Apusic 金蝶天燕

生成时间：2026-07-13

代理功能手册

金蝶Apusic负载均衡器

版权声明

本文档所涉及的软件著作权、版权等知识产权已依法进行了注册，由金蝶天燕云计算股份有限公司合法拥有。受《中华人民共和国著作权法》《计算机软件保护条例》《知识产权保护条例》和相关国际版权条约、法律、法规以及其它知识产权法律和条约的保护。未经授权许可，不得非法使用。

免责声明

本文档包含的版权信息由金蝶天燕云计算股份有限公司合法拥有，受法律的保护，金蝶天燕云计算股份有限公司对本文档可能涉及到的非金蝶天燕云计算股份有限公司的信息不承担任何责任。在法律允许的范围内，您可以查阅并仅能够在《中华人民共和国著作权法》规定的合法范围内复制和打印本文档。任何单位和个人未经金蝶天燕云计算股份有限公司书面授权许可，不得使用、修改、再发布本文档的任何部分和内容，否则将被视为侵权，金蝶天燕云计算股份有限公司有依法追究其责任的权利。

本文档如有更新，不另行通知。对本文档中的问题您可向金蝶天燕云计算股份有限公司告知或查询。未经本公司明确授予的任何权利均予保留。

商标声明

Apusic是深圳市金蝶天燕云计算股份有限公司向中华人民共和国国家商标局申请注册的注册商标，注册商标专用权由金蝶天燕合法拥有，受法律保护。未经金蝶天燕的书面许可，任何单位及个人不得以任何方式或理由对该商标的任何部分进行使用、复制、修改、传播、抄录或与其它产品捆绑使用销售。凡侵犯金蝶天燕商标权的，金蝶天燕将依法追究其法律责任。本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

目录

1 前言

本文档为金蝶Apusic负载均衡器软件V2.0.6标准版的代理功能手册，详细介绍ALB代理相关功能的使用、配置及管理方法，包括负载均衡、健康检查、流量控制、灰度发布、TCP/UDP代理、HTTPS与国密、HTTP/3、正向代理等。

1.1 适用对象

本文档适用于IT信息化业务负责人、研发经理、软件项目经理、软件架构师、运维工程师。

1.2 相关文档

了解更多ALB V2.0.6产品相关的信息，请参阅以下ALB V2.0.6产品手册文档集：

序号	手册文档	说明
1	金蝶Apusic负载均衡器软件 V2.0.6 发版说明	介绍了ALB 产品版本更新内容 。
2	金蝶Apusic负载均衡器软件 V2.0.6 快速使用手册	简单介绍了如何快速上手使用ALB 。
3	金蝶Apusic负载均衡器软件 V2.0.6 安装手册	详细介绍如何在各操作系统上安装ALB，以及ALB服务启停操作，产品的注册过程。
4	金蝶Apusic负载均衡器软件 V2.0.6 代理功能手册	详细介绍 ALB 相关功能的使用、配置、管理及配套工具的使用方法。
5	金蝶Apusic负载均衡器软件 V2.0.6 管控台用户手册	详细介绍ALB管控台相关功能的使用和操作说明。
6	金蝶Apusic负载均衡器软件 V2.0.6 开发手册	详细介绍ALB外部API、alb-cli命令行工具及AI Agent集成方法。
7	金蝶Apusic负载均衡器软件 V2.0.6 迁移手册	详细介绍ALB历史版本迁移升级到V2.0.6版本的说明。
8	金蝶Apusic负载均衡器软件 V2.0.6 运维手册	详细介绍ALB的监控、运维、安全加固等运维说明。
9	金蝶Apusic负载均衡器软件 V2.0.6 性能优化手册	详细介绍ALB性能调优的说明。

1.3 技术支持

ALB产品提供全面的技术支持服务，您可以通过以下方式获得技术支持：

- 网址：www.apusic.com
- 电话：400-855-5800
- 邮箱：support@apusic.com
- 金蝶云社区：https://vip.kingdee.com/?productId=73&productLineId=14&lang=zh-CN

