

第 13 章 云时代的 HTTP：Web 的各种强化技术

[日] 涩川喜规 · 详解HTTP：协议基础与Go语言实现

如今，HTTP 已不仅仅是浏览器和服务器之间交互的协议，而成了各种系统之间的通信协议。另外，正如上一章介绍的那样，Web 服务也不再仅使用一台收发 HTTP 的 Web 应用程序服务器来实现所有功能，多个小型服务器协作的微服务等开始受到广泛关注。

当然，大型 Web 服务很早之前就有了。通过在 Nginx 中添加模块，并进行详细设置，基本上就可以实现接下来介绍的大部分功能。通过各种云服务，开发人员开始能够接触这些大型系统所特有的技术。即使企业没有超大规模的数据中心，也可以使用云服务来处理大量请求，轻松地构建耐久性好的系统。

本章将介绍利用云技术实现的大型服务的 HTTP 相关的内容。

另外，2019 年，围绕支持大型 Web 服务的中间件 Kubernetes 的 KubeCon 大会共收到了 1500 项提议，可见这是一个相当火热的领域。几乎每个月会出现新的中间件和工具。因篇幅有限，本书不能详尽地介绍所有最新工具的相关信息，因此，与之前一样，笔者将重点介绍能够帮助大家轻松掌握新技术的基础知识。

13.1 更大规模的 Web 系统的结构

上一章介绍了在一台服务器上运行应用程序的方法。实际上，在提供服务时，随着用户数量的增多，一台服务器的处理能力将无法的需要，一旦服务器发生故障，所有的服务就都停止了。

为了增加可同时处理的请求数，可以在前台设置**负载均衡器**，在负载均衡器下面配置多台前面章节中介绍的服务器。负载均衡器会根据负载和会话数来分散负载。

现在云平台 and 容器得到普及，包含负载均衡器的结构更容易实现了。

在云服务中，通过对软件加以设置，我们可以轻松地修改网络设置、改变处理的服务器数量等，从而实现灵活的结构。云服务提供的负载均衡器具有自动调优功能，可以根据负载大小灵

活增减服务器数量。

除此之外，云服务还具有可轻松提高可用性的优点。我们通常使用的云服务的数据中心位于多个区域，比如日本、新加坡等。另外，如图 13-1 所示，各个区域内部还会分成多个可用区（AZ）。各个可用区的机架、电源和网络等在物理上是独立设置的，即使发生小的故障，也不会同时停止服务。为了确保冗余性，可以在日本和新加坡分别启动 2 个实例，也就是至少启动 4 个服务实例。如果没有云服务，我们就需要在每个地区建立数据中心。这里，各个区域的流量通过负载均衡器和 DNS 等结构进行分配，详细内容将在下一节中介绍。

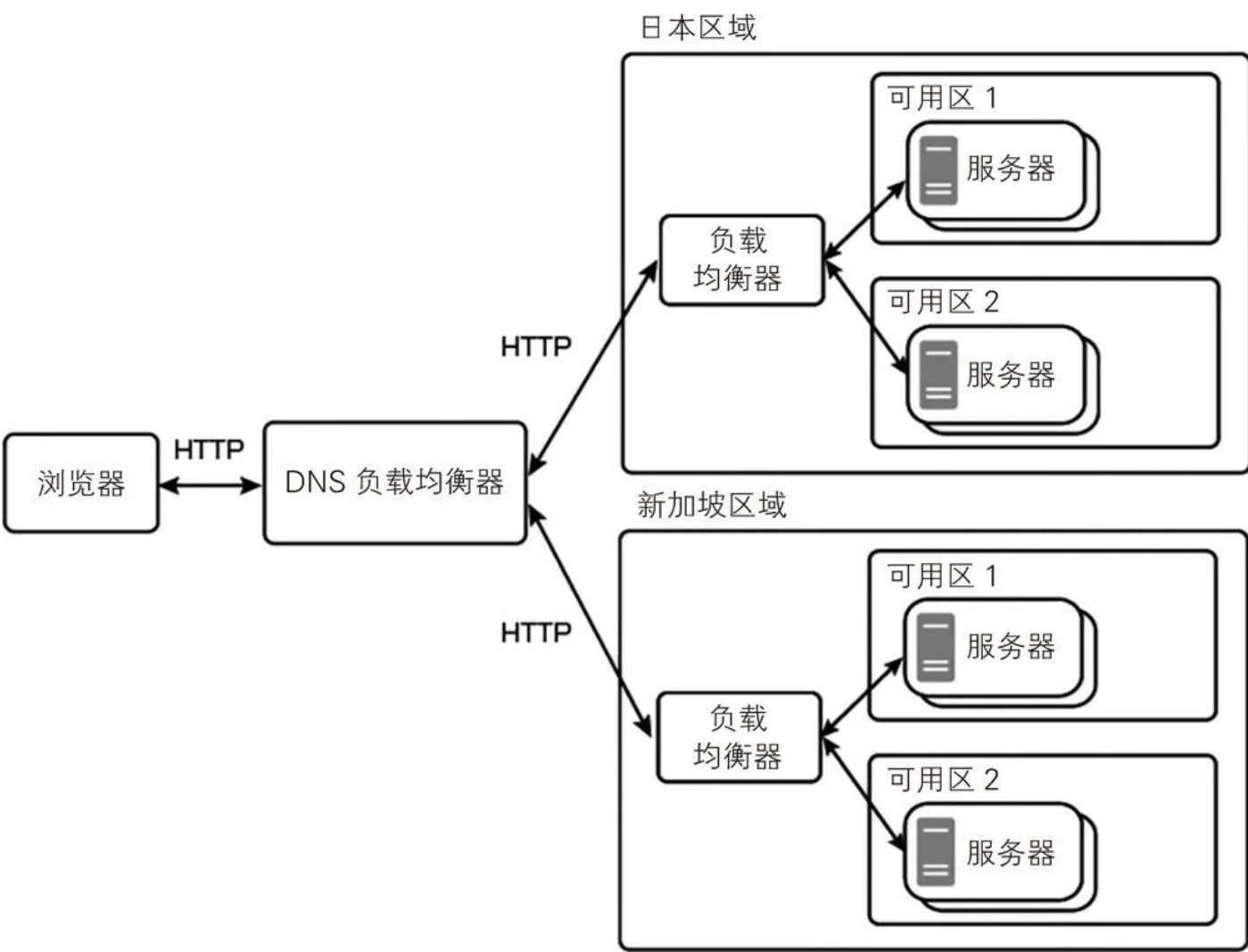


图 13-1 云服务的数据中心

13.2 DNS

本章后面将介绍目前越来越重要的 CDN。为了理解其原理和用法，笔者先来介绍一下 **DNS**（Domain Name Service，域名系统）的相关知识。

在互联网中，域名就是地址，是访问网站的钥匙。不过，域名只是为了方便人们记忆和访问，实际上我们会使用 IP 地址来查找通信数据包到达的服务器，而根据域名找到 IP 地址的操作就是通过 DNS 实现的。

在访问网站时，首先查询 DNS，这样就可以获得要访问的网站的 IP 地址。HTTP/3 基于 UDP 协议，之前版本的 HTTP 基于 TCP 协议，这两个协议指定发送目的地的 IP 地址。在知道 IP 地址后，就可以使用 HTTP 进行通信了。

□ 复制代码

```
1 $ nslookup xxxx.com
2 Server:          100.115.92.193
3 Address:         100.115.92.193#53
4
5 Non-authoritative answer:
6 Name:   xxxx.com
7 Address: 93.184.216.34
```

从浏览器等客户端来看，查询域名并获取 IP 地址的是 DNS。除去采用层次结构的服务器端和缓存等特定位置外，DNS 的结构非常简单。即便如此，近几年也仍出现了一些关于 DNS 的新话题。

13.2.1 DNS 预查询

在 HTTP 的一次通信中，在通信开始之前就应该查询完 DNS，在通信开始之后就不再查询了。不过，在 HTTP 访问返回的 HTML 文件中，经常会引用外部域的资源（比如 Web 字体）。在 Web 服务的运行过程中，一定会耗费成本的处理就是通信传输，比如以下通信。

