

[N3xtChen's Blog \(/n3xtchen\)](#)    [时间线 \(/n3xtchen/archive\)](#)

[文章分类 \(/n3xtchen/categories\)](#)    [标签 \(/n3xtchen/tags\)](#)

# N3xt-Tech 博客

Sharing Funny Tech With You

## Nginx — 理解HTTP 代理，负载均衡，缓冲（Buffering）和缓存（Caching）

---

19 February 2016

### 介绍

这里，我们将介绍 **Nginx** 的 Http 代理功能（请求（**request**）通过 **Nignx** 传递到后端服务器，进行后续处理）。**Nginx** 经常设置为 反向代理（**Reverse Proxy**）帮助 横向扩展（**scale out**：通过增加独立服务器来增加运算能力）基础架构（**infrastructure**）来提升负载能力或者传递请求给下一级代理服务器。

接下来，我们将讨论如何使用 **Nginx** 的 负载均衡（**load balance**）功能来 横向扩展（**scale out**）服务器。我们同时还会探讨使用 缓冲（**buffering**）和 缓存（**caching**）技术来提升代理性能。

## 常规的代理信息

如果你之前只是部署单台 Web 服务器，那你可能会想知道为什么需要代理请求。

横向扩展（**scale out**）提升 基础架构（**infrastructure**）的能力是使用代理的原因之一。**Nginx** 的设计初衷就是被用来处理并发请求，是客户端接触点的理想选择。代理服务器可以传递 **request** 给多个能处理大量任务的后端服务器，达到跨设备分散负载的目的。这样的设计同样也能帮助你更容易得添加服务器或者下线需要维护的服务器。

当应用服务器没有直接处理 **request** 的能力的时候，代理服务器就可以发挥作用了。很多框架（包括 Web 服务器）不如专门设计成高性能服务器（如 **Nginx**）那样健壮。这种场景下，把 **Nginx** 放在这些服务之前，可以提升用户体验和安全性。

**Nginx** 通过接收 **request**，把它转发给其他服务器处理来完成代理过程的。**request** 的处理结果会返回 **Nginx**，然后转发给客户端。实例中的其他服务器可以是远程机器，本地服务，甚至是由 **Nginx** 定义的其他虚拟服务。由 **Nginx** 代理的服务器，我们称之为 **upstream**（上游）服务。

**Nginx** 可以代理使用 **http(s)**, **FastCGI**, **SCGI** 和 **uwsgi** 的请求，或者为每种代理类型指定不同指令的 **memcached** 协议。在这个指南中，我们专注于 **http** 协议。**Nginx** 实例负责传递 **request**，并把各个信息融合成一个 **upstream** 可理解的格式。

## 解构一个简单的 HTTP 代理传递过程

最简单的代理类型莫过于把一个 **request** 导向到单一使用 http 协议通信的服务器了。我们把这类代理统称为 **proxy pass**，由 `proxy_pass` 指令处理。

`proxy_pass` 指令主要在 `location` 的 **context**（中文含义：语境或上下文）中使用。它还可以在 `location` 和 `limit_except` 的 **context** 的 `if` 语法块中使用。当 **request** 匹配到一个包含 `proxy_pass` 的 `location` 中时，该 **request** 将会被指令转发（Forward）到这个链接去。

让我们看一个例子：

```
# server context

location /match/here {
    proxy_pass http://example.com;
}

. . .
```

在上面代码片段中，`proxy_pass` 语句中的服务器地址没有提供 URI。在该模式下，**request** 的 URI 会原封不动地直接传递给 **upstream** 服务器。来看个例子：

- **Nginx** 所接受的 **request** 的原始 URI: `/match/here/please`
- `example.com` 从 **Nginx** 接收到的形式: `http://example.com/match/here/please`

让我们一起看看另外一个场景：

```
# server context

location /match/here {
    proxy_pass http://example.com/new/prefix;
}

. . .
```

上述例子中，代理服务器在尾部定义了 URI。当 URI 放到 `proxy_pass` 定义中时，**request** 中匹配这个 `location` 的部分会在传递的过程中将会被这个 URI 直接覆盖掉，再来看个例子：