您在取得技术支持时，请提供如下信息：

1. 您的姓名
2. 公司信息与联系方式
3. 操作系统及其版本
4. 产品版本号
5. 出现异常及错误的日志、截图等详细信息

2 功能模块清单

2.1 ALB功能模块清单

模块	功能说明
--with-http_stub_status_module	启用 stub_status 接口，提供运行状态统计信息。
--with-http_ssl_module	启用 HTTPS/SSL 支持。
--with-ipv6	启用 IPv6 监听支持。
--with-http_mp4_module	支持 MP4 流媒体伪流（pseudo-streaming）。
--with-http_auth_request_module	实现子请求认证（auth_request），方便与外部认证服务集成。
--with-http_gzip_static_module	支持直接发送 .gz 预压缩文件，节省 CPU。
--with-http_realip_module	从 X-Forwarded-For/X-Real-IP 等头中获取真实客户端 IP。
--with-http_gunzip_module	对不支持 gzip 的客户端自动解压响应内容。
--with-http_sub_module	响应体字符串替换（sub_filter）。
--with-stream	启用 4 层 TCP/UDP 代理（stream 子系统）。
--with-stream_ssl_module	为 stream 子系统提供 SSL/TLS 支持。
--with-threads	开启线程池支持，提升文件 I/O 性能。
--with-file-aio	启用异步文件 I/O（Linux AIO）。
--with-http_v2_module	启用 HTTP/2 支持。
--with-http_v3_module	启用 HTTP/3（QUIC）支持。
--with-http_addition_module	在响应前后追加内容（add_before_body / add_after_body）。
--with-http_dav_module	支持 WebDAV 协议（PUT、DELETE、MKCOL 等）。
--with-http_random_index_module	目录索引时随机选择默认文件。
--with-http_secure_link_module	生成和校验带时效、防篡改的“安全链接”。
--with-mail	启用邮件代理（IMAP/POP3/SMTP）子系统。
--with-mail_ssl_module	为邮件代理提供 SSL/TLS 支持。

--with-http_slice_module	支持大文件分片下载（Range Slice）。
--with-stream_ssl_preread_module	在 4 层代理里预读 SNI/ALPN，实现按域名路由。
alb_healthcheck_module	ALB主动健康检查模块，支持四层和七层代理节点主动探活。
alb-module-vts	提供详细流量 / 状态统计接口（/status）。
alb-module-stream-sts	为 stream 子系统提供实时状态统计。
alb-module-sts	为 http 子系统提供实时状态统计。
alb_http_proxy_connect_module	支持 HTTP CONNECT 方法，正向代理 HTTPS 流量。
alb-rtmp-module	引入 RTMP/HLS/DASH 直播与流媒体功能。
alb_brotli	支持 Brotli 压缩算法，减少传输体积。
alb-module-lua	引入 Lua 脚本功能，可扩展 ALB 功能。
alb-headers-more	设置和清除-请求和响应头。

3 负载均衡算法

3.1 轮询

轮询是ALB默认的负载均衡算法，按时间顺序将请求轮流分配给后端服务器。如果有服务器宕机，ALB会自动将其剔除。

如下配置多个后端服务器后，默认使用轮询负载均衡算法。

```
upstream backend {
    server backend1.example.com;
    server backend2.example.com;
}
```

3.2 权重轮询

在轮询的基础上，根据指定的权重分配请求，权重越高的服务器处理的请求越多，适用于服务器性能不均的情况。

如下配置多个后端服务器后，通过使用weight参数指定权重。

```
upstream backend {
    server backend1.example.com weight=10;   # 通过weight参数设置权重
    server backend2.example.com weight=20;
}
```

3.3 IP哈希

根据客户端的IP地址进行哈希运算，将请求分配给后端服务器。可以保证来自同一IP地址的请求始终被分配到同一台后端服务器。

如下配置多个后端服务器后，通过使用ip_hash指令启用IP Hash负载均衡算法。

```
upstream backend {
    ip_hash;   # 开启IP哈希负载均衡
    server backend1.example.com;
```

```
    server backend2.example.com;
}
```

3.4 最少连接数

根据后端服务器的当前活动连接数量进行分配，将请求分配给活动连接最少的后端服务器。适合请求处理时间差异较大的场景

如下配置多个后端服务器后，通过使用least_conn指令启用最少连接数负载均衡算法。

```
upstream backend {
    least_conn;  # 开启最少连接数负载均衡
    server backend1.example.com;
    server backend2.example.com;
}
```

4 健康检查

ALB支持对后端服务器进行健康检查，当后端服务器的响应不符合预期时，ALB会将其剔除。

4.0.1 被动健康检查

被动健康检查由ALB自动执行，根据后端服务器的响应状态码和超时时间来判断其是否健康。

ALB的被动健康检查通过 `fail_timeout` 和 `max_fails` 参数进行配置。

- `fail_timeout`：指定在连续失败后，ALB将后端服务器标记为不健康的时间段。默认值为10秒。
- `max_fails`：指定在 `fail_timeout` 时间段内，连续失败的次数。默认值为3次。

