此操作的一种方法是，在您不希望进行缓存的 location 代码块内，使用一组字符串不能为空或为 0 的变量：

```
proxy_cache_bypass $http_cache_bypass;
```

该配置指示 NGINX，如果名为 cache_bypass 的 HTTP 请求头的值不是 0，则绕过缓存。本例使用请求头作为变量来确定是否应该绕过缓存 —— 客户端需要专门为其请求设置此请求头。

详解

我们有很多使用场景都不要求缓存请求。为此，NGINX 提供了 proxy_cache_bypass 指令。这样，当值设置为非空或非零时，请求将被发送到上游（upstream）服务器，而不是从缓存中拉取。绕过缓存的不同需求和场景将由您的应用的用例来决定。绕过缓存可以简单到只需使用请求头或响应头就能搞定，也会复杂到协同配合多个 map 代码块来解决。

绕过缓存的原因有很多，比如为了故障排除或调试。如果您总是拉取缓存的页面，或者您的缓存键只是针对一个用户标识符的，那么复盘问题可能会很难。因此，具备绕过缓存的能力十分重要。绕过缓存的方法有很多，包括但不限于设置特定的 cookie、请求头或请求参数时等。您还可以通过设置 proxy_cache off;，彻底关闭给定上下文（例如 location 代码块）的缓存功能。

4.5 缓存性能

问题

通过在客户端缓存内容来提高性能。

解决方案

使用客户端的 cache-control HTTP 请求头：

```
location ~* \.(css|js)$ {
    expires 1y;
    add_header Cache-Control "public";
}
```

这个 location 代码块指定客户端可以缓存 CSS 和 JavaScript 文件的内容。expires 指令指示客户端将缓存资源的有效期设置为一年。add_header 指令将 HTTP 响应头 Cache-Control 添加到响应中，值为 public，允许沿途的所有缓存服务器缓存资源。如果将值设置为 private，则只允许客户端缓存资源。

详解

缓存性能有许多影响因素，其中磁盘速度最为重要。NGINX 配置中有很多指令可以帮助提升缓存性能。其中一种方法是设置 HTTP 响应头，让客户端只缓存响应但是不向 NGINX 发送请求，这样只需读取缓存便可。

4.6 NGINX Plus 之缓存清除

问题

使缓存中的对象变得无效。

解决方案

使用 proxy_cache_purge 指令（NGINX Plus 的清除功能）以及非空或零值变量：

```
map $request_method $purge_method {
    PURGE 1;
    default 0;
}
```

```
server {
    # ...
    location / {
        # ...
        proxy_cache_purge $purge_method;
    }
}
```

在此示例中，如果使用 PURGE 方法请求特定对象，那么它的缓存将被清除。以下是使用 curl 清除名为 *main.js* 的文件缓存的示例：

```
$ curl -XPURGE localhost/main.js
```

详解

处理静态文件的一种常见方法是将文件的哈希放在文件名中。当您推出新的代码和内容时，这可以确保您的 CDN 将其识别为新文件（因为 URI 已更改）。但是，对设置了不适合此模型的缓存键的动态内容来说，这种方法并不好用。在每种缓存场景中，您都必须有一种对应的清除缓存的方法。NGINX Plus 的处理方法很简单，只要将 proxy_cache_purge 指令的值设置为非零或非空，与请求匹配的缓存项就被会清除掉。此外还有一种简便方法，那就是为 PURGE 映射请求方法。但是您可能需要配合使用 geo_ip 模块或简单的身份验证方法，以确保不是任何人都能清除您宝贵的缓存项。NGINX 还允许使用 * 来清除与常见 URI 前缀相匹配的缓存项。要使用通配符，您需要使用 purger=on 参数配置 proxy_cache_path 指令。

4.7 缓存切片

问题

需要通过将文件分段来提高缓存效率。

解决方案

使用 NGINX slice 指令及其嵌入式变量将缓存结果分段：

```
proxy_cache_path /tmp/mycache keys_zone=mycache:10m;
server {
    # ...
    proxy_cache mycache;
    slice 1m;
    proxy_cache_key $host$uri$is_args$args$slice_range;
```

```
        proxy_set_header Range $slice_range;
        proxy_http_version 1.1;
        proxy_cache_valid 200 206 1h;

        location / {
            proxy_pass http://origin:80;
        }
    }
```

详解

此配置为服务器定义并启用了一个缓存区。slice 指令指示 NGINX 将响应分成 1MB 的文件段。缓存文件根据 proxy_cache_key 指令进行存储。请注意，该指令使用了一个名为 slice_range 的嵌入式变量。向源端发送请求时也使用了同一变量作为请求头，并且该请求的 HTTP 版本已升级到 HTTP/1.1，因为 1.0 不支持字节范围的请求。200 或 206 响应码的缓存有效期设为 1 个小时，然后定义位置和源端。

Cache Slice 模块是为了交付 HTML5 视频开发的，它使用字节范围的请求将伪流内容传输到浏览器。默认情况下，NGINX 能够从缓存中提供字节范围的请求。如果对未缓存的内容发出了一个字节范围的请求，NGINX 将向源端请求整个文件。当您使用 Cache Slice 模块时，NGINX 仅向源端请求必要的文件段。如果请求范围大于切片大小（包括整个文件），就会触发对每个所需文件段的子请求，并对这些文件段进行缓存处理。当所有文件段都被缓存后，系统将组合响应并将它们发送给客户端，这使得 NGINX 能够更有效地缓存并提供请求的范围内的内容。

Cache Slice 模块只能用于不会进行更改的大文件。NGINX 每接收一个来自源端的文件段，就验证一次 ETag。如果源端的 ETag 发生了变化，那么由于缓存不再有效，NGINX 就会中止这个段的缓存填充。如果内容确实发生了变化，文件也变小了，或者如果源端可以处理缓存填充过程中的负载峰值，那么最好使用下面“其他参考资料”所列博客中描述的 Cache Lock 模块。默认情况下不创建该模块，您需要在构建 NGINX 时使用 --with-http_slice_module 配置来启用。

其他参考资料

“使用 NGINX 和 NGINX Plus 智能高效地进行字节范围的缓存”

第 5 章
可编程性和自动化

5.0 简介

可编程性是指通过编程进行交互的能力。NGINX Plus 的 API 可以帮助实现这一点，即通过 HTTP 接口与 NGINX Plus 的配置和行为进行交互。该 API 支持通过 HTTP 请求添加或删除上游（upstream）服务器，从而实现对 NGINX Plus 的重新配置。NGINX Plus 中的键值（key-value）存储功能支持其他级别的动态配置 —— 您可以利用 HTTP 调用注入 NGINX Plus 动态路由或控制流量所需的信息。本章将探讨 NGINX Plus API 以及通过该 API 暴露的键值存储模块。

配置管理工具能够实现服务器的自动化安装和配置，这在云时代是非常宝贵的实用程序。通过使用这些工具，大型 Web 应用的工程师不必再手动配置服务器，只需一次性写好配置和代码，就能够以可重复、可测试和模块化的方式构建出许多具有相同配置的服务器。本章介绍了当前市面上最受欢迎的一些配置管理工具，并讨论了如何使用它们安装 NGINX 以及如何将基本配置模板化。其中有一些很基础的例子，但却很好地演示了如何在每个平台上启用 NGINX 服务器。

5.1 NGINX Plus API

问题

您有一个动态的环境，需要随时重新配置 NGINX Plus。

解决方案

配置 NGINX Plus API，以便通过 API 调用启用和删除服务器：

```
upstream backend {
    zone http_backend 64k;
}
server {
    # ...
    location /api {
        api [write=on];
        # 限制访问 API 的指令
        # 见第 7 章
    }

    location = /dashboard.html {
        root /usr/share/nginx/html;
    }
}
```

NGINX Plus 的这一配置创建了一个带有共享内存区的上游（upstream）服务器，允许启用 /api location 代码块中的 API，并为 NGINX Plus 仪表盘提供了一个 location。

您可以使用 API 添加上线的服务器：

```
$ curl -X POST -d '{"server":"172.17.0.3"}' \
    'http://nginx.local/api/3/http/upstreams/backend/servers/'
{
    "id":0,
    "server":"172.17.0.3:80",
    "weight":1,
    "max_conns":0,
    "max_fails":1,
    "fail_timeout":"10s",
    "slow_start":"0s",
    "route":"",
    "backup":false,
    "down":false
}
```

此示例中的 curl 调用请求 NGINX Plus 将新服务器添加到后端上游配置中。HTTP 采用了 POST 方法，将 JSON 对象作为正文传输，并返回 JSON 响应。JSON 响应显示了服务器对象配置 —— 请注意，它生成了一个新的 id，其他配置均设置为默认值。

NGINX Plus API 是 RESTful，因此请求 URI 中带有参数。

URI 的格式如下：

```
/api/{version}/http/upstreams/{httpUpstreamName}/servers/
```

您可以利用 NGINX Plus API 列出上游池中的服务器：

```
$ curl 'http://nginx.local/api/3/http/upstreams/backend/servers/'
[
    {
        "id":0,
        "server":"172.17.0.3:80",
        "weight":1,
        "max_conns":0,
        "max_fails":1,
        "fail_timeout":"10s",
        "slow_start":"0s",
        "route":"",
        "backup":false,
        "down":false
    }
]
```

此示例中的 curl 调用请求 NGINX Plus 列出 backend 上游池中的所有服务器。目前我们只有一台在以前的 API curl 调用中添加的服务器。该请求将返回一个上游服务器对象，它包含服务器所有可配置的选项。

如以下代码所示，使用 NGINX Plus API 清空上游服务器的连接，为将它从上游池中删除做好准备。有关连接清空的详细信息，请参见实操指南 2.8。

```
$ curl -X PATCH -d '{"drain":true}' \
    'http://nginx.local/api/3/http/upstreams/backend/servers/0'
{
    "id":0,
    "server":"172.17.0.3:80",
    "weight":1,
    "max_conns":0,
    "max_fails":1,
    "fail_timeout":
    "10s","slow_start":
    "0s",
    "route":"",
    "backup":false,
    "down":false,
    "drain":true
}
```

这个 curl 调用指定了请求方法为 PATCH，传输 JSON 正文以指示清空服务器的连接，并指定将服务器 ID 附加到 URI。通过列出上一个 curl 命令指示添加的上游服务器，我们找到了服务器 ID。

NGINX Plus 将开始执行清空连接的进程，它用时比较长，相当于应用的会话长度。要检查正在实施连接清空的服务器正在提供多少个活动连接，请使用以下调用并查找相关服务器的 active 属性：

```
$ curl 'http://nginx.local/api/3/http/upstreams/backend'
{
    "zone" : "http_backend",
    "keepalive" : 0,
    "peers" : [
        {
            "backup" : false,
            "id" : 0,
            "unavail" : 0,
            "name" : "172.17.0.3",
            "requests" : 0,
            "received" : 0,
            "state" : "draining",
            "server" : "172.17.0.3:80",
            "active" : 0,
            "weight" : 1,
            "fails" : 0,
            "sent" : 0,
            "responses" : {
                "4xx" : 0,
                "total" : 0,
                "3xx" : 0,
                "5xx" : 0,
                "2xx" : 0,
                "1xx" : 0
            },
            "health_checks" : {
                "checks" : 0,
                "unhealthy" : 0,
                "fails" : 0
            },
            "downtime" : 0
        }
    ],
    "zombies" : 0
}
```

等所有连接都清空后，使用 NGINX Plus API 将服务器从上游池中彻底删除：

```
$ curl -X DELETE \
    'http://nginx.local/api/3/http/upstreams/backend/servers/0'
[]
```

curl 命令使用 DELETE 方法向更新服务器状态时使用的同一 URI 发出了请求。DELETE 方法指示 NGINX 删除服务器。此 API 调用返回仍留在池中的所有服务器及

其 ID。由于一开始是一个空池，而后只通过 API 添加了一台服务器，清空连接后又将其删除，现在我们还是只剩下一个空池。

详解

NGINX Plus 独有的 API 允许动态应用服务器随时自行添加到 NGINX 配置中或者从 NGINX 配置中删除。服务器上线后就会被注册到池中，NGINX 也将开始向新添加的服务器发送负载。当需要删除服务器时，服务器可以请求 NGINX Plus 清空其连接，然后在被关闭之前自行从上游池中删除。这样基础架构无需人工干预，便可以在一定程度上自动化地调整规模。

其他参考资料

NGINX Plus REST API 文档

5.2 使用 NGINX Plus 的键值（Key-Value）存储功能

问题

您需要 NGINX Plus 根据应用的输入做出动态的流量管理决策。

解决方案

本节将以动态拦截列表为例演示流量管理决策。

设置集群感知型键值存储和 API，然后添加键和值：

```
keyval_zone zone=blocklist:1M;
keyval $remote_addr $blocked zone=blocklist;

server {
    # ...
    location / {
        if ($blocked) {
            return 403 'Forbidden';
        }
        return 200 'OK';
    }
}
server {
```

```
    # ...
    # 限制访问 API 的指令
    # 见第 6 章
    location /api {
        api write=on;
    }
}
```

NGINX Plus 的这一配置使用 keyval_zone 目录构建了一个名为 blocklist 的键值存储共享内存区，并将内存限制设置为 1MB。然后 keyval 指令映射键的值，从而将第一个参数 $remote_addr 匹配到区中的新变量 $blocked。这一新变量用来确定 NGINX Plus 是应该服务请求还是返回 403 Forbidden 代码。

使用此配置启动 NGINX Plus 服务器后，您可以向本地计算机发送 curl 命令，服务器可能会返回 200 OK 响应：

```
$ curl 'http://127.0.0.1/'
OK
```

现在将本地机器的 IP 地址添加到值为 1 的键值存储中：

```
$ curl -X POST -d '{"127.0.0.1":"1"}' \
    'http://127.0.0.1/api/3/http/keyvals/blocklist'
```

这个 curl 命令提交了带 JSON 对象的 HTTP POST 请求，其中包含一个待提交到 blocklist 共享内存区的键值对象。键值存储 API URI 的格式如下：

```
/api/{version}/http/keyvals/{httpKeyvalZoneName}
```

本地机器的 IP 地址现已添加到值为 1 的 blocklist 键值区中。在接下来的请求中，NGINX Plus 会在键值区中查询 $remote_addr，查找条目并将值映射到变量 $blocked。该变量将在 if 语句中进行计算。变量赋值后，if 计算结果为 True，NGINX Plus 返回 403 Forbidden 状态码：

```
$ curl 'http://127.0.0.1/'
Forbidden
```

您可以通过使用 PATCH 方法发送请求来更新或删除键：

```
$ curl -X PATCH -d '{"127.0.0.1":null}' \
    'http://127.0.0.1/api/3/http/keyvals/blocklist'
```

如果值为 null，NGINX Plus 将删除键，并且请求将再次返回 200 OK。

详解

键值存储是 NGINX Plus 独有的功能，允许应用将信息注入到 NGINX Plus 中。上述示例中的 $remote_addr 变量用于创建动态拦截列表。您可以使用 NGINX Plus 可能

用作变量的任何键（例如会话 cookie）填充键值存储，并为 NGINX Plus 提供一个外部值。在 NGINX Plus R16 中，键值存储具备集群感知能力，这意味着您只需为一台 NGINX Plus 服务器提供键值更新便可，这样所有服务器都会收到信息。

在 NGINX Plus R19 中，键值存储启用了 type 参数，支持为特定类型的键添加索引。默认情况下，键的类型值为 string，您也可以选择使用 ip 和 prefix。string 类型不构建索引，并且所有键的请求都必须是确切的匹配，而 prefix 允许部分键匹配（只要键的前缀匹配即可）。ip 类型允许使用 CIDR notation。在我们的示例中，如果已经指定 type=ip 为键值区的参数，那么我们也可以提供要拦截的整个 CIDR 范围，例如使用 192.168.0.0/16 拦截 RFC 1918 私有网络地址范围，或者使用 127.0.0.1/32 拦截本地主机，这种方法的效果与示例中呈现的一样。

其他参考资料

“使用 NGINX Plus 键值存储进行动态带宽限制”

5.3 在 NGINX 中使用 NJS 模块暴露 JavaScript 功能

问题

需要 NGINX 执行对请求或响应的自定义逻辑。

解决方案

通过为 NGINX 安装 NJS 模块启用 JavaScript。以下软件包安装步骤假设您已经为您的 Linux 发行版添加了官方 NGINX 仓库（如第 1 章所示）。

Debian/Ubuntu：

```
apt-get install nginx-module-njs
```

Debian/Ubuntu 和 NGINX Plus：

```
apt-get install nginx-plus-module-njs
```

RHEL/CentOS：

```
yum install nginx-module-njs
```

RHEL/CentOS 和 NGINX Plus：

```
yum install nginx-plus-module-njs
```

如果您的 NGINX 配置中没有 JavaScript 文件目录，请创建一个：

```
mkdir -p /etc/nginx/njs
```

创建一个名为 */etc/nginx/njs/jwt.js* 的 JavaScript 文件，且该文件应包含以下内容：

```
function jwt(data) {
    var parts = data.split('.').slice(0,2)
        .map(v=>Buffer.from(v, 'base64url').toString())
        .map(JSON.parse);
    return { headers:parts[0], payload: parts[1] };
}
function jwt_payload_subject(r) {
    return jwt(r.headersIn.Authorization.slice(7)).payload.sub;
}
function jwt_payload_issuer(r) {
    return jwt(r.headersIn.Authorization.slice(7)).payload.iss;
}
export default {jwt_payload_subject, jwt_payload_issuer}
```

提供的 JavaScript 示例定义了一个解码 JSON Web Token（JWT）的函数。此外它还定义了其他两个函数，它们使用 JWT 解码器返回 JWT 中的特定键。导出这些函数，以供 NGINX 使用。这些函数将返回 JWT 中的常见键：subject 和 issuer。代码中的 .slice（7）部分负责删除 Authorization 请求头值的前 7 个字符。JWT 出现以后，类型值变成 Bearer。Bearer 是 6 个字符，我们还需要删除空格分隔符，这也我们分割前 7 个字符的原因。有一些身份验证服务未提供类型（比如 AWS Cognito），因为此类服务会更改分割数或完全删除它，以便 jwt 函数仅接收令牌值。

在 NGINX 核心配置中加载 NJS 模块。在 http 代码块中导入并使用 JavaScript：

```
load_module /etc/nginx/modules/ngx_http_js_module.so;

http {
    js_path "/etc/nginx/njs/";
    js_import main from jwt.js;
    js_set $jwt_payload_subject main.jwt_payload_subject;
    js_set $jwt_payload_issuer main.jwt_payload_issuer;
    ...
}
```

提供的 NGINX 配置动态加载了 NJS 模块，并导入了我们先前定义的 JavaScript 文件。使用 NGINX 指令将 NGINX 变量设置为 JavaScript 函数的返回值。

使用变量证明 JavaScript 的逻辑。定义返回 JavaScript 设置的变量的服务器：

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name _;
    location / {
        return 200 "$jwt_payload_subject $jwt_payload_issuer";
    }
}
```

提供的配置将生成一台服务器，以返回客户端通过 Authorization 请求头提供的 subject 和 issuer 值。这些值由定义的 JavaScript 代码解码。

为了验证代码的有效性，应使用给定的 JWT 向服务器发送请求。以下是 JWT 的 JSON 格式，您可以用它来验证代码的有效性：

```
{
    "iss": "nginx",
    "sub": "alice",
    "foo": 123,
    "bar": "qq",
    "zyx": false
}
```

用给定的 JWT 向服务器发送请求，以验证 JavaScript 代码是否运行正常并返回了正确的值：

```
curl 'http://localhost/' -H \
"Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1\
NiIsImV4cCI6MTU4NDcyMzA4NX0.eyJpc3MiOiJuZ2lueCIsInN1YiI6Im\
FsaWNlIiwiZm9vIjoxMjMsImJhciI6InFxIiwie\
nl4IjpmYWxzZX0.Kftl23Rvv9dIso1RuZ8uHaJ83BkKmMtTwch09rJtwgk"

alice nginx
```

详解

NGINX 的 NJS 模块可以在处理请求和响应的过程中暴露 JavaScript 标准功能。该模块允许您将业务逻辑嵌入到代理层中。我们之所以选择 JavaScript，是因为它的应用非常广泛。

NJS 模块支持在请求进入 NGINX 以及 NGINX 返回响应时注入逻辑。您可以通过解码 JSON Web Token 来验证和操作穿过代理的请求（如本节所示）。NJS 模块还能够通过 JavaScript 逻辑传输响应数据流，进而操作来自上游（upstream）服务的响应。如“其他参考资料”所示，NJS 还可以赋予 stream 服务应用层感知能力。

其他参考资料

njs 脚本语言文档
NGINX Plus njs 模块的安装

5.4 使用通用编程语言扩展 NGINX

问题

您需要 NGINX 使用通用编程语言执行一些自定义扩展。

解决方案

在用 C 语言编写自定义 NGINX 模块之前，请先评估您的用例是否适合使用其他编程语言模块。C 编程语言是一种非常强大且高效的语言，但是也有许多其他语言模块能够支持所需的自定义。NGINX 引入了 NGINX JavaScript（NJS），只需启用一个模块就可以将 JavaScript 的强大功能暴露到 NGINX 配置中。您也可以使用 Lua 和 Perl 模块。

使用这些语言模块时，您要么需要导入一个包含代码的文件，要么直接在配置中定义代码块。

要使用 Lua，请安装 Lua 模块和以下 NGINX 配置，以定义一个内联的 Lua 脚本：

```
load_module modules/ndk_http_module.so;
load_module modules/ngx_http_lua_module.so;

events {}

http {
    server {
        listen 8080;
        location / {
            default_type text/html;
            content_by_lua_block {
                ngx.say("hello, world")
            }
        }
    }
}
```

Lua 模块通过 ngx 模块定义的对象提供自己的 NGINX API。与 NJS 中的 request 对象一样，ngx 对象也有描述请求和操作响应的属性和方法。

安装了 Perl 模块后，此示例将使用 Perl 从运行时环境设置 NGINX 变量：

```
load_module modules/ngx_http_perl_module.so;

events {}

http {
    perl_set $app_endpoint 'sub { return $ENV{"APP_DNS_ENDPOINT"}; }';
    server {
        listen 8080;
        location / {
                proxy_pass http://$app_endpoint
            }
        }
    }
}
```

上述示例表明，这些语言模块不仅仅是返回响应，而且还会暴露更多功能。perl_set 指令将一个 NGINX 变量设置为从 Perl 脚本返回的数据。这个示例比较有限，只是返回了用作请求代理端点的系统环境变量。

详解

NGINX 具备出色的可扩展性，这为启用更多功能开创了更多可能。NGINX 可通过使用 C 语言模块自定义代码来扩展，您可以从一开始构建时就把这些模块编译到 NGINX 中，也可以在配置中动态加载。目前还有一些模块可以暴露 JavaScript（njs）、Lua 和 Perl 的功能与语法。在多数情况下这些预先存在的模块是够用的 —— 除非将 NGINX 的自定义功能分配出去。现在开源社区中有很多为这些模块构建的脚本。

该解决方案演示了 NGINX 和 NGINX Plus 中可用的 Lua 和 Perl 脚本语言的基本用法。无论是寻求响应、设置变量、发送子请求还是定义复杂的重写，这些 NGINX 模块都能提供相应的支持。

其他参考资料

NGINX Plus Lua 模块的安装
NGINX Plus Perl 模块的安装
NGINX Lua 模块文档
NGINX Perl 模块文档

5.5 使用 Chef 安装

问题

使用 Chef 安装和配置 NGINX，以将 NGINX 配置作为代码管理，并与其余的 Chef 配置保持一致。

解决方案

从 Chef Supermarket 安装由 Sous Chefs 维护的 NGINX cookbook：

```
knife supermarket install nginx
```

此 cookbook 以资源为基础，这意味着它为您提供了可以供您在自己的 cookbook 中使用的 Chef 资源。为您的 NGINX 用例创建 cookbook。该 cookbook 将包含从 Supermarket 安装的 nginx cookbook 依赖包。依赖包安装成功后，您可以使用提供的资源。为安装 NGINX 创建 recipe：

```
nginx_install 'nginx' do
    source 'repo'
end
```

将 source 设置为 repo 后，NGINX 将从 NGINX Inc. 维护的仓库进行安装，该仓库提供了最新的版本。

使用 recipe 中的 nginx_config 资源覆盖核心 NGINX 配置：

```
nginx_config 'nginx' do
    default_site_enabled true
    keepalive_timeout 65
    worker_processes 'auto'
    action :create
    notifies :reload, 'nginx_service[nginx]', :delayed
end
```

使用 recipe 中的 `nginx_site` 资源配置 NGINX server 代码块：

```
nginx_site 'test_site' do
    mode '0644'

    variables(
        'server' => {
            'listen' => [ '*:80' ],
            'server_name' => [ 'test.example.com' ],
```

```
            'access_log' => '/var/log/nginx/test_site.access.log',
            'locations' => {
                '/' => {
                    'root' => '/var/www/nginx-default',
                    'index' => 'index.html index.htm',
                },
            },
        }
    )

    action :create
    notifies :reload, 'nginx_service[nginx]', :delayed
end
```

详解

Chef 是一款使用 Ruby 编写的配置管理工具，它可以在客户端/服务器关系或独立配置中运行。Chef 拥有一个非常庞大的社区（称为 Supermarket），其中有许多公开可用的 cookbook。用户可以使用命令行实用程序 Knife 从 Supermarket 安装和维护公共 cookbook。Chef 的功能十分强悍，我们所展示的只是冰山一角。

Supermarket 中的公共 NGINX cookbook 非常灵活，提供了从软件包管理器或源代码轻松安装 NGINX 的选项，能够编译和安装多种不同的模块并将基础配置模板化。本节演示了如何从 NGINX Inc. 维护的仓库安装 NGINX，并简单举例说明了如何配置 server 代码块以托管 HTML 文件。您可以在 nginx_site 资源中添加自己的模板，以进一步控制您的 NGINX 配置。

其他参考资料

Chef 文档
Chef Supermarket 之 NGINX cookbook

5.6 使用 Ansible 安装

问题

使用 Ansible 安装和配置 NGINX，以将 NGINX 配置作为代码管理，并与其余的 Ansible 配置保持一致。

解决方案

从 Ansible Galaxy 安装 Ansible NGINX collection：

```
ansible-galaxy collection install nginxinc.nginx_core
```

创建一个 playbook，使用 collection 和 nginx role 安装 NGINX：

```
---
- hosts: all
  collections:
    - nginxinc.nginx_core
  tasks:
    - name: Install NGINX
      include_role:
        name: nginx
```

要配置 NGINX，请添加 task 并使用 nginx_config role，同时根据自己的用例覆盖默认模板的变量：