- **Nginx** 所接受的 **request** 的原始 URI: /match/here/please
- **upstream** 服务器从 **Nginx** 接收到的形式: http://example.com/new/prefix/please，这里 /match/here 被替换成 /new/prefix

有时，这样的替换是失效。这时，定义在 `proxy_pass` 的尾部的 URI 会被忽略，**Nginx** 直接把来自客户端或被其他 **Nginx** 的指令修改的 URI 传递给 **upstream** 服务器。

例如，使用正则表达式匹配 `location`，URI 的匹配出现争议时，**Nginx** 直接发送客户端 **upstream** 的原始 URI。还有另一个例子，当一个 **rewrite** 指令在同一个地址中使用，会导致客户端的 URI 被重写，但是仍然在同一个 **block** 下处理。这时，传递的 URI 是重写后的。

## 理解 Nginx 处理 Headers 的方式

如果你希望 **upstream** 能合理地处理 **request**，那仅仅传递 URI 是不够的。来自于 **Nginx** 的 **request** 和直接来源于客户端的 **request** 之间还是有区别的。这里最大的差异来自于 **request** 的 **Headers**（头信息）。

当 **Nginx** 代理一个 **request**，它会自动对 **Headers** 做一些调整：

- **Nginx** 会去除任何空的 **Headers**。转发空值是没有意义的；它只会让 **request** 变得臃肿。
- **Nginx** 默认把名称包含下划线的 **Headers** 视为无效，直接移除。如果你希望让这类型的信息生效，那你要把 `underscores_in_headers` 指令设置成 `on`，否则这样的头信息将不会把他发送给后端服务器。
- `Host` 会被重写由 `$proxy_host` 定义的值。它可以是由 `proxy_pass` 指令定义的 **upstream** 的 IP（或者名称）和端口。
- **Headers** 中的 **Connection** 改成 `close`。这个 **Headers** 用在两个服务器创建特定连接的信号信息。在这个实例中，**Nginx** 把它设置成 `close`，一旦原始 **request** 被响应，**upstream** 的这个连接将被关闭。**upstream** 不应该期望这个连接被持久化。

从第一点看来，我们可以确定任何不希望被转发的 **Headers** 都应该被设置成空字符串。带空值的 **Headers** 会被完全删除掉。

接下来一点用来设置如果你的后端应用想要接受非标准的 **Headers**，你应该确保它们不应该带下划线。如果你需要的 **Headers** 使用了下划线，你需要把 `underscores_in_headers` 指令设置成 `on`（在 `http` 的 **context** 或者为这个 IP 和端口组合声明的默认服务器的 **context** 中有效）。如果你不想这么做，**Nginx** 将会把这类 **Headers** 标记为无效，并在传递给 **upstream** 之前把它丢弃。

**Headers** 的 **Host** 在大部分代理场景中都起着重要作用，它默认被设置成 `$proxy_host` 的值，一个由 `proxy_pass` 定义的包含 IP 或名称和端口的值。这样的默认设定是为了让 **Nginx** 确保 **upstream** 可以响应的地址是唯一的（它直接从连接信息取出）。

**Host** 常见的值如下：

- `$proxy_host`：它把 **Host** 设置成从 `proxy_pass` 定义的 IP 或名称加上端口的组合。从 **Nginx** 的角度看，它是默认以及安全的，但是经常不是被代理服务器需要的来正确处理请求的值。
- `$http_host`：它把 **Host** 设置成客户端 **request** 的 **Headers** 中相关信息。这个 **Headers** 由客户端发送，可以被 **Nginx** 使用。这个变量名前缀是 `$http_`，后面紧跟着 **Headers** 的名称，以小写命名，任何斜杠都会被下划线替换。虽然 `$http_host` 在大部分情况可用的，但是当客户端的 **request** 没有有效的 **Host** 信息的时候，会导致传输失败。
- `$host`：这个是偏好设置，它可以是来自 **request** 的主机名，请求中的 **Host** 或者匹配请求的服务器名。

在大部分情况，你将会把 **Host** 设置成 `$host` 变量。它是最灵活的，经常为被代理的服务器提供尽可能精确的 **Host** 信息。

## 配置或者重置 Headers

为了适配代理连接，我们可以使用 `proxy_set_header` 指令。例如，为了改变我们之前讨论的 **Host** 以及其它的 **Headers** 中的配置，我们可以这么做：

```
# server context

location /match/here {
    proxy_set_header HOST $host;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

    proxy_pass http://example.com/new/prefix;
}

. . .
```