例如，如果你希望在某个后端服务器连续失败 3 次后将其标记为不可用，并在接下来的 30 秒内不向其发送请求，你可以这样配置

```
upstream backend {
    server 192.168.0.1:80 fail_timeout=30s max_fails=3;
    server 192.168.0.2:80 fail_timeout=30s max_fails=3;
}
```

注：

- 如果后端服务器在 fail_timeout 设置的时间结束后仍然不可用，ALB 会再次尝试向该服务器发送请求。
- 当后端服务器恢复正常后，ALB 会自动将其重新加入到轮询队列中。
- 使用这些指令可以帮助减少由于网络波动等原因造成的短暂失败对用户体验的影响。

4.0.2 主动健康检查

主动健康检查由ALB定期执行，通过向后端服务器发送特定的请求判断是否健康。如果请求成功返回，则认为后端服务器是健康的；否则，将其标记为不健康。

4.0.2.1 七层HTTP代理健康检查

ALB的主动健康检查通过 `check` 指令进行配置。

```
upstream backend {
    server 192.168.0.1:80;
    server 192.168.0.2:80;

    # 启用健康检查
```

```
    check interval=3000 rise=2 fall=3 timeout=2000 type=http;

    # 定义健康检查的 URL
    check_http_send "GET /healthcheck HTTP/1.0\r\nHost:
www.example.com\r\n\r\n";
    check_http_expect_alive http_2xx http_3xx;
}
```

解释配置指令:

- check：开启健康检查，配置健康检查的基本参数。
- interval：定义两次检查之间的间隔时间（单位毫秒）。
- rise：定义连续几次成功响应后认为后端服务器是健康的。
- fall：定义连续几次失败响应后认为后端服务器是不健康的。
- timeout：定义单次健康检查请求的超时时间（单位毫秒）。
- type：定义健康检查请求的类型，如 http 或 tcp。
- check_http_send：定义发送给后端服务器的 HTTP 请求。
- 这里使用了一个 GET请求来检查 /healthcheck 路径。
- check_http_expect_alive：定义期望从后端服务器收到的成功响应的状态码范围。
- http_2xx 和 http_3xx 表示期望收到 2xx 和 3xx 状态码。

4.0.2.2 四层TCP代理健康检查

ALB主动健康检查支持TCP代理，以下是样例:

```
stream {
    upstream tcp-cluster {
        server 127.0.0.1:22;
        server 192.168.0.2:22;
        check interval=3000 rise=2 fall=5 timeout=5000
default_down=true type=tcp;
    }
    server {
        listen 522;
        proxy_pass tcp-cluster;
    }

    upstream udp-cluster {
```

```
        server 127.0.0.1:53;
        server 8.8.8.8:53;  # 示例DNS服务器
        check interval=3000 rise=2 fall=5 timeout=5000
default_down=true type=udp;
    }
    server {
        listen 53 udp;
        proxy_pass udp-cluster;
    }
}
```

4.0.2.3 不同类型说明

1. TCP 健康检查 (type=tcp)，**适用于所有 TCP 后端服务**

TCP 健康检查是最简单的类型之一，它仅需要建立一个 TCP 连接到后端服务器，并读取一个字节来判断服务器是否可连通。如下为实例配置

```
http {
    upstream backend {
        server 192.168.0.1:80;
        server 192.168.0.2:80;

        check interval=3000 rise=2 fall=3 timeout=2000 type=tcp;

    }

    server {
        listen 80;
        server_name example.com;

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

2. HTTP 健康检查 (type=http)，**主要用于七层 HTTP 代理**

HTTP 健康检查通过发送一个 HTTP 请求到后端服务器来判断服务器的状态。如下为实例配置

```
http {
    upstream backend {
        server 192.168.0.1:80;
        server 192.168.0.2:80;

        check interval=3000 rise=2 fall=3 timeout=2000 type=http;

        # 发送一个 GET 请求到 /healthcheck 路径
        check_http_send "GET /healthcheck HTTP/1.0\r\nHost:
www.example.com\r\n\r\n";

        # 期望服务器返回 2xx 或 3xx 状态码表示健康
        check_http_expect_alive http_2xx http_3xx;
    }

    server {
        listen 80;
        server_name example.com;

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

3. SSL Hello 健康检查 (type=ssl_hello)

SSL Hello 健康检查通过发送 SSL 客户端 Hello 包，并接收服务器的 SSL Hello 包来判断服务器是否健康。如下为实例配置

```
http {
    upstream backend {
        server 192.168.0.1:443;
```

```
        server 192.168.0.2:443;

        check interval=3000 rise=2 fall=3 timeout=2000
type=ssl_hello;
    }

    server {
        listen 443 ssl;
        server_name example.com;

        location / {
            proxy_pass https://backend;
        }
    }
}
```

