Nginx 从入门到企业实战（四）

Nginx 企业级用法

1. Nginx 与 FastCGI 构建高速 Web 站点

我们在生产运维中最熟悉的名词莫过于 `LNMP` 或者 `LAMP`，其中 `N` 表示 `Nginx`，而 `P` 则表示 `PHP`，在之前 `LAMP` 的环境中，`Apache` 也就是 `httpd` 支持通过模块的方式调用 `php` 程序，但是这种性能较差，后来出现了 `FastCGI`，而 `php-fpm` 就是 `FastCGI` 的管理器，我们可以在 `PHP 5.3.2` 及更新版本中直接开启并使用即可。

`PHP-CGI` 解释器每进程消耗 `7至25兆` 内存，生产中我们一般按照 `30M` 来估计一个进程占用内存，根据机器的内存空间，来进行进程参数的调整。

> **`FastCGI` 是一个常驻型的 `CGI(Common Gateway Interface)`，它可以一直执行，只要激活后，不会每次都要花费时间去 `fork` 一次（这是 `CGI` 最为人诟病的 `fork-and-execute` 模式）。它还支持分布式的运算，即 `FastCGI` 程序可以在网站服务器以外的主机上执行并且接受来自其它网站服务器来的请求。**
>
> **`FastCGI` 其主要行为是将 `CGI` 解释器进程保持在内存中并因此获得较高的性能。我们知道，`CGI` 解释器的反复加载是 `CGI` 性能低下的主要原因，如果 `CGI` 解释器保存在内存中并接受 `FastCGI` 进程管理器调度，那么就可以提供良好的性能、伸缩性等。**

我们不侧重介绍 `php-fpm`，只简单的介绍下安装和简单优化，重点内容还是 `Nginx` 与 `php-fpm` 之间的交互。我们在 `192.168.48.101` 节点上通过 `yum` 源安装 `php-fpm`，目前 `CentOS-7 - Updates` 源中最新版本为 `php-fpm-5.4.16-46.1.el7_7.x86_64`。

```
[root@app-php ~]# rpm -ql php-fpm-5.4.16-46.1.el7_7.x86_64
/etc/logrotate.d/php-fpm
/etc/php-fpm.conf
/etc/php-fpm.d
/etc/php-fpm.d/www.conf
/etc/sysconfig/php-fpm
/run/php-fpm
/usr/lib/systemd/system/php-fpm.service
/usr/lib/tmpfiles.d/php-fpm.conf
/usr/sbin/php-fpm
/usr/share/doc/php-fpm-5.4.16
/usr/share/doc/php-fpm-5.4.16/fpm_LICENSE
/usr/share/doc/php-fpm-5.4.16/php-fpm.conf.default
/usr/share/fpm
/usr/share/fpm/status.html
/usr/share/man/man8/php-fpm.8.gz
/var/log/php-fpm
```

其中 `/etc/php-fpm.conf` 为 `php-fpm` 的主配置文件，但是其中 `php-fpm.d/www.conf` 才是我们需要关注的。有几个参数需要我们关注：

```
listen = 127.0.0.1:9000
```

设定 `php-fpm` 监听地址，如果是非本地访问，那么需要修改此监听地址。

```
listen.allowed_clients = 127.0.0.1
```

设置允许连接 `php-fpm` 的客户端地址，多个 `IP` 地址需要使用逗号 `,` 隔开，也可以设置为空或者 `any` 表示允许所有地址连接。

```
user = apache
group = apache
```

设置启用 `php-fpm` 进程的用户和组。

```
pm = static|dynamic
```

设置进程管理方式，可以选择静态 `static` 或者动态 `dynamic`，一般在服务器内存比较小的情况下，选择使用 `dynamic` ，而如果内存比较充足，又是专用 `php-fpm` 服务器，可以使用 `static` ，性能会更好一些。

```
pm.max_children = 50
```

最大 `php-fpm` 子进程数，如果是 `pm = static` ，该值表示 `static` 固定的子进程数。

```
pm.start_servers = 5
```

设置初始启动时的 `php-fpm` 子进程数，默认情况下， `pm.start_servers` = `min_spare_servers + (max_spare_servers - min_spare_servers) / 2`

```
pm.min_spare_servers = 5
```

设置空闲进程数最小值，如果空闲进程小于此值，则创建新的子进程。

```
pm.max_spare_servers = 35
```

设置空闲进程数最大值，如果空闲进程大于此值，则进行清理。

```
pm.max_requests = 500
```

设置每个子进程重新创建之前服务的请求数。即当一个 `PHP-CGI` 进程处理的请求数累积到 `500` 个后，自动重启该进程。对于可能存在内存泄漏的第三方模块来说是非常有用的。如果设置为 `0` 则一直接受请求。

```
pm.status_path = /status
```

设置 `php-fpm` 状态页面的网址。如果没有设置，则无法访问状态页面。`Default Value: not set`

```
ping.path = /ping
```

设置 `php-fpm` 监控页面的 `ping` 网址， 如果没有设置，则无法访问 `ping` 页面。

```
ping.response = pong
```

用于定义 `ping` 请求的返回相应，返回为 `HTTP 200` 的 `text/plain` 格式文本，默认值: `pong` 。

```
request_slowlog_timeout = 0
```

当一个请求超过该设置的超时时间后，就会将对应的 `PHP` 调用堆栈信息完整写入到慢日志中。设置为 `0` 表示 `off`

```
request_terminate_timeout = 0
```

设置单个请求的超时中止时间。该选项可能会对 `php.ini` 设置中的 `max_execution_time` 因为某些特殊原因没有中止运行的脚本有用。设置为 `0` 表示 `off` 。当经常出现 `502` 错误时可以尝试更改此选项。

> **注意：此时 Nginx 作为代理服务器响应给客户端 502 代码，是因为 php 进程执行超时，被强制终止，导致无法响应 Nginx 代理进程。**

```
rlimit_files = 1024
```

设置文件打开描述符的 `rlimit` 限制，可以使用 `ulimit -n` 查看。

示例配置：

```
[www]
listen = 9009
listen.allowed_clients = 127.0.0.1,192.168.48.100
user = apache
group = apache
pm = dynamic
pm.max_children = 50
pm.start_servers = 20
pm.min_spare_servers = 5
```

```
pm.max_spare_servers = 35
pm.max_requests = 10240
request_terminate_timeout = 10
request_slowlog_timeout = 10
slowlog = /var/log/php-fpm/www-slow.log
php_admin_value[error_log] = /var/log/php-fpm/www-error.log
php_admin_flag[log_errors] = on
php_value[session.save_handler] = files
php_value[session.save_path] = /var/lib/php/session
```

重启或者重载 `php-fpm` 进程。关于 `php-fpm` 的知识我们先简单介绍到这。

`Nginx` 不像 `httpd` 可以通过模块调用 `php` 程序，他需要通过和 `php-fpm` 进程进行通信，来实现 `php` 代码的调用。而在 `Nginx` 中，我们通过 `ngx_http_fastcgi_module` 模块与 `php-fpm` 进程来实现通信交互。该模块的很多用法、指令都和 `ngx_http_proxy_module` 模块非常接近，其实两个模块就是为了和后端不同协议（`http` & `FastCGI`）进行交互而设计的。我们直接给出一个生产案例配置，我相信有了前面 `ngx_http_proxy_module` 模块的学习，理解这个案例应该很简单。

```
proxy_cache_path            /var/cache/nginx/proxy_cache
                            levels=1:2 keys_zone=proxycache:20m
                            inactive=300s max_size=1g;
proxy_connect_timeout       15s;
proxy_send_timeout          10s;
proxy_read_timeout          10s;
#-------------------------------------------------------------
fastcgi_cache_path          /var/cache/nginx/fcig_cache
                            levels=1:2:1 keys_zone=fcgicache:20m
                            inactive=300s max_size=1g;
fastcgi_connect_timeout     15s;
fastcgi_send_timeout        15s;
fastcgi_read_timeout        15s;
fastcgi_keep_conn           on;
server {
    listen                  80 default_server;
    server_name             www.a.com a.com;
    root                    /data/nginx/a;
    charset                 utf-8;

    location / {
        index               index.html;
    }
    location  /api/ {
        proxy_pass          http://192.168.48.101:8080;
        proxy_cache         proxycache;
        proxy_cache_valid   200 302 10m;
        proxy_cache_valid   301 1h;
        proxy_cache_valid   any 1m;
    }
    location ~* \.php$ {
```

```
        fastcgi_pass        192.168.48.101:9009;
        fastcgi_index       index.php;
        fastcgi_param       SCRIPT_FILENAME    /data/nginx/app-
php/$fastcgi_script_name;
        include             /etc/nginx/fastcgi_params;
        fastcgi_cache       fcgicache;
        fastcgi_cache_key   $request_uri;
        fastcgi_cache_valid 200 302 10m;
        fastcgi_cache_valid 301 1h;
        fastcgi_cache_valid any 1m;
    }
}
```