```
- name: Configure NGINX
  include_role:
    name: nginx_config
  vars:
    nginx_config_http_template_enable: true
    nginx_config_http_template:
      - template_file: http/default.conf.j2
        deployment_location: /etc/nginx/conf.d/default.conf
        config:
          servers:
            - core:
                listen:
                  - port: 80
                server_name: localhost
              log:
                access:
                  - path: /var/log/nginx/access.log
                    format: main
              sub_filter:
                sub_filters:
                  - string: server_hostname
                    replacement: $hostname
                once: false
              locations:
                - location: /
                  core:
                    root: /usr/share/nginx/html
                    index: index.html

    nginx_config_html_demo_template_enable: true
    nginx_config_html_demo_template:
      - template_file: www/index.html.j2
        html_file_name: index.html
        html_file_location: /usr/share/nginx/html
        web_server_name: Ansible NGINX collection
```

详解

Ansible 使用 Python 编写而成，是一款应用广泛、功能强大的配置管理工具。task 的配置位于 YAML 中，您可以使用 Jinja2 模板语言将文件模板化。Ansible 提供了订阅模式的服务器：Ansible Tower。但它通常是在本地机器中使用，您也可以直接将服务器构建到客户端中或者独立使用服务器。Ansible 会将 SSH 批量导入到服务器中并运行配置。与其他配置管理工具一样，Ansible 也有一个大型社区 —— Ansible Galaxy，里面有很多公开可用的 role，其中不乏一些复杂巧妙的 role，您可以将它们用在自己的 playbook 中。

该解决方案使用了由 NGINX Inc. 维护的公共 role 集合来安装 NGINX 并生成样本配置。此示例中使用的配置生成了一个 NGINX demo HTML 文件模板，并把它放在了 */usr/share/nginx/html/index.html* 中。此外它还将 NGINX 配置文件模板化，以便生成侦听 localhost:80 的 server 代码块，并配置一个 location 代码块来提供演示文件。示例中提供的 NGINX 配置模板非常全面，但是 nginx_config role 允许您提供自己的模板，以便在进行全面控制的同时利用预先构建和维护的 Ansible 配置。

其他参考资料

NGINX 提供的 Ansible collection
Ansible 文档

5.7 使用 Consul 模板自动进行配置

问题

使用 Consul 自动化进行 NGINX 配置，以响应环境的变化。

解决方案

使用 consul-template 守护进程和模板文件将您选择的 NGINX 配置文件模板化：

```
upstream backend { {{range service "app.backend"}}
    server {{.Address}};{{end}}
}
```

此示例是一个 Consul 模板文件，能够帮助生成上游（upstream）配置代码块的模板。该模板将在 Consul 中标记为 app.backend 的节点循环，并为 Consul 中的每个节点生成一个带有其 IP 地址的服务器指令。

consul-template 守护进程通过命令行运行，它可以在每次修改配置文件的模板时重载 NGINX：

```
# consul-template -consul-addr consul.example.internal -template \
./upstream.template:/etc/nginx/conf.d/upstream.conf:"nginx -s reload"
```

此命令指示 consul-template 守护进程连接到 consul.example.internal 的 Consul 集群，使用当前工作目录中名为 *upstream.template* 的文件将文件模板化，同时将生成的内容输出到 */etc/nginx/conf.d/upstream.conf*，然后在每次修改模板化文件时重载 NGINX。-template 标记使用由模板文件、输出位置以及执行模板化流程后运行的命令组成的字符串，这三个变量用冒号隔开。如果正在运行的命令有空格，请用双引号把它括起来。-consul 标记负责告知守护进程 Consul 集群要要连接到什么。

详解

Consul 是一款功能强大的服务发现和配置存储工具，它通过一个类似目录的结构存储节点和键值（key-value）对信息，并支持 RESTful API 交互。Consul 还在每个客户端上提供了 DNS 接口，支持查找连接到集群的节点的域名。consul-template 守护进程也会用到 Consul 集群，该工具通过将文件模板化来响应 Consul 节点、服务或键值对的变化。这使得 Consul 成为一个非常强有力的 NGINX 自动化选择。consul-template 还可以指示守护进程在修改模板后运行命令。这样您就可以重新加载 NGINX 配置，并让您的 NGINX 配置与环境一起激活。借助 Consul 和 consul-template，您能够随着环境的变化动态调整 NGINX 配置。基础架构、配置和应用信息会进行集中存储，consul-template 能够在必要时根据事件进行订阅和重新模板化。有了这项技术，NGINX 就可以实现动态重新配置，从而应对服务器、服务和应用版本等项目的添加和删除。

其他参考资料

“使用 NGINX Plus 的服务发现集成功能实现负载均衡”
“使用 NGINX 和 Consul 模板实现负载均衡”
Consul 主页
“使用 Consul 模板进行服务配置”
Consul 模板 GitHub

第 6 章
身份验证

6.0 简介

NGINX 能够对客户端进行身份验证。使用 NGINX 对客户端请求进行身份验证可以减轻服务器的工作负载，并能够阻止未经身份验证的请求到达应用服务器。NGINX 开源版模块包括基本身份验证和身份验证子请求。NGINX Plus 专有的 JSON Web Tokens（JWT）验证模块可与使用身份验证标准 OpenID Connect 的第三方身份验证提供商集成。

6.1 HTTP 基本身份验证

问题

需要通过 HTTP 基本身份验证保护应用或内容。

解决方案

生成以下格式的文件，其中的密码使用某个受支持的格式进行了加密或哈希处理：

```
# comment
name1:password1
name2:password2:comment
name3:password3
```

第一个字段是用户名，第二个字段是密码，冒号是分隔符。第三个字段为可选项，您可以使用该字段对每个用户进行评论。NGINX 能理解几种不同的密码格式，其中一个是使用 C 函数 crypt() 加密的密码。该函数通过 openssl passwd 命令暴露在命令行中。安装 openssl 后，您可以使用以下命令创建加密的密码字符串：

```
$ openssl passwd MyPassword1234
```

输出结果将是一个字符串，可供 NGINX 在密码文件中使用。

在 NGINX 配置中使用 auth_basic 和 auth_basic_user_file 指令，实现基本身份验证：

```
location / {
    auth_basic         "Private site";
    auth_basic_user_file conf.d/passwd;
}
```

您可以在 http、server 或 location 上下文中使用 auth_basic 指令。auth_basic 指令带一个字符串参数，如有未经授权的用户访问，该参数将显示在基本身份验证弹窗中。auth_basic_user_file 指定了用户文件的路径。

如要测试配置，您可以使用带 -u 或 --user 标志的 curl 来构建请求的 Authorization 请求头：

```
$ curl --user myuser:MyPassword1234 https://localhost
```

详解

您可以通过几种方式生成具有不同格式和安全等级的基本身份验证密码。Apache 的 htpasswd 命令也可以生成密码。openssl 和 htpasswd 都可以使用 apr1 算法生成密码，并且 NGINX 也能理解这种格式的密码。该密码也可以是轻型目录访问协议（LDAP）和 Dovecot 使用的 Salted SHA-1 格式。NGINX 支持其他更多格式和哈希算法；然而，许多算法容易遭到暴力破解攻击，因此人们认为这些算法不安全。

您可以使用基本身份验证来保护整个 NGINX 主机的上下文、特定的虚拟服务器甚至只是特定的 location 代码块。基本身份验证不会取代 Web 应用的用户身份验证，但可以保护私人信息的安全。实际上，基本身份验证是由服务器返回 401 Unauthorized HTTP 代码和响应头 WWW-Authenticate 完成的。该响应头的值为 Basic realm=*"your string"*。收到此响应后，浏览器将提示输入用户名和密码。用户名和密码用冒号连接和分隔，然后用 base64 编码，最后在名为 Authorization 的请求头中发送。Authorization 请求头将指定一个 Basic 和 user:password 编码字符串。服务器对请求头进行解码，并根据提供的 auth_basic_user_file 进行验证。由于用户名和密码字符串仅通过 base64 编码，我们建议在基本身份验证中使用 HTTPS。

6.2 身份验证子请求

问题

希望通过第三方身份验证系对请求进行身份验证。

解决方案

使用 http_auth_request_module 请求访问身份验证服务，在响应请求之前验证身份：

```
location /private/ {
    auth_request     /auth;
    auth_request_set $auth_status $upstream_status;
}

location = /auth {
    internal;
    proxy_pass              http://auth-server;
    proxy_pass_request_body off;
    proxy_set_header        Content-Length "";
    proxy_set_header        X-Original-URI $request_uri;
}
```

auth_request 指令带一个 URI 参数，该参数必须是本地内部位置。auth_request_set 指令允许您设置身份验证子请求的变量。

详解

http_auth_request_module 可验证 NGINX 服务器处理的每个请求的身份。该模块将使用子请求来确定是否授予请求访问权。子请求是指 NGINX 将请求传递给内部替代位置，并在将请求路由到目的地之前观察它的响应。auth 位置将原始请求（包括正文和请求头）传递到身份验证服务器。子请求的 HTTP 状态码决定是否授予请求访问权。如果子请求返回 HTTP 200 状态码，则表示身份验证成功，请求已完成。如果子请求返回 HTTP 401 或 403，则原始请求也将返回 HTTP 401 或 403。

如果您的身份验证服务未请求请求正文，则可以使用 proxy_pass_request_body 指令删除请求正文，如上所示。这种做法可减少请求的大小和时间。由于响应正文被丢弃，Content-Length 请求头必须设置为空字符串。如果身份验证服务需要知道请求访问的 URI，则需要将该值放入身份验证服务检查和验证的自定义请求头中。如果您确实希望

将子请求中的某些内容保留给身份验证服务（例如响应头或其他信息），您可以使用 auth_request_set 指令从响应数据中创建新的变量。

6.3 使用 NGINX Plus 验证 JWT

问题

需要在使用 NGINX Plus 处理请求之前验证 JWT。

解决方案

使用 NGINX Plus 的 HTTP JWT 身份验证模块来验证令牌签名，并将 JWT 声明和请求头作为 NGINX 变量嵌入：

```
location /api/ {
    auth_jwt         "api";
    auth_jwt_key_file conf/keys.json;
}
```

此配置可以验证该位置的 JWT。auth_jwt 指令传递一个字符串，该字符串被用作身份验证领域。auth_jwt 配置带一个保存 JWT 的变量的可选令牌参数。默认情况下，根据 JWT 标准使用 Authentication 请求头。auth_jwt 指令还可用于从继承的配置中消除所需的 JWT 身份验证的影响。要关闭（off）身份验证，只需将关键字传递给 auth_jwt 指令。要取消继承的身份验证要求，只需将 off 关键字传递到 auth_jwt 指令。auth_jwt_key_file 带一个参数。该参数是采用标准 JSON Web Key（JWK）格式的密钥文件的路径。

详解

NGINX Plus 支持验证 JSON Web 签名类型的令牌，而不是将整个令牌进行加密的 JSON Web 加密类型。NGINX Plus 支持验证使用 HS256、RS256 和 ES256 算法签名的签名。使用 NGINX Plus 验证令牌可以节省向身份验证服务发出子请求所需的时间和资源。NGINX Plus 可破解 JWT 请求头和有效载荷，并将标准请求头和声明捕获到嵌入式变量中，供您使用。auth_jwt 指令可在 http、server、location 及 limit_except 上下文中使用。

其他参考资料

RFC JSON Web Signature 标准文档
RFC JSON Web 算法标准文档
RFC JSON Web Token 标准文档
NGINX Plus JWT 身份验证
“借助 JWT 和 NGINX Plus 验证 API 客户端身份”

6.4 创建 JSON Web Key

问题

需要使用 JSON Web Key（JWK）才能使用 NGINX Plus。

解决方案

NGINX Plus 使用 RFC 标准中指定的 JWK 格式。该标准允许在 JWK 文件中包含一个关键对象数组。

以下是该密钥文件的示例：

```
{"keys":
  [
    {
        "kty":"oct",
        "kid":"0001",
        "k":"OctetSequenceKeyValue"
    },
    {
        "kty":"EC",
        "kid":"0002"
        "crv":"P-256",
        "x": "XCoordinateValue",
        "y": "YCoordinateValue",
        "d": "PrivateExponent",
        "use": "sig"
    },
    {
        "kty":"RSA",
        "kid":"0003"
        "n": "Modulus",
        "e": "Exponent",
        "d": "PrivateExponent"
    }
  ]
}
```

所示的 JWK 文件演示了 RFC 标准中指出的三类初始密钥。这些密钥的格式也是 RFC 标准的一部分。kty 属性是密钥类型。该文件显示了三种密钥类型：Octet Sequence（oct）、EllipticCurve（EC）和 RSA 类型。kid 属性是密钥 ID。这些密钥的标准中指定了其他属性。更多信息请查看这些标准的 RFC 文档。

详解

有许多提供不同语言的库可以生成 JWK。建议创建一个密钥服务，作为 JWK 的中央权限，以定期创建和轮换您的 JWK。为了增强安全性，建议让 JWK 与 SSL/TLS 证书一样安全。使用适当的用户和组权限保护密钥文件。最好将它们保存在主机的内存中。您可以通过创建类似 ramfs 的内存文件系统来实现。定期轮换密钥也很重要；您可以选择创建一个密钥服务来创建公钥和私钥，并通过 API 将它们提供给应用和 NGINX。

其他参考资料

RFC JSON Web Key 标准文档

6.5 使用 NGINX Plus 验证 JSON Web Token

问题

使用 NGINX Plus 验证 JSON Web Token。

解决方案

使用 NGINX Plus 自带的 JWT 模块保护位置或服务器，并指示 airt_jwt 指令使用 $cookie_auth_token 作为要验证的令牌：

```
location /private/ {
    auth_jwt "Google Oauth" token=$cookie_auth_token;
    auth_jwt_key_file /etc/nginx/google_certs.jwk;
}
```

此配置指示 NGINX Plus 通过 JWT 验证保护 */private/* URI 路径。Google OAuth 2.0 OpenID Connect 使用 cookie auth_token 而不是默认的 bearer 令牌。因此，您必须指示 NGINX 在此 cookie 中查找令牌，而不是在 NGINX Plus 的默认位置中查找。我们将在实操指南 6.6 中介绍如何将 auth_jwt_key_file 位置设置为任意路径。

详解

此配置说明了如何使用 NGINX Plus 验证 Google OAuth 2.0 OpenID Connect JWT。NGINX Plus 的 HTTP JWT 身份验证模块支持验证符合 RFC JSON Web Signature 规范的 JWT，允许任何使用 JWT 的 SSO 授权立即在 NGINX Plus 层中进行验证。OpenID 1.0 协议层位于 OAuth 的上面。OpenID 2.0 身份验证协议添加了身份，支持使用 JWT 来证明发送请求的用户的身份。NGINX Plus 可通过令牌的签名验证令牌自签名以来是否经过修改。这允许 Google 使用一种异步签名方法，可在保护其私人 JWK 的同时分发公共 JWK。

其他参考资料

“借助 JWT 和 NGINX Plus 验证 API 客户端身份”

6.6 使用 NGINX Plus 自动获取和缓存 JSON Web Key Set

问题

希望 NGINX Plus 自动请求提供商的 JSON Web Key Set（JWKS）并进行缓存。

解决方案

利用缓存区和 auth_jwt_key_request 指令自动更新密钥：

```
proxy_cache_path /data/nginx/cache levels=1 keys_zone=foo:10m;

server {
    # ...

    location / {
        auth_jwt             "closed site";
        auth_jwt_key_request /jwks_uri;
    }

    location = /jwks_uri {
        internal;
        proxy_cache foo;
        proxy_pass  https://idp.example.com/keys;
    }
}
```

在此示例中，auth_jwt_key_request 指令指示 NGINX Plus 从内部子请求中检索 JWK。该子请求被定向到 /jwks_uri，后者将请求代理给身份提供商。默认请求缓存时间为 10 分钟，以限制开销。

详解

NGINX Plus R17 中引入了 auth_jwt_key_request 指令。此功能支持 NGINX Plus 服务器在发出请求时动态更新其 JWK。使用子请求方法来获取 JWK，这意味着指令指向的位置必须是 NGINX Plus 服务器本地的位置。在上面的示例中，子请求的位置被锁定，以确保仅响应内部的 NGINX Plus 请求。还可使用缓存来确保仅在必要时发出 JWK 检索请求，并且不会导致身份提供商过载。auth_jwt_key_request 指令在 http、server、location 和 limit_except 上下文中有效。

其他参考资料

“借助 JWT 和 NGINX Plus 验证 API 客户端身份”
NGINX Plus “借助 JSON Web Key Set 缓存加快 JWT 验证”

6.7 使用 NGINX Plus 通过现有的 OpenID Connect SSO 验证用户身份

问题

希望将 NGINX Plus 与 OpenID Connect（OIDC）身份提供商集成在一起。

解决方案

该解决方案由许多配置要素和一些 NGINX JavaScript 代码组成。身份提供商（IdP）必须支持 OpenID Connect 1.0。NGINX Plus 将在授权代码流中充当 OIDC 的中继方。

NGINX Inc. 维护了一个 GitHub 公共仓库，该仓库包含作为 OIDC 与 NGINX Plus 集成参考实现的配置和代码。 “其他参考资料” 部分中的仓库链接提供了关于如何使用自己的 IdP 设置参考实现的最新说明。

详解

该解决方案只与参考实现有关，可确保各位读者拥有最新的解决方案。提供的参考将 NGINX Plus 配置为 OpenID Connect 1.0 授权代码流的中继方。在此配置中，如果将对受保护资源未经授权的请求发送到 NGINX Plus，NGINX Plus 首先会将该请求重定向到 IdP。IdP 会让客户端完成自己的登录流程，然后向 NGINX Plus 返回一个身份验证代码。然后，NGINX Plus 直接与 IdP 通信，用身份验证代码交换一组 ID 令牌。这些令牌使用 JWT 进行验证，并存储在 NGINX Plus 的键值（key-value）存储中。通过使用键值存储，采用高可用性（HA）配置的所有 NGINX Plus 节点都能获得这组令牌。在这个过程中，NGINX Plus 为客户端生成了一个会话 cookie，该 cookie 被用于在键值存储中查找令牌的密钥。然后，客户端的 cookie 被重定向到初始请求的资源。随后的请求将通过使用 cookie 查找 NGINX Plus 键值存储中的 ID 令牌来进行验证。

该功能支持与大多数主要身份提供商进行集成，包括 CA 单点登录（以前称为 SiteMinder）、ForgeRock OpenAM、Keycloak、Okta、OneLogin 和 Ping Identity。作为一项标准，OIDC 与身份验证密切相关 —— 前面提到的身份提供商只是可能集成的一个子集。

其他参考资料

“借助 OpenID Connect 和 NGINX Plus 对访问当前应用的用户进行身份验证”
OpenID Connect
NGINX OpenID Connect GitHub

第 7 章
安全控制

7.0 简介

安全性必须要分层实施，因此您的安全模型必须要有多个层面才能真正得到强化。本章介绍了通过 NGINX 和 NGINX Plus 保护 Web 应用安全的许多不同方法。您可以将这些安全防护方法结合使用来增强安全性。您可能会注意到，本章没有涉及将 NGINX 用作 Web 应用防火墙（WAF）的 ModSecurity 3.0 NGINX 模块。如欲了解 WAF 功能的更多信息，请下载《ModSecurity 3.0 和 NGINX：快速入门指南》。请注意，NGINX Plus 的 NGINX ModSecurity WAF 将于 2024 年 3 月 31 日起停用。更多信息请阅读本博客。

7.1 基于 IP 地址的访问

问题

根据客户端的 IP 地址控制访问。

解决方案

使用 HTTP 或 stream 访问模块控制对受保护资源的访问：

```
location /admin/ {
    deny  10.0.0.1;
    allow 10.0.0.0/20;
    allow 2001:0db8::/32;
    deny  all;
}
```

给定的 location 代码块允许来自 10.0.0.0/20 中的任何 IPv4 地址访问（10.0.0.1 除外），允许来自 2001:0db8::/32 子网中的 IPv6 地址访问，并在收到来自其他任何地址的请求后返回 403。allow 和 deny 指令在 http、server 和 location 上下文以及 TCP/UDP 的 stream、server 上下文中有效。按顺序检查规则，直到找到与远程地址匹配的规则为止。

详解

互联网上的宝贵资源和服务必须要进行多层保护，NGINX 就是其中一层的安全卫士。deny 指令可阻止对给定上下文的访问，而 allow 指令可用来允许被阻止访问的子集。您可以使用 IP 地址、IPv4 或 IPv6、无类别域间路由（CIDR）块范围，关键字 all 和 Unix 套接字。在保护资源时，通常会允许一个内部 IP 地址块，并拒绝所有访问请求。

7.2 允许跨域资源共享

问题

您提供了来自另一个域的资源，需要允许跨域资源共享（CORS），以便浏览器能够利用这些资源。

解决方案

根据 request 方法更改请求头，启用 CORS：

```
map $request_method $cors_method {
  OPTIONS 11;
  GET  1;
  POST 1;
  default 0;
}
server {
  # ...
  location / {
    if ($cors_method ~ '1') {
        add_header 'Access-Control-Allow-Methods'
           'GET,POST,OPTIONS';
        add_header 'Access-Control-Allow-Origin'
           '*.example.com';
        add_header 'Access-Control-Allow-Headers'
```

```
                    'DNT,
                     Keep-Alive,
                     User-Agent,
                     X-Requested-With,
                     If-Modified-Since,
                     Cache-Control,
                     Content-Type';
        }
        if ($cors_method = '11') {
            add_header 'Access-Control-Max-Age' 1728000;
            add_header 'Content-Type' 'text/plain; charset=UTF-8';
            add_header 'Content-Length' 0;
            return 204;
        }
      }
    }
```

这个示例中的内容很多，但通过使用 map 将 GET 和 POST 方法进行分组，简化了示例中的内容。OPTIONS 请求方法向客户端返回有关该服务器 CORS 规则的 *preflight* 请求。在 CORS 下允许使用 OPTIONS、GET 和 POST 方法。设置 Access-Control-Allow-Origin 标头，允许该服务器提供的内容也可以在与此请求头匹配的原始页面上使用。preflight 请求可以在客户端缓存 1,728,000 秒，相当于 20 天。

详解

当 JavaScript 等资源所请求的资源属于其他域时，就会发生 CORS。当请求被视为跨域的时候，浏览器就要遵守 CORS 规则。如果浏览器没有专门允许其使用资源的请求头，那么它将不会使用该资源。为了允许其他子域使用我们的资源，我们必须设置 CORS 请求头，这可通过 add_header 指令完成。如果请求方法是具有标准内容类型的 GET、HEAD 或 POST，并且没有特殊的请求头，则浏览器将发出请求并且只检查来源。其他请求方法将导致浏览器发出 preflight 请求，以检查它将遵守的关于该资源的服务器条款。如果您没有正确设置这些请求头，那么浏览器会在尝试利用这些资源时出错。

7.3 客户端加密

问题

在 NGINX 服务器和客户端之间加密流量。

解决方案

使用一个 SSL 模块加密流量，例如 ngx_http_ssl_module 或 ngx_stream_ssl_module：

```
http{ # All directives used below are also valid in stream
    server {
        listen 8443 ssl;
        ssl_certificate /etc/nginx/ssl/example.crt;
        ssl_certificate_key /etc/nginx/ssl/example.key;
    }
}
```

此配置将服务器设置为侦听使用 SSL/TLS 加密的端口 8443。指令 ssl_certificate 定义了提供给客户端的证书和可选的证书链。ssl_certificate_key 指令定义了 NGINX 用来解密请求和加密响应的密钥。许多 SSL/TLS 协商配置默认为 NGINX 版本的缺省配置。

详解

安全传输层是最常见的加密传输中信息的方法。截至本文撰写之时，TLS 协议比 SSL 协议更受欢迎。这是因为现在人们认为 SSL 版本 1 至 3 不安全。尽管协议名称可能不同，但 TLS 仍然建立了安全套接字层。NGINX 能够让您的服务保护您和客户端之间的信息，进而保护客户端和您的业务。在使用 CA 签名的证书时，需要将证书与证书颁发机构链连接起来。在连接证书和证书颁发机构链时，证书应位于证书颁发机构链文件之上。如果证书颁发机构提供了多个文件作为证书颁发机构链的中间证书，则它们会按顺序进行分层。有关排序问题，请参阅证书提供商的文档。

其他参考资料

Mozilla 安全/服务器端 TLS 页面
Mozilla SSL 配置生成器
SSL Labs 的 SSL 服务器测试

7.4 高级客户端加密

问题

具有高级客户端-服务器加密配置需求。

解决方案

NGINX 的 http 和 stream 模块可完全控制接受的 SSL/TLS 握手。证书和密钥可通过文件路径或可变值的方式提供给 NGINX。NGINX 根据其配置为客户端提供可接受的协议、密码和密钥类型列表。客户端与 NGINX 服务器之间的最高标准是经过协商的。NGINX 可以将客户端-服务器 SSL/TLS 的协商结果缓存一段时间。

下面的示例同时展示了许多选项，意在说明客户端-服务器协商的复杂性：

```
http { # All directives used below are also valid in stream
    server {
        listen 8443 ssl;
        # Set accepted protocol and cipher
        ssl_protocols TLSv1.2 TLSv1.3;
        ssl_ciphers HIGH:!aNULL:!MD5;

        # RSA certificate chain loaded from file
        ssl_certificate /etc/nginx/ssl/example.crt;
        # RSA encryption key loaded from file
        ssl_certificate_key /etc/nginx/ssl/example.pem;

        # Elliptic curve cert from variable value
        ssl_certificate $ecdsa_cert;
        # Elliptic curve key as file path variable
        ssl_certificate_key data:$ecdsa_key_path;

        # Client-Server negotiation caching
        ssl_session_cache shared:SSL:10m;
        ssl_session_timeout 10m;
    }
}
```

服务器接受 SSL 协议版本 TLSv1.2 和 TLSv1.3。接受的密码设置为 HIGH，这是最高标准的宏；对于 aNULL 和 MD5，显式拒绝是通过 ! 符号来表示的。

使用了两组证书密钥对。传递给 NGINX 指令的值展示了提供 NGINX 证书密钥值的不同方式。变量被解释为文件的途径。当带有前缀 data: 时，变量值被解释为直接值。可

提供多种证书密钥格式，以实现与客户端的反向兼容性。客户端支持的、服务器接受的最强标准将是协商的结果。

如果 SSL/TLS 密钥作为直接值变量暴露，则有可能会被配置记录或暴露。如果将密钥值暴露为变量，请确保您拥有严格的变更和访问控制措施。

SSL 会话缓存和超时允许 NGINX worker 进程在给定的时间内缓存和存储会话参数。作为单个实例化中的进程，NGINX worker 进程会相互共享此缓存（而非在机器之间共享）。还有许多其他会话缓存选项可以帮助提高所有类型的用例的性能或安全性。您可以结合使用不同的会话缓存选项。但是，指定没有默认值的缓存选项后，默认的内置会话缓存将会关闭。

详解

在上面的高级配置示例中，NGINX 为客户端提供了备受推重的密码算法 —— TLS 版本 1.2 或 1.3 的 SSL/TLS 选项，以及使用 RSA 或椭圆曲线密码（ECC）格式密钥的能力。客户端所能实现的最强协议、密码和关键格式是协商的结果。该配置指示 NGINX 将协商结果缓存 10 分钟，可用内存分配为 10MB。

测试发现，ECC 证书的速度比同等强度的 RSA 证书快。密钥占用空间较小，可以提供更多的 SSL/TLS 连接及更快的握手速度。NGINX 允许您配置多个证书和密钥，然后为客户端浏览器提供最佳证书。您可以利用更新的技术，但仍然服务旧客户端。

在此示例中，NGINX 正在加密它与客户端之间的流量。但是，与上游（upstream）服务器的连接也可能进行了加密。实操指南 7.5 中演示了 NGINX 与上游服务器之间的协商。

其他参考资料

Mozilla 安全/服务器端 TLS 页面
Mozilla SSL 配置生成器
SSL Labs 的 SSL 服务器测试

7.5 Upstream 加密

问题

需要加密 NGINX 和上游服务之间的流量，并为合规性法规设置特定的协商规则；或者上游（upstream）服务位于有安全防护措施的网络之外。

解决方案

使用 HTTP 代理模块的 SSL 指令来指定 SSL 规则：