4. MySQL 健康检查 (type=mysql)

MySQL 健康检查通过建立 MySQL 连接并接收问候响应来判断数据库服务器是否健康。

```
http {
    upstream backend {
        server 192.168.0.1:3306;
        server 192.168.0.2:3306;

        check interval=3000 rise=2 fall=3 timeout=2000 type=mysql;
    }

    server {
        listen 80;
        server_name example.com;

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

```
    }
}
```

5. FastCGI 健康检查 (type=fastcgi)

FastCGI 健康检查通过发送一个 FastCGI 请求，并接收响应来判断服务器是否健康。

```
http {
    upstream backend {
        server 192.168.0.1:9000;
        server 192.168.0.2:9000;

        check interval=3000 rise=2 fall=3 timeout=2000 type=fastcgi;
    }

    server {
        listen 80;
        server_name example.com;

        location / {
            proxy_pass fastcgi://backend;
        }
    }
}
```

6. UDP 健康检查 (type=udp)，**仅支持四层代理**

UDP 健康检查通过发送一个 UDP 请求，并接收响应来判断服务器是否健康。

```
stream {
    upstream udp-cluster {
        server 127.0.0.1:53;
        server 8.8.8.8:53;  # 示例DNS服务器
        check interval=3000 rise=2 fall=5 timeout=5000
default_down=true type=udp;
    }
    server {
```

```
        listen 53 udp;
        proxy_pass udp-cluster;
    }
}
```

4.0.2.4 主动健康检查状态获取

ALB支持通过指定接口获取主动健康检查状态，如下：

1. 开启主动健康检查状态获取

```
# 省略其他配置

http {
    server {
        listen 8080;

        # 获取主动健康检查状态URL
        # 通过ALB的8080端口的/status接口获取健康状态数据页面
        location /status {
            healthcheck_status html;
        }
    }
    # 省略其他配置
}
```

访问8080端口/status接口，即可获取健康状态数据页面：

ALB upstream status monitor

http upstream servers

up: 1 down: 1 total: 2

Index	Upstream	Name	Status	Rise counts	Fall counts	Check type	Check port
0	backend	172.21.61.67:8006	up	6	0	ssl_hello	0
1	backend	172.21.61.46:22	down	0	6	ssl_hello	0

stream upstream servers

up: 1 down: 1 total: 2

Index	Upstream	Name	Status	Rise counts	Fall counts	Check type	Check port
0	tcp-cluster	127.0.0.1:22	down	0	6	tcp	0
1	tcp-cluster	172.21.61.66:22	up	6	0	tcp	0

total servers(check enabled): 4

2. 支持类型

- html：返回HTML页面
- json：返回JSON数据
- csv：返回CSV数据
- prometheus：返回Prometheus格式数据

5 监控

ALB提供丰富的监控指标，包括：

1. 简单流量统计数据：提供基础连接和请求统计，不支持Prometheus采集格式。
2. Prometheus-exporter：支持将流量数据以Prometheus格式暴露，便于与Prometheus监控系统集成。
3. VTS监控数据：提供ALB精细化监控数据，支持HTML页面展示、JSON数据格式和Prometheus采集格式，功能最为全面。

注： 上面三种方式可按需选择其中一种。

5.0.1 简单流量统计数据

ALB提供简单的流量统计信息，其监控数据有：

- Active connections: 当前活跃的连接数。
- Server accepts, handled, requests: 自ALB启动以来接受、处理的连接数以及收到的请求数。
- Reading: 当前正在读取客户端请求的数量。
- Writing: 当前正在向客户端写入响应的数量。
- Waiting: 当前处于等待状态的连接数。这些连接已经完成了读取操作，等待下一次写入操作。

5.0.1.1 配置简单流量统计数据和获取

在ALB配置文件中，添加以下内容：

```
server {
    listen 8080;              # node_status的端口

    location /node_status {
        stub_status on;       # 开启node_status功能
    }
}
```

启动ALB后，访问http://alb_ip:8080/node_status即可查看。

5.0.2 Prometheus-exporter（Prometheus监控数据）

ALB提供Prometheus监控功能，默认不启用，如需启用，请切换到utils/exporter目录，

执行：`./start-exporter.sh`，启动ALB的exporter。

5.0.2.1 启动说明

启动ALB的exporter必须要在配置文件中开启stub_status功能，如下：

```
server {
  listen 8080;              # node_status的端口
  location /node_status {  # 必须要使用node_status
        stub_status on;    # 开启stub_status
        access_log off;
        allow 127.0.0.1;
    }
}
```

在执行start-exporter脚本后，要求输入

- node_status对应的server监听端口，默认为8080.
- exporter监听端口，默认为9113

如下为输入过程：