这里有几个需要补充的参数，需要跟大家介绍下：

fastcgi_pass：设置后端 `FastCGI` 服务器的地址，注意后端支持 `地址+端口` 或者 `Unix Socket`，但是不要加 `http(s)` 或者后面加上 `uri`，因为这代理的是 `FastCGI` 协议，而不是 `http(s)` 协议。基于语法：

```
Syntax: fastcgi_pass address;
Default:  —
Context:  location, if in location
```

示例：

```
fastcgi_pass localhost:9000;
```

---

fastcgi_index：用来设置 `FastCGI` 服务器默认的主页资源，跟我们前面介绍的 `index` 类似。基本语法：

```
Syntax: fastcgi_index name;
Default:  —
Context:  http, server, location
```

示例：

```
fastcgi_index index.php;
```

---

fastcgi_param：设置发往后端 `FastCGI` 服务器的参数，需要设置的参数很多，不过绝大多数都已经被写入到默认配置文件中了。其中有一条参数非常非常重要，即：

核心传参：`fastcgi_param SCRIPT_FILENAME /path/to/PHP-FILE/$fastcgi_script_name;` 非常重要

注意：后面的路径一定要写你的 `PHP` 代码所在的位置，其中 `$fastcgi_script_name` 会是你访问的 `php` 文件名。

其他的一些参数默认已经被放在 `/etc/nginx/fastcgi_params` 文件中了，可以使用绝对路径，也可以使用相对路径（相对于 `/etc/nginx` ）。基本语法：

```
Syntax: fastcgi_param parameter value [if_not_empty];
Default:  —
Context:  http, server, location
```

示例配置：

```
fastcgi_param    SCRIPT_FILENAME    /data/nginx/app-php/$fastcgi_script_name;
include          /etc/nginx/fastcgi_params;
```

---

fastcgi_keep_conn ：用来设置在收到后端服务器响应时，是否立刻关闭连接。默认是 `off` ，最好修改为 `on` ，表示与后端服务器开启长连接。基本语法：

```
Syntax: fastcgi_keep_conn on | off;
Default:  fastcgi_keep_conn off;
Context:  http, server, location
This directive appeared in version 1.1.4.
```

fastcgi_cache_key ：这个参数需要提及一下，因为在前面 `proxy_cache_key` 是有默认值的，但是在 `fastcgi_module` 中没有默认值，需要大家手动设置。基本语法：

```
Syntax: fastcgi_cache_key string;
Default:  —
Context:  http, server, location
```

比如，你可以这样设置：

```
fastcgi_cache_key localhost:9000$request_uri;
```

其他的一些参数这里就不一一介绍了，需要注意的是，开启 `fastcgi_cache` 会极大的提升性能。生产中一般会通过脚本程序每天定时获取一次页面，以便生产缓存数据（缓存下来的数据为静态页面），提升用户访问速度。

---

2. 配置 Nginx 后端集群环境

在并发量较大的环境下，一般我们会选择横向升级多个节点来解决访问压力问题，在 `Nginx` 代理设置中，其实也有相应的模块来设定多后端节点。前面我们跟大家演示的都是单台 `Web` 或者 单台后端节点，下面我们就为大家演示后端集群的代理配置。这里需要引入模块 **`ngx_http_upstream_module`** 。注意该模块是用来定义后端一组服务器，可以支持的协议除了 `http（proxy_pass）` 外还有 `FastCGI(fastcgi_pass)` 、uwsgi_pass, scgi_pass, memcached_pass 等代理协议。配置示例：

```
upstream backend {
    server backend1.example.com       weight=5;
    server backend2.example.com:8080;
    server unix:/tmp/backend3;

    server backup1.example.com:8080   backup;
    server backup2.example.com:8080   backup;
}

server {
    location / {
        proxy_pass http://backend;
    }
}
```

上述配置为基本用法，我们将对一些常见的配置参数进行介绍。

upstream：用来定义一组服务器。这些服务器可以监听不同的端口。而且，监听在 `TCP` 和 `UNIX` 域套接字的服务器可以混用。基本语法：

```
Syntax: upstream name { ... }
Default:  —
Context:  http
```

示例配置：

```
upstream backend {
    server backend1.example.com weight=5;
    server 127.0.0.1:8080       max_fails=3 fail_timeout=30s;
    server unix:/tmp/backend3;

    server backup1.example.com  backup;
}
```

默认情况下，`nginx` 按加权轮转的方式将请求分发到各服务器。在上面的例子中，每 `7` 个请求会通过以下方式分发：`5` 个请求分到 `backend1.example.com`，一个请求分到第二个服务器，一个请求分到第三个服务器。与服务器通信的时候，如果出现错误，请求会被传给下一个服务器，直到所有可用的服务器都被尝试过。如果所有服务器都返回失败，客户端将会得到最后通信的那个服务器的（失败）响应结果。

---

server：定义服务器的地址 `address` 和其他参数 `parameters`。地址可以是域名或者 `IP` 地址，端口是可选的，或者是指定 `unix:` 前缀的 `UNIX` 域套接字的路径。如果没有指定端口，就使用 `80` 端口。如果一个域名解析到多个 `IP`，本质上是定义了多个 `server`。基本语法：

```
Syntax: server address [parameters];
Default:  —
Context:  upstream
```

可用参数：

`weight`：设定服务器的权重，默认是 `1`。

`max_conns`：设置后端服务器最大活动连接数，在 `Nginx 1.11.5` 版本之后支持，默认为 `0` 表示不限制。

`max_fails`：设定 `Nginx` 与后端服务器通信的尝试失败的次数。在 `fail_timeout` 参数定义的时间段内，如果失败的次数达到此值，`Nginx` 就认为服务器不可用。在下一个 `fail_timeout` 时间段，服务器不会再被尝试。 失败的尝试次数默认是 `1`。设为 `0` 就会停止统计尝试次数，认为服务器是一直可用的。

`fail_timeout`：用来设定下面两个时间值：

- 统计失败尝试次数的时间段。在这段时间中，服务器失败次数达到 `max_fails` 指定的尝试次数，服务器就被认为不可用。
- 服务器被认为不可用的时间段，即一旦被标记为不可用，则在下一个 `fail_timeout` 时间段内都不会再尝试与此服务器进行连接。

`backup`：标记为备用服务器。当所有的主服务器不可用以后，请求会被传给这些备用服务器。该参数不可与 `hash`, `ip_hash`, and `random` 等负载均衡方法一起使用。

`down`：标记服务器为永久不可用，可以配合 `ip_hash` 实现灰度发布。

还有些参数不长使用，我们就不介绍了，感兴趣的同学可以参考 官方文档。

---

hash：基于某个 `key` 的 `hash` 表实现请求的后端调度，此处的 `key` 可以是文本、变量或者二者的结合。

基本语法：

```
Syntax: hash key [consistent];
Default:  —
Context:  upstream
This directive appeared in version 1.7.2.
```

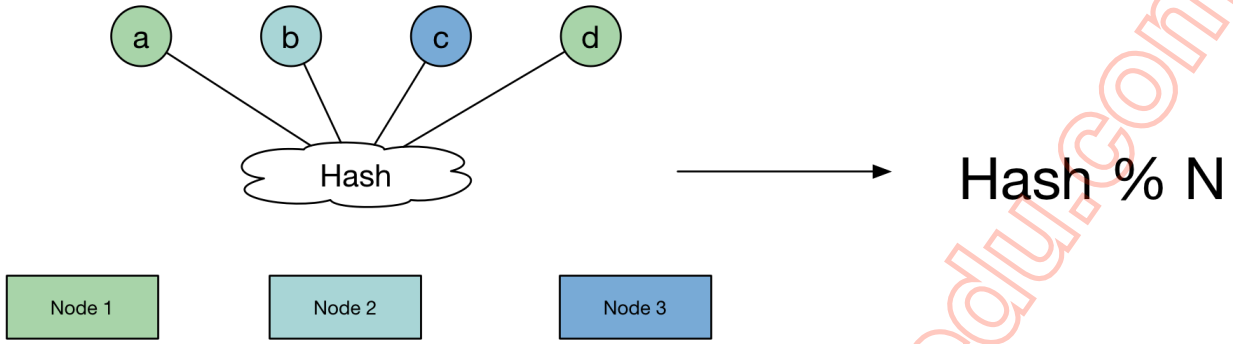
注意：该指令的核心功能可以让我们队请求进行分类，让同一种请求发往到同一个 `upstream server`，非常实用与后端为 **缓存服务器**（比如 `squid` 或者 `varnish`）的情况。比如我们可以这样配置：

```
hash $request_uri;
```

