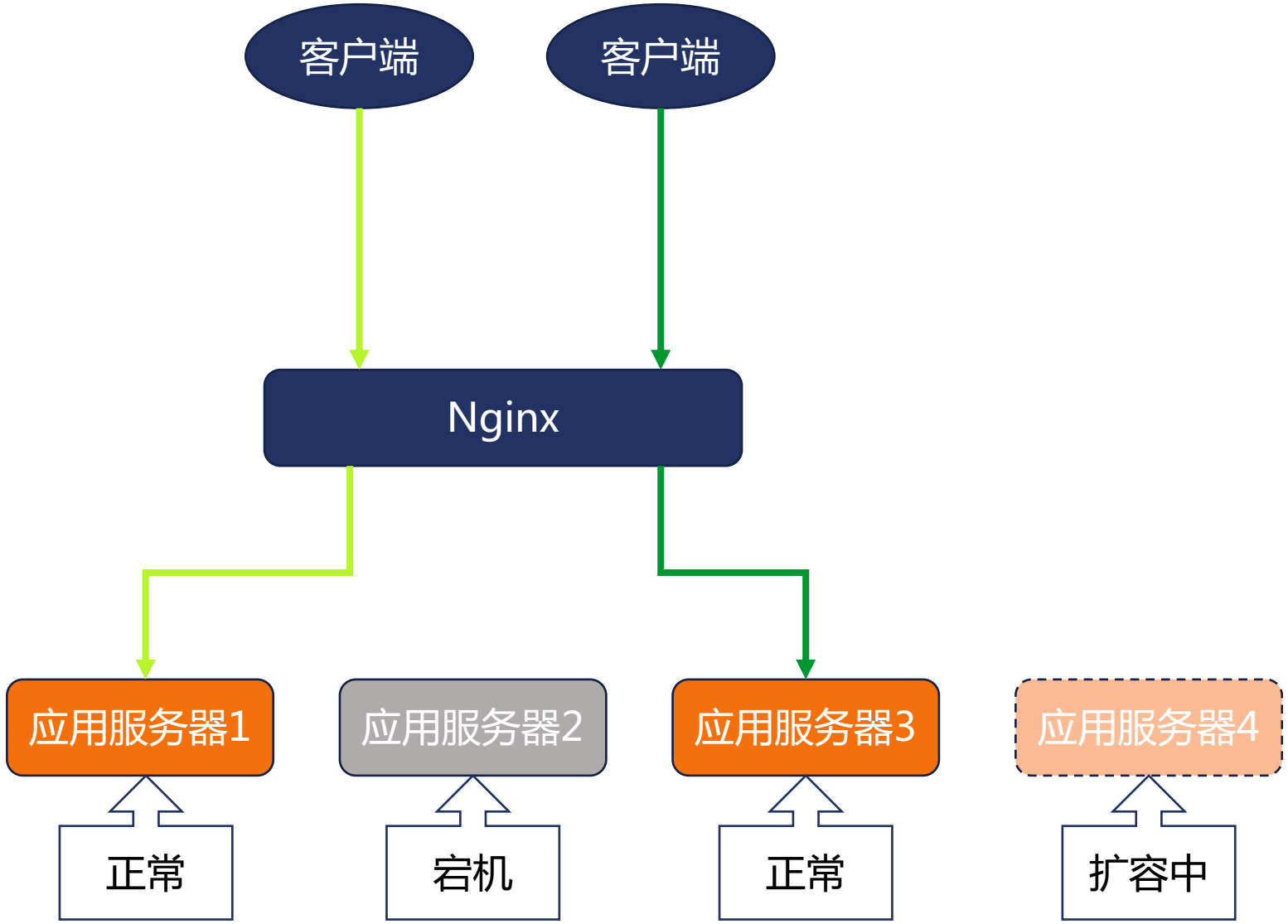
第四部分

反向代理与负载均衡

扫码试看/订阅

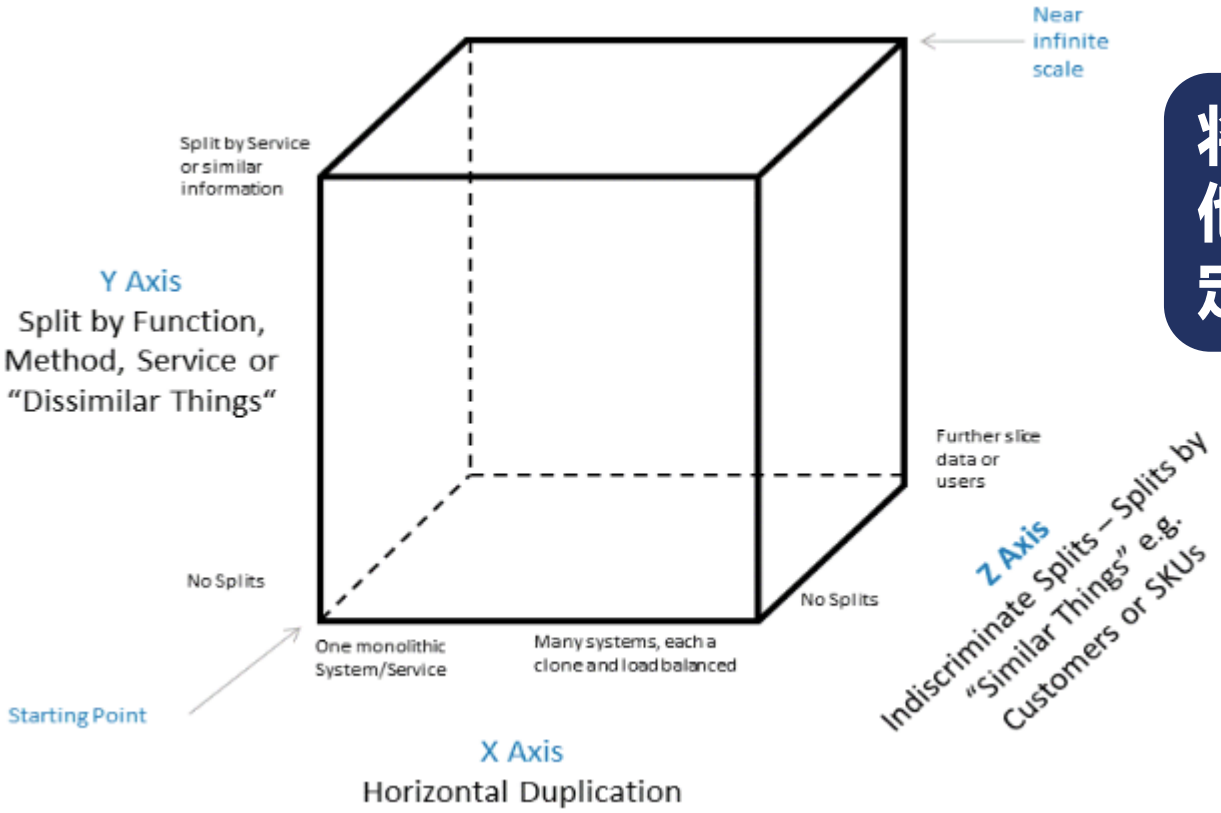
《Nginx 核心知识100讲》

负载均衡



Nginx 在 AKF 扩展立方体上的应用

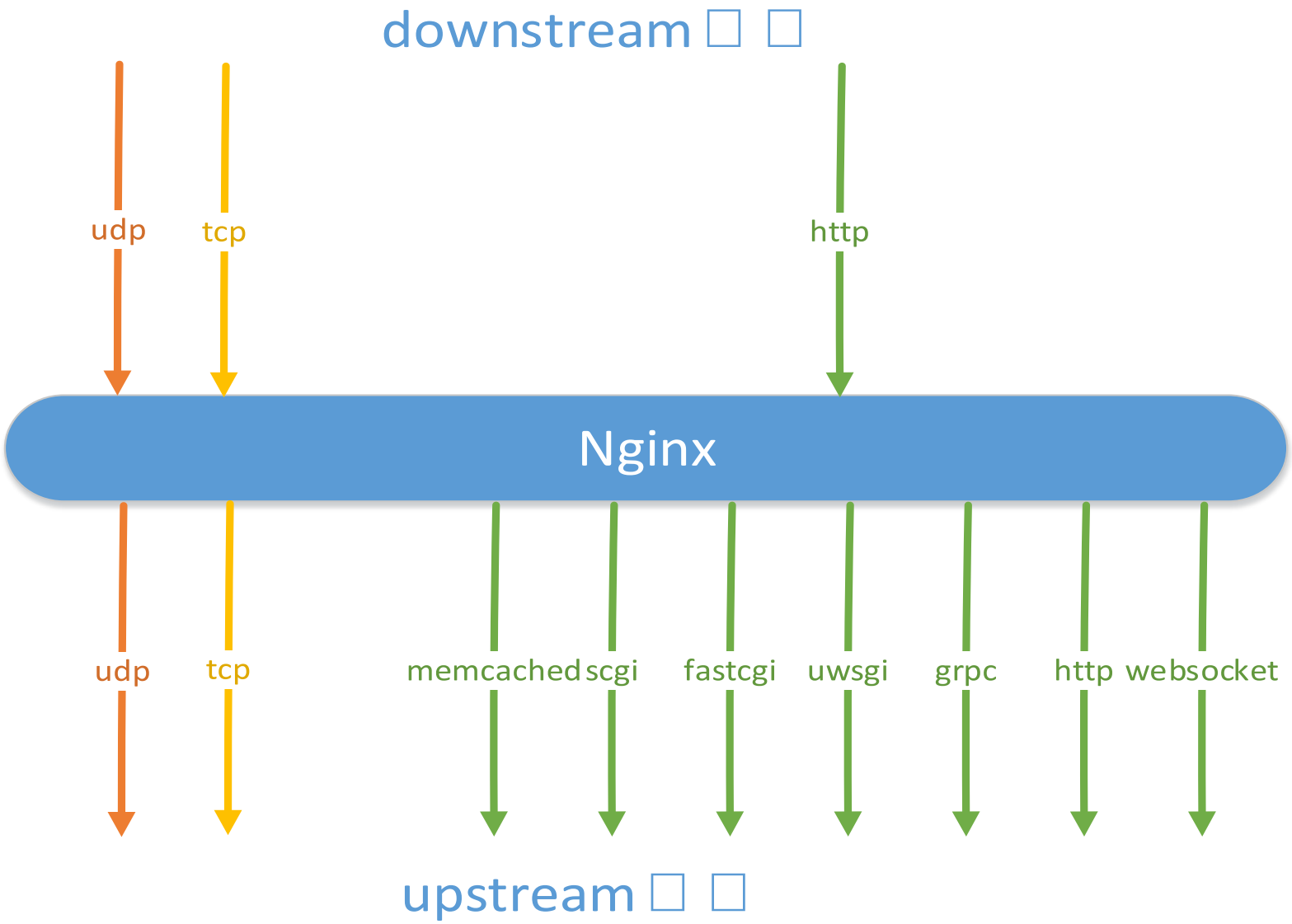
基于 URL 对功能进行分发



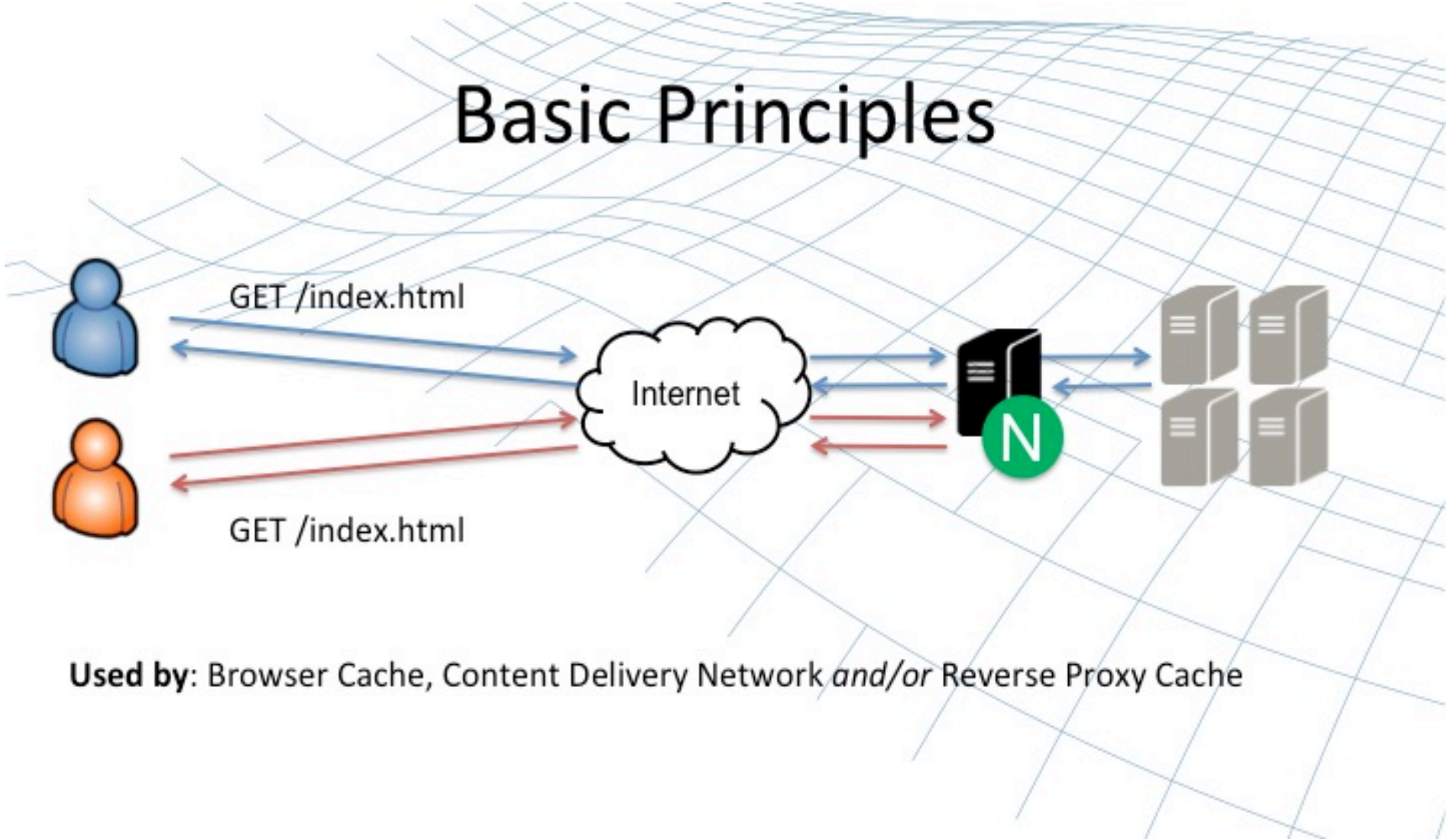
将用户 IP 地址或者其他信息映射到某个特定的服务或者集群

基于Round-Robin 或者least-connected 算法分发请求

支持多种协议的反向代理



反向代理与缓存


Basic Principles
GET /index.html
Internet
N
GET /index.html
Used by: Browser Cache, Content Delivery Network and/or Reverse Proxy Cache

指定上游服务地址的 upstream 与 server 指令

Syntax: **upstream** *name* { ... }

Default: —

Context: http

Syntax: **server** *address* [*parameters*];

Default: —

Context: upstream

指定上游服务地址的 upstream 与 server 指令

功能

指定一组上游服务器地址，其中，地址可以是域名、IP地址或者unix socket地址。可以在域名或者IP地址后加端口，如果不加端口，那么默认使用80端口。

通用参数

- **backup**：指定当前server为备份服务，仅当非备份server不可用时，请求才会转发到该server
- **down**：标识某台服务已经下线，不在服务

加权 Round-Robin 负载均衡算法

功能

在加权轮询的方式访问 server 指令指定的上游服务。

集成在 Nginx 的 upstream 框架中

指令

weight

- 服务访问的权重，默认是1

max_conns

- server的最大并发连接数，仅作用于单worker进程。默认是0，表示没有限制。

max_fails

- 在fail_timeout时间段内，最大的失败次数。当达到最大失败时，会在fail_timeout秒内这台server不允许再次被选择。

fail_timeout

- 单位为秒，默认值为10秒。具有2个功能：
- 指定一段时间内，最大的失败次数max_fails。
- 到达max_fails后，该server不能访问的时间。

对上游服务使用 keepalive 长连接

功能

- 通过复用连接，降低nginx与上游服务器建立、关闭连接的消耗，提升吞吐量的同时降低时延

模块

- ngx_http_upstream_keepalive_module，默认编译进nginx，通过--without-http_upstream_keepalive_module移除

对上游连接的http头部设定

```
proxy_http_version 1.1;
proxy_set_header Connection "";
```

upstream_keepalive 的指令

Syntax: **keepalive** *connections*;

Default: —

Context: upstream

1.15.3非稳定版本新增指令：

Syntax: **keepalive_requests** *number*;

Default: keepalive_requests 100;

Context: upstream

Syntax: **keepalive_timeout** *timeout*;

Default: keepalive_timeout 60s;

Context: upstream

指定上游服务域名解析的 resolver 指令

Syntax: **resolver** *address* ... [valid=*time*] [ipv6=on|off];
Default: —
Context: http, server, location

Syntax: **resolver_timeout** *time*;
Default: resolver_timeout 30s;
Context: http, server, location

基于客户端 IP 地址的 Hash 算法实现负载均衡：upstream_ip_hash模块

功能

以客户端的IP地址作为hash算法的关键字，映射到特定的上游服务器中。

- 对IPV4地址使用前3个字节作为关键字，对IPV6则使用完整地址
- 可以使用round-robin算法的参数。
- 可以基于realip模块修改用于执行算法的IP地址

模块

ngx_http_upstream_ip_hash_module，通过--without-http_upstream_ip_hash_module禁用模块

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

基于任意关键字实现 Hash 算法的负载均衡：upstream_hash模块

极客时间

功能

通过指定关键字作为hash key，基于hash算法映射到特定的上游服务器中。

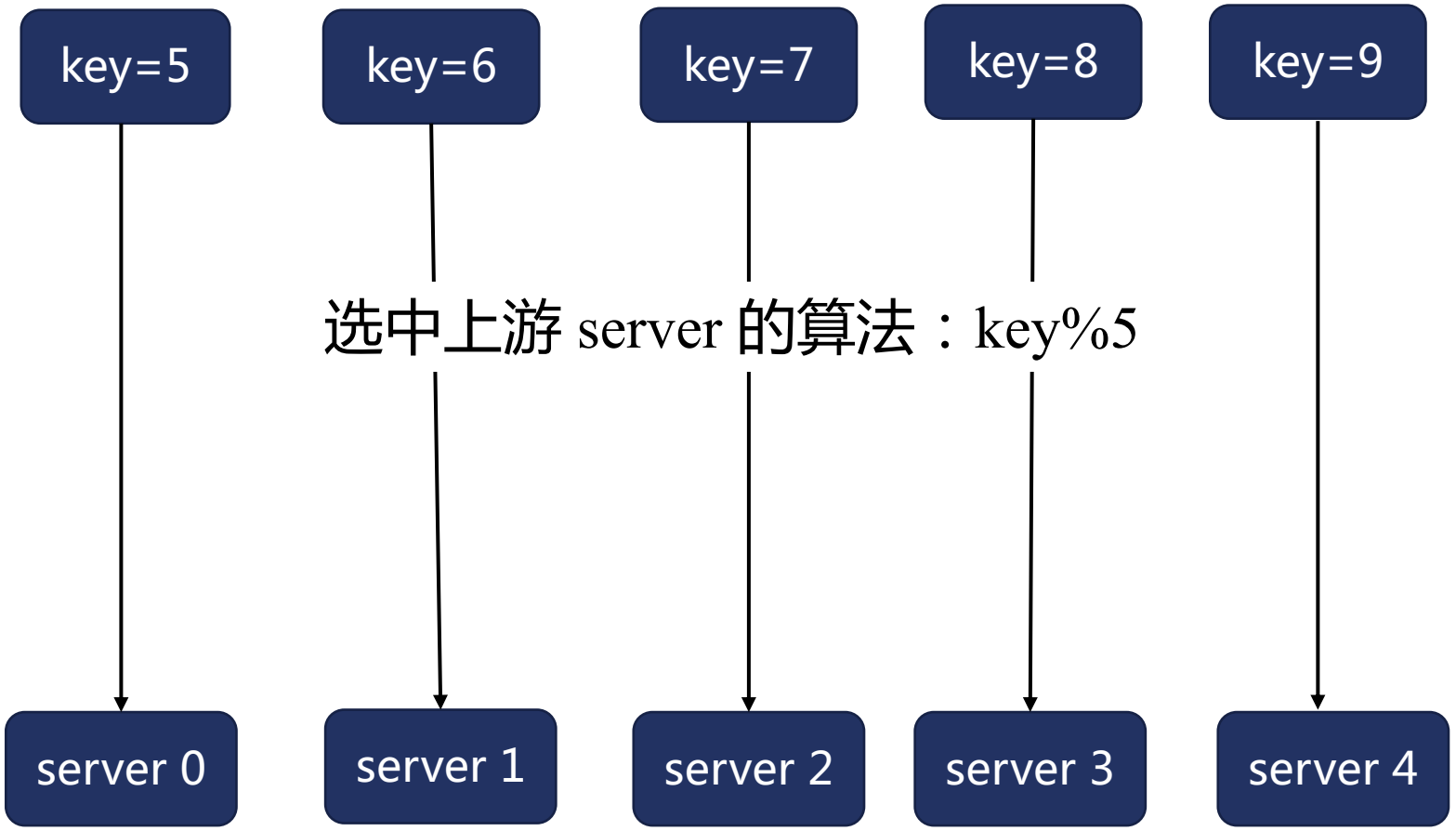
- 关键字可以含有变量、字符串。
- 可以使用round-robin算法的参数。

模块

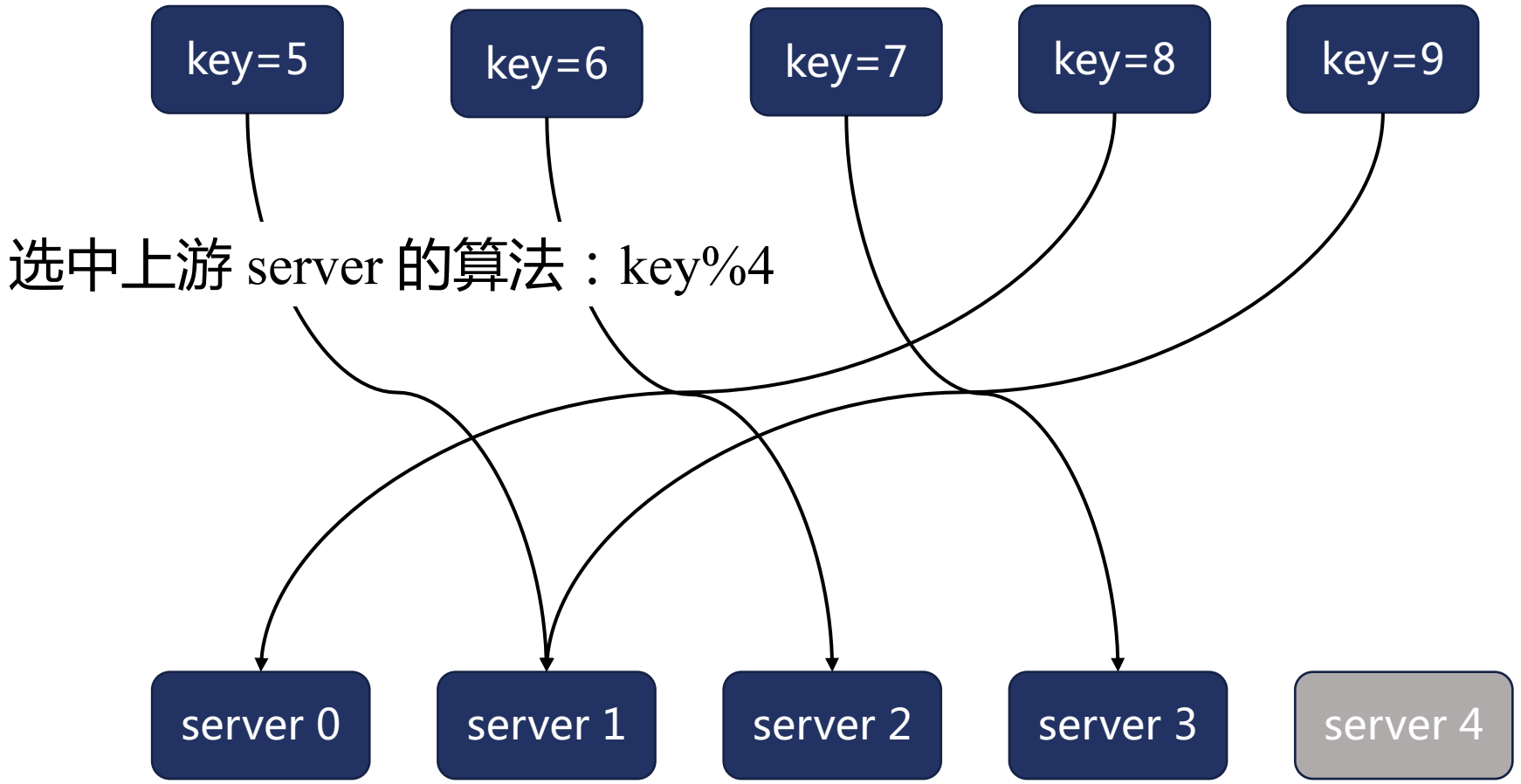
ngx_http_upstream_hash_module，通过--without-http_upstream_ip_hash_module禁用模块

Syntax: **hash key [consistent];**
Default: —
Context: upstream

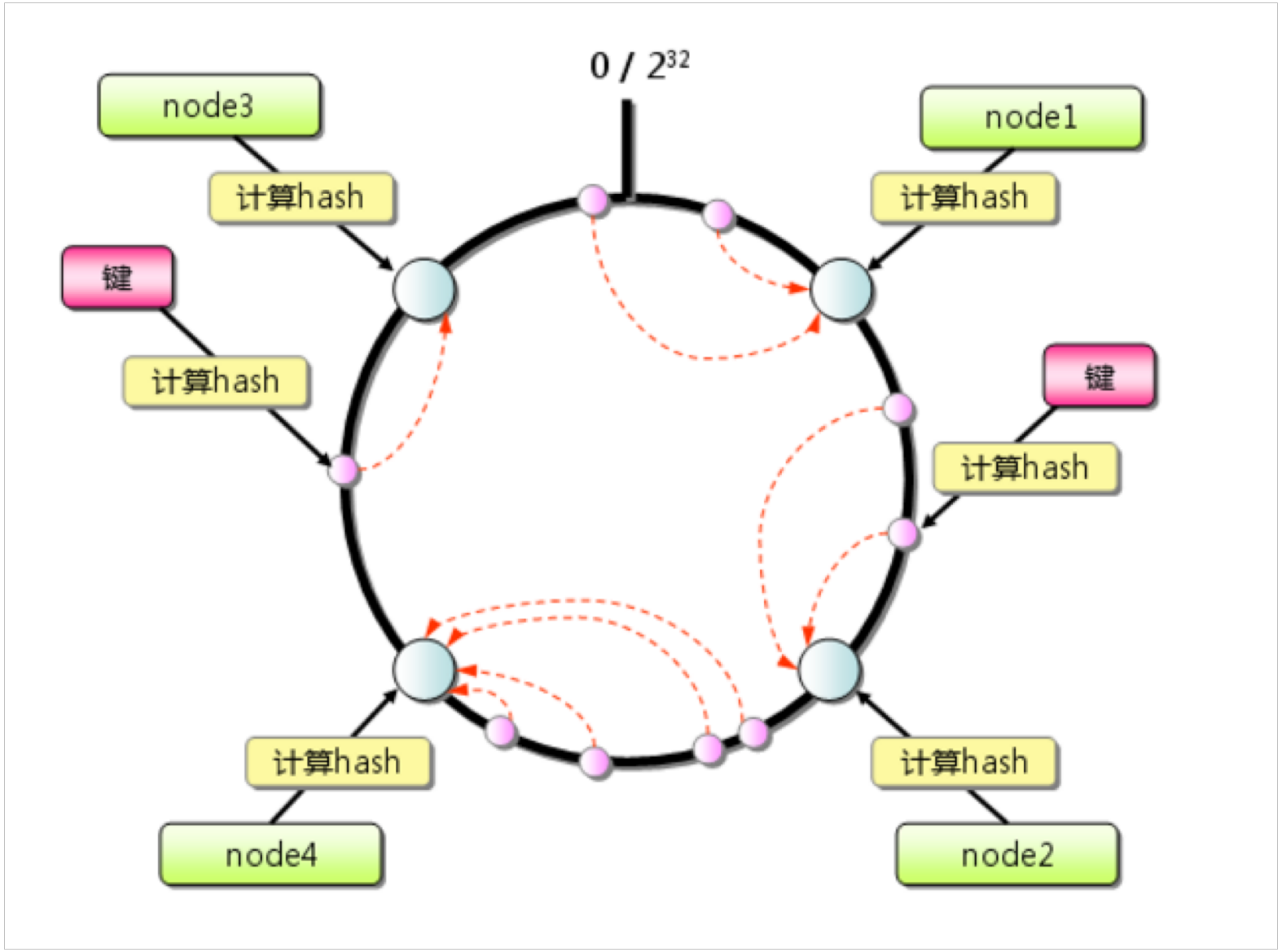
Hash 算法的问题


key=5
key=6
key=7
key=8
key=9
选中上游 server 的算法：key%5
server 0
server 1
server 2
server 3
server 4