查询网站的 DNS

请求 HTML 文件

请求 HTML 文件中的 CSS

查询 CSS 中的 Web 字体所在域的 DNS

如果在第一次 HTTP 响应时就返回接下来可能会访问的域名，提前进行 DNS 查询，那么接下来的 DNS 查询就可以与 CSS 查询同时进行，从而省掉第二次 DNS 查询的时间。

虽然也可以使用 `<link>` 标签来指定 DNS 查询，但在 HTTP/2 的相关章节中介绍的 `preconnect` 除了 DNS 查询之外，还会执行 TCP 握手和 TLS 握手，所以现在已经不再使用 `<link>` 标签了¹。

13.2.2 DNS 服务器的缓存

与 DNS 的名称相关联的 IP 地址并不会频繁变化，因此可以使用缓存，使第二次跳转的请求速度变快。

实际持有指定域名与 IP 地址的映射表的 DNS 服务器称为**权威服务器**。如果每次访问网页时都访问权威服务器，那么权威服务器的访问量就会变得非常庞大。通常企业或组织等的网络中会设置 DNS 缓存服务器，缓存服务器会访问上位服务器，最终由该域名的权威服务器返回 IP 地址。因此，访问域的人越多，缓存服务器中的缓存就越多，返回响应所需的时间就越短。对 DNS 来说，缓存是一种不可或缺的机制（图 13-2）。

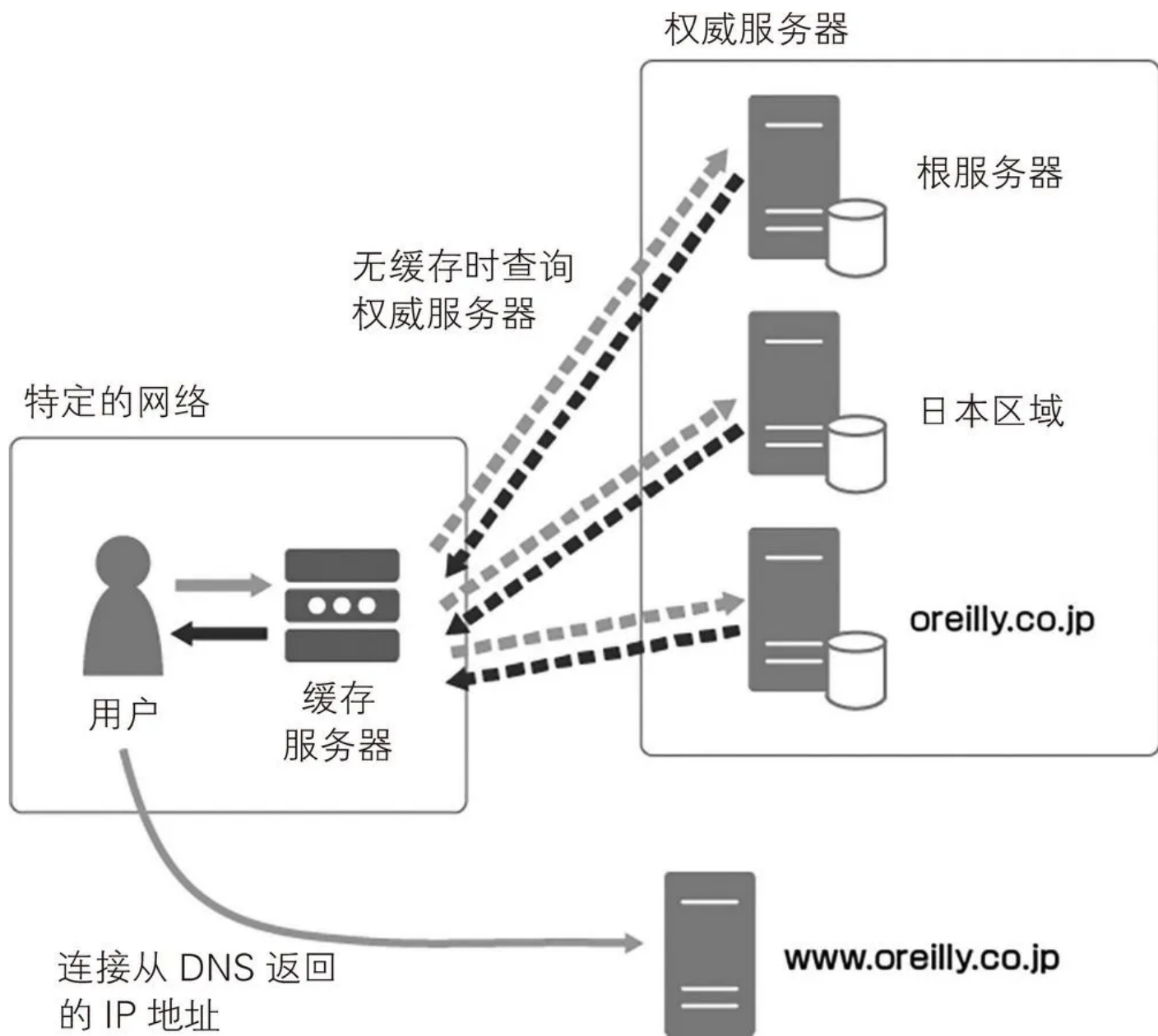


图 13-2 缓存对 DNS 服务器的查询

笔者在介绍 HTTP 的 `Cache-Control` 首部时提到过，缓存是有生命周期的。生命周期通过上位服务器中的 TTL (Time To Live, 存活时间) 值进行设置。在超过 TTL 之后，缓存服务器一旦接收到请求，就会将请求转给上位服务器。在生命周期结束之前，缓存服务器不会访问 DNS，因此，即使修改了 IP 地址，也有可能继续引用旧的 IP 地址。

因此，我们通常会将 TTL 设置为比较长的时间，在修改 IP 地址之前将其改短，然后再恢复为原来设置的长度，这样就能尽快访问新的 IP 地址。根据 Rackspace 网站的介绍，TTL 一般会设置为 1 天 (86 400 秒)，在修改 IP 地址之前改为 5 分钟 (300 秒)。

13.2.3 DNS 客户端的缓存

浏览器通过在短时间内缓存 DNS 的结果来实现高速化。根据之前的信息，Firefox 和 Chrome 的缓存时间都在 1 分钟左右。由于首次下载内容时会使用 Keep-Alive 来保持连接，所以 DNS 不会被重新请求，但在随后对服务器进行 Ajax 通信，以及获取新增内容时，这样做会有一些效果。

在编程语言库中，Java 的 OpenJDK 11 默认将失败结果缓存 10 秒。成功结果的缓存时间在安装了安全管理器的情况下没有限制，在未安装安全管理器的情况下取决于具体实现。Go 语言默认不进行缓存，但 Mercari 公司提供了 OSS 包，在标准的 `net/http` 客户端中添加缓存结构。

除了速度方面的原因之外，由于 Kubernetes DNS 的运行不够稳定，有的浏览器也会引入缓存来提高稳定性。除了 GKE (Google Kubernetes Engine) 之外，在本地运行的 Kubernetes 也不稳定，偶尔有响应延迟或无响应的情况发生。DNS 的缓存正好可以解决这个问题。

13.2.4 DNS 负载均衡

最开始介绍的域名和 IP 地址的映射属于 DNS 的 **A 记录**。根据 A 记录就可以知道 IP 地址。我们也可以设置多个 A 记录，也就是为一个域名分配多个地址。

在这种情况下，响应的 IP 地址在每次调用时都会变化。利用这一点，可以提供多个服务器来分散负载，这称为 **DNS 轮询**。该方法很简单，仅使用标准的 DNS 规范就可以实现。

还有一些 DNS 服务具有更高级的功能。AWS 的 Route 53 还可以仅返回后台的确认过动作的主机名称，或者对多个连接目标设置权重等。

Route 53 的缺点是在进行细致的分配控制和客户端缓存的情况下无法很好地分散负载，连接目标可能会有偏差。AWS 的 Elastic Load Balancer (ELB) 采用两层结构，利用 Route 53 执行跨区域的负载均衡，并在后台部署中间件，进行细致的控制。

13.2.5 DNS 引流

除了指定从域名到 IP 地址的映射的 A 记录之外，DNS 还可以返回从一个域名到另一个域名的映射，那就是 **CNAME**（别名）记录。

使用 CNAME 可以将 DNS 请求移交给其他权威服务器。大型 CDN 服务商 Akamai 就在使用 CDN 的服务器的 CNAME 中指定了其管理的域，这样一来，最终访问的就是 Akamai 的权威 DNS 服务器。我们可以使用 CDN 提供的各种功能，比如根据用户所处的区域返回其他缓存服务器的 IP 地址等。