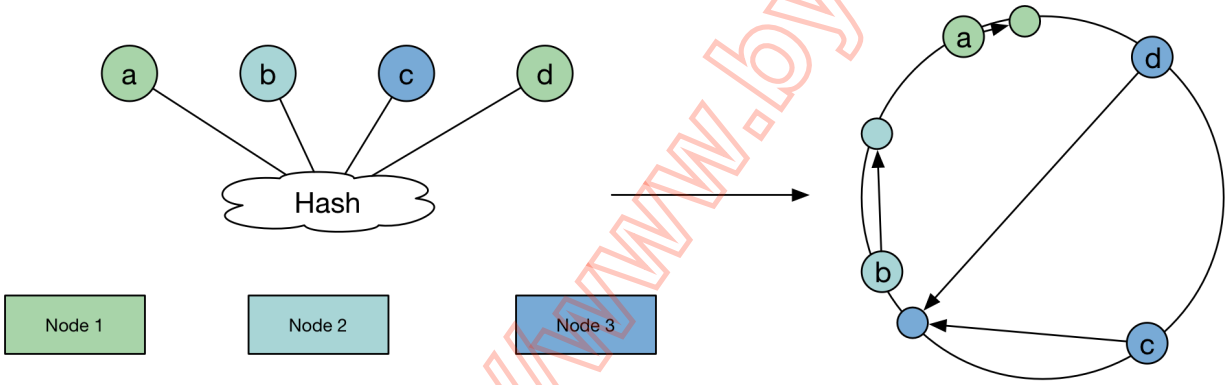
但是这样配置有一个核心问题，一旦后端新增或者删除某个节点，可能导致所有缓存都无法命中，造成缓存穿透，出现“雪崩”等重点故障。不过，也不用担心，我们可以通过添加参数 `consistent` 参数来实现一致性 `hash` 算法，将影响讲到最低。

```
hash $request_uri consistent;
```

关于 `hash` 算法和一致性 `hash` 算法介绍：



普通 `hash` 算法，通过将用户请求计算出 `hash` 值，然后将该 `hash` 值与节点数量 `N` 取余，得出请求落到的节点位置。算法非常精确，但是一旦节点数据发生变化，可能导致所有缓存数据失效——**于是引入了 "一致性 hash 算法"**。

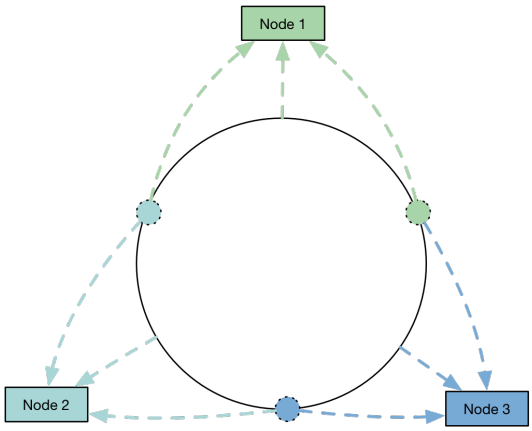


通过引入了 `hash 环` （`2^32` 个位置）的概念，这里涉及到两次 `hash` 运算，一次是对用户请求进行 `hash` 运算，得出的值与 `2^32` 取余，会落在环上的某个位置，该位置固定。另外一次是对节点地址加权后进行 `hash` 运算，比如：

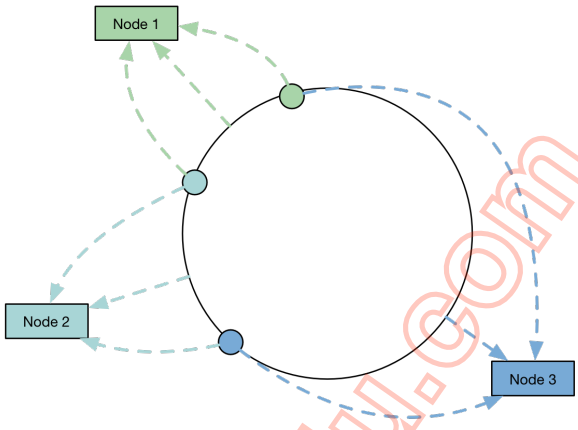
`server A weight=1` 、`server B weight=2` 、`server C weight=3` ，那么节点 `A` 就会对 `hash(A-IP)` 得出一个 `hash` 值；对于节点 `B` ，就会对 `hash(B-IP)` 和 `hash(B-IP + 1)` 计算，得出两个 `hash` 值，依次类推节点 `C` 。然后三个节点的 `hash` 值再与 `2^32` 求余，得出的值也一定会落在 `hash` 环，且出现的概率与权重有直接关系。然后将请求数据计算出的落点顺时针移动，离该落点最近的节点 `hash` 求余后的落点，即为请求调度的后端节点。

这样即使一个节点失效，影响的只是离该节点位置最近的一些请求落点，而不会导致全局后端节点失效。

解决了这个问题，大家很快发现还有一个问题比较严重，如下：

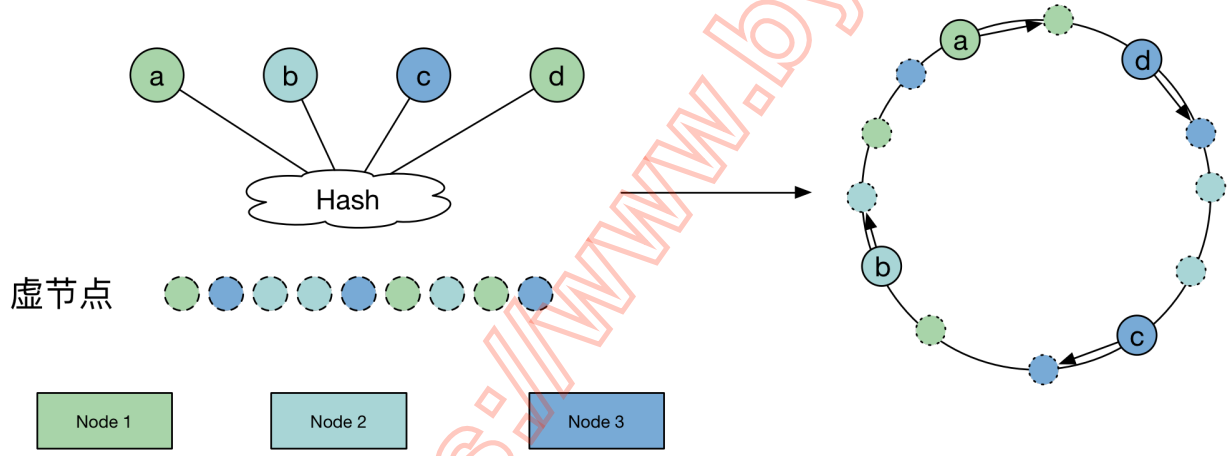


理想情况



实际情况

即：理想情况下，对节点的 `hash` 计算落地是均匀分布在 `hash` 环上，这样可以保证请求调度到各个节点上的几率趋于均衡状态，但是实际情况几乎不可能实现这种均匀分布，而且极端情况下可能会有某个节点（或者多个节点）的多权重位置相近，这就导致调度请求的极不平衡，那这种情况该如何解决呢？



即：通过引入虚拟节点的方式来实现。我们知道分配不均衡很大程度上是因为数据量太小（就像抛硬币，当抛的次数越多，正反两面的概率就越接近 `1:1` 平衡），那么我们就可以通过按比例增加服务器的权重来实现这种平衡性。即把服务器的权重全部乘以 `100`，那么各服务器之间的权重比率还是不变的。这样就使得我们的 `hash` 环比较平衡。

大家对一致性 `hash` 算法不了解的，也可以参考这篇文章 白话解析：一致性 `hash` 算法

---

ip_hash：指定服务器组的负载均衡（调度）方法，请求基于客户端的IP地址在服务器间进行分发。`IPv4` 地址的前三个字节或者 `IPv6` 的整个地址，会被用来作为一个散列 `key`。这种方法可以确保从同一个客户端过来的请求，会被传给同一台服务器。除了当服务器被认为不可用的时候，这些客户端的请求会被传给其他服务器，而且很有可能也是同一台服务器。基本语法：

```
Syntax: ip_hash;
Default:  —
Context:  upstream
```

> **注意1：从 `1.3.2` 和 `1.2.2` 版本开始支持 `IPv6` 地址。**
>
> **注意2：从 `1.3.1` 和 `1.2.2` 版本开始，`ip_hash` 的负载均衡方法才支持设置服务器权重值。**

如果其中一个服务器想暂时移除或者进行灰度发布，应该加上 `down` 参数。这样可以保留当前客户端 `IP` 地址散列分布。

例子：

```
upstream backend {
 ip_hash;

 server backend1.example.com;
 server backend2.example.com;
 server backend3.example.com down;
 server backend4.example.com;
}
```

注意：`ip_hash` 也是一致性 `hash` 算法，主要区别是这里以客户端源 `IP` 地址（而非 `hash consistent` 中用户请求的 `key`）进行 `hash` 运算。

---

keepalive：为每个 `worker` 进程保留的空闲长连接数量，这些长连接会被放入缓存，可以节约 `Nginx`（作为客户端）端口，并减少连接管理的消耗。 如果连接数大于这个值时，最久未使用的连接会被关闭。

> **需要注意的是，`keepalive` 指令不会限制 `Nginx` 进程与上游服务器的连接总数。 新的连接总会按需被创建。 `connections` 参数应该稍微设低一点，以便上游服务器也能处理额外新进来的连接。**

基本语法：

```
Syntax: keepalive connections;
Default:  —
Context:  upstream
This directive appeared in version 1.1.4.
```