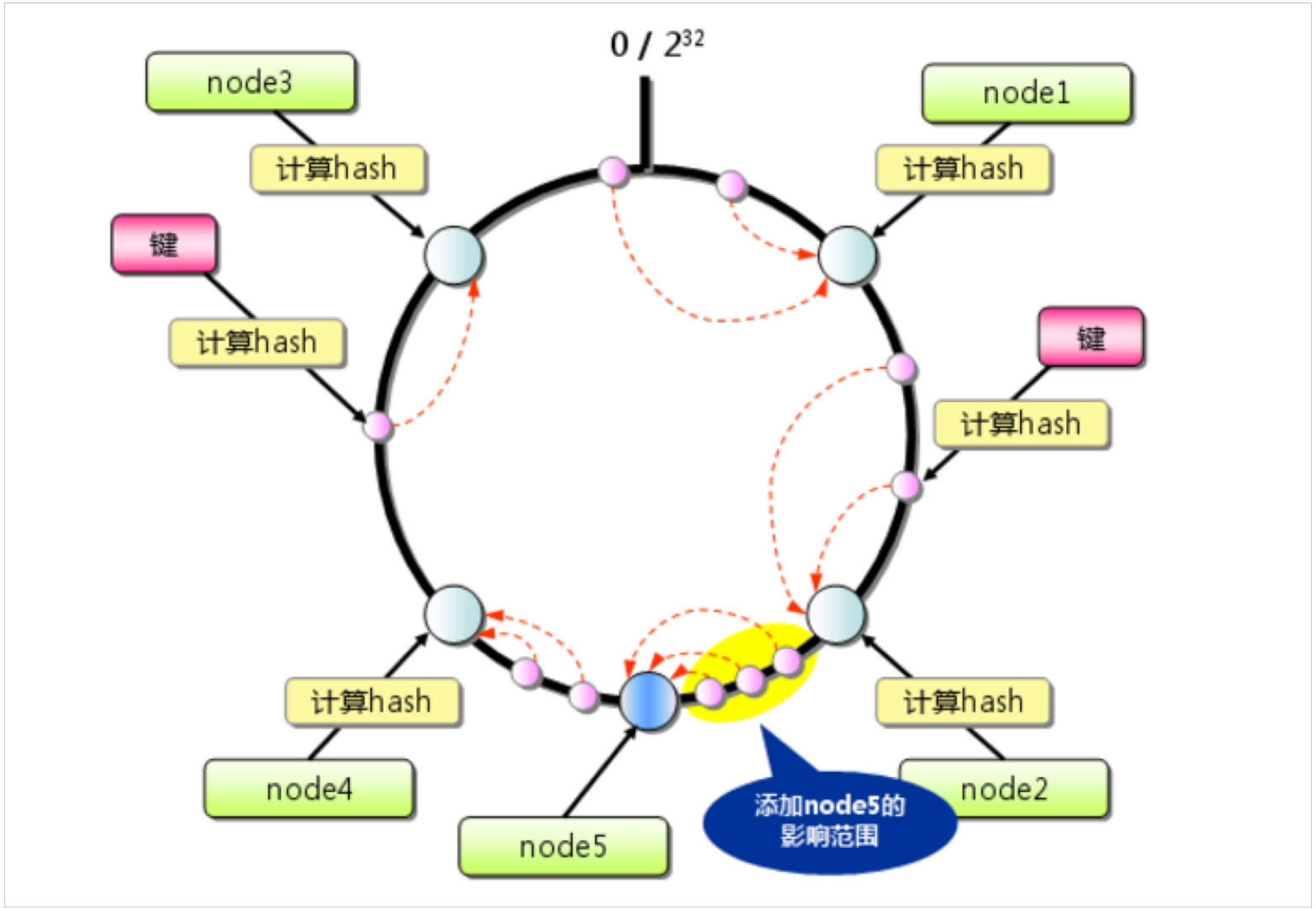
宕机或者扩容时，hash算法引发大量路由变更，可能导致缓存大范围失效



一致性 Hash 算法：扩容前



一致性 Hash 算法：扩容后



使用一致性 Hash 算法：upstream_hash 模块

Syntax: **hash** *key* [consistent];

Default: —

Context: upstream

优先选择连接最少的上游服务器：upstream_least_conn模块

极客时间

功能

从所有上游服务器中，找出当前并发连接数最少的一个，将请求转发到它。

- 如果出现多个最少连接服务器的连接数都是一样的，使用round-robin算法。

模块

ngx_http_upstream_ least_conn _module，通过--without-http_upstream_ip_hash_module禁用模块

Syntax: **least_conn;**
Default: —
Context: upstream

使用共享内存使负载均衡策略对所有worker进程生效：upstream_zone模块

功能

分配出共享内存，将其他upstream模块定义的负载均衡策略数据、运行时每个上游服务的状态数据存放在共享内存上，以对所有nginx worker进程生效

模块

ngx_http_upstream_ zone _module， 通过--without-http_upstream_ip_hash_module禁用模块

Syntax: **zone name [size];**
Default: —
Context: upstream

upstream 模块间的顺序：功能的正常运行

```
ngx_module_t *ngx_modules[] = {
    …… ……
    &ngx_http_upstream_hash_module,
    &ngx_http_upstream_ip_hash_module,
    &ngx_http_upstream_least_conn_module,
    &ngx_http_upstream_random_module,
    &ngx_http_upstream_keepalive_module,
    &ngx_http_upstream_zone_module,
    …… ……
};
```

upstream 模块提供的变量（不含 cache）

upstream_addr
上游服务器的IP地址，格式为可读的字符串，例如127.0.0.1:8012
upstream_connect_time
与上游服务建立连接消耗的时间，单位为秒，精确到毫秒
upstream_header_time
接收上游服务发回响应中http头部所消耗的时间，单位为秒，精确到毫秒
upstream_response_time
接收完整的上游服务响应所消耗的时间，单位为秒，精确到毫秒
upstream_http_名称
从上游服务返回的响应头部的值

upstream 模块提供的变量（不含 cache）

upstream_bytes_received

从上游服务接收到的响应长度，单位为字节

upstream_response_length

从上游服务返回的响应包体长度，单位为字节

upstream_status

上游服务返回的HTTP响应中的状态码。如果未连接上，该变量值为502

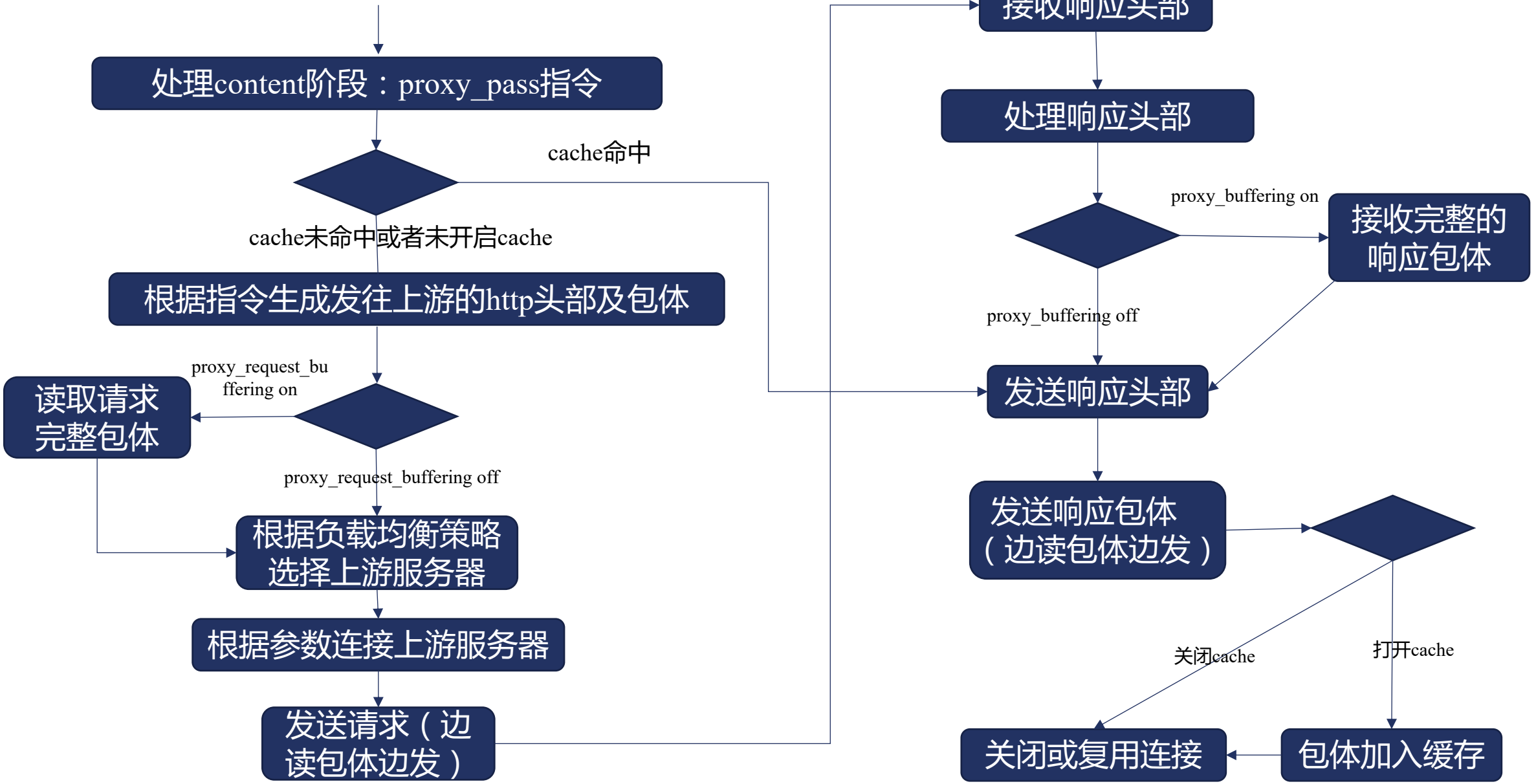
upstream_cookie_名称

从上游服务发回的响应头Set-Cookie中取出的cookie值

upstream_trailer_名称

从上游服务的响应尾部取到的值

HTTP 反向代理流程


处理content阶段：proxy_pass指令
cache命中
cache未命中或者未开启cache
根据指令生成发往上游的http头部及包体
proxy_request_buffering on
读取请求完整包体
proxy_request_buffering off
根据负载均衡策略选择上游服务器
根据参数连接上游服务器
发送请求（边读包体边发）
接收响应头部
处理响应头部
proxy_buffering on
接收完整的响应包体
proxy_buffering off
发送响应头部
发送响应包体（边读包体边发）
关闭cache
打开cache
关闭或复用连接
包体加入缓存

对 HTTP 协议的反向代理：proxy 模块

功能

对上游服务使用http/https协议进行反向代理

指令

ngx_http_proxy_module，

默认编译进nginx，

通过--without-http_proxy_module禁用

开启指令：proxy_pass

Syntax: **proxy_pass** *URL*;

Default: —

Context: location, if in location, limit_except

对 HTTP 协议的反向代理：proxy 模块

URL参数规则

- **URL必须以http://或者https://开头，接下来是域名、IP、unix socket地址或者upstream的名字，前两者可以在域名或者IP后加端口。最后是可选的URI。**
- **当URL参数中携带URI与否，会导致发向上游请求的URL不同：**
  - 不携带URI，则将客户端请求中的URL直接转发给上游
    - location后使用正则表达式、@名字时，应采用这种方式
  - 携带URI，则对用户请求中的URL作如下操作：
    - 将location参数中匹配上的一段替换为该URI
- **该URL参数中可以携带变量**
- **更复杂的URL替换，可以在location内的配置添加rewrite break语句**

proxy 模块：生成发往上游的请求行

Syntax: **proxy_method** *method*;
Default: —
Context: http, server, location

Syntax: **proxy_http_version** 1.0 | 1.1;
Default: proxy_http_version 1.0;
Context: http, server, location

proxy 模块：生成发往上游的请求头部

Syntax: **proxy_set_header** *field value*;

Default: proxy_set_header Host $proxy_host;
proxy_set_header Connection close;

Context: http, server, location

注意：若value的值为空字符串，则整个header都不会向上游发送

Syntax: **proxy_pass_request_headers** on | off;

Default: proxy_pass_request_headers on;

Context: http, server, location

proxy 模块：生成发往上游的包体

Syntax: **proxy_pass_request_body** on | off;
Default: proxy_pass_request_body on;
Context: http, server, location

Syntax: **proxy_set_body** *value*;
Default: —
Context: http, server, location

接收客户端请求的包体：收完再转发还是边收边转发？

Syntax: **proxy_request_buffering** on | off;
Default: proxy_request_buffering on;
Context: http, server, location

on

- 客户端网速较慢
- 上游服务并发处理能力低
- 适应高吞吐量场景

off

- 更及时的响应
- 降低nginx读写磁盘的消耗
- 一旦开始发送内容，proxy_next_upstream功能失败

客户端包体的接收

Syntax: **client_body_buffer_size** *size*;
Default: client_body_buffer_size 8k|16k;
Context: http, server, location

Syntax: **client_body_in_single_buffer** on | off;
Default: client_body_in_single_buffer off;
Context: http, server, location

存在包体时，接收包体所分配的内存

- **若接收头部时已经接收完全部包体，则不分配**
- **若剩余待接收包体的长度小于client_body_buffer_size，则仅分配所需大小**
- **分配client_body_buffer_size大小内存接收包体**
  - 关闭包体缓存时，该内存上内容及时发送给上游
  - 打开包体缓存
    - 该段大小内存用完时，写入临时文件，释放内存

最大包体长度限制

Syntax: **client_max_body_size** *size*;

Default: client_max_body_size 1m;

Context: http, server, location

仅对请求头部中含有Content-Length有效超出最大长度后，返回413错误

临时文件路径格式

Syntax: **client_body_temp_path** *path* [*level1* [*level2* [*level3*]]];
Default: client_body_temp_path client_body_temp;
Context: http, server, location

Syntax: **client_body_in_file_only** on | clean | off;
Default: client_body_in_file_only off;
Context: http, server, location

读取包体时的超时

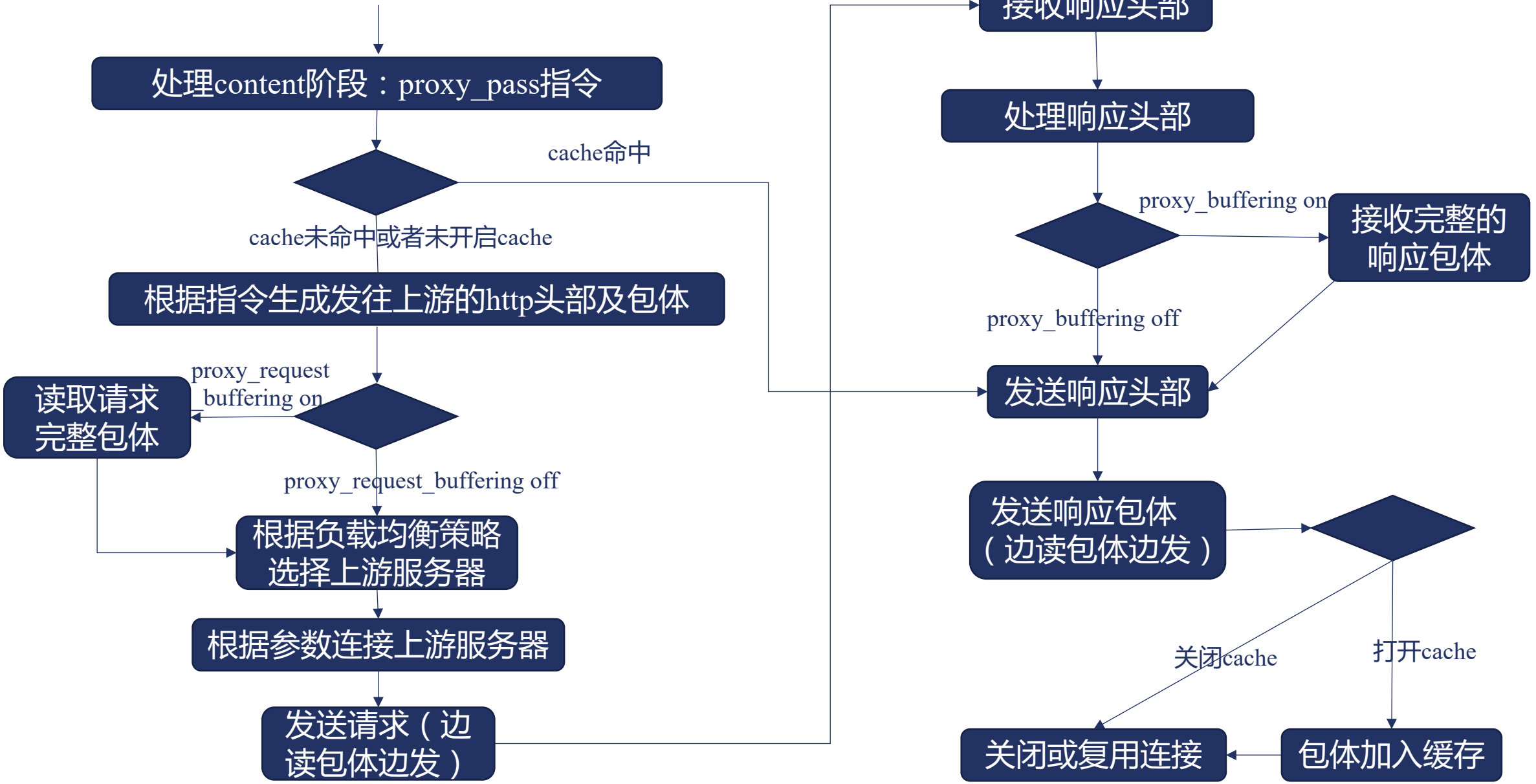
Syntax: **client_body_timeout** *time*;

Default: client_body_timeout 60s;

Context: http, server, location

读取包体时超时，则返回408错误

HTTP 反向代理流程


处理content阶段：proxy_pass指令
cache命中
cache未命中或者未开启cache
根据指令生成发往上游的http头部及包体
proxy_request
_buffering on
读取请求
完整包体
proxy_request_buffering off
根据负载均衡策略
选择上游服务器
根据参数连接上游服务器
发送请求（边
读包体边发）
接收响应头部
处理响应头部
proxy_buffering on
接收完整的
响应包体
proxy_buffering off
发送响应头部
发送响应包体
（边读包体边发）
关闭cache
打开cache
关闭或复用连接
包体加入缓存

向上游服务建立连接

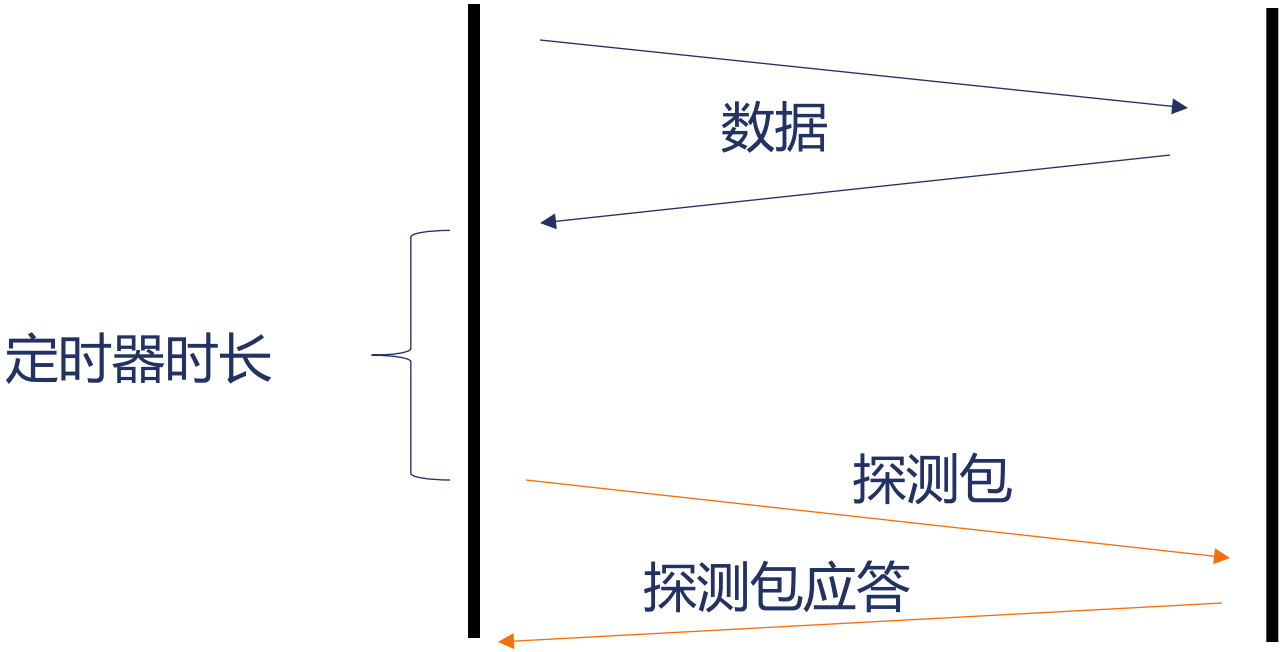
Syntax:	**proxy_connect_timeout time;**
Default:	proxy_connect_timeout 60s;
Context:	http, server, location

超时后，会向客户端生成http响应，响应码为502

Syntax:	**proxy_next_upstream** http_502 | ..;
Default:	proxy_next_upstream error timeout;
Context:	http, server, location

上游连接启用TCP keepalive

Syntax:	**proxy_socket_keepalive** on | off;
Default:	proxy_socket_keepalive off;
Context:	http, server, location



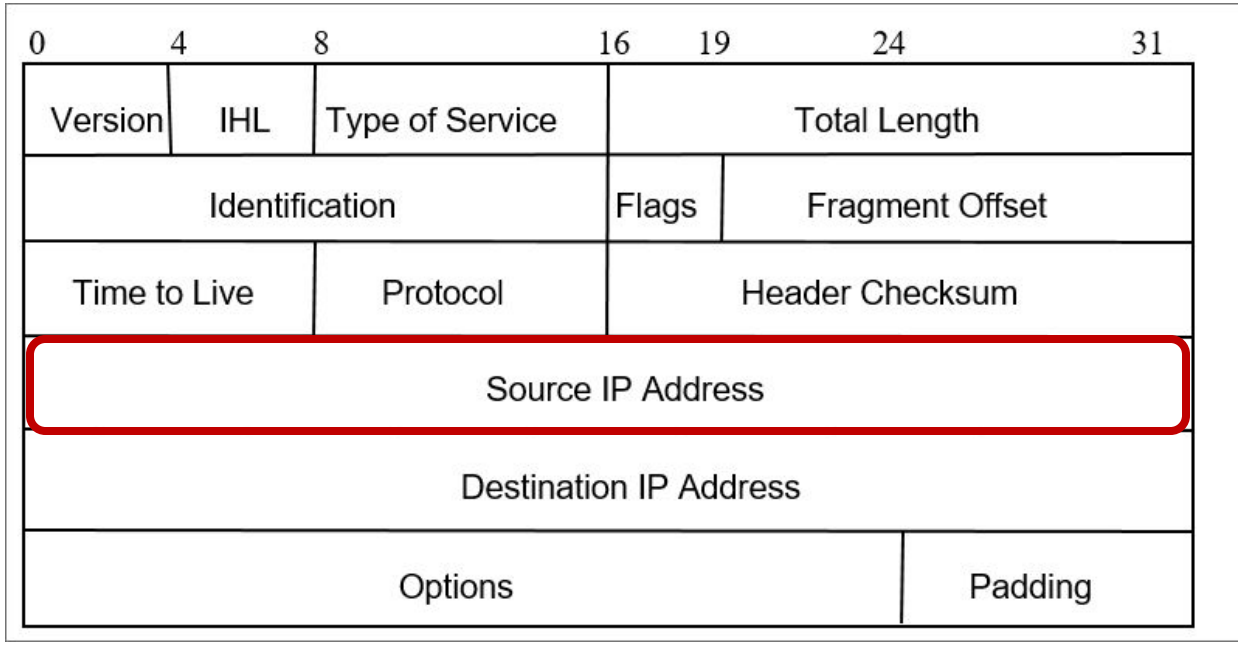
上游连接启用HTTP keepalive

Syntax:	**keepalive** connections;
Default:	—
Context:	upstream

Syntax:	**keepalive_requests** *number*;
Default:	keepalive_requests 100;
Context:	upstream

修改TCP连接中的local address

Syntax: **proxy_bind** address [transparent] | off;
Default: —
Context: http, server, location



- 可以使用变量：
  - proxy_bind $remote_addr;
- 可以使用不属于所在机器的IP地址：
  - proxy_bind $remote_addr transparent;

当客户端关闭连接时

Syntax:	**proxy_ignore_client_abort** on | off;
Default:	proxy_ignore_client_abort off;
Context:	http, server, location

向上游发送HTTP请求

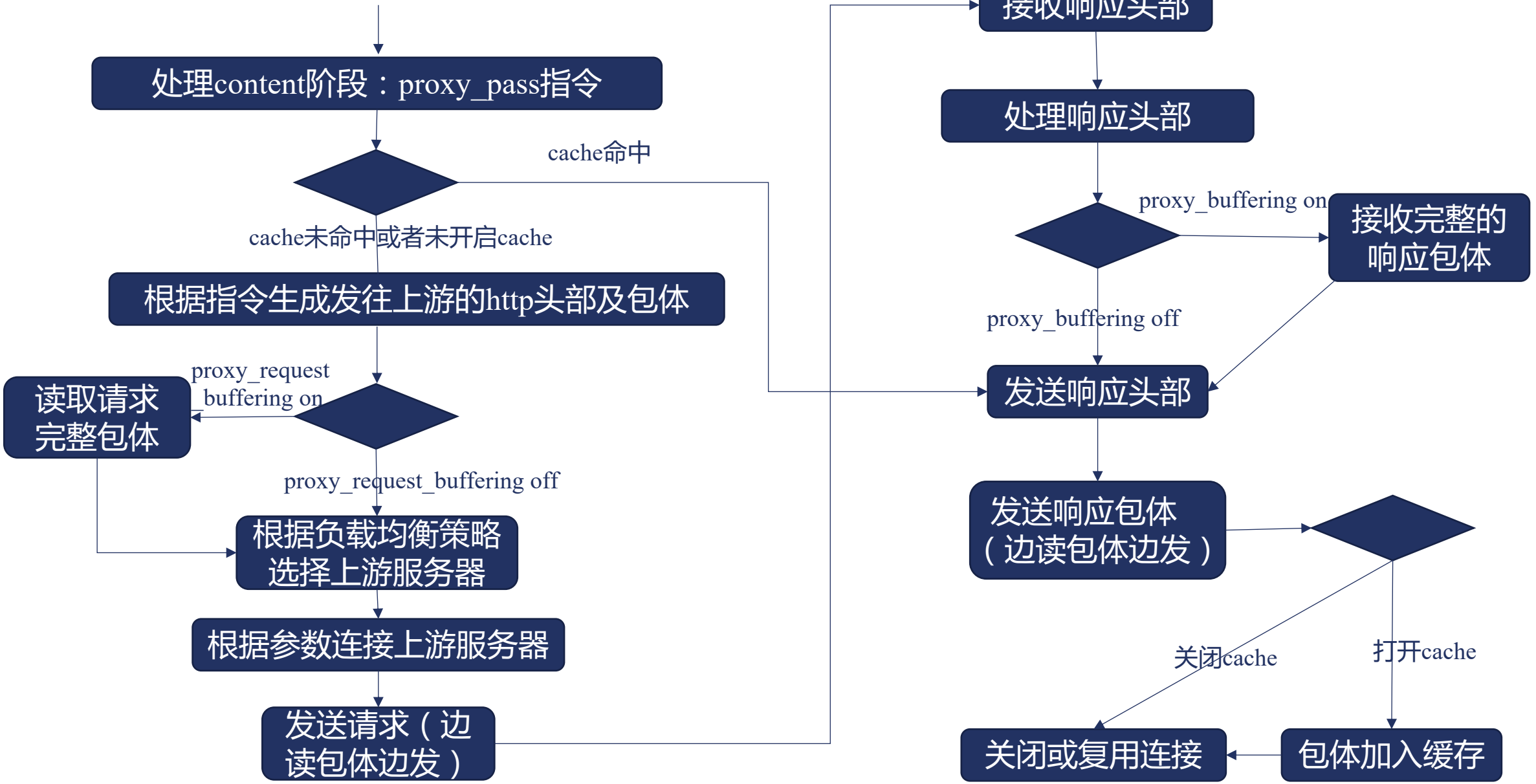
Syntax:	**proxy_send_timeout** time;
Default:	proxy_send_timeout 60s;
Context:	http, server, location