表 13-1 是 Akamai 的设置示例。

表 13-1 Akamai 的设置示例

名称	TTL	记录类型	目标地址
www	600	CNAME	主机名

13.2.6 使用 SRV 记录的服务发现

拥有多台计算机的人可以在资源管理器或 Finder 中看到同一网络内的计算机。除了计算机之外，同一网络内的 Chromecast 和智能家居设备也会被自动检测到。这种用于查找具有特定功能的服务的机制就叫作**服务发现**。

服务发现的方式有很多种。DNS 中就存在用于服务发现的 **SRV 记录**，该记录定义在 RFC 2782 中。下面是引自 Wikipedia 的 SRV 记录页面的例子（表 13-2）。

1

#

①

②

③

④

⑤

⑥

⑦

⑧

```
_sip._tcp.××××.com. 86400 IN SRV 10 60 5060 主机名
_sip._tcp.××××.com. 86400 IN SRV 10 20 5060 主机名
_sip._tcp.××××.com. 86400 IN SRV 10 20 5060 主机名
_sip._tcp.××××.com. 86400 IN SRV 20 0 5060 主机名
```

复制代码

表 13-2 SRV 记录的内容

序号	记录	说明
①	服务	服务类型
②	协议	TCP 或 UDP
③	名称	运行服务的域
④	TTL	记录的生命周期
⑤	优先级	值越小越优先访问
⑥	权重	在优先级相同的服务中访问的比例
⑦	端口	运行服务的端口号
⑧	主机	可准确指定的主机名

DNS 用于根据主机名查找 IP 地址，而 SRV 记录则用于根据服务类型查找主机名。在上面的记录中，①②③是查询。DNS 可以获得这 4 条记录的信息。收到信息的客户端会访问优先级数值较小的服务。在存在多个服务的情况下，必须根据权重来选择服务。在优先级高的所有服务都无法使用的情况下，再选择优先级次之的服务。

AWS 的 Amazon Elastic Container Service（ECS）就提供了使用 SRV 记录的服务发现功能。

大家可能不太熟悉 SRV 记录，SRV 记录常用于客户端操作系统，而非云。Microsoft 和 Apple 公司以前都拥有查找网络内的计算机的结构，后来 Apple 公司开发的 Bonjour 软件的实现方式成了事实标准。Linux 和 BSD 的操作系统 Avahi、Windows 10 2018 April Update 中也引入了这一方式。

实现 Bonjour 的 DNS 技术包含 RFC 6762 中定义的多播 DNS（mDNS）和 RFC 6763 中定义的基于 DNS 的服务发现（DNS-SD）。

其中，mDNS 使用 UDP 的多播向 DNS 发送请求，通过这种方法，即使不使用本地的 DNS 服务器，也可以解析主机名。这对于查找家庭网络内的服务来说非常方便。不过，云服务不支持多播，所以无法使用。

DNS-SD 通过组合 SRV 记录和 TXT 记录（用于说明的记录），可以实现更高级的服务发现。

13.3 反向代理

本地网络通过**代理**访问外部网络，而当从外部传来请求时，最先接收到该请求的就是**反向代理**。通常，发送请求的操作系统或浏览器会显式地设置通过代理来访问外部网络，而对于反向代理，在浏览器看来，它与运行服务的 Web 服务器是一样的，一般意识不到它的存在。

反向代理这个名称只是表示拓扑位置，实际的操作会根据代理的不同而有很大差异。例如，在边缘侧运行的缓存内容的反向代理称为 CDN，而接收大量请求，并将处理分配给下面的多个服务器以实现负载均衡的反向代理称为负载均衡器。反向代理实现的功能有很多，比如提供认证功能、压缩内容、输出日志、在发送静态内容的同时将动态内容发送给后台服务器等。使用 TLS 接收请求、执行加密等处理也大多是使用前端的反向代理进行的。上一章介绍的在构建前端的 JavaScript 和 CSS 的同时将服务器 API 的请求发送给后台服务器的 Web 服务器也使用了反向代理。

从技术方面来看，反向代理是 HTTP 的 Web 应用程序。不过，它并不是独立运作的，而是会向其他服务中继请求、改写请求或响应、根据情况进行缓存等。

许多支持云服务的反向代理等以高速 Web 服务器 Nginx 为基础。Nginx 除了具有 Web 服务器的功能，还具有反向代理的功能。API 网关 Kong、Kubernetes 上使用的负载均衡器 Ingress 等也以 Nginx 为基础。

用 Go 语言实现反向代理

反向代理可以使用 Go 语言的标准库来实现。下面，笔者来介绍一下用于实现以下处理的钩子函数的用法。

我们先来实现并运行位于反向代理的背后的服务器，接下来创建的反向代理会将请求重定向到该服务器（代码清单 13-1）。

代码清单 13-1 原始服务器

□ 复制代码

```
1 package main
2
3 import (
4     "io"
5     "log"
6     "net/http"
7 )
8
9 func main() {
10     http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
11         log.Println("receive request")
12         io.WriteString(w, "Hello from Origin Server")
13     })
14     log.Println("Origin Server start at :9001")
15     log.Fatalln(http.ListenAndServe(":9001", nil))
16 }
```

先运行一下该服务器，在浏览器中打开 `http://localhost:9001`，就可以看到上面代码中的消息。

改写请求

我们先来看一下最简单的改写请求的相关内容。反向代理的实现中会用到 `httputil` 包的 `ReverseProxy` 结构体，它是作为 HTTP 的处理器运行的。如果使用该代理，我们就不需要自己添加 `X-Forwarded-For` 首部并在首部中记述中继的代理信息了。

在改写请求时，在该结构体的 `Director` 字段中定义改写请求的函数。通过在该函数中设置请求目的地的 URL，就可以将请求转发给所设置的服务器。

□ 复制代码

```
1 package main
2
3 import (
4     "log"
5     "net/http"
6     "net/http/httputil"
7 )
8
9 func main() {
10     director := func(request *http.Request) {
11         request.URL.Scheme = "http"
12         request.URL.Host = ":9001"
13     }
14     rp := &httputil.ReverseProxy{
15         Director: director
16     }
17     server := http.Server{
18         Addr:      "127.0.0.1:9000",
19         Handler: rp,
20     }
21     log.Println("Start Listening at :9000")
22     log.Fatalln(server.ListenAndServe())
23 }
```

除了修改请求目的地之外，还可以改写路径、首部和主体。

□ 复制代码

```
1 // 改写路径
2 request.URL.Path = "/new-path"
3 // 改写首部
4 request.Header.Add("New-Header", "From Proxy")
5 // 改写主体
6 request.Body = ioutil.NopCloser(strings.NewReader("body from proxy"))
```

如果仅修改请求目的地，则还可以使用简单的构造函数。

[复制代码](#)

```
1 target, _ := url.Parse("http://127.0.0.1:9000")
2 rp := httputil.NewSingleHostReverseProxy(target)
```

改写响应

`ModifyResponse` 用于改写响应。该函数还可以改写主体、首部和状态码。

[复制代码](#)

```
1 package main
2
3 import (
4     "log"
5     "net/http"
6     "net/http/httputil"
7 )
8
9 func main() {
10     director := func(request *http.Request) {
11         request.URL.Scheme = "http"
12         request.URL.Host = ":9001"
13     }
14     modifier := func(res *http.Response) error {
15         body, err := ioutil.ReadAll(res.Body)
16         if err != nil {
17             return fmt.Errorf("Reading body error: %w", err)
18         }
19         newBody := bytes.NewBuffer(body)
20         newBody.WriteString("via Proxy")
21         res.Body = ioutil.NopCloser(newBody)
22         res.Header.Set("Content-Length", strconv.Itoa(newBody.Len()))
23         return nil
24     }
25
26     rp := &httputil.ReverseProxy{
27         Director:    director,
28         ModifyResponse: modifier,
29     }
30     server := http.Server{
31         Addr:    "127.0.0.1:9000",
32         Handler: rp,
33     }
34     log.Println("Start Listening at :9000")
}
```

```
35     log.Fatalln(server.ListenAndServe())
36 }
```