上述配置把 **request** 中 **Headers** 的 **Host** 设置成 `$host` 变量，它将包含 **request** 的原始主机名。**Headers** 的 `X-Forwarded-Proto` 提供了关于原始 **request** 的 **Headers** 中被代理服务器协议（http 还是 https）。

`X-Real-IP` 被设置成客户端的 IP 地址，以便代理服务器做判定或者记录基于该信息的日志。`X-Forwarded-For` 是一个包含整个代理过程经过的所有服务器 IP 的地址列表。在上述例子中，我们把它设置成 `$proxy_add_x_forwarded_for` 变量。这个变量包含了从客户端获取的 `X-Forwarded-For` 和 **Nginx** 服务器的 IP（按照 **request** 的顺序）。

当然，我们也可以把 `proxy_set_header` 指令移到 `server` 或者 `http` 的 **context** 中，让它同时在该 **context** 的多个 `location` 中生效：

```
# server context

proxy_set_header HOST $host;
proxy_set_header X-Forwarded-Proto $scheme;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

location /match/here {
    proxy_pass http://example.com/new/prefix;
}

location /different/match {
    proxy_pass http://example.com;
}
```

## 为负载均衡代理服务器定义 Upstream 语境 (Context)

在上一个例子中，我们演示了如何为了一个单台后端服务器实现简单的 http 代理。**Nginx** 让我们很容易通过指定一个后端服务器集群池子来扩展这个配置。

我们使用 **upstream** 指令来定义服务器群的池子（pool）。这个配置假设服务器列表中的每台机器都可以处理来自客户端的 **request**。我们可以通过它轻轻松松地 横向扩展（**scale out**）我们的 基础架构（**infrastructure**）。**upstream** 指令必须定义在 **Nginx** 的 http 的 **context** 中。

让我们一起看个简单的例子：

```
# http context

upstream backend_hosts {
    server host1.example.com;
    server host2.example.com;
    server host3.example.com;
}

server {
    listen 80;
    server_name example.com;

    location /proxy-me {
        proxy_pass http://backend_hosts;
    }
}
```

上述例子，我们设置一个叫做 `backend_hosts` 的 **upstream context**。一旦定义了，这个名称可以直接在 `proxy_pass` 中使用，就和常规的域名一样。如你所见，在我们的服务器块内，所有指向 `example.com/proxy-me/...` 的 **request** 都会被传递到我们定义的池子中。在池子里，会根据配置的算法选取一台服务器。默认，它只是一个简单的**round-robin**（循环选择）处理（即每一个请求都会按顺序传递给不同的服务器）。

## 改变 Upstream 均衡算法

你可以通过指令修改 **upstream** 池子的 均衡算法（**balancing algorithm**）：

- **round-robin**（循环选择）：默认的算法。在其它算法没被指定的情况下，它会被使用。upstream **context** 定义的每一个服务器都会按顺序接受 **request**。
- **least\_conn**（最少连接）：指定新的 **request** 永远只会传递给拥有最少连接的后端服务器。在后端连接需要被持久化的情况下，这个算法将很有效。



- **ip\_hash**: 这种 均衡算法 (**balancing algorithm**) 是基于客户端的 IP 来分发 **request** 到不同的服务器。把客户端 IP 的前三位八进制数作为键值来决定由哪台服务器处理。这样，同一 IP 的客户端每次只会由同一个台服务器处理；它能保证 **session**（会话）的一致性。
- **hash**: 这种 均衡算法 (**balancing algorithm**) 主要运用于缓存代理。这是唯一一种需要用户提供数据的算法；算法根据用户所提供数据的哈希值来决定服务器的分配。它可以是文本，变量或者文本和变量的组合。

修改 均衡算法 (**balancing algorithm**) ，应该像下面那样：

```
# http context

upstream backend_hosts {

    least_conn;

    server host1.example.com;
    server host2.example.com;
    server host3.example.com;
}

. . .
```

