NGINX
Part of F5

nginx-cn.net

NGINX开源社区基础培训系列课程（第二季）

第二讲：如何高效地均衡应用层负载

主讲人：陶辉

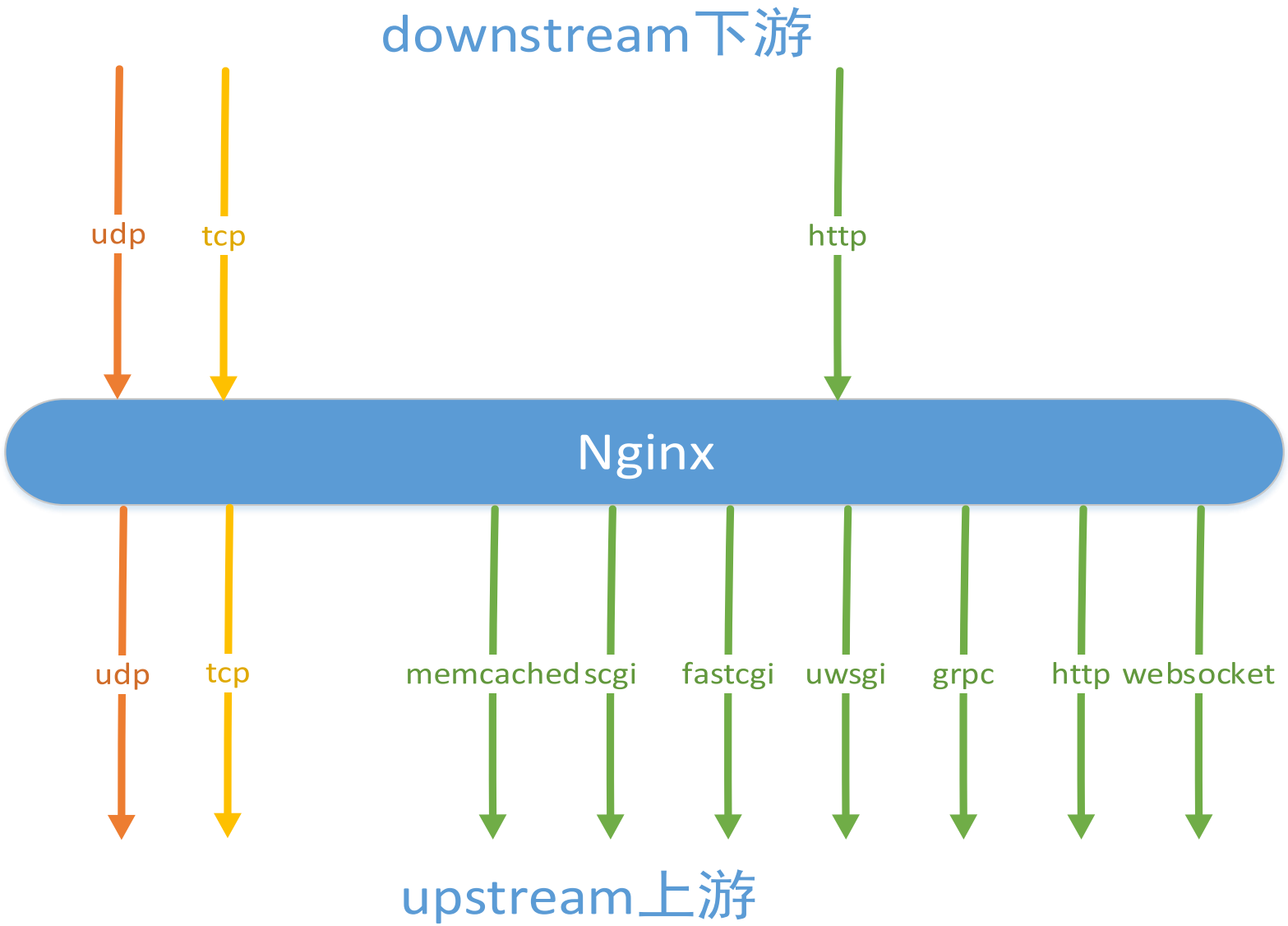
关注“NGINX开源社区”公众号参与后继活动。

NGINX开源社区群

NGINX开源社区公众号

NGINX

深入剖析HTTP负载均衡


downstream下游
udp
tcp
http
Nginx
udp
tcp
memcached
scgi
fastcgi
uwsgi
grpc
http
websocket
upstream上游