接收上游的HTTP响应头部

Syntax:	**proxy_buffer_size** *size*;
Default:	proxy_buffer_size 4k|8k;
Context:	http, server, location

error.log：upstream sent too big header

HTTP 反向代理流程


处理content阶段：proxy_pass指令
cache命中
cache未命中或者未开启cache
根据指令生成发往上游的http头部及包体
proxy_request
_buffering on
读取请求
完整包体
proxy_request_buffering off
根据负载均衡策略
选择上游服务器
根据参数连接上游服务器
发送请求（边
读包体边发）
接收响应头部
处理响应头部
proxy_buffering on
接收完整的
响应包体
proxy_buffering off
发送响应头部
发送响应包体
（边读包体边发）
关闭cache
打开cache
关闭或复用连接
包体加入缓存

接收上游的HTTP包体

Syntax:	**proxy_buffers** number size;
Default:	proxy_buffers 8 4k|8k;
Context:	http, server, location

接收上游的HTTP包体

Syntax:	**proxy_buffering** on | off;
Default:	proxy_buffering on;
Context:	http, server, location

X-Accel-Buffering头部
yes
no

Syntax:	**proxy_max_temp_file_size** *size*;
Default:	proxy_max_temp_file_size 1024m;
Context:	http, server, location

Syntax:	**proxy_temp_file_write_size** *size*;
Default:	proxy_temp_file_write_size 8k|16k;
Context:	http, server, location

Syntax:	**proxy_temp_path** path [level1 [level2 [level3]]];
Default:	proxy_temp_path proxy_temp;
Context:	http, server, location

及时转发包体

Syntax:	**proxy_busy_buffers_size** size;
Default:	proxy_busy_buffers_size 8k|16k;
Context:	http, server, location

接收上游时网络速度相关指令

极客时间

Syntax:	**proxy_read_timeout** *time*;
Default:	proxy_read_timeout 60s;
Context:	http, server, location

Syntax:	**proxy_limit_rate** *rate*;
Default:	proxy_limit_rate 0;
Context:	http, server, location

上游包体的持久化

Syntax:	**proxy_store_access** users:permissions ...;
Default:	proxy_store_access user:rw;
Context:	http, server, location

Syntax:	**proxy_store** on | off | string;
Default:	proxy_store off;
Context:	http, server, location

返回响应-加工响应内容


```
&ngx_http_write_filter_module,
&ngx_http_header_filter_module,
&ngx_http_chunked_filter_module,
&ngx_http_v2_filter_module,
&ngx_http_range_header_filter_module,
&ngx_http_gzip_filter_module,
&ngx_http_postpone_filter_module,
&ngx_http_ssi_filter_module,
&ngx_http_charset_filter_module,
&ngx_http_sub_filter_module,
&ngx_http_addition_filter_module,
&ngx_http_userid_filter_module,
&ngx_http_headers_filter_module,
&ngx_http_echo_module,
&ngx_http_xss_filter_module,
&ngx_http_srcache_filter_module,
&ngx_http_lua_module,
&ngx_http_headers_more_filter_module,
&ngx_http_rds_json_filter_module,
&ngx_http_rds_csv_filter_module,
&ngx_http_copy_filter_module,
&ngx_http_range_body_filter_module,
&ngx_http_not_modified_filter_module,
```

禁用上游响应头部的功能

Syntax:	**proxy_ignore_headers** *field* ...;
Default:	—
Context:	http, server, location

- **功能**
  - 某些响应头部可以改变nginx的行为，使用proxy_ignore_headers可以禁止它们生效
- **可以禁用功能的头部**
  - X-Accel-Redirect：由上游服务指定在nginx内部重定向，控制请求的执行
  - X-Accel-Limit-Rate：由上游设置发往客户端的速度限制，等同limit_rate指令
  - X-Accel-Buffering：由上游控制是否缓存上游的响应
  - X-Accel-Charset：由上游控制Content-Type中的Charset
  - 缓存相关：
    - X-Accel-Expires：设置响应在nginx中的缓存时间，单位秒；@开头表示一天内某时刻
    - Expires：控制nginx缓存时间，优先级低于X-Accel-Expires
    - Cache-Control：控制nginx缓存时间，优先级低于X-Accel-Expires
    - Set-Cookie：响应中出现Set-Cookie则不缓存，可通过proxy_ignore_headers禁止生效
    - Vary：响应中出现Vary: *则不缓存，同样可禁止生效

转发上游的响应：Proxy_hide_header指令

Syntax:	**proxy_hide_header** *field*;
Default:	—
Context:	http, server, location

- **proxy_hide_header功能**：
  - 对于上游响应中的某些头部，设置不向客户端转发
- **proxy_hide_header默认不转发响应头部**：
  - Date：由ngx_http_header_filter_module过滤模块填写，值为nginx发送响应头部时的时间
  - Server：由ngx_http_header_filter_module过滤模块填写，值为nginx版本
  - X-Pad：通常是Apache为避免浏览器BUG生成的头部，默认忽略
  - X-Accel-：用于控制nginx行为的响应，不需要向客户端转发
- **proxy_pass_header**
  - 对于已经被proxy_hide_header的头部，设置向上游转发

Syntax:	**proxy_pass_header** *field*;
Default:	—
Context:	http, server, location

修改返回的Set-Cookie头部

Syntax:	**proxy_cookie_domain** off; **proxy_cookie_domain** *domain replacement;*
Default:	proxy_cookie_domain off;
Context:	http, server, location

Syntax:	**proxy_cookie_path** off; **proxy_cookie_path** *path replacement*;
Default:	proxy_cookie_path off;
Context:	http, server, location

修改返回的Location头部

Syntax:	**proxy_redirect** default; **proxy_redirect** off; **proxy_redirect** redirect replacement;
Default:	proxy_redirect default;
Context:	http, server, location

上游返回失败时的处理办法

Syntax:	**proxy_next_upstream** error | timeout | invalid_header | http_500 | http_502 | http_503 | http_504 | http_403 | http_404 | http_429 | non_idempotent | off ...;
Default:	proxy_next_upstream error timeout;
Context:	http, server, location

- **前提**
  - 没有向客户端发送任何内容
- **配置**
  - error
  - timeout
  - invalid_header
  - http_
  - non_idempotent
  - off

限制proxy_next_upstream的时间与次数

Syntax:	**proxy_next_upstream_timeout** *time*;
Default:	proxy_next_upstream_timeout 0;
Context:	http, server, location

Syntax:	**proxy_next_upstream_tries** *number*;
Default:	proxy_next_upstream_tries 0;
Context:	http, server, location

用error_page拦截上游失败响应

当上游响应的响应码大于等于300时，应将响应返回客户端还是按error_page指令处理

Syntax:	**proxy_intercept_errors** on | off;
Default:	proxy_intercept_errors off;
Context:	http, server, location

双向认证时的指令示例


指定自身使用的证书：
proxy_ssl_certificate
proxy_ssl_certificate_key
验证客户端证书：
ssl_verify_client
ssl_client_certificate
http
https双向认证
客户端
nginx
nginx
验证服务器证书：
proxy_ssl_verify
proxy_ssl_trusted_certificate
指定自身使用的证书：
ssl_certificate
ssl_certificate_key

对下游使用证书

Syntax:	**ssl_certificate** *file*;
Default:	—
Context:	http, server

Syntax:	**ssl_certificate_key** *file*;
Default:	—
Context:	http, server

验证下游证书

Syntax:	**ssl_verify_client** on | off | optional | optional_no_ca;
Default:	ssl_verify_client off;
Context:	http, server

Syntax:	**ssl_client_certificate** *file*;
Default:	—
Context:	http, server

对上游使用证书

Syntax:	**proxy_ssl_certificate** *file*;
Default:	—
Context:	http, server, location

Syntax:	**proxy_ssl_certificate_key** *file*;
Default:	—
Context:	http, server, location

验证上游的证书

Syntax:	**proxy_ssl_trusted_certificate** *file*;
Default:	—
Context:	http, server, location

Syntax:	**proxy_ssl_verify** on | off;
Default:	proxy_ssl_verify off;
Context:	http, server, location

ssl模块提供的变量（1）

- **安全套件**
  - ssl_cipher: 本次通讯选用的安全套件，例如ECDHE-RSA-AES128-GCM-SHA256
  - ssl_ciphers: 客户端支持的所有安全套件
  - ssl_protocol: 本次通讯选用的TLS版本，例如TLSv1.2
  - ssl_curves: 客户端支持的椭圆曲线，例如secp384r1:secp521r1
- **证书**
  - ssl_client_raw_cert: 原始客户端证书内容
  - ssl_client_escaped_cert: 返回客户端证书做urlencode编码后的内容
  - ssl_client_cert: 对客户端证书每一行内容前加tab制表符空白，增强可读性。
  - ssl_client_fingerprint: 客户端证书的SHA1指纹

ssl模块提供的变量（2）

- **证书结构化信息**
  - ssl_server_name: 通过TLS插件SNI(Server Name Indication)获取到的服务域名
  - ssl_client_i_dn:依据RFC2253获取到证书issuer dn信息，例如： CN=…,O=…,L=…,C=…
  - ssl_client_i_dn_legacy:依据RFC2253获取到证书issuer dn信息，例如： /C=…/L=…/O=…/CN=…
  - ssl_client_s_dn: 依据RFC2253获取到证书subject dn信息，例如：CN=…,OU=…,O=…,L=…,ST=…,C=…
  - ssl_client_s_dn_legacy: 同样获取subject dn信息，格式为：/C=…/ST=…/L=…/O=…/OU=…/CN=…
- **证书有效期**
  - ssl_client_v_end: 返回客户端证书的过期时间，例如Dec 1 11:56:11 2028 GMT
  - ssl_client_v_remain: 返回还有多少天客户端证书过期，例如针对上面的ssl_client_v_end其值为3649
  - ssl_client_v_start: 客户端证书的颁发日期，例如Dec 4 11:56:11 2018 GMT
- **连接有效性**
  - ssl_client_serial: 返回连接上客户端证书的序列号，例如8BE947674841BD44
  - ssl_early_data: 在TLS1.3协议中使用了early data且握手未完成返回1，否则返回空字符串
  - ssl_client_verify: 如果验证失败为FAILED:原因，如果没有验证证书则为NONE，验证成功则为SUCCESS
  - ssl_session_id: 已建立连接的sessionid
  - ssl_session_reused: 如果session被复用（参考session缓存）则为r，否则为.

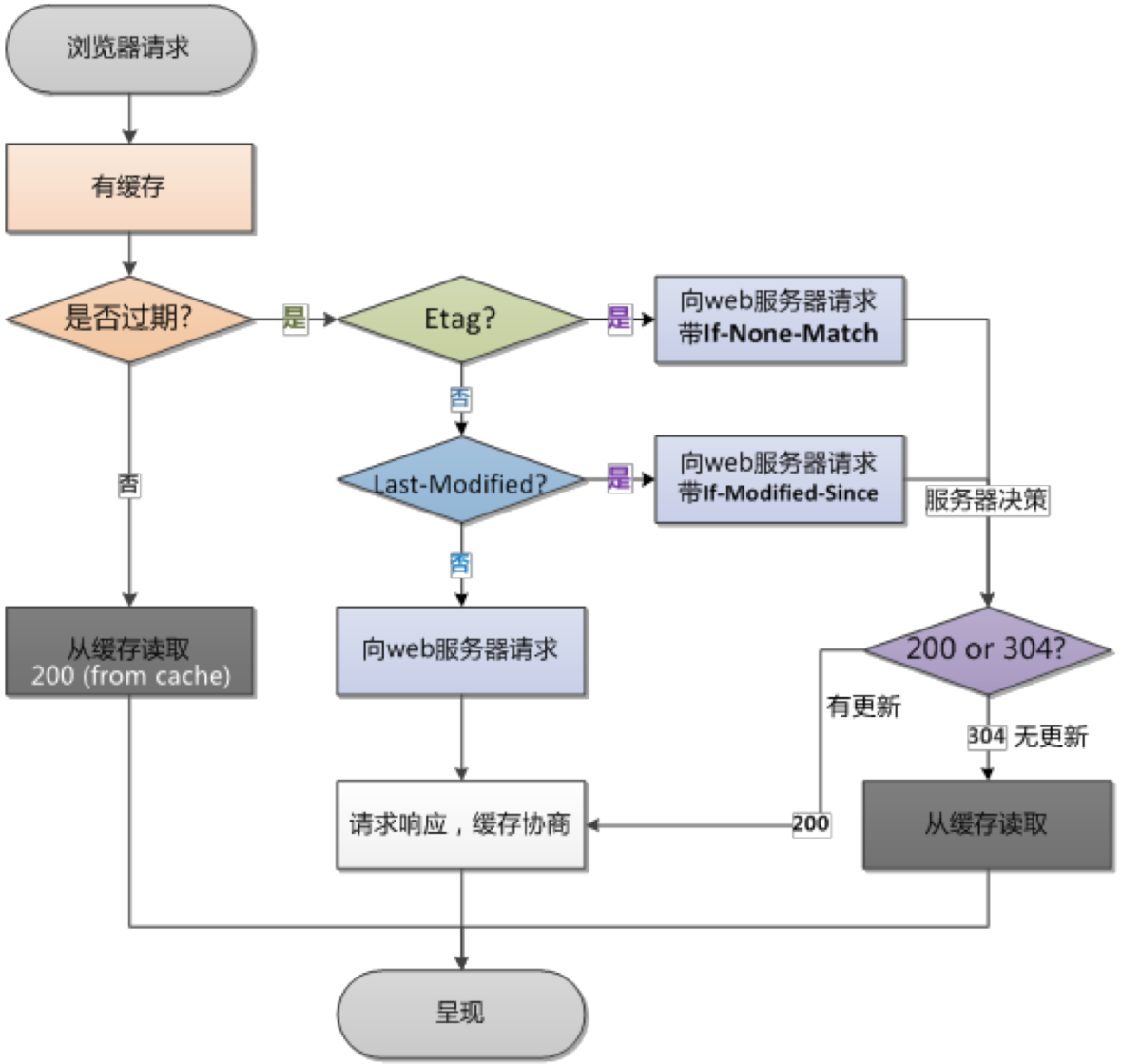
创建证书命令示例

- **创建根证书**
  - 创建CA私钥
    - openssl genrsa -out ca.key 2048
  - 制作CA公钥
    - openssl req -new -x509 -days 3650 -key ca.key -out ca.crt
- **签发证书**
  - 创建私钥
    - openssl genrsa -out a.pem 1024
    - openssl rsa -in a.pem -out a.key
  - 生成签发请求
    - openssl req -new -key a.pem -out a.csr
  - 使用CA证书进行签发
    - openssl x509 -req -sha256 -in a.csr -CA ca.crt -CAkey ca.key -CAcreateserial -days 3650 -out a.crt
  - 验证签发证书是否正确
    - openssl verify -CAfile ca.crt a.crt

浏览器缓存与nginx缓存

- **浏览器缓存**
  - 优点
    - 使用有效缓存时，没有网络消耗，速度最快
    - 即使有网络消耗，但对失效缓存使用304响应做到网络流量消耗最小化
  - 缺点
    - 仅提升一个用户的体验
- **nginx缓存**
  - 优点
    - 提升所有用户的体验
    - 相比浏览器缓存，有效降低上游服务的负载
    - 通过304响应减少nginx与上游服务间的流量消耗
  - 缺点
    - 用户仍然保持网络消耗
- **同时使用浏览器与nginx缓存**

浏览器缓存


浏览器请求
有缓存
是否过期?
是
Etag?
是
向web服务器请求
带If-None-Match
否
Last-Modified?
是
向web服务器请求
带If-Modified-Since
否
服务器决策
否
从缓存读取
200 (from cache)
向web服务器请求
200 or 304?
有更新
304 无更新
请求响应，缓存协商
200
从缓存读取
呈现

Etag头部

ETagHTTP响应头是资源的特定版本的标识符。这可以让缓存更高效，并节省带宽，因为如果内容没有改变，Web服务器不需要发送完整的响应。而如果内容发生了变化，使用ETag有助于防止资源的同时更新相互覆盖（“空中碰撞”）。

如果给定URL中的资源更改，则一定要生成新的Etag值。 因此Etags类似于指纹，也可能被某些服务器用于跟踪。 比较etags能快速确定此资源是否变化，但也可能被跟踪服务器永久存留。

W/ 可选

'W/'(大小写敏感) 表示使用弱验证器。 弱验证器很容易生成，但不利于比较。

强验证器是比较的理想选择，但很难有效地生成。 相同资源的两个弱Etag值可能语义等同，但不是每个字节都相同。

etag指令

Syntax:	**etag** on | off;
Default:	etag on;
Context:	http, server, location

生成规则：

```
ngx_sprintf(etag->value.data, "\"%xT-%xO\"",
                              r->headers_out.last_modified_time,
                              r->headers_out.content_length_n)
```

If-None-Match

If-None-Match 是一个条件式请求首部。对于 GETGET 和 HEAD 请求方法来说，当且仅当服务器上没有任何资源的 ETag 属性值与这个首部中列出的相匹配的时候，服务器端会才返回所请求的资源，响应码为 200 。对于其他方法来说，当且仅当最终确认没有已存在的资源的 ETag 属性值与这个首部中所列出的相匹配的时候，才会对请求进行相应的处理。

对于 GET 和 HEAD 方法来说，当验证失败的时候，服务器端必须返回响应码 304 （ Not Modified，未改变 ）。对于能够引发服务器状态改变的方法，则返回 412 （ Precondition Failed，前置条件失败 ）。需要注意的是，服务器端在生成状态码为 304 的响应的时候，必须同时生成以下会存在于对应的 200 响应中的首部：Cache-Control、Content-Location、Date、ETag、Expires 和 Vary 。

ETag 属性之间的比较采用的是**弱比较算法**，即两个文件除了每个比特都相同外，内容一致也可以认为是相同的。例如，如果两个页面仅仅在页脚的生成时间有所不同，就可以认为二者是相同的。
当与 If-Modified-Since 一同使用的时候，If-None-Match 优先级更高（假如服务器支持的话）。

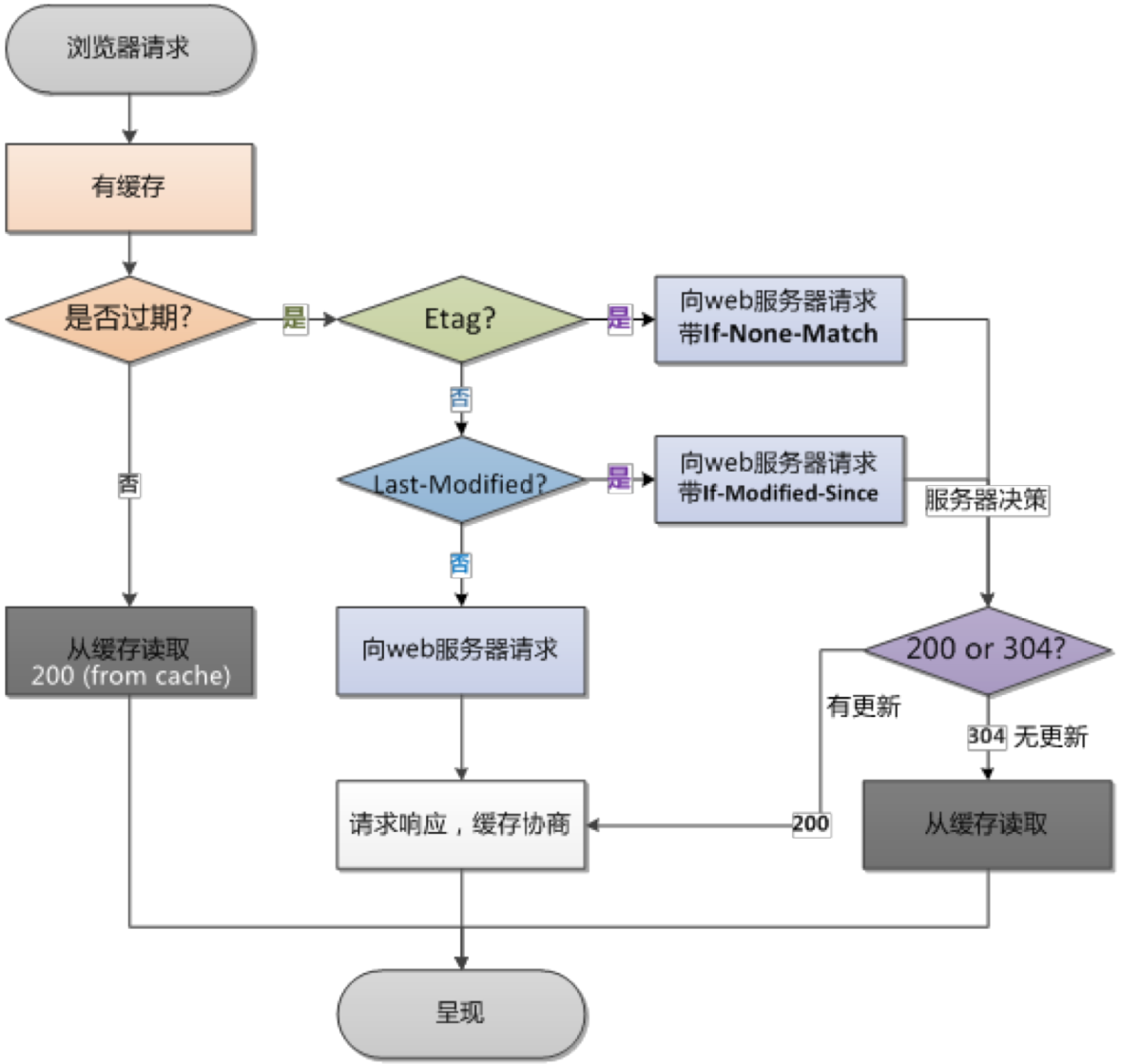
以下是两个常见的应用场景：
•采用 GET 或 HEAD 方法，来更新拥有特定的ETag 属性值的缓存。
•采用其他方法，尤其是 PUT，将 If-None-Match used 的值设置为 * ，用来生成事先并不知道是否存在的文件，可以确保先前并没有进行过类似的上传操作，防止之前操作数据的丢失。这个问题属于更新丢失问题的一种。

If-Modified-Since头部

If-Modified-Since 是一个条件式请求首部，服务器只在所请求的资源在给定的日期时间之后对内容进行过修改的情况下才会将资源返回，状态码为 200 。如果请求的资源从那时起未经修改，那么返回一个不带有消息主体的 304 响应，而在 Last-Modified 首部中会带有上次修改时间。 不同于 If-Unmodified-Since, If-Modified-Since 只可以用在 GET 或 HEAD 请求中。

当与 If-None-Match 一同出现时，它（If-Modified-Since）会被忽略掉，除非服务器不支持 If-None-Match。

浏览器缓存


浏览器请求
有缓存
是否过期?
是
Etag?
是
向web服务器请求
带If-None-Match
否
Last-Modified?
是
向web服务器请求
带If-Modified-Since
否
服务器决策
否
从缓存读取
200 (from cache)
向web服务器请求
200 or 304?
有更新
304 无更新
请求响应，缓存协商
200
从缓存读取
呈现

not_modified过滤模块

- **功能**
  - 客户端拥有缓存，但不确认缓存是否过期，于是在请求中传入 If-Modified-Since或者If-None-Match头部，该模块通过将其值与响应中的Last-Modified值相比较，决定是通过200返回全部内容，还是仅返回304 Not Modified头部，表示浏览器仍使用之前的缓存。

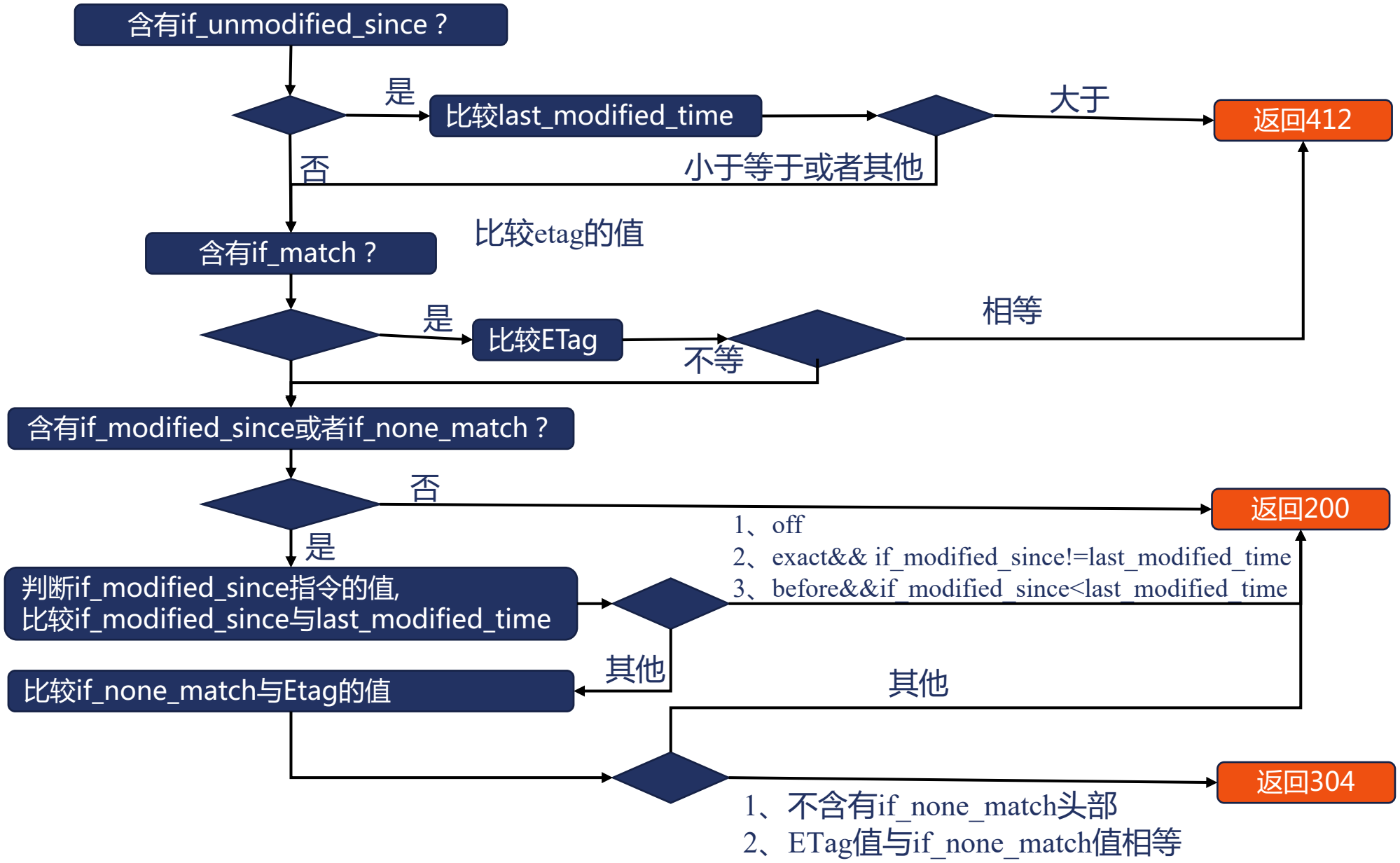
- **使用前提**
  - 原返回响应码为200

expires指令

Syntax:	**expires** [modified] *time*; **expires** epoch | max | off;
Default:	expires off;
Context:	http, server, location, if in location

- **max:**
  - Expires: Thu, 31 Dec 2037 23:55:55 GMT
  - Cache-Control: max-age=315360000（10年）
- **off：不添加或者修改Expires和Cache-Control字段**
- **epoch：**
  - Expires: Thu, 01 Jan 1970 00:00:01 GMT
  - Cache-Control: no-cache
- **time：设定具体时间，可以携带单位**
  - 一天内的具体时刻可以加@，比如下午六点半：@18h30m
    - 设定好Expires，自动计算Cache-Control
    - 如果当前时间未超过当天的time时间，则Expires到当天time，否则是第二天的time时刻
  - 正数
    - 设定Cache-Control时间，计算出Expires
  - 负数
    - Cache-Control: no-cache ，计算出Expires

not_modified过滤模块



not_modified过滤模块

Syntax:	**if_modified_since** off | exact | before;
Default:	if_modified_since exact;
Context:	http, server, location

- **off**
  - 忽略请求中的if_modified_since头部
- **exact**
  - 精确匹配if_modified_since头部与last_modified的值
- **before**
  - 若if_modified_since大于等于last_modified的值，则返回304

If-Match

请求首部 **If-Match** 的使用表示这是一个条件请求。在请求方法为 GET 和 HEAD 的情况下，服务器仅在请求的资源满足此首部列出的 ETag 之一时才会返回资源。而对于 PUT 或其他非安全方法来说，只有在满足条件的情况下才可以将资源上传。

The comparison with the stored ETag 之间的比较使用的是**强比较算法**，即只有在每一个比特都相同的情况下，才可以认为两个文件是相同的。在 ETag 前面添加 W/ 前缀表示可以采用相对宽松的算法。

以下是两个常见的应用场景：

For GET 和 HEAD 方法，搭配 Range首部使用，可以用来保证新请求的范围与之前请求的范围是对同一份资源的请求。如果 ETag 无法匹配，那么需要返回 416 (Range Not Satisfiable，范围请求无法满足) 响应。

对于其他方法来说，尤其是 PUT, If-Match 首部可以用来避免更新丢失问题。它可以用来检测用户想要上传的不会覆盖获取原始资源之后做出的更新。如果请求的条件不满足，那么需要返回 412 (Precondition Failed，先决条件失败) 响应。