上述例子中，拥有最少连接数的服务器将会优先选择。ip\_hash 指令也可以用同样的方式设置，来保证 **session**（会话）的一致性。

至于 hash 方法，你应该提供要哈希的键。可以是任何你想要的：

```
# http context

upstream backend_hosts {

    hash $remote_addr$remote_port consistent;

    server host1.example.com;
    server host2.example.com;
    server host3.example.com;
}

. . .
```

上述的例子，**request** 的分发是基于客户端的 IP 和端口；我们还可以添加另外的参数 **consistent**，它实现了 **ketama consistent hashing** 算法。基本上，它意味着如果你的 **upstream** 服务器改变了，可以保证对 **cache**（缓存）的最小影响。

## 设置服务器权重

在后端服务器声明中，每一台的服务器默认是权重平分的。它假定每一台服务器都能且应该处理同一量级的负载（考虑到 均衡算法（**balancing algorithm**）的影响）。然而，你也可以为服务器设置其它的权重。

```
# http context

upstream backend_hosts {
    server host1.example.com weight=3;
    server host2.example.com;
    server host3.example.com;
}

. . .
```

上述例子中，`host1.example.com` 可以比其它服务器多接受 2 倍的流量。默认，每一台服务器的权重都是 1。

## 使用 Buffer 缓解后端的负载

对于大部分使用代理的用户来说，最关心的事情之一就是增加一台服务器对性能的影响。在大部分场景下，利用 **Nginx** 的 **buffer**（缓冲）和 **cache**（缓存）能力，可以大大地减轻负担。

在代理过程中，两个连接速度不一会对客户端的体验带来不良的影响：

- 从客户端到代理服务器的连接
- 从代理服务器到后端服务器的连接

**Nginx** 可以根据你希望优化哪一个连接来调整它的行为。

没有 **buffer**（缓冲），数据将会直接从代理服务器传输到客户端。如果客户端的速度足够快（假设），你可以直接把 **buffer**（缓冲）关掉，让数据尽可能快速地到达；如果使用 **buffer**（缓冲），**Nginx** 将会临时存储后端 **response**（响应），然后慢慢把数据推送给客户端；如果客户端很慢，**Nginx** 会提前关闭后端服务器的连接。它可以任意控制分发的节奏。

**Nginx** 默认的 **buffer**（缓冲）设计的初衷就是因为客户端之间速度存在差异。我们可以使用如下指令来调整 **buffer**（缓冲）速度。你可以把 **buffer**（缓冲）配置在 `http`，`server` 或者 `location` 的 **context** 中。必须注意指令为每一个，**request** 配置的大小；在客户端的 **request** 很多的情况下，如果把值调的过大，会很影响性能：

- `proxy_buffering`：这个指令控制所在 **context** 或者子 **context** 的 **buffer**（缓冲）是否打开。默认是 `on`。
- `proxy_buffers`：这个指令控制 **buffer**（缓冲）的数量（第一个参数）和大小（第二个参数）。默认是 8 个 **buffer**（缓冲），每个 **buffer**（缓冲）大小是 1 个内存页（4k 或 8k）。增加 **buffer** 的数量可以缓冲更多的信息。
- `proxy_buffer_size`：是来自后端服务器 **response** 信息的一部分，它包含 **Headers**，从 **response** 分离出来。这个指令设置 **response** 的缓冲。默认，它和 `proxy_buffers` 一样，但是因为它仅用于 **Headers**，所有它的值一般设置得更低。
- `proxy_busy_buffer_size`：这个指令设置忙时 **buffer**（缓冲）的最大值。一个客户端一次只能从一个 **buffer**（缓冲）中读取数据的同时，剩下的 **buffer**（缓冲）会被放到队列中，等待发送到客户端。这个指令控制在这个状态下 **buffer**（缓冲）的空间大小
- `proxy_max_temp_file_size`：当代理服务器的 **response** 太大超出配置的 **buffer**（缓冲）的时候，它来控制 **Nginx** 单次可以写入临时文件的最大数据量。
- `proxy_temp_path`：当代理服务器的 **response** 太大超出配置的 **buffer**（缓冲）的时候，**Nginx** 写临时文件的路径。