改写通信部分

前面介绍的钩子函数用于修改通信过程中的请求或响应，而通信本身正常进行。在 Go 语言中，HTTP 客户端的 `http.Client` 和 `httputil.ReverseProxy` 都持有 `Transport` 字段。该字段是持有 `RoundTrip()` 方法的 `http.RoundTripper` 接口。该接口负责将请求转换为响应，也就是实际进行通信。

与前面介绍的钩子函数不同，该钩子函数可以通过一次响应将处理发往多个后端，或者从缓存返回响应。

下面是通信失败时重试 3 次的示例代码。

[复制代码](#)

```
1 package main
2
3 import (
4     "fmt"
5     "log"
6     "net/http"
7     "net/http/httputil"
8     "net/url"
9     "time"
10 )
11
12 type RetryTransport struct {
13 }
14
15 func (RetryTransport) RoundTrip(req *http.Request) (resp *http.Response, err error) {
16     for i := 0; i < 3; i++ {
17         resp, err := http.DefaultTransport.RoundTrip(req)
18         if err != nil {
19             log.Println("fail")
20             time.Sleep(time.Second)
21             continue
22         }
23         return resp, nil
24     }
25 }
```

```
25     return nil, fmt.Errorf("failed to request to %s", req.URL.String())
26 }
27
28 func main() {
29     target, _ := url.Parse("http://127.0.0.1:9001")
30     rp := httpputil.NewSingleHostReverseProxy(target)
31     rp.Transport = &RetryTransport{}
32     server := http.Server{
33         Addr:      "127.0.0.1:9000",
34         Handler: rp,
35     }
36     log.Println("Start Listening at :9000")
37     log.Fatalln(server.ListenAndServe())
38 }
```

相信通过这些示例代码，大家已经了解了反向代理的结构及功能。本章后半部分会介绍一些云管理服务。在云时代，我们已经基本上不再需要自己创建中间件，但可以使用中间件来创建对这些服务进行本地测试的服务器。

13.4 CDN

近年来，**CDN**（Content Delivery Network，内容分发网络）一词被应用程序开发人员广泛使用。在这之前，CDN 虽然属于基础设施层，但在不了解该层的人看来，CDN 是“快速缓存”“减轻 Web 服务器负载”的处理。现在，CDN 具备用于创建高速的 Web 服务的各种功能，作为 Web 服务组成元素的一部分，其存在感越来越强。

CDN 技术十分复杂，除了 HTTP 之外，它还涉及下层的 TCP/IP。大家在访问互联网时，一般需要与互联网供应商签订合约，加入其网络。而 CDN 就是辅助网络连接，提高网络运行速度和稳定性的技术。

从互联网服务供应商的角度来看，CDN 可以削减付给上游网络的成本，而从实际进行 HTTP 通信的客户端和服务器的角度来看，CDN 主要包含下面两个功能。

提高通信本身的速度和稳定性

作为靠近用户的高级代理服务器

关于 CDN 的详细结构和发展历史等内容，本书并未涉及，大家可以参考《Akamai：不为人知的互联网巨头》² 等图书。

13.4.1 提高通信本身的速度和稳定性

CDN 服务器遍布用户使用的服务供应商的网络。用户在访问配置了 CDN 的网站时，会通过前面介绍的 DNS 结构访问 CDN 提供的域。CDN 提供的域会返回距离用户较近（根据 IP 地址来推断）的主机的 IP 地址，从而将用户引导到自己公司的代理服务器上。

前面提到，计算机（也包含智能手机等）通过加入互联网供应商的网络，可以访问互联网，这里的网络称为 AS（Autonomous System，自治系统）。连接这些 AS 的是互联网，所有的 AS 组成 Tier1、Tier2、Tier3 这样的分层网络。

最上层的 Tier1 的数量非常少，日本 NTT 通信公司是其中之一。当连接服务供应商的用户访问网络时，为了寻找与目标 Web 服务连接的 AS，会根据需要按 Tier3、Tier2、Tier1 的顺序向上查找网络，然后逆向找到 Web 服务（图 13-3）。

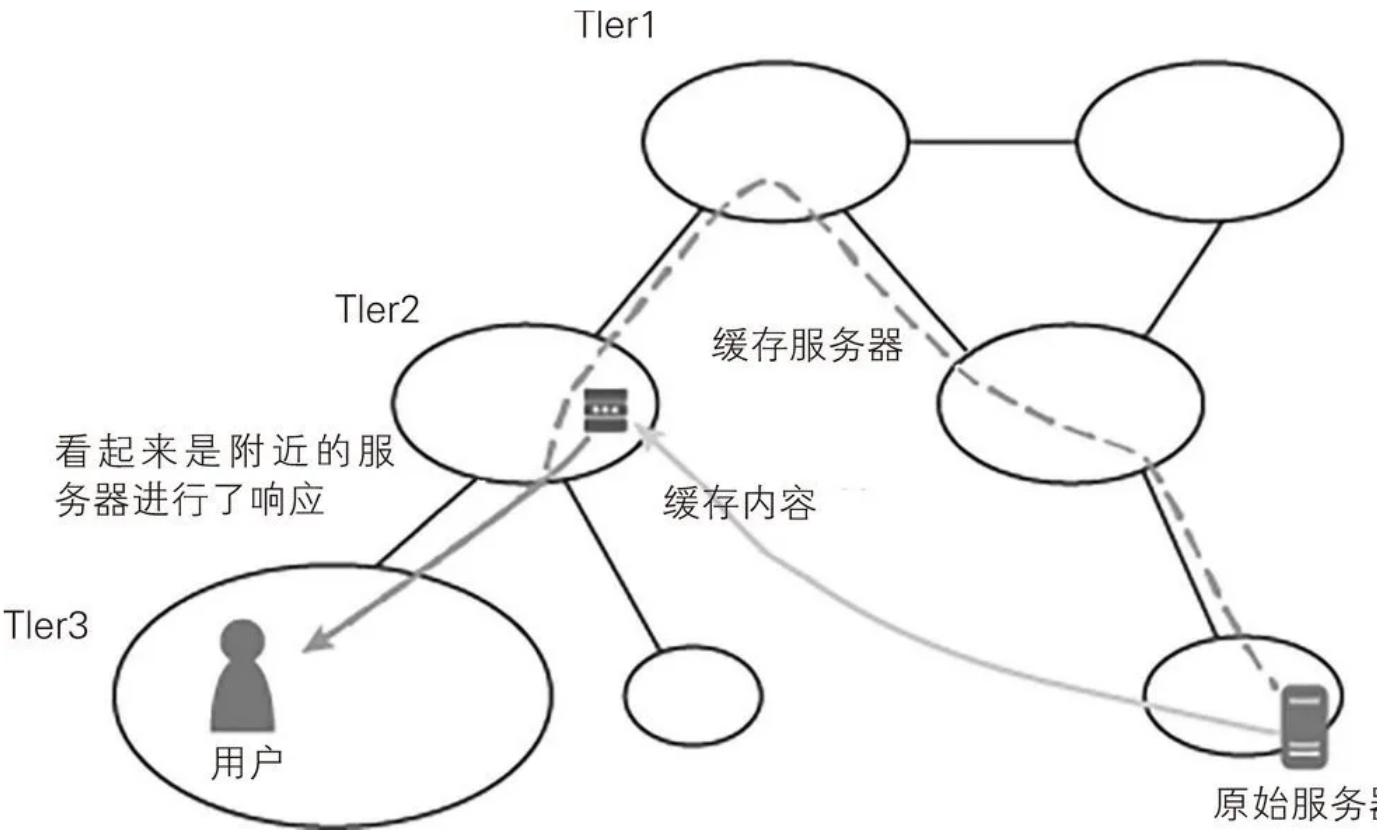


图 13-3 沿着 AS 的层级查找目标 Web 服务

在这样的树结构中，CDN 就像提供了一条捷径，因此，使用 CDN 可以实现高速化。当发生海底电缆切断等故障时，CDN 可以作为备用线路运行，从而提供稳定的通信。

有报告指出，在导入 CDN 后，从日本访问位于美国的服务器的响应时间可以缩短一半。

13.4.2 作为靠近用户的高级代理服务器

如前所述，CDN 将服务器放在用户附近。如果网站的内容可以从这里发送，通信的延迟时间就会变得非常短。另外，CDN 的各个企业也在争相为代理服务器追加各种功能。

在靠近用户的位置发送内容

如果从原始服务器发送大量数据，就需要准备带宽非常大的网络来应对来自全世界的请求，服务器也需要变得非常强大。如果从距离用户较近的位置发送数据，原始服务器的网络带宽就可以不用那么大，负载也会降低。在下载 Windows Update 等大文件，或者发送视频等内容的环境下，这样做的效果十分显著。实际上，提供视频服务的 Netflix 和 Google 既是内容分发商，又是 CDN 运营商³。