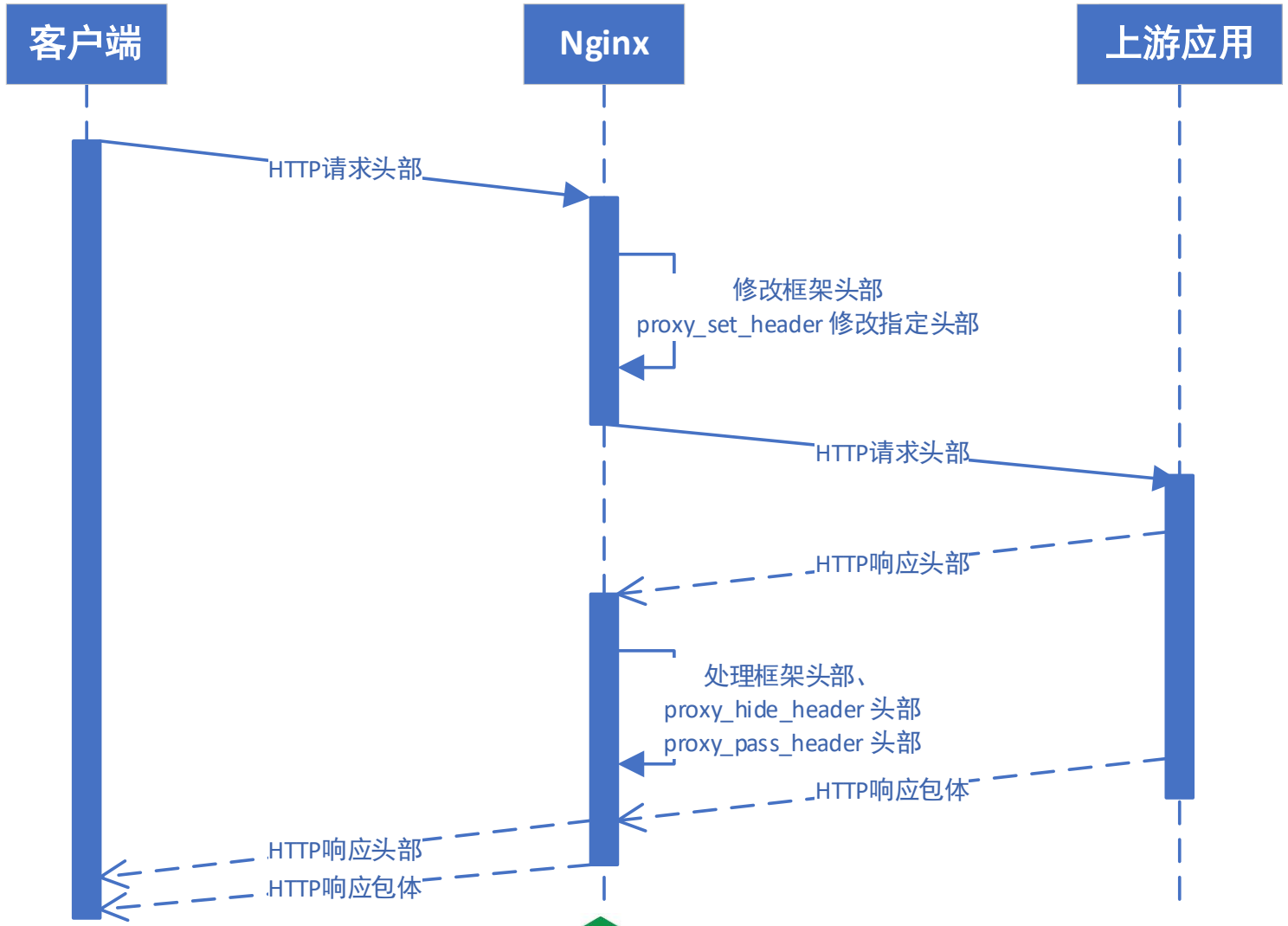
课程范围：负载均衡

- 下游协议
  - HTTP/1
  - HTTP/2
- 上游协议
  - HTTP/1
  - uwsgi/scgi/fastcgi

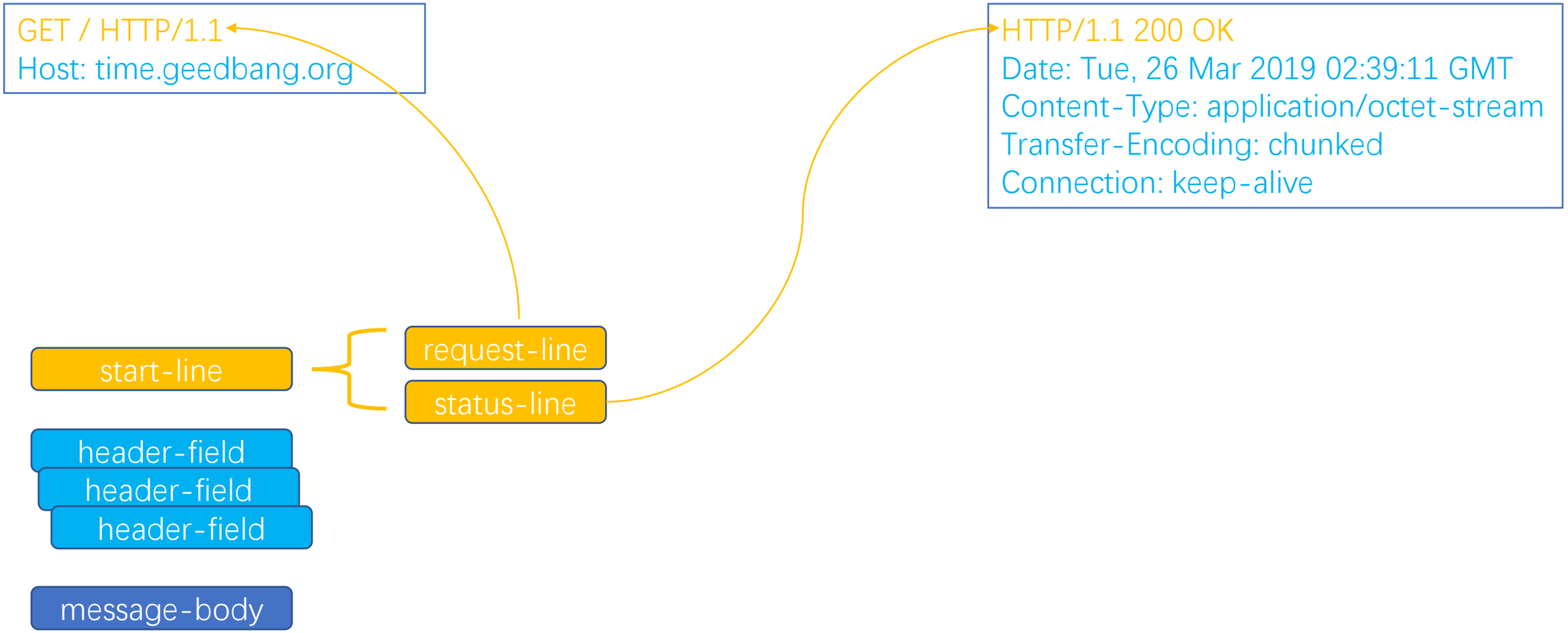
Nginx对哪些HTTP头部另眼相看？

- proxy_hide_header与proxy_pass_header冲突时，以谁为准？
- 为什么使用长连接时，一定要重设proxy_set_header？
- 如何向客户端返回上游的Server头部？
- 如何重设发向上游的HTTP头部？

HTTP反向代理流程


客户端
Nginx
上游应用
HTTP请求头部
修改框架头部
proxy_set_header 修改指定头部
HTTP请求头部
HTTP响应头部
处理框架头部、
proxy_hide_header 头部
proxy_pass_header 头部
HTTP响应包体
HTTP响应头部
HTTP响应包体

HTTP协议格式


GET / HTTP/1.1
Host: time.geedbang.org
HTTP/1.1 200 OK
Date: Tue, 26 Mar 2019 02:39:11 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive
start-line
request-line
status-line
header-field
header-field
header-field
message-body

定制请求行

- proxy_pass
- proxy_method
- proxy_http_version
  - 默认值：proxy_http_version 1.0;

Nginx默认处理的请求头部（1）

- 默认不转发的头部
  - TE
  - Keep-Alive
  - Expect
  - Upgrade

Nginx默认处理的请求头部（2）

- 默认要做转换的头部
  - Host
    - $proxy_host
  - Connection
    - close
  - Content-Length
    - $proxy_internal_body_length
  - Transfer-Encoding
    - $proxy_internal_chunked

Nginx默认处理的请求头部（3）

- 开启HTTP缓存后默认处理的头部
  - If-Modified-Since
    - $upstream_cache_last_modified
  - If-None-Match
    - $upstream_cache_etag
  - 不转发头部
    - If-Unmodified-Since
    - If-Match
    - Range
    - If-Range

设置发往上游的HTTP请求

- **两个开关**
  - proxy_pass_request_headers
  - proxy_pass_request_body
- **proxy_set_header**
  - value为空，表示不转发
  - 默认值
    - proxy_set_header Host $proxy_host;
      - $http_host、$host何意?
    - proxy_set_header Connection close;

示例：Nginx如何支持Websocket?

```
location /wsapp/ {
    proxy_pass http://wsbackend;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "Upgrade";
    proxy_set_header Host $host;
}
```

示例：
如何打开
HTTP长连接？

```
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 "";
        ...
    }
}
```