另外，我们在 `http` 或者 `FastCGI` 的代理中使用该指令，还有一些需要特别注意的点：

对于 `HTTP` 代理，`proxy_http_version` 指令应该设置为 `1.1`，同时 `Connection` 头的值也应被清空。

```
upstream http_backend {
 server 127.0.0.1:8080;

 keepalive 16;
```

```
 }

 server {
  ...

  location /http/ {
      proxy_pass http://http_backend;
      proxy_http_version 1.1;
      proxy_set_header Connection "";
      ...
  }
 }
```

示例：

```
upstream http_server {
    server                  192.168.48.101:8080 ;
    server                  192.168.48.101:8088 ;
    keepalive               15;
}
server {
    listen                   80 default_server;
    server_name              www.a.com a.com;
    root                     /data/nginx/a;
    charset                  utf-8;

    location / {
        index                index.html;
    }
    location  /api/ {
        proxy_pass           http://http_server;
        proxy_set_header     Connection "";
        proxy_http_version   1.1;
    }
}
```

可以看到请求协议为 `http 1.1`，默认为 `http 1.0`，另外，如果不设置 `proxy_set_header Connection ""`，则会显示 `Connection close`。

```
    192.168.48.100.41960 > 192.168.48.101.8080: Flags [P.], cksum 0xa49d (correct), seq 1:589, ack 1, win 115, opti
ons [nop,nop,TS val 382948122 ecr 159272860], length 588: HTTP, length: 588
      GET /api/index.html HTTP/1.1
      Host: http_server
      Cache-Control: max-age=0
      Upgrade-Insecure-Requests: 1
      User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/8
0.0.3987.162 Safari/537.36 Edg/80.0.361.109
      Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/s
igned-exchange;v=b3;q=0.9
      Accept-Encoding: gzip, deflate
      Accept-Language: zh-Hans-CN,zh-CN;q=0.9,zh;q=0.8,en;q=0.7,en-GB;q=0.6,en-US;q=0.5
      If-None-Match: "5e902eb3-26"
      If-Modified-Since: Fri, 10 Apr 2020 08:30:43 GMT
```

看下响应报文：

```
    192.168.48.101.8080 > 192.168.48.100.41956: Flags [P.], cksum 0xe2f3 (incorrect -> 0xa1da), seq 1:180, ack 563,
 win 236, options [nop,nop,TS val 159212165 ecr 382887427], length 179: HTTP, length: 179
        HTTP/1.1 304 Not Modified
        Server: nginx/1.16.1
        Date: Fri, 10 Apr 2020 08:58:17 GMT
        Last-Modified: Fri, 10 Apr 2020 08:39:35 GMT
        Connection: keep-alive
        ETag: "5e9030c7-25"
```

如果没有设置代理首部和代理协议，则无法使用代理长连接模式，如下：

```
    192.168.48.100.41954 > 192.168.48.101.8080: Flags [P.], cksum 0x427d (correct), seq 1:608, ack 1, win 115, opti
ons [nop,nop,TS val 382601983 ecr 158926722], length 607: HTTP, length: 607
        GET /api/index.html HTTP/1.0
        Host: http server
        Connection: close
        Cache-Control: max-age=0
        Upgrade-Insecure-Requests: 1
        User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/8
0.0.3987.162 Safari/537.36 Edg/80.0.361.109
        Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/s
igned-exchange;v=b3;q=0.9
        Accept-Encoding: gzip, deflate
        Accept-Language: zh-Hans-CN,zh-CN;q=0.9,zh;q=0.8,en;q=0.7,en-GB;q=0.6,en-US;q=0.5
        If-None-Match: "5e902eb3-26"
        If-Modified-Since: Fri, 10 Apr 2020 08:30:43 GMT

16:53:31.786476 IP (tos 0x0, ttl 64, id 39873, offset 0, flags [DF], proto TCP (6), length 52)
    192.168.48.101.8080 > 192.168.48.100.41954: Flags [.], cksum 0xe240 (incorrect -> 0x9fb2), seq 1, ack 608, win
236, options [nop,nop,TS val 158926722 ecr 382601983], length 0
16:53:31.787663 IP (tos 0x0, ttl 64, id 39874, offset 0, flags [DF], proto TCP (6), length 335)
    192.168.48.101.8080 > 192.168.48.100.41954: Flags [FP.], cksum 0xe35b (incorrect -> 0x33db), seq 1:284, ack 608
, win 236, options [nop,nop,TS val 158926723 ecr 382601983], length 283: HTTP, length: 283
        HTTP/1.1 200 OK
        Server: nginx/1.16.1
        Date: Fri, 10 Apr 2020 08:53:31 GMT
        Content-Type: text/html; charset=utf-8
        Content-Length: 37
        Last-Modified: Fri, 10 Apr 2020 08:39:35 GMT
        Connection: close
        ETag: "5e9030c7-25"
        Accept-Ranges: bytes

        <h1>This is app-php.com WebSite</h1>
```

对于 `FastCGI` 的服务器，需要设置 `fastcgi_keep_conn` 指令来让启用 `keepalive` 长连接：

```
upstream fastcgi_backend {
 server 127.0.0.1:9000;

 keepalive 8;
}

server {
 ...

 location /fastcgi/ {
     fastcgi_pass fastcgi_backend;
     fastcgi_keep_conn on;
     ...
 }
}
```

注意1：当使用的负载均衡方法不是默认的轮转法时，必须在 `keepalive` 指令之前配置。

注意2：`SCGI and uwsgi protocols` 还不支持这种长连接形式。

least_conn：指定服务器组的负载均衡方法，根据其权重值，将请求发送到活跃连接数最少的那台服务器。 如果这样的服务器有多台，那就采取有权重的轮转法进行尝试。当我们开启长连接时，可以使用这种调度算法。

基本语法：

```
Syntax: least_conn;
Default:  —
Context:  upstream
This directive appeared in versions 1.3.1 and 1.2.2.
```

---

3. Nginx 动态编译模块

`Nginx 1.9.11` 开始增加加载动态模块支持，从此不再需要替换 `Nginx` 文件即可增加第三方扩展。下面我们来演示动态编译 `echo-nginx-module` 模块，该模块可以直接输出字符串或者变量等信息，非常方便我们调试。下面为执行步骤，以我们通过 `EPEL` 源安装的 `Nginx` 为例，为此模块新增 `echo_nginx_module` 模块。步骤如下：

3.1 下载与 `RPM` 包相同版本的 `Nginx` 源文件

```
[root@app-php ~]# nginx -v
nginx version: nginx/1.16.1
```

下载 `Nginx-1.16.1` 源码包

```
[root@app-php src]# wget http://nginx.org/download/nginx-1.16.1.tar.gz
[root@app-php src]# tar xf nginx-1.16.1.tar.gz
[root@app-php src]# ls
nginx-1.16.1  nginx-1.16.1.tar.gz
```

下载需要编译的模块源码

```
[root@app-php src]# git clone https://github.com/openresty/echo-nginx-
module.git
```

3.2 查看当前 `Nginx` 已经编译的模块

```
[root@app-php src]# nginx -V
nginx version: nginx/1.16.1
built by gcc 4.8.5 20150623 (Red Hat 4.8.5-39) (GCC)
built with OpenSSL 1.0.2k-fips  26 Jan 2017
TLS SNI support enabled
configure arguments: --prefix=/usr/share/nginx --sbin-path=/usr/sbin/nginx --
modules-path=/usr/lib64/nginx/modules --conf-path=/etc/nginx/nginx.conf --
error-log-path=/var/log/nginx/error.log --http-log-
path=/var/log/nginx/access.log --http-client-body-temp-
path=/var/lib/nginx/tmp/client_body --http-proxy-temp-
path=/var/lib/nginx/tmp/proxy --http-fastcgi-temp-
path=/var/lib/nginx/tmp/fastcgi --http-uwsgi-temp-
path=/var/lib/nginx/tmp/uwsgi --http-scgi-temp-path=/var/lib/nginx/tmp/scgi --
pid-path=/run/nginx.pid --lock-path=/run/lock/subsys/nginx --user=nginx --
group=nginx --with-file-aio --with-ipv6 --with-http_ssl_module --with-
http_v2_module --with-http_realip_module --with-stream_ssl_preread_module --
with-http_addition_module --with-http_xslt_module=dynamic --with-
http_image_filter_module=dynamic --with-http_sub_module --with-http_dav_module
--with-http_flv_module --with-http_mp4_module --with-http_gunzip_module --
with-http_gzip_static_module --with-http_random_index_module --with-
http_secure_link_module --with-http_degradation_module --with-
http_slice_module --with-http_stub_status_module --with-
http_perl_module=dynamic --with-http_auth_request_module --with-mail=dynamic -
-with-mail_ssl_module --with-pcre --with-pcre-jit --with-stream=dynamic --
with-stream_ssl_module --with-google_perftools_module --with-debug --with-cc-
opt='-O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstack-
protector-strong --param=ssp-buffer-size=4 -grecord-gcc-switches -
specs=/usr/lib/rpm/redhat/redhat-hardened-cc1 -m64 -mtune=generic' --with-ld-
opt='-Wl,-z,relro -specs=/usr/lib/rpm/redhat/redhat-hardened-ld -Wl,-E'
```