```
location / {
    proxy_pass https://upstream.example.com;
    proxy_ssl_verify on;
    proxy_ssl_verify_depth 2;
    proxy_ssl_protocols TLSv1.2;
}
```

这些代理指令为 NGINX 设定了需要遵守的特定 SSL 规则。配置的指令可确保 NGINX 验证上游服务上的证书和证书链是否有效，验证深度最多为两个证书。proxy_ssl_protocols 指令指定 NGINX 只使用 TLS 版本 1.2。默认情况下，NGINX 不会验证上游证书并接受所有 TLS 版本。

详解

HTTP 代理模块的配置指令非常庞大，如果您需要加密上游流量，那么至少应该启用验证。您只需更改传递给 proxy_pass 指令的值对应的协议，即可通过 HTTPS 进行代理。但是，这不会验证上游证书。其他指令（如 proxy_ssl_certificate 和 proxy_ssl_ssl_certificate_key）允许您锁定上游加密，以增强安全性。您还可以指定 proxy_ssl_crl 或证书吊销列表，该列表列出了被认为无效的证书。这些 SSL 代理指令有助于增强系统在自己的网络或跨公共互联网的通信通道。

7.6 保护位置

问题

使用 secret 保护 location 代码块。

解决方案

通过 secure link 模块和 secure_link_secret 指令，仅允许拥有安全链接的用户访问资源：

```
location /resources {
    secure_link_secret mySecret;
    if ($secure_link = "") { return 403; }

    rewrite ^ /secured/$secure_link;
}

location /secured/ {
    internal;
    root /var/www;
}
```

这种配置创建了一个内部和公共 location 代码块。请求 URI 需包含 md5 哈希字符串（可通过提供给 secure_link_secret 指令的 secret 进行验证），否则公共 location 代码块 /resources 将返回 403 Forbidden。URI 中的哈希值需进行验证，否则 $secure_link 变量将是一个空字符串。

详解

使用 secret 保护资源是一种有效的文件保护方法。结合使用 secret 与 URI，然后对该字符串进行 md5 哈希计算，并在 URI 中使用这个 md5 哈希算法的十六进制摘要。将哈希值放置在链接中，并通过 NGINX 进行评估。NGINX 知道要请求的文件的路径，因为它位于哈希算法之后的 URI 中。NGINX 还知道您的 secret，因为它通过 secure_link_secret 指令提供。NGINX 能够快速验证 md5 哈希算法，并将 URI 存储在 $secure_link 变量中。如果无法验证哈希值，则将变量设置为空字符串。需要注意的是，传递给 secure_link_secret 的参数必须是静态字符串，不能是变量。

7.7 使用 secret 生成安全链接

问题

需要使用 secret 从应用中生成安全链接。

解决方案

NGINX 中的 secure link 模块接受 md5 哈希字符串的十六进制摘要，其中字符串由 URI 路径和 secret 组成。在上一节实操指南 7.6 的基础上，我们将创建一个安全链接，

与前面的配置示例搭配使用，因为 */var/www/secured/index.html* 上有一个文件。要生成 md5 哈希算法的十六进制接要，我们可以使用 Unix openssl 命令：

```
$ echo -n 'index.htmlmySecret' | openssl md5 -hex
(stdin)= a53bee08a4bf0bbea978ddf736363a12
```

此处展示了我们要保护的 URI *index.html*，它与我们的 secret mySecret 拼接在一起。该字符串被传递到 openssl 命令，输出 md5 十六进制摘要。

下面是使用 Python Standard Library 中包含的 hashlib 库在 Python 中生成相同哈希摘要的示例：

```
import hashlib
hashlib.md5.(b'index.htmlmySecret').hexdigest()
'a53bee08a4bf0bbea978ddf736363a12'
```

现在我们得到了这个哈希摘要，并可以在 URL 中使用它了。我们的示例是 *www.example.com* 通过我们的 */resources* 位置请求 */var/www/secured/index.html* 上的文件。我们的完整 URL 将是：

```
www.example.com/resources/a53bee08a4bf0bbea978ddf736363a12/\
index.html
```

详解

我们可以通过多种方式，用多种语言生成哈希摘要。请记住：URI 路径在 secret 之前；字符串中没有回车；使用 md5 哈希算法的十六进制摘要。

7.8 保护过期的位置

问题

需要保护一个位置，该位置的链接将在未来某个时间过期，并且特定于某个客户端。

解决方案

利用 secure link 模块中的其他指令设置过期时间，并在 secure link 中使用变量：

```
location /resources {
    root /var/www;
    secure_link $arg_md5,$arg_expires;
    secure_link_md5 "$secure_link_expires$uri$remote_addrmySecret";
    if ($secure_link = "") { return 403; }
    if ($secure_link = "0") { return 410; }
}
```

secure_link 指令带两个参数，这两个参数用两个逗号隔开。第一个参数是保存 md5 哈希值的变量。本例中使用了 md5 的 HTTP 参数。第二个参数是一个变量，该变量保存了以 Unix 时间戳格式显示的链接过期时间。secure_link_md5 指令带一个参数，该参数声明了用于生成 md5 哈希值的字符串的格式。与其他配置一样，如果哈希值未经验证，则 $secure_link 变量将设置为空字符串。但是，使用这种用法，如果哈希值匹配，但时间已过期，则 $secure_link 变量将设置为 0。

详解

这种保护链接的用法比实操指南 7.6 中所示的 secure_link_secret 更灵活、更简洁。通过这些指令，您可以在哈希字符串中使用 NGINX 可用的任意数量的变量。在哈希字符串中使用特定于用户的变量可增强安全性，因为用户无法交换到受保护资源的链接。建议使用类似于 $remote_addr 或 $http_x_forwarded_for 的变量，或者由应用生成的会话 cookie 请求头。secure_link 的参数可以来自您喜欢的任何变量，并且能够以最合适的方式命名。问题是：您拥有访问权限吗？您是否在链接有效期内访问？如果您没有访问权限，将会收到消息 Forbidden。如果您拥有访问权限，但链接却过期了，将会收到消息 Gone。HTTP 410 Gone 非常适合过期的链接，因为该条件应被视为永久性的。

7.9 生成过期链接

问题

需要生成一个过期的链接。

解决方案

为过期的时间生成一个 Unix 时间戳格式的时间戳。在 Unix 系统上，您可以使用如下所示的日期进行测试：

```
$ date -d "2030-12-31 00:00" +%s --utc
1924905600
```

接下来，您需要连接哈希字符串，以匹配使用 secure_link_md5 指令配置的字符串。在本例中，我们要使用的字符串将是 1924905600/resources/index.html127.0.0.1 mySecret。md5 哈希值与十六进制摘要稍有不同。它是二进制格式的 md5 哈希值，base64 编码，加号（+）转换成了连字符（-），斜杠（/）转换成了下划线（_），删除了等号（=）。以下是在 Unix 系统上操作的一个示例：

```
$ echo -n '1924905600/resources/index.html127.0.0.1 mySecret' \
    | openssl md5 -binary \
    | openssl base64 \
    | tr +/  -_ \
    | tr -d =
sqysOw5kMvQBL3j9ODCyoQ
```

现在我们得到了哈希值，我们可以将它和过期日期用作参数：

```
/resources/index.html?md5=sqysOw5kMvQBL3j9ODCyoQ&expires=1924905600
```

下面是 Python 中使用相对过期时间的更实际的示例，它将链接设置为生成后一小时过期。在撰写本文之时，此示例适用于 Python 2.7 和使用 Python 标准库的 3.x：

```
from datetime import datetime, timedelta
from base64 import b64encode
import hashlib

# 设置环境变量
resource = b'/resources/index.html'
remote_addr = b'127.0.0.1'
host = b'www.example.com'
mysecret = b'mySecret'

# 生成过期时间戳
now = datetime.utcnow()
expire_dt = now + timedelta(hours=1)
expire_epoch = str.encode(expire_dt.strftime('%s'))

# 计算字符串的 md5 哈希值
uncoded = expire_epoch + resource + remote_addr + mysecret
md5hashed = hashlib.md5(uncoded).digest()

# 对字符串进行 base64 编码和转换
b64 = b64encode(md5hashed)
unpadded_b64url = b64.replace(b'+', b'-')\
    .replace(b'/', b'_')\
    .replace(b'=', b'')

# 格式化并生成链接
linkformat = "{}{}?md5={}?expires={}"
securelink = linkformat.format(
    host.decode(),
    resource.decode(),
    unpadded_b64url.decode(),
    expire_epoch.decode()
)
print(securelink)
```

详解

有了这种模式，我们能够生成具有特殊格式并能在 URL 中使用的安全链接。secret 通过使用从未发送给客户端的变量来提供安全性。您可以使用尽可能多的其他变量来保护位置的安全。md5 哈希计算和 base64 编码是通用的、轻量级的，并且支持几乎任何语言。

7.10 HTTPS 重定向

问题

将未加密的请求重定向到 HTTPS。

解决方案

使用 rewrite 指令将所有 HTTP 流量发送到 HTTPS：

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name _;
    return 301 https://$host$request_uri;
}
```

此配置用于侦听 IPv4 和 IPv6 以及任何主机名的默认服务器的 80 端口。return 语句向使用同一主机和请求 URI 的 HTTPS 服务器返回 301 永久重定向。

详解

在适当的情况下始终重定向到 HTTPS，这是非常重要的。您可能会发现，不需要重定向所有请求，而只需要重定向在客户端和服务器之间传递敏感信息的请求。在这种情况下，您可能希望只将 return 语句放在特定的位置，例如 */login*。

7.11 在 NGINX 之前终止 SSL/TLS 后重定向到 HTTPS

问题

需要重定向到 HTTPS，然而，您已在 NGINX 之前的某个层终止了 SSL/TLS。

解决方案

使用常见的 X-Forwarded-Proto 请求头确定是否需要重定向：

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name _;
    if ($http_x_forwarded_proto = 'http') {
        return 301 https://$host$request_uri;
    }
}
```

这种配置与 HTTPS 重定向十分相似。然而，在这个配置中，我们只在请求头 X-Forwarded-Proto 与 HTTP 相同时进行重定向。

详解

在 NGINX 前面的某个层终止 SSL/TLS 是一种常见的做法，为的是节省计算成本。但是，您需要确保每个请求都是 HTTPS 请求，而终止 SSL/TLS 的层面不具有重定向能力。但它可以设置代理请求头。这种配置适用于 Amazon Web Services Elastic Load Balancer（AWS ELB）等层面，这些层可免费卸载 SSL/TLS。这是一个小技巧，可保护 HTTP 流量的安全。

7.12 HTTP 严格传输安全协议

问题

指示浏览器永远不通过 HTTP 发送请求。

解决方案

设置 Strict-Transport-Security 请求头，使用 HTTP 严格传输安全协议（HSTS）增强功能：

```
add_header Strict-Transport-Security max-age=31536000;
```

此配置将 Strict-Transport-Security 请求头的使用时间设置为最长一年。如 HTTP 请求尝试访问该域，这将指示浏览器始终进行内部重定向，这样所有请求都将通过 HTTPS 发出。

详解

对于某些应用来说，一个 HTTP 请求遭到中间人攻击就可能会摧毁整个公司。如果一

个包含敏感信息的表单通过 HTTP 发送，势必会造成损害，就算是通过 NGINX 进行 HTTPS 重定向也于事无补。这个可选的安全增强功能可指示浏览器，永远不要发出 HTTP 请求，因此请求始终都在加密后发出。

其他参考资料

RFC HTTP 严格传输安全协议标准文档
OWASP HTTP 严格传输安全协议备忘单

7.13 提供多种安全方法

问题

需要提供多种方法将安全性传递到封闭的站点。

解决方案

使用 satisfy 指令，指示 NGINX 您想要满足所用的任何或所有安全方法：

```
location / {
    satisfy any;

    allow 192.168.1.0/24;
    deny  all;

    auth_basic           "closed site";
    auth_basic_user_file conf/htpasswd;
}
```

此配置告知 NGINX 请求 location/ 的用户需要满足其中一种安全方法：请求需要来自 *192.168.1.0/24* CIDR 代码块，或者必须能够提供可以在 *conf/htpasswd* 文件中找到的用户名和密码。satisfy 指令带两个选项：any 或 all。

详解

satisfy 指令是一个不错的方法，它提供了多种验证 Web 应用的方法。在 satisfy 指令中指定 any，代表用户必须满足其中一个安全要求。在 satisfy 指令中指定 all，代表用户必须满足所有安全要求。这个指令可以与实操指南 7.1 中详述的 http_access_module、实操指南 6.1 中详述的 http_auth_basic_module、实操指南 6.2 中详述的 http_auth_request_module 及实操指南 6.3 中详述的 http_auth_jwt_module 一起使用。

只有通过多层才能真正实现安全性。satisfy 指令可帮助您为需要深度安全规则的位置和服务器实现这个目标。

7.14 NGINX Plus 动态应用层 DDoS 防护

问题

需要动态分布式拒绝服务（DDOS）防护解决方案。

解决方案

使用 NGINX Plus 构建集群感知型速率限制和自动拦截列表：

```
limit_req_zone   $remote_addr zone=per_ip:1M rate=100r/s sync;
                 # Cluster-aware rate limit
limit_req_status 429;

keyval_zone zone=sinbin:1M timeout=600 sync;
              # Cluster-aware "sin bin" with
              # 10-minute TTL
keyval $remote_addr $in_sinbin zone=sinbin;
              # Populate $in_sinbin with
              # matched client IP addresses

server {
    listen 80;
    location / {
        if ($in_sinbin) {
            set $limit_rate 50; # Restrict bandwidth of bad clients
        }

        limit_req zone=per_ip;
              # Apply the rate limit here
        error_page 429 = @send_to_sinbin;
              # Excessive clients are moved to
              # this location
        proxy_pass http://my_backend;
    }

    location @send_to_sinbin {
        rewrite ^ /api/3/http/keyvals/sinbin break;
              # Set the URI of the
              # "sin bin" key-val
        proxy_method POST;
        proxy_set_body '{"$remote_addr":"1"}';
        proxy_pass http://127.0.0.1:80;
    }
```

```
        location /api/ {
            api write=on;
            # directives to control access to the API
        }
    }
```

详解

该解决方案通过使用同步键值（key-value）存储来使用同步速率限制，以动态响应 DDoS 攻击并减轻其影响。提供给 limit_req_zone 和 keyval_zone 指令的 sync 参数将共享内存区与 NGINX Plus active-active 集群中的其他机器同步。上面的示例标识了每秒发送超过 100 个请求的客户端 —— 无论哪个 NGINX Plus 节点接收请求。当客户端超过速率限制时，将通过调用 NGINX Plus API 将其 IP 地址添加到“sin bin”键值存储中。集群内的 sin bin 是同步的。无论哪个 NGINX Plus 节点接收请求，来自 sin bin 中的客户端的其他请求都会面临非常低的带宽限制。限制带宽比直接拒绝请求更可取，因为它不会向客户端清楚地表明 DDoS 防护已生效。10 分钟后，客户端将自动从 sin bin 中移除。

7.15 安装和配置 NGINX Plus 的 NGINX App Protect WAF 模块

问题

安装和配置 NGINX App Protect WAF 模块。

解决方案

请参考《NGINX App Protect WAF 管理指南》。请不要跳过有关从单独的仓库中安装 NGINX App Protect WAF 签名的部分。

确保 NGINX App Protect WAF 模块由 NGINX Plus 在 main 上下文中使用 load_module 指令动态加载，并使用 app_protect_* 指令启用：

```
user nginx;
worker_processes auto;

load_module modules/ngx_http_app_protect_module.so;

# ... Other main context directives

http {
    app_protect_enable on;
```

```
        app_protect_policy_file "/etc/nginx/AppProtectTransparentPolicy.json";
        app_protect_security_log_enable on;
        app_protect_security_log "/etc/nginx/log-default.json"
            syslog:server=127.0.0.1:515;

        # ... Other http context directives
    }
```

在该示例中，app_protect_enable 指令设置为 on，启用了当前上下文的模块。该指令以及下面所有指令在 http 上下文，以及 HTTP 的 server 和 location 上下文中都是有效的。app_protect_policy_file 指令指向一个 NGINX App Protect WAF 策略文件，接下来我们将对其进行定义；如果没有定义，则使用默认策略。接下来配置了安全日志，并需要远程日志服务器。例如，我们将其配置为本地系统日志（Syslog）侦听器。app_protect_security_log 指令带两个参数，第一个参数是一个定义了日志设置的 JSON 文件，第二个参数是日志数据流目的地。日志设置文件将稍后在本节展示。

构建一个 NGINX App Protect WAF 策略文件，将其命名为 */etc/nginx/AppProtectTransparentPolicy.json*：

```
{
    "policy": {
        "name": "transparent_policy",
        "template": { "name": "POLICY_TEMPLATE_NGINX_BASE" },
        "applicationLanguage": "utf-8",
        "enforcementMode": "transparent"
    }
}
```

该策略文件通过使用模板来配置默认的 NGINX App Protect WAF 策略，将策略名称设置为 transparent_policy，将 enforcementMode 设置为 transparent，这表示 NGINX Plus 将记录日志但不会拦截日志。透明（transparent）模式非常适合在新策略实施之前对其进行测试。

将 enforcementMode 改为 blocking，启用拦截模式。该策略文件可以命名为 */etc/nginx/AppProtectBlockingPolicy.json*。要在两个文件之间切换，请在 NGINX Plus 配置中更新 app_protect_policy_file 指令：

```
{
    "policy": {
        "name": "blocking_policy",
        "template": { "name": "POLICY_TEMPLATE_NGINX_BASE" },
        "applicationLanguage": "utf-8",
        "enforcementMode": "blocking"
    }
}
```

要启用 NGINX App Protect WAF 的某些保护特性，需启用一些违规行为：

```
{
    "policy": {
        "name": "blocking_policy",
        "template": { "name": "POLICY_TEMPLATE_NGINX_BASE" },
        "applicationLanguage": "utf-8",
        "enforcementMode": "blocking",
        "blocking-settings": {
            "violations": [
                {
                    "name": "VIOL_JSON_FORMAT",
                    "alarm": true,
                    "block": true
                },
                {
                    "name": "VIOL_PARAMETER_VALUE_METACHAR",
                    "alarm": true,
                    "block": false
                }
            ]
        }
    }
}
```

在上面的示例中，向策略添加了两种违规行为。注意，VIOL_PARAMETER_VALUE_METACHAR 未设置为 block，而只是设置为 alarm；而 VIOL_JSON_FORMAT 设置为 block 和 alarm。当设置为 blocking 时，该功能允许重写默认的 enforcementMode。当 enforcementMode 设置为 transparent 时，将优先使用默认的强制设置。

构建一个 NGINX Plus 日志文件，将其命名为 */etc/nginx/log-default.json*：

```
{
   "filter":{
      "request_type":"all"
   },
   "content":{
      "format":"default",
      "max_request_size":"any",
      "max_message_size":"5k"
   }
}
```

该文件由 app_protect_security_log 指令在 NGINX Plus 配置中定义，并且是 NGINX App Protect WAF 日志的必要文件。

详解

该解决方案演示了使用 NGINX App Protect WAF 模块配置 NGINX Plus 的基础知识。NGINX App Protect WAF 模块支持完整的 Web 应用防火墙（WAF）定义。这些定义源自 F5 高级应用安全防护功能。这套全面的 WAF 攻击签名已得到广泛的实践测试和证明。通过在 NGINX Plus 安装过程中添加这些签名，可同时获得强大的 F5 应用安全防护功能以及 NGINX 平台的敏捷性。

安装并启用该模块后，大多数配置都在策略文件中完成。本节中的策略文件展示了如何启用主动拦截、被动监控和透明模式，并解释了如何在出现违规时重写该功能。违规行为只是所提供的其中一种防护措施。其他防护措施包括 HTTP 合规、规避技术、攻击特征、服务器技术、数据保护等等。要追踪 NGINX App Protect WAF 日志，需要使用 NGINX Plus 日志格式，并将日志发送到远程侦听服务、文件或 /dev/stderr。

如果您正在使用 NGINX Controller ADC[4]，那么您可以通过 NGINX Controllers App Security 组件启用 NGINX App Protect WAF 功能，并通过 Web 界面查看 WAF 指标。

其他参考资料

《NGINX App Protect WAF 管理指南》
《NGINX App Protect WAF 配置指南》
《NGINX Controller App Security 指南》
《NGINX App Protect DoS 部署指南》

[4] NGINX Controller 已于 2022 年正式停售，点击了解相关新产品 NGINX Management Suite。

第 8 章
HTTP/2

8.0 简介

HTTP/2 是 HTTP 协议的主要修订版。此修订版的大部分更改都集中在传输层，例如在单个 TCP 连接上实现完整的请求和响应多路复用。通过压缩 HTTP 请求头字段提高了效率，并添加了请求优先级支持。该协议的另一个重要补充是支持服务器向客户端推送消息。本章详细介绍了在 NGINX 中启用 HTTP/2 的基本配置，以及如何配置 Google open source remote procedure call（gRPC）和 HTTP/2 服务器推送支持。

8.1 基本配置

问题

想要利用 HTTP/2。

解决方案

在 NGINX 服务器上启用 HTTP/2：

```
server {
    listen 443 ssl http2 default_server;

    ssl_certificate    server.crt;
    ssl_certificate_key server.key;
    # ...
}
```

详解

要启用 HTTP/2，您只需要将 http2 参数添加到 listen 指令即可。然而，尽管该协议不需要将连接封装在 SSL/TLS 中，但 HTTP/2 客户端的某些实现仅支持加密连接上的 HTTP/2。另一个问题是，HTTP/2 规范阻止了许多 TLS 1.2 密码套件，因此会导致握手失败。NGINX 默认使用的密码套件不在规范阻止列表上。TLS 的应用层协议协商（Application-Layer Protocol Negotiation）允许应用层协商在安全连接上使用哪个协议，以避免额外往返。为了测试设置是否正确，您可以在 Chrome 和 Firefox 浏览器中安装一个插件，显示站点何时使用 HTTP/2，或者在命令行上执行 nghttp 实用程序。

其他参考资料

HTTP/2 RFC 密码套件黑名单
Chrome HTTP/2 和 SPDY 指示器插件
Firefox HTTP 指示器插件

8.2 gRPC

问题

需要终止、检查、路由或负载均衡 gRPC 方法调用。

解决方案

使用 NGINX 代理 gRPC 连接：

```
server {
    listen 80 http2;

    location / {
        grpc_pass grpc://backend.local:50051;
    }
}
```

在这个配置中，NGINX 侦听端口 80 上未加密的 HTTP/2 流量，并将这些流量代理到端口 50051 上名为 backend.local 的机器。grpc_pass 指令指示 NGINX 将该通信视为 gRPC 调用。后端服务器前面的 grpc:// 不是必需的，但它可以直接显示后端通信是未加密的。

要在客户端和 NGINX 之间利用 TLS 加密，并在将调用传递给应用服务器之前终止加密，需像在第一节中那样启用 SSL 和 HTTP/2：

```
server {
    listen 443 ssl http2 default_server;

    ssl_certificate server.crt;
    ssl_certificate_key server.key;
    location / {
        grpc_pass grpc://backend.local:50051;
    }
}
```

该配置在 NGINX 上终止了 TLS，并通过未加密的 HTTP/2 将 gRPC 通信传递到应用。

要配置 NGINX 加密与应用服务器的 gRPC 通信，提供端到端加密流量，只需修改 grpc_pass 指令，在服务器信息之前指定 grpcs://（注意增加的 s 表示安全通信）：

```
        grpc_pass grpcs://backend.local:50051;
```

您还可以根据 gRPC URI（包括软件包、服务和方法）通过 NGINX 去路由调用到不同的后端服务。方法是使用 location 指令：

```
location /mypackage.service1 {
    grpc_pass grpc://$grpc_service1;
}
location /mypackage.service2 {
    grpc_pass grpc://$grpc_service2;
}
location / {
    root /usr/share/nginx/html;
    index index.html index.htm;
}
```

该配置示例使用 location 指令在两个单独的 gRPC 服务之间路由传入的 HTTP/2 流量，并使用一个 location 提供静态内容。对 mypackage.service1 服务的方法调用被定向到变量 grpc_service1 的值，可能包含主机名或 IP 和可选端口。对 mypackage.service2 的调用被定向到变量 grpc_ser vice2 的值。location / 捕获任何其他 HTTP 请求并提供静态内容。这展示了 NGINX 如何在相同的 HTTP/2 端点和相应的路由下支持 gRPC 和非 gRPC。

对 gRPC 调用的负载均衡也类似于非 gRPC HTTP 流量：

```
upstream grpcservers {
    server backend1.local:50051;
    server backend2.local:50051;
}
server {
    listen 443 ssl http2 default_server;

    ssl_certificate server.crt;
    ssl_certificate_key server.key;
    location / {
        grpc_pass grpc://grpcservers;
    }
}
```

upstream 代码块处理 gRPC 的方式与处理其他 HTTP 流量的方式完全相同。唯一的区别是 upstream 由 grpc_pass 引用。

详解

NGINX 能够接收、代理、负载均衡、路由和终止加密的 gRPC 调用。借助 gRPC 模块，NGINX 可设置、更改或删除 gRPC 调用请求头，设置请求超时，及设置上游（upstream）SSL/TLS 规范。当 gRPC 通过 HTTP/2 协议进行通信时，您可以配置 NGINX 在同一端点上接受 gRPC 和非 gRPC Web 流量。

8.3 HTTP/2 服务器推送

问题

需要预先将内容推送到客户端。

解决方案

使用 NGINX 的 HTTP/2 服务器推送功能：

```
server {
    listen 443 ssl http2 default_server;

    ssl_certificate server.crt;
    ssl_certificate_key server.key;
    root /usr/share/nginx/html;

    location = /demo.html {
        http2_push /style.css;
        http2_push /image1.jpg;
    }
}
```

详解

要使用 HTTP/2 服务器推送功能，您必须像在实操指南 8.1 中所做的那样，将服务器配置为使用 HTTP/2。配置完成后，您可以指示 NGINX 使用 http2_push 指令预先推送特定文件。该指令带一个参数，即推送到客户端的文件的完整 URI 路径。

如果代理应用包含名为 Link 的 HTTP 响应头，则 NGINX 也可以自动将资源推送到客户端。该 HTTP 响应能够指示 NGINX 预载指定的资源。要启用该功能，请在 NGINX 配置中添加 http2_push_preload on。

第 9 章
复杂的媒体串流

9.0 简介

本章涵盖了 MPEG-4（MP4）或 Flash Video（FLV）格式的 NGINX 的流媒体。NGINX 被广泛用于向大众分发和串流内容。NGINX 支持行业标准格式和串流技术，本章将对此展开介绍。NGINX Plus 可通过 HTTP Live Stream（HLS）模块动态分割内容，及提供已分段媒体的 HTTP Dynamic Streaming（HDS）。NGINX 具有原生的带宽限制功能，并且 NGINX Plus 的高级功能提供了比特率限制，能够以最有效的方式交付内容，同时以最少的服务器资源触及最多用户。

9.1 传输 MP4 和 FLV 格式的文件

问题

需要流式传输 MP4 或 FLV 格式的数字媒体。

解决方案

指定一个 HTTP location 代码块，提供 *.mp4* 或 *.flv* 格式的视频。NGINX 将使用渐进式下载或 HTTP 伪流技术流式传输媒体并支持搜索：

```
http {
    server {
        # ...