CDN 的效果显著，但在某些情况下也会耗费很大的成本。比如，在一些智能手机游戏中，在游戏说明刚结束时随机分发一个道具，为了得到想要的道具，着迷的玩家可能会反复进行游戏的卸载和安装（Reset Marathon）。App Store 上能够发布的应用程序都很小，随着游戏变得丰富，内容不断增大，在这种情况下，服务器通常会使用 CDN 来发布追加的内容，其大小有时甚至能达到 GB 级别。由于用户重复下载大量内容，所以支付给 CDN 的费用也非常高。特别是人气 IP 的游戏，玩家的热情很高，下载量能达到上千万次。10 年前 AWS CloudFront 的价格约为 20 日元 /GB，现在变成了约 10 日元 /GB⁴。

具有高级功能的代理服务器

前面介绍过，CDN 服务器分散在用户附近，并且不会给原始服务器带来负担。利用这个优点，CDN 可以添加各种功能，比如在用户附近执行各种处理等。

首先，通过分散大量的访问，可以应对近年来规模越来越大的 DDoS 攻击。如果向原始服务器集中发起访问，那么将很难修改服务器设置，而如果使用 CDN，我们就可以减少对原始服

务器的请求数量。CDN 还提供了中间件 WAF 来提高安全性，检测非法访问并阻止。

另外，CDN 还添加了许多功能，比如使用 URL 模式修改后面的调用目标、限制能够访问的 IP 地址、对缓存进行详细的设置、在终端使用 JavaScript 等编程语言来自定义响应等。

CDN 还拥有 ESI (Edge Side Includes) 功能，以新闻网站为例，其大部分内容是面向所有用户的，只有一小部分内容是用户特有的，在这种情况下，该功能会在 CDN 上创建整合好控件的 HTML，并将其返回给客户端。

像这样，如今 CDN 已经成为构成 Web 应用程序的不可或缺的系统元素。

Cache-Control (2)

CDN 的服务是作为代理运行的，它提供了高级且快速的缓存功能。第 2 章介绍了指示服务器和客户端一对一通信时的缓存动作的指令，而通过指定 `Cache-Control` 首部的指令，也可以对 CDN 进行指示。另外，也可以指定从服务器向代理发送的响应首部的指令。

下面，我们先来看一下在来自客户端的请求首部中可以使用的指令。

```
no-cache
```

即使 CDN 进行了缓存，也会忽略缓存，直接向原始服务器发起请求。

```
no-store
```

与响应的 `no-store` 一样，删除 CDN 中的缓存。

```
max-age
```

代理中保存的缓存在超过指定时间后，就不可以使用了。

```
max-stale
```

即使响应过期，但只要未超过这里指定的时间，客户端就也可以照常接收该响应。如果未指定具体时间，就表示永远有效。

```
min-fresh
```

要求代理返回这里指定的时间内的缓存数据。

```
no-transform
```

要求代理不可以修改内容。

`only-if-cached`

要求代理仅在缓存的情况下返回响应，否则返回 `504 Gateway Timeout`。在设置该指令后，除了第一次通信之外，代理将不再访问原始服务器。

在响应首部中，可以通过设置以下指令来指示服务器向代理发送的缓存控制。当然，前面介绍的服务器向客户端发送的指令同样适用于代理。

`s-maxage=n`

与 `max-age` 一样，表示可以直接使用缓存的期间。这是针对共享缓存的指令，优先级高于 `max-age`。

`no-transform`

要求代理不可以修改内容。

`must-revalidate`

与 `no-cache` 基本相同，是中间代理服务器的指挥官。在代理服务器询问原始服务器时，如果原始服务器没有响应，则期待代理服务器向客户端返回 `504 Gateway Timeout`。

`proxy-revalidate`

与 `must-revalidate` 一样，但只针对共享缓存。

13.4.3 CDN 的注意事项

虽然 CDN 功能强大，但在使用时也有一些地方需要注意。

首先，虽然 CDN 的目的是实现无缓存的高速通信，但到原始服务器的跃点数会发生改变，这会导致从应用程序服务器获取实际用户的 IP 地址的方法变复杂。

在缓存的情况下，可以类比服务器和浏览器之间的缓存情况来理解 `Cache-Control`。不过，相比浏览器缓存仅处理用户信息，在 CDN 中，如果 `Cache-Control` 使用不当，影响会更大。

在浏览器与原始服务器一对一的通信中，`private` 指令并不是很重要。由于带有该指令的内容针对各个用户是不一样的，所以 CDN 不会缓存内容。相反，如果不设置该指令和 `no-store`，CDN 就会缓存内容。即使针对各个用户的内容不同，如果 URL 相同，最开始的内容也会被缓存，后面用户看到的都是第一个用户的内容⁵。

13.5 负载均衡器

负载均衡器在前台接收请求，根据需要将请求分配给后端的多个实例。人们认为好的负载均衡器可以发挥出后端所有实例的性能，但如果不能准确把握各个实例的瓶颈，就很难实现完全均衡。实际上，我们可以使用简单的算法来进行分配，从而实现某种程度上的均衡。例如，可以使用随机分配的方法、依次给所管理的实例分配任务的轮询方法、按最后连接时间分配的方法，以及使用散列来决定目标服务器的方法等⁶。

要想让负载均衡器正确分配任务，需要做好准备工作。比如对服务器进行健康检查，将不能正常运行的服务器排除在分配对象之外。另外，当负载达到一定程度时，负载均衡器会通过增加实例来降低负载。

高级 DNS 服务 AWS Route 53 也会将无法正常运行的服务器排除在连接目标之外，但负载均衡器拥有更多高级的功能。在 AWS 的 Application Load Balancer (ALB) 中，只需编写简单的规则，就能实现复杂的控制，例如将未认证的用户重定向到登录页面、向指定的客户端（机器人或搜索引擎的爬虫等）返回其他内容，以及让极少数的用户进行 A/B 测试等。

负载均衡器是将任务从一个端点开始分配给下面多个服务的中继服务，而在多个服务相关联的微服务通信的情况下，需要进行高级的路由选择处理，比如为了防止出现单点故障而以多对多的方式连接通信路径、通过断路器等使流量剧增时后续服务不停止。在这种情况下，一般会使用 Istio 等中间件。

连接耗尽

负载均衡器为各种服务提供了中继功能，我们有时需要修改负载均衡器的设置，将应用程序所连接的服务从旧版切换到新版。这时，如果突然切断与旧服务的连接，就会给正在使用服务的用户带来困扰。

在这种情况下，可以通过**连接耗尽**（connection draining）将新的请求发送给新版的服务，并允许对旧版的请求一直执行到结束。

AWS 的负载均衡器和 GCP 的负载均衡器都提供了连接耗尽功能。

13.6 API 网关

API 网关也是后端服务的接口。API 网关由各种云服务提供，总体包含如下功能。这些功能大多也可以通过其他云服务实现。API 网关专门用于在前端和后端的边界提供服务。

API 的设计支持（设计、从 OpenAPI 导入、发布文档、返回 Mock 的响应等）

API 的版本管理

流量控制

API 认证

日志和监视

网关 API 还有其他功能，比如在开发请求 API 使用费的服务时统计各个用户的使用量、以 HTTP 方式访问非 HTTP 后端等。

13.7 健康检查

随着 Web 服务的结构越来越复杂，**健康检查**变得越来越重要。前面介绍的负载均衡器等为了进行健康检查，会向指定的 HTTP 路径发送请求，如果存在问题，就不再向该主机发送请求。AWS 的 ECS 也会检查容器内部的应用程序是否正常运行，如果存在问题，就重新启动各个容器。