正如你看到的，**Nginx** 提供了这几个指令来调整 **buffer**（缓冲）行为。大部分时间，你不需要关心这些指令中的大部分；但是它们中的一些会很有用，可能最有用的就是 `proxy_buffer` 和 `proxy_buffer_size` 这两个指令。

下面这个例子中增加每个 **upstream** 可用代理 **buffer**（缓冲）数，减少存储 **Headers** 的 **buffer**（缓冲）数：

```
# server context

proxy_buffering on;
proxy_buffer_size 1k;
proxy_buffers 24 4k;
proxy_busy_buffers_size 8k;
proxy_max_temp_file_size 2048m;
proxy_temp_file_write_size 32k;

location / {
    proxy_pass http://example.com;
}
```

相反，如果你的客户端足够快到你可以直接传输数据，你就可以完全关掉 **buffer**（缓冲）。实际上，即使 **upstream** 比客户端快很多，**Nginx** 还是会使用 **buffer**（缓冲）的，但是它会直接清空客户端的数据，不会让它进入到 **buffer**（缓冲）池子。如果客户端很慢，这会导致 **upstream** 连接会一直开到客户端处理完为止。当 **buffer**（缓冲）被设置为 `off` 的时候，只有 `proxy_buffer_size` 指令定义的 **buffer**（缓冲）会被使用。

```
# server context

proxy_buffering off;
proxy_buffer_size 4k;

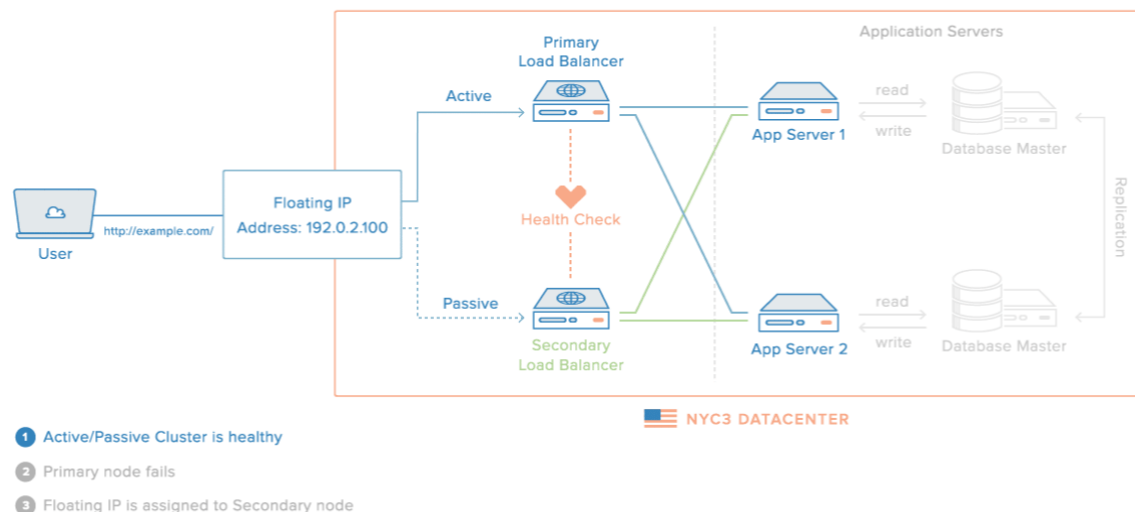
location / {
    proxy_pass http://example.com;
}
```

## 高可用性（可选）

你可以通过添加一个冗余的负载均衡器来使 **Nginx** 代理更加健壮，创建一个高可用性基础设施。

一个高可用（**HA**）的配置是一种容许单点错误（single point of failure）的基础设施，你的负载均衡是这个配置的一部分。当你的负载均衡器不可用或者你需要下线维护，你可以通过配置多个负载均衡器防止潜在的停机风险。

这是基本高可用架构图：



这个例子中，在静态 IP（它可以映射到一台或多台服务器）背后配置多个负载均衡器（一个是激活的，其它的一或多个是被动激活的）。客户端 **request** 从静态 IP 路由到激活的负载均衡器，然后到后端服务器。想了解更多，请阅读 [this section of How To Use Floating IPs](https://www.digitalocean.com/community/tutorials/how-to-use-floating-ips-on-digitalocean#how-to-implement-an-ha-setup) (https://www.digitalocean.com/community/tutorials/how-to-use-floating-ips-on-digitalocean#how-to-implement-an-ha-setup)。