转发哪些HTTP响应头部？（1）

- 默认不转发的头部
  - Date
  - Server
  - X-Pad
  - X-Accel-
    - Expires
    - Redirect
    - Limit-Rate
    - Buffering
    - Charset

转发哪些HTTP响应头部？（2）

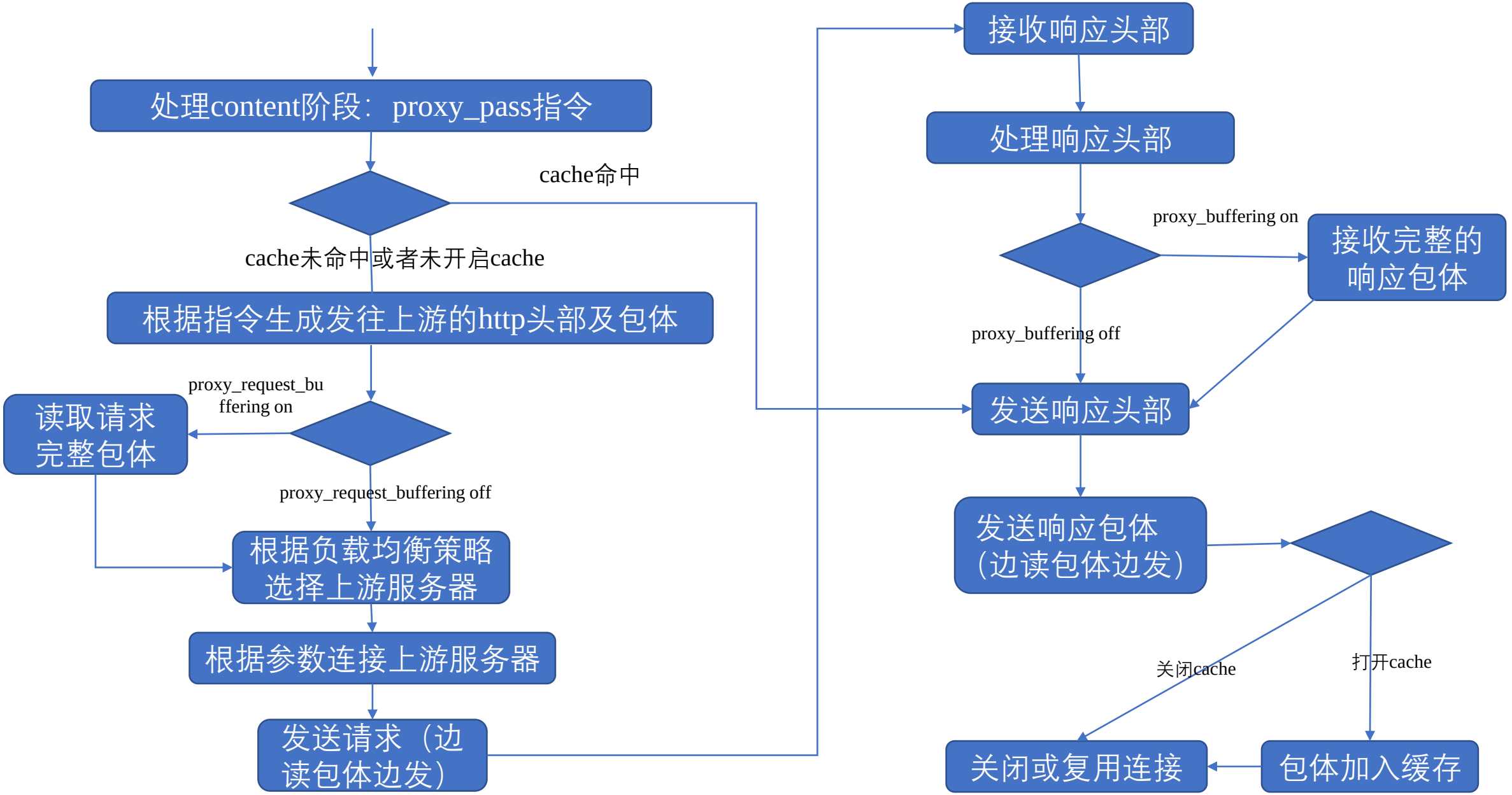
- proxy_hide_header
- proxy_pass_header

示例：上游的Server头部是怎样被换掉的?

- **hide_header初始化**
  - ngx_http_proxy_hide_headers
  - proxy_hide_header
  - proxy_pass_header
- **处理Server响应头部**
  - 是否存在hide_header中?
    - 不在，添加到headers_out结构体中
    - 在，不添加到headers_out结构体中
  - header_filter过滤模块
    - 检测headers_out->server不存在
      - 根据server_token指令，添加Server头部

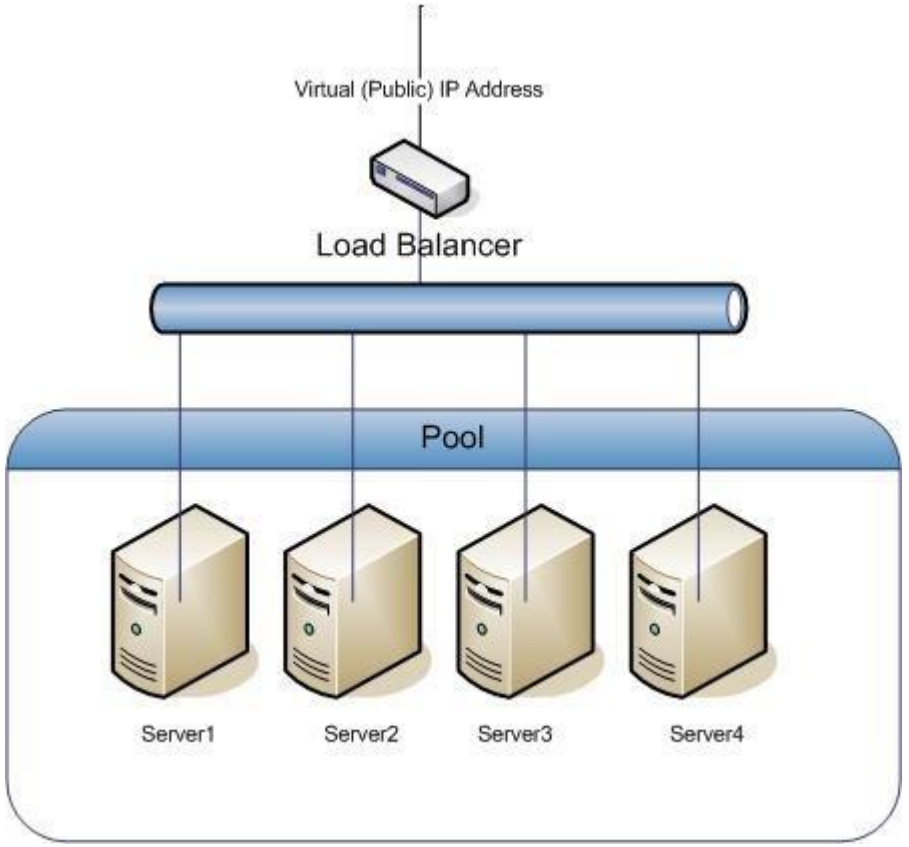
提升头部处理速度的哈希表

- proxy_headers_hash_bucket_size
- proxy_headers_hash_max_size


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

负载均衡算法分类

- **基于服务器负载的均衡算法**
  - 带权重的RoundRobin算法
  - Least Conn算法
  - 随机选择算法
  - 基于请求时延的算法
  - 基于服务器心跳的算法
- **基于用户请求的均衡算法**
  - 哈希算法
  - 一致性哈希算法