健康检查的对象基本上是 Web 应用程序。在这种情况下，通常会根据对 `/health` 和 `/healthcheck` 等路径的请求是否正常执行来进行健康检查。

即使 Web 服务需要认证，一般也不会对健康检查的路径进行认证，或者也可以查看客户端的用户代理名称（AWS 的 Elastic Load Balancer 中为 `ELB-HealthChecker/2.0`

等)，绕过认证。

健康检查可以根据目的分为两类，即 Liveness Prove 和 Readiness Prove。

13.7.1 LivenessProve

Liveness Prove 用于确认服务是否正在运行。通常情况下，当请求到来时，服务器会准备一个仅返回 200 的端点。在服务器进程停止的情况下，由于无法对请求进行响应，所以健康检查的代理能够检测到错误。

□ 复制代码

```
1 http.HandleFunc("/health", func(w http.ResponseWriter, r *http.Request) {  
2     w.Write([]byte("OK"))  
3 })
```

还有一种情况是设置没有问题（比如端口指定没有错误等），但 CPU 或内存已经耗尽，无法再处理请求。Kubernetes 会判断服务器进程需要重启，并重新启动 Pod（由多个容器组成的部署单位）。

13.7.2 ReadinessProve

Readiness Prove 用于确认是否能够提供服务。Liveness Prove 仅用于确认进程是否启动，而 Readiness Prove 也用于确认后端依赖的服务，比如数据库或外部 API 等。如果采用 Go 或 Rust 等编程语言来实现 Web 应用程序，采用使用了字节码的编程语言或脚本语言来编写中间件，则有时启动时间是不一样的，这时就需要进行健康检查。

在这种情况下，有时虽然服务器本身启动了，但启动的准备工作并未完成，或者暂时因资源耗尽而不能返回响应，因此，Kubernetes 会停止对该 Pod 的请求。

13.8 VPC

云服务可以通过软件设置来定义网络结构，大型服务中常用的是 VPC（Virtual Private Cloud，虚拟私有云）。VPC 是通过软件定义来创建网络的。

VPC 可以在云服务中创建私有的网络空间，其最大的优点就是安全。使用 VPC 可以有效地管理连接线路。例如，对外部网络隐藏需要谨慎管理的信息，尤其是包含个人信息的数据库等。另外，在公司内部，在需要准确划分职责等情况下，也可以使用 VPC。VPC 内部会配置服务器实例等。VPC 还可以用于切断子网。

我们可以通过 VPC 使用云服务提供的管理服务。VPC 与外部的连接方式有使用互联网网关的方式、对等连接和直接连接等（图 13-4）。使用互联网网关能够与外部连接。对等连接用于在 VPC 之间建立连接。另外，通过直接连接或互连功能，可以在不经过互联网的情况下使用专线连接内部网络和 VPC。我们也可以使用 VPN 来连接网络。

使用 VPC 可以实现如下服务。

1. 数据库或文件存储不对外部网络公开，仅允许云上的 Web 服务器连接。
2. 用于管理的 Web 服务器不对外部网络公开，仅接受内部网络的访问。
3. 批处理服务器等在外部看不到的地方运行。

另外，即使是同一个 VPC，如果云服务不同，VPC 的功能也会有所不同。在 AWS 中，在为了提高可用性而希望在多个区域使用 VPC 的情况下，需要在各个区域创建 VPC 网络，各区域之间以对等连接的方式连接。GCP 可以创建跨区域的 VPC。

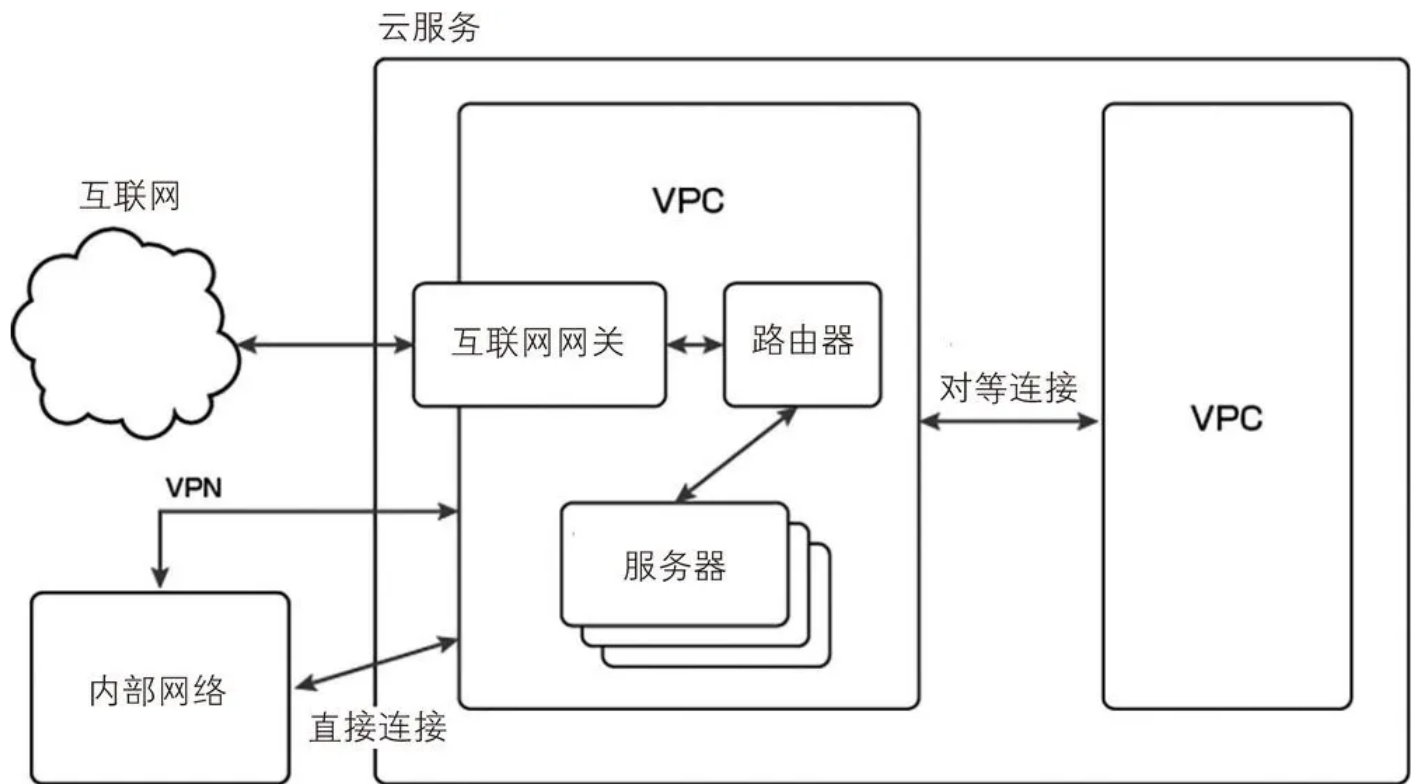


图 13-4 VPC 与外部的连接方式

13.9 微服务与认证

第 5 章介绍了各种认证方式。

在多层微服务架构中，当多个服务向认证服务器查询用户信息时，向认证服务发送的请求就会非常多。因此，微服务中通常使用第 5 章介绍的 JWT (JSON Web Token)。只要有证书，各个服务就可以确认包含签名在内的令牌是否正确，从而分散负载。常见的做法是，仅在持有令牌的情况下，服务才可以通过 API 网关。

由于 JWT 中的信息量非常大，所以如果将这些信息传递给用户可以查看的浏览器，或者保存在能够通过 JavaScript 访问的浏览器的本地存储中，就会存在安全风险。尤其在使用 JavaScript 的情况下，第三方脚本也能够自由地访问信息，这就增加了跨站请求伪造的危险。

因此，我们可以在被称为 BFF 的前端服务器与浏览器之间使用随机认证令牌，由 BFF 发行 JWT 令牌，并将 JWT 发送给后端的微服务。

云服务提供了认证相关的各种功能。大家可以为运行服务器应用程序的服务器实例设置 IAM (Identity & Access Management, 身份识别与访问管理) 角色, 只允许指定的服务器访问数据库。Google Cloud Platform 中通过给实例添加 “标签”, 并设置网络规则, 从而实现了访问控制。