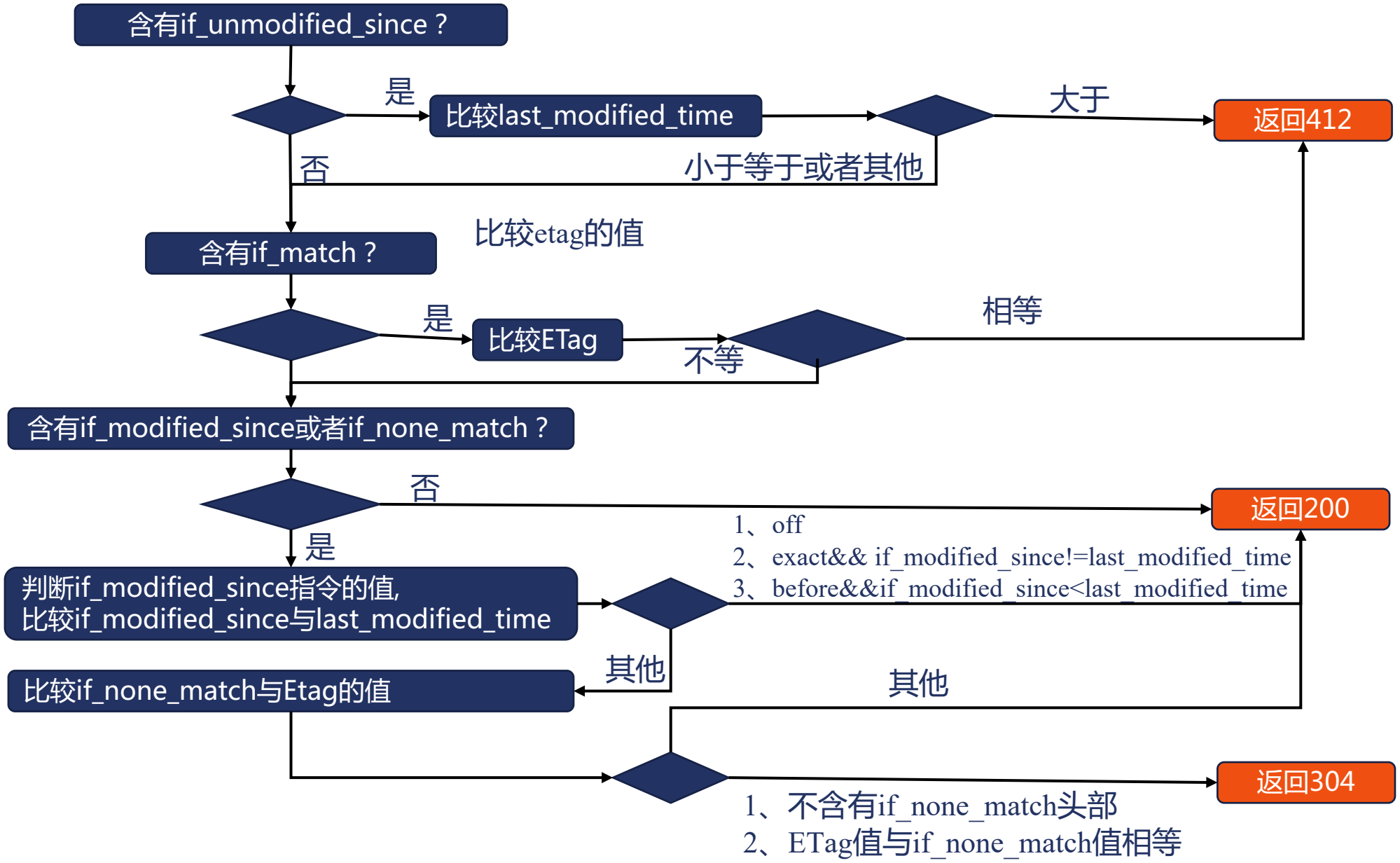
If-Unmodified-Since

HTTP协议中的 **If-Unmodified-Since** 消息头用于请求之中，使得当前请求成为条件式请求：只有当资源在指定的时间之后没有进行过修改的情况下，服务器才会返回请求的资源，或是接受 POST 或其他 non-safe 方法的请求。如果所请求的资源在指定的时间之后发生了修改，那么会返回 412 (Precondition Failed) 错误。
常见的应用场景有两种：

•与 non-safe 方法如 POST 搭配使用，可以用来优化并发控制，例如在某些wiki应用中的做法：假如在原始副本获取之后，服务器上所存储的文档已经被修改，那么对其作出的编辑会被拒绝提交。

•与含有 If-Range 消息头的范围请求搭配使用，用来确保新的请求片段来自于未经修改的文档。

not_modified过滤模块



nginx缓存：定义存放缓存的载体

Syntax:	**proxy_cache** zone | off;
Default:	proxy_cache off;
Context:	http, server, location

Syntax:	**proxy_cache_path** path [levels=levels] [use_temp_path=on|off] keys_zone=name:size [inactive=time] [max_size=size] [manager_files=number] [manager_sleep=time] [manager_threshold=time] [loader_files=number] [loader_sleep=time] [loader_threshold=time] [purger=on|off] [purger_files=number] [purger_sleep=time] [purger_threshold=time];
Default:	—
Context:	http

proxy_cache_path指令（1）

- **path**
  - 定义缓存文件存放位置
- **levels**
  - 定义缓存路径的目录层级，最多3级，每层目录长度为1或者2字节
- **use_temp_path**
  - on使用proxy_temp_path定义的临时目录
  - off直接使用path路径存放临时文件
- **keys_zone**
  - name是共享内存名字，由proxy_cache指令使用
  - size是共享内存大小，1MB大约可以存放8000个key
- **inactive**
  - 在inactive时间内没有被访问的缓存，会被淘汰掉
  - 默认10分钟
- **max_size**
  - 设置最大的缓存文件大小，超出后由cache manager进程按LRU链表淘汰

proxy_cache_path指令（2）

- **manager_files**
  - cache manager进程在1次淘汰过程中，淘汰的最大文件数
  - 默认100
- **manager_sleep**
  - 执行一次淘汰循环后cache manager进程的休眠时间
  - 默认200毫秒
- **manager_threshold**
  - 执行一次淘汰循环的最大耗时
  - 默认50毫秒
- **loader_files**
  - cache loader进程载入磁盘中缓存文件至共享内存，每批最多处理的文件数
  - 默认100
- **loader_sleep**
  - 执行一次缓存文件至共享内存后，进程休眠的时间
  - 载入默认200毫秒
- **loader_threshold**
  - 每次载入缓存文件至共享内存的最大耗时
  - 默认50毫秒

缓存的关键字

Syntax:	**proxy_cache_key** *string*;
Default:	proxy_cache_key $scheme$proxy_host$request_uri;
Context:	http, server, location

缓存什么样的响应

Syntax:	**proxy_cache_valid** *[code ...] time*;
Default:	—
Context:	http, server, location

- **对不同的响应码缓存不等的时长**
  - 例如：code 404 5m;
- **只标识时间**
  - 仅对以下响应码缓存
    - 200
    - 301
    - 302
- **通过响应头部控制缓存时长**
  - X-Accel-Expires，单位秒
    - 为0时表示禁止nginx缓存内容
    - 通过@设置缓存到一天中的某一时刻
  - 响应头若含有Set-Cookie则不缓存
  - 响应头含有Vary: *则不缓存

哪些内容不使用缓存？

参数为真时，响应不存入缓存

Syntax:	**proxy_no_cache** *string* ...;
Default:	—
Context:	http, server, location

参数为真时，不使用缓存内容

Syntax:	**proxy_cache_bypass** *string* ...;
Default:	—
Context:	http, server, location

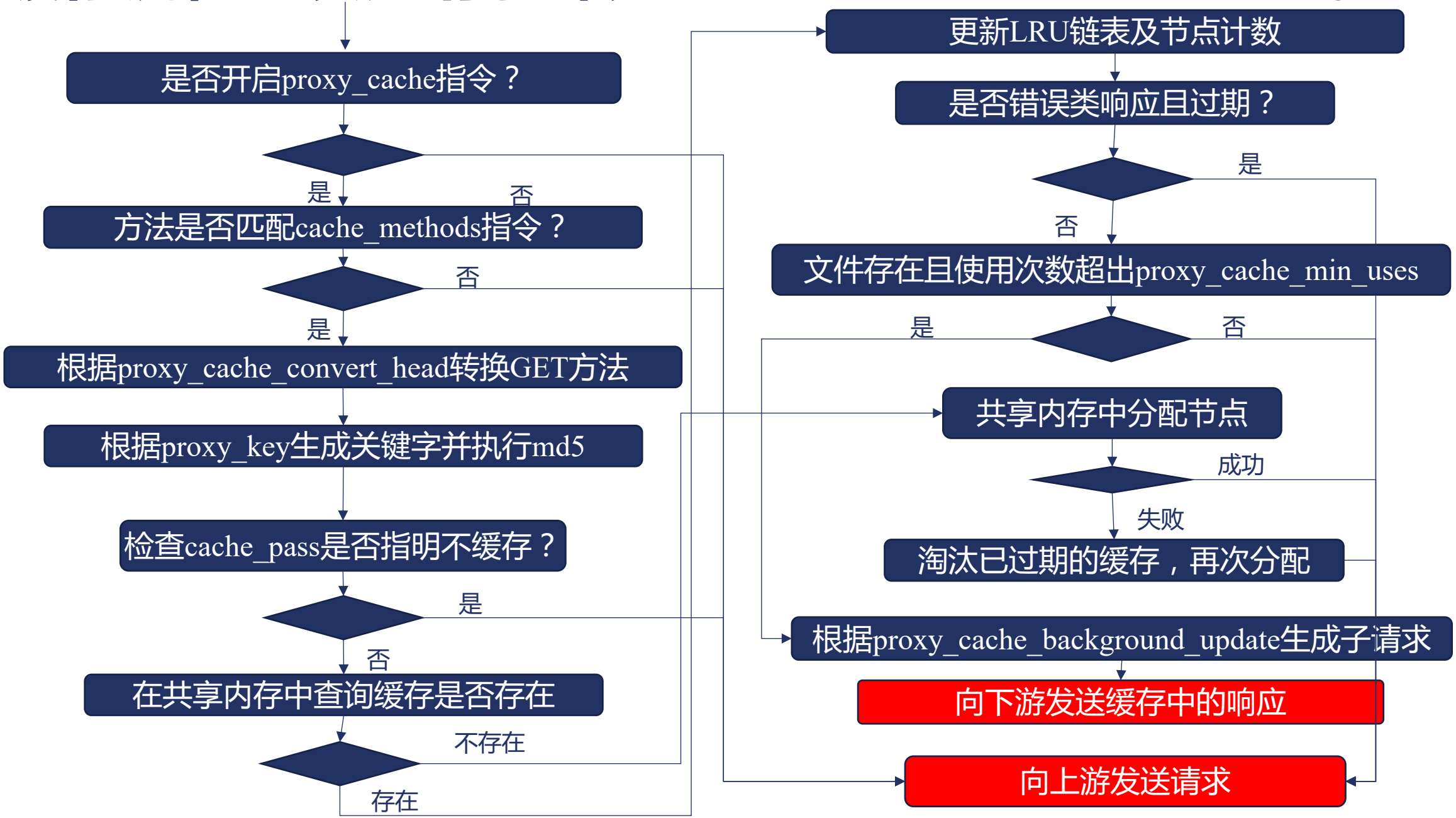
变更HEAD方法

Syntax:	**proxy_cache_convert_head** on | off;
Default:	proxy_cache_convert_head on;
Context:	http, server, location

upstream_cache_status变量

- **upstream_cache_status**
  - MISS：未命中缓存
  - HIT：命中缓存
  - EXPIRED：缓存已经过期
  - STALE：命中了陈旧的缓存，
  - UPDATING：内容陈旧，但正在更新
  - REVALIDATED：Nginx验证了陈旧的内容依然有效
  - BYPASS：响应是从原始服务器获得的

缓存流程：发起请求部分


是否开启proxy_cache指令？
是
否
方法是否匹配cache_methods指令？
否
是
根据proxy_cache_convert_head转换GET方法
根据proxy_key生成关键字并执行md5
检查cache_pass是否指明不缓存？
是
否
在共享内存中查询缓存是否存在
不存在
存在
更新LRU链表及节点计数
是否错误类响应且过期？
是
否
文件存在且使用次数超出proxy_cache_min_uses
是
否
共享内存中分配节点
成功
失败
淘汰已过期的缓存，再次分配
根据proxy_cache_background_update生成子请求
向下游发送缓存中的响应
向上游发送请求

对哪个method方法使用缓存返回响应

Syntax:	**proxy_cache_methods** GET | HEAD | POST ...;
Default:	proxy_cache_methods GET HEAD;
Context:	http, server, location

X-Accel-Expires头部

X-Accel-Expires

Syntax:	X-Accel-Expires [offseconds]
Default:	X-Accel-Expires off

从上游服务定义缓存多长时间

0表示不缓存当前响应

@前缀表示缓存到当天的某个时间

Vary头部

Vary 是一个HTTP响应头部信息，它决定了对于未来的一个请求头，应该用一个缓存的回复(response)还是向源服务器请求一个新的回复。它被服务器用来表明在 content negotiation algorithm（内容协商算法）中选择一个资源代表的时候应该使用哪些头部信息（headers）.

在响应状态码为 304 Not Modified 的响应中，也要设置 Vary 首部，而且要与相应的 200 OK 响应设置得一模一样。

- Vary: *
  - 所有的请求都被视为唯一并且非缓存的，使用Cache-Control: private,来实现则更适用，这样用于说明不存储该对象更加清晰。
  - 若没有通过proxy_ignore_headers设置忽略，则不缓存响应
- Vary: <header-name>, <header-name>, ...
  - 逗号分隔的一系列http头部名称，用于确定缓存是否可用。

Set-Cookie头部

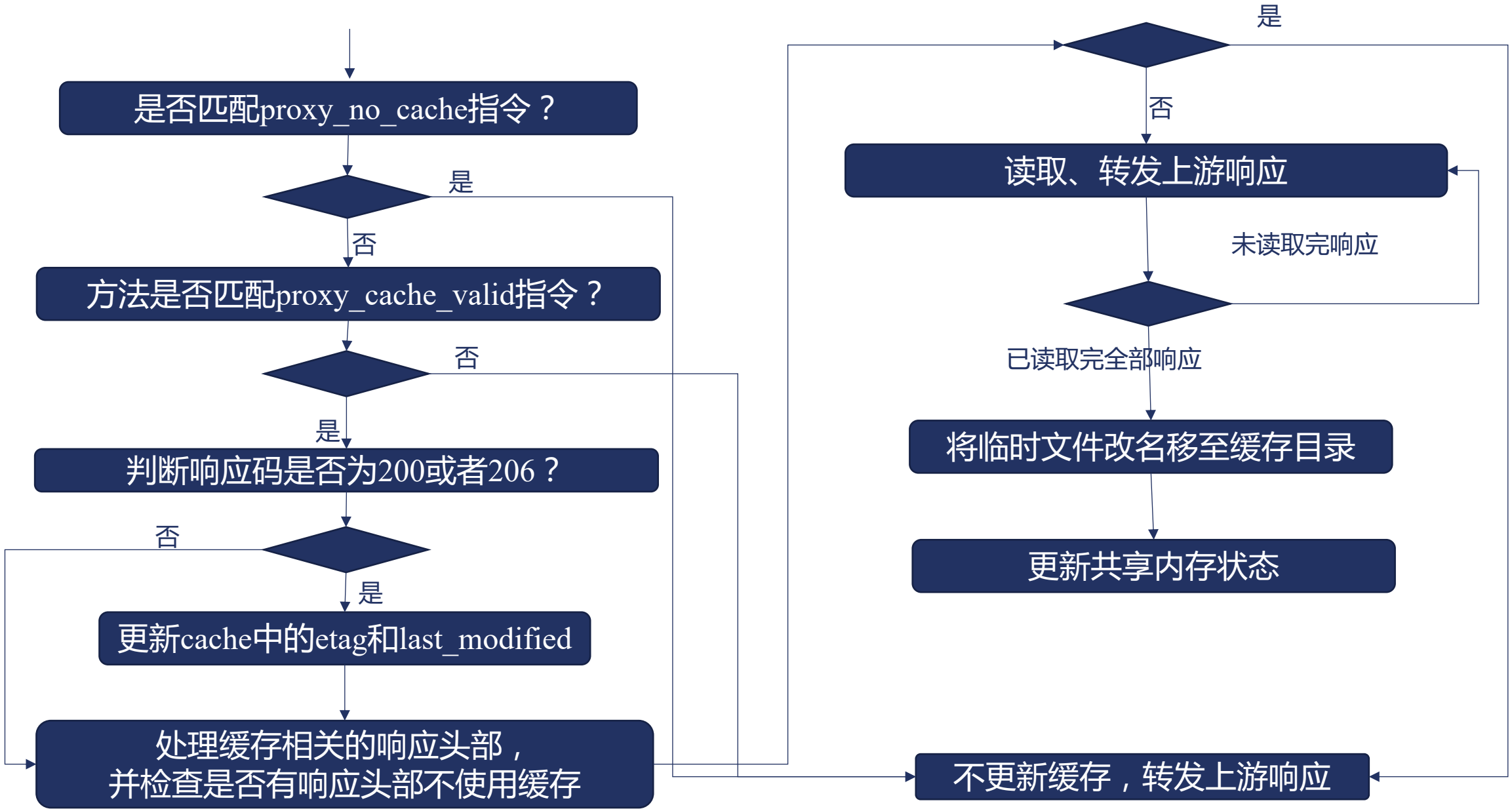
Set-Cookie: <cookie-name>=<cookie-value>
Set-Cookie: <cookie-name>=<cookie-value>; Expires=<date>
Set-Cookie: <cookie-name>=<cookie-value>; Max-Age=<non-zero-digit>
Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>
Set-Cookie: <cookie-name>=<cookie-value>; Path=<path-value>
Set-Cookie: <cookie-name>=<cookie-value>; Secure
Set-Cookie: <cookie-name>=<cookie-value>; HttpOnly

Set-Cookie: <cookie-name>=<cookie-value>; SameSite=Strict
Set-Cookie: <cookie-name>=<cookie-value>; SameSite=Lax

Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>; Secure; HttpOnly

若Set-Cookie头部没有被proxy_ignore_headers设置忽略，则不对响应进行缓存

缓存流程：接收上游响应


是否匹配proxy_no_cache指令？
是
否
方法是否匹配proxy_cache_valid指令？
否
是
判断响应码是否为200或者206？
否
是
更新cache中的etag和last_modified
处理缓存相关的响应头部，
并检查是否有响应头部不使用缓存
是
否
读取、转发上游响应
未读取完响应
已读取完全部响应
将临时文件改名移至缓存目录
更新共享内存状态
不更新缓存，转发上游响应

合并回源请求—减轻峰值流量下的压力

Syntax:	**proxy_cache_lock** on | off;
Default:	proxy_cache_lock off;
Context:	http, server, location

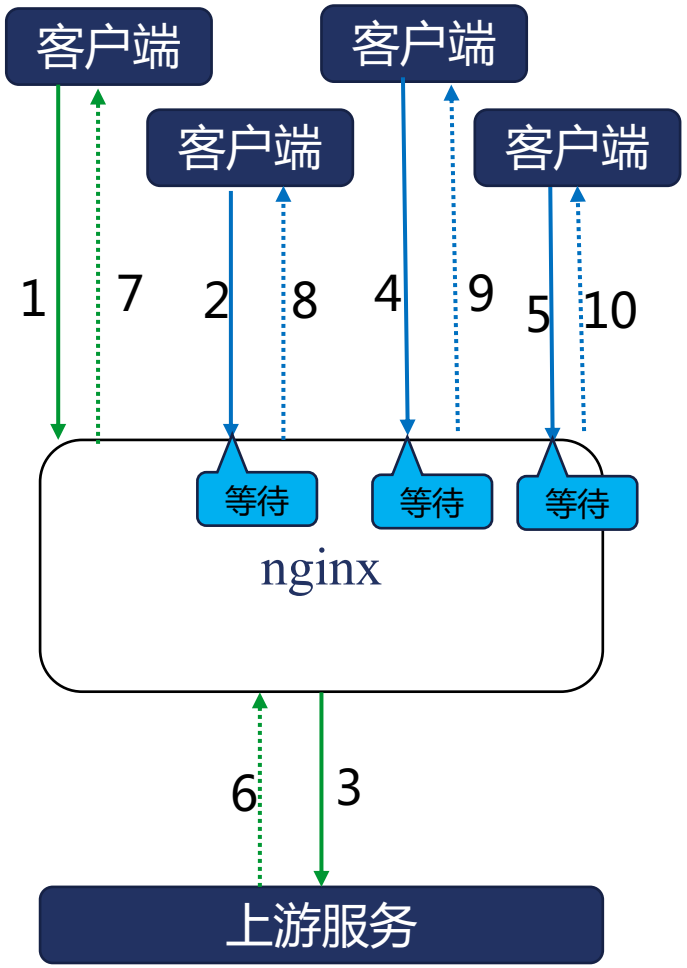
同一时间，仅第1个请求发向上游，其他请求等待第1个响应返回或者超时后，使用缓存响应客户端

Syntax:	**proxy_cache_lock_timeout** *time*;
Default:	proxy_cache_lock_timeout 5s;
Context:	http, server, location

等待第1个请求返回响应的最大时间，到达后直接向上游发送请求，但不缓存响应

Syntax:	**proxy_cache_lock_age** *time*;
Default:	proxy_cache_lock_age 5s;
Context:	http, server, location

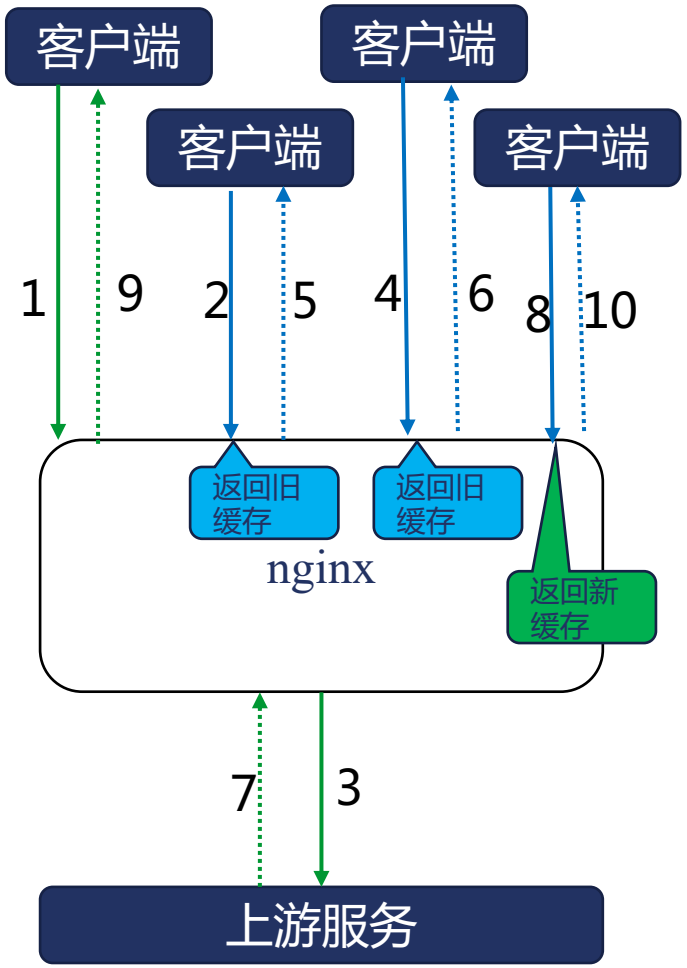
上一个请求返回响应的超时时间，到达后再放行一个请求发向上游



减少回源请求—使用stale陈旧的缓存

Syntax:	**proxy_cache_use_stale** error | timeout | invalid_header | updating | http_500 | http_502 | http_503 | http_504 | http_403 | http_404 | http_429 | off ...;
Default:	proxy_cache_use_stale off;
Context:	http, server, location

Syntax:	**proxy_cache_background_update** on | off;
Default:	proxy_cache_background_update off;
Context:	http, server, location



proxy_cache_use_stale指令：定义陈旧缓存的用法

- **updating**
  - 当缓存内容过期，有一个请求正在访问上游试图更新缓存时，其他请求直接使用过期内容返回客户端
  - stale-while-revalidate
    - 缓存内容过期后，定义一段时间，在这段时间内updating设置有效，否则请求仍然访问上游服务
    - 例如：Cache-Control: max-age=600, stale-while-revalidate=30
  - stale-if-error
    - 缓存内容过期后，定义一段时间，在这段时间内上游服务出错后就继续使用缓存，否则请求仍然访问上游服务。stale-while-revalidate包括stale-if-error场景
    - 例如： Cache-Control: max-age=600, stale-if-error=1200
- **error**
  - 当与上游建立连接、发送请求、读取响应头部等情况出错时，使用缓存
- **timeout**
  - 当与上游建立连接、发送请求、读取响应头部等情况出现定时器超时，使用缓存
- **http**_(500|502|503|504|403|404|429)
  - 缓存以上错误响应码的内容

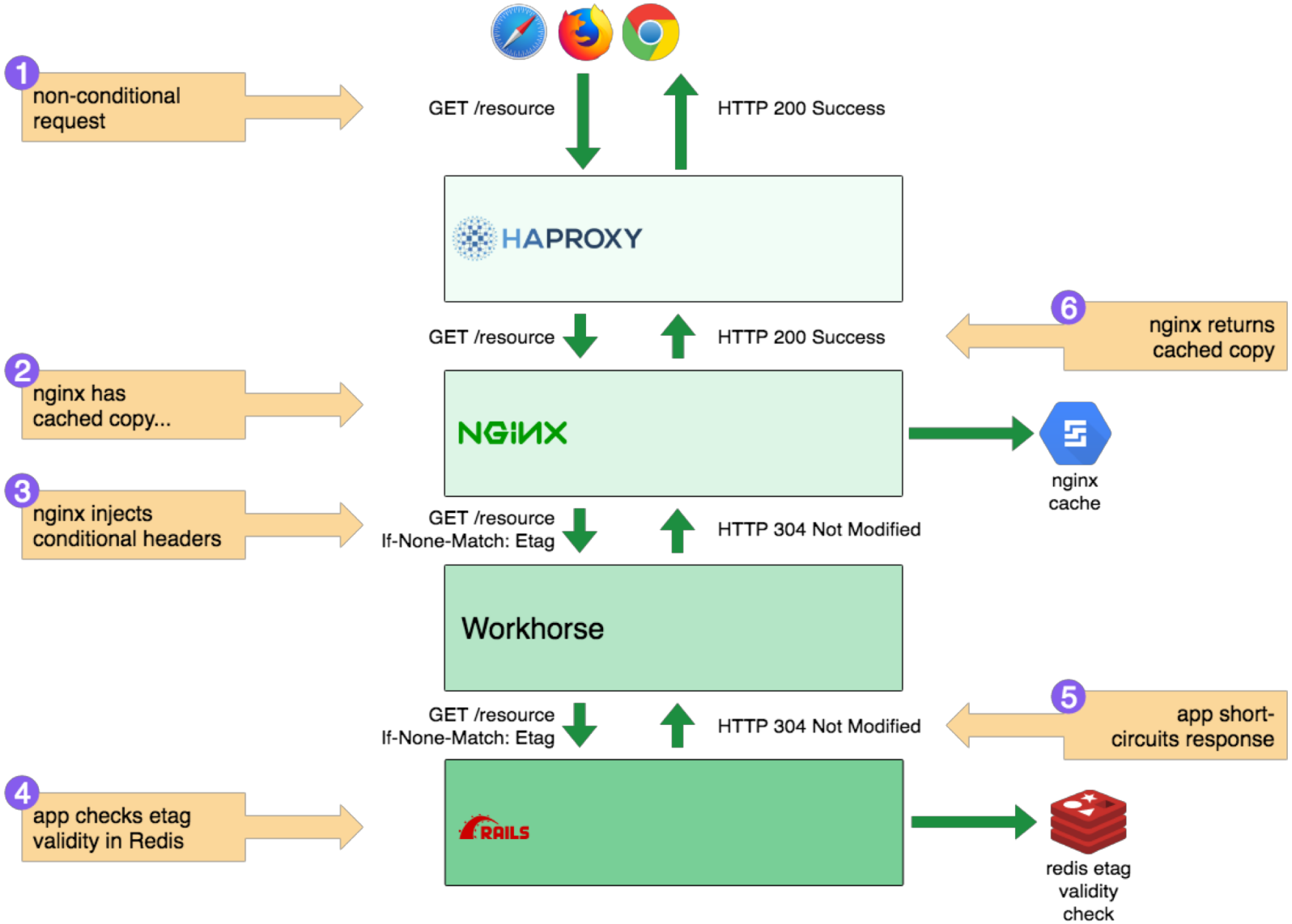
缓存有问题的响应

Syntax:	**proxy_cache_background_update** on | off;
Default:	proxy_cache_background_update off;
Context:	http, server, location

当使用proxy_cache_use_stale允许使用过期响应时，将同步生成一个子请求，通过访问上游服务更新过期的缓存

Syntax:	**proxy_cache_revalidate** on | off;
Default:	proxy_cache_revalidate off;
Context:	http, server, location