基于上游负载的均衡模块（1）

- ngx_http_upstream_module框架模块
  - 基本的server指令解析
    - weight：权重，默认为1
    - max_conns：最大并发连接数，默认为0表示不加限制
    - max_fails与fail_timeout：在fail_timeout时间段内，若失败次数超过max_fails，则将server在fail_timeout内置为不可用。
      - max_fails默认值为1
      - fail_timeout默认值10秒
    - backup：备份服务器，仅针对哈希算法、random算法有效
    - down：标识服务器永久下线

RoundRobin算法

- server 链表
- 不可用控制
  - 次数
    - fails
    - max_fails
  - 时间
    - fail_timeout
    - checked
    - accessed
- 最大连接数控制
  - conns
  - max_conns

```
struct ngx_http_upstream_rr_peer_s {
    struct sockaddr                *sockaddr;
    socklen_t                       socklen;
    ngx_str_t                       name;
    ngx_str_t                       server;
    ngx_int_t                       current_weight;
    ngx_int_t                       effective_weight;
    ngx_int_t                       weight;
    ngx_uint_t                      conns;
    ngx_uint_t                      max_conns;
    time_t                          accessed;
    ngx_uint_t                      max_fails;
    ngx_uint_t                      fails;
    time_t                          checked;
    time_t                          fail_timeout;
    ngx_msec_t                      slow_start; //NginxPlus
    ngx_msec_t                      start_time; //NginxPlus
    ngx_uint_t                      down;
#if (NGX_HTTP_UPSTREAM_ZONE)
    ngx_atomic_t                    lock;
#endif
    ngx_http_upstream_rr_peer_t     *next;
};
```

权重的实现

- server 链表
- 权重参数
  - current_weight
  - effective_weight
    - 初始值weight
    - peer->effective_weight -= peer->weight / peer->max_fails
  - weight

```
struct ngx_http_upstream_rr_peer_s {
    struct sockaddr                *sockaddr;
    socklen_t                       socklen;
    ngx_str_t                       name;
    ngx_str_t                       server;
    ngx_int_t                       current_weight;
    ngx_int_t                       effective_weight;
    ngx_int_t                       weight;
    ngx_uint_t                      conns;
    ngx_uint_t                      max_conns;
    time_t                          accessed;
    ngx_uint_t                      max_fails;
    ngx_uint_t                      fails;
    time_t                          checked;
    time_t                          fail_timeout;
    ngx_msec_t                      slow_start; //NginxPlus
    ngx_msec_t                      start_time; //NginxPlus
    ngx_uint_t                      down;
#if (NGX_HTTP_UPSTREAM_ZONE)
    ngx_atomic_t                    lock;
#endif
    ngx_http_upstream_rr_peer_t    *next;
};
```


权重是如何实现的?

```
upstream rrBackend {
        server localhost:8001 weight=1;
        server localhost:8002 weight=2;
        server localhost:8003 weight=3;
}
```

current_weight	current_weight +=effective_weight	total	选中server	current_weight
[0,0,0]	[1,2,3]	6	8003	[1,2,-3]
[1,2,-3]	[2,4,0]	6	8002	[2,-2,0]
[2,-2,0]	[3,0,3]	6	8003	[3,0,-3]
[3,0,-3]	[4,2,0]	6	8001	[-2,2,0]
[-2,2,0]	[-1,4,3]	6	8002	[-1,-2,3]
[-1,-2,3]	[0,0,6]	6	8003	[0,0,0]
[0,0,0]	[1,2,3]	6	8003	[1,2,-3]

基于上游负载的均衡模块（2）

- **ngx_http_upstream_least_conn_module模块**
  - 基于上游的最小连接数进行负载均衡
  - 支持weight、down、max_fails、fail_timeout选项
- **指令**
  - least_conn

基于上游负载的均衡模块（3）

- ngx_http_upstream_random_module模块
- 指令
  - Syntax: random [two [method]];
    - 在权重基础上，随机选择上游server
    - two least_conn
      - 在权重基础上，随机选择出2个上游server，再选择并发连接数最小的server
      - 开源Nginx method仅支持least_conn
  - Default: —
  - Context:upstream

基于Worker进程还是Nginx服务?

- **ngx_http_upstream_zone_module模块**
  - 将ngx_http_upstream_rr_peer_s链表拷贝到共享内存中
- **指令zone**
  - Syntax: zone name [size];
    - size省略或者为0时，会按需分配共享内存
  - Default: —
  - Context:upstream

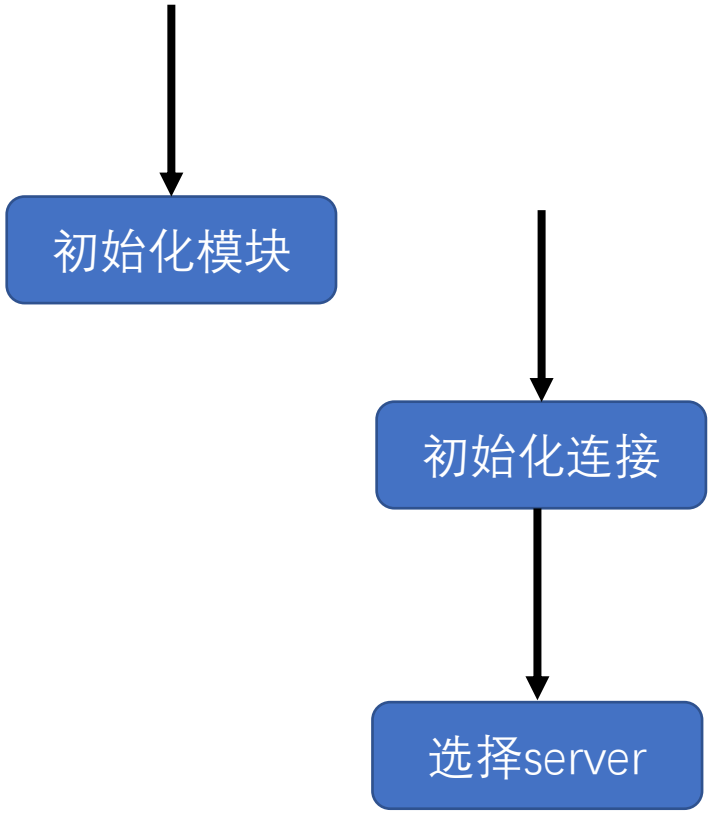
实现HTTP长连接的负载均衡模块

- **http_upstream_keepalive_module模块**
  - 从ngx_http_upstream_get_keepalive_peer到original_get_peer
- **指令keepalive**
  - Syntax: keepalive connections;
- **指令keepalive_requests**
  - Syntax: keepalive_requests number;
  - Default: keepalive_requests 100;
- **指令keepalive_timeout**
  - Syntax: keepalive_timeout timeout;
  - Default: keepalive_timeout 60s;