13.10 分布式追踪

追踪是指, 通过微观查看处理的执行顺序和耗时来掌握系统的状况。在微服务中, 处理跨多个进程执行, 在这种情况下, 我们很难了解到底发生了什么, 而通过**分布式追踪**, 即可轻松掌握这种跨系统的处理的状况。近年来, 使用火焰图 (flame graph) 将负载可视化的方法越来越普遍, 分布式追踪就是它的延伸。

分布式追踪系统起源于 Google 为追踪公司内部的服务而开发的 Dapper。关于 Dapper 的论文发表于 2010 年⁷, 后来 Twitter 公司基于这篇论文开发了 Zipkin, Uber 公司开发了 Jaeger。GCP 的 Stackdriver、AWS 的 X-Ray、New Relic APM、Datadog APM 等 SaaS 的记录和监控服务中也提供了该功能, 由此, 分布式追踪普及开来。

随着各种各样的后端服务的出现, 要想将它们嵌入应用程序, 就需要用到各种后端服务专用的 SDK。在这种情况下, 出现了能够处理多个后端服务的 OpenTracing, OpenTracing 后来成为 CNCF (Cloud Native Computing Foundation, 云原生计算基金会) 的一个项目。Google 也开发了内部库 Census 的发布版——OpenCensus。后来, OpenCensus 与 OpenTracing 合并为 OpenTelemetry, 它也成为 CNCF 的一个项目。在笔者编写本书时, 该项目正在开发中。

分布式追踪按一个请求这种独立的单位来获取日志, 这就是一个 Trace, Trace 拥有特定的 Trace ID。在访问数据库或外部 API 时, 使用 Span 来获取耗费的时间。Span 具有父子关系。后端服务会收到 Span 或 Trace ID 的开始时刻和结束时刻, 并整合这些信息, 将一个请求的处理过程中各部分花费的时间可视化。

在介绍这些功能之前, 我们先来看一下单个应用程序中使用 OpenTelemetry 的示例代码。

□ 复制代码

```
1 // ex.com/basic 是所有 Span 的前缀
```

```

2 tracer := global.TraceProvider().Tracer("ex.com/basic")
3
4 tracer.WithSpan(context.Background(), "foo", func(ctx context.Context) error {
5     tracer.WithSpan(ctx, "bar", func(ctx context.Context) error {
6         tracer.WithSpan(ctx, "baz", func(ctx context.Context) error {
7             return nil
8         })
9         return nil
10    })
11    return nil
12 })

```

现在只是创建了 Span，在创建该 Span 前后，以及创建的过程中应该会执行某些繁重的处理。在上面的代码中存在如图 13-5 所示的嵌套的 Span。OpenTelemetry 在 Span 结束时发送信息。该示例代码中存在 3 个 Span，因此会发送 3 次信息。存在父子关系的 Span 持有 128 位的公共的 Trace ID。各个 Span 持有 64 位的 Span ID。如果存在父子关系，父 Span 的 Span ID 也会被发送。Stackdriver 和 Jaeger 等服务将接收这些 Span 信息，根据 Trace ID 或 Span ID 汇总信息，并将 Span 的图像展示给用户。



图 13-5 嵌套的 Span

13.10.1 向子任务传递追踪信息的 HTTP 首部

分布式追踪的关键是可以跨子系统进行追踪。假设我们使用前端的 Web 服务器接收请求，将一部分处理交给后端的 API 服务器，这时基本上会使用本书介绍的 HTTP 或 gRPC 进行通

信。关于 HTTP，W3C 正在制定 `Trace Context` 规范的草案，其中规定了跨系统的追踪中所需信息的传播方法。前面介绍的 `OpenTelemetry` 就是以该规范⁸ 为基础的。

我们试着调用一下外部服务。如果将 `Request` 传递给 `W3C()` 函数，就会返回设置了所需参数的新 `Request`。

□ 复制代码

```
1 // import "go.opentelemetry.io/otel/sdk/plugin/httptrace"
2
3 req, err := http.NewRequest("GET", "http://xxx.com", nil)
4 req := httptrace.W3C(ctx, req)
5 response := client.Do(re)
```

该规范中定义了两个首部。

`traceparent`：包含分布式追踪中必不可少的信息

`tracestate`：包含分布式追踪的库的实现所依赖的值等 Key-Value 值

前者是必不可少的属性，后者是可选项。除了版本号之外，`traceparent` 中还包含前面介绍的 Trace ID、父 Span ID 和采样的选项。在接收了请求的服务器中，如果添加这些信息创建新的 Span，服务器就可以对信息进行整合，最后输出将客户端和服务器的结果合并后的内容。

`traceparent` 如下所示。

□ 复制代码

```
1 traceparent: 00-67123c8eb0007ce146241a4c8b049865-a8b3e8afbe078fdc-01
```

开头的数值是版本号，随后是 Trace ID，第 3 组数值是 Span ID，最后是是否采样的标识。当微服务接收到来自其他服务的请求时，如果请求带有该首部，就会创建新的 Span ID，将使

用该首部接收的 Span ID 作为父 Span ID，与 Trace ID 一起发送。这样一来，即使是跨网络的访问，也可以在一个位置确认以相同的任务为起点的请求的日志。

`tracestate` 首部用于附加依赖于库或实现的元数据。它是一个标签，作用是方便使用 SaaS 服务等来查找日志，其中包含了某些信息。

当然，即使没有跨子系统，或者应用程序是依赖于数据库或外部 SaaS 服务的单独的应用程序，通过在应用程序内部包含分布式追踪的日志，也能够监控在调用外部服务前后在什么地方花费了时间，从而更容易发现可以改进的地方（图 13-6）。

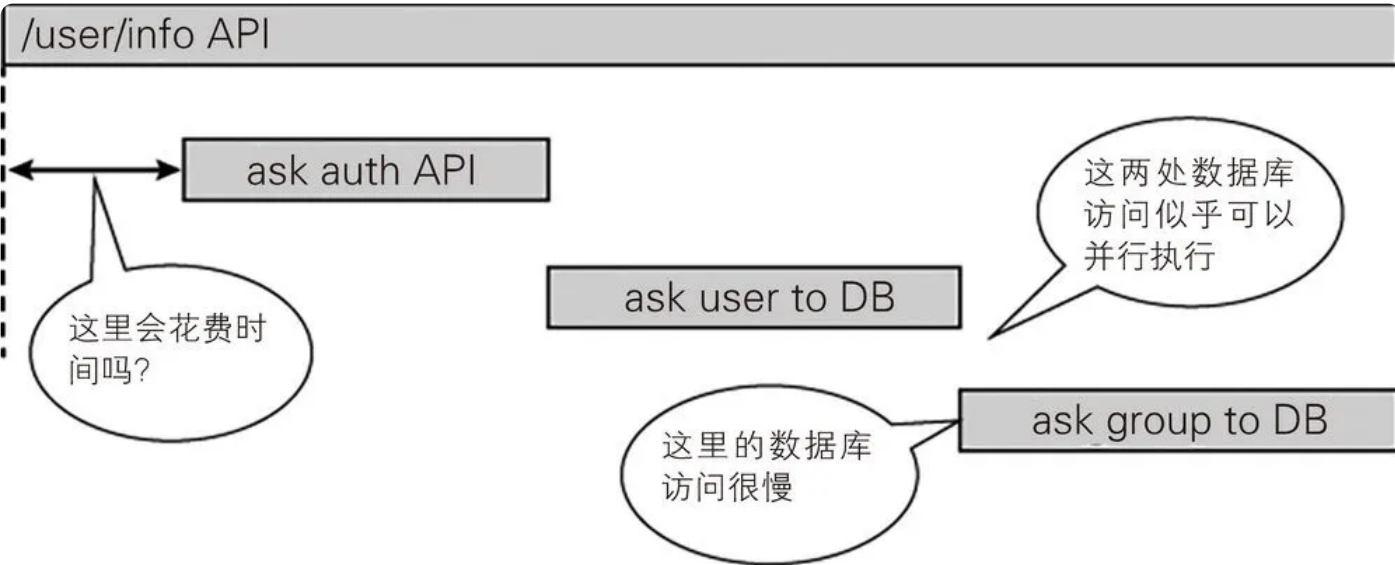


图 13-6 通过追踪发现应用程序的瓶颈

13.10.2 向浏览器传递服务器内部的时间信息的 HTTP 首部

在性能调优时，用于显示 Span 及结果的工具中比较受欢迎的实际上是 Google Chrome 的开发者工具。在使用 Go 标准库进行追踪时，该工具可以生成用于在 Chrome 上显示的文件。