注意：使用源码包编译新的模块需要确保原来 `RPM` 包中已经还有的模块必须编译进去，否则会出现 `is not binary compatible` 的报错。

另外，对于上面的编译参数：

```
--with-cc-opt='-O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -
fstack-protector-strong --param=ssp-buffer-size=4 -grecord-gcc-switches -
specs=/usr/lib/rpm/redhat/redhat-hardened-cc1 -m64 -mtune=generic' --with-ld-
opt='-Wl,-z,relro -specs=/usr/lib/rpm/redhat/redhat-hardened-ld -Wl,-E'
```

`--with-cc-opt` 设置C编译器参数；

`--with-ld-opt` 设置链接文件参数；

这两个参数与红帽设置 `RPM` 文件的一些参数，我们可以通过 `yum install redhat-rpm-config` 来获取，否则直接编译会出错。

3.3 动态编译模块

有一些包需要提前安装好，如下：

```
yum install gcc libxml2 libxml2-devel libxslt libxslt-devel gd gd-devel pcre
pcre-devel perl perl-devel openssl openssl-devel perl-ExtUtils-Embed google-
perftools-devel
```

设置编译参数：

```
./configure --prefix=/usr/share/nginx --sbin-path=/usr/sbin/nginx --modules-
path=/usr/lib64/nginx/modules --conf-path=/etc/nginx/nginx.conf --error-log-
path=/var/log/nginx/error.log --http-log-path=/var/log/nginx/access.log --
http-client-body-temp-path=/var/lib/nginx/tmp/client_body --http-proxy-temp-
path=/var/lib/nginx/tmp/proxy --http-fastcgi-temp-
path=/var/lib/nginx/tmp/fastcgi --http-uwsgi-temp-
path=/var/lib/nginx/tmp/uwsgi --http-scgi-temp-path=/var/lib/nginx/tmp/scgi --
pid-path=/run/nginx.pid --lock-path=/run/lock/subsys/nginx --user=nginx --
group=nginx --with-file-aio --with-ipv6 --with-http_ssl_module --with-
http_v2_module --with-http_realip_module --with-stream_ssl_preread_module --
with-http_addition_module --with-http_xslt_module=dynamic --with-
http_image_filter_module=dynamic --with-http_sub_module --with-http_dav_module
--with-http_flv_module --with-http_mp4_module --with-http_gunzip_module --
with-http_gzip_static_module --with-http_random_index_module --with-
http_secure_link_module --with-http_degradation_module --with-
http_slice_module --with-http_stub_status_module --with-
http_perl_module=dynamic --with-http_auth_request_module --with-mail=dynamic -
-with-mail_ssl_module --with-pcre --with-pcre-jit --with-stream=dynamic --
with-stream_ssl_module --with-google_perftools_module --with-debug --with-cc-
opt='-O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstack-
protector-strong --param=ssp-buffer-size=4 -grecord-gcc-switches -
specs=/usr/lib/rpm/redhat/redhat-hardened-cc1 -m64 -mtune=generic' --with-ld-
opt='-Wl,-z,relro -specs=/usr/lib/rpm/redhat/redhat-hardened-ld -Wl,-E' --add-
dynamic-module=/root/src/echo-nginx-module/
```

当没有错误提示时，才可以进行编译：

```
Configuration summary
  + using system PCRE library
  + using system OpenSSL library
  + using system zlib library

  nginx path prefix: "/usr/share/nginx"
  nginx binary file: "/usr/sbin/nginx"
  nginx modules path: "/usr/lib64/nginx/modules"
  nginx configuration prefix: "/etc/nginx"
  nginx configuration file: "/etc/nginx/nginx.conf"
  nginx pid file: "/run/nginx.pid"
  nginx error log file: "/var/log/nginx/error.log"
  nginx http access log file: "/var/log/nginx/access.log"
  nginx http client request body temporary files:
"/var/lib/nginx/tmp/client_body"
```

```
  nginx http proxy temporary files: "/var/lib/nginx/tmp/proxy"
  nginx http fastcgi temporary files: "/var/lib/nginx/tmp/fastcgi"
  nginx http uwsgi temporary files: "/var/lib/nginx/tmp/uwsgi"
  nginx http scgi temporary files: "/var/lib/nginx/tmp/scgi"

./configure: warning: the "--with-ipv6" option is deprecated
```

重新编译 `Nginx` 模块

```
[root@app-php nginx-1.16.1]# make -j4
```

注意：此时已经编译好我们所需的模块了，千万不要执行 `make install`，否则会覆盖我们原有的 `Nginx`!!!

3.4 拷贝模块并修改配置文件

我们刚才编译好的模块位于 `./objs/` 目录下：

```
[root@app-php nginx-1.16.1]# ls ./objs/ngx_http_echo_module.so
./objs/ngx_http_echo_module.so
```

将该模块拷贝至 `Nginx` 默认模块路径下：

```
[root@app-php nginx-1.16.1]# cp ./objs/ngx_http_echo_module.so
/usr/lib64/nginx/modules/
```

创建模块引入文件：

```
[root@app-php nginx-1.16.1]# echo "load_module
/usr/lib64/nginx/modules/ngx_http_echo_module.so;" >>
/usr/share/nginx/modules/mod-http-echo.conf
```

3.5 重启 `Nginx` 进程

加载动态模块，最好重启 `Nginx` 进程

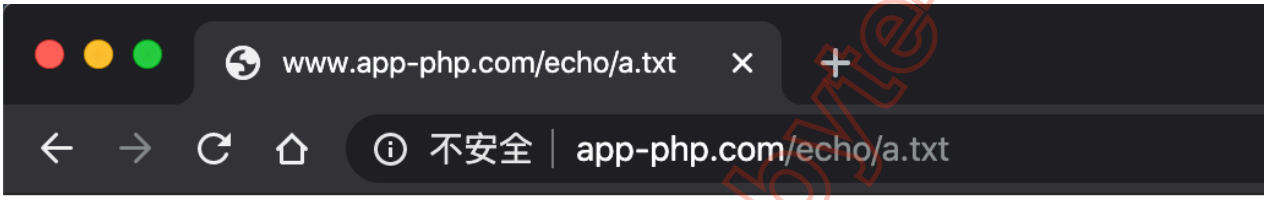
```
[root@app-php nginx-1.16.1]# nginx -s stop
[root@app-php nginx-1.16.1]# nginx
# 或者
[root@app-php nginx-1.16.1]# systemctl restart nginx
```

需要注意：动态加载的模块不支持通过 `nginx -V` 选项查看。

3.6 简单测试下

在测试环境或者大家学习的过程中，建议使用 `echo` 参数，它可以非常方便的调试某些信息，可以直接打印出来，如下：

```
server {
    server_name    www.app-php.com app-php.com;
    listen         80 default_server;
    root           /data/nginx/app-php/;
    charset        utf-8;
    access_log     /var/log/nginx/php-access.log  mylog;
    location  /    {
        root       /data/nginx/app-php;
        index      index.html index.php;
    }
    location  /echo/ {
        echo       "请求信息：$request_uri";
    }
}
```



请求信息：/echo/a.txt

4. 全站 HTTP 跳转 HTTPS 协议

同样，我们以 `http://www.a.com` 为例，要求所有访问该页面的请求全部跳转至 `https://www.a.com/`，且请求的 `URI` 和 参数 `$query_string` 要保留下来。

先给出常见的几种方法：

【使用 `if` 进行协议判断】——最差

这种情况下，多为把 `http` 和 `https` 写在了同一个 `server` 中，配置如下：

```
server {
    listen         80 default_server;
    listen         443 ssl default_server;
    server_name    www.a.com;
    ssl_certificate "/data/nginx/ssl/nginx.crt";
    ssl_certificate_key "/data/nginx/ssl/nginx.key";
    root           /data/nginx/a;
    charset        utf-8;
    if ( $scheme = http ){
        rewrite  ^/(.*)$  https://www.a.com/$1 permanent;
    }
}
```

这种配置看起来简洁很多，但是性能是最差的。首先每次连接进来都需要 `Nginx` 进行协议判断，其次判断为 `http` 协议时进行地址匹配、重写、返回、再次判断，最后还有正则表达式的处理 ... ... 所以，生产上我们极不建议这种写法。另外，能少用 `if` 的尽量不用，如果一定要使用，也最好在 `location` 段，并且结合 `return` 或者 `rewrite ... last` 来使用。

【`rewrite` 方法1】——差

```
server {
    listen        80 default_server;
    server_name   www.a.com a.com;
    rewrite ^/(.*)$  https://www.a.com/$1 permanent;
}
server {
    listen        443 ssl default_server;
    server_name   www.a.com a.com;
    ssl_certificate "/data/nginx/ssl/nginx.crt";
    ssl_certificate_key "/data/nginx/ssl/nginx.key";
    root          /data/nginx/a;
    charset       utf-8;
}
```

请求测试：**`http://www.a.com/a.html?a=3`**

```
❯ curl -I http://www.a.com/a.html\?a\=3
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Fri, 03 Apr 2020 13:03:01 GMT
Content-Type: text/html
Content-Length: 162
Connection: keep-alive
Location: https://www.a.com/a.html?a=3
```