```
root@root:/opt/ALB-V2.0.6-SE-amd64/utils/exporter$ ./start-
exporter.sh
input alb server listening port(请输入node_status监听端口):
alb server listening port(ALB node_status端口):  8080
input exporter listening port(请输入监听端口):
exporter listening port is 9113
exporter are running now!(exporter启动成功)
```

5.0.2.2 Prometheus监控数据获取

通过start-exporter成功后，可以通过浏览器或者prometheus访问：`http://IP:9113/metrics`获取监控数据。

注： 如果上面修改了exporter的监听端口，上面url中的9113也一并修改。

5.0.3 VTS监控数据

vts模块提供更详细的监控信息，包括连接数、状态码等。与上面两种监控方式相比，vts模块可以提供更加全面和丰富的监控数据，通过vts模块，可以更加深入了解ALB的运行状态。

5.0.3.1 监控指标说明

监控类别	指标名称	说明

1. **基本信息**	主机信息	代理服务器所在的主机名或 IP 地址
	版本号	ALB 或监控模块的版本
	服务器运行时间	ALB 进程已运行的时间（通常以秒为单位）
2. **连接信息**	当前客户端连接数	当前活跃的客户端连接数量
	读取客户端连接总数	累计从客户端读取请求的连接数
	写入客户端连接总数	累计向客户端写入响应的连接数
	已处理客户端连接总数	累计已处理完成的客户端连接总数
3. HTTP **虚拟主机监控**	请求总数	该虚拟主机接收的总请求数
	每秒请求量（RPS）	当前每秒处理的请求数
	1xx 错误码统计	所有 1xx 状态码（如 100, 101）的累计次数
	2xx 错误码统计	所有 2xx 成功响应（如 200, 201）的累计次数
	3xx 错误码统计	所有重定向响应（如 301, 302）的累计次数
	4xx 错误码统计	客户端错误（如 404, 403）的累计次数
	5xx 错误码统计	服务端错误（如 500, 502）的累计次数
	所有错误码统计	所有非 2xx 响应的总和（或全部状态码分类汇总）
	发送的总流量	该虚拟主机向客户端发送的总字节数
	接收的总流量	该虚拟主机从客户端接收的总字节数
	发送流量速率	当前每秒发送的字节数（B/s）
	接收流量速率	当前每秒接收的字节数（B/s）
4. HTTP **缓存监控**	缓存命中数	请求命中缓存的次数
	缓存未命中数	请求未命中缓存的次数
	缓存旁路数	因配置 bypass 而跳过缓存的请求次数
	缓存已过期数	缓存项已过期被丢弃的次数
	缓存失效数	因主动失效（如 purge）导致的缓存移除次数
	缓存更新数	缓存被更新（如 stale-while-revalidate）的次数

	重新验证的缓存数	需要向源站重新验证的缓存请求数
	缓存总数	缓存命中 + 未命中 + 旁路等的总和（或当前缓存项总数）
5. HTTP 上游服务器监控	服务状态	upstream server 当前状态（up/down/checking）
	服务权重值	配置的 weight 值
	最大失败次数	max_fails 配置值
	故障后停止服务的时间值	fail_timeout 配置值（单位：秒）
	请求总数	发往该 upstream 的总请求数
	每秒请求量	当前每秒发往该 upstream 的请求数
	1xx 错误码统计	所有 1xx 状态码的累计次数
	2xx 错误码统计	所有 2xx 成功响应的累计次数
	3xx 错误码统计	所有重定向响应的累计次数
	4xx 错误码统计	客户端错误的累计次数
	5xx 错误码统计	服务端错误的累计次数
	所有错误码统计	upstream 返回的所有状态码分类汇总
	发送的总流量	与该 upstream 之间的总发送字节数
	接收的总流量	与该 upstream 之间的总接收字节数
	发送流量速率	与 upstream 通信的当前发送速率（B/s）
	接收流量速率	与 upstream 通信的当前接收速率（B/s）
6. Stream 虚拟主机监控	请求总数	Stream（TCP/UDP）会话总数
	每秒请求数	当前每秒新建会话数
	连接成功/失败统计	⚠ Stream 层无 HTTP 状态码，此项统计连接建立成功或失败次数
	连接成功/失败统计	同上
	连接重定向统计	同上

	连接拒绝统计	同上
	服务端错误统计	同上
	所有错误码统计	可指连接异常、超时、拒绝等非 HTTP 错误
	发送的总流量	Stream 代理的总发送字节数
	接收的总流量	Stream 代理的总接收字节数
	发送流量速率	当前 Stream 发送速率（B/s）
	接收流量速率	当前 Stream 接收速率（B/s）
7. Stream 上游服务器监控	服务状态	Stream upstream 的健康状态
	服务权重值	配置的 weight 值
	最大失败次数	max_fails 配置值
	故障后停止服务的时间值	fail_timeout 配置值（单位：秒）
	请求总数	发往该 Stream upstream 的总连接数
	每秒请求数	当前每秒新建连接数
	连接异常统计	同 Stream 虚拟主机，通常指连接结果而非 HTTP 状态
	所有错误码统计	连接失败、超时等统计
	发送的总流量	与该 Stream upstream 的总发送字节数
	接收的总流量	与该 Stream upstream 的总接收字节数
	发送流量速率	当前发送速率（B/s）
	接收流量速率	当前接收速率（B/s）

5.0.3.2 vts模块配置和查看

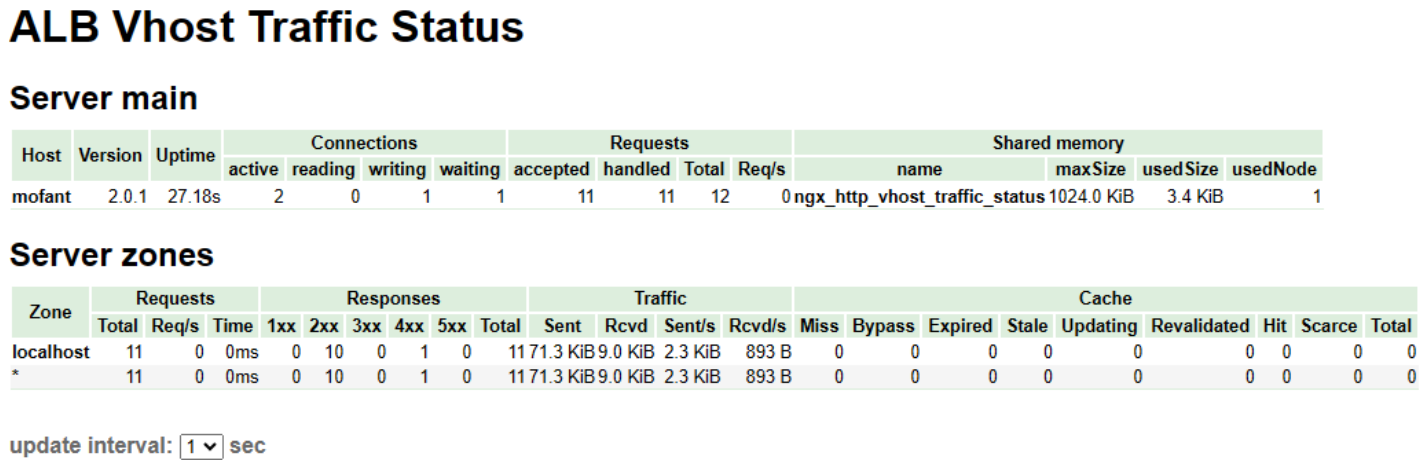
如下是开启配置vts的样例：