直接将服务器的结果返回给浏览器会怎么样呢？`Server-Timing` 就是用于将服务器的结果返回给浏览器的首部。首部的内容是“`标签 ;desc=` 说明 `”;dur=` 时间（毫秒）”。多个时间信息也可以一起返回，中间用逗号分开即可。时间和说明可以省略。为了在主体发送完成后返回总共花费的时间，这里使用了 HTTP/1.1 中引入的 `Trailer` 首部

⁹（图 13-7）。

□ 复制代码

```
1 Server-Timing: cache;desc="Cache Read";dur=23.2
2 Trailer: Server-Timing
3
4 (主体)
5
6 Server-Timing: total;dur=123.4
```

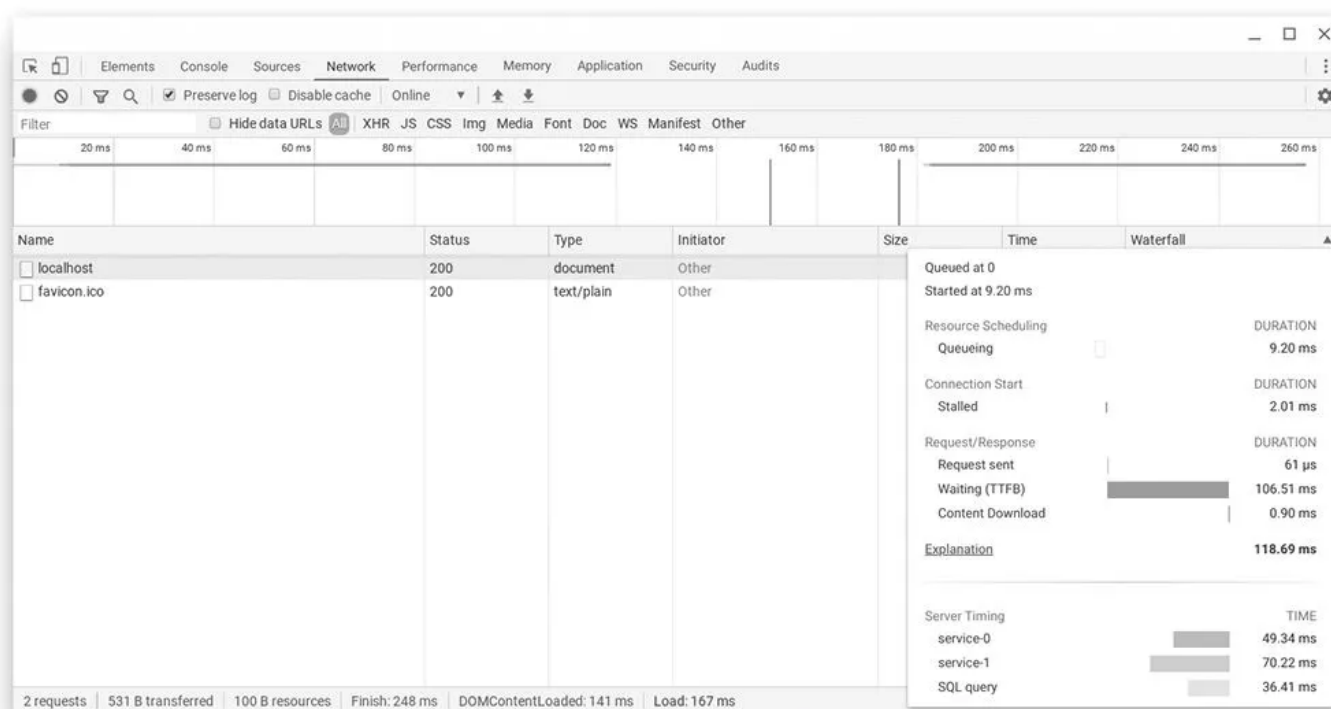


图 13-7 通过开发者工具中的网络标签也可以确认服务器端的处理时间

不过，由于服务器和客户端的时间不是同步的，所以这里并没有开始时间信息，信息量也不像分布式追踪那样大。即便如此，在服务器的结构比较简单、仅实现数据库访问和单个 API 访问的情况下，这样就可以了。

大家也可以自己生成首部，或者使用生成该首部的 Go 语言辅助库。

13.11 其他技术元素

前面我们从 HTTP 的角度对云服务的网络基础设施相关的常见概念进行了介绍。跳出 HTTP 的范畴，云服务的网络基础设施中还有很多其他概念。

例如，Service Mesh 对大型网络进行抽象化，除了分散负载之外，还提供了许多功能。在讲解 DNS 时，笔者简单介绍了服务发现，服务发现也用于减少内部服务之间的紧密连接。另外，本书基本上只介绍使用 HTTP 的同步通信，而在大型服务中，为了提高可用性，使用消息队列将负载分散到异步批处理中的做法也比较常见。

13.12 本章小结

本章介绍了在使用云服务构建系统时所涉及的 HTTP 的相关信息。

得益于云服务，我们可以组合使用能够进行各种 HTTP 通信的中间件来构建系统。本章介绍了 CDN 和负载均衡器等在中继通信的同时提供各种附加价值的服务。上一章介绍的单个 Web 应用程序是系统的构建模块，通过组合这些模块，我们可以构建一个能够处理大量访问的系统。

在笔者刚开始学习编程时，IBM 公司发展势头正猛，发布了许多面向对象设计理论及支撑理论的工具。在 IBM 的技术博客 developerWorks 中，有很多文章是关于业界热点的，理解了这些文章，就能够了解行业的大致情况。现在，业界的 R&D 话题中心变为了 Amazon、Google、Microsoft 等的云服务，对云服务的投资也越来越大。

云服务的变化很大，本章着眼于基础，按照不同的层介绍了各种中间件，例如，CDN 可以在边缘执行代码，还可以实现在其他层进行的 A/B 测试等功能。笔者认为今后各层的重叠范围会越来越大，边界也会越来越模糊。

在这个变化最显著的领域，希望各位读者能够享受这种变化。

AI智能总结

本文深入介绍了云时代的 HTTP 技术，强调了 HTTP 在各种系统间的通信协议中的重要性。随着用户数量增多，单台服务器的处理能力无法满足需求，因此需要负载均衡器和多台服务器协作的微服务。云平台 and 容器的普及使得包含负载均衡器的结构更容易实现。另外，通过云服务可以轻松修改网络设置、改变处理的服务器数量，实现灵活的结构。文章还介绍了 DNS 的相关知识，包括 DNS 预查询、DNS 服务器的缓存、DNS 客户端的缓存和 DNS 负载均衡。此外，还介绍了反向代理的概念和实现方式，以及使用 Go 语言实现反向代理的方法。总的来说，本文强调了云技术对于实现大型服务的重要性，以及 DNS 在互联网通信中的关键作用。同时，还涵

盖了负载均衡、API 网关和健康检查等多个方面，对于了解云时代的 HTTP 技术和相关概念具有重要参考价值。

- [1]: 根据 MDN 中对 `Cache-Control` 的介绍，推荐同时指定 `no-cache` 和 `no-store` 标签，这样一来，仅支持 `Cache-Control` 的浏览器也会有好的效果。不过，现在主要的浏览器都已经支持了。
- [2]: 原书名为『アカマイ知られざるインターネットの巨人』，暂无中文版。——译者注
- [3]: Netflix 的 95% 的数据是通过缓存服务器处理的。
- [4]: 如果是大额合同，应该会有折扣，但当时还流行过“CDN 破产”这种说法。
- [5]: 本书第 1 版出版后不久，Mercari 公司就发生了缓存相关的重大事件。当时的应对措施和问题报告等都值得我们借鉴。
- [6]: 关于各种负载均衡算法，可参考《SRE: Google 运维解密》一书。
- [7]: 论文名为“Dapper, a Large-Scale Distributed Systems Tracing Infrastructure”。
- [8]: OpenTelemetry 合并的 OpenCensus 和 OpenTracing 遵循 Zipkin 规定的 b3-propagation 规则，现在 OpenTelemetry 也遵循该规则。
- [9]: 关于 首部的详细内容，请参考附录。

精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。