        location /videos/ {
```

```
            mp4;
        }
        location ~ \.flv$ {
            flv;
        }
    }
}
```

示例中的第一个 location 代码块告知 NGINX，*videos* 目录中的文件是 MP4 格式的，可以通过渐进式下载进行传输。第二个 location 代码块告知 NGINX，所有以 *.flv* 结尾的文件都是 FLV 格式的，可以利用 HTTP 伪流技术进行传输。

详解

NGINX 中的流视频或音频文件处理就像单个指令一样简单。渐进式下载使客户能够在文件下载完成之前播放媒体文件。NGINX 为两种格式的媒体提供未下载部分搜索支持。

9.2 使用 NGINX Plus 的 HLS 模块进行流式传输

问题

需要 HTTP Live Streaming（HLS）处理 MP4 文件中封装的 H.264/AAC 编码内容。

解决方案

利用 NGINX Plus 的 HLS 模块进行实时分段、分包和多路复用，及控制分段缓冲等等，例如转发 HLS 参数：

```
location /hls/ {
    hls; # Use the HLS handler to manage requests

    # Serve content from the following location
    alias /var/www/video;

    # HLS parameters
    hls_fragment            4s;
    hls_buffers         10 10m;
    hls_mp4_buffer_size     1m;
    hls_mp4_max_buffer_size 5m;
}
```

该 location 代码块指示 NGINX 流式传输来自 */var/www/video* 目录的 HLS 媒体，并将该媒体分割成 4 秒长度的片段。HLS 缓冲区的数量设置为 10，大小为 10MB。初始 MP4 缓冲区大小设置为 1MB，最大为 5MB。

详解

NGINX Plus 的 HLS 模块支持动态复用转换 MP4 媒体文件。许多指令都支持控制媒体的分段和缓冲方式。location 代码块必须配置为使用 HLS 处理程序将媒体作为 HLS 流进行传输。HLS 分段时长设置为几秒，指示 NGINX 按时间长度对媒体进行分段。缓冲的数据量通过 hls_buffers 指令进行设置，该指令指定了缓冲区的数量和大小。允许客户端在缓冲数据达到一定数量（由 hls_mp4_buffer_size 指定）后开始播放媒体文件。但这可能需要较大的缓冲区，因为视频相关的元数据可能会超过初始缓冲区的大小。缓冲区的上限通过 hls_mp4_max_buffer_size 设置。NGINX 可通过这些缓冲变量优化最终用户体验。为这些指令选择正确的值需要了解目标受众和媒体。举例来说，如果您的大部分媒体是大型视频文件，并且目标受众拥有很高的带宽，那么您可以选择更大的最大缓冲区和更长时间的分段。这样，与内容相关的元数据在初始下载时不会出错，并且用户能够接收到更大的片段。

9.3 使用 NGINX Plus 的 HDS 模块进行流式传输

问题

需要支持已经分段且与元数据分开的 Adobe HTTP Dynamic Streaming（HDS）文件。

解决方案

NGINX Plus 通过 F4F 模块支持分段的 FLV 文件，使用该功能向用户提供 Adobe Adaptive Streaming（Adobe 自适应流媒体）：

```
location /video/ {
    alias /var/www/transformed_video;
    f4f;
    f4f_buffer_size 512k;
}
```

该示例指示 NGINX Plus 使用 NGINX Plus F4F 模块向客户端提供来自磁盘某个位置的此前已分段的媒体。索引文件（*.f4x*）的缓冲区大小设置为 512KB。

详解

NGINX Plus F4F 模块支持 NGINX 向最终用户提供此前已分段的媒体。这样的配置就像在 HTTP location 代码块内部使用 f4f 处理程序一样简单。f4f_buffer_size 指令为此类媒体的索引文件配置缓冲区大小。

9.4 使用 NGINX Plus 限制带宽

问题

需要限制下游（downstream）媒体串流客户端的带宽，且不会影响观看体验。

解决方案

利用 NGINX Plus 的比特率限制功能处理 MP4 媒体文件：

```
location /video/ {
    mp4;
    mp4_limit_rate_after 15s;
    mp4_limit_rate       1.2;
}
```

该配置允许下游客户端在应用比特率限制之前下载 15 秒。15 秒钟过后，客户端能够以 120% 的比特率速率下载媒体，因此客户端的下载速度始终快于播放速度。

详解

NGINX Plus 的比特率限制能够让流媒体服务器根据传输的媒体动态限制带宽，客户端可以下载尽可能多的内容，从而确保无缝的用户体验。实操指南 9.1 中描述的 MP4 处理程序将这个 location 代码块指定为流式传输 MP4 媒体格式。速率限制指令（例如 mp4_limit_rate_after）告知 NGINX 仅在指定的时间（几秒钟）之后限制流量的速度。MP4 速率限制方面的另一个指令是 mp4_limit_rate，该指令指定了允许客户端下载的比特率（相对于媒体的比特率）。赋予 mp4_limit_rate 指令的值 1 指定 NGINX 将带宽限制为媒体的比特率（1 比 1）。如果为 mp4_limit_rate 指令提供的值超过 1，用户将能够以超过观看的速度下载，进而缓冲媒体，并在下载时无缝观看媒体内容。

第 10 章
云部署

10.0 简介

云提供商的出现改变了 Web 应用托管的现状。过去，配置新机器等流程通常需要花费数小时到几个月的时间，如今，只需点击或调用 API 就能创建一个流程。这些云提供商通过按使用付费（即用即付）的模式租赁虚拟机（称为基础设施即服务（IaaS））或托管软件解决方案。工程师可以构建整个测试环境，并在不需要时将其拆除。这些云提供商还支持应用根据性能需求即时实现水平扩展。本章介绍了 NGINX 和 NGINX Plus 在几个主要云提供商平台上的基本部署。

10.1 AWS 上的自动配置

问题

在 Amazon Web Services（AWS）上自动配置 NGINX 服务器，以实现机器的自动配置。

解决方案

利用 EC2 UserData 以及预配置的 Amazon Machine Image（AMI）。使用 NGINX 以及安装的任何受支持的软件包创建 Amazon Machine Image。利用 Amazon Elastic Compute Cloud（EC2）UserData 在运行时配置任何环境特定的配置。

详解

在 AWS 上配置 NGINX 服务器时，有三种思维方式：

启动时配置
: 首先使用通用的 Linux 镜像启动，然后在启动时运行配置管理或 shell 脚本，以配置服务器。这种方式的启动速度很慢，并且容易出错。

完全配置 AMI
: 完全配置服务器，然后刻录 AMI。这种方式的启动速度非常快，并且非常准确。但在配置周围环境时不够灵活，而且维护许多镜像可能会很复杂。

部分配置 AMI
: 它集成了前面两种方式的优势。部分配置是将软件要求安装并刻录到 AMI，并且环境配置在启动时完成。这种方式比完全配置灵活，比启动时配置快速。

无论选择部分配置还是完全配置 AMI，都需要自动化配置过程。要创建 AMI 构建流水线，建议使用几个工具：

配置管理工具
: 配置管理工具在代码中定义了服务器的状态，例如运行哪个版本的 NGINX、以什么用户的身份运行、要使用的 DNS 解析器以及由谁代理上游（upstream）。该配置管理代码可以像软件项目一样进行源代码控制和版本控制。第 5 章描述了一些热门的配置管理工具，例如 Chef 和 Ansible。

来自 HashiCorp 的 Packer
: Packer 可用于自动运行几乎任何虚拟化或云平台的配置管理，并在运行成功后刻录机器镜像。Packer 首先在您选择的平台上构建了一个虚拟机（VM），然后使用 SSH 连接虚拟机，运行您指定的任何配置，并刻录镜像。您可以利用 Packer 运行配置管理工具，并根据您的规格要求安全可靠地刻录机器镜像。

如要在启动时进行环境配置，您可以利用 Amazon EC2 UserData 在实例第一次启动时运行命令。如果您使用的是部分配置方法，则可以在启动时利用它来配置基于环境的项目。基于环境的配置可能包括侦听的服务器名称、要使用的解析器、要代理的域名或开始使用的上游（upstream）服务器池。UserData 是一个 base64 编码的字符串，在第一次启动和运行时下载。UserData 可以像 AMI 中其他启动进程访问的环境文件一样简单，也可以是用 AMI 中存在的任何语言编写的脚本。UserData 通常是一个 bash

脚本，用于指定变量或下载变量，并将其传递给配置管理工具。配置管理工具可确保系统配置无误，模板配置文件基于环境变量及重新加载服务。UserData 运行之后，NGINX 机器应已安全可靠地进行了完全配置。

10.2 无需 AWS ELB 将流量路由到 NGINX 节点

问题

需要将流量路由到多个 active-active NGINX 节点或创建一个 active-passive 故障转移集，从而无需在 NGINX 前面放置负载均衡器便可以实现高可用性。

解决方案

使用 Amazon Route 53 DNS 服务将流量路由到多个 active-active NGINX 节点，或者在一组 active-passive NGINX 节点之间配置健康检查和故障转移。

详解

DNS 一直都对不同的服务器实施负载均衡，即使是迁移到云也不会改变这一点。Amazon 的 Route 53 服务提供了 DNS 服务，它具有许多高级特性，并且都可通过一个 API 获得。Route 53 服务具有所有典型的 DNS 技巧，例如单个 A 记录上的多个 IP 地址和加权 A 记录。在运行 active-active NGINX 节点时，您需要使用其中一个 A 记录特性在所有节点上分配负载。当一个 A 记录中列出多个 IP 地址时，使用轮询算法。加权分布可通过定义 A 记录中每个服务器 IP 地址的权重来实现负载的不均匀分布。

Route 53 另外一个更有趣的功能是健康检查功能。您可以通过建立 TCP 连接或发送 HTTP 或 HTTPS 请求来配置 Route 53 监控端点的健康。健康检查具有高度可配置的选项，包括 IP、主机名、端口、URI 路径、间隔速率、监控和地理位置。通过这些健康检查，Route 53 可在 IP 地址失效时终止其循环。您还可以配置 Route 53 在发生故障时将其转移到次级记录，这将是一个高度可用的 active-passive 设置。

Route 53 具有基于地理位置的路由功能，支持以最少的延迟将客户端路由到距其最近的 NGINX 节点。按照地理位置进行路由时，您的客户端将被定向到最近的运行健康的物理位置。当运行采用 active-active 配置的多组基础架构时，可通过使用健康检查自动故障转移到另一个地理位置。

当使用 Route 53 DNS 将流量路由到 Auto Scaling 组的 NGINX 节点时，需要自动创建和删除 DNS 记录。如要在 NGINX 节点扩展时自动添加和删除 Route 53 上的 NGINX 机器，您可以使用 Amazon 的 Auto Scaling 生命周期挂钩触发 NGINX 机器中的脚本或在 Amazon Lambda 上独立运行的脚本。这些脚本将使用 Amazon 命令行接口（CLI）或软件开发套件（SDK）与 Amazon Route 53 API 进行连接，在启动时或终止前添加或删除 NGINX 机器 IP 和之前配置的健康检查。

其他参考资料

"使用 Amazon Route 53 和 NGINX Plus 实现全局服务器负载均衡"

10.3 NLB Sandwich

问题

需要自动扩展 NGINX（开源版）层，并在应用服务器之间均匀、轻松地分配负载。

解决方案

创建一个网络负载均衡器（NLB）。通过控制台创建 NLB 时，系统会提示您创建一个新的目标组。如果不通过控制台创建 NLB，则需要创建此类资源并将其附加到 NLB 上的侦听器。通过启动配置创建一个 Auto Scaling 组，并且该启动配置中配置安装了 NGINX 开源版的 EC2 实例。Auto Scaling 组有一个链接到目标组的配置，该配置自动将 Auto Scaling 组中的任何实例注册到第一次启动时配置的目标组。目标组由 NLB 上的侦听器引用。将上游（upstream）应用放在另一个网络负载均衡器和目标组之后，然后将 NGINX 配置为代理应用 NLB。

详解

这种模式十分常见，被称为 NLB sandwich（参见图 10-1），它将 NGINX 开源版放在 NLB 后面的一个 Auto Scaling 组中，同时将应用 Auto Scaling 组放在另一个 NLB 的后面。在每一层之间使用 NLB 的原因是 NLB 能够与 Auto Scaling 组完美配合使用；它们能够自动注册新的节点并删除被终止的节点，还能运行运行健康检查，并只将流量传递给健康的节点。

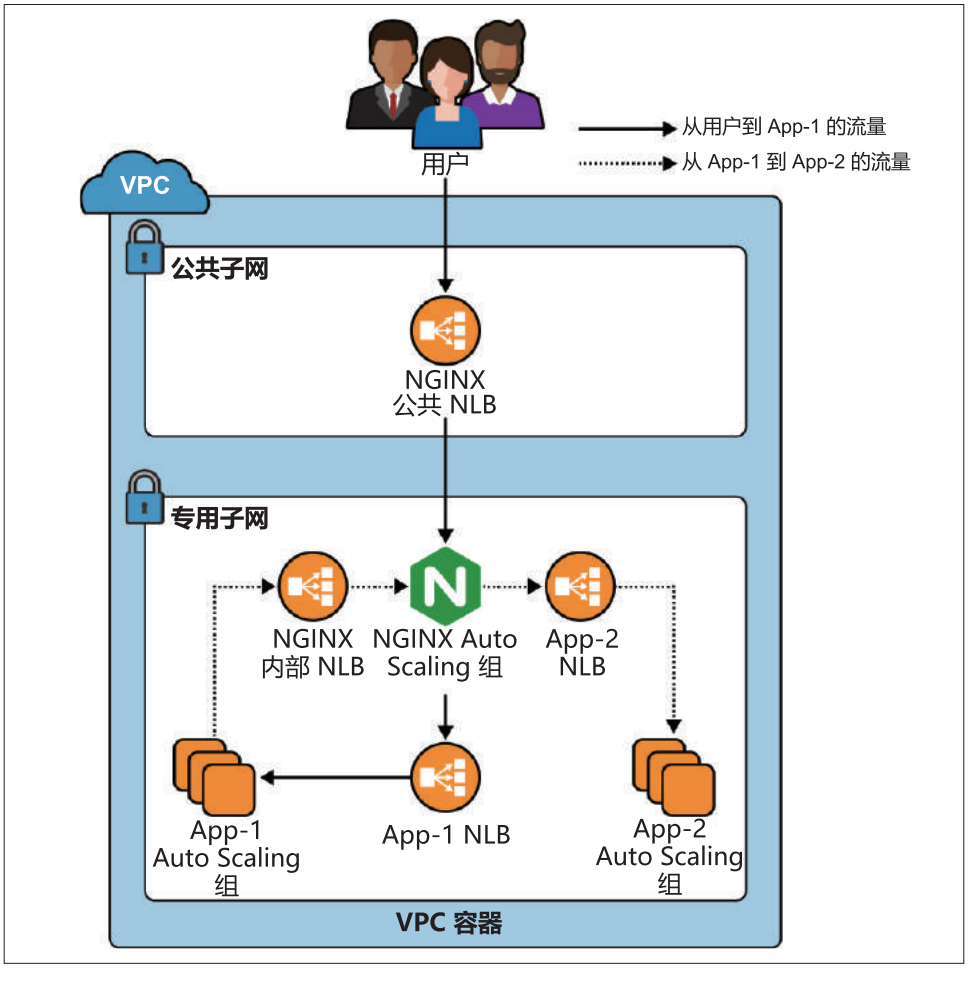


图 10-1. 该图描述了 NGINX 的 NLB Sandwich 模式，它具有一个供内部应用使用的内部 NLB。用户向 App-1 发出请求，App-1 通过 NGINX 向 APP-2 发出请求，从而完成用户的请求。

您可能需要为 NGINX（开源版）层构建第二个内部 NLB，以便应用中的服务通过 NGINX Auto Scaling 组调用其他服务，而无需离开网络并通过公共 NLB 重新进入。这使得 NGINX 位于应用中所有网络流量的中间位置，从而成为应用流量路由的核心。该模式过去称为 *elastic load balancer sandwich*，然而当使用 NGINX 时，NLB 更受欢迎，因为 NLB 是四层负载均衡器，而 ELB 和 ALB 是七层负载均衡器。七层负载均衡器通过 PROXY 协议转换请求，无需使用 NGINX。NGINX Plus 提供了 NLB 的功能集，因此该模式仅适用于 NGINX 开源版。

您也可以在其他云提供商的产品中使用此模式。如果您使用 Azure 负载均衡器和规模集（scale set），或 GCP 负载均衡器和 Auto Scaling 组，则此概念同样适用。此模式的核心价值是使用云原生服务自动注册和负载均衡扩展的应用服务器，以及通过 NGINX 开源版执行代理逻辑。

10.4 从 AWS Marketplace 进行部署

问题

需要轻松地在 AWS 中运行 NGINX Plus，并获得即用即付许可。

解决方案

通过 AWS Marketplace 部署。访问 AWS Marketplace，然后搜索“NGINX Plus”（参见图 10-2）。选择基于您所选择的 Linux 发行版的 AMI，查看详细信息、条款和定价，然后点击 Continue（继续）链接。在下一页，您可以一键接受条款并部署 NGINX Plus，或接受条款并使用 AMI。

详解

通过 AWS Marketplace 部署 NGINX Plus 既轻松简单，又能获得即用即付的许可。您不仅不需要安装任何东西，而且还能享受即用即付的许可模式，无需购买一年期限的许可。该解决方案支持您按需使用 NGINX Plus。您也可以将 NGINX Plus Marketplace AMI 用作溢出容量。最常见的做法是购买与预期负载价值相符的许可，并在 Auto Scaling 组中将 Marketplace AMI 用作溢出容量。这种策略可确保您只需按使用支付许可费。

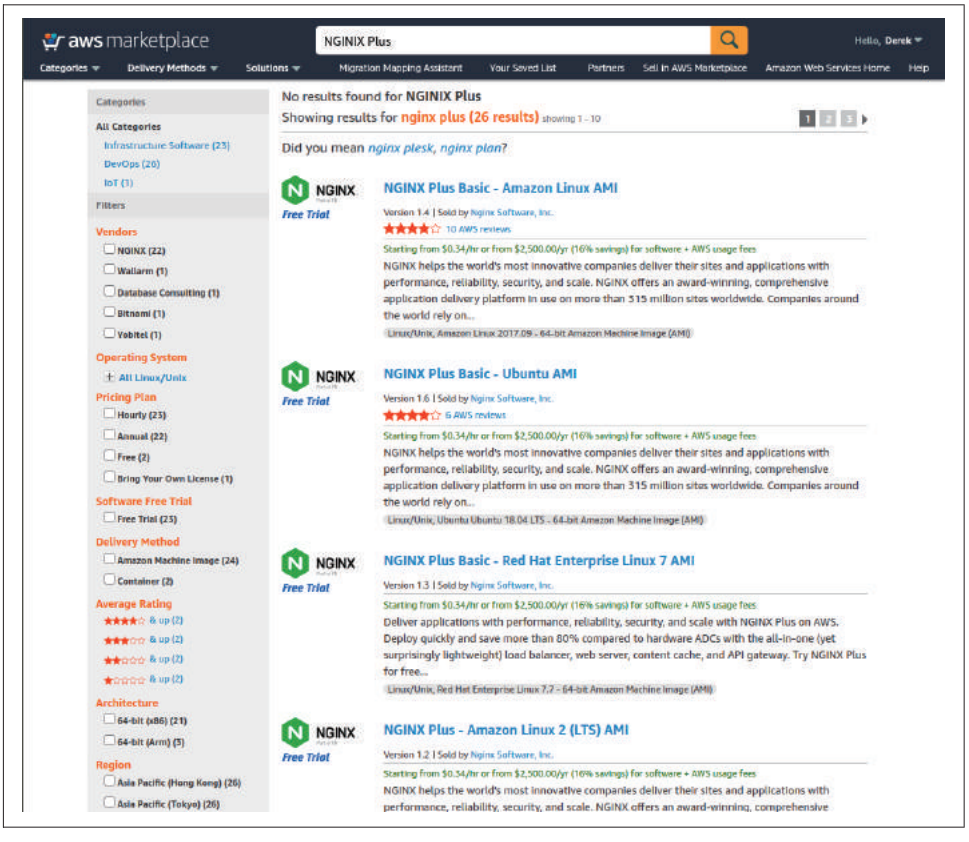


图 10-2. 在 AWS Marketplace 上搜索 NGINX Plus

10.5 在 Azure 上创建 NGINX 虚拟机镜像

问题

为自己的 NGINX 服务器创建虚拟机（VM）镜像，并根据您自己的看法，对其进行配置，以快速创建更多服务器或用于规模集（scale set）。

解决方案

从您选择的基于 Linux 的操作系统中创建虚拟机。启动虚拟机后，登录并安装 NGINX 或 NGINX Plus，您可以选择自己喜欢的方式，从源代码安装，或是通过正在运行的 Linux 发行版的软件包管理工具安装。根据需要配置 NGINX 并创建新的虚拟机镜像。要创建虚拟机镜像，必须先将虚拟机泛化。在对虚拟机进行泛化之前，需要删除 Azure 配置的用户，通过 SSH 进行连接，然后运行以下命令：

```
$ sudo waagent -deprovision+user -force
```

此命令删除了 Azure 在创建虚拟机时配置的用户。-force 选项直接跳过确认步骤。安装 NGINX 或 NGINX Plus 并删除已配置的用户后，即可退出会话。

使用 Azure 登录命令将 Azure CLI 连接到 Azure 帐户，然后确保使用 Azure Resource Manager 模式。现在释放虚拟机：

```
$ azure vm deallocate -g <ResourceGroupName> \
                        -n <VirtualMachineName>
```

虚拟机被释放后，就能够通过 azure vm generalize 命令对其进行泛化：

```
$ azure vm generalize -g <ResourceGroupName> \
                        -n <VirtualMachineName>
```

虚拟机被泛化后，则可以创建镜像。下面的命令将创建一个镜像，同时生成一个 Azure Resources Manager（ARM）模板，用于引导该镜像：

```
$ azure vm capture <ResourceGroupName> <VirtualMachineName> \
    <ImageNamePrefix> -t <TemplateName>.json
```

命令行将输出结果，显示镜像已创建，模板正在保存到指定位置，请求已完成。您可以使用该 ARM 模板从新创建的镜像中创建另一个虚拟机。但是，要使用 Azure 创建的这个模板，必须先创建一个新的网络接口：

```
$ azure network nic create <ResourceGroupName> \
    <NetworkInterfaceName> \
    <Region> \
    --subnet-name <SubnetName> \
    --subnet-vnet-name <VirtualNetworkName>
```

此命令的输出数据将显示新创建网络接口的详细信息。输出数据的第一行是网络接口 ID，您需要使用该 ID 来使用 Azure 创建的 ARM 模板。获得 ID 后，便可以使用 ARM 模板创建部署：