```
http {
    vhost_traffic_status_zone;                  # 启用vts模块

    ...
```

```
    server {

        ...

        location /status {                    # 配置vts查看的uri地址
            vhost_traffic_status_display;     # 当前/status的uri展示vts监控指标
            vhost_traffic_status_display_format html;   # 设置默认格式为html页面
        }
    }
}
```

- HTML监控数据查看

通过浏览器访问查看监控页面：`http://IP:Port/status`



ALB Vhost Traffic Status

Server main

Host	Version	Uptime	Connections				Requests				Shared memory			
			active	reading	writing	waiting	accepted	handled	Total	Req/s	name	maxSize	usedSize	usedNode
mofant	2.0.1	27.18s	2	0	1	1	11	11	12	0	ngx_http_vhost_traffic_status	1024.0 KiB	3.4 KiB	1

Server zones

Zone	Requests			Responses						Traffic				Cache								
	Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s	Miss	Bypass	Expired	Stale	Updating	Revalidated	Hit	Scarce	Total
localhost	11	0	0ms	0	10	0	1	0	11	71.3 KiB	9.0 KiB	2.3 KiB	893 B	0	0	0	0	0	0	0	0	0
*	11	0	0ms	0	10	0	1	0	11	71.3 KiB	9.0 KiB	2.3 KiB	893 B	0	0	0	0	0	0	0	0	0

update interval: 1 sec

- JSON数据格式查看

通过浏览器或者curl等获取JSON数据：`http://IP:Port/status/format/json`

- Prometheus监控指标数据接口

可以通过Prometheus直接集成：`http://IP:Port/status/format/prometheus`

5.0.3.3 自定义监控指标

`vhost_traffic_status_filter_by_set_key` 指令允许你指定一个键名（key name），VTS 会根据这个键名来过滤流量统计信息。通常，这个键名对应于 set 指令中定义的一个变量.