框架与模块(1)

- upstream模块的初始化顺序

```
ngx_module_t *ngx_modules[] = {
...
    &ngx_http_upstream_module,
...
    &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,
...
    &ngx_http_lua_upstream_module,
...
}
```



框架与模块(2)

- 初始化
  - Nginx启动时各算法的初始化
    - ngx_http_upstream_init_round_robin
    - ngx_http_upstream_init_hash
    - ngx_http_upstream_init_chash
    - ngx_http_upstream_init_ip_hash
    - ngx_http_upstream_init_least_conn
    - ngx_http_upstream_init_random
    - ngx_http_upstream_init_keepalive
  - 建立上游连接时，算法的初始化
    - ngx_http_upstream_init_round_robin_peer
    - ngx_http_upstream_init_hash_peer
    - ngx_http_upstream_init_chash_peer
    - ngx_http_upstream_init_ip_hash_peer
    - ngx_http_upstream_init_least_conn_peer
    - ngx_http_upstream_init_random_peer
    - ngx_http_upstream_init_keepalive_peer

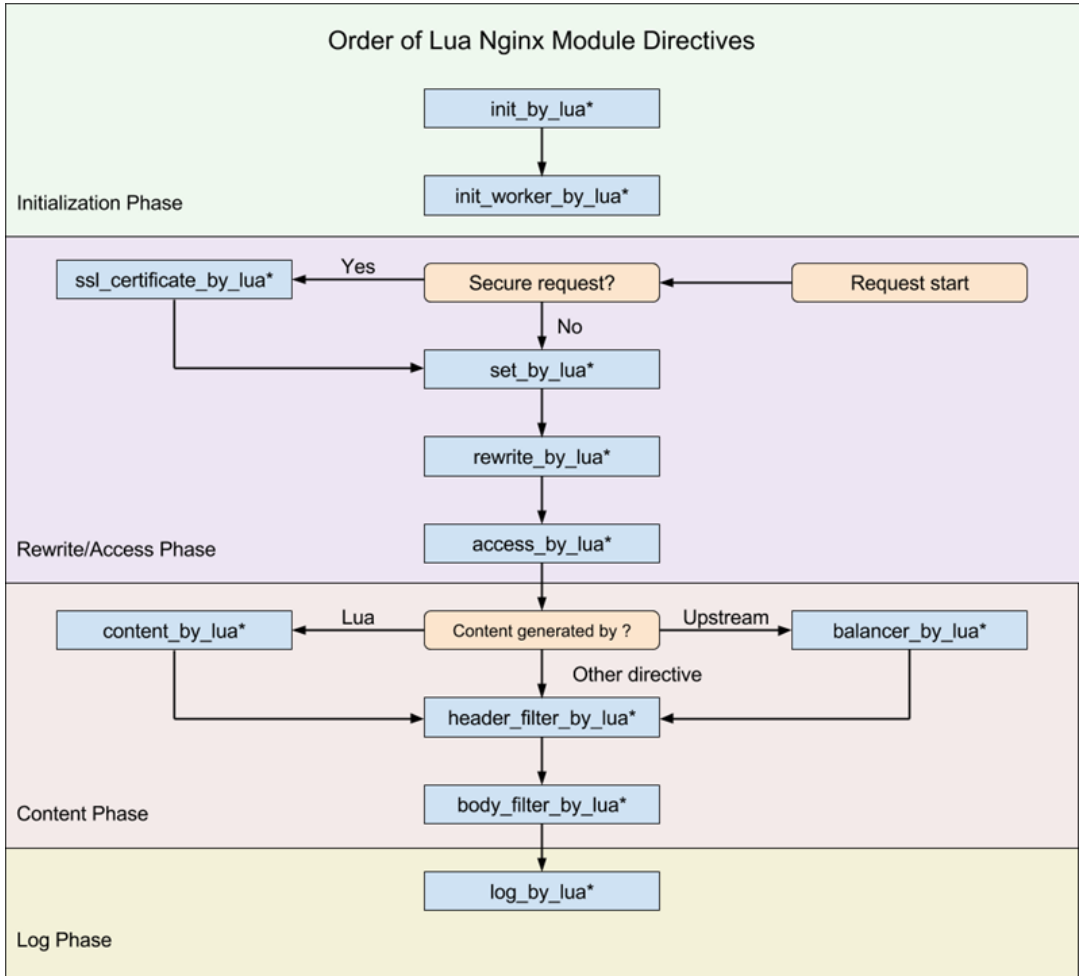
```
typedef struct {
    ngx_http_upstream_init_pt       init_upstream;
    ngx_http_upstream_init_peer_pt  init;
    void                           *data;
} ngx_http_upstream_peer_t;
```

框架与模块(3)

- 选择server
  - ngx_stream_upstream_get_round_robin_peer
  - ngx_http_upstream_get_hash_peer
  - ngx_http_upstream_get_chash_peer
  - ngx_http_upstream_get_ip_hash_peer
  - ngx_http_upstream_get_least_conn_peer
  - ngx_http_upstream_get_random_peer
  - ngx_http_upstream_get_random2_peer
  - ngx_http_upstream_get_keepalive_peer

Openresty

- **balancer_by_lua_block**
  - syntax: balancer_by_lua_block { lua-script }
  - context: upstream
- **balancer_by_lua_file**
  - syntax: balancer_by_lua_file <path-to-lua-script-file>
- **API**
  - syntax: ok, err = balancer.set_current_peer(host, port)
  - context: balancer_by_lua*