```
$ azure group deployment create <ResourceGroupName> \
    <DeploymentName> \
    -f <TemplateName>.json
```

您将会被提示输入多个输入变量，例如 vmName、adminUserName、adminPassword 和 networkInterfaceId。输入虚拟机名称以及管理用户名和密码。使用从上一个命令中获取的网络接口 ID 作为 networkInterfaceId 提示符的输入。这些变量将作为参数传递到 ARM 模板，并用于从您创建的自定义 NGINX 或 NGINX Plus 镜像中创建新的虚拟机。输入必要的参数后，Azure 将开始从自定义镜像中创建新的虚拟机。

详解

在 Azure 中创建自定义镜像，使您可以随意创建预配置的 NGINX 或 NGINX Plus 服务器的副本。您可使用 Azure ARM 模板连续多次根据需要快速可靠地部署相同的服务器。您可以利用模板中的虚拟机镜像路径，创建不同的基础架构集，例如虚拟机扩展集或其他具有不同配置的虚拟机。

其他参考资料

"如何安装 Azure CLI"
"通过 Azure CLI 登录"
"如何创建虚拟机或 VHD 的托管镜像"

10.6 通过 Azure 上 NGINX 规模集（scale set）进行负载均衡

问题

需要在 Azure 负载均衡器后面扩展 NGINX 节点，以获得高可用性并使用动态资源。

解决方案

创建一个公用或内部的 Azure 负载均衡器。将上一节中创建的 NGINX 虚拟机镜像或实操指南 10.7 中描述的 Marketplace 中的 NGINX Plus 镜像部署到 Azure 虚拟机规模集（VMSS）中。负载均衡器和 VMSS 部署完成后，在负载均衡器上为 VMSS 配置一个后端池。为您想要接受流量的端口和协议设置负载均衡规则，并将其定向到后端池。

详解

通过扩展 NGINX 来实现高可用性或处理峰值负载而不过度提供资源十分常见。而在 Azure 中，可通过 VMSS 来实现这一点。使用 Azure 负载均衡器可在扩展时轻松添加和删除资源池中的 NGINX 节点。借助 Azure 负载均衡器，您可以检查后端池的健康状况，并且只能将流量传递到健康的节点。您可以在 NGINX 前面运行内部 Azure 负载均衡器，这样就可以只通过内部网络访问 NGINX。您可以使用 NGINX 代理 VMSS 内某个应用前的内部负载均衡器，然后使用这个负载均衡器轻松地在池中进行注册和注销。

10.7 通过 Azure Marketplace 进行部署

问题

需要轻松地在 Azure 中运行 NGINX Plus，并获得即用即付许可。

解决方案

通过 Azure Marketplace 部署 NGINX Plus 虚拟机镜像：

1. 在 Azure 仪表板中，选择 New（新）图标，然后使用搜索栏搜索“NGINX”。显示搜索结果。
2. 在列表中，选择 NGINX Inc. 发布的 NGINX Plus 虚拟机镜像。
3. 当系统提示您选择部署模型时，选择“Resource Manager（资源管理器）”选项，然后点击“Create（创建）”按钮。
4. 之后系统将提示您填写表格，指定虚拟机的名称、磁盘类型、默认用户名和密码或 SSH 密钥对公共密钥、接收账单的帐户、想要使用的资源组及位置。
5. 表格填写完毕后，点击“OK（确定）”。表格进入验证阶段。
6. 出现提示后，选择虚拟机大小，然后点击“Select（选择）”按钮。
7. 在下一个面板中，您可以选择可选配置，这将是基于先前所选资源组的默认配置。更改并接受这些选项之后，点击 OK（确定）。
8. 在下一个页面中，查看摘要。您可以选择以 ARM 模板的形式下载这个配置，以便通过 JSON 模板更快地再次创建这些资源。
9. 查看并下载模板后，您可以点击“OK（确定）”，进入购买页面。此页面将通知您使用该虚拟机要产生的费用。点击 Purchase（购买），NGINX Plus 框开始启动。

详解

Azure 和 NGINX 只需几个配置表格就可以轻松地在 Azure 中创建 NGINX Plus 虚拟机。Azure Marketplace 是您按需使用 NGINX Plus 并获得即用即付许可的好方法。借助这种模式，您可以使用 NGINX Plus 的特性，或者在已许可的 NGINX Plus 服务器上按需使用它的溢出容量。

10.8 部署到 Google Compute Engine

问题

需要在 Google Compute Engine 中创建 NGINX 服务器，对 Google Compute 或 App Engine 的其余资源进行负载均衡或代理。

解决方案

在 Google Compute Engine 中启动新的虚拟机。选择虚拟机名称、分区、机器类型和启动盘。配置身份和访问管理、防火墙以及任何高级配置。创建虚拟机。

虚拟机创建完成后，通过 SSH 或 Google Cloud Shell 登录虚拟机。通过指定操作系统类型的软件包管理器安装 NGINX 或 NGINX Plus。根据您的需求配置 NGINX，然后重新加载。

另外，您可通过 Google Compute Engine 启动脚本安装和配置 NGINX，这是创建虚拟机时的高级配置选项。

详解

Google Compute Engine 可即刻提供高度可配置的虚拟机。虚拟机启动不费吹灰之力，即可为您创造各种可能性。Google Compute Engine 在虚拟化的云环境中提供了出色的网络和计算功能。借助 Google Compute 实例，无论何时何地您都可以使用 NGINX 服务器的全部功能。

10.9 创建 Google Compute Image

问题

需要创建 Google Compute Image，以快速实例化虚拟机或为实例组创建实例模板。

解决方案

如实操指南 10.8 中所述创建一个虚拟机。在虚拟机实例上安装和配置 NGINX 后，将启动盘的自动删除状态设为 false。要设置磁盘的自动删除状态，请编辑虚拟机。在磁

盘配置下的“Edit（编辑）”页面上有一个标记为“Delete boot disk when instance is deleted（删除实例时删除启动盘）”的复选框。取消勾选此复选框并保存虚拟机配置。实例的自动删除状态设置为 false 后，删除实例。出现提示后，请勿选择提供删除启动盘的复选框。通过执行这些任务，您将会得到一个未连接的 NGINX 启动盘。

删除实例并得到未连接的启动盘后，您就可以创建 Google Compute Image 了。在 Google Compute Engine 控制台的“Image（镜像）”部分，选择“Create Image（创建镜像）”。系统将提示您输入镜像名称、系列、描述、加密类型和源。您需要使用的源类型是磁盘；为源磁盘选择未连接的 NGINX 启动盘。选择 Create（创建），Google Compute Cloud 将从磁盘创建镜像。

详解

您可以利用 Google Cloud Images 创建引导盘与刚刚创建的服务器相同的虚拟机。创建镜像的价值在于可确保此镜像的每个实例都是相同的。在动态环境中的启动过程中安装软件包时，除非使用私有仓库的版本锁定，否则您将面临软件包版本和更新在生产环境中运行之前未被验证的风险。借助机器镜像，您可以验证在该机器上运行的每个软件包都完全与测试相同，从而增强了服务提供时的可靠性。

其他参考资料

“创建、删除和弃用自定义镜像”

10.10 创建 Google App Engine 代理

问题

需要为 Google App Engine 创建代理，以便在应用之间进行上下文切换或在自定义域中提供 HTTPS 服务。

解决方案

在 Google Compute Cloud 中使用 NGINX。在 Google Compute Engine 中创建虚拟机，或创建一个安装了 NGINX 的虚拟机镜像，并将其作为引导盘创建实例模板。实例模板创建完成后，创建一个使用该模板的实例组。

将 NGINX 配置为代理 Google App Engine 端点。Google App Engine 是公共的，所以确保代理使用 HTTPS，同时还要确保不会在 NGINX 实例中终止 HTTPS，从而允许信息在 NGINX 和 Google App Engine 之间进行不安全的传递。因为 App Engine 仅提供了一个 DNS 端点，所以您将使用 proxy_pass 指令，而不是 NGINX 开源版中的 upstream 代码块，因为开源版无法解析上游（upstream）服务器的 DNS 名称。在代理到 Google App Engine 时，确保将端点设置为 NGINX 中的变量，然后在 proxy_pass 中使用该变量，从而确保 NGINX 对每个请求都进行 DNS 解析。为了让 NGINX 执行任何 DNS 解析，您还需要使用 resolver 指令，定向到您喜欢的 DNS 解析器。谷歌为大众提供了 IP 地址 8.8.8.8。如果您正在使用 NGINX Plus，您将能够在代理到 Google App Engine 时，使用 upstream 代码块中 server 指令上的 resolve 标志、keepalive 连接并获得 upstream 模块的其他好处。

您可以选择将 NGINX 配置文件存储在 Google Storage 中，然后使用实例的启动脚本在启动时下拉配置。无需刻录新镜像即可更改配置。但是，它会增加 NGINX 服务器的启动时间。

详解

如果您使用自己的域，并且希望通过 HTTPS 提供应用，则需要在 Google App Engine 前运行 NGINX。目前，Google App Engine 不支持上传自己的 SSL 证书。因此，如果您想在 appspot.com 以外的其他域为应用提供加密服务，则需使用 NGINX 创建一个代理来侦听您的自定义域。NGINX 将对自己与客户端之间，以及自己与 Google App Engine 之间的通信进行加密。

您想在 Google App Engine 前运行 NGINX 的另一个原因是，在同一个域下托管多个 App Engine 应用，并使用 NGINX 进行基于 URI 的上下文切换。微服务是一种流行的架构，使用 NGINX 等代理路由流量十分常见。Google App Engine 简化了应用的部署，与 NGINX 相结合可提供功能完整的应用交付平台。

第 11 章
容器/微服务

11.0 简介

容器在应用层提供了一层抽象，将软件包和依赖项的安装从部署流程转移到了构建流程。这点很重要，因为工程师现在发布的代码单元无论在何种环境中都以统一方式运行和部署。作为可运行单元提供的容器可降低环境之间发生依赖项和配置混乱的风险。考虑到这一点，组织都迫切希望将应用部署到容器平台上。在容器平台上运行应用时，通常会尽可能地将堆栈（包括代理或负载均衡器）容器化。NGINX 和 NGINX Plus 可以很容易地容器化。它们还包含许多有助于容器化应用交付的特性。本章将重点介绍 NGINX 和 NGINX Plus 容器镜像的构建，有助于在容器化环境中运行的特性，以及 Kubernetes 和 OpenShift 之上的镜像部署。

在进行容器化时，通常的做法是将服务分解到多个较小的应用中，然后再通过 API 网关将分解的服务连接起来。本章第一节介绍了使用 NGINX 作为 API 网关，对请求进行保护、校验、身份验证并将请求路由到适当服务的常见用例场景。最后一节侧重于安全性，介绍了 NGINX Service Mesh，这是一种在容器化环境中使用的基础架构模式，可优化并保护上述服务之间的连接。

在容器中运行 NGINX 或 NGINX Plus 时，应留意几个架构注意事项。当将服务容器化，使用 Docker 日志驱动程序时，必须将日志输出到 */dev/stdout* 并将错误日志定向到 */dev/stderr*。通过这样做，可以将日志传输到 Docker 日志驱动程序，Docker 驱动程序则能够将日志路由到本地合并日志服务器。

在容器化环境中使用 NGINX Plus 时，还需要考虑负载均衡方法。least_time 负载均衡方法专为容器化网络叠加层而设计。NGINX Plus 偏爱较短的响应时间，会将传入请求传递给平均响应时间最短的上游服务器。当所有服务器经过充分的负载均衡，在等量负载下运行后，NGINX Plus 可通过优选最近网络上的服务器，降低网络延迟。

11.1 使用 NGINX 作为 API 网关

问题

需要使用 API 网关对用例的传入请求进行校验、身份验证、操纵和路由。

解决方案

使用 NGINX 或 NGINX Plus 作为 API 网关。API 网关为接入一个或多个应用编程接口（API）提供了入口点。NGINX 非常适合这一角色。本节将重点介绍一些核心概念，并列出本书中可提供更多相关详情的章节。值得一提的是，NGINX 已就该主题发布了一本完整的电子书，即由 Liam Crilly 编写的《将 NGINX 部署为 API 网关》。

首先在 API 网关配置文件中为 API 网关定义 server 块。使用诸如 */etc/nginx/api_gateway.conf* 之类的名称命名。

```
server {
    listen 443 ssl;
    server_name api.company.com;
    # SSL 设置，请见第 7 章

    default_type application/json;
}
```

在服务器定义中添加一些基本的错误处理响应：

```
proxy_intercept_errors on;

error_page 400 = @400;
location @400 { return 400 '{"status":400,"message":"Bad request"}\n'; }

error_page 401 = @401;
location @401 { return 401 '{"status":401,"message":"Unauthorized"}\n'; }

error_page 403 = @403;
location @403 { return 403 '{"status":403,"message":"Forbidden"}\n'; }

error_page 404 = @404;
location @404 { return 404 '{"status":404,"message":"Resource not found"}\n'; }
```

可直接将上一节中的 NGINX 配置添加到 */etc/nginx/api_gateway.conf* 内的 server 块中或单独文件中，并通过 include 指令导入。有关 include 指令的介绍请参阅实操指南 18.1。

使用 include 指令将该服务器配置导入到 http 上下文中的主配置文件 *nginx.conf* 中：

```
include /etc/nginx/api_gateway.conf;
```

现在，需要定义上游服务端点。第 2 章介绍了负载均衡，并讨论了 upstream 代码块。值得注意的是，upstream 在 http 上下文中有效，在服务器上下文中无效。必须在 server 代码块外部包含或设置下列内容：

```
upstream service_1 {
    server 10.0.0.12:80;
    server 10.0.0.13:80;
}
upstream service_2 {
    server 10.0.0.14:80;
    server 10.0.0.15:80;
}
```

可根据用例将内联服务声明为内含文件或按服务包含的文件。还存在应将服务定义为代理位置端点的情况；在这种情况下，建议将该端点定义为可供自始至终使用的变量。第 5 章“可编程性和自动化”讨论了从 upstream 代码块中自动添加和删除机器的实现方法。

在 server 代码块中为每项服务构建一个内部可路由的位置：

```
location = /_service_1 {
    internal;
    # 服务通用配置
    proxy_pass http://service_1/$request_uri;
}
location = /_service_2 {
    internal;
    # 服务通用配置
    proxy_pass http://service_2/$request_uri;
}
```

通过为这些服务定义内部可路由位置，可一次性定义通用于这些服务的配置，从而避免反复定义。

现在我们需要构建负责为给定服务定义特定 URI 路径的 location 代码块。这些代码块将对请求进行适当的验证和路由。API 网关可以只基于路径路由请求，也可以为接受的每个 API URI 定义特定规则。在后一种情况中，需要设计文件组织结构，并使用 NGINX includes 导入配置文件。有关该概念的讨论请参阅实操指南 18.1。

为 API 网关新建目录：

```
mkdir /etc/nginx/api_conf.d/
```

在适合配置结构的路径上的一个文件中定义 location 代码块，构建服务用例的规范。使用 rewrite 指令将请求定向到先前配置的 location 代码块，然后 location 代码块将请求代理到服务。在下面的示例中，rewrite 指令指示 NGINX 重新处理 URL 发生更改的请求。该示例定义了特定于 API 资源的规则，限制了 HTTP 方法，然后使用 rewrite 指令将请求发送到了先前为该服务定义的内部通用代理位置：

```
location /api/service_1/object {
    limit_except GET PUT { deny all; }
    rewrite ^ /_service_1 last;
}
location /api/service_1/object/[^/]*$ {
    limit_except GET POST { deny all; }
    rewrite ^ /_service_1 last;
}
```

为每项服务重复执行此步骤。通过为用例有效组织文件和目录结构，进行逻辑划分。利用本书提供的信息，对 API location 代码块进行尽可能具体、严格的配置。

如果将单独文件用于上述 location 或 upstream 代码块，则需要确保已将它们包含到服务器上下文中：

```
server {
    listen 443 ssl;
    server_name api.company.com;
    # SSL 设置，请见第 7 章

    default_type application/json;

    include api_conf.d/*.conf;

}
```

利用第 6 章中描述的其中一种方法，或者使用下面的预共享 API 密钥（注意 map 指令只在 http 上下文中有效），启用身份验证来保护私有资源：

```
map $http_apikey $api_client_name {
    default "";

    "j7UqLLB+yRv2VTCXXDZ1M/N4" "client_one";
    "6B2kbyrrTiIN8S8JhSAxb63R" "client_two";
    "KcVgIDSY4Nm46m3tXVY3vbgA" "client_three";
}
```

利用第 2 章中介绍的使用限制方法，通过 NGINX 保护后端服务免遭攻击。在 http 上下文中，定义一个或多个请求限制共享内存区：

```
limit_req_zone $http_apikey
    zone=limitbyapikey:10m rate=100r/s;
limit_req_status 429;
```

使用速率限制和身份验证保护给定上下文：

```
location /api/service_2/object {
    limit_req zone=limitbyapikey;

    # 考虑将这些 if 语句写入文件中，
    # 并且需要使用 include。
    if ($http_apikey = "") {
        return 401;
    }
    if ($api_client_name = "") {
        return 403;
    }

    limit_except GET PUT { deny all; }
    rewrite ^ /_service_2 last;
}
```

测试对 API 网关的一些调用：

```
curl -H "apikey: 6B2kbyrrTiIN8S8JhSAxb63R" \
    https://api.company.com/api/service_2/object
```

详解

API 网关为接入应用程序编程接口（API）提供了入口点。这听起来有点笼统含糊，下面就让我们深入研究下许多不同层次上的集成点。需要通信（集成）的任何两个独立服务都应持有 API 版本契约。此类版本契约定义了服务的兼容性。API 网关会强制执行此类契约，在服务之间对请求进行身份验证、授权、转换和路由。

本节介绍了 NGINX 如何通过对传入请求进行验证、身份验证，以及将其定向到特定服务和限制其使用，发挥 API 网关的作用。该策略在微服务架构中广受欢迎，在微服务架构中，单个 API 产品会被分解给多个不同的服务。

现在，借助上述信息，您可以按照适合用例的确切规格来构建 NGINX 服务器配置。通过综合应用文中介绍的核心概念，您可以根据任何因素，对 URI 路径的使用进行身份验证和授权，路由或重写请求，限制使用，并定义有效请求。API 网关永远不会只有一款解决方案，因为其解决方案与用例紧密相关，具有无限可能性。

API 网关为运维团队和应用团队提供了一个终极协作空间，可帮助打造真正的 DevOps 组织。应用开发团队定义给定请求的有效性参数。而此类请求的交付通常由 IT 团队（网络、基础架构、安全和中间件团队）管理。API 网关充当这两层之间的接口。API 网关的构造需要来自各方的输入。这种配置应被置于某种源控制之下。许多现代源控制仓库都具有代码所有者的概念。此概念允许将某些文件设置为需要特定用户的批准。通过这种方式，团队不仅可以开展协作，还可以各自验证特定于各自部门的更改。

使用 API 网关时需要注意 URI 路径。在示例配置中，整个 URI 路径都被传递给上游服务器。这意味着 service_1 示例需要在 /api/service_1/* 路径上具有处理程序。如要以这种方式执行基于路径的路由，则最好是该应用与其他应用没有相互冲突的路由。

如果确实存在相互冲突的路由，则可以采取下列其中一种解决办法。通过编辑代码解决冲突，或者通过向一个或两个应用添加 URL 前缀配置，将其中一个应用移到另一个上下文中。对于无法编辑的现成软件，可以重写请求 URI upstream。但是，如果应用在消息体中返回链接，则需要使用正则表达式（regex）来重写请求的消息体，然后再将其提供给客户端 —— 应避免消息体中出现链接。

其他参考资料

《将 NGINX 部署为 API 网关》电子书

11.2 在 NGINX Plus 中使用 DNS SRV 记录

问题

希望将现有 DNS SRV 记录实施用作 NGINX Plus 上游服务器的源。

解决方案

在上游服务器上下文中将该服务指令的值设置为 http，以指示 NGINX 将 SRV 记录用作负载均衡池：

```
http {
    resolver 10.0.0.2 valid=30s;

    upstream backend {
        zone backends 64k;
        server api.example.internal service=http resolve;
    }
}
```

此功能为 NGINX Plus 独有。此配置使用 server 指令指示 NGINX Plus 解析来自位于 10.0.0.2 的 DNS 服务器的 DNS，并设置上游服务器池。server 指令的 resolve 参数指示该指令基于 DNS 记录 TTL 或 resolver 指令中的 valid override 参数定期重新解析域名。service=http 参数和值告诉 NGINX 该 SRV 记录包含 IP 和端口列表，并指示 NGINX 在这些 IP 和端口上进行负载均衡，仿佛它们是使用 server 指令配置的。

详解

随着对云基础架构的需求量和采用率的上升，动态基础架构正变得越来越受欢迎。自动缩放环境可横向缩放，增加或减少池中的服务器数量以满足负载需求。横向缩放需要一个可以向池中添加或从池中删除资源的负载均衡器。借助 SRV 记录，可以将保存服务器列表的责任转交给 DNS。对于容器化环境来说，这种配置别具吸引力，因为可能有容器在可变端口号上运行应用，也可能有容器在同一 IP 地址上运行应用。需要注意的是，UDP DNS 记录有效载荷限制为 512 个字节左右。

11.3 使用官方 NGINX 镜像

问题

需要使用来自 Docker Hub 的 NGINX 镜像，以快速启动并运行。

解决方案

使用来自 Docker Hub 的 NGINX 镜像。该镜像包含默认配置。如需要更改配置，要么挂载本地配置目录，要么为镜像构建 Dockerfile 和 ADD 以更改配置。现在我们挂载一个卷，其中 NGINX 的默认配置提供静态内容来演示其单命令功能：

```
$ docker run --name my-nginx -p 80:80 \
    -v /path/to/content:/usr/share/nginx/html:ro -d nginx
```

如果无法在本地找到，docker 命令就会从 Docker Hub 中拉取 nginx:latest 镜像。然后，该命令会将此 NGINX 镜像作为 Docker 容器运行，将 localhost:80 映射到 NGINX 容器的端口 80。它还会将本地目录 */path/to/content/* 作为容器卷以只读模式安装到 */usr/share/nginx/html/*。默认 NGINX 配置将此目录作为静态内容提供。当指定从本地机器映射到容器时，则将本地机器上的端口或目录映射到容器上的端口或目录。

详解

NGINX 通过 Docker Hub 提供官方 Docker 镜像。官方 Docker 镜像支持您在 Docker 中快速启动并运行首选应用交付平台 NGINX。在本节中，我们在容器中仅使用一个命令就完成了 NGINX 的快速启动和运行！我们在此示例中使用的官方 NGINX Docker mainline 镜像是由 Debian Jessie Docker 镜像构建而来。不过，您也可以选择基于 Alpine Linux 的官方镜像。这些官方镜像的 Dockerfile 和源代码都可以在 GitHub 上找到。您可通过构建自己的 Dockerfile 并在 FROM 命令中指明所基于的官方镜像名称，扩展官方镜像。您还可以将 NGINX 配置目录作为 Docker 卷安装，以覆盖 NGINX 配置，而无需修改官方镜像。

其他参考资料

“官方 NGINX Docker 镜像”
“GitHub 上的 NGINX Docker 仓库”

11.4 创建 NGINX Dockerfile

问题

需要创建 NGINX Dockerfile，以创建 Docker 镜像。

解决方案

从首选发行版的 Docker 镜像的 FROM 命令行开始。使用 RUN 命令安装 NGINX。使用 ADD 命令添加 NGINX 配置文件。使用 EXPOSE 命令指示 Docker 暴露给定端口，或者在将镜像作为容器运行时手动执行此操作。将镜像实例化成容器时，使用 CMD 启

动 NGINX。需要在前台运行 NGINX。为此，需要使用 -g "daemon off;" 启动 NGINX，或将 daemon off; 添加到配置中。本示例将采用后一种办法，将 daemon off; 添加到主上下文中的配置文件内。还可以按需将 NGINX 配置更改为将访问日志记录到 */dev/stdout* 并将错误日志记录到 */dev/stderr*，从而将日志添加到 Docker 守护进程中，这支持您基于所选择的 Docker 日志驱动程序，更轻松地访问日志。

```
FROM centos:7

# 安装 EPEL 仓库以获取 NGINX 并安装 NGINX
RUN yum -y install epel-release && \
    yum -y install nginx

# 将本地配置文件添加到镜像中
ADD /nginx-conf /etc/nginx

EXPOSE 80 443

CMD ["nginx"]
```

目录结构如下：

```
.
├── Dockerfile
└── nginx-conf
    ├── conf.d
    │   └── default.conf
    ├── fastcgi.conf
    ├── fastcgi_params
    ├── koi-utf
    ├── koi-win
    ├── mime.types
    ├── nginx.conf
    ├── scgi_params
    ├── uwsgi_params
    └── win-utf
```

我选择将整个 NGINX 配置托管到此 Docker 目录中，以便于访问所有配置。只需在 Dockerfile 中编写一行命令，即可添加所有这些 NGINX 配置。

详解

当需要完全控制所安装的软件包和更新时，创建自己的 Dockerfile 大有帮助。通常的做法是保留自己的镜像仓库，以确保基础镜像稳定可靠，并且经过团队测试后才投入到生产环境中使用。

11.5 构建 NGINX Plus Docker 镜像

问题

需要构建 NGINX Plus Docker 镜像，以在容器化环境中运行 NGINX Plus。

解决方案

使用此 Dockerfile 来构建 NGINX Plus Docker 镜像。需要下载名为 *nginx-repo.crt* 和 *nginx-repo.key* 的 NGINX Plus 仓库证书，并将它们与此 Dockerfile 保存在同一目录中。随后，使用此 Dockerfile 完成 NGINX Plus 的其余安装工作以供使用，并将 NGINX 访问和错误日志链接到 Docker 日志收集器。

```
FROM debian:stretch-slim

LABEL maintainer="NGINX <docker-maint@nginx.com>"

# 从客户门户（https://cs.nginx.com）
# 下载证书和密钥，并复制到 build 上下文中

COPY nginx-repo.crt /etc/ssl/nginx/
COPY nginx-repo.key /etc/ssl/nginx/

# 安装 NGINX Plus
RUN set -x \
  && APT_PKG="Acquire::https::plus-pkgs.nginx.com::" \
  && REPO_URL="https://plus-pkgs.nginx.com/debian" \
  && apt-get update && apt-get upgrade -y \
  && apt-get install \
    --no-install-recommends --no-install-suggests\
    -y apt-transport-https ca-certificates gnupg1 \
  && \
  NGINX_GPGKEY=573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62;\
  found=''; \
  for server in \
    ha.pool.sks-keyservers.net \
    hkp://keyserver.ubuntu.com:80 \
    hkp://p80.pool.sks-keyservers.net:80 \
    pgp.mit.edu \
  ; do \
   echo "Fetching GPG key $NGINX_GPGKEY from $server"; \
   apt-key adv --keyserver "$server" --keyserver-options \
      timeout=10 --recv-keys "$NGINX_GPGKEY" \
    && found=yes \
    && break;\
  done;\
  test -z "$found" && echo >&2 \
    "error: failed to fetch GPG key $NGINX_GPGKEY" && exit 1; \
```

```
    echo "${APT_PKG}Verify-Peer "true";"\
      >> /etc/apt/apt.conf.d/90nginx \
    && echo \
      "${APT_PKG}Verify-Host "true";">>\
      /etc/apt/apt.conf.d/90nginx \
    && echo "${APT_PKG}SslCert     \
      "/etc/ssl/nginx/nginx-repo.crt";" >> \
      /etc/apt/apt.conf.d/90nginx \
    && echo "${APT_PKG}SslKey      \
      "/etc/ssl/nginx/nginx-repo.key";" >> \
        /etc/apt/apt.conf.d/90nginx \
    && printf \
      "deb ${REPO_URL} stretch nginx-plus" \
      > /etc/apt/sources.list.d/nginx-plus.list \
    && apt-get update && apt-get install -y nginx-plus \
    && apt-get remove --purge --auto-remove -y gnupg1 \
    && rm -rf /var/lib/apt/lists/*

  # 将请求日志转发到 Docker 日志收集器
  RUN ln -sf /dev/stdout /var/log/nginx/access.log \
    && ln -sf /dev/stderr /var/log/nginx/error.log

  EXPOSE 80
  STOPSIGNAL SIGTERM

  CMD ["nginx", "-g", "daemon off;"]
```

如要使用此 Dockerfile 构建 Docker 镜像，则在包含 Dockerfile 以及 NGINX Plus 仓库证书和密钥的目录中运行以下命令：

```
$ docker build --no-cache -t nginxplus .
```

此 docker build 命令使用 --no-cache 标记来确保无论何时进行此项构建，都能重新从 NGINX Plus 仓库中拉取 NGINX Plus 软件包以应用更新。如果允许在 NGINX Plus 上使用与先前构建相同的版本，则可以省略 --no-cache 标记。在本示例中，新 Docker 镜像被标记为 nginxplus。

详解

通过为 NGINX Plus 创建自己的 Docker 镜像，您可以按需配置 NGINX Plus 容器，并将其部署到任何 Docker 环境中。这支持您在容器化环境中充分利用 NGINX Plus 的强大功能和高级特性。此 Dockerfile 不使用 Dockerfile ADD 命令来添加配置，需要您手动添加配置。

其他参考资料

“使用 Docker 部署 NGINX 和 NGINX Plus”

11.6 使用 NGINX 中的环境变量

问题

需要使用 NGINX 配置内的环境变量，以便将同一容器镜像用于不同的环境。

解决方案

使用 ngx_http_perl_module 从您的环境设置 NGINX 中的变量：

```
daemon off;
env APP_DNS;
include /usr/share/nginx/modules/*.conf;
# ...
http {
    perl_set $upstream_app 'sub { return $ENV{"APP_DNS"}; }';
    server {
        # ...
        location / {
            proxy_pass https://$upstream_app;
        }
    }
}
```

如要使用 perl_set，则必须安装 ngx_http_perl_module；如果从源代码构建，则可通过动态或静态加载该模块，安装此模块。默认情况下，NGINX 会从其环境中擦除环境变量；您需要使用 env 指令声明不希望删除的任何变量。perl_set 指令包含两个参数：要设置的变量名称和提供结果的 Perl 字符串。

以下 Dockerfile 动态加载 ngx_http_perl_module，并从软件包管理实用程序中安装此模块。从面向 CentOS 的软件包实用程序中安装模块时，这些模块会被放置在 */usr/lib64/nginx/modules/* 目录中，而动态加载这些模块的配置文件则会被放置在 */usr/share/nginx/modules/* 目录中。这就是为什么在前面的配置片段中，我们将所有配置文件都保存到了此路径：

```
FROM centos:7

# 安装 EPEL 仓库以获取 NGINX 并安装 NGINX
RUN yum -y install epel-release && \
    yum -y install nginx nginx-mod-http-perl

# 将本地配置文件添加到镜像中
ADD /nginx-conf /etc/nginx
```

```
EXPOSE 80 443

CMD ["nginx"]
```

详解

一种典型的 Docker 用法是利用环境变量来改变容器的运行方式。可以使用 NGINX 配置中的环境变量将 NGINX Dockerfile 用于多个不同的环境。

11.7 Kubernetes Ingress Controller（Kubernetes Ingress 控制器）

问题

正在 Kubernetes 上部署应用，需要一个 Ingress Controller（Ingress 控制器）。

解决方案

确保可以访问 Ingress Controller 镜像。对于 NGINX，可以使用来自 Docker Hub 的 *nginx/nginx-ingress* 镜像。而对于 NGINX Plus，则需要构建自己的镜像并将其托管在私有 Docker 仓库中。可以在 NGINX 文档中查找有关如何构建和推送 NGINX Plus Kubernetes Ingress Controller 的说明。

访问 GitHub 上 kubernetes-ingress 仓库中的 Kubernetes Ingress Controller 部署文件夹。以下命令均从仓库本地副本所在的目录运行。

为 Ingress Controller 创建命名空间和服务（service）帐户；两者均命名为 nginx-ingress:

```
$ kubectl apply -f common/ns-and-sa.yaml
```

使用 TLS 证书和密钥为 Ingress 控制器加密：

```
$ kubectl apply -f common/default-server-secret.yaml
```

该证书和密钥为自签名，由 NGINX Inc. 创建，用于测试和示例目的。建议使用自己的密钥，因为此密钥已公开。

或者，可以创建一个用于自定义 NGINX 配置的配置映射（所提供的配置映射为空；有关更多信息，请参阅“ConfigMap”和“注解”自定义）：

```
$ kubectl apply -f common/nginx-config.yaml
```

如果在集群中启用基于角色的访问控制（RBAC），则创建集群角色并将其绑定到服务帐户。必须是集群管理员才能执行此步骤：