可以看到实现了 `http` 到 `https` 的跳转，并且保留了参数。

【`rewrite` 方法2】——好

不使用正则表达式，而使用变量来提升性能：

```
server {
    listen        80 default_server;
    server_name   www.a.com;
    rewrite ^ https://www.a.com$request_uri? permanent;
}
server {
    listen        443 ssl default_server;
    server_name   www.a.com;
    ssl_certificate "/data/nginx/ssl/nginx.crt";
    ssl_certificate_key "/data/nginx/ssl/nginx.key";
    root          /data/nginx/a;
    charset       utf-8;
}
```

注意：`$request_uri` 已经包含了查询参数，所以要在其重写规则后面加上 `?` 以禁止再次传递参数。这种方法避免了 `Nginx` 内部处理正则的性能损坏，相比较上面的方式好了很多。

请求测试：**`http://www.a.com/a.html?a=3`**

```
❯ curl -I http://a.com/a.html\?a\=3
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Fri, 03 Apr 2020 13:16:16 GMT
Content-Type: text/html
Content-Length: 162
Connection: keep-alive
Location: https://www.a.com/a.html?a=3
```

【使用 `return` 实现最优解】——最好

虽然上面我们使用参数代替了正则，但是 `rewrite` 规则会先对 `URL` 进行匹配，匹配上了再执行相应的规则。而 `return` 没有匹配 `URL` 层面的性能消耗，直接返回用户新的连接，所以是最优的解决方案。

```
server {
    listen        80 default_server;
    server_name   www.a.com;
    return        302  https://www.a.com$request_uri;
}
server {
    listen        443 ssl default_server;
    server_name   www.a.com;
    ssl_certificate "/data/nginx/ssl/nginx.crt";
    ssl_certificate_key "/data/nginx/ssl/nginx.key";
    root          /data/nginx/a;
    charset       utf-8;
}
```

注意：在 `return` 中，`$request_url` 后面不用加 `?`（加 `?` 用来避免携带参数是 `rewrite` 中的特性）。

如果希望实现永久重定向，则使用 `return 301 https://XXX` ，不过我们两个域名都会使用，所以更多情况下使用 `302` 临时重定向。

---

5. 新老域名的替换

比如某些情况下，我们希望使用新的域名，但是一些老用户依然在使用老域名，比如之前京东使用的域名是 `http://www.360buy.com` ，现在使用 `https://www.jd.com` ，如果依然希望依然保留原来的域名，但是想让用户切换到新域名，那么我们就可以这么做。

这里我以 `www.b.com` 作为老域名，新域名为 `www.a.com` ，配置如下：

```
server {
    listen        80;
    server_name   www.b.com b.com;
    return        301  http://www.a.com$request_uri;
}
server {
    listen        80 default_server;
    server_name   www.a.com a.com;
    return        302  https://www.a.com$request_uri;
}
server {
    listen        443 ssl default_server;
    server_name   www.a.com a.com;
    ssl_certificate "/data/nginx/ssl/nginx.crt";
    ssl_certificate_key "/data/nginx/ssl/nginx.key";
    root          /data/nginx/a;
    charset       utf-8;
}
```

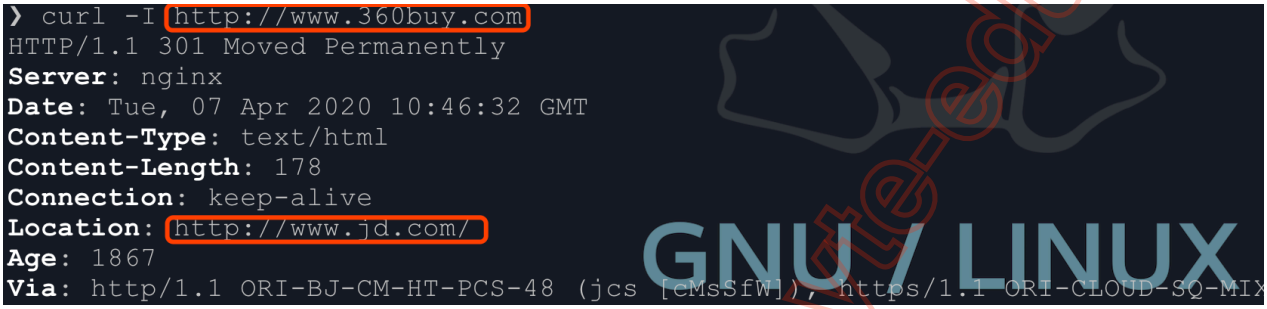
请求测试 **`http://www.b.com`**

```
❯ curl -I http://www.b.com/
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Fri, 03 Apr 2020 21:57:15 GMT
Content-Type: text/html
Content-Length: 162
Connection: keep-alive
Location: http://www.a.com/
```

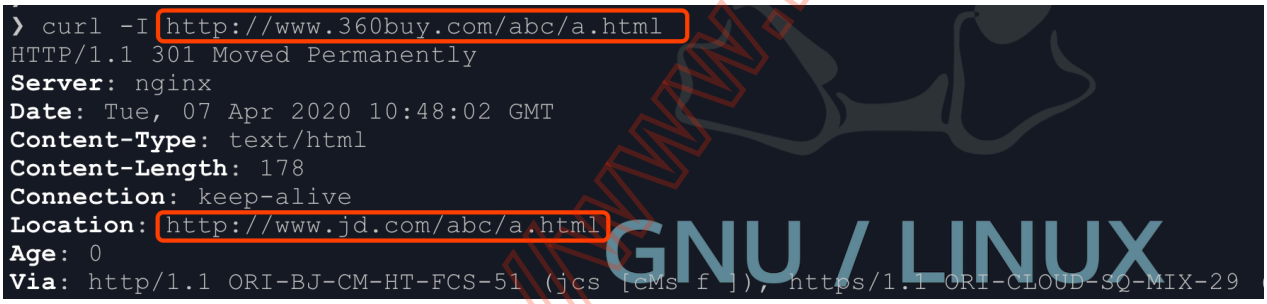
可以看到，实现了 `301` 永久重定向，下面我们测试下参数传递，请求 **`http://www.b.com/abc/a.html`**

```
❯ curl -I http://www.b.com/abc/a.html
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Fri, 03 Apr 2020 21:57:10 GMT
Content-Type: text/html
Content-Length: 162
Connection: keep-alive
Location: http://www.a.com/abc/a.html
```

可以看到参数也能够正常传递，我们看下京东是怎么做的，如下图：



同样也是使用 `301` 永久重定向。再看下参数传递的情况：



顺便说下，`301` 永久重定向，浏览器会缓存重定向后的地址，当用户下次请求时，会直接使用缓存的地址。但是 `302` 临时重定向则浏览器不会缓存地址，用户下次请求时，会继续请求原来的网站。永久重定向时网络爬虫会爬取新的域名，增加新域名权重。临时重定向则爬虫不会更新域名，依然使用老域名。所以，如果你的网站不想使用某个域名了，那么使用永久重定向，如果还打算使用，就使用临时重定向，比如我们上面的 `http` 跳转 `https` 的应用实例其实就是 `302` 临时重定向。

---

6. 根据浏览器语言跳转不同页面

在一些比较大型全球性的网站，比如 `https://developer.mozilla.org` 或者 `https://www.ibm.com` 等，都会有多国语言页面，而且是根据你浏览器语言自动实现的。如下：

```
❯ curl -I -H 'accept-language: zh-CN' https://www.ibm.com/
HTTP/2 303
server: GHost
content-length: 0
location: https://www.ibm.com/cn-zh/?ar=1
date: Tue, 07 Apr 2020 04:21:14 GMT
... ...
```

注意：此时 `location https://www.ibm.com/cn-zh/?ar=1`，当我们修改请求头部，如下：

```
❯ curl -I -H 'accept-language: en-US' https://www.ibm.com/
HTTP/2 303
server: GHost
content-length: 0
location: https://www.ibm.com/cn-en/?ar=1
date: Tue, 07 Apr 2020 04:22:26 GMT
... ...
```

此时 `location` 跳转到 `https://www.ibm.com/cn-zh/?ar=1`，像这种功能，我们就要借助 `if` 与 `set`、`rewrite` 等功能来实现。现在我们来实现这个功能：

我们希望当浏览器语言为 `zh|zh-CN` 时，访问 `https://www.a.com/` 跳转到 `https://www.a.com/cn-zh`，而当浏览器语言为 `en|en-US` 是，访问页面跳转到 `https://www.a.com/cn-en`，配置如下：

```
server {
    listen        80 default_server;
    server_name   www.a.com a.com;
    return        302  https://www.a.com$request_uri;
}
server {
    listen        443 ssl default_server;
    server_name   www.a.com a.com;
    ssl_certificate "/data/nginx/ssl/nginx.crt";
    ssl_certificate_key "/data/nginx/ssl/nginx.key";
    root          /data/nginx/a;
    charset       utf-8;
    location = / {
      if ( $http_accept_language ~* "zh|zh-CN") {
          set $language cn-zh;
      }
      if ( $http_accept_language ~* "en|en-US") {
          set $language cn-en;
      }
      rewrite ^/$  /$language ;
    }
    location / {
      index index.html;
    }
}
```