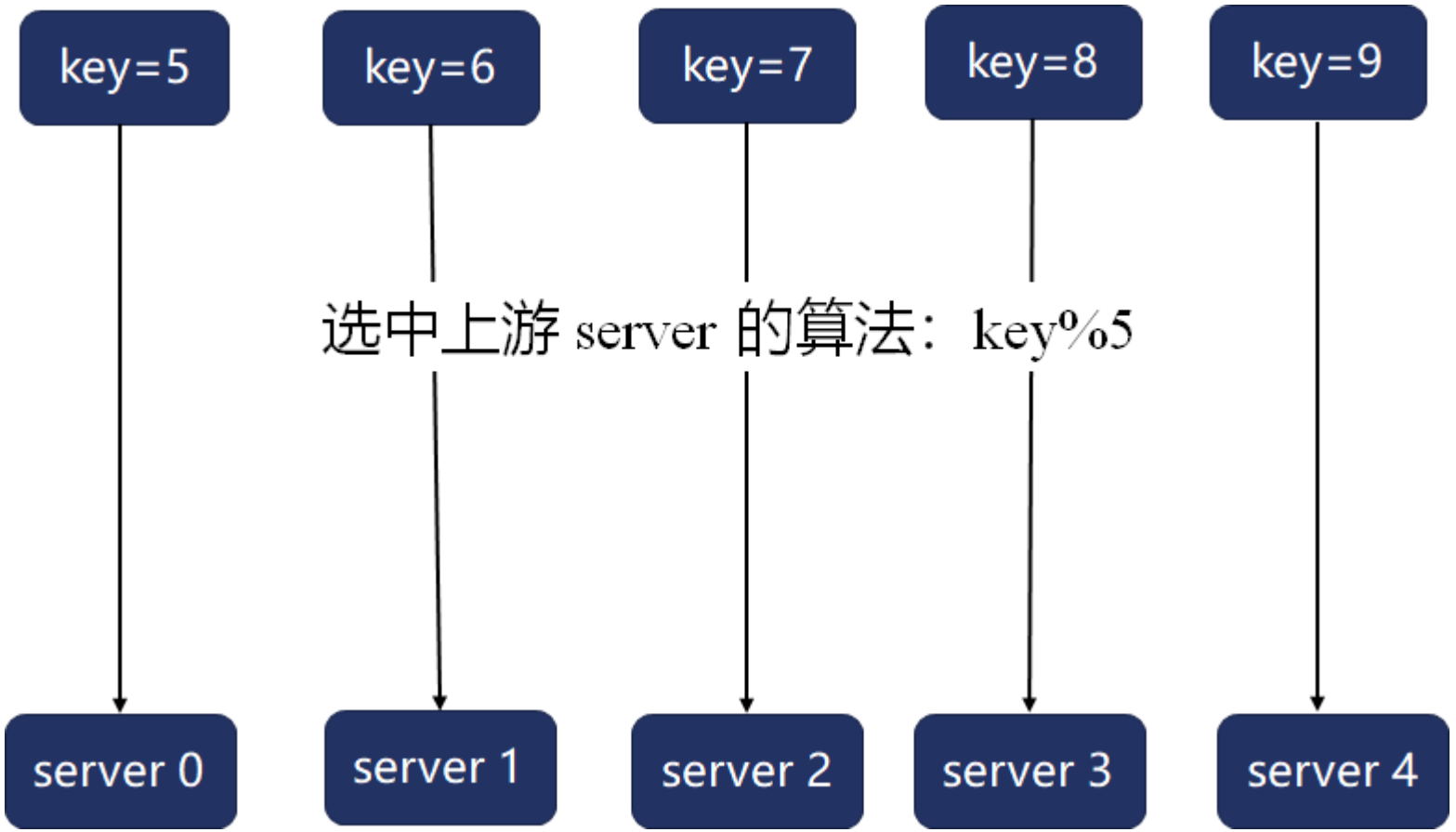
基于用户请求的均衡模块（4）

- http_upstream_ip_hash_module模块
- 指令
  - ip_hash
    - IPv4：前3个字节
    - IPv6：全部16个字节

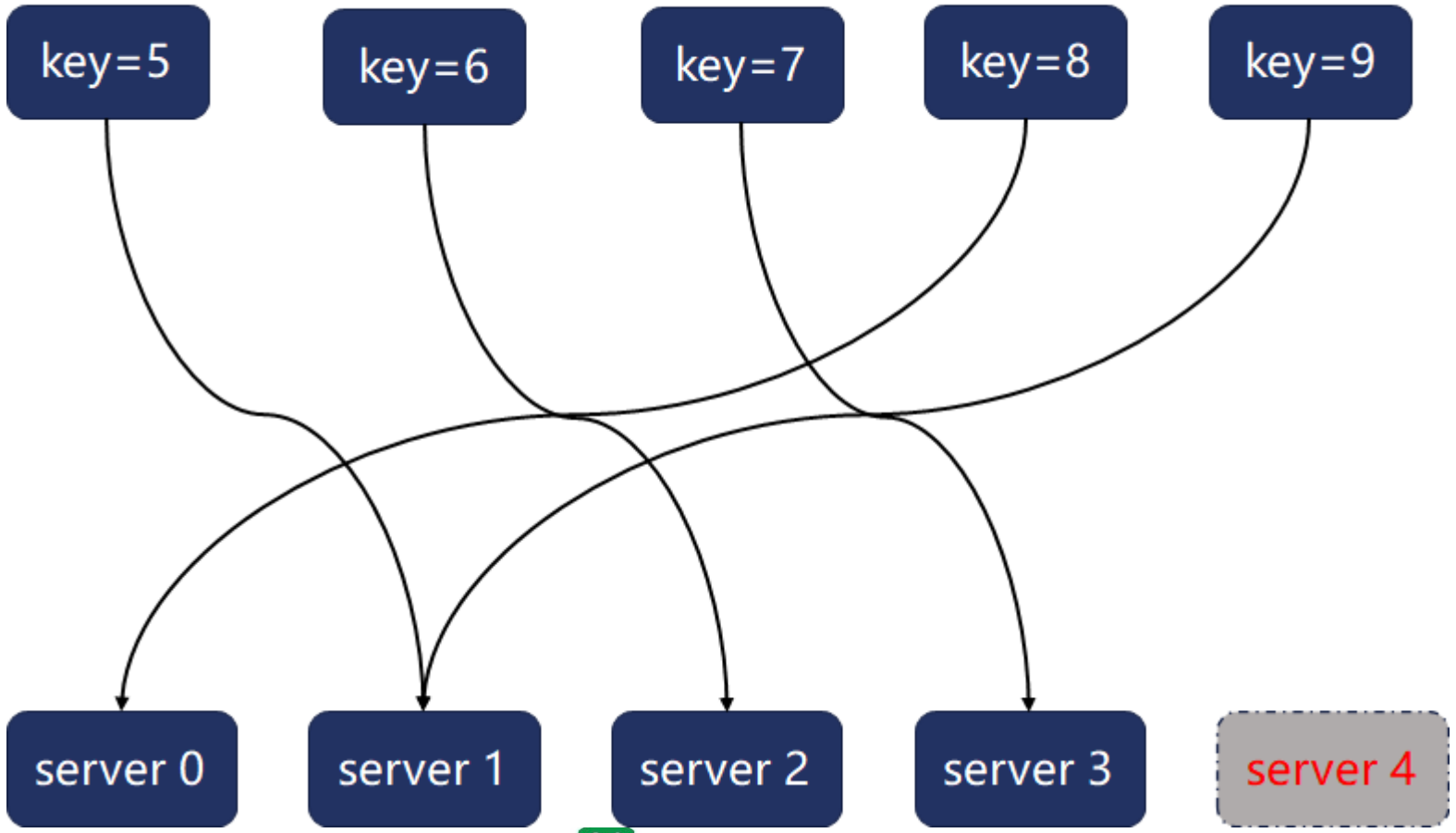
基于用户请求的均衡模块（5）

- **http_upstream_hash_module模块**
  - 基于权重、max_fails与fail_timeout、max_conns实现
- **指令**
  - Syntax: hash key [~~consistent~~];
  - Default: —
  - Context:upstream