```
$ kubectl apply -f rbac/rbac.yaml
```

现在部署 Ingress Controller。此仓库中提供了两个示例部署：Deployment 和 DaemonSet。如要动态更改 Ingress Controller 副本的数量，则使用 Deployment。使用 Daemonset 在每个节点或节点子集上部署 Ingress Controller。

如要使用 NGINX Plus Deployment 清单，则必须更改 YAML 文件并指定自己的仓库和镜像。

对于 NGINX Deployment：

```
$ kubectl apply -f deployment/nginx-ingress.yaml
```

对于 NGINX Plus Deployment：

```
$ kubectl apply -f deployment/nginx-plus-ingress.yaml
```

对于 NGINX DaemonSet：

```
$ kubectl apply -f daemon-set/nginx-ingress.yaml
```

对于 NGINX Plus DaemonSet：

```
$ kubectl apply -f daemon-set/nginx-plus-ingress.yaml
```

验证 Ingress 控制器正在运行：

```
$ kubectl get pods --namespace=nginx-ingress
```

如果创建了一个 DaemonSet，Ingress Controller 的端口 80 和 443 就会被映射到容器所在节点上的相同端口。如要访问 Ingress Controller，则使用 Ingress Controller 所在节点上的端口和 IP 地址。如果部署了 Deployment，则继续执行后续步骤。

对于 Deployment 方法，有两种 Ingress Controller Pod 访问方法。一种方法是指示 Kubernetes 随机分配映射到 Ingress Controller Pod 的节点端口。这是使用 NodePort 类型创建服务。另一方法是使用 LoadBalancer 类型创建服务。当创建 LoadBalancer 类型服务时，Kubernetes 会为给定云平台构建负载均衡器，例如 Amazon Web Services（AWS）、Microsoft Azure 和 Google Cloud Compute。

如要创建 NodePort 类型服务，则使用以下命令：

```
$ kubectl create -f service/nodeport.yaml
```

如要静态地配置面向 pod 打开的端口，则更改 YAML 并将 nodePort: {port} 属性添加到每个打开的端口。

如要为 Google Cloud Compute 或 Azure 创建 loadBalancer 类型服务，则使用以下代码：

```
$ kubectl create -f service/loadbalancer.yaml
```

如要为 Amazon Web Services 创建 LoadBalancer 类型服务，则使用以下代码：

```
$ kubectl create -f service/loadbalancer-aws-elb.yaml
```

在 AWS 上，Kubernetes 启用代理协议，在 TCP 模式下创建典型的 ELB。必须将 NGINX 配置为使用代理协议。为此，可以将以下内容添加到前面提到的与 *common/nginx-config.yaml* 文件相关的 config 映射中：

```
proxy-protocol: "True"
real-ip-header: "proxy_protocol"
set-real-ip-from: "0.0.0.0/0"
```

然后，更新配置映射：

```
$ kubectl apply -f common/nginx-config.yaml
```

现在，可通过两种方式访问 pod：通过其 NodePort ，或通过向代表其创建的负载均衡器发起请求。

详解

在撰写本文时，Kubernetes 是容器编排和管理领域的领先平台。Ingress Controller 是将流量路由到应用其余部分的边缘 pod。NGINX 非常适合这个角色，并且可通过注解简化配置。NGINX Ingress 项目通过 Docker Hub 镜像提供开箱即用的 NGINX 开源版 Ingress Controller，NGINX Plus 只需简单几步即可添加仓库证书和密钥。通过在 Kubernetes 集群中安装 NGINX Ingress Controller，您不仅可以畅享 NGINX 的所有特性，还可以使用 Kubernetes 联网和 DNS 路由流量等新增特性。

11.8 Prometheus Exporter 模块

问题

正在使用 Prometheus 监控将 NGINX 部署到环境中，需要 NGINX 统计数据。

解决方案

使用 NGINX Prometheus Exporter 获取 NGINX 或 NGINX Plus 统计数据，并将这些数据发送给 Prometheus。

NGINX Prometheus Exporter 模块使用 GoLang 编写而成，在 GitHub 上作为二进制文件分发，也在 Docker Hub 上作为预构建 Docker 镜像提供。

默认情况下，为 NGINX 启动该 exporter 程序，并且该 exporter 程序只收集 stub_status 信息。如要为 NGINX 开源版运行该 exporter 程序，首先需确保启用 stub 状态（如未启用，请参阅实操指南 13.1，详细了解如何启用）。然后运行以下 Docker 命令：

```
docker run -p 9113:9113 nginx/nginx-prometheus-exporter:0.8.0 \
    -nginx.scrape-uri http://{nginxEndpoint}:8080/stub_status
```

如要将该 exporter 程序与 NGINX Plus 一起使用，以利用 NGINX Plus API 收集更多数据，则必须使用标记来切换该 exporter 程序的上下文。可参阅实操指南 13.2，了解如何打开 NGINX Plus API。使用以下 Docker 命令为 NGINX Plus 环境运行该 exporter 程序：

```
docker run -p 9113:9113 nginx/nginx-prometheus-exporter:0.8.0 \
    -nginx.plus -nginx.scrape-uri http://{nginxPlusEndpoint}:8080/api
```

详解

Prometheus 是一种极为常见的指标监控解决方案，在 Kubernetes 生态系统中应用广泛。NGINX Prometheus Exporter 模块虽然是一个极其简单的组件，但支持在 NGINX 和常用监控平台之间进行预构建集成。使用 NGINX 时，stub 状态提供的数据虽不多，但很重要，可帮助了解 NGINX 节点当前的工作量。利用 NGINX Plus API 可以收集更多有关 NGINX Plus 服务器的统计数据，exporter 程序会将所有这些数据都发送给 Prometheus。无论哪种情况，收集的信息都可用作宝贵的监控数据，而且都会被发送给 Prometheus，您只需接通电源，坐享 NGINX 统计数据提供的洞察即可。

其他参考资料

“NGINX Prometheus Exporter GitHub”
NGINX stub_status 模块文档
NGINX Plus API 模块文档
“NGINX Plus 监控仪表盘”

11.9 使用 NGINX Secure Service Mesh 实现 mTLS

问题

希望使用 NGINX Secure Service Mesh 在服务之间实现双向 TLS（mTLS）身份验证。

解决方案

使用 nginx-meshcli 工具在宽容模式下实现 mTLS，同时验证：

```
nginx-meshctl deploy ...--mtls-mode permissive
```

通过从磁盘创建 YAML 配置文件指定 root 证书颁发机构，配置适当的公钥基础架构（PKI）。SPIRE 将此配置用于其上游权威机构：

```
apiVersion: v1
upstreamAuthority: disk
config:
    cert_file_path: /path/to/rootCA.crt
    key_file_path: /path/to/rootCA.key
```

如果在 AWS 上进行部署，则使用 AWS Secrets Manager 插件从 AWS Secrets Manager 拉取证书颁发机构证书和密钥：

```
apiVersion: "v1"
upstreamAuthority: "awssecret"
config:
    region: "us-east-1"
    cert_file_arn: "arn:aws:secretsmanager:us-east-1:123456789012:secret:
        /certificate-authority/test-certificate"
    key_file_arn: "arn:aws:secretsmanager:us-east-1:123456789012:secret:
        /certificate-authority/test-key"
```

使用下列内容部署 mTLS 配置，向 SPIRE 告知上游权威机构 nginx-meshcli:

```
nginx-meshctl deploy ...--mtls-upstream-ca-conf /path/to/upstream_authority.yaml
```

对部署进行验证：

```
kubectl get pods -n nginx-mesh
NAME                READY   STATUS    RESTARTS   AGE
...
spire-agent-sv2tv   1/1     Running   0          2h
spire-server-0      2/2     Running   0          2h
...
```

验证完 mTLS 配置，并确认应用按预期运行后，建议启用严格的 mTLS 模式：

```
nginx-meshctl deploy ...--mtls-mode strict
```

详解

Service mesh（服务网格）是用于处理分布式应用之间通信的专用基础架构层。NGINX Service Mesh 在 Kubernetes 中使用服务级 sidecar 模式。NGINX sidecar 处理服务间通信、监控和安全性。NGINX Service Mesh 和 nginx-meshctl 工具支持轻松为环境部署 service mesh。SideCar 注入能够自动为每个服务配置 NGINX sidecar，开启此功能后，您只需继续专注于核心服务即可，无需从头部署此基础结构层实施所需的广泛配置。

本节聚焦于 mTLS 的实现。mTLS 在客户端和服务器之间提供身份验证 —— 客户端和服务器通过经双方加密的连接，相互验证彼此的身份。在我们的 service mesh 中，任一给定服务也可以是另一服务的客户端，因为这些服务可相互通信。

本节介绍了如何在宽容模式下启用 NGINX Service Mesh，以便发生配置错误时，通信可恢复正常状态。在此阶段，作为负责对证书和密钥进行签名和分发的服务，SPIRE 默认生成自签名证书。然后，配置可通过提供特定证书颁发机构（CA）证书和密钥对来得到增强。本节介绍了提供 CA 密钥对的两种方式：一种是直接通过磁盘，另一种是通过 AWS Secrets Manager。如果您正好在使用 AWS Certificate Manager 来支持私有 CA，那么也可以指示 SPIRE 使用 AWS ACM 对服务证书进行签名。配置完成后，验证配置，然后启用 mTLS 严格模式，这要求通过 mTLS 对所有服务间通信进行身份验证。

其他参考资料

“NGINX Service Mesh 中的 mTLS 架构”
“使用 mTLS 保护网格流量”
企业级服务网格架构之路（O'Reilly）

第 12 章
高可用性部署模式

12.0 简介

容错架构将系统分成多个相同的独立堆栈。使用 NGINX 等负载均衡器来分发负载，以优化配置。高可用性的核心理念是在多个活跃节点上进行负载均衡或 active-passive 故障转移。高可用性应用不会出现单点故障；每个组件，包括负载均衡器本身，都必须应用上述其中一种理念。对我们来说，这就意味着对 NGINX 实现高可用。NGINX 经专门设计，可配置为 active-active 或 active-passive 故障转移模式。本章详细介绍了如何运行多台 NGINX 服务器以确保负载均衡层的高可用性。

12.1 NGINX Plus HA（高可用性）模式

问题

需要高可用性负载均衡解决方案。

解决方案

从 NGINX Plus 仓库中安装 nginx-ha-keepalived 包，利用 keepalived 实现 NGINX Plus 的 HA 模式。

详解

nginx-ha-keepalived 包基于 keepalived，管理暴露给客户端的虚拟 IP 地址。在 NGINX 服务器上运行的另一个进程可确保 NGINX Plus 和 keepalived 进程保持运行。Keepalived 进程使用虚拟路由器冗余协议（VRRP）将短消息（通常被称为心跳消息）发送到备份服务器。如果备份服务器连续三个周期接收不到心跳消息，备份服

务器就会启动故障转移，将虚拟 IP 地址切换到自己这里并成为主服务器。nginx-ha-keepalived 的故障转移功能经配置，可识别自定义故障情形。

12.2 通过 DNS 实现负载均衡器的负载均衡

问题

需要在两台或多台 NGINX 服务器之间分发负载。

解决方案

通过向 DNS A 记录中添加多个地址，使用 DNS 轮询 NGINX 服务器。

详解

运行多个负载均衡器时，可通过 DNS 分发负载。A 记录允许在单个 FQDN 下列出多个 IP 地址。DNS 将自动轮询所有列出的 IP。DNS 还为加权记录提供加权轮询，其工作方式与第 2 章中所描述的 NGINX 的加权轮询工作方式相同。这些功能帮助极大。但是，当 NGINX 服务器处理请求遇到故障时，记录可能不会被删除，这是个问题。DNS 提供商（Amazon Route 53 和 Dyn DNS）通过其 DNS 产品提供健康检查和故障转移功能，可帮助缓解这些问题。如果使用 DNS 帮助 NGINX 进行负载均衡，那么当某台 NGINX 服务器被标记为删除时，最好在删除上游服务器时遵循 NGINX 所使用的协议。首先，通过从 DNS 记录中删除服务器的 IP 地址，停止向其发送新连接，然后在停止或关闭服务之前允许连接清空。

12.3 在 EC2 上实现负载均衡

问题

正在 AWS 上使用 NGINX，而 NGINX Plus HA（高可用性）模式不支持 Amazon IP。

解决方案

配置 NGINX 服务器的 Auto Scaling 组并将 Auto Scaling 组链接到目标组，然后将目标组连接到 NLB，从而将 NGINX 置于 AWS NLB 之后。或者，也可以使用 AWS 控制台、命令行接口或 API，手动将 NGINX 服务器添加到目标组中。

详解

由于 EC2 IP 地址以不同的方式使用，AWS 不支持浮动虚拟 IP 地址，所以基于 keepalived 的 NGINX Plus HA 解决方案将无法在 AWS 上使用。这并非意味着 NGINX 无法在 AWS 云中实现高可用性；实际上，事实正好相反。亚马逊产品 AWS NLB 可以在多个被称作*可用区*的在物理上分隔的数据中心之间进行本地负载均衡，提供主动健康检查，并提供 DNS CNAME 端点。在 AWS 上实现 NGINX 高可用性的一种常见解决方案是将 NGINX 层置于 NLB 之后。NGINX 服务器可按需自动添加到目标组中或从目标组中删除。NLB 无法替代 NGINX；NGINX 可提供许多 NLB 无法提供的功能，例如多种负载均衡方法、速率限制、高速缓存和七层路由。尽管 AWS ALB 能够基于 URI 路径和主机请求头执行七层负载均衡，但是无法提供 WAF 高速缓存、带宽限制、HTTP/2 服务器推送等 NGINX 可提供的功能。如果 NLB 无法满足需求，还有许多其他解决方案可供选择。其中一种选择是 DNS 解决方案：AWS Route 53 提供健康检查和 DNS 故障转移。

12.4 NGINX Plus 配置同步

问题

正在运行 HA（高可用性）NGINX Plus 层，需要实现服务器配置同步。

解决方案

使用 NGINX Plus 独有的配置同步功能。该功能的配置步骤如下：

从 NGINX Plus 软件包仓库中安装 nginx-sync 包。

对于 RHEL 或 CentOS：

```
$ sudo yum install nginx-sync
```

对于 Ubuntu 或 Debian：

```
$ sudo apt-get install nginx-sync
```

授予主机以 root 身份 SSH 访问对等机。

为 root 权限生成 SSH 身份验证密钥对，并检索公钥：

```
$ sudo ssh-keygen -t rsa -b 2048
$ sudo cat /root/.ssh/id_rsa.pub
ssh-rsa AAAAB3Nz4rFgt...vgaD root@node1
```

获取主节点的 IP 地址：

```
$ ip addr
1: lo: mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: mtu 1500 qdisc pfifo_fast state UP group default qlen \ 1000
    link/ether 52:54:00:34:6c:35 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.2/24 brd 192.168.1.255 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::5054:ff:fe34:6c35/64 scope link
        valid_lft forever preferred_lft forever
```

ip addr 命令将转储机器接口信息。跳过环回接口，它通常显示在最前面。查找主接口 inet 后面的 IP 地址。在本示例中，IP 地址是 192.168.1.2。

将公钥分发到 root 用户在每个对等节点上的 *authorized_keys* 文件，并指定仅从主 IP 地址授权：

```
$ sudo echo 'from="192.168.1.2" ssh-rsa AAAAB3Nz4rFgt...vgaD \
    root@node1' >> /root/.ssh/authorized_keys
```

将以下命令行添加到 */etc/ssh/sshd_config* 中并在所有节点上重载 sshd：

```
$ sudo echo 'PermitRootLogin without-password' >> \
    /etc/ssh/sshd_config
$ sudo service sshd reload
```

验证主节点上的 root 用户能否对每个对等节点进行 ssh 免密登录：

```
$ sudo ssh root@node2.example.com
```

在主机上使用以下配置创建配置文件 */etc/nginx-sync.conf*：

```
NODES="node2.example.com node3.example.com node4.example.com"
CONFPATHS="/etc/nginx/nginx.conf /etc/nginx/conf.d"
EXCLUDE="default.conf"
```

本示例配置演示了该功能的三个常用配置参数：NODES、CONFIGPATHS 和 EXCLUDE。NODES 参数被设置为用空格隔开的主机名或 IP 地址的字符串；主节点会将其配置更改推送给对等节点。CONFIGPATHS 参数表示应同步的文件或目录。最后，可以使用 EXCLUDE 参数将配置文件从同步进程中删除。在我们的示例中，主节点向名为 node2.example.com、node3.example.com 和 node4.example.com 的对等节点推送主要 NGINX 配置文件的配置更改并添加目录 */etc/nginx/nginx.conf* 和 */etc/nginx/conf.d*。如果同步进程发现名为 *default.conf* 的文件，由于该文件已被配置为 EXCLUDE，因此不会被推送给对等节点。

高级配置参数可以配置 NGINX 二进制文件、RSYNC 二进制文件、SSH 二进制文件、diff 二进制文件、lockfile 和备份目录的位置。还有一个参数使用 sed 为给定文件提供模板。如欲了解有关高级参数的更多信息，请参阅 “NGINX 集群配置同步” 。

对配置进行测试：

```
$ nginx-sync.sh -h # 显示使用信息
$ nginx-sync.sh -c node2.example.com # 与节点 2 比较配置
$ nginx-sync.sh -C # 与所有对等节点比较主要配置
$ nginx-sync.sh # 在所有对等节点上同步配置并重载 NGINX
```

详解

这项 NGINX Plus 独有功能可帮助高效管理 HA 配置中的多台 NGINX Plus 服务器，您只需更新主节点，将配置同步到所有其他对等节点即可。配置同步的自动化可降低在配置同步过程中发生错误的风险。为避免将不良配置发送给对等节点，*nginx-sync.sh* 应用提供了多项保障措施，包括测试主节点的配置，在对等节点上创建配置备份，以及在重载之前验证对等节点的配置。尽管最好使用配置管理工具或 Docker 对配置进行同步，但如果尚不能通过这些方式管理环境，NGINX Plus 配置同步特性将会是不错的选择。

12.5 与 NGINX Plus 的状态共享和区域同步

问题

需要 NGINX Plus 在大量高可用性服务器之间同步其共享内存区。

解决方案

配置区域同步，然后使用 sync 参数配置 NGINX Plus 共享内存区：

```
stream {
    resolver 10.0.0.2 valid=20s;

    server {
        listen 9000;
        zone_sync;
        zone_sync_server nginx-cluster.example.com:9000 resolve;
        # ...安全措施
    }
}
http {
    upstream my_backend {
        zone my_backend 64k;
        server backends.example.com resolve;
        sticky learn zone=sessions:1m
                     create=$upstream_cookie_session
                     lookup=$cookie_session
                     sync;
    }

    server {
        listen 80;
        location / {
            proxy_pass http://my_backend;
        }
    }
}
```

详解

zone_sync 模块是 NGINX Plus 的一项独有功能，能够让 NGINX Plus 真正实现集群。如配置所示，必须将一台 stream 服务器设为配置为 zone_sync。在本示例中，选用了侦听端口 9000 的服务器。NGINX Plus 与 zone_sync_server 指令定义的其他服务器通

信。可以为此指令设置一个域名，使域名解析到多个 IP 地址以构成动态集群，或者静态定义一系列 zone_sync_server 指令以避免单点故障。应限制对区域同步服务器的访问；可以为 zone_sync 模块设置特定 SSL/TLS 指令以进行机器身份验证。将 NGINX Plus 配置为集群的好处是，可以对共享内存区进行速率限制、粘性学习（sticky-learn）会话和键值（key-value）存储等方面的同步。在所提供的示例中，sticky learn 指令在末端添加了 sync 参数。在本示例中，用户基于名为 session 的 cookie 绑定到上游服务器。在没有 zone_sync 模块的情况下，如果用户向不同的 NGINX Plus 服务器发出请求，则可能会丢失会话。如果存在 zone_sync 模块，所有 NGINX Plus 服务器都将能够感知到会话及其绑定到的上游服务器。

第 13 章
高级活动监控

13.0 简介

为了确保应用以最佳性能和精度运行，您需要清晰地了解有关其活动的监控指标。NGINX Plus 提供高级监控仪表盘和 JSON feed，支持对所有传入应用的请求进行深度监控。NGINX Plus 活动监控提供有关请求、上游服务器池、高速缓存、健康状态等的洞察。本章详细介绍了 NGINX Plus 仪表盘、NGINX Plus API 和开源 stub 状态模块的功能以及它们所带来的可能性。

13.1 启用 NGINX 开源版的 stub 状态

问题

需要对 NGINX 进行基本的监控。

解决方案

启用 NGINX HTTP 服务器内 location 代码块中的 stub_status 模块：

```
location /stub_status {
    stub_status;
    allow 127.0.0.1;
    deny all;
    # Set IP restrictions as appropriate
}
```

通过向 stub 状态发送请求来测试配置：

```
$ curl localhost/stub_status
Active connections: 1
server accepts handled requests
 1 1 1
Reading: 0 Writing: 1 Waiting: 0
```

详解

stub_status 模块支持对 NGINX 开源版服务器进行一些基本的监控。返回的信息包括活跃连接数、已接受的连接总数、已处理的连接数和服务请求数。此外，还显示了当前正处于读入、写入或等待状态的连接数。所提供的信息为全局信息，并非特定于定义 stub_status 指令的父 server。这意味着可以将 stub 状态托管在受保护的 server 上。出于安全原因，除本地流量以外，我们拦截了对监控功能的所有访问。该模块将活跃连接数作为嵌入式变量提供，以用于日志和其他地方。这些变量包括 $connections_active、$connections_reading、$connections_writing 以及 $connections_waiting。

13.2 启用 NGINX Plus 监控仪表盘

问题

需要有关流经 NGINX Plus 服务器的流量的深度指标。

解决方案

使用实时活动监控仪表盘：

```
server {
    # ...
    location /api {
        api [write=on];
        # Directives limiting access to the API
        # See chapter 7
    }

    location = /dashboard.html {
        root   /usr/share/nginx/html;
    }
}
```

NGINX Plus 配置提供 NGINX Plus 状态监控仪表盘。此配置设置了 HTTP 服务器，以提供 API 和状态仪表盘。状态仪表盘从 */usr/share/nginx/html* 目录中作为静态内容提供。该仪表盘会向位于 */api/* 的 API 发起请求，以实时检索和显示状态。

详解

NGINX Plus 提供高级状态监控仪表盘。此状态仪表盘提供有关 NGINX 系统状态的详细信息，例如活跃连接数、正常运行时间、上游服务器池信息等。有关该控制台的图示，请参见图 13-1。

NGINX Controller ADC 为监控分布于许多不同位置的 NGINX Plus 服务器提供了一个以应用为中心的视图。Infrastructure（基础架构）选项卡将有关服务器、环境和应用的信息和指标都集中到了单个界面上。有关该控制台的图示，请参见图 13-2。

状态仪表盘的登录界面提供了对整个系统的概览。点击 HTTP Zones（HTTP 区域）选项卡，即可获得有关 NGINX 配置中所有 HTTP 服务器的详细信息，包括 1xx~5xx 响应数和总响应数，以及每秒请求数和当前吞吐量。HTTP Upstreams（HTTP 上游）选项卡显示了有关上游服务器状态的详细信息，包括服务器故障期间服务的请求数、状态代码处理的响应数，以及其他统计信息，例如健康检查通过或失败次数。TCP/UDP Zones（TCP/UDP 区域）选项卡详细显示了流经 TCP 或 UDP 串流的流量和连接数。TCP/UDP Upstreams（TCP/UDP 上游）选项卡显示了 TCP/UDP 上游池中每台上游服务器所服务的请求数，以及健康检查通过和失败的详细信息和响应时间。Caches（高速缓存）选项卡显示了有关高速缓存占用空间、已服务、写入和绕过流量以及命中率的信息。NGINX 状态仪表盘在监控应用和流量的关键指标方面发挥着重要作用。

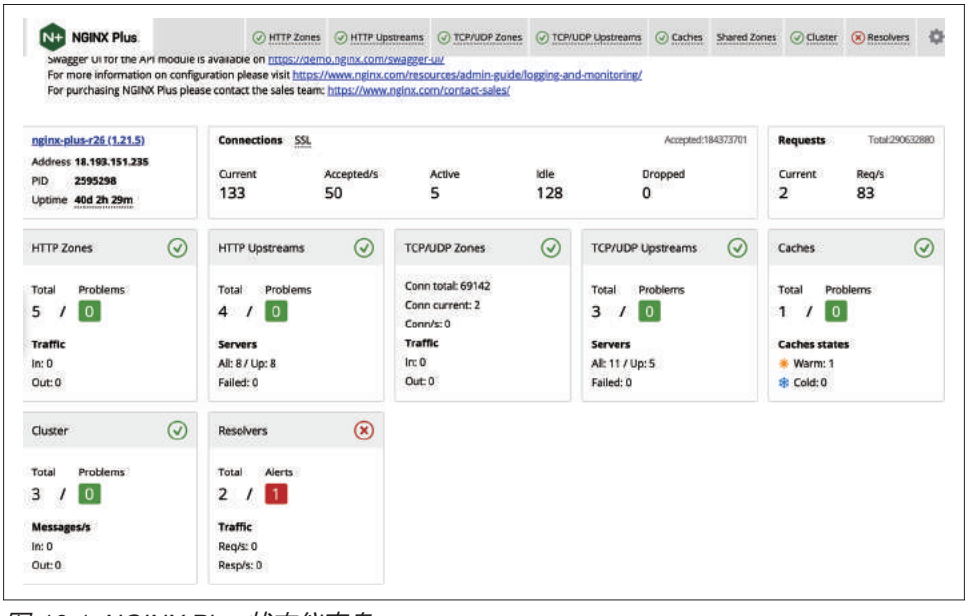


图 13-1. NGINX Plus 状态仪表盘

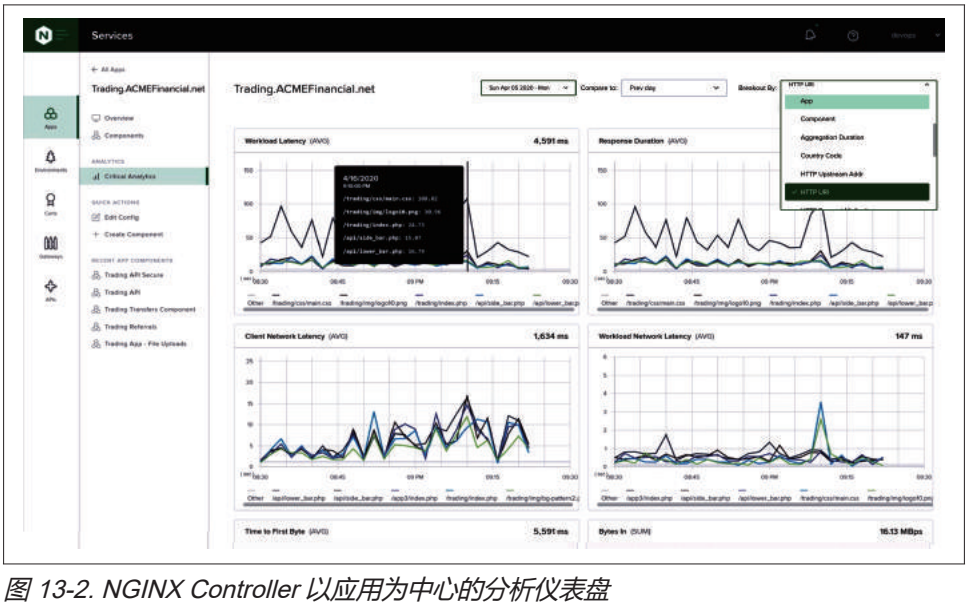


图 13-2. NGINX Controller 以应用为中心的分析仪表盘

其他参考资料

NGINX Plus 状态仪表盘演示
F5 NGINX Controller 产品页面（NGINX Controller 已于 2022 年正式停售，点击了解相关新产品 NGINX Management Suite）

13.3 使用 NGINX Plus API 收集指标

问题

需要通过 API 访问 NGINX Plus 状态仪表盘提供的详细指标。

解决方案

利用 RESTful API 收集指标。该示例通过 json_pp 导出输出结果，以便于读取：