更新缓存时，使用If-Modified-Since和If-None-Match作为请求头部，预期内容未发生变更时通过304来减少传输的内容


1
non-conditional request
GET /resource
HTTP 200 Success
HAPROXY
6
nginx returns cached copy
GET /resource
HTTP 200 Success
2
nginx has cached copy...
NGINX
nginx cache
3
nginx injects conditional headers
GET /resource
If-None-Match: Etag
HTTP 304 Not Modified
Workhorse
5
app short-circuits response
GET /resource
If-None-Match: Etag
HTTP 304 Not Modified
4
app checks etag validity in Redis
RAILS
redis etag validity check

及时清除缓存

- **模块**
  - 第三方模块ngx_cache_purge ：https://github.com/FRiCKLE/ngx_cache_purge
  - 使用--add-module=指令添加模块到nginx中
- **功能**
  - 接收到指定HTTP请求后立刻清除缓存

```
•syntax:    proxy_cache_purge on|off|<method> [from all|<ip> [.. <ip>]]
•default:   none
•context:   http, server, location
```

```
•syntax:    proxy_cache_purge zone_name key
•default:   none
•context:   location
```

七层反向代理对照：构造请求内容

	uwsgi反向代理	fastcgi反向代理	scgi反向代理	http反向代理
指定上游	uwsgi_pass	fastcgi_pass	scgi_pass	proxy_pass
是否传递请求头部	uwsgi_pass_request_headers	fastcgi_pass_request_headers	scgi_pass_request_headers	proxy_pass_request_headers
是否传递请求包体	uwsgi_pass_request_body	fastcgi_pass_request_body	scgi_pass_request_body	proxy_pass_request_body
指定请求方法名				proxy_method
指定请求协议				proxy_http_version
增、改请求头部				proxy_set_header
设置请求包体				proxy_set_body
是否缓存请求包体	uwsgi_request_buffering	fastcgi_request_buffering	scgi_request_buffering	proxy_request_buffering

七层反向代理对照：建立连接并发送请求

	uwsgi反向代理	fastcgi反向代理	scgi反向代理	http反向代理
连接上游超时时间	uwsgi_connect_timeout	fastcgi_connect_timeout	scgi_connect_timeout	proxy_connect_timeout
连接绑定地址	uwsgi_bind	fastcgi_bind	scgi_bind	proxy_bind
使用TCPkeepalive	uwsgi_socket_keepalive	fastcgi_socket_keepalive	scgi_socket_keepalive	proxy_socket_keepalive
忽略客户端关连接	uwsgi_ignore_client_abort	fastcgi_ignore_client_abort	scgi_ignore_client_abort	proxy_ignore_client_abort
设置HTTP头部用到的哈希表				proxy_headers_hash_bucket_size
设置HTTP头部用到的哈希表				proxy_headers_hash_max_size
发送请求超时时间	uwsgi_send_timeout	scgi_send_timeout	fastcgi_send_timeout	proxy_send_timeout

七层反向代理对照：接收上游响应

	uwsgi反向代理	fastcgi反向代理	scgi反向代理	http反向代理
是否缓存上游响应	uwsgi_buffering	fastcgi_buffering	scgi_buffering	proxy_buffering
存放上游响应的目录	uwsgi_temp_path	fastcgi_temp_path	scgi_temp_path	proxy_temp_path
写文件缓存大小	uwsgi_temp_file_write_size	fastcgi_temp_file_write_size	scgi_temp_file_write_size	proxy_temp_file_write_size
临时文件最大大小	uwsgi_max_temp_file_size	fastcgi_max_temp_file_size	scgi_max_temp_file_size	proxy_max_temp_file_size
接收响应头部缓存	uwsgi_buffer_size	fastcgi_buffer_size	scgi_buffer_size	proxy_buffer_size
接收响应包体缓存	uwsgi_buffers	fastcgi_buffers	scgi_buffers	proxy_buffers
缓存完成前转发包体	uwsgi_busy_buffers_size	fastcgi_busy_buffers_size	scgi_busy_buffers_size	proxy_busy_buffers_size
持久化包体文件	uwsgi_store	fastcgi_store	scgi_store	proxy_store
设定包体文件权限	uwsgi_store_access	fastcgi_store_access	scgi_store_access	proxy_store_access
读取响应超时时间	uwsgi_read_timeout	fastcgi_read_timeout	scgi_read_timeout	proxy_read_timeout
读取响应限速	uwsgi_limit_rate	fastcgi_limit_rate	scgi_limit_rate	proxy_limit_rate

七层反向代理对照：转发响应

	uwsgi反向代理	fastcgi反向代理	scgi反向代理	http反向代理
减少发向客户端的响应头部	uwsgi_hide_header	fastcgi_hide_header	scgi_hide_header	proxy_hide_header
禁用响应头部功能	uwsgi_ignore_headers	fastcgi_ignore_headers	scgi_ignore_headers	proxy_ignore_headers
替换Set-Cookie头部中的域名				proxy_cookie_domain
替换Set-Cookie头部中的URL				proxy_cookie_path
修改重定向响应中Location的值				proxy_redirect
传递头部到客户端	uwsgi_pass_header	fastcgi_pass_header	scgi_pass_header	proxy_pass_header
出错时更换上游	uwsgi_next_upstream	fastcgi_next_upstream	scgi_next_upstream	proxy_next_upstream
更换上游超时	uwsgi_next_upstream_timeout	fastcgi_next_upstream_timeout	scgi_next_upstream_timeout	proxy_next_upstream_timeout
更换上游重试次数	uwsgi_next_upstream_tries	fastcgi_next_upstream_tries	scgi_next_upstream_tries	proxy_next_upstream_tries
拦截上游错误响应	uwsgi_intercept_errors	fastcgi_intercept_errors	scgi_intercept_errors	proxy_intercept_errors

七层反向代理对照：SSL

	uwsgi反向代理	fastcgi反向代理	scgi反向代理	http反向代理
配置用于上游通讯的证书	uwsgi_ssl_certificate			proxy_ssl_certificate
配置用于上游通讯的私钥	uwsgi_ssl_certificate_key			proxy_ssl_certificate_key
指定安全套件	uwsgi_ssl_ciphers			proxy_ssl_ciphers
指定吊销证书链CRL文件验证上游的证书	uwsgi_ssl_crl			proxy_ssl_crl
指定域名验证上游证书中域名	uwsgi_ssl_name			proxy_ssl_name
当私钥有密码时指定密码文件	uwsgi_ssl_password_file			proxy_ssl_password_file
指定具体某个版本的协议	uwsgi_ssl_protocols			proxy_ssl_protocols
传递SNI信息至上游	uwsgi_ssl_server_name			proxy_ssl_server_name
是否重用SSL连接	uwsgi_ssl_session_reuse			proxy_ssl_session_reuse
验证上游服务的证书	uwsgi_ssl_trusted_certificate			proxy_ssl_trusted_certificate
是否验证上游服务的证书	uwsgi_ssl_verify			proxy_ssl_verify
设置验证证书链的深度	uwsgi_ssl_verify_depth			proxy_ssl_verify_depth

七层反向代理对照：缓存类指令（1）

极客时间

	uwsgi反向代理	fastcgi反向代理	scgi反向代理	http反向代理
指定共享内存	uwsgi_cache	fastcgi_cache	scgi_cache	proxy_cache
缓存文件存放位置	uwsgi_cache_path	fastcgi_cache_path	scgi_cache_path	proxy_cache_path
指定哪些请求不使用缓存	uwsgi_cache_bypass	fastcgi_cache_bypass	scgi_cache_bypass	proxy_cache_bypass
开启子请求更新陈旧缓存	uwsgi_cache_background_update	fastcgi_cache_background_update	scgi_cache_background_update	proxy_cache_background_update
定义缓存关键字	uwsgi_cache_key	fastcgi_cache_key	scgi_cache_key	proxy_cache_key
使用range协议的偏移	uwsgi_cache_max_range_offset	fastcgi_cache_max_range_offset	scgi_cache_max_range_offset	proxy_cache_max_range_offset
缓存哪些请求方法	uwsgi_cache_methods	fastcgi_cache_methods	scgi_cache_methods	proxy_cache_methods
多少请求后再缓存	uwsgi_cache_min_uses	fastcgi_cache_min_uses	scgi_cache_min_uses	proxy_cache_min_uses
缓存哪些响应及时长	uwsgi_cache_valid	fastcgi_cache_valid	scgi_cache_valid	proxy_cache_valid
强制使用range协议	uwsgi_force_ranges	fastcgi_force_ranges	scgi_force_ranges	proxy_force_ranges

七层反向代理对照：缓存类指令（2）

	uwsgi反向代理	fastcgi反向代理	scgi反向代理	http反向代理
有陈旧内容使用304	uwsgi_cache_revalidate	fastcgi_cache_revalidate	scgi_cache_revalidate	proxy_cache_revalidate
返回陈旧的缓存内容	uwsgi_cache_use_stale	fastcgi_cache_use_stale	scgi_cache_use_stale	proxy_cache_use_stale
指定哪些响应不会写入缓存	uwsgi_no_cache	fastcgi_no_cache	scgi_no_cache	proxy_no_cache
将HEAD方法转换为GET方法				proxy_cache_convert_head
加锁减少回源请求	uwsgi_cache_lock	fastcgi_cache_lock	scgi_cache_lock	proxy_cache_lock
回源请求到达该超时时间后再放行	uwsgi_cache_lock_age	fastcgi_cache_lock_age	scgi_cache_lock_age	proxy_cache_lock_age
等待请求的最长等待时间	uwsgi_cache_lock_timeout	fastcgi_cache_lock_timeout	scgi_cache_lock_timeout	proxy_cache_lock_timeout

七层反向代理对照：独有配置

uwsgi反向代理	fastcgi反向代理	scgi反向代理	http反向代理
uwsgi_modifier1			
uwsgi_modifier2			
uwsgi_param	fastcgi_param	scgi_param	
	fastcgi_index		
	fastcgi_catch_stderr		

memcached反向代理

- **功能**：
  - 将HTTP请求转换为memcached协议中的get请求，转发请求至上游memcached服务
  - get命令：get <key>*\r\n
  - 控制命令：<command name> <key> <flags> <exptime> <bytes> [noreply]\r\n
  - 通过设置memcached_key变量构造key键

- **模块**：
  - ngx_http_memcached_module，通过--without-http_memcached_module禁用功能

memcached指令

	memcached反向代理	http反向代理
指定上游	memcached_pass	proxy_pass
连接绑定地址	memcached_bind	proxy_bind
接收响应头部缓存	memcached_buffer_size	proxy_buffer_size
连接上游超时时间	memcached_connect_timeout	proxy_connect_timeout
强制使用range协议	memcached_force_ranges	proxy_force_ranges
针对设置key时的flag，对相应flag添加gzip响应头部	memcached_gzip_flag	
出错时更换上游	memcached_next_upstream	proxy_next_upstream
更换上游超时	memcached_next_upstream_timeout	proxy_next_upstream_timeout
更换上游重试次数	memcached_next_upstream_tries	proxy_next_upstream_tries
读取响应超时时间	memcached_read_timeout	proxy_read_timeout
发送请求超时时间	memcached_send_timeout	proxy_send_timeout
使用TCPkeepalive	memcached_socket_keepalive	proxy_socket_keepalive

websocket反向代理

功能：

由ngx_http_proxy_module模块实现

配置

```
proxy_http_version 1.1;
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection "upgrade";
```

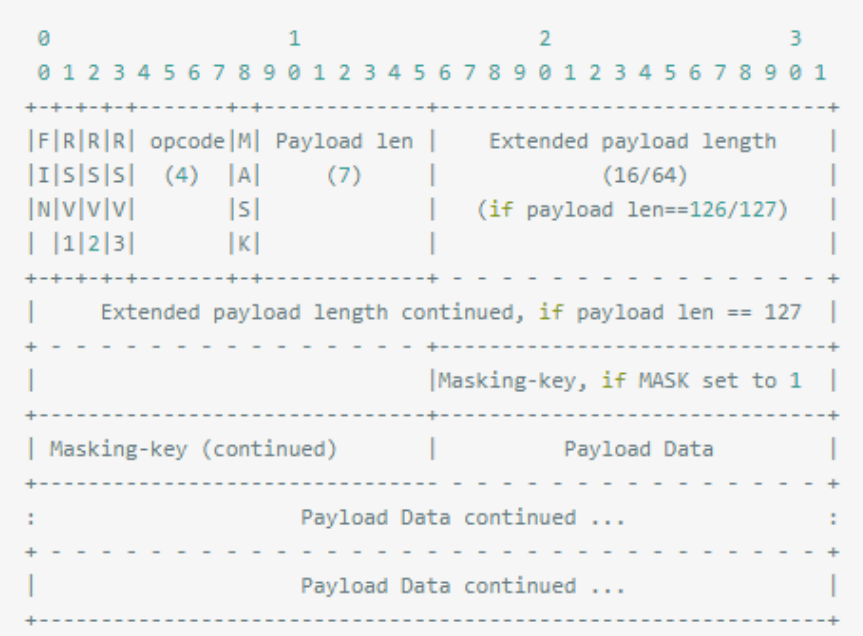
协议升级

GET /?encoding=text HTTP/1.1
Host: websocket.taohui.tech
Accept-Encoding: gzip, deflate
Sec-WebSocket-Version: 13
Origin: http://www.websocket.org
Sec-WebSocket-Extensions: permessage-deflate
Sec-WebSocket-Key:
c3SkgVxVCDhVCp69PJFf3A==
Connection: keep-alive, Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket

HTTP/1.1 101 Web Socket Protocol Handshake
Server: openresty/1.13.6.2
Date: Mon, 10 Dec 2018 08:14:29 GMT
Connection: upgrade
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: content-type
Access-Control-Allow-Headers: authorization
Access-Control-Allow-Headers: x-websocket-extensions
Access-Control-Allow-Headers: x-websocket-version
Access-Control-Allow-Headers: x-websocket-protocol
Access-Control-Allow-Origin: http://www.websocket.org
Sec-WebSocket-Accept:
yA9O5xGLp8SbwCV//OepMPw7pEI=
Upgrade: websocket

websocket协议帧（1）

- FIN，共1位，标记消息是否是最后1帧，1个消息由1个或多个数据帧构成，若消息由1帧构成，起始帧就是结束帧。
- RSV1，RSV2，RSV3，各1位，预留位，用于自定义扩展。如果没有扩展，各位值为0；如果定义了扩展，即为非0值。如果接收的帧中此处为非0，但是扩展中却没有该值的定义，那么关闭连接。
- ***Payload length***：7 bits 或 7 bits + 16 bits 或 7 + 64 bits。若值为
  - 0-125 ：则Payload data的长度即为该值
  - 126 ：那么接下来的2个字节才是Payload Data的长度（unsigned）
  - 127 ：那么接下来的8个字节才是Payload Tada的长度（unsigned）



具体每一bit的意思

```
FIN      1bit 表示信息的最后一帧
RSV 1-3  1bit each 以后备用的 默认都为 0
Opcode   4bit 帧类型，稍后细说
Mask     1bit 掩码，是否加密数据，默认必须置为1
Payload  7bit 数据的长度
Masking-key      1 or 4 bit 掩码
Payload data     (x + y) bytes 数据
Extension data   x bytes  扩展数据
Application data y bytes  程序数据
```

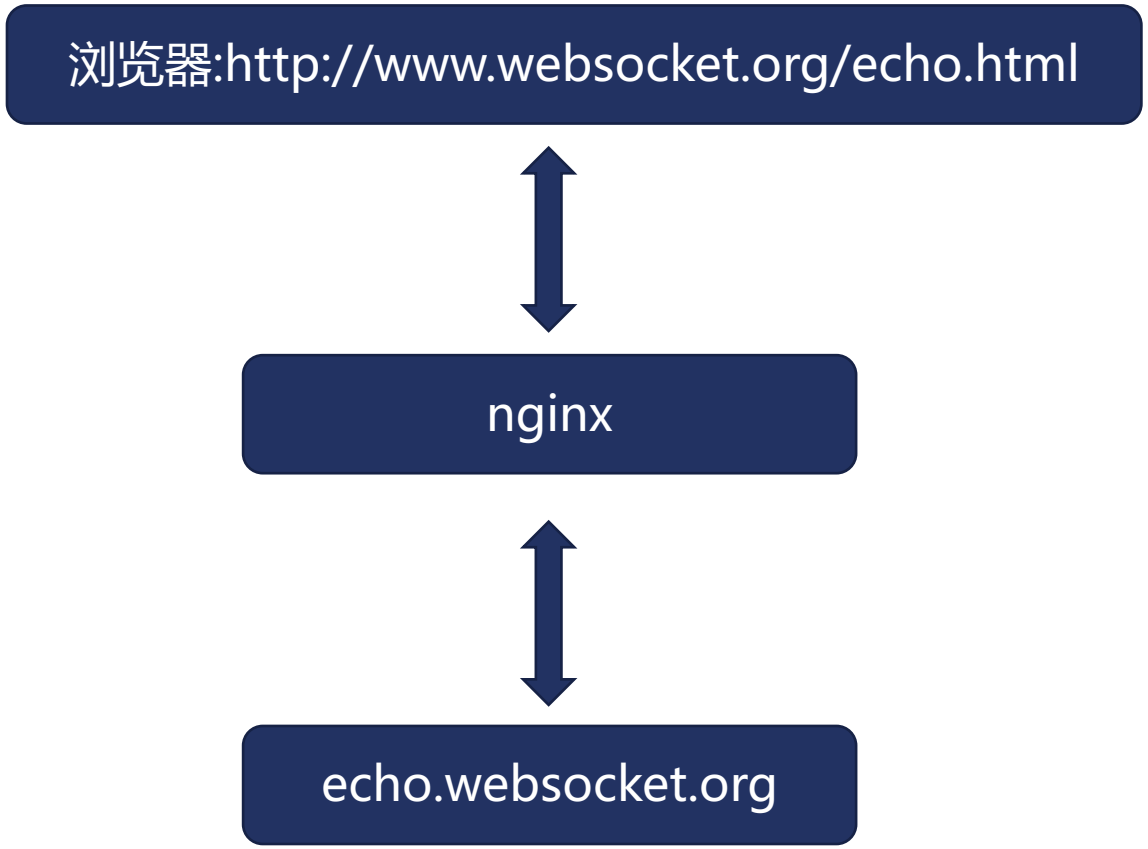
websocket协议帧（2）

- **OPCODE：4位**
  - 解释PayloadData，如果接收到未知的opcode，接收端必须关闭连接。
  - 0x0表示附加数据帧
  - 0x1表示文本数据帧
  - 0x2表示二进制数据帧
  - 0x3-7暂时无定义，为以后的非控制帧保留
  - 0x8表示连接关闭
  - 0x9表示ping ，心跳请求
  - 0xA表示pong ，心跳响应
  - 0xB-F暂时无定义，为以后的控制帧保留
- MASK，共1位，掩码位，表示帧中的数据是否经过加密，客户端发出的数据帧需要经过掩码处理，这个值都是1。如果值是1，那么Masking-key域的数据就是掩码秘钥，用于解码PayloadData，否则Masking-key长度为0。
- Masking-key ，0或者4个字节，当MASK位为1时，4个字节，否则0个字节

websocket协议和扩展

- **数据分片**
  - 有序
- **不支持多路复用**
  - A Multiplexing Extension for WebSockets
- **不支持压缩**
  - Compression Extensions for WebSocket
- **扩展头部**
  - Sec-WebSocket-Version：客户端发送，表示它想使用的WebSocket 协议版本（13表示RFC 6455）。如果服务器不支持这个版本，必须回应自己支持的版本。
  - Sec-WebSocket-Key：客户端发送，自动生成的一个键，作为一个对服务器的“挑战”，以验证服务器支持请求的协议版本；
  - Sec-WebSocket-Accept：服务器响应，包含Sec-WebSocket-Key 的签名值，证明它支持请求的协议版本；
  - Sec-WebSocket-Protocol：用于协商应用子协议：客户端发送支持的协议列表，服务器必须只回应一个协议名；
  - Sec-WebSocket-Extensions：用于协商本次连接要使用的WebSocket 扩展：客户端发送支持的扩展，服务器通过返回相同的首部确认自己支持一或多个扩展。

websocket环境测试

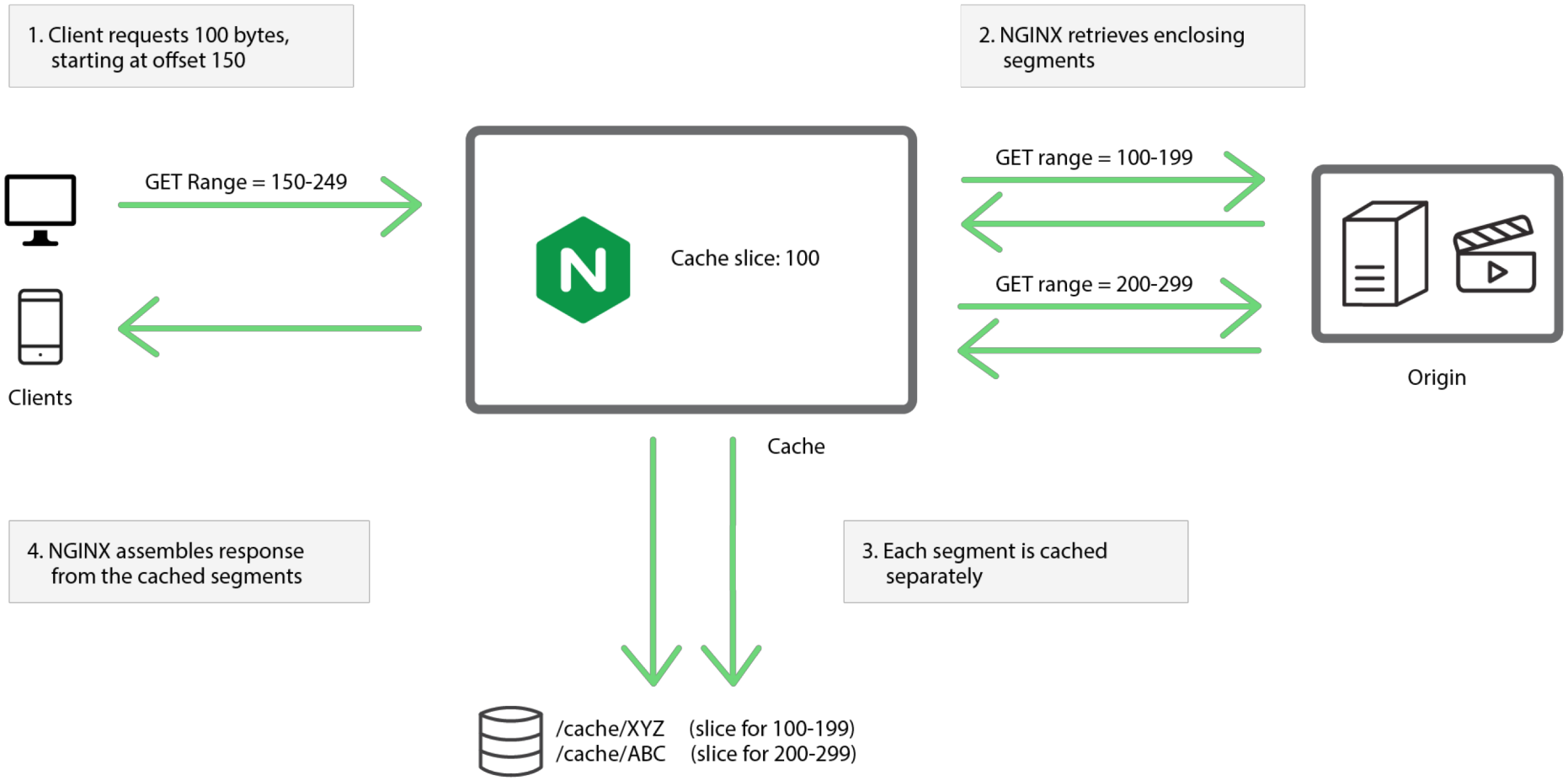


slice模块

Syntax:	**slice** *size*;
Default:	slice 0;
Context:	http, server, location

- **功能**
  - 通过range协议将大文件分解为多个小文件，更好的用缓存为客户端的range协议服务
- **模块**
  - http_slice_module，通过--with-http_slice_module启用功能

slice模块运行流程



open_file_cache

Syntax:	**open_file_cache** off; **open_file_cache** max=*N* [inactive=*time*];
Default:	open_file_cache off;
Context:	http, server, location

缓存哪些元信息？

- 文件句柄
- 文件修改时间
- 文件大小
- 文件查询时的错误信息
- 目录是否存在

```
typedef struct {
    ngx_fd_t                fd;
    ngx_file_uniq_t         uniq;
    time_t                  mtime;
    off_t                   size;
    off_t                   fs_size;
    off_t                   directio;
    size_t                  read_ahead;

    ngx_err_t               err;
    char                   *failed;

    time_t                  valid;

    ngx_uint_t              min_uses;

#if (NGX_HAVE_OPENAT)
    size_t                  disable_symlinks_from;
    unsigned                disable_symlinks:2;
#endif

    unsigned                test_dir:1;
    unsigned                test_only:1;
    unsigned                log:1;
    unsigned                errors:1;
    unsigned                events:1;

    unsigned                is_dir:1;
    unsigned                is_file:1;
    unsigned                is_link:1;
    unsigned                is_exec:1;
    unsigned                is_directio:1;
} ? end {anonngx_open_file_info_t} ?  ngx_open_file_info_t;
```

其他open_file_cache指令

极客时间

Syntax:	**open_file_cache_errors** on | off;
Default:	open_file_cache_errors off;
Context:	http, server, location

Syntax:	open_file_cache_min_uses number;
Default:	open_file_cache_min_uses 1;
Context:	http, server, location

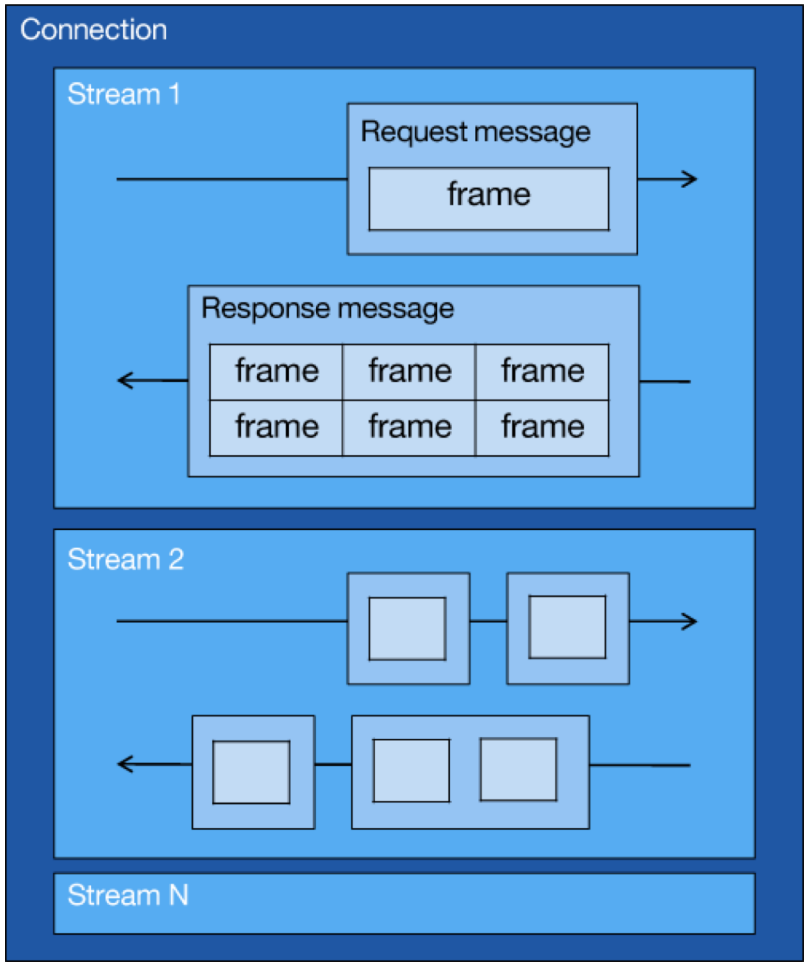
Syntax:	open_file_cache_valid time;
Default:	open_file_cache_valid 60s;
Context:	http, server, location