分别创建 `$document_root/{en-zh,cn-zh}` 目录，以及相关的 `index.html` 测试页面：

```
[root@web-nginx conf.d]# mkdir /data/nginx/a/{cn-zh,cn-en} -pv
mkdir: 已创建目录 "/data/nginx/a/cn-zh"
mkdir: 已创建目录 "/data/nginx/a/cn-en"
[root@web-nginx conf.d]# echo "中文页面" >> /data/nginx/a/cn-zh/index.html
[root@web-nginx conf.d]# echo "English Page" >> /data/nginx/a/cn-en/index.html
```

此时我们修改请求头部，来实现不同的页面

```
❯ curl -I -k -H 'accept-language: zh' https://www.a.com/
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Fri, 03 Apr 2020 15:49:46 GMT
Content-Type: text/html
Content-Length: 162
Location: https://www.a.com/cn-zh/
Connection: keep-alive
```

可以看到页面跳转到 `https://www.a.com/cn-zh/`，看下响应页面：

```
❯ curl -L -k -H 'accept-language: zh' https://www.a.com/
中文页面
```

现在，我们修改成英文语言，如下：

```
❯ curl -I -k -H 'accept-language: en' https://www.a.com/
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Fri, 03 Apr 2020 15:52:35 GMT
Content-Type: text/html
Content-Length: 162
Location: https://www.a.com/cn-en/
Connection: keep-alive

❯ curl -L -k -H 'accept-language: en' https://www.a.com/
English Page
```

只有访问 `http(s)://www.a.com/` 即 `uri = /` 时，才会做语言设置匹配。但是大家思考下，上面的配置有问题吗？假如我使用非 `zh|zh-CN|en|en-US` 语言访问时，会出现什么问题呢？

```
❯ curl -I -k -H 'accept-language: heihei' https://www.a.com/
HTTP/1.1 500 Internal Server Error
Server: nginx
Date: Fri, 03 Apr 2020 15:55:30 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 170
Connection: close
```

可以看到造成了内部循环重定向，服务端出现 `500` 报错。那么如何解决这个问题呢？

两种实现方案：

【只需要在 `rewrite` 规则后加上 `break` 标志位】

```
server {
    listen        80 default_server;
    server_name   www.a.com a.com;
    return        302  https://www.a.com$request_uri;
}
server {
    listen        443 ssl default_server;
    server_name   www.a.com a.com;
    ssl_certificate "/data/nginx/ssl/nginx.crt";
    ssl_certificate_key "/data/nginx/ssl/nginx.key";
    root          /data/nginx/a;
    charset       utf-8;
    location = / {
      if ( $http_accept_language ~* "zh|zh-CN") {
          set $language cn-zh;
      }
      if ( $http_accept_language ~* "en|en-US") {
          set $language cn-en;
      }
      rewrite ^/$  /$language break;
    }
    location / {
      index index.html;
    }
}
```

此时无论是否能够匹配到 `zh|zh-CN|en|en-US` 都不会有问题，大家可以试着再去测试下。还有一种方式，我们可以再加一个 `if` 判断语句，如下：

```
server {
    listen        80 default_server;
    server_name   www.a.com a.com;
    return        302  https://www.a.com$request_uri;
}
server {
    listen        443 ssl default_server;
    server_name   www.a.com a.com;
    ssl_certificate "/data/nginx/ssl/nginx.crt";
    ssl_certificate_key "/data/nginx/ssl/nginx.key";
    root          /data/nginx/a;
    charset       utf-8;
    location = / {
      if ( $http_accept_language ~* "zh|zh-CN") {
```

```
            set $language cn-zh;
        }
        if ( $http_accept_language ~* "en|en-US") {
            set $language cn-en;
        }
        if ( $http_accept_language !~* "zh|zh-CN|en|en-US") {
            return  https://www.a.com/index.html;
        }
        rewrite ^/$  /$language;
    }
    location / {
      index index.html;
    }
}
```

这种方法也可以解决，但是多了一次 `if` 判断，所以还是建议上面的方式。其实解决问题的方法还有很多，前提是大家能够搞清楚 `return` 、`rewrite` 以及相关的 `flag` 的各种功能，当然，最终选用哪种，还需要根据性能参数以及自己网站的设计规划【比如，我们上面设计的方式是，只有访问根时才做语言匹配，你也可以根据任何页面做语言划分】。

> **注意：上述设置只能适应于浏览器单语言，如果是同时支持 `zh|en` 的话，则只能匹配到 `zh` ，所以我们可以这样修改下：**
>
> ```
> location =/ {
> if ( $http_accept_language ~* "^(zh|zh-cn)"){
>     set $language cn-zh;
> }
> if ( $http_accept_language ~* "^(en|en-us)"){
>     set $language en-us;
> }
> rewrite ^/$ /$language break;
> }
> ```
>
> **知识点：浏览器的请求头 `Accept-Language` 优先级最高的语言会出现在该字段最左侧。**

---

7. 根据客户端不同，使用不同页面响应

在一些用户体验比较好的网站，一般都会适配 `PC` 端 或者 移动端，即当用户是用电脑访问页面 `www.a.com` 时，使用 `www.a.com` 进行响应，而用手机或者 `pad` 访问时，使用 `m.a.com` 进行响应。现在绝大多数都会使用这种方式去匹配，在早期还有可能会使用不同的 `uri` 进行响应。比如 `www.taobao.com` 、`www.jd.com` 等使用不同子域名进行响应，如下：

```
❯ curl -I https://www.taobao.com
HTTP/2 200
server: Tengine
content-type: text/html; charset=utf-8
date: Tue, 07 Apr 2020 07:28:30 GMT
vary: Accept-Encoding
vary: Ali-Detector-Type
cache-control: max-age=60, s-maxage=90
x-snapshot-age: 1
... ...
```

正常使用电脑端请求没有什么变化，此时我们修改下 `User-Agent` 看下：

```
❯ curl -I -H "User-Agent: iPhone" https://www.taobao.com
HTTP/2 302
server: Tengine
content-type: text/html
content-length: 258
location: http://m.taobao.com/?sprefer=sypc00
... ...
```

可以看到当使用移动端时，页面发生了 `302` 临时重定向，跳转到 `http://m.taobao.com/?sprefer=sypc00`，我们现在希望也实现类似的功能。

配置如下：

```
server {
    listen        80 default_server;
    server_name   www.a.com a.com;
    return        302  https://www.a.com$request_uri;
}
server {
    listen        443 ssl default_server;
    server_name   www.a.com a.com;
    ssl_certificate "/data/nginx/ssl/nginx.crt";
    ssl_certificate_key "/data/nginx/ssl/nginx.key";
    root          /data/nginx/a;
    charset       utf-8;
    if ( $http_user_agent ~* "android|iphone|ipad|pad"){
      return  302 http://m.a.com$request_uri;
    }
    location / {
      index index.html;
    }
}

server {
    listen         80;
```

```
    server_name    m.a.com;
    root           /data/nginx/a/m;
    index          index.html;

}
```

请求测试：`https://www.a.com/images/a.jpeg`

使用 `PC` 端请求：

```
❯ curl -k -I https://www.a.com/images/a.jpeg
HTTP/1.1 200 OK
Server: nginx
Date: Fri, 03 Apr 2020 19:25:42 GMT
Content-Type: image/jpeg
Content-Length: 24855
Last-Modified: Wed, 01 Apr 2020 16:19:22 GMT
Connection: keep-alive
ETag: "5e84bf0a-6117"
Accept-Ranges: bytes
```

可以看到正常 `200` 响应，并未产生跳转。我们修改请求头之后再去请求，如下：

```
❯ curl -k -I -H "User-Agent: iphone" https://www.a.com/images/a.jpeg
HTTP/1.1 302 Moved Temporarily
Server: nginx
Date: Fri, 03 Apr 2020 19:23:47 GMT
Content-Type: text/html
Content-Length: 138
Connection: keep-alive
Location: http://m.a.com/images/a.jpeg
```

可以看到 `location http://m.a.com/images/a.jpeg`，我们在前面介绍过，在 `$request_uri` 中会携带参数，同时跳转之后依然携带之前的 `uri + query_string` 才是比较合理的方式。

顺便说一下，通过变量 `$http_user_agent` 也可以实现网站的防爬虫。实现方式和我们上面的思路大致相同，此处不再赘述。

---

8. 使用正则表达式完成某些特性需求

我们不可能介绍所有的生产案例，更多的是以典型案例来讨论解决思路，上面的实例我们实现了根据客户端的某些特征值来实现不同的 `URL` 重写，但是我们尽可能的避免使用正则表达式，现在我们介绍下正则表达式的使用方法，虽然 `Nginx` 在处理正则上会耗费一些性能，但是可以实现一些比较"高难度"的操作。如下：