```
$ curl "demo.nginx.com/api/3/" | json_pp
[
    "nginx",
    "processes",
    "connections",
    "ssl",
    "slabs",
    "http",
    "stream"
]
```

curl 调用向 API 顶层发起请求，以显示 API 的其他部分。

如要获取有关 NGINX Plus 服务器的信息，则使用 /api/{version}/nginx URI:

```
$ curl "demo.nginx.com/api/3/nginx" | json_pp
{
    "version" : "1.15.2",
    "ppid" : 79909,
    "build" : "nginx-plus-r16",
    "pid" : 77242,
    "address" : "206.251.255.64",
    "timestamp" : "2018-09-29T23:12:20.525Z",
    "load_timestamp" : "2018-09-29T10:00:00.404Z",
    "generation" : 2
}
```

如要限制 API 返回的信息，则使用下列参数：

```
$ curl "demo.nginx.com/api/3/nginx?fields=version,build" \ | json_pp
{
    "build" : "nginx-plus-r16",
    "version" : "1.15.2"
}
```

可以从 /api/{version}/connections URI 获取请求连接数统计：

```
$ curl "demo.nginx.com/api/3/connections" | json_pp
{
    "active" : 3,
    "idle" : 34,
    "dropped" : 0,
    "accepted" : 33614951
}
```

可以从 /api/{version}/http/requests URI 收集请求数：

```
$ curl "demo.nginx.com/api/3/http/requests" | json_pp
{
    "total" : 52107833,
    "current" : 2
}
```

可以使用 /api/{ver sion}/http/server_zones/{httpServerZoneName} URI 检索有关特定服务器区域的统计数据：

```
$ curl "demo.nginx.com/api/3/http/server_zones/hg.nginx.org" \ | json_pp
{
    "responses" : {
        "1xx" : 0,
        "5xx" : 0,
        "3xx" : 938,
        "4xx" : 341,
        "total" : 25245,
        "2xx" : 23966
    },
    "requests" : 25252,
    "discarded" : 7,
    "received" : 5758103,
    "processing" : 0,
    "sent" : 359428196
}
```

仪表盘可清晰地显示 API 返回的所有数据。这些数据富有深度，按逻辑有序呈现。可以在本节实操指南的结尾找到有关资源的链接。

详解

NGINX Plus API 可返回有关 NGINX Plus 服务器的多方面统计信息。您可以收集有关 NGINX Plus 服务器及其进程、连接和 slab 的信息。还可以查看有关 NGINX 内 http 和流媒体服务器的信息，包括服务器、上游、上游服务器和键值存储，以及有关 HTTP 缓存区的信息和统计数据。这能够为您或第三方指标聚合器提供有关 NGINX Plus 服务器状态的深入视图。

借助 NGINX Instance Manager API，可以一次查询多台 NGINX Plus 服务器的指标。NGINX Controller 可提供一种独特的以应用为中心的指标视图。

其他参考资料

NGINX HTTP API 模块文档
NGINX API REST UI
“指标 API 使用”教程

第 14 章

利用访问日志、错误日志和请求跟踪进行调试和故障排除

14.0 简介

日志记录是了解应用的基础。使用 NGINX 您可有效控制对您和您的应用有意义的日志信息。NGINX 允许您根据不同的上下文将日志以不同的格式拆分到不同的文件中，并更改错误日志的日志级别，以更深入地了解当前状况。NGINX 支持对接 Syslog，天生就具备了将日志以流的形式输出给集中式日志服务器的能力。NGINX 和 NGINX Plus 还支持为请求发起全链路跟踪。在本章中，我们将介绍访问和错误日志，基于 Syslog 协议的流式传输，以及利用 NGINX 和 OpenTracing 所生成的请求标识符的实现端到端请求跟踪。

14.1 配置访问日志

问题

需要配置访问日志格式，以将内置变量添加到请求日志中。

解决方案

配置访问日志格式：

```
http {
    log_format geoproxy
               '[$time_local] $remote_addr '
               '$realip_remote_addr $remote_user '
               '$proxy_protocol_server_addr $proxy_protocol_server_port '
               '$request_method $server_protocol '
               '$scheme $server_name $uri $status '
               '$request_time $body_bytes_sent '
               '$geoip_city_country_code3 $geoip_region '
               '"$geoip_city" $http_x_forwarded_for '
               '$upstream_status $upstream_response_time '
               '"$http_referer" "$http_user_agent"';
    # ...
}
```

该日志格式配置被命名为 geoproxy，使用了大量内置变量来演示 NGINX 日志记录功能。该配置可输出发起请求时服务器上的本地时间、打开连接的 IP 地址，以及客户端的 IP 地址，因为 NGINX 根据 geoip_proxy 或 real_header 指令知道这些信息。

当 server 下的 listen 指令使用了 proxy_protocol 参数时，带 $proxy_protocol_server_ 前缀的变量可用于从 PROXY 协议头中获取服务端相关信息。$remote_user 表示通过 Basic 认证的用户名；后面还包含请求方法、协议版本和协议类型，例如 HTTP 或 HTTPS。当然也记录了匹配的 server 上下文名称、请求 URI 和响应状态码。

记录的统计信息包括以毫秒为单位的处理时间以及发送给客户端的消息体大小。此外，有关国家、地区和城市的信息也被记录在内。X-Forwarded-For HTTP 请求头可表明请求是否来自其它代理服务器转发。upstream 模块支持一些用于显示上游服务器响应状态和上游请求响应时间的内置变量。最后，有关客户端请求来源和客户端所用浏览器的信息也被记录在内。

log_format 指令仅在 HTTP 上下文中有效。可选参数 escape 可指定字符串转义类型，可选值有：default、json 和 none。none 表示禁用转义字符。default 时，字符 "、\ 以及其他值小于 32 或大于 126 的字符用 \xXX 来转义。如果找不到变量值，则使用连字符（-）替代。对于 json 转义，JSON 字符串中所有不允许的字符都将被转义，如：字符 " 和 \ 用 \" 和 \\ 来表示，值小于 32 的字符用 \n、\r、\t、\b、\f 或 \u00XX 来表示。

此日志配置的日志条目如下所示：

```
[25/Nov/2016:16:20:42 +0000] 10.0.1.16 192.168.0.122 Derek
GET HTTP/1.1 http www.example.com / 200 0.001 370 USA MI
"Ann Arbor" - 200 0.001 "-" "curl/7.47.0"
```

如要使用此日志格式，则使用 access_log 指令，并将日志文件路径和格式名称 geoproxy 用作参数：

```
server {
    access_log /var/log/nginx/access.log geoproxy;
    # ...
}
```

access_log 指令使用日志文件路径和格式名称作为参数。该指令在许多上下文中都有效，可根据上下文采用不同的日志路径和/或日志格式。buffer、flush 和 gzip 等参数可配置日志写入日志文件的频率以及文件是否被压缩。还可通过 if 指令设置输出条件，如果条件值为 0 或空字符串，则访问将不被记录。

详解

NGINX 中的 log 模块允许您为许多不同的场景配置日志格式，以根据需要记录到不同日志文件中。在实际运用中，为不同上下文配置不同的日志格式非常有用，您可以在日志中使用不同的模块和内置变量，或者集中提供所有必要信息。日志也能记录以 JSON 格式和 XML 格式创建的字符串。这些日志将帮助您了解流量模式、客户端使用情况和客户端来源等。此外，访问日志还可以帮助您发现与上游服务器或特定 URI 相关的响应延迟和问题。访问日志还可用于在测试环境中解析和回放流量模式，以模仿真实的用户交互。日志在应用故障排除、调试或市场分析中发挥着至关重要的作用。

14.2 配置错误日志

问题

需要配置错误日志，以更深入地了解 NGINX 服务器问题。

解决方案

使用 error_log 指令定义日志路径和日志级别：

```
error_log /var/log/nginx/error.log warn;
```

在配置 error_log 指令时，路径是必选的，但日志级别是可选的，默认为 error。除 if 语句以外，该指令在所有上下文中都有效。日志级别包括 debug（调试）、info（信息）、notice（注意）、warn（警告）、error（错误）、crit（严重）、alert（警报）和 emerg（紧急）。上述日志级别的排列顺序也是严重程度从低到高的顺序。debug 日志级别仅在 NGINX 配置了 --with-debug 标记时可用。

详解

当配置文件无法正常工作时，应首先查看错误日志。错误日志也是定位由 FastCGI 等应用服务器所产生的错误的利器。还可以使用错误日志来调试 worker 进程连接、内存分配、客户端 IP 和服务器。错误日志虽然不支持自定义日志格式，但也遵循特定格式 —— 先列出日期，随后列出日志级别和具体消息。

14.3 转发日志到 Syslog

问题

需要将日志转发到 Syslog 侦听器，以便将日志聚合到集中式日志服务中。

解决方案

使用 error_log 和 access_log 指令将日志发送到 Syslog 侦听器：

```
error_log syslog:server=10.0.1.42 debug;

access_log syslog:server=10.0.1.42,tag=nginx,severity=info geoproxy;
```

error_log 和 access_log 指令的 syslog 参数后面紧跟冒号和一些参数选项。这些选项包括必要的日志服务标识，如：需要连接的 IP 地址、DNS 名称或 Unix socket，以及 facility、severity、tag 和 nohostname 等可选标记。server 选项在指定指 IP 地址或 DNS 名称时设置端口号，默认为 UDP 514。facility 选项是指日志消息的 facility 值，取 Syslog RFC 标准定义的 23 个值之一；默认值为 local7。tag 选项表示消息的标记，默认值为 nginx。severity 选项默认为 info，表示所发送消息的日志级别。nohostname 标记禁止将 hostname 字段添加到 syslog 消息头中，不取值。

详解

Syslog 是一种在单台服务器或服务器集群上发送和收集日志消息的标准协议。在多个主机上运行相同服务的多个实例时，将日志发送到集中位置有助于调试。这被称作聚合日志。聚合日志允许您在一个地方查看日志，无需切换服务器，并通过时间戳将日志文件合并在一起。常见的日志聚合堆栈有 Elasticsearch、Logstash 和 Kibana，统称为 ELK 堆栈。NGINX 可使用 access_log 和 error_log 指令，轻松将这些日志传输到 Syslog 侦听器。

14.4 请求跟踪

问题

需要将 NGINX 日志与应用日志相关联，以了解请求的端到端处理过程。

解决方案

使用 request 的唯一识别变量，并将其传递给应用以进行记录：

```
log_format trace  '$remote_addr - $remote_user [$time_local] '
                  '"$request" $status $body_bytes_sent '
                  '"$http_referer" "$http_user_agent" '
                  '"$http_x_forwarded_for" $request_id';
upstream backend {
    server 10.0.0.42;
}
server {
    listen 80;
    # 将 X-Request-ID 标头添加到对客户端的响应中
    add_header X-Request-ID $request_id;
    location / {
        proxy_pass http://backend;
        # 将 X-Request-ID 标头发送到应用端
        proxy_set_header X-Request-ID $request_id;
        access_log /var/log/nginx/access_trace.log trace;
    }
}
```

在本示例配置中，配置了名为 trace 的 log_format，并在日志中使用了 $request_id 变量。还在发起上游请求时，通过使用 proxy_set_header 指令将请求 ID 添加到请求头

里，将此 $request_id 变量传递给了上游应用。此外，还通过使用 add_header 指令，在响应头中设置请求 ID，将请求 ID 传回客户端。

详解

该功能在 NGINX Plus R10 和 NGINX 1.11.0 版本中可用，$request_id 提供了一个随机生成的 32 位十六进制字符串，这些字符可以用来唯一性标识请求。通过将此标识符传递给客户端和应用，您可以将日志与请求相关联。在前端客户端，你可通过响应头收到该请求唯一性标识字符串，并且可以利用它在日志中搜索与请求相关联的条目。您还需要在应用日志中捕获和记录此请求唯一性标头，从而在日志之间建立真正的端到端关联。基于这项功能，NGINX 让跨应用栈跟踪请求变为可能。

14.5 用于 NGINX 的 OpenTracing

问题

拥有一台支持 OpenTracing 的跟踪服务器，需要与 NGINX 或 NGINX Plus 相集成。

解决方案

确保拥有一台兼容 OpenTracing 的服务器，并且在 NGINX 或 NGINX Plus 节点上安装了正确的客户端。

需要使用面向该特定服务器的插件配置文件。该解决方案将演示 Jaeger 和 Zipkin。

名为 */etc/jaeger/jaeger-config.json* 的 Jaeger 插件配置示例如下所示：

```
{
    "service_name": "nginx",
    "sampler": {
        "type": "const",
        "param": 1
    },
    "reporter": {
        "localAgentHostPort": "Jaeger-server-IP-address:6831"
    }
}
```

名为 */etc/zipkin/zipkin-config.json* 的 Zipkin 插件配置示例如下所示：

```
{
    "service_name": "nginx",
    "collector_host": "Zipkin-server-IP-address",
    "collector_port": 9411
}
```

如果使用 NGINX Plus，则按照《NGINX Plus OpenTracing 管理员指南》从 NGINX Plus 仓库安装 OpenTracing 模块。

如果使用开源 NGINX，则访问“NGINX OpenTracing 模块版本”页面，查找与系统兼容的预构建动态模块或使用源代码编译该 NGINX 模块。或者，在 Docker 环境中，可使用 Docker Hub 上名为 opentracing/nginx-opentracing 的镜像快速开始测试。

使用动态加载的模块（包括 NGINX Plus 安装）时，可通过添加下述 load_module 指令，向 NGINX 告知该模块在文件系统上的位置，来确保将其加载到 NGINX 配置中。值得注意的是，load_module 指令仅在主（顶级）上下文中有效。

```
load_module modules/ngx_http_opentracing_module.so;
```

当一个兼容 OpenTracing 的服务器开始侦听，客户端已安装到 NGINX 节点上，插件配置部署完毕，以及 NGINX 模块加载完毕后，即可对 NGINX 进行配置，以发起跟踪请求。以下代码提供了加载跟踪插件并将 NGINX 配置为标记请求的示例。其中包含加载跟踪插件的指令，而 Jaeger 和 Zipkin 插件及其配置文件的默认位置已在前面提供。可根据用例取消对适当厂商示例的注释。

```
# 加载厂商跟踪器
#opentracing_load_tracer /usr/local/libjaegertracing_plugin.so
#                        /etc/jaeger/jaeger-config.json;
#opentracing_load_tracer /usr/local/lib/libzipkin_opentracing_plugin.so
#                        /etc/zipkin/zipkin-config.json;

# 开启所有请求追踪
opentracing on;

# 设置捕捉 NGINX 变量值的附加标记
opentracing_tag bytes_sent $bytes_sent;
opentracing_tag http_user_agent $http_user_agent;
opentracing_tag request_time $request_time;
opentracing_tag upstream_addr $upstream_addr;
opentracing_tag upstream_bytes_received $upstream_bytes_received;
opentracing_tag upstream_cache_status $upstream_cache_status;
opentracing_tag upstream_connect_time $upstream_connect_time;
opentracing_tag upstream_header_time $upstream_header_time;
opentracing_tag upstream_queue_time $upstream_queue_time;
opentracing_tag upstream_response_time $upstream_response_time;
```

```
server {
    listen 9001;

    location / {
        # 用于 OpenTracing Span 的操作名
        # 默认为 location 代码块的名称，
        # 但可取消对此指令的注释，以进行自定义。
        #opentracing_operation_name $uri;

        # 传播活动 Span 上下文 upstream，
        # 以便后端可继续跟踪。
        opentracing_propagate_context;

        # 应用的 location 服务示例
        proxy_pass http://10.0.0.2:8080;
    }
}
```

详解

OpenTracing 设置并非无关紧要，相反它在性能和事务的分布式监控方面发挥着重要作用。这些工具支持团队高效发现根本原因和开展依赖性分析，利用数据精准定位问题。NGINX 可以用作 API 网关，对应用之间的请求进行路由和授权，并为在复杂系统中跟踪请求提供全面信息。

NGINX 可使用自身可用的任何变量标记请求，以帮助跟踪系统用户充分了解请求的行为。本示例提供了使用 OpenTracing 跟踪代理请求的有限示例。由于 opentracing_tag 指令在 http、server 和 location 上下文中均有效，因此可利用 NGINX 获取大量数据。

其他参考资料

“OpenTracing NGINX 模块 GitHub”
《NGINX Plus OpenTracing 动态模块管理员指南》
Datadog 指南中的“代理跟踪”插件指南
“面向 NGINX 和 NGINX Plus 的 OpenTracing”
“采用 NGINX Ingress Controller for Kubernetes 支持 OpenTracing”

第 15 章
性能调优

15.0 简介

NGINX 调优是一种艺术。无论何种类型的服务器或应用，其性能调优都取决于许多可变项，包括但不限于环境、用例、需求和相关物理组件。瓶颈驱动型性能调优十分常见，意指遇到瓶颈后才进行测试，确定瓶颈，改进限制，并不断重复，直至满足性能需求。在本章中，我们建议在进行系统调优时，使用自动化工具进行测试，并收集和衡量测试结果。本章还介绍了如何通过连接调优保持客户端和上游服务器的长连接，以及如何通过优化操作系统调优满足更多并发连接需求。

15.1 使用压测工具实现测试自动化

问题

需要使用压测工具实现测试自动化，以确保测试一致性和可重复性。

解决方案

使用 HTTP 压测工具，例如 Apache Jmeter、Locust、Gatling 或团队标准化的任何测试工具。为压测工具创建配置，对您的 Web 应用做全面测试，包括对服务进行测试。查看从测试中收集的指标，以建立基线。缓慢增加模拟的并发用户数，以模拟典型的生产使用情况并确定改进点。对 NGINX 进行调优并不断重复此流程，直至实现预期性能。

详解

通过使用自动化测试工具来定义测试，可通过一致的测试，构建 NGINX 调优指标。必须能够重复测试并衡量性能优劣以进行科学分析。在对 NGINX 配置进行任何调整之前，先进行测试以建立基线，这样才能衡量配置更改是否实现了性能优化。对每次更改进行衡量，有助于确定性能得以提升的根源。

15.2 保持客户端长连接

问题

需要增加可在单个客户端连接上发起的请求数以及空闲连接可保持的时长。

解决方案

使用 keepalive_requests 和 keepalive_timeout 指令，更改可在单个连接上发起的请求数以及空闲连接可保持打开状态的时长。

```
http {
    keepalive_requests 320;
    keepalive_timeout 300s;
    # ...
}
```

keepalive_requests 指令默认值为 100，keepalive_timeout 指令默认值为 75 秒。

详解

一般情况下，单个连接上的默认请求数能够满足客户端需求，因为现代浏览器能够为每台服务器（根据 FQDN）打开多个连接。与同一域名的并发开放连接数通常仍限制为小于 10，因此就这一点而言，仍会发生在单个连接上发送多个请求的情况。内容交付网络常用的 HTTP/1.1 创建多个域名指向内容服务器，并以编码的方式轮换所用域名，从而使浏览器打开更多连接。如果前端应用不断轮询后端应用以进行更新，那么这些连接优化可能会很有帮助，因为允许更多请求和保持更长时间的连接可减少需要建立的连接数量。

15.3 保持上游长连接

问题

需要保持上游（upstream）服务器长连接，以通过连接复用提高性能。

解决方案

在 upstream 上下文中使用 keepalive 指令，保持上游服务器连接开放以供复用：

```
proxy_http_version 1.1;
proxy_set_header Connection "";

upstream backend {
    server 10.0.0.42;
    server 10.0.2.56;

    keepalive 32;
}
```

upstream 上下文中的 keepalive 指令会为每个 NGINX worker 进程保持打开状态的连接开启高速缓存。该指令表示每个 worker 进程可保持打开状态的最大空闲连接数。在 upstream 代码块上使用的代理模块指令在确保 keepalive 指令正常用于上游服务器连接方面发挥着不可或缺的作用。proxy_http_version 指令指示代理模块使用 HTTP 版本 1.1，以允许在单个保持打开状态的连接上连续发送多个请求。proxy_set_header 指令指示代理模块删除默认请求头 close，以允许连接保持打开状态。

详解

需要保持上游服务器长连接，以缩短建立连接所需的时间，并允许 worker 进程直接将请求分发到空闲连接上进行处理。需要注意的是，长连接的数量可以超过 keepalive 指令中指定的连接数量，因为长连接和空闲连接并不等同。keepalive 连接数应尽量少，以允许上游服务器接受其他接入连接。该 NGINX 调优技巧可通过减少连接建立开销，提升服务器性能。

15.4 响应缓冲

问题

需要在内存中为上游服务器和客户端之间的响应启用缓冲区，以避免将响应写入临时文件。

解决方案

调整代理缓冲区设置，以允许 NGINX 将响应内容写入内存缓冲区：

```
server {
    proxy_buffering on;
    proxy_buffer_size 8k;
    proxy_buffers 8 32k;
    proxy_busy_buffer_size 64k;
    # ...
}
```

proxy_buffering 指令的值可以是 on 或 off，默认是 on。proxy_buffer_size 表示用于读取来自代理服务器的响应首包和响应头的缓冲区大小，默认为 4K 或 8K，具体取决于平台。proxy_buffers 指令包含两个参数：缓冲区数量和缓冲区大小。默认情况下，proxy_buffers 指令被设置为 8 个缓冲区，依据平台不同单个缓冲区容量为 4k 或 8k。proxy_busy_buffer_size 指令限定了繁忙缓冲区的大小，繁忙缓冲区可支持 NGINX 在还未完全读取上游服务响应内容的情况下直接响应客户端。繁忙缓冲区的大小默认为代理缓冲区或正常缓冲区大小的两倍。如果禁用代理缓冲区，那么发生故障时，将无法把请求发送给下一台上游服务器，因为请求内容已由 NGINX 发送。

详解

代理缓冲区能显著提升代理服务性能，具体取决于响应内容的大小。调整缓冲区设置可能会产生不利影响，因此务必要参考响应消息体的平均大小并反复进行全面测试。非必要情况下不要设置过大缓冲区，因为这会占用大量的 NGINX 内存。可以只为已知会返回大型响应消息体的特定位置设置大型缓冲区，从而优化性能。

其他参考资料

“NGINX proxy_request_buffering 模块配置”

15.5 访问日志的缓冲

问题

当系统处于高负载状态时，需要启用日志缓冲，以降低 NGINX worker 进程发生阻塞的可能性。

解决方案

设置访问日志的缓冲区大小和刷新时间：

```
http {
    access_log /var/log/nginx/access.log main buffer=32k
        flush=1m gzip=1;
}
```

access_log 指令的 buffer 参数表示内存缓冲区的大小，在写入磁盘之前日志数据可保存在内存缓冲区中。access_log 指令的 flush 参数设置在写入磁盘之前，日志可保存在缓冲区中的最长时间。当使用 gzip 时，日志数据在写入日志之前会进行压缩，具有 1（最快轻度压缩）到 9（最慢极限压缩）九个压缩级别。

详解

将日志数据缓冲到内存中可能只是一种很小的优化手段。但是，对于存在大量请求的站点和应用来说，这会显著影响磁盘和 CPU 的使用率。当 access_log 指令中包含 buffer 参数时，如果日志条目无法再进入缓冲区，日志会直接写入磁盘。如果同时包含 flush 参数和 buffer 参数，当缓冲区中的数据超过指定期限时，日志也会写入磁盘。如果跟踪日志，您可能会发现启用缓冲时，延迟可达到 flush 参数指定的时长。

15.6 操作系统调优

问题

为应对尖峰负载或高流量站点，需要优化操作系统以接受更多连接。

解决方案

检查 *net.core.somaxconn* 的内核设置，该参数表示内核可排队等待 NGINX 处理的最大连接数。如果将其设置为大于 512 的值，则需要设置 NGINX 配置中 listen 指令的 backlog 参数以进行匹配。当需要检查此内核设置时，内核日志会给予明确提示。NGINX 会非常快速地处理连接，因此对于大多数用例，无需更改此设置。

通常更需要调大可打开文件描述符（open file descriptor）上限。在 Linux 中会为每个连接打开一个文件句柄；因此，如果将 NGINX 用作代理或负载均衡器，由于需要与上游服务建立连接，NGINX 可能会为每个连接打开两个文件句柄。为了满足大量连接需求，可能需要使用内核选项 sys.fs.file_max 调大系统的文件描述符上限；而对于系统用户，NGINX 就按照 */etc/security/limits.conf* 文件中的配置运行。此外，可能还需要更改 worker_connections 和 worker_rlimit_nofile 的值。它们都是 NGINX 配置中的指令。

开启更多临时端口以支持更多连接。当 NGINX 充当反向代理或负载均衡器时，每个上游连接都会为响应流量打开一个临时端口。根据系统配置，服务器可能不会将所有临时端口都打开。如要确认打开端口的数量，请查看内核设置 net.ipv4.ip_local_port_range 的设置。该设置表示所能使用的临时端口范围。通常将此内核设置设为从 1024 到 65535。1024 是 TCP 注册端口结束的位置，65535 是动态或临时端口结束的位置。请记住，下限应高于最高的开放侦听服务端口号。

详解

当为了满足大量连接需求而开始调优时，应首先优化操作系统。可以根据特定用例，对内核进行多方面的优化。但是，内核调优不应只是一时兴起，而是应根据其性能表现来衡量更改，以确保 更改有所帮助。如前所述，可以根据内核日志中记录的消息来确定何时开始优化内核，或者当 NGINX 在错误日志中给予明确提示时，开始优化内核。

第 16 章
NGINX Instance Manager 简介

16.0 简介

NGINX Instance Manager[5] 与 NGINX 实例代理协同运行。该服务可以扫描网络中的 NGINX 实例以及 TLS 证书。提供实例和证书清单后，您可通过单个界面或 API 查看 Web 服务器和代理的状态。提供 API 以支持集成到现有工具和工作流，实现自动化功能，扫描、检查和更新数据平面。

NGINX 实例代理从 NGINX 或 NGINX Plus 收集指标，并允许直接通过集中界面进行配置检查和更新。可以将 NGINX Instance Manager 用作指标收集集成点，这可帮助降低将 NGINX stub 状态或 NGINX Plus 指标 API 端点添加到指标收集整合平台的复杂性。

NGINX Instance Manager 需要商业许可，但提供免费试用版，以便您自行探索其功能和价值。

16.1 设置概述

问题

需要设置 NGINX Instance Manager。

解决方案

NGINX Instance Manager 服务器的安装流程与其他授权 NGINX 产品的安装流程相同。必须先获得许可文件，以及软件包或仓库证书密钥对。获得必要的文件后，使

[5] NGINX Instance Manager 已于 2022 年正式合并入新产品 NGINX Management Suite，点击了解详情。

用系统的软件包管理器安装软件。有关最新、最确切的安装信息，请参阅《NGINX Instance Manager 安装指南》。

详解

安装完 NGINX Instance Manager 后，查看其用户界面。刚开始清单为空（请参见 图 16-1）。

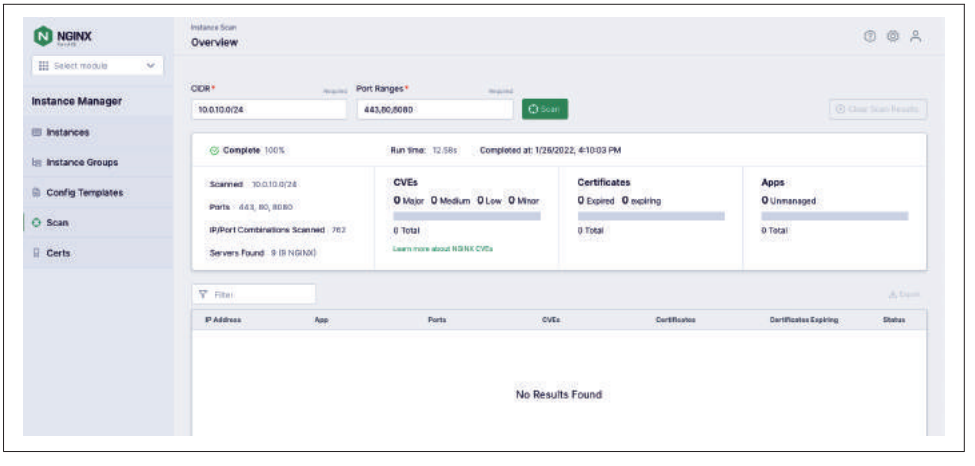


图 16-1. NGINX Instance Manager 用户界面

如要查找 NGINX 实例，则导航到 Scan（扫描）选项卡（请参见图 16-2）。可以使用扫描功能在网络上查找 NGINX 和 NGINX Plus 服务器。扫描工具使用 NMAP 来查找 Web 服务器所在的特定端口。如果有主机正在侦听所扫描的端口，NGINX Instance Manager 会尝试确定它是否为 NGINX 服务器，如果是，则进一步确定其版本。确认方法是向服务器发送请求并检查 Server 响应头。

扫描完成后，会出现一张包含全部所发现实例的表格。该表格会详细列出有关每个实例的信息。如果发现 NGINX 实例，NGINX Instance Manager 就会基于 NGINX 版本查找并显示任何常见漏洞披露（CVE）。然后基于这些结果，通过自动发现必要的补丁来确保 NGINX 服务器安全运行。状态栏会显示哪些 NGINX 实例由 NGINX Instance Manager 管理以及哪些实例需要安装代理。

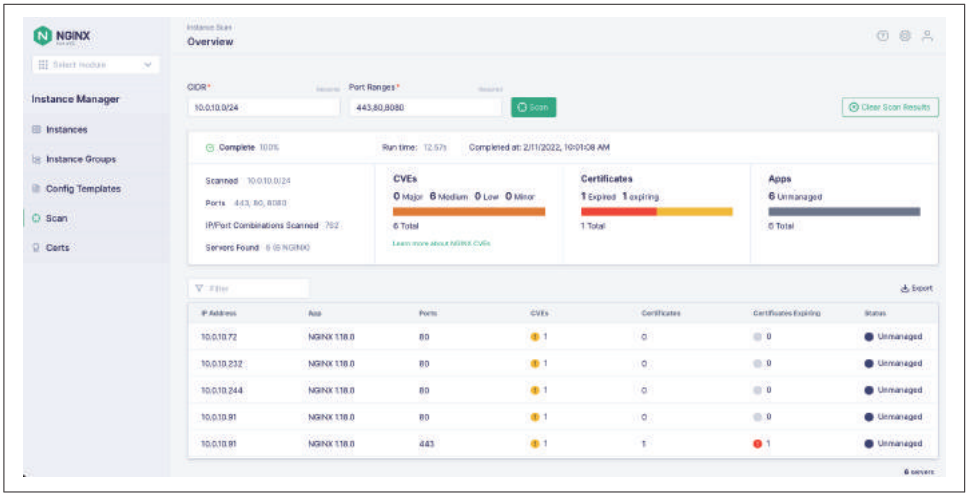


图 16-2. NGINX Instance Manager 扫描

在 Certs 页面上，可以扫描为 TLS 配置的主机。该工具的工作方式与主机扫描相同；但是，它会检查所使用的证书，并提供有关证书的信息，例如通用名和到期日期。该工具可帮助发现证书存放位置以及可能需要更新的地方。

在 Settings（设置）中，可以查看有关 NGINX Instance Manager、用户许可证以及 NGINX Instance Manager 文档、API 和指标遥测配置的信息。也可以在 settings（设置）中配置用户和角色，以控制谁拥有访问权限以及他们可以做什么。

在管理实例之前，必须先在 NGINX 机器上安装 NGINX Instance Manager 代理。如欲了解如何安装代理，请参阅下一节。

其他参考资料

《NGINX Instance Manager 安装指南》
《NGINX Instance Manager 管理员指南》

16.2 代理安装

问题

希望在 NGINX 或 NGINX Plus 机器上安装 NGINX Instance Manager 代理。

解决方案

使用以下命令从 NGINX Instance Manager 服务器下载和执行安装脚本，以安装 NGINX Instance Manager 代理：