HTTP2主要特性

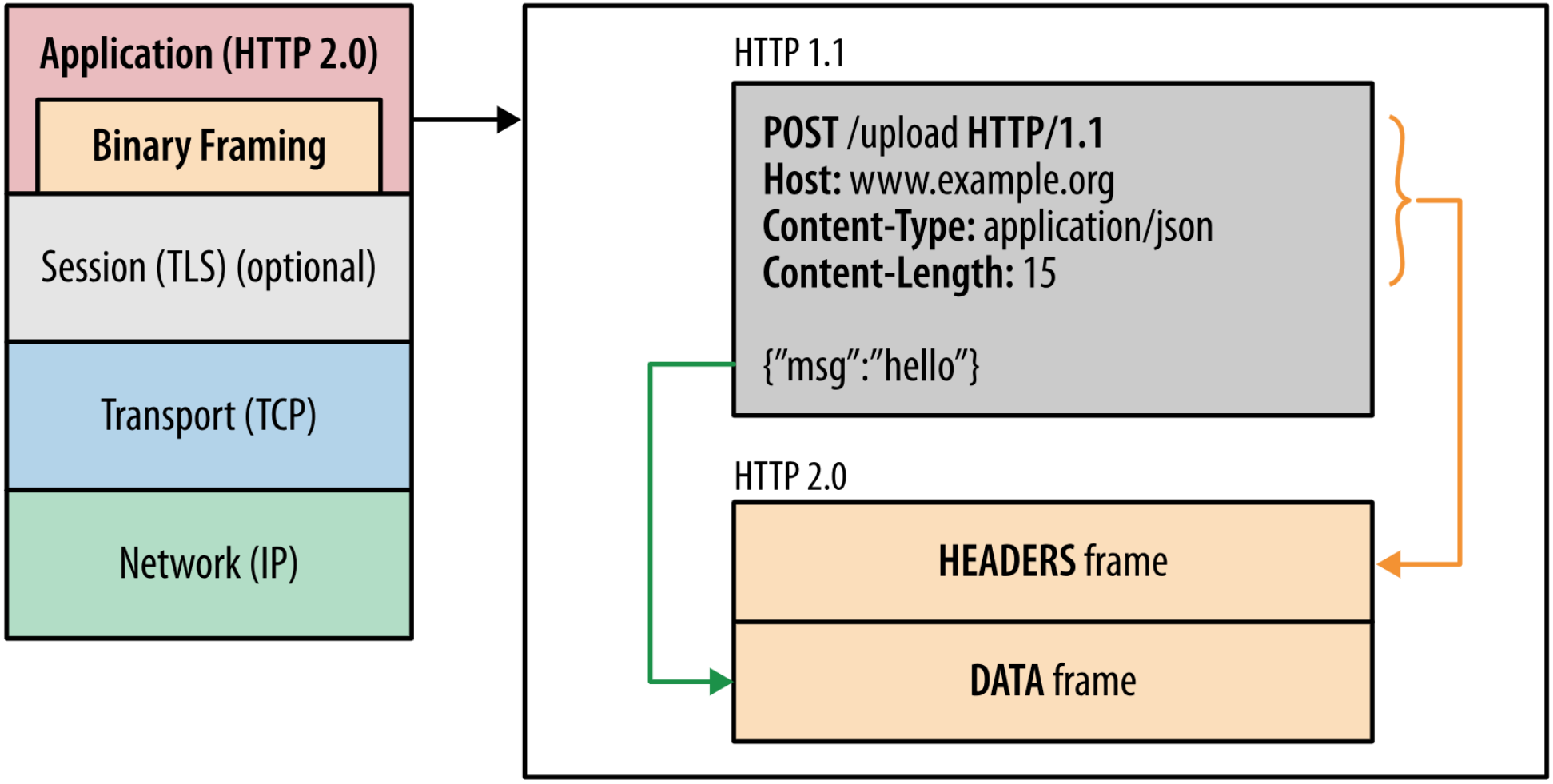
- **传输数据量的大幅减少**
  - 以二进制方式传输
  - 标头压缩
- **多路复用及相关功能**
  - 消息优先级
- **服务器消息推送**
  - 并行推送

HTTP2.0核心概念

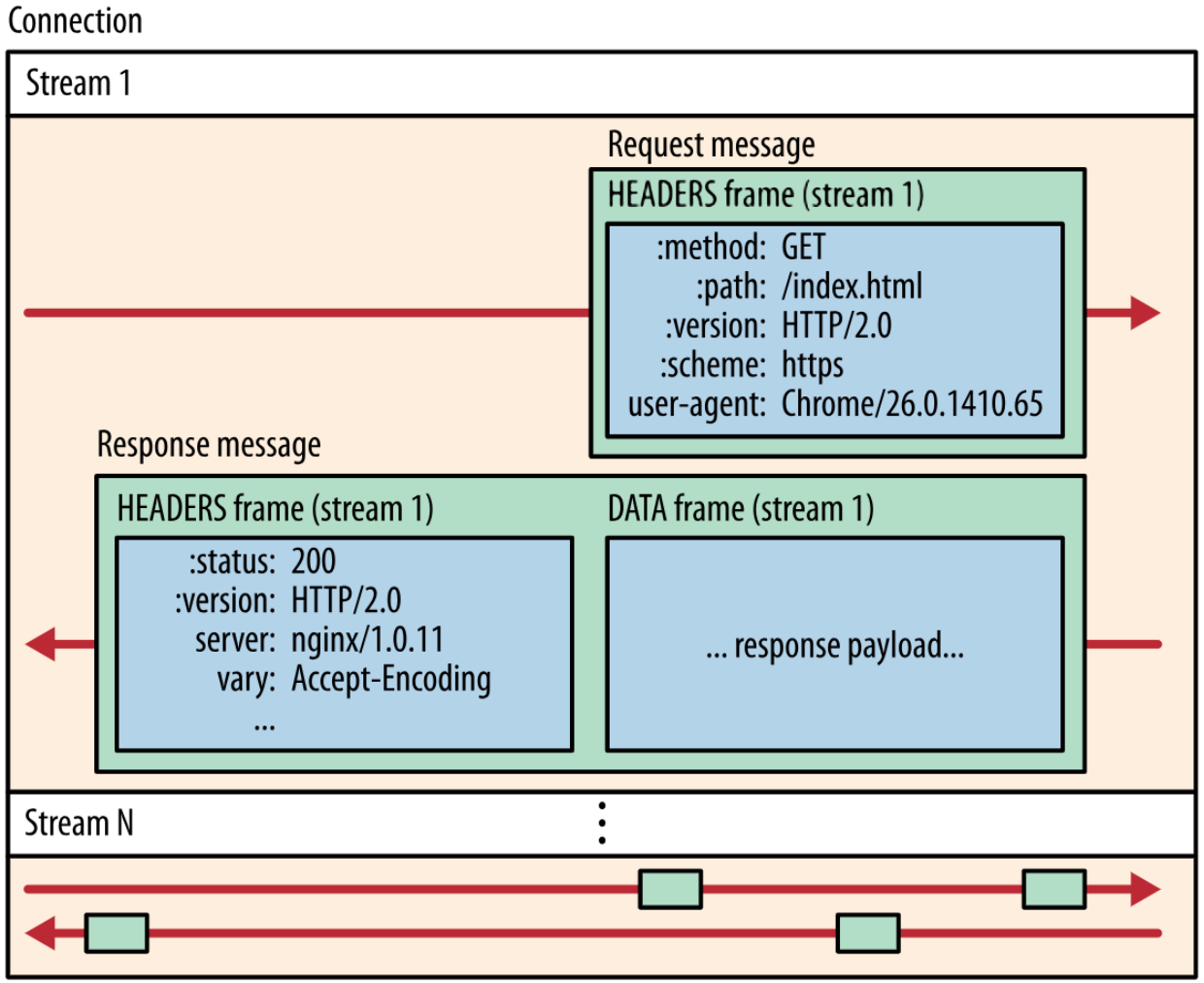
- 连接Connection：1个TCP连接，包含一个或者多个Stream
- 数据流Stream：一个双向通讯数据流，包含多条Message
- 消息Message：对应HTTP1中的请求或者响应，包含一条或者多条Frame
- 数据帧Frame：最小单位，以二进制压缩格式存放HTTP1中的内容



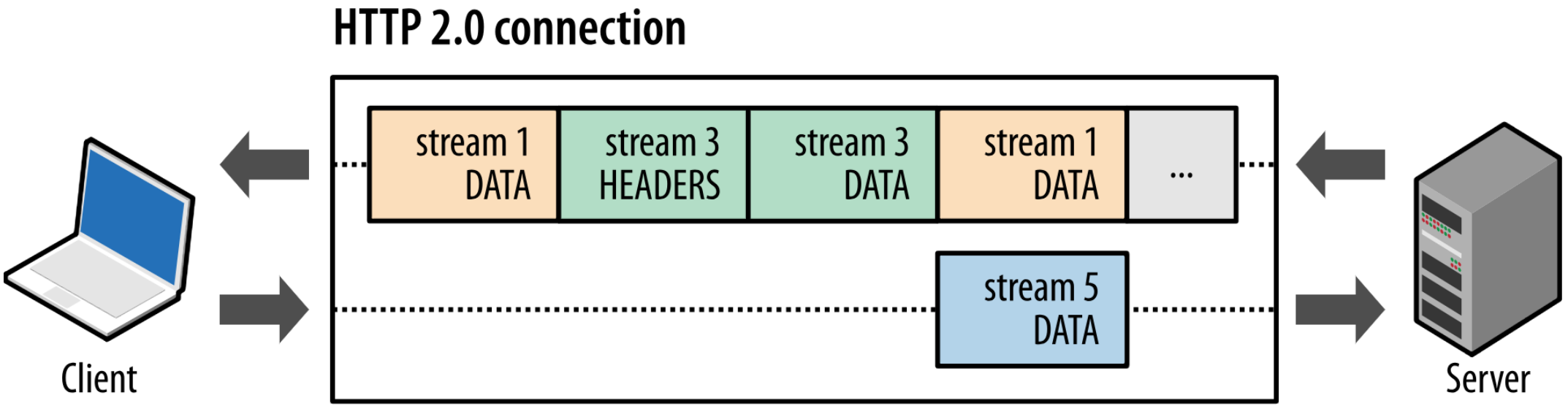
协议分层


Application (HTTP 2.0)
Binary Framing
Session (TLS) (optional)
Transport (TCP)
Network (IP)
HTTP 1.1
POST /upload HTTP/1.1
Host: www.example.org
Content-Type: application/json
Content-Length: 15
{"msg":"hello"}
HTTP 2.0
HEADERS frame
DATA frame

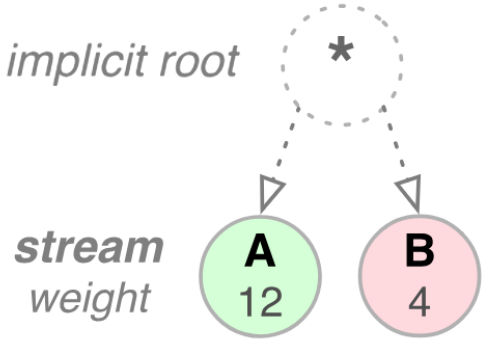
多路复用

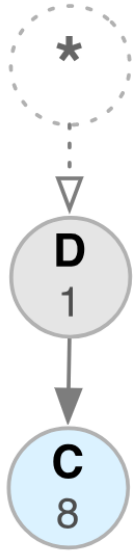

Connection
Stream 1
Request message
HEADERS frame (stream 1)
:method: GET
:path: /index.html
:version: HTTP/2.0
:scheme: https
user-agent: Chrome/26.0.1410.65
Response message
HEADERS frame (stream 1)
:status: 200
:version: HTTP/2.0
server: nginx/1.0.11
vary: Accept-Encoding
...
DATA frame (stream 1)
... response payload...
Stream N

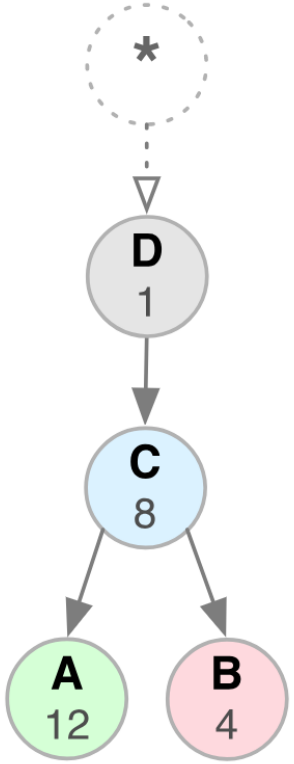
传输中无序，接收时组装

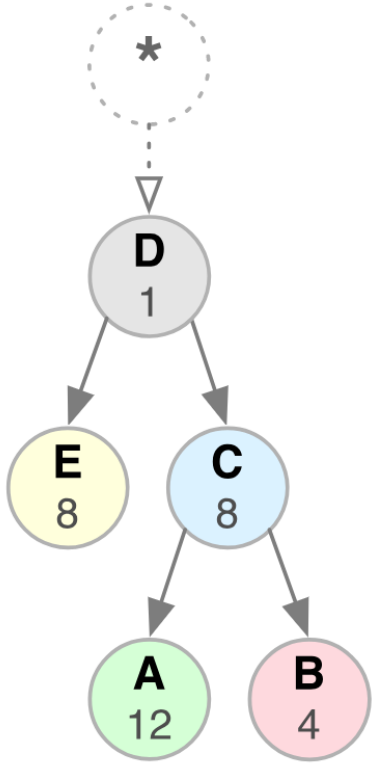

HTTP 2.0 connection
stream 1
DATA
stream 3
HEADERS
stream 3
DATA
stream 1
DATA
...
stream 5
DATA
Client
Server

数据流优先级



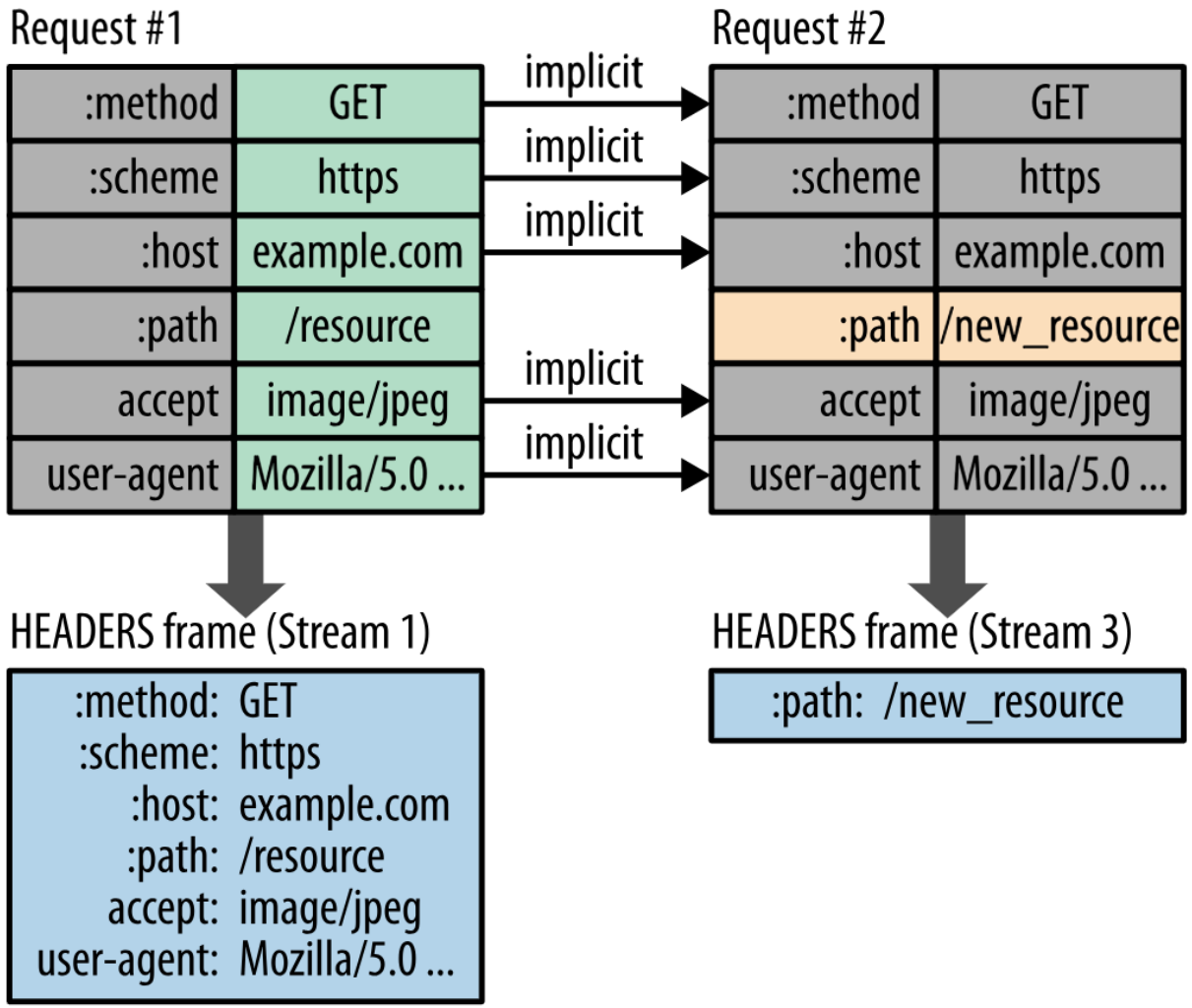




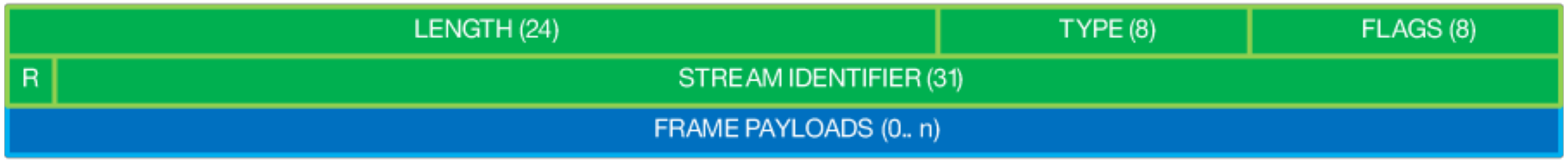


- 每个数据流有优先级（1-256）
- 数据流间可以有依赖关系

标头压缩


Request #1
:method GET
:scheme https
:host example.com
:path /resource
accept image/jpeg
user-agent Mozilla/5.0 ...
implicit
implicit
implicit
implicit
implicit
Request #2
:method GET
:scheme https
:host example.com
:path /new_resource
accept image/jpeg
user-agent Mozilla/5.0 ...
HEADERS frame (Stream 1)
:method: GET
:scheme: https
:host: example.com
:path: /resource
accept: image/jpeg
user-agent: Mozilla/5.0 ...
HEADERS frame (Stream 3)
:path: /new_resource

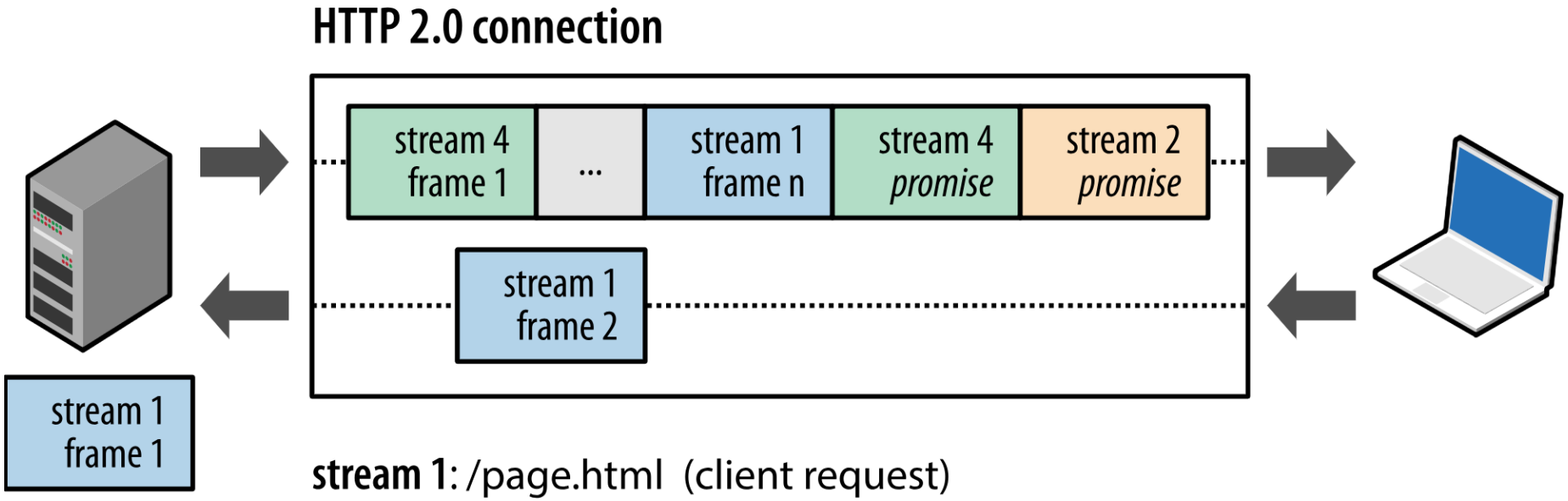
Frame格式



TYPE类型：

- HEADERS：帧仅包含 HTTP 标头信息。
- DATA：帧包含消息的所有或部分有效负载。
- PRIORITY：指定分配给流的重要性。
- RST_STREAM：错误通知：一个推送承诺遭到拒绝。终止流。
- SETTINGS：指定连接配置。
- PUSH_PROMISE：通知一个将资源推送到客户端的意图。
- PING：检测信号和往返时间。
- GOAWAY：停止为当前连接生成流的停止通知。
- WINDOW_UPDATE：用于管理流的流控制。
- CONTINUATION：用于延续某个标头碎片序列。

服务器推送PUSH



stream 1: /page.html (client request)
stream 2: /script.js (push promise)
stream 4: /style.css (push promise)

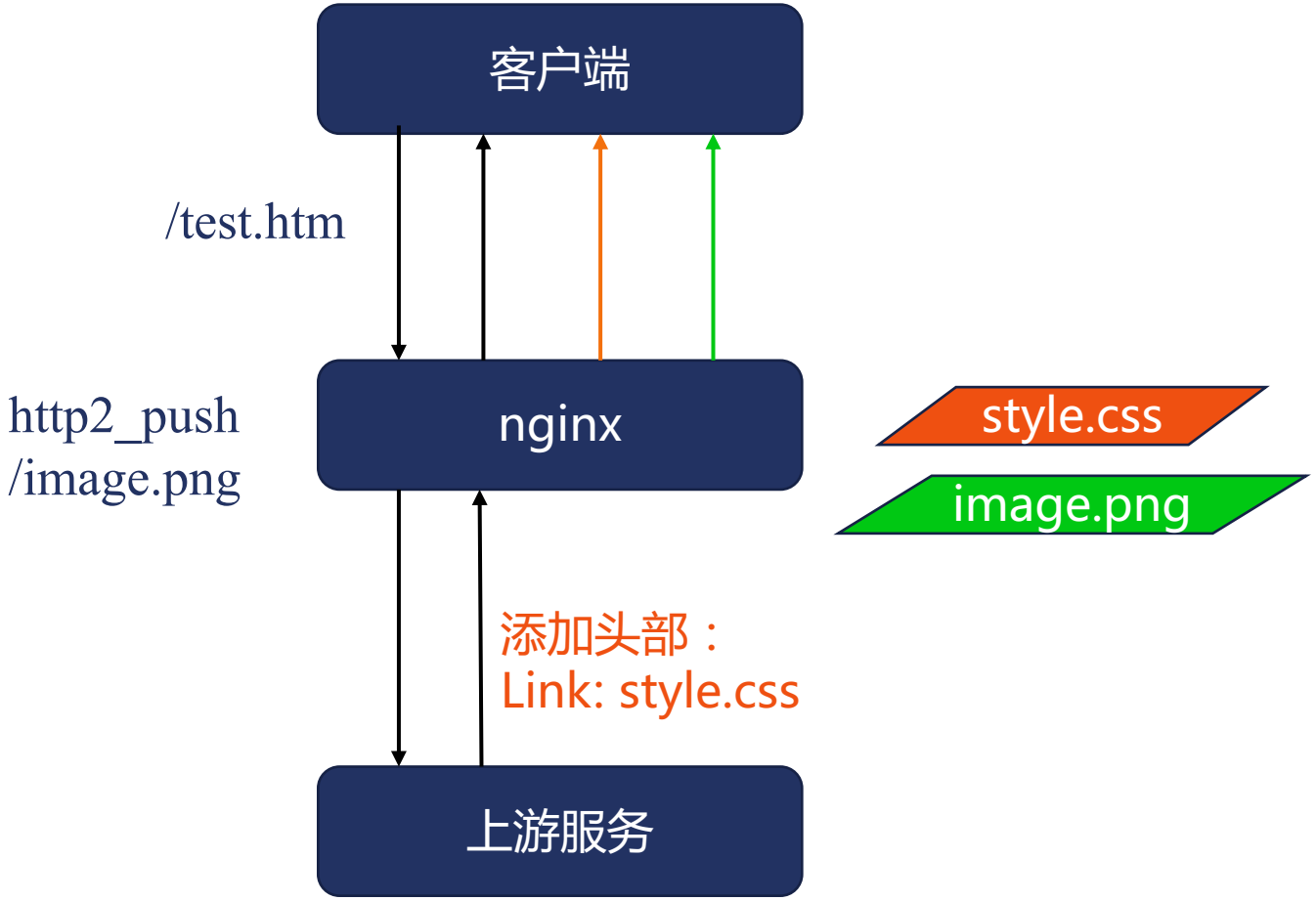
http2

- **模块：**
  - ngx_http_v2_module，通过--with-http_v2_module编译nginx加入http2协议的支持。
- **功能：**
  - 对客户端使用http2协议提供基本功能
- **前提：**
  - 开启TLS/SSL协议
- **使用方法：**
  - listen 443 ssl http2;

nginx推送资源

Syntax:	**http2_push_preload** on | off;
Default:	http2_push_preload off;
Context:	http, server, location

Syntax:	**http2_push** *uri* | off;
Default:	http2_push off;
Context:	http, server, location



测试nginx http2协议的客户端工具

- 在https://github.com/nghttp2/nghttp2/releases下载安装
- centos下使用yum安装：yum install nghttp2

最大并行推送数

Syntax:	**http2_max_concurrent_pushes** *number*;
Default:	http2_max_concurrent_pushes 10;
Context:	http, server

超时控制

Syntax:	**http2_recv_timeout** *time*;
Default:	http2_recv_timeout 30s;
Context:	http, server

Syntax:	**http2_idle_timeout** *time*;
Default:	http2_idle_timeout 3m;
Context:	http, server

并发请求控制

Syntax:	**http2_max_concurrent_pushes** *number*;
Default:	http2_max_concurrent_pushes 10;
Context:	http, server

Syntax:	**http2_max_concurrent_streams** *number*;
Default:	http2_max_concurrent_streams 128;
Context:	http, server

Syntax:	**http2_max_field_size** *size*;
Default:	http2_max_field_size 4k;
Context:	http, server

连接最大处理请求数

Syntax:	**http2_max_requests** *number*;
Default:	http2_max_requests 1000;
Context:	http, server

Syntax:	**http2_chunk_size** *size*;
Default:	http2_chunk_size 8k;
Context:	http, server, location

设置响应包体的分片大小

Syntax:	**http2_chunk_size** *size*;
Default:	http2_chunk_size 8k;
Context:	http, server, location

缓冲区大小设置

极客时间

Syntax:	**http2_recv_buffer_size** *size*;
Default:	http2_recv_buffer_size 256k;
Context:	http

Syntax:	**http2_max_header_size** *size*;
Default:	http2_max_header_size 16k;
Context:	http, server

Syntax:	**http2_body_preread_size** *size*;
Default:	http2_body_preread_size 64k;
Context:	http, server

grpc反向代理

- **grpc协议**
  - https://grpc.io/

- **模块**
  - ngx_http_grpc_module，通过--without-http_grpc_module禁用
  - 依赖ngx_http_v2_module模块

grpc指令对照表

	grpc反向代理	http反向代理
指定上游	grpc_pass	proxy_pass
连接绑定地址	grpc_bind	proxy_bind
接收响应头部缓存	grpc_buffer_size	proxy_buffer_size
连接上游超时时间	grpc_connect_timeout	proxy_connect_timeout
出错时更换上游	grpc_next_upstream	proxy_next_upstream
更换上游超时	grpc_next_upstream_timeout	proxy_next_upstream_timeout
更换上游重试次数	grpc_next_upstream_tries	proxy_next_upstream_tries
读取响应超时时间	grpc_read_timeout	proxy_read_timeout
发送请求超时时间	grpc_send_timeout	proxy_send_timeout
使用TCPkeepalive	grpc_socket_keepalive	proxy_socket_keepalive
减少发向客户端的响应头部	grpc_hide_header	proxy_hide_header
禁用响应头部功能	grpc_ignore_header	proxy_ignore_header
拦截上游错误响应	grpc_intercept_errors	proxy_intercept_errors
传递头部到客户端	grpc_pass_header	proxy_pass_header
增、改请求头部	grpc_set_header	proxy_set_header

grpc指令对照表SSL部分

	grpc反向代理	http反向代理
配置用于上游通讯的证书	grpc_ssl_certificate	proxy_ssl_certificate
配置用于上游通讯的私钥	grpc_ssl_certificate_key	proxy_ssl_certificate_key
指定安全套件	grpc_ssl_ciphers	proxy_ssl_ciphers
指定吊销证书链CRL文件验证上游的证书	grpc_ssl_crl	proxy_ssl_crl
指定域名验证上游证书中域名	grpc_ssl_name	proxy_ssl_name
当私钥有密码时指定密码文件	grpc_ssl_password_file	proxy_ssl_password_file
指定具体某个版本的协议	grpc_ssl_protocols	proxy_ssl_protocols
传递SNI信息至上游	grpc_ssl_server_name	proxy_ssl_server_name
是否重用SSL连接	grpc_ssl_session_reuse	proxy_ssl_session_reuse
验证上游服务的证书	grpc_ssl_trusted_certificate	proxy_ssl_trusted_certificate
是否验证上游服务的证书	grpc_ssl_verify	proxy_ssl_verify
设置验证证书链的深度	grpc_ssl_verify_depth	proxy_ssl_verify_depth

stream模块处理请求的7个阶段

POST_ACCEPT	realip
PREACCESS	limt_conn
ACCESS	access
SSL	ssl
PREREAD	ssl_preread
CONTENT	return, stream_proxy
LOG	access_log

stream中的ssl

Syntax:	**stream** { ... }
Default:	—
Context:	main

Syntax:	**server** { ... }
Default:	—
Context:	stream

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

传输层相关的变量（1）

binary_remote_addr	客户端地址的整型格式，对于IPv4是4字节，对于 IPv6是16字节
connection	递增的连接序号
remote_addr	客户端地址
remote_port	客户端端口
proxy_protocol_addr	若使用了proxy_protocol协议则返回协议中的地址，否则返回空
proxy_protocol_port	若使用了proxy_protocol协议则返回协议中的端口，否则返回空