哈希算法的问题


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

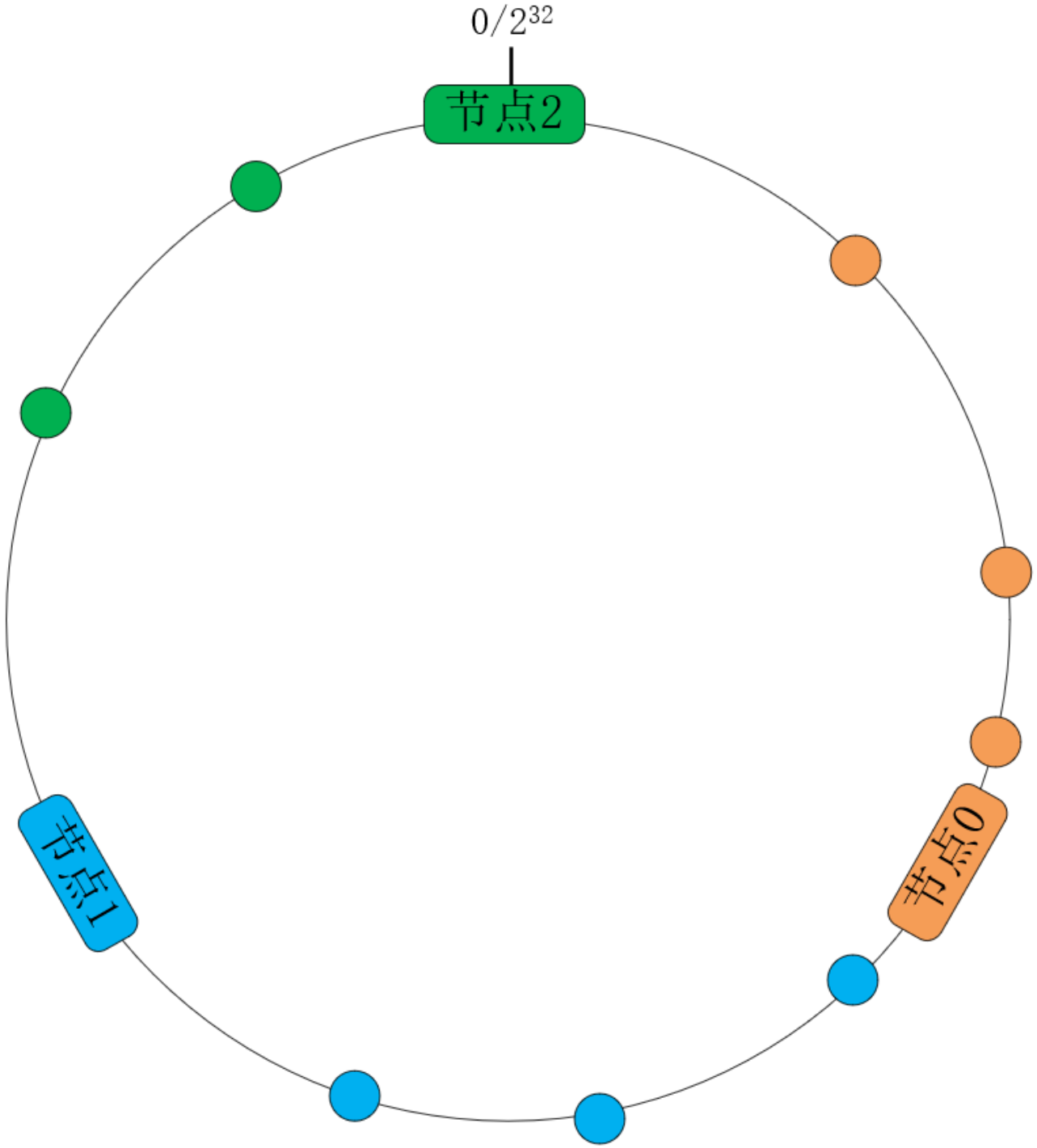
选中上游 server 的算法：key%4



基于用户请求的均衡模块（6）

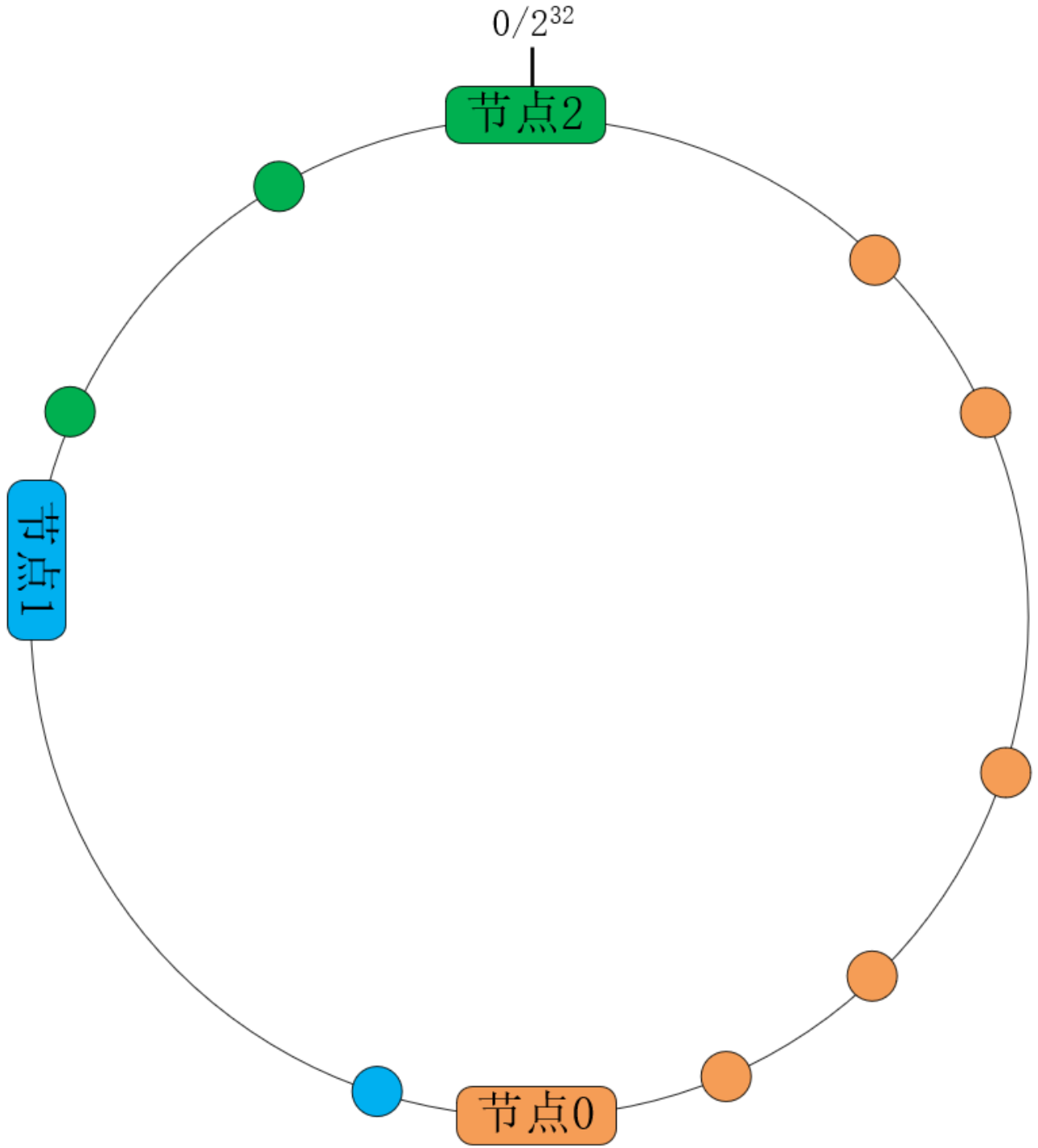
- **http_upstream_hash_module模块**
  - 基于权重、max_fails与fail_timeout、max_conns实现
- **指令**
  - Syntax: hash key [consistent];
- **一致性哈希**
  - 基于CRC32算法构造32位哈希值
  - 每个权重分配160个虚拟节点
  - 基于二分法，在logN时间复杂度内找到server