## 配置代理缓存来减少响应时间

**buffer**（缓冲）帮助减轻后端服务器负担达到处理更多 **request** 目的的同时，**Nginx** 还提供一个从后端服务器缓存内容的功能，减少要连接 **upstream** 的次数。

## 配置代理缓存

我们使用 `proxy_cache_path` 指令来为代理的内容设置缓存。它会创建一个直接用于代理服务器返回的数据存储区域。`proxy_cache_path` 指令必须在 `http` 的 **context** 中设置。

下面例子中，我们将会配置这个和相关指令来设置我们的缓存系统。

```
# http context

proxy_cache_path /var/lib/nginx/cache levels=1:2 keys_zone=backcache:8m max_size=50m;
proxy_cache_key "$scheme$request_method$host$request_uri$is_args$args";
proxy_cache_valid 200 302 10m;
proxy_cache_valid 404 1m;
```

我们可以使用 `proxy_cache_path` 指令来定义缓存的存储路径。在这个例子，我使用 `/var/lib/nginx/cache` 这个路径。如果这个路径不存在，你需要创建这个目录，并赋予正确的权限：

```
sudo mkdir -p /var/lib/nginx/cache
sudo chown www-data /var/lib/nginx/cache
sudo chmod 700 /var/lib/nginx/cache
```

参数 `levels=` 用来指定缓存的组织形式。**Nginx** 将会通过哈希键值创建一个缓存 **key**（在下方配置）。上述我们选择的 level 采用 2 级目录结构，内存空间的大小是 8m，假设我们的哈希键值为 `b7f54b2df7773722d382f4809d65087c`，那存储该内容的目录结构是：`/var/lib/nginx/cache/backcache/c/87/b7f54b2df7773722d382f4809d65087c`，注意到规

律没有？参数 `keys_zone=` 定义缓存区域（我们称之为 `backzone`）的名称。这个也是我们定义存储多少元数据的地方。在这个场景中，我们存储 8 MB 的键。**Nginx** 将每一兆会存储 8000 个实体。参数 `max_size` 用来定义实际缓存数据的最大尺寸。

现在，我归纳下：

```
proxy_cache_path {cache_root:缓存路径}
    levels={n:从缓存键值倒数n个字符作为一级目录}:{m:从缓存键值倒数第 n 个字符开始 m 个字符作为二级目录}
    keys_zone={cache_name:该缓存在缓存路径的目录名}:8m;
```

最终该键缓存的目录结构是：

```
{cache_root}/{cache_name}/{数字:从缓存键值倒数n个字符作为一级目录}/{从缓存键值倒数n个字符}/{从缓存键值倒数第 n 个字符开始 m 个字符作为二级目录}
```

上面我们使用的另一个指令就是 `proxy_cache_key`。它用来设置用来存储缓存值的键。

`proxy_cache_valid` 指令可以被指定多次。它允许我们基于不同状态码存储不同值。在我们的例子中，我们存储 **200** 状态（成功）和 **302** 状态（重定向）缓存时间为 10 分钟，和 **404** 状态为 1 分钟后清除缓存。

现在，我们已经配置好缓存区域，但是我们仍然需要告诉 **Nginx** 在哪里使用缓存。

在我们定义代理到后端的 `location` 中，我们配置缓存的使用：



```
# server context

location /proxy-me {
    proxy_cache backcache;
    proxy_cache_bypass $http_cache_control;
    add_header X-Proxy-Cache $upstream_cache_status;

    proxy_pass http://backend;
}

. . .
```

使用 `proxy_cache` 指令，我们可以指定所在 **context** 可以使用 `backcache` 的缓存区域。**Nginx** 将在传递到后端之前检查可用的缓存实体。

`proxy_cache_bypass` 指令用来设置 `$http_cache_control` 变量。它告知代理服务器发请求的客户端是否需要请求一个新鲜，未缓存版本的资源。

我们还可以增加一个多余 **Headers** 信息（`X-Proxy-Cache`）。我们把这个 **Headers** 设置成 `$upstream_cache_status`。它设置 **Headers** 来告知用户该 **request** 的缓存是否被命中，丢失，或者被绕过。在 debug 的时候，该配置特别有用；并且对客户端来说也很重要。