传输层相关的变量（2）

protocol	传输层协议，值为TCP或者UDP
server_addr	服务器端地址
server_port	服务器端端口

传输层相关的变量（3）

- **bytes_received**
  - 从客户端接收到的字节数
- **bytes_sent**
  - 已经发送到客户端的字节数
- **status**
  - 200：session成功结束
  - 400：客户端数据无法解析，例如proxy_protocol协议的格式不正确
  - 403：访问权限不足被拒绝，例如access模块限制了客户端IP地址
  - 500：服务器内部代码错误
  - 502：无法找到或者连接上游服务
  - 503：上游服务不可用

Nginx 系统变量

time_local	以本地时间标准输出的当前时间，例如14/Nov/2018:15:55:37 +0800
time_iso8601	使用 ISO 8601 标准输出的当前时间，例如2018-11-14T15:55:37+08:00
nginx_version	Nginx 版本号
pid	所属 worker 进程的进程 id
pipe	使用了管道则返回 p，否则返回 .
hostname	所在服务器的主机名，与 hostname 命令输出一致
msec	1970年1月1日到现在的时间，单位为秒，小数点后精确到毫秒

content阶段：return模块

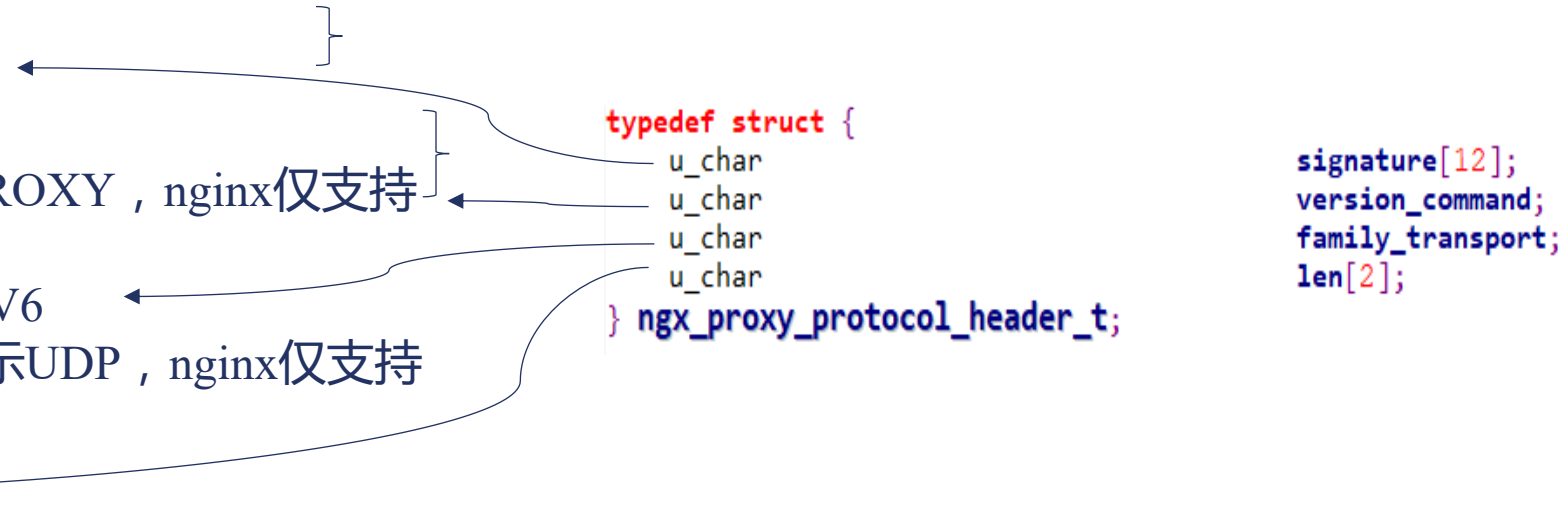
极客时间

Syntax:	**return** *value*;
Default:	—
Context:	server

proxy_protocol协议

- **v1协议**
  - PROXY TCP4 202.112.144.236 10.210.12.10 5678 80\r\n
  - PROXY TCP6 2001:da8:205::100 2400:89c0:2110:1::21 6324 80\r\n
  - PROXY UKNOWN\r\n

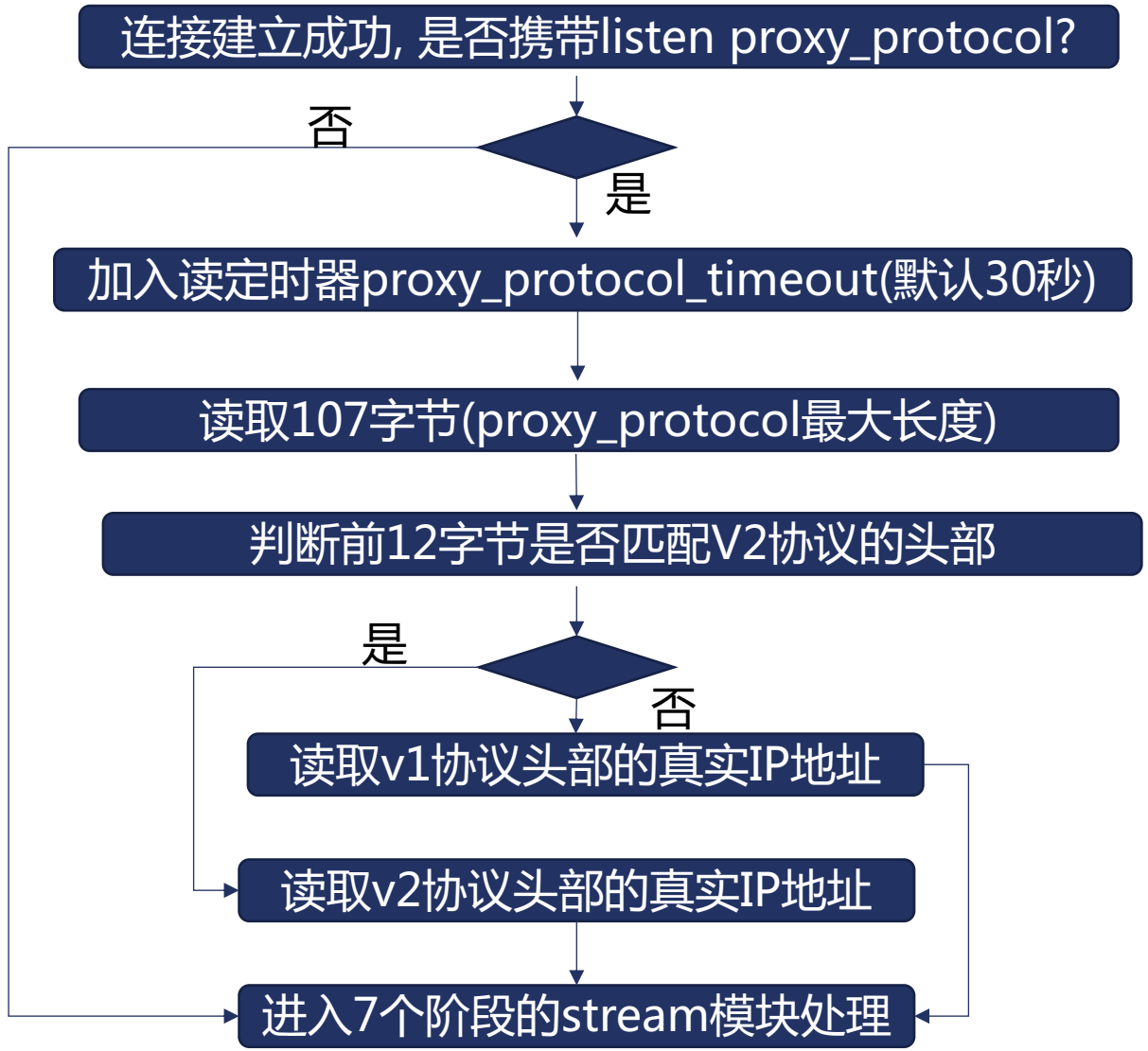
- **v2协议**
  - 12字节签名：\r\n\r\n\0\r\nQUIT\n
  - 4位协议版本号：2
  - 4位命令：0表示LOCAL，1表示PROXY，nginx仅支持PROXY
  - 4位地址族：1表示IPV4，2表示IPV6
  - 4位传输层协议：1表示TCP，2表示UDP，nginx仅支持TCP协议
  - 2字节地址长度



读取proxy_protocol协议的超时控制

Syntax:	**proxy_protocol_timeout** timeout;
Default:	proxy_protocol_timeout 30s;
Context:	stream, server

stream处理proxy_protocol流程



post_accept阶段：realip模块

功能：

通过proxy_protocol协议取出客户端真实地址，并写入remote_addr及remote_port变量。同时使用realip_remote_addr和realip_remote_port保留TCP连接中获得的原始地址。

模块：

ngx_stream_realip_module ，通过--with-stream_realip_module启用功能

Syntax:	**set_real_ip_from** *address* | *CIDR* | unix:;
Default:	—
Context:	stream, server

PREACCESS阶段的limit_conn模块

功能：

限制客户端的并发连接数。使用变量自定义限制依据，基于共享内存所有worker进程同时生效。

模块：

ngx_stream_limit_conn_module，通过--without-stream_limit_conn_module禁用模块

limit_conn模块的指令

Syntax:	**limit_conn_zone** key zone=name:size;
Default:	—
Context:	stream

Syntax:	**limit_conn** zone number;
Default:	—
Context:	stream, server

Syntax:	**limit_conn_log_level** info | notice | warn | error;
Default:	limit_conn_log_level error;
Context:	stream, server

ACCESS阶段的access模块

功能：

根据客户端地址（realip模块可以修改地址）决定连接的访问权限

模块：

ngx_stream_access_module，通过--without-stream_access_module禁用模块

access模块的指令

Syntax:	**allow** address | CIDR | unix: | all;
Default:	—
Context:	stream, server

Syntax:	**deny** address | CIDR | unix: | all;
Default:	—
Context:	stream, server

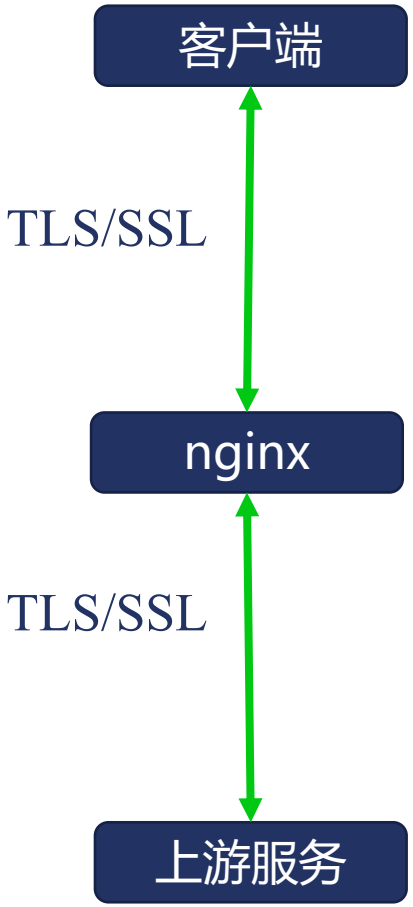
log阶段：stream_log模块

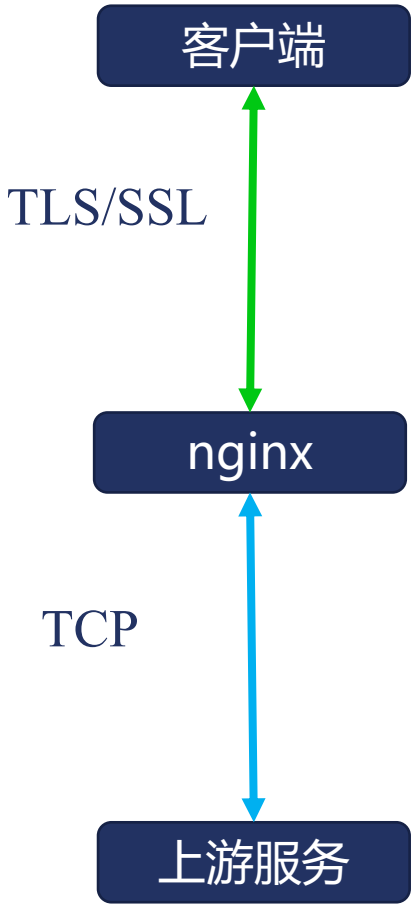
Syntax:	**access_log** path *format* [buffer=*size*] [gzip[=*level*]] [flush=*time*] [if=*condition*]; **access_log** off;
Default:	access_log off;
Context:	stream, server

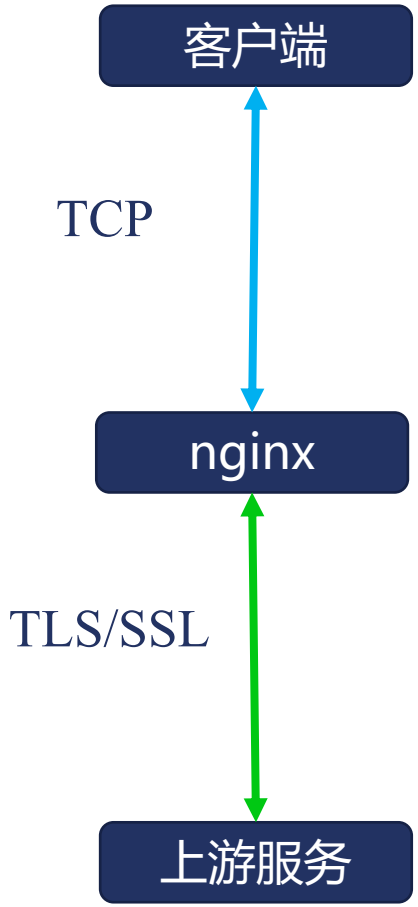
Syntax:	**log_format** name [escape=default|json|none] *string* ...;
Default:	—
Context:	stream

Syntax:	**open_log_file_cache** max=*N* [inactive=*time*] [min_uses=*N*] [valid=*time*]; **open_log_file_cache** off;
Default:	open_log_file_cache off;
Context:	stream, server

stream模块TLS/SSL应用场景







stream中的ssl

功能：

使stream反向代理对下游支持TLS/SSL协议

模块：

ngx_stream_ssl_module，默认不编译进nginx，通过--with-stream_ssl_module加入

stream ssl指令对比http模块：配置基本参数

	stream ssl	http ssl
配置服务器证书	ssl_certificate	ssl_certificate
配置服务器证书私钥	ssl_certificate_key	ssl_certificate_key
指定安全套件	ssl_ciphers	ssl_ciphers
指定吊销证书链CRL	ssl_crl	ssl_crl
密码交换DH算法参数	ssl_dhparam	ssl_dhparam
指定密码交换时使用哪条椭圆曲线	ssl_ecdh_curve	ssl_ecdh_curve
TLS握手的超时时间	ssl_handshake_timeout	
指定私钥的密码文件	ssl_password_file	ssl_password_file
是否启用服务器偏好的安全套件	ssl_prefer_server_ciphers	ssl_prefer_server_ciphers
指定支持的TLS协议	ssl_protocols	ssl_protocols

stream ssl指令对比http模块：提升性能

	stream ssl	http ssl
使用session缓存提升性能	ssl_session_cache	ssl_session_cache
指定使用ticket票据时的加解密文件，默认使用随机字串	ssl_session_ticket_key	ssl_session_ticket_key
是否启用ticket票据	ssl_session_tickets	ssl_session_tickets
指定session重用的超时时间	ssl_session_timeout	ssl_session_timeout

stream ssl指令对比http模块：验证客户端证书

	stream ssl	http ssl
是否验证客户端的证书	ssl_verify_client	ssl_verify_client
可信CA证书，用于验证客户端证书	ssl_client_certificate	ssl_client_certificate
验证客户端证书是否可信	ssl_trusted_certificate	ssl_trusted_certificate
验证客户端证书链的深度	ssl_verify_depth	ssl_verify_depth

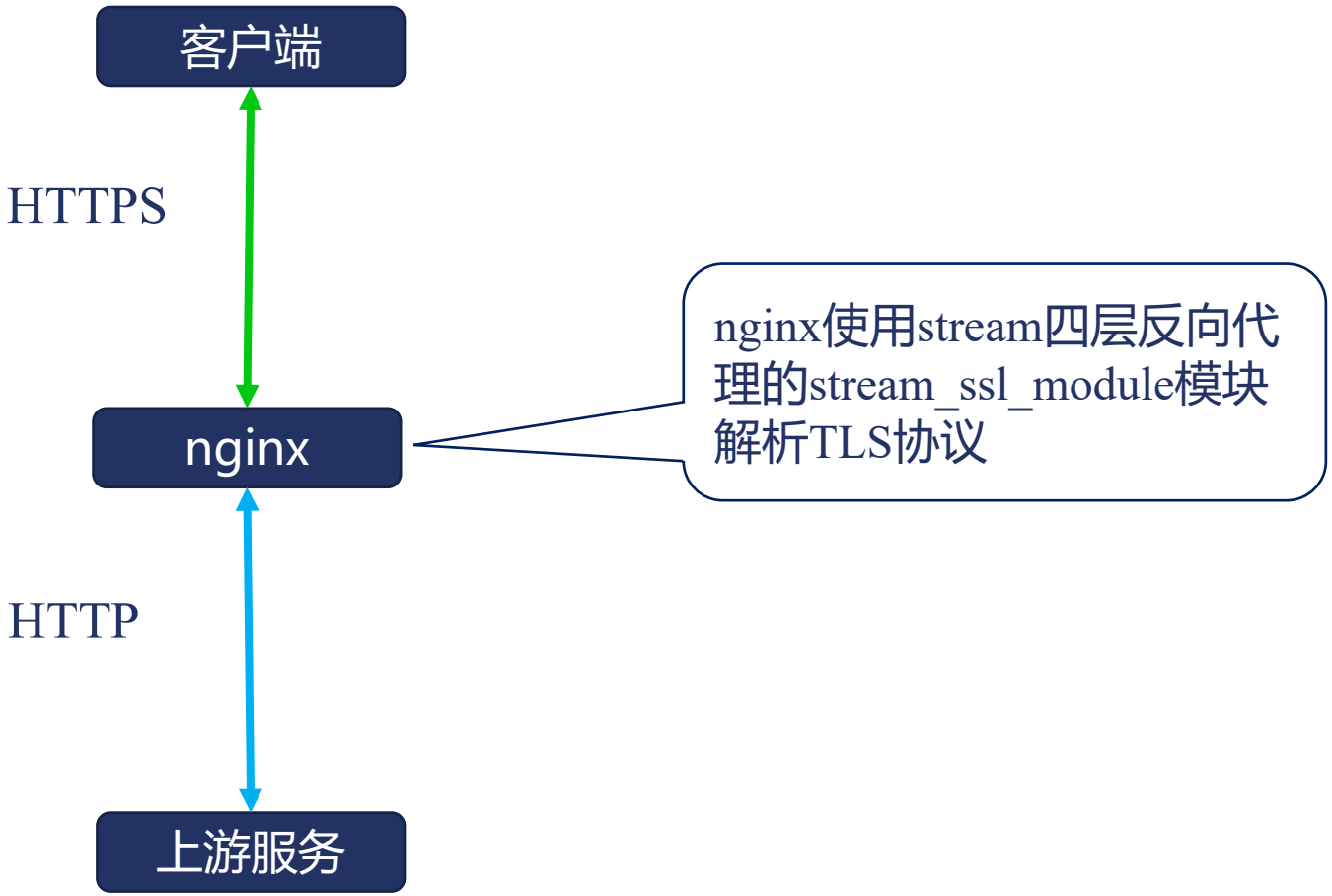
stream ssl模块提供的变量（1）

- **安全套件**
  - ssl_cipher: 本次通讯选用的安全套件，例如ECDHE-RSA-AES128-GCM-SHA256
  - ssl_ciphers: 客户端支持的所有安全套件
  - ssl_protocol: 本次通讯选用的TLS版本，例如TLSv1.2
  - ssl_curves: 客户端支持的椭圆曲线，例如secp384r1:secp521r1
- **证书**
  - ssl_client_raw_cert: 原始客户端证书内容
  - ssl_client_escaped_cert: 返回客户端证书做urlencode编码后的内容
  - ssl_client_cert: 对客户端证书每一行内容前加tab制表符空白，增强可读性。
  - ssl_client_fingerprint: 客户端证书的SHA1指纹

stream ssl模块提供的变量（2）

- **证书结构化信息**
  - ssl_server_name: 通过TLS插件SNI(Server Name Indication)获取到的服务域名
  - ssl_client_i_dn:依据RFC2253获取到证书issuer dn信息，例如： CN=…,O=…,L=…,C=…
  - ~~ssl_client_i_dn_legacy:依据RFC2253获取到证书issuer dn信息，例如： /C=…/L=…/O=…/CN=…~~
  - ssl_client_s_dn: 依据RFC2253获取到证书subject dn信息，例如：CN=…,OU=…,O=…,L=…,ST=…,C=…
  - ~~ssl_client_s_dn_legacy: 同样获取subject dn信息，格式为：/C=…/ST=…/L=…/O=…/OU=…/CN=…~~
- **证书有效期**
  - ssl_client_v_end: 返回客户端证书的过期时间，例如Dec 1 11:56:11 2028 GMT
  - ssl_client_v_remain: 返回还有多少天客户端证书过期，例如针对上面的ssl_client_v_end其值为3649
  - ssl_client_v_start: 客户端证书的颁发日期，例如Dec 4 11:56:11 2018 GMT
- **连接有效性**
  - ssl_client_serial: 返回连接上客户端证书的序列号，例如8BE947674841BD44
  - ~~ssl_early_data: 在TLS1.3协议中使用了early data且握手未完成返回1，否则返回空字符串~~
  - ssl_client_verify: 如果验证失败为FAILED:原因，如果没有验证证书则为NONE，验证成功则为SUCCESS
  - ssl_session_id: 已建立连接的sessionid
  - ssl_session_reused: 如果session被复用（参考session缓存）则为r，否则为.

STREAM SSL模块实战



SSL_PREREAD模块

- **模块**
  - stream_ssl_preread_module，使用--with-stream_ssl_preread_module启用模块
- **功能**
  - 解析下游TLS证书中信息，以变量方式赋能其他模块
- **提供变量**
  - $ssl_preread_protocol
    - 客户端支持的TLS版本中最高的版本，例如TLSv1.3
  - $ssl_preread_server_name
    - 从SNI中获取到的服务器域名
  - $ssl_preread_alpn_protocols
    - 通过ALPN中获取到的客户端建议使用的协议，例如h2,http/1.1

stream模块处理请求的7个阶段

POST_ACCEPT	realip
PREACCESS	limt_conn
ACCESS	access
SSL	ssl
PREREAD	ssl_preread
CONTENT	return, stream_proxy
LOG	access_log

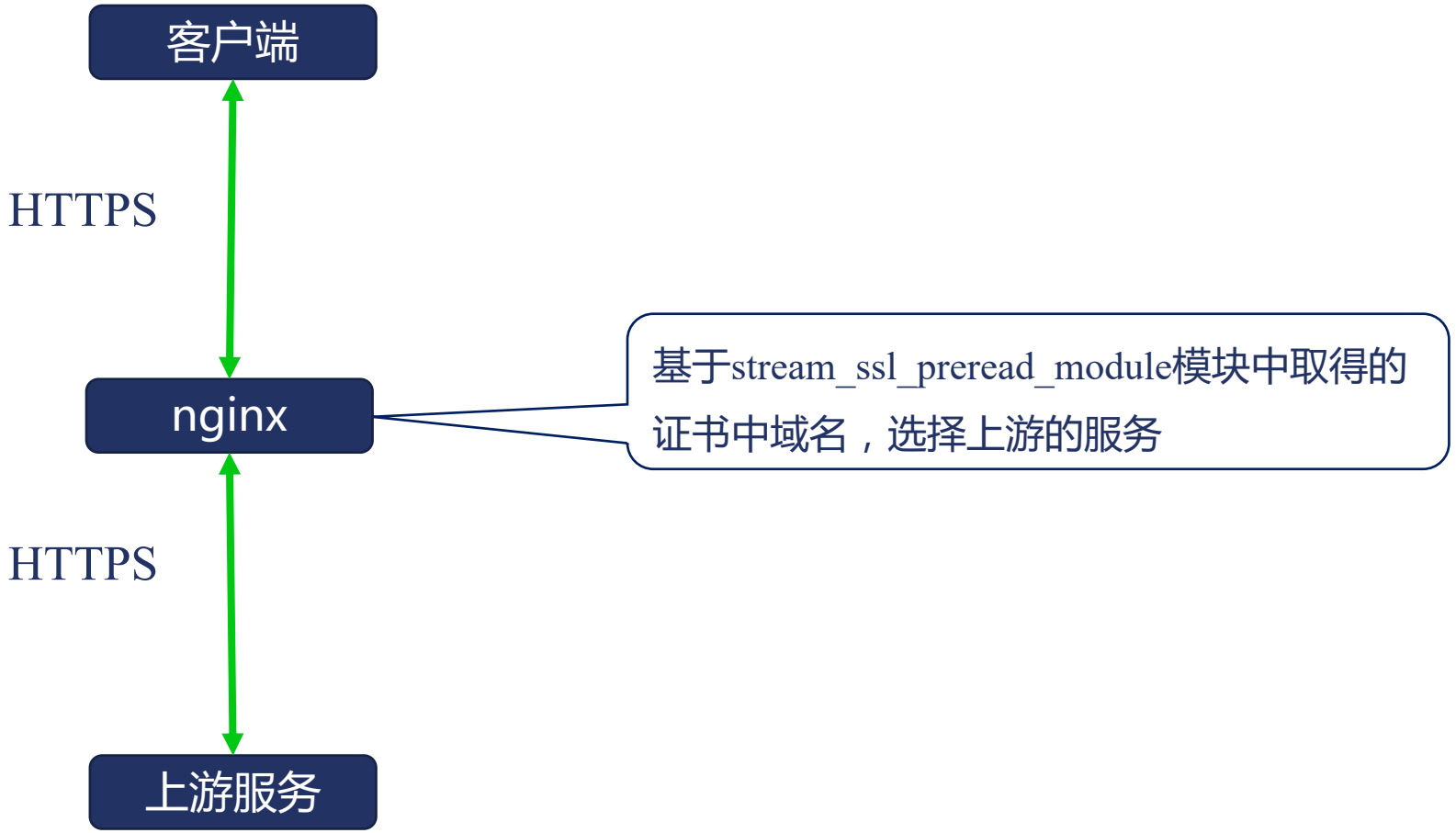
preread阶段：ssl_preread模块

Syntax:	**preread_buffer_size** *size*;
Default:	preread_buffer_size 16k;
Context:	stream, server

Syntax:	**preread_timeout** *timeout*;
Default:	preread_timeout 30s;
Context:	stream, server

Syntax:	**ssl_preread** on | off;
Default:	ssl_preread off;
Context:	stream, server

STREAM SSL_PREREAD模块实战



反向代理stream_proxy模块

- **模块**
  - ngx_stream_proxy_module，默认在Nginx中
- **功能**
  - 提供TCP/UDP协议的反向代理
  - 支持与上游的连接使用TLS/SSL协议
  - 支持与上游的连接使用proxy protocol协议