```
vhost_traffic_status_filter_by_set_key key_name;
# 配置区域： http, server, location
```

说明： 通过用户定义的变量启用键。键是用于计算流量的键字符串。名称是用于计算流量的组字符串。键和名称可以包含变量，例如 $host、$server_name。如果指定，则名称所属的组。如果没有指定第二个参数名称，则键所属的组。

如下实例：

```
http {
    vhost_traffic_status_zone;                  # 启用vts模块

    ...

    server {

        ...
        vhost_traffic_status_filter_by_set_key  $uri uri::*;   # 过滤uri，统计不同uri的流量情况
        location /status {                      # 配置vts查看的uri地址
            vhost_traffic_status_display;       # 当前/status的uri展示vts监控指标
            vhost_traffic_status_display_format html;   # 设置默认格式为html页面
        }
    }
}
```

配置完后，即可查看不同uri的流量情况：

ALB Vhost Traffic Status

Server main

Host	Version	Uptime	Connections				Requests				Shared memory			
			active	reading	writing	waiting	accepted	handled	Total	Req/s	name	maxSize	usedSize	usedNode
mofant	2.0.1	46.43s	2	0	1	1	123	123	122	1	ngx_http_vhost_traffic_status	1024.0 KiB	20.7 KiB	6

Server zones

Zone	Requests			Responses						Traffic				Cache								
	Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s	Miss	Bypass	Expired	Stale	Updating	Revalidated	Hit	Scarce	Total
localhost	121	1	0ms	0	117	0	4	0	121	547.0 KiB	95.6 KiB	8.2 KiB	899 B	0	0	0	0	0	0	0	0	0
*	121	1	0ms	0	117	0	4	0	121	547.0 KiB	95.6 KiB	8.2 KiB	899 B	0	0	0	0	0	0	0	0	0

Filters

uri::*

Zone	Requests			Responses						Traffic				Cache								
	Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s	Miss	Bypass	Expired	Stale	Updating	Revalidated	Hit	Scarce	Total
/status/	2	0	0ms	0	2	0	0	0	2	105.0 KiB	2.0 KiB	0 B	0 B	0	0	0	0	0	0	0	0	0
/index.html	6	0	0ms	0	6	0	0	0	6	5.0 KiB	6.1 KiB	0 B	0 B	0	0	0	0	0	0	0	0	0
/node_status	1	0	0ms	0	1	0	0	0	1	242 B	125 B	0 B	0 B	0	0	0	0	0	0	0	0	0
/status/format/json	39	1	0ms	0	39	0	0	0	39	238.7 KiB	34.1 KiB	8.2 KiB	899 B	0	0	0	0	0	0	0	0	0
/favicon.ico	3	0	0ms	0	0	0	3	0	3	2.1 KiB	2.9 KiB	0 B	0 B	0	0	0	0	0	0	0	0	0

update interval: 1 sec

JSON

5.0.3.4 stream监控

ALB-VTS模块支持stream模块的监控，但默认情况下不启用，需手动配置开启。其用法如下：

```
http {
    ...

    server {

        ...

        location /stream/status {      # 监控指标暴露uri
            stream_server_traffic_status_display;
            stream_server_traffic_status_display_format html;
        }
    }
}

stream {
    stream_server_traffic_status_zone;   # 在stream块中启用stream vts模
```

```
块

    ...