## 缓存结果的注意事项

缓存可以极大地提高代理服务器的性能。然而，配置缓存的时候，还是要需要考虑挺多的东西的。

首先，任何用户相关的数据都不应该被缓存。因为这样会导致一个用户数据的结果被呈现到另一个用户。如果你的站点是纯静态的，那这可能就不是问题了。

如果你的站点有些动态元素，那你就需要在后端服务器考虑到这一点。处理它的方式依赖于后端处理方式。对于隐私内容，你应该把 `Cache-Control` 设置成 `no-cache`, `no-store`, 或者 `private`，这个依赖于数据本身：

- `no-cache` :表示必须先与服务器确认返回的响应是否被更改，然后才能使用该响应来满足后续对同一个网址的请求。因此，如果存在合适的验证令牌 (ETag)，`no-cache` 会发起往返通信来验证缓存的响应，如果资源未被更改，可以避免下载。
- `no-store` : 直接禁止浏览器和所有中继缓存存储返回的任何版本的响应 - 例如：一个包含个人隐私数据或银行数据的响应。每次用户请求该资源时，都会向服务器发送一个请求，每次都会下载完整的响应。
- `private` :浏览器可以缓存`private`响应，但是通常只为单个用户缓存，因此，不允许任何中继缓存对其进行缓存 - 例如，用户浏览器可以缓存包含用户私人信息的 HTML 网页，但是 CDN 不能缓存。这个在缓存用户浏览器数据的时候很有用，但是代理服务器不会在后续的请求中承认数据的有效性。
- `public` :说明请求的是公共信息，它可以被任意的缓存。

控制这个行为相关的 **Headers** 还有 `max-age`，它控制资源缓存的过期时间。

根据数据的敏感度，正确地设置这些 **Headers**，将会帮助你有效地利用缓存，既能保障你的隐私数据安全的同时，还能让动态内容进行有效地刷新。

如果你的后端服务器也使用 **Nginx**，你可以像下面这样使用 `expires` 指令，它将会为 `Cache-Control` 设置 `max-age`：

```
location / {  
    expires 60m;          # 给请求的 Header 添加 Cache-Control:max-age=3660;  
}  
  
location /check-me {  
    expires -1; # 给请求的 Headers 添加 Cache-Control:no-cache;  
}
```

在上面的例子中, 第一个块允许内容被缓存 60 分钟。第二个块则把 Cache-Control 头设置成 no-cache。还想设置其他值, 你可以使用 add\_header 指令:

```
location /private {  
    expires -1;  
    add_header Cache-Control "no-store";  
    # 给请求的 Headers 添加 Cache-Control:no-cache, no-store;  
}
```


## 结语

**Nginx** 是第一个也是最重要的反向代理服务器, 还可以作为 Web 服务器使用。因为这样的设计决策, 代理请求到另一个服务器变得更简单。**Nginx** 也足够灵活, 允许你根据需求对代理配置进行灵活的控制。

### 参考文献:

- Understanding Nginx HTTP Proxying, Load Balancing, Buffering, and Caching (<https://www.digitalocean.com/community/tutorials/understanding-nginx-http-proxying-load-balancing-buffering-and-caching>)
- Google Developers Logo- HTTP 缓存 (<https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=zh-cn>)
- 13 Caching in HTTP (<https://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html>)

 [Nginx <sup>1</sup> \(/n3xtchen/categories.html#Nginx-ref\)](/n3xtchen/categories.html#Nginx-ref)

 [nginx <sup>2</sup> \(/n3xtchen/tags.html#nginx-ref\)](/n3xtchen/tags.html#nginx-ref)

← Previous

[\(/n3xtchen/swift/2016/02/14/swift-tut4/\)](/n3xtchen/swift/2016/02/14/swift-tut4/)

[\(/n3xtchen/javascript/2016/03/08/reactd3/\)](/n3xtchen/javascript/2016/03/08/reactd3/)

© 2019 n3xtchen with help from Jekyll Bootstrap (<http://jekyllbootstrap.com>) and Twitter Bootstrap (<http://twitter.github.com/bootstrap/>)