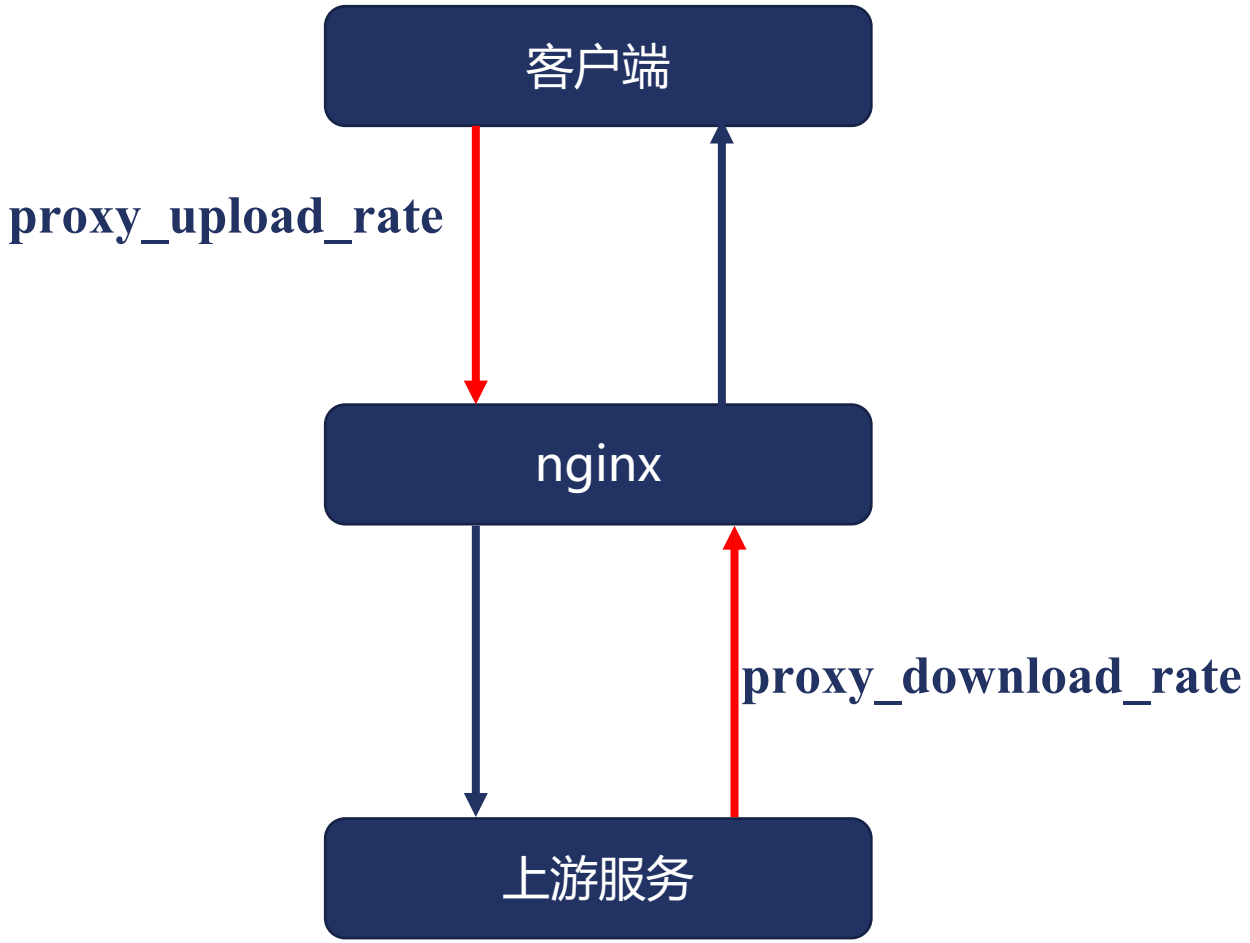
proxy模块对上下游的限速指令

限制读取上游服务数据的速度

Syntax:	**proxy_download_rate** rate;
Default:	proxy_download_rate 0;
Context:	stream, server

限制读取客户端数据的速度

Syntax:	**proxy_upload_rate** rate;
Default:	proxy_upload_rate 0;
Context:	stream, server



stream反向代理指令

	stream反向代理	http反向代理
指定上游	proxy_pass	proxy_pass
连接绑定地址	proxy_bind	proxy_bind
该值同时用于定义接收上游数据、接收下游数据的缓冲区大小	proxy_buffer_size	proxy_buffer_size
连接上游超时时间	proxy_connect_timeout	proxy_connect_timeout
TCP连接使用proxy_protocol协议	proxy_protocol	
出错时更换上游	proxy_next_upstream	proxy_next_upstream
更换上游超时	proxy_next_upstream_timeout	proxy_next_upstream_timeout
更换上游重试次数	proxy_next_upstream_tries	proxy_next_upstream_tries
读取响应超时时间	proxy_timeout	proxy_read_timeout
发送请求超时时间	proxy_timeout	proxy_send_timeout
使用TCPkeepalive	proxy_socket_keepalive	proxy_socket_keepalive

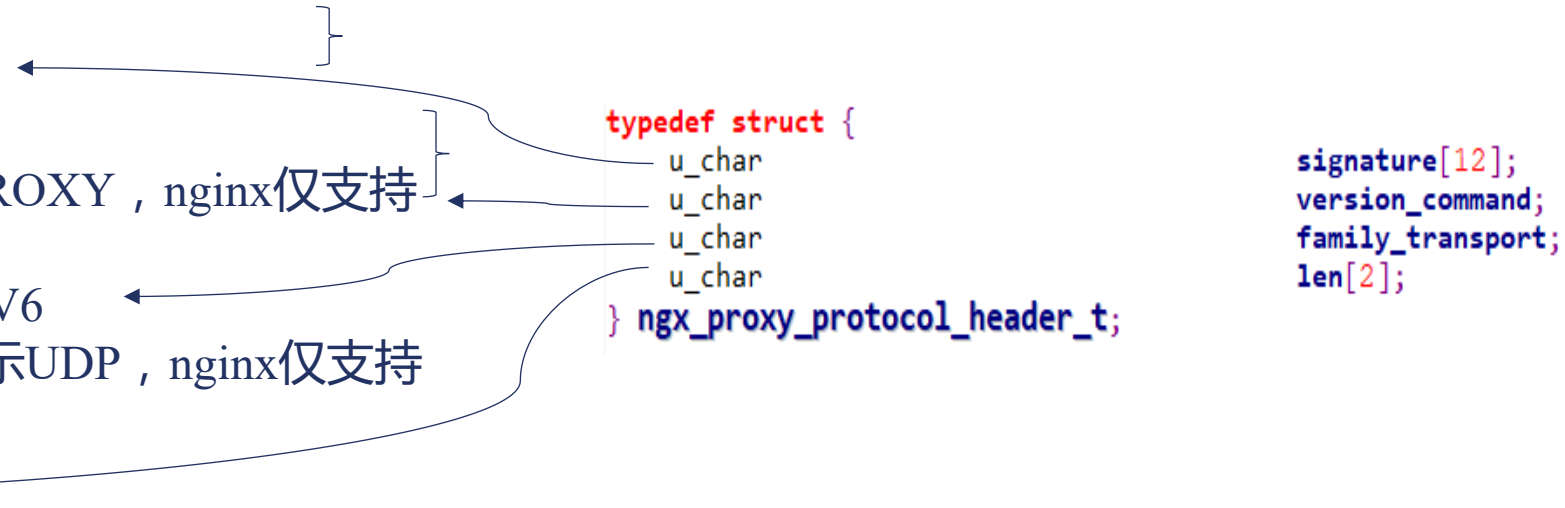
stream ssl指令与http proxy模块对照表

	stream	http反向代理
是否对上游使用ssl	proxy_ssl	
配置服务器证书	proxy_ssl_certificate	proxy_ssl_certificate
配置服务器证书私钥	proxy_ssl_certificate_key	proxy_ssl_certificate_key
指定安全套件	proxy_ssl_ciphers	proxy_ssl_ciphers
指定吊销证书链CRL	proxy_ssl_crl	proxy_ssl_crl
指定域名验证上游证书中域名	proxy_ssl_name	proxy_ssl_name
当私钥有密码时指定密码文件	proxy_ssl_password_file	proxy_ssl_password_file
指定具体某个版本的协议	proxy_ssl_protocols	proxy_ssl_protocols
传递SNI信息至上游	proxy_ssl_server_name	proxy_ssl_server_name
是否重用SSL连接	proxy_ssl_session_reuse	proxy_ssl_session_reuse
验证上游服务的证书	proxy_ssl_trusted_certificate	proxy_ssl_trusted_certificate
是否验证上游服务的证书	proxy_ssl_verify	proxy_ssl_verify
设置验证证书链的深度	proxy_ssl_verify_depth	proxy_ssl_verify_depth

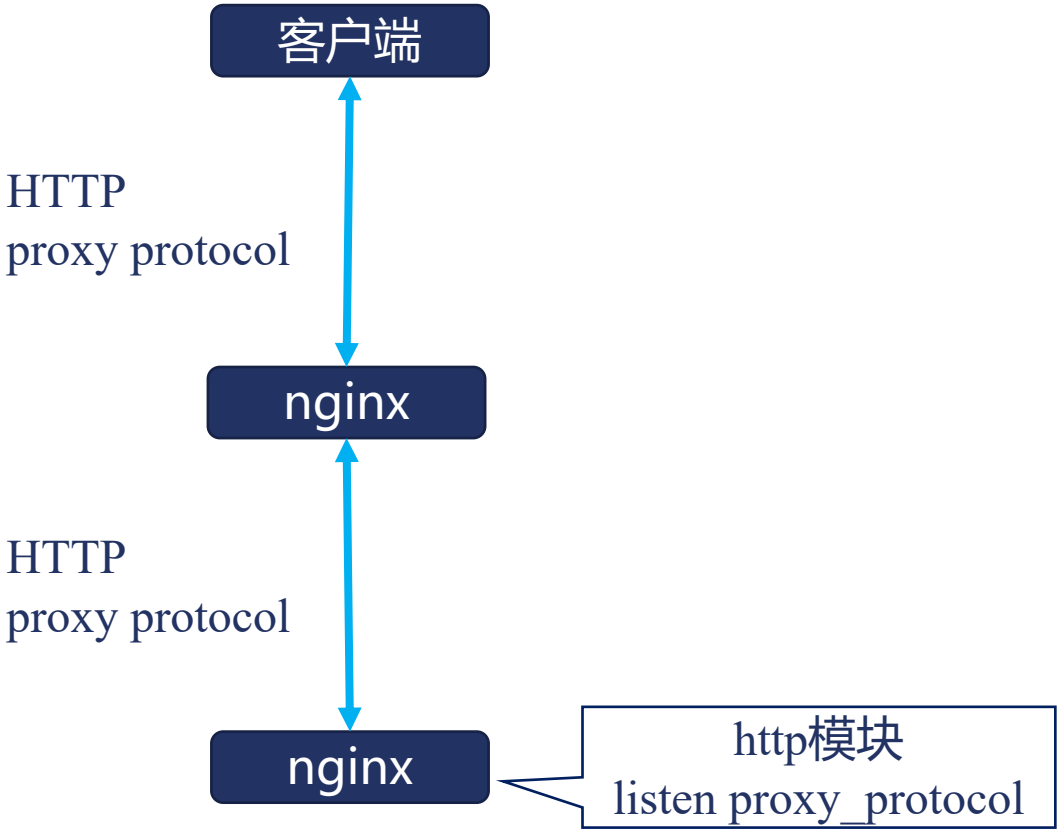
proxy_protocol协议

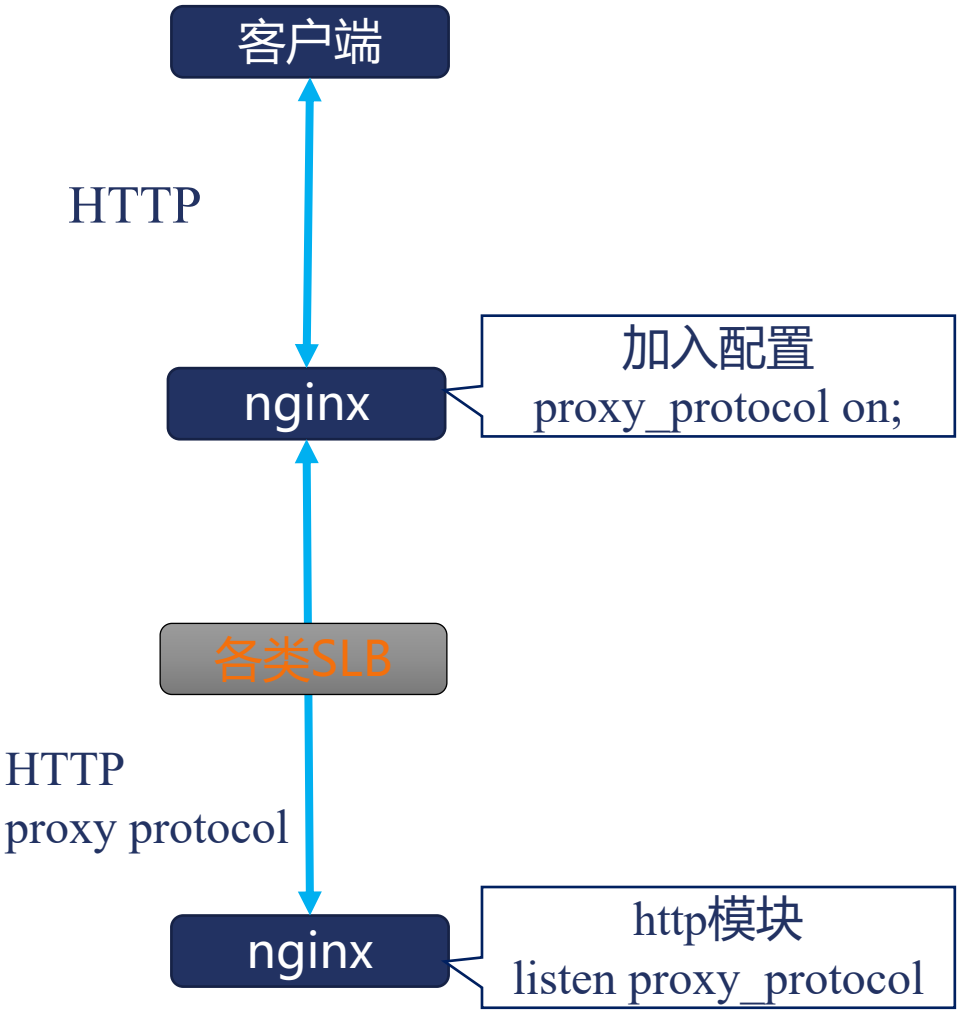
- **v1协议**
  - PROXY TCP4 202.112.144.236 10.210.12.10 5678 80\r\n
  - PROXY TCP6 2001:da8:205::100 2400:89c0:2110:1::21 6324 80\r\n
  - PROXY UKNOWN\r\n

- **v2协议**
  - 12字节签名：\r\n\r\n\0\r\nQUIT\n
  - 4位协议版本号：2
  - 4位命令：0表示LOCAL，1表示PROXY，nginx仅支持PROXY
  - 4位地址族：1表示IPV4，2表示IPV6
  - 4位传输层协议：1表示TCP，2表示UDP，nginx仅支持TCP协议
  - 2字节地址长度



stream proxy模块实战





UDP反向代理的理论依据

UDP反向代理

- 指定一次会话session中最多从客户端接收到多少报文就结束session。（1.15.7非稳定版本）
  - 仅会话结束时才会记录access日志
  - 同一个会话中，nginx使用同一端口连接上游服务
  - 设置为0表示不限制，每次请求都会记录access日志

Syntax:	**proxy_requests** number;
Default:	proxy_requests 0;
Context:	stream, server

- 指定对应一个请求报文，上游应返回多少个响应报文。
  - 与proxy_timeout结合使用，控制上游服务是否不可用

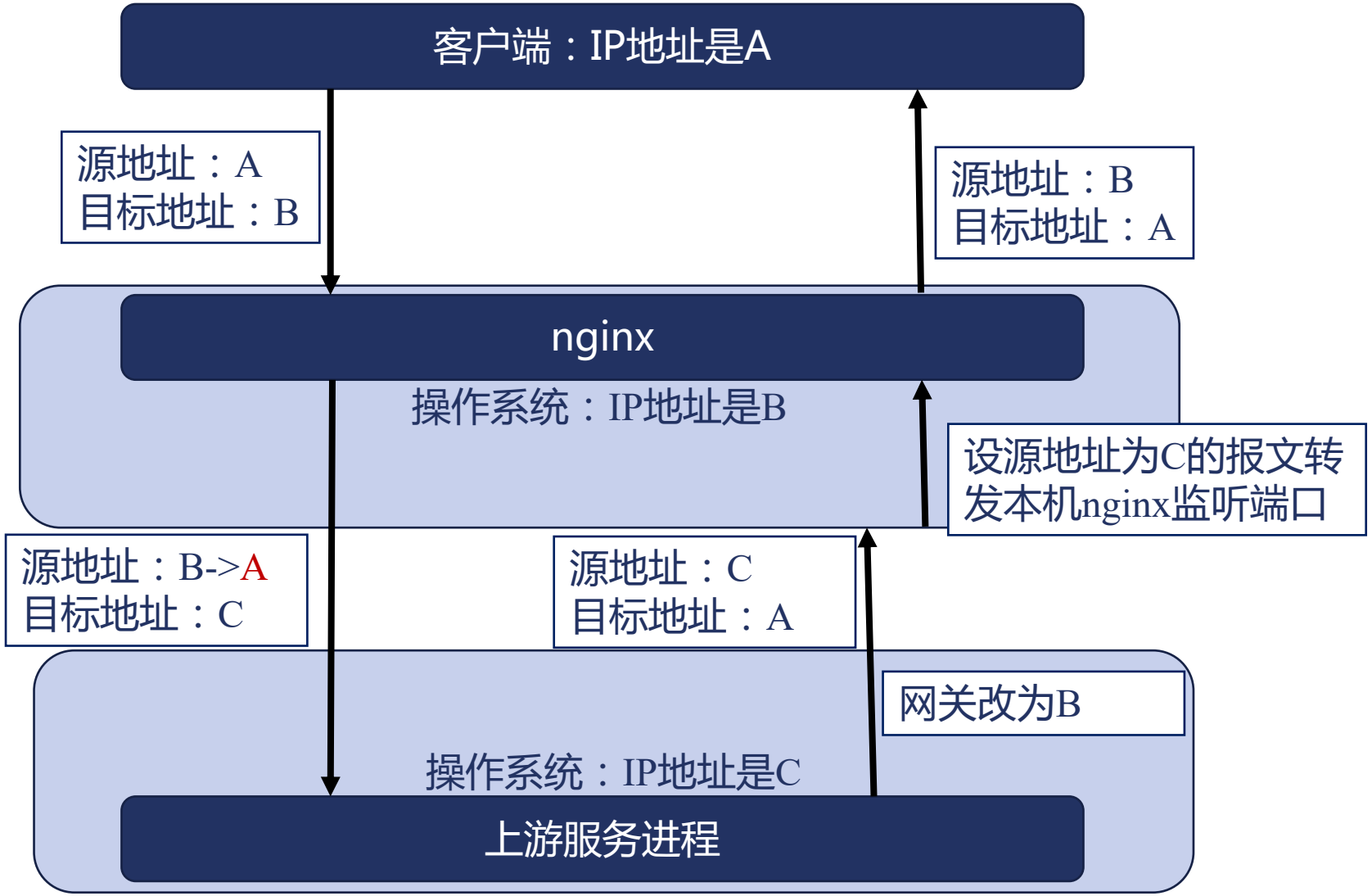
Syntax:	**proxy_responses** number;
Default:	—
Context:	stream, server

透传IP地址

- **proxy_protocol协议**

- **修改IP报文**
  - **步骤**
    - 修改IP报文中的源地址
    - 修改路由规则
  - **方案**
    - IP地址透传：经由nginx转发上游返回的报文给客户端（TCP/UDP）
    - DSR：上游直接发送报文给客户端（仅UDP）

IP地址透传示意图


客户端：IP地址是A
源地址：A
目标地址：B
源地址：B
目标地址：A
nginx
操作系统：IP地址是B
设源地址为C的报文转
发本机nginx监听端口
源地址：B->A
目标地址：C
源地址：C
目标地址：A
网关改为B
操作系统：IP地址是C
上游服务进程

IP地址透传

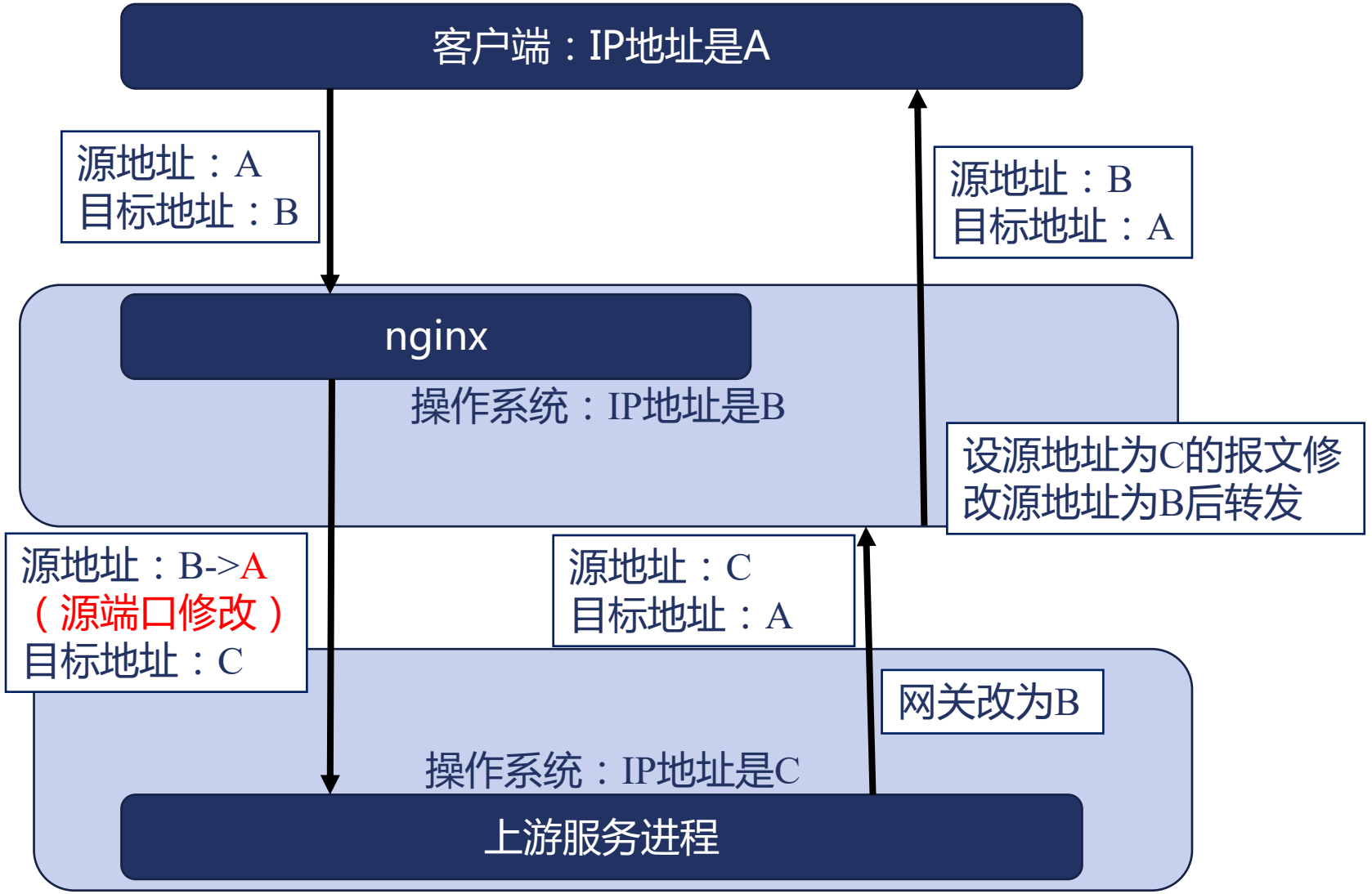
- proxy_bind $remote_addr transparent;
- 调节上游服务所在主机上的网关为nginx所在主机:

 # route del default gw 10.0.2.2（原网关IP）
 # route add default gw 172.16.0.1（Nginx所在主机的IP）

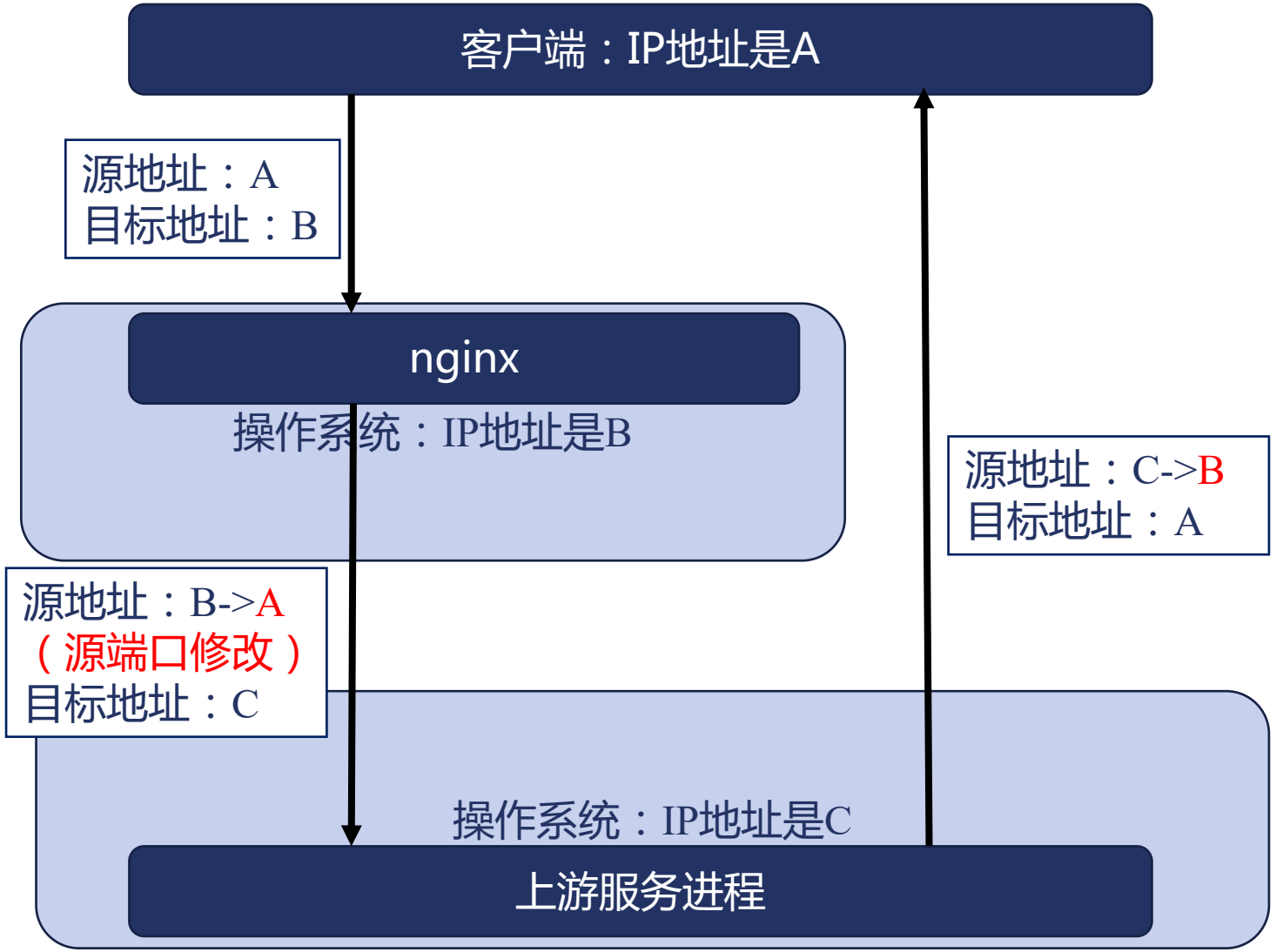
- 调节nginx所在主机上的路由规则，使它把接收自上游的、目标IP是客户端IP的报文转发nginx

 # ip rule add fwmark 1 lookup 100
 # ip route add local 0.0.0.0/0 dev lo table 100
 # iptables -t mangle -A PREROUTING -p tcp -s 172.16.0.0/28 --sport 80 -j MARK --set-xmark 0x1/0xffffffff

DSR方案示意图（1）


客户端：IP地址是A
源地址：A
目标地址：B
源地址：B
目标地址：A
nginx
操作系统：IP地址是B
设源地址为C的报文修改源地址为B后转发
源地址：B->A
（源端口修改）
目标地址：C
源地址：C
目标地址：A
网关改为B
操作系统：IP地址是C
上游服务进程

DSR方案示意图（2）


客户端：IP地址是A
源地址：A
目标地址：B
nginx
操作系统：IP地址是B
源地址：B->A
（源端口修改）
目标地址：C
源地址：C->B
目标地址：A
操作系统：IP地址是C
上游服务进程

DSR上游服务直接向客户端回包

- **简化版ip透传，由上游服务直接将报文发送给客户端**
  - 加proxy_responses 0
  - proxy_bind $remote_addr:$remote_port transparent;
  - 若上游服务在内网无公网ip，则可由nginx所在主机转发
    - 在上游服务所在主机上添加路由
      - # route add default gw nginx-ip-address
  - 允许操作系统转发ip报文
    - # sysctl -w net.ipv4.ip_forward=1
  - 转发时修改源地址为nginx所在主机的地址

```
# tc qdisc add dev eth0 root handle 10: htb
# tc filter add dev eth0 parent 10: protocol ip prio 10 u32 match ip src 172.16.0.11
match ip sport 53 action nat egress 172.16.0.11 192.168.99.10
# tc filter add dev eth0 parent 10: protocol ip prio 10 u32 match ip src 172.16.0.12
match ip sport 53 action nat egress 172.16.0.12 192.168.99.10
# tc filter add dev eth0 parent 10: protocol ip prio 10 u32 match ip src 172.16.0.13
match ip sport 53 action nat egress 172.16.0.13 192.168.99.10
# tc filter add dev eth0 parent 10: protocol ip prio 10 u32 match ip src 172.16.0.14
match ip sport 53 action nat egress 172.16.0.14 192.168.99.10
```

- **问题**
  - nginx检测不到上游服务是否回包
  - 负载均衡策略受限

扫码试看/订阅

《Nginx 核心知识100讲》