```
curl -k https://{NIM hostname or IP}/install/nginx-agent | sudo sh
```

对于基本安装，必须执行下列步骤。如要设置显示的实例名称、位置或实例组成员资格，则可以更新 */etc/nginx-agent/dynamic-agent.conf* 文件并重新启动 nginx-agent 服务：

```
#
# /etc/nginx-agent/dynamic-agent.conf
# NGINX Agent 的动态配置文件。
# 本文件的目的是跟踪
# 可通过 API 和代理安装脚本
# 动态更改的代理配置值。
# 您可以编辑此文件，但是用于修改本系统标记的 API 调用
# 会重写本文件中的标记值。
# API 调用可修改的代理配置值
# 如下所示：
# - tags #
# 代理安装脚本可修改的
# 代理配置值如下所示：
# - instance_name # - instance_group # - location
# 示例值
instance_name: nginx1.example.org
instance_group: internal
location: cloud1
```

详解

在 NGINX 或 NGINX Plus 实例上安装并运行 NGINX Instance Manager 代理后，可以在 NGINX Instance Manager 的清单页面上看到它们。有关主机的信息将一张表中详细列出，包括主机名、NGINX 安装、标签及其连接状态。

选择一个实例后，右侧会出现一个更详细的视图，并且可以选择编辑。选择 Edit（编辑）后，界面上会显示实例的当前 NGINX 配置。然后，可以使用编辑器进行更改或分析配置。编辑器右上角的绿色下拉式按钮支持切换可配置的文件。进行更改后，可以保存配置。如要从保存的配置中重载 NGINX，则必须发布。通过将保存和发布操作分开，可以在 NGINX 尝试重载配置之前对多个文件进行编辑。

16.3 使用 API 实现 NGINX 发现、配置和监控自动化

问题

希望使用 NGINX Instance Manager API 实现发现、配置或监控自动化。

解决方案

NGINX Instance Manager 服务器通过 Swagger UI 托管 OpenAPI Spec，以记录已安装版本 API。可以在 Docs Menu（文档菜单）区域找到此文档。

定期扫描网络中的 NGINX 实例及其证书可能会很有帮助。可借助这些信息发现需要修补的 CVE 或即将到期的证书。

如要开始扫描 NGINX 实例或证书，则向 NGINX Instance Manager 端点 /api/platform/v1/servers/scan 发送 POST 请求，并在请求体中详细说明希望扫描的 CIDR 范围和端口。实例扫描还允许列出端口范围：

```
curl -X 'POST' \
    'https://nginx-manager.example.com/api/platform/v1/servers/scan' \
    -H 'accept: application/json' \
    -H 'Content-Type: application/json' \
    -d '{
    "cidr": "string",
    "hostDiscovery": "icmp",
    "portRanges": [
    "10.0.1.0/24"
    ]
}
```

还可以向同一 API 端点发送 DELETE 请求，以取消扫描。GET 请求将指示返回当前扫描的状态。

完成扫描后，可以向 /api/platform/v1/scan/servers 发送 GET 请求，以返回有关已知 NGINX 实例或证书的信息。

如要列出所有托管的 NGINX 实例，则向 /api/platform/v1/instances 发送 GET 请求。响应内容将提供实例 ID，可在后续请求中使用这些实例 ID 列出配置文件。这些配置文

件可通过 API 进行更新和发布。此功能可用于对 NGINX 实例进行自动化配置更新。可查看 OpenAPI 规范以了解最新用例。

NGINX Instance Manager 还支持访问托管实例的指标信息。外部指标收集工具可通过 API 直接从 NGINX Instance Manager 请求指标信息。可通过访问“实例清单”页面，选择实例并单击“视图指标链接”，来查看有限的指标信息。

详解

当今 Web 技术的运行环境依赖于自动化和监控。NGINX Instance Manager 的 RESTful API 支持将 NGINX 管理集成到现有自动化和 DevOps 工作流中。指标收集和分析为制定有关配置更改的决策以优化服务交付提供了基础。可以将 NGINX Instance Manager 用作在所有 NGINX 实例中收集指标的集成点，从而降低指标收集的复杂性。

其他参考资料

“NGINX Instance Manager API 概览”
“指标 API 使用”教程

第 17 章
NGINX Controller 简介

17.0 简介

NGINX Controller[6] 是一个面向应用环境的以应用为中心的控制平面。NGINX Controller 提供了一个统一的界面，允许查看和配置整个 NGINX Plus 服务器群，无论服务器位于何处。NGINX Controller 支持团队将更多精力放在使用 NGINX Plus 所提供的应用上，而非原始 NGINX Plus 配置上。

本章简单介绍了 NGINX Controller 设置、如何连接 NGINX Plus 服务器实例以及如何使用 NGINX Controller API。NGINX Controller 是一款需要许可的企业产品。可以从 F5 NGINX Controller 产品页面申请免费试用。

17.1 设置概述

问题

希望设置 NGINX Controller 环境。

解决方案

查看官方《NGINX Controller 安装指南》，了解最新安装流程。以下是设置指南中需要注意的一些提示、观察结果和标注。

将 NGINX Controller 3.x 作为 Kubernetes 堆栈安装。务必要在开始安装之前查看所有技术规范。需要使用外部 PostgreSQL 数据库。NGINX Controller 安装程序以压缩包的形式提供。解包后，需要以非 root 用户身份运行 *install.sh* 脚本。

[6] NGINX Controller 已于 2022 年正式停售，点击了解相关新产品 NGINX Management Suite。

由于某些操作系统镜像的分发方式，软件包仓库可能存在差异，这可能会造成一定的安装困难。经测试发现，Ubuntu 20.04 的一致性看似最高，建议基于该操作系统进行试用和研究。请记住，NGINX 支持团队可协助您快速启动并运行 NGINX Controller。

安装程序需要安装许多工具才能正确运行。大多数工具在许多操作系统上都是标配，但 jq 工具除外。在运行任何安装程序脚本之前，需确保系统已安装所有必要工具。

helper.sh 脚本在安装包中提供，可协助安装或更改已安装的基本配置。例如，可使用 supportspkg 参数构建调试和日志信息包，然后将其发送给 NGINX 支持团队，从而让他们快速了解相关情况。可使用 prereqs 参数安装所需软件包并设置 Kubernetes。如要查看 NGINX Controller 的日志，则可使用 ./helper.sh logs。

当运行 installer 命令时，它会检查系统要求，并按需进行额外安装。安装程序会提示输入数据库信息。目前支持 PostgreSQL，但它必须在远程服务器上。提供的用户信息必须支持创建数据库。这些信息可作为命令行参数传递给安装程序。

还需要提供时间序列数据库卷。可以使用本地磁盘、NSF 卷或 AWS EBS 卷。如果选择使用 AWS EBS 卷，系统需要获得适当的 AWS IAM 权限才能将卷附加到实例上。

界面上会显示最终用户许可协议，必须同意协议才能继续。阅读完协议后，按 q 退出协议，然后按 y 同意协议。

需要使用 SMTP 服务器，以通过电子邮件邀请用户或发送电子邮件通知。如果尚不能使用 SMTP 服务器，则可稍后使用 *helper.sh* 脚本来配置这些设置。在提示框中输入一些通用值，将主机设置为 localhost，将端口设置为 25，并拒绝身份验证和 TLS。在 SMTP 配置完成之前，NGINX Controller 将无法发送电子邮件。

使用 FQDN 生成代理配置，并且应将其设置为依赖域。在组织名称提示框中输入一个便于识别的名称，团队或公司名称即可。当在管理员用户提示框中输入值时，要注意使用电子邮件进行系统登录。

SSL/TLS 证书路径可通过安装程序的命令参数或以环境变量的形式提供。如果找不到，安装程序将提示生成自签名的证书。

安装完成后，安装程序会提供指向 NGINX Controller 的链接。最后，按照这个链接，使用管理员凭据登录即可。

详解

NGINX Controller 为管理应用提供了单个控制平面。该界面被划分为多个不同的视图：Platform（平台）、Infrastructure（基础架构）、Services（服务）和 Analytics（分析）。这可为手头的特定任务提供清晰简洁的视图。

“Platform”视图用于管理 NGINX Controller 和用户访问。“Infrastructure”视图提供运行 NGINX Controller 代理的机器的详细信息。下一节将介绍如何通过安装代理将 NGINX Plus 服务器添加到 NGINX Controller 中。

“Services”视图显示了 NGINX Controller 以应用为中心的属性。NGINX Controller 对应用、环境、网关和 API 进行有序组织，支持快速重组和部署。

其他参考资料

《NGINX Controller 安装指南》
《NGINX Controller 技术规范》
“为 NGINX Controller 安装和准备 PostgreSQL 数据库”

17.2 连接 NGINX Plus 与 NGINX Controller

问题

已经安装 NGINX Controller，需要为 NGINX Plus 实例连接代理。

解决方案

如果尚未安装 NGINX Plus，请参考实操指南 1.3 在线获取 NGINX Plus 节点。

查找 NGINX Controller 安装文档的最佳方法是访问 *https://<Controller-FQDN>/docs/infrastructure/agent*。在此文档位置，可以找到有关 NGINX Controller 代理运行所需技术规范的信息，以及有关 NGINX Controller 代理安装和管理的信息。

可轻松将 NGINX Controller 代理安装到现有 NGINX Plus 服务器上。只需从 8443 端口上的 NGINX Controller API 检索安装程序脚本，并使用 API 密钥运行该脚本即可。NGINX Controller 用户界面为安装环境提供了易用的复制粘贴指令。安装完成后，必须使用系统的服务管理器来启动 NGINX Controller 代理。

待 NGINX Controller 代理服务运行后，可以看到一个实例在 NGINX Controller “Infrastructure”视图中运行。

详解

在本节中，我们将 NGINX Plus 服务器作为实例添加到了 NGINX Controller 中。现在“Infrastructure（基础架构）”视图中显示 NGINX Plus 系统清单以及 API 请求列表。当有一个或多个实例在 NGINX Controller 中运行时，可在“Infrastructure”视图中使用“Graphs（图表）”选项卡监控重要服务器和 NGINX Plus 指标。在“Platform（平台）”视图的“Agents（代理）”选项卡下，有一个设置可以启用 NGINX 配置分析器。打开“Infrastructure”视图时，会激活“Analysis（分析）”选项卡。“Analysis”选项卡提供与 NGINX Plus 安装及其当前配置相关的信息。

安装完 NGINX Controller 代理后，会出现一个崭新的 NGINX Plus 节点，这时可制作该机器的可启动镜像，或为这些安装构建配置管理，以便能够复制机器。配置完实例后，即可开始设置服务，包括应用及其环境和提供方式。

其他参考资料

《NGINX Controller 代理安装指南》

17.3 使用 API 驱动 NGINX Controller

问题

已经了解如何配置 NGINX Controller 实体，希望使用 API 实现配置流程的自动化。

解决方案

确保 API 端口（默认为 8443）上的 NGINX Controller 已连接到网络。

NGINX Controller 完全由 API 驱动。该界面仅使用该 API 来提供点击访问和仪表盘。访问 *https://<Controller-FQDN>/docs/api/overview*，查看 NGINX Controller 安装文档中的 API 概述，以了解有关对象、权限和身份验证的基本信息。API 引用位于 *https://<Controller-FQDN>/docs/api/api-reference*。

如要使用 API 快速启动自动化，则查看 NGINX Controller 界面中已配置的实体，编辑实体并查看 API 规范。API 规范将显示创建相关对象的方法、路径和需要的有效载荷。只需替换一些变量，即可启动 NGINX Controller 环境自动化。

详解

对于工程师们，API 可能是他们与 NGINX Controller 进行交互的主要接口，也可能是 Web 接口，无论扮演何种角色，都至关重要。显示 Web 接口 API 调用这一新增功能可帮助您更轻松地查看 API 引用，并加快任务自动化。面向 NGINX Controller 的 Ansible 集合可协助 NGINX Controller 自动化。

其他参考资料

“面向 NGINX Controller 的 Ansible 集合入门”

17.4 通过 NGINX Controller 应用安全防护开启 WAF

问题

正在使用 NGINX Controller ADC，希望为应用开启 Web 应用防火墙（WAF）功能。

解决方案

如果尚未安装，则按照《NGINX App Protect WAF 安装指南》根据所用平台将 App Protect 模块安装到 NGINX Plus 节点上。

导航到 NGINX Controller 中现有应用组件的配置。在“Security”部分，找到 WAF 请求头或设置。开启 WAF 并保存。

WAF 现在正通过默认 WAF 策略处理应用请求。默认策略被设置为就所有签名发出警告，但只拦截高精确度签名。精确度由确定误报率的算法来确定。这意味着可以在收集有关默认策略所报告安全事件的数据时立即开始拦截有害请求。被标记和拦截的请求都将显示在 NGINX Controller 用户界面中，并适当标识。NGINX Controller ADC 将显示触发 WAF 违规的违规事件和 WAF 统计数据。

确保应用正在处理一些流量。测试一个通常会被 WAF 拦截或标记的请求。以下是一个常见的 SQL 注入请求：

```
curl https://<appComponentEndpoint>/?query=9999999%20UNION%20SELECT%201%2C2
```

如果发起的请求被视为安全事件，NGINX Controller ADC 将报告安全分析数据。在此应用和应用组件的“安全分析”部分，找到这些指标。在图 17-1 中，可以看到 NGINX Controller 如何显示被 WAF 标记的请求的指标信息。

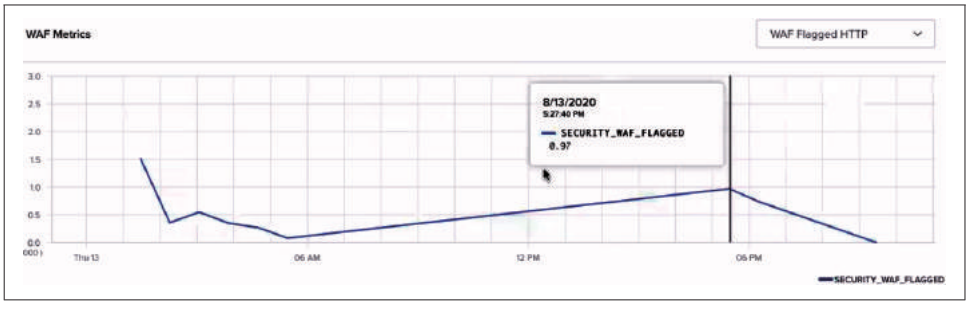


图 17-1. NGINX App Security WAF 标记了一些请求

当显示某些被标记的请求时，也可以在“Security Events”页面上查看事件。在该页面中可以找到 NGINX App Security WAF 所标记或拦截的每个请求的详细信息。

在启用更严格的策略之前，首先应确认正常有效的应用流量不会被标记。如果正常行为被标记，则应检查用于确定个人安全事件的推理。如果正常应用流量被标记，则可能是应用存在需要修补的漏洞。图 17-2 显示了 WAF 检测到的所有流量，表明了常规流量和违反 WAF 规则的请求之间的关系。

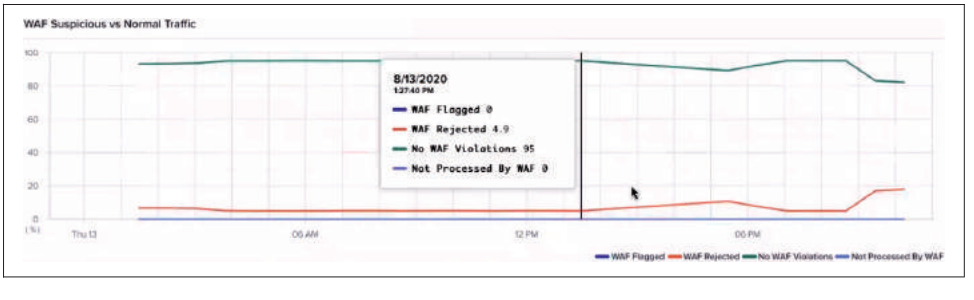


图 17-2. 开启 WAF 的 NGINX Controller ADC 报告的统计信息示例

详解

NGINX Controller App Security 支持轻松使用 WAF 来保护应用。根据所收集的监控信息，您可以随着时间的推移密切关注安全攻击趋势，查看事件细节以开展进一步调查，并确定必须采取何种措施。

Web 应用防火墙在当今 Web 安全架构中至关重要。应用不断受到企图使用常见漏洞入侵服务的尝试攻击的轰炸。通过阻止这些请求到达应用服务，不仅可以保护 Web 应用，还可以为合法的客户端请求保留资源。

WAF 不只适用于外部客户端；应考虑使用 WAF 来保护内部流量，以确保遭到入侵的服务不会影响其他服务。

第 18 章
实用运维提示和结论

18.0 简介

本章将介绍一些实用的运维提示，并对本书进行归纳总结。虽然我们已经在书中讨论了许多与运维工程师相关的概念，但是我还想分享一些有帮助的技巧。在本章中，我将介绍如何确保配置文件清晰简洁，以及如何调试配置文件。

18.1 使用 includes 简化配置

问题

需要清理庞杂的配置文件，将配置按逻辑划分为模块化配置集。

解决方案

使用 include 指令来引用配置文件、目录或掩码：

```
http {
    include config.d/compression.conf;
    include sites-enabled/*.conf
}
```

include 指令中只包含一个参数，可以是文件的路径或是与许多文件匹配的掩码。该指令在任何上下文中都有效。

详解

使用 include 语句，保持 NGINX 配置清晰简洁。可以对配置进行逻辑分组，以避免配置文件达到数百行。可以创建模块化配置文件，然后将这些文件添加到配置中的多个位置，无需复制配置。以 NGINX 大多数软件安装包中都提供的 *fastcgi_param* 配置文件为例。如果在单个 NGINX 上管理多台 FastCGI 虚拟服务器，则可以将此配置文件添加到需要这些 FastCGI 参数的任何位置或上下文中，而无需复制此配置。另一个示例是 SSL 配置。如果正在运行需要类似 SSL 配置的多台服务器，只需编写此配置一次，然后将其添加到所需位置即可。通过对配置进行逻辑分组，可确保配置简洁有序。只需编辑单个文件即可更改一组配置文件，无需在大型配置文件中的多个位置更改多组配置块。对于从业人员来说，最好将配置分组到不同文件中并使用 include 语句。

18.2 调试配置

问题

从 NGINX 服务器收到预料之外的响应。

解决方案

调试配置，并谨记以下提示：

- NGINX 在处理请求时查找特定匹配规则。这虽然增加了手动配置的难度，但却是运行 NGINX 的最有效方法。有关 NGINX 如何处理请求的更多信息，请访问“其他参考资料”部分中的文档链接。
- 可以打开调试日志记录。对于调试日志记录，需确保 NGINX 软件包带有 --with-debug 标记。大多数常用软件包都带有此标记，但如果是自行构建的软件包或最简化软件包，则至少要确认一下。确保开启调试后，可将 error_log 指令的日志级别设置为 debug: error_log /var/log/nginx/error.log debug。
- 可以开启对特定连接的调试。debug_connection 指令在 events 上下文中有效，并将 IP 或 CIDR 范围用作参数。该指令可以被多次声明，以添加多个 IP 地址或 CIDR 范围进行调试。这可帮助调试生产问题，并避免因调试所有连接而降低性能。

- 可以只对特定虚拟服务器进行调试。error_log 指令在主 http、mail、stream、server 和 location 上下文中都有效，可以只在需要的上下文中设置 debug 日志级别。
- 可以启用内核转储并从中获取回溯。可通过操作系统或 NGINX 配置文件启用内核转储。更多相关信息请参阅“其他参考资料”部分中的管理员指南。
- 可以开启 rewrite_log 指令：rewrite_log on 记录重写语句的执行过程。

详解

NGINX 平台规模庞大，配有诸多强大功能。但是，这也有可能带来极大不便。调试时，务必要弄清如何通过配置跟踪请求；如果遇到问题，可添加调试日志级别以寻求帮助。调试日志虽繁杂冗长，但可清晰地显示 NGINX 正如何处理请求，以及配置中存在问题的地方。

其他参考资料

“NGINX 如何处理请求”
《NGINX 调试管理员指南》
NGINX rewrite_log 模块文档

结语

《NGINX 完全指南》聚焦于高性能负载均衡、安全性以及 NGINX 和 NGINX Plus 服务器的部署和维护。本书介绍了 NGINX 应用交付平台的多项强大功能。NGINX Inc. 不断开发新功能，始终保持领先地位。

书中包含许多简短的指南实操指南，可帮助您更好地理解使 NGINX 成为现代 Web 核心的一些指令和模块。NGINX 服务器不仅是 Web 服务器和反向代理，还是一个完整的应用交付平台，具备身份验证完整功能，并在应用环境中安全运行。

索引

第 1 章（1-8）

第 2 章（9-23）

第 3 章（25-34）

第 4 章（35-41）

第 5 章（43-58）

第 6 章（59-67）

第 7 章（69-87）

第 8 章（89-93）

第 9 章（95-98）

第 10 章（99-111）

第 11 章（113-130）

第 12 章（131-137）

第 13 章（139-145）

第 14 章（147-154）

第 15 章（155-160）

第 16 章（161-166）

第 17 章（167-173）

第 18 章（175-177）

关于作者

Derek DeJonghe 对技术充满热情。他在 Web 开发、系统管理和网络方面拥有扎实的专业背景和丰富的行业经验，对现代 Web 架构有着全面的了解。Derek 领导了一支站点可靠性和云解决方案工程师团队，旨在为各种各样的应用提供可自我修复的自动扩展基础架构。在为客户设计、构建和维护高可用性应用的同时，他还为大型组织提供云转型方面的咨询服务。Derek 及其团队始终站在技术浪潮的最前沿，每天都致力于研发云最佳实践。Derek 在弹性云架构方面成绩卓著，革新了云部署安全性和可维护性，可充分满足客户需求。

书末题署

《NGINX 完全指南》封面上的动物是欧亚猞猁（*Lynx lynx*），是最大的猞猁物种，分布范围广泛，从西欧延伸到中亚。

猞猁的耳尖生有黑色耸立簇毛，两颊毛发浓密而粗糙。皮毛颜色从黄灰色到灰褐色，腹部为白色。这只猞猁通身布满黑斑，与南部地区的亚种相比，北部地区的亚种更灰白，斑点更少。

与其他猞猁物种不同，欧亚猞猁捕食较大的有蹄类动物，例如鹿、麋，甚至是驯养的绵羊。成年猞猁每天消耗两到五磅肉，可长达一周食用单一食物。

欧亚猞猁在二十世纪中叶濒临灭绝，后来经过坚持不懈的保护，降级为无危物种。O'Reilly 封面上的许多动物都濒临灭绝，但对世界意义重大。

封面插画由 Karen Montgomery 参考 Shaw 著作 *Zoology*《动物学》中的黑白版画绘制而成。封面字体为 Gilroy Semibold 和 Guardian Sans。正文字体为 Adobe Minion Pro，标题字体为 Adobe Myriad Condensed，代码字体为 Dalton Maag's Ubuntu Mono。