由于网站 A （域名 `api.a.com` ）服务器在迁移，暂时将数据都放在了 `www.a.com/api` 目录下，我们希望用户请求 `http://api.a.com/abc/12.html?a=3&b=4` 将地址重写为 `https://www.a.com/api/abc/12.html?a=3&b=4`，实现方式如下：

```
server {
    listen        80 default_server;
    server_name   www.a.com a.com;
    return        302  https://www.a.com$request_uri;
}
server {
    listen        443 ssl default_server;
    server_name   www.a.com a.com;
    ssl_certificate "/data/nginx/ssl/nginx.crt";
    ssl_certificate_key "/data/nginx/ssl/nginx.key";
    root          /data/nginx/a;
    charset       utf-8;
    if ( $http_user_agent ~* "android|iphone|ipad|pad"){
      return  302 http://m.a.com$request_uri;
    }
    location / {
      index index.html;
    }
}

server {
    listen         80;
    server_name    api.a.com;
    if ( $host ~* (.*)\.(.*)\.(.*) ){
      set $domain1 $1;
    }
    location / {
      return  302  https://www.a.com/$domain1$request_uri;
    }
}
```

我们需要清楚的知道，究竟需要哪些字符串，然后通过正则表达式来获取，比如上面我们需要二级域名，那么就通过 `$host ~* (.*)\.(.*)\.(.*)` 方式将域名进行分组，注意这个时候一定要把分组信息通过变量保存下来，以备下面使用。

请求测试：**`http://api.a.com/abc/12.html?a=3&b=4`**

```
❯ curl -I http://api.a.com/abc/12.html\?a\=3\&b\=4
HTTP/1.1 302 Moved Temporarily
Server: nginx
Date: Fri, 03 Apr 2020 20:14:58 GMT
Content-Type: text/html
Content-Length: 138
Connection: keep-alive
Location: https://www.a.com/api/abc/12.html?a=3&b=4
```

同学们自己也可以测试下其他页面，应该都是可以满足我们之前提的需求的。比如，当我们访问根域名时，也会自动跳转，如下：

```
❯ curl -I http://api.a.com
HTTP/1.1 302 Moved Temporarily
Server: nginx
Date: Fri, 03 Apr 2020 20:17:00 GMT
Content-Type: text/html
Content-Length: 138
Connection: keep-alive
Location: https://www.a.com/api/
```

---

9. 使用网站维护页面提高用户体验

很多时候我们选择上线新系统时，用户的访问请求可能会出现一些不太友好的页面，比如服务无响应、超时等等，一般情况下我们会选择将用户的请求全部重写到某个友好的维护页面，当然，前提是不要大幅度修改 `Nginx` 配置，比如我们想无论用户请求 `http://www.a.com` 的任意页面，都返回维护页面 `https://www.a.com/maintain.html`，配置如下：

```
server {
    listen        80 default_server;
    server_name   www.a.com a.com;
    return        302  https://www.a.com$request_uri;
}
server {
    listen        443 ssl default_server;
    server_name   www.a.com a.com;
    ssl_certificate "/data/nginx/ssl/nginx.crt";
    ssl_certificate_key "/data/nginx/ssl/nginx.key";
    root          /data/nginx/a;
    charset       utf-8;
    rewrite       ^(.*)$    /maintain.html break;
    if ( $http_user_agent ~* "android|iphone|ipad|pad"){
      return  302 http://m.a.com$request_uri;
    }
    location / {
      index index.html;
    }
}
```

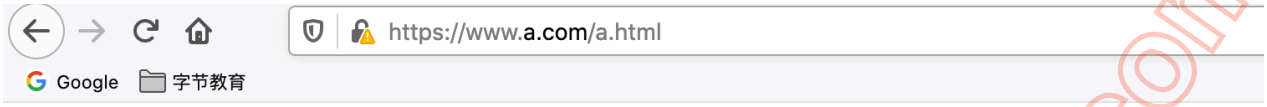
也就是我们只需要在 `server` 段配置一条 `rewrite ^(.*)$ /maintain.html break;` 规则即可。

> **另外，大家思考下，可以使用 `return 302 URL` 、 `301` 或者 `302` 重定向吗？什么情况下可以，什么情况下不能使用，会造成内部死循环？**
>
> **知识点：`return` 返回的 `code + URL` 如果是本机的，可以使用 `uri`，但是如果 `return` 到本机，则会发生内部重定向过多，导致出现死循环。如果 `return` 到外网就没有问题。**

创建维护页面：

```
[root@web-nginx conf.d]# echo "<h1>网站正在维护中，请稍后访问</h1>" >
/data/nginx/a/maintain.html
```

请求测试：



网站正在维护中，请稍后访问

此时无论我们请求任何页面都会重写 `URL`，使用 `/maintain.html` 页面响应。

```
❯ curl -k -I https://www.a.com/abc/test.html
HTTP/1.1 200 OK
Server: nginx
Date: Fri, 03 Apr 2020 21:30:31 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 49
Last-Modified: Fri, 03 Apr 2020 21:27:30 GMT
Connection: keep-alive
ETag: "5e87aa42-31"
Accept-Ranges: bytes

❯ curl -k  https://www.a.com/abc/test.html
<h1>网站正在维护中，请稍后访问</h1>
```

注意：此时不是重定向，而是地址重写。

---

10. 使用 `Nginx` 做四层代理

`Nginx` 从 `1.9.0` 开始，新增加了一个 `stream` 模块，用来实现四层协议（`TCP`）的转发、代理或者负载均衡等。在 `1.9.13` 版本中，开始支持 `UDP` 四层代理。生产上有些小型的四层调度可能会使用 `Nginx`，我们简单介绍下这个模块的用法。（这个模块中的很多参数和 `http` 模块非常类似，甚至相同，大家也可以类比去理解）

listen：设置四层代理监听地址，基本语法：

```
Syntax: listen address:port [ssl] [udp] [proxy_protocol] [backlog=number]
[rcvbuf=size]
        [sndbuf=size] [bind] [ipv6only=on|off] [reuseport]
        [so_keepalive=on|off|[keepidle]:[keepintvl]:[keepcnt]];
Default:  —
Context:  server
```

我们在前面 `http` 核心模块中 `listen` 指令介绍的一些参数，在这里含义也是一样的，比如 `backlog`、`reuseport`、`so_keeepalive` 等。

---

proxy_pass：设置代理服务器的地址。该地址可以指定为域名或 `IP` 地址加端口，或 `UNIX` 套接字路径。

基本语法：

```
Syntax: proxy_pass address;
Default:  —
Context:  server
```

示例：

```
proxy_pass localhost:12345;
```

```
proxy_pass unix:/tmp/stream.socket;
```

也可以支持变量：

```
proxy_pass $upstream;
```

---

proxy_timeout：在客户端与代理服务器连接上的两次连续读写操作之间设置超时。如果在此时间内没有发送数据，则连接被关闭。默认为 `10m`；基本语法：

```
Syntax: proxy_timeout timeout;
Default:  proxy_timeout 10m;
Context:  stream, server
```

> **Sets the `timeout` between two successive read or write operations on client or proxied server connections. If no data is transmitted within this time, the connection is closed.**

---

proxy_connect_timeout：设置 `Nginx` 与被代理的服务器尝试建立连接的超时时长；默认为 `60s`；

基本语法：

```
Syntax: proxy_connect_timeout time;
Default:  proxy_connect_timeout 60s;
Context:  stream, server
```

当然，在 `stream` 中，也可以使用相应的 `upstream` 模块，模块名称 `ngx_stream_upstream_module`，设计到的语法我们在前面的 `ngx_http_upstream_module` 中都介绍过，只不过支持的参数没有 `http_stream` 模块中多，示例配置：

```
upstream backend {
    hash $remote_addr consistent;

    server backend1.example.com:12345  weight=5;
    server backend2.example.com:12345  max_fails=3 fail_timeout=30s;
    server unix:/tmp/backend3;

    server backup1.example.com:12345   backup;
    server backup2.example.com:12345   backup;
}

server {
    listen 12346;
    proxy_pass backend;
}
```

注意：`ngx_stream_upstream_module` 中不支持 `ip_hash` 调度算法。

生产上一般还是比较少使用 `Nginx` 做四层代理的，可能有些自己开发的应用确实需要轻量级四层代理，我们此处就以代理后端 `SSHD` 服务为案例了，姑且不去考虑案例的合理性，只是做技术性演示。

简单示例：

后端两个节点分别为 `192.168.48.101` 和 `192.168.48.102`，其中 `sshd` 都监听在 `22` 端口，我们使用 `192.168.48.100` 的 `22022` 端口进行代理转发，示例如下：

在 `nginx.conf` 配置文件中增加如下配置信息：

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

创建 `/etc/nginx/stream_conf.d/stream.conf` 配置文件，如下：

```
upstream ssh_back {
    server     192.168.48.101:22;
    server     192.168.48.102:22;
}

server {
    listen     22022;
    proxy_pass  ssh_back;
}
```

这是最基本的四层代理配置，默认使用轮询算法。我们也可以修改基于客户端地址的一致性 `hash` 算法，如下：

```
upstream ssh_back {
    hash        $remote_addr  consistent;
    server      192.168.48.101:22;
    server      192.168.48.102:22 max_fails=3 fail_timeout=30s;
}

server {
    listen      22022;
    proxy_pass  ssh_back;
}
```

这样就会根据客户端的 `IP` 地址进行一致性 `hash` 运算。只有当后端节点出现故障时，才会将请求调度转移到其他节点。