一致性哈希（1）



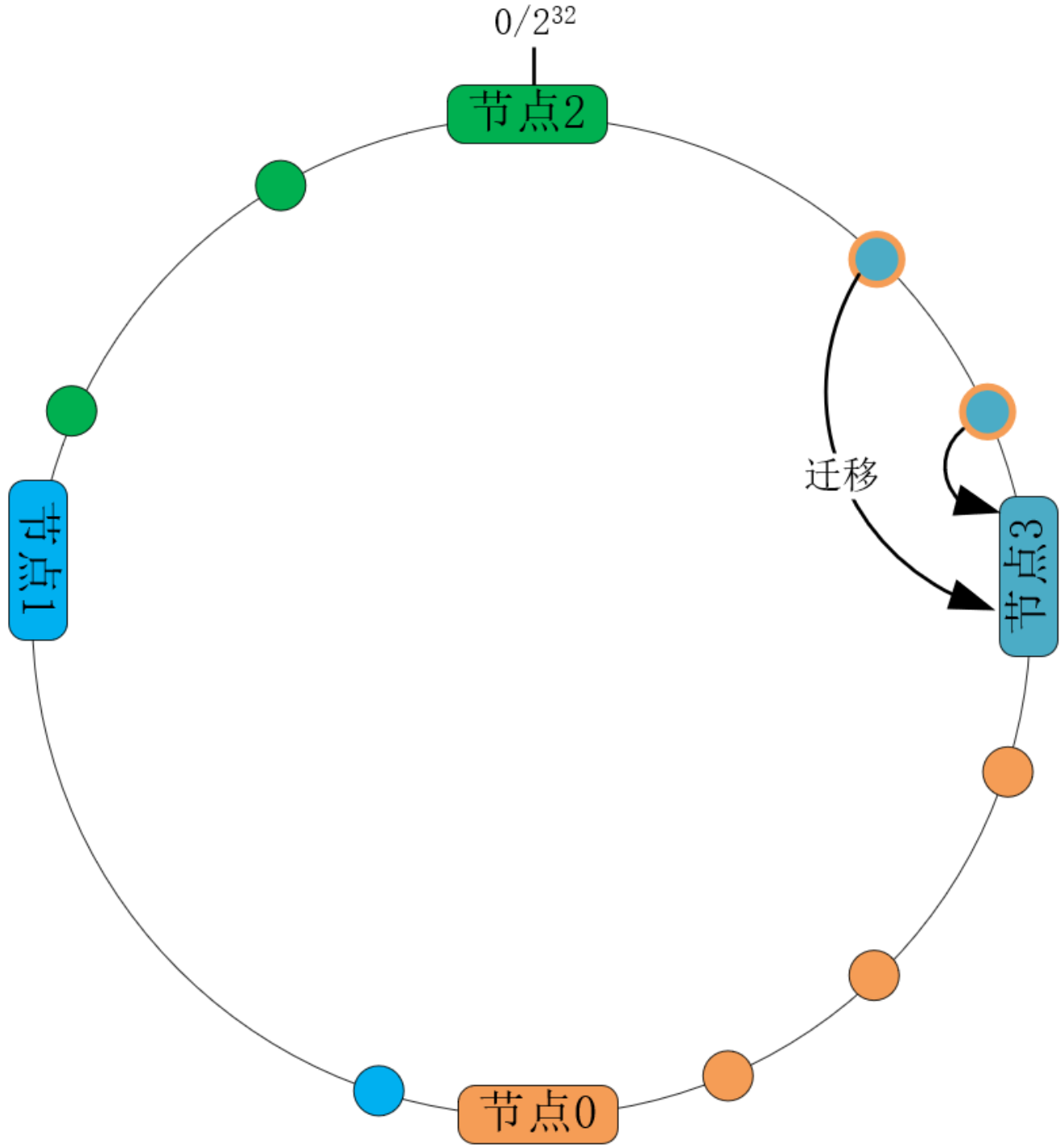
一致性哈希（2）

- 支持权重



一致性哈希（3）

- 数据迁移



谢谢

NGINX
Part of F5

近期活动推荐

New

扫码查看
日程安排

KubeCon 2020 开源盛会，关注F5社区后继活动，获得直播入场券。
https://cncf.lfasiallc.cn/schedule/cn

扫码观看

7月09日- 7月30日| 每周四 | 20:00-21:00
下一场课程：7月23日 周四 晚上8-9点
主题：怎样向客户端隐藏应用层错误　　讲师：陶辉

NGINX
Part of F5

近期活动推荐

扫码进群
获取录播及课程ppt

主题：应用交付的软件可编程动态能力

讲师均为F5内部顶级解决方案顾问及技术专家

扫码回看

F5一年一度的重磅盛会，3大主题演讲，5大技术专题论坛，30+技术话题，100+解决方案，由F5技术专家，行业领头羊客户，F5全球合作伙伴为大家一一呈现。

NGINX
Part of F5

关注我们

NGINX开源社区微信

NGINX 社区微信群

NGINX开源社区官方微博

NGINX开源社区是F5/NGINX面向所有NGINX用户的官方社区。我们秉持“开放，包容，沟通，贡献（Open，Inclusive，Connect，Contribution）”之宗旨，与业界共建开放、包容、活跃的“NGINX用户之家”；秉承开源的精神，在社区治理上高度开放，为所有NGINX的用户，开发者和技术爱好者，提供一个方便学习、讨论的场所。也期待您成为此社区中活跃的一员，贡献您的文章，博客，代码，踊跃讨论与回答问题，打造您个人品牌和影响力。

点击访问NGINX开源社区网站：nginx-cn.net