    server {
        ...
    }
}
```

- HTML监控数据查看

通过浏览器访问查看监控页面：`http://IP:Port/stream/status`

ALB Stream Traffic Status

Server main

Host	Version	Uptime	Connections				Requests				Shared memory			
			active	reading	writing	waiting	accepted	handled	Total	Req/s	name	maxSize	usedSize	usedNode
mofant	2.0.1	26.35s	2	0	1	1	457	457	1597	1	stream_server_traffic_status	1024.0 KiB	0 B	0

Server zones

Zone	Port	Protocol	Requests			Responses						Traffic			
			Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s
*	0	TCP	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B

Upstreams

tcp_backend

Server	State	Response Time			Weight	MaxFails	FailTimeout	Requests			Responses						Traffic			
		Connect	FirstByte	Response				Total	Req/s	Time	1xx	2xx	3xx	4xx	5xx	Total	Sent	Rcvd	Sent/s	Rcvd/s
172.21.32.106:9898	up	0ms	0ms	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B
172.21.32.107:9898	up	0ms	0ms	0ms	1	1	10	0	0	0ms	0	0	0	0	0	0	0 B	0 B	0 B	0 B

update interval: 1 sec

JSON

- JSON数据格式查看

通过浏览器或者curl等获取JSON数据：`http://IP:Port/stream/status/format/json`

- Prometheus监控指标数据接口

可以通过Prometheus直接集成：`http://IP:Port/stream/status/format/prometheus`

5.0.3.5 VTS监控指标grafana集成

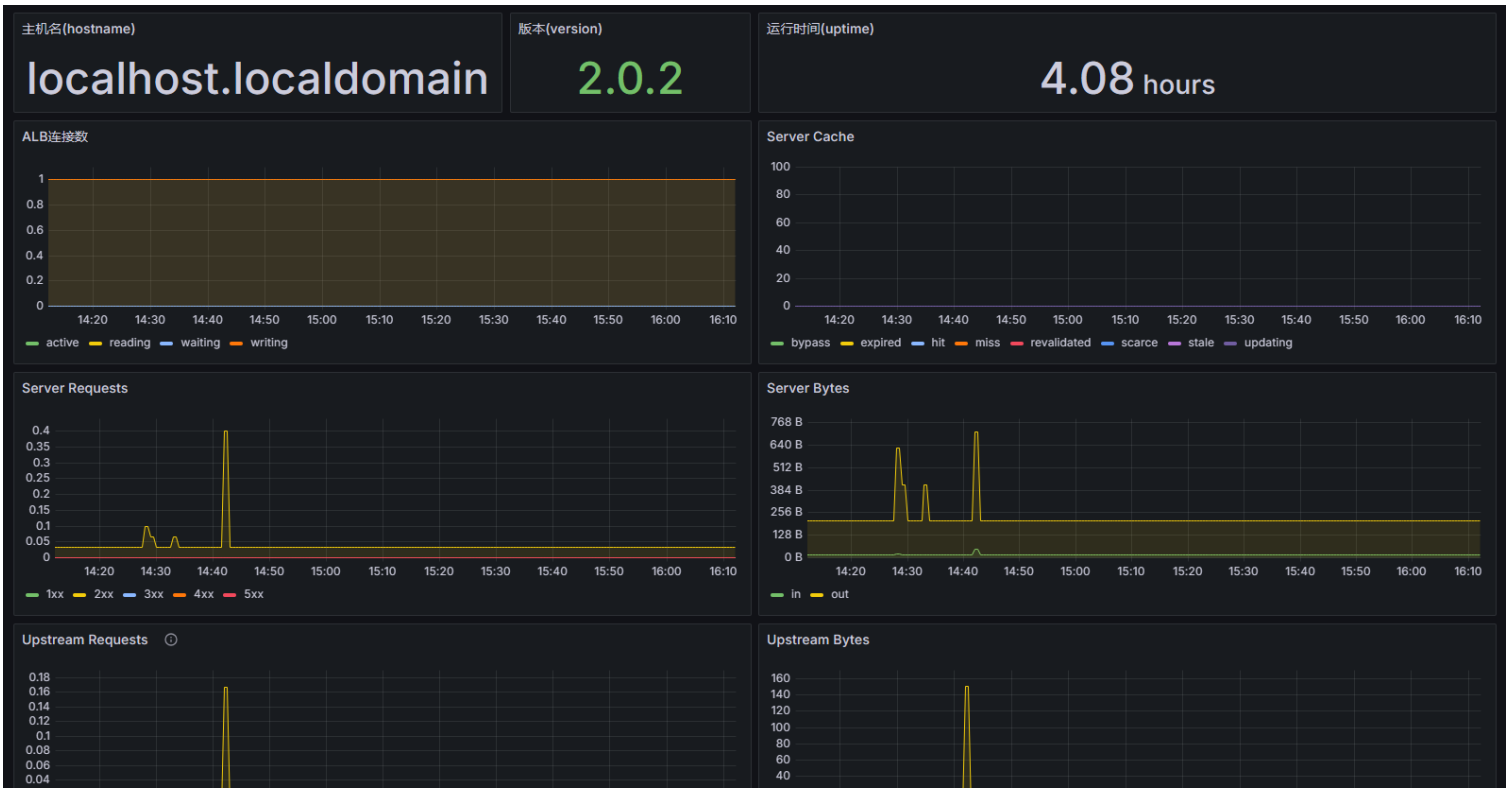
上面VTS监控指标提供prometheus采集格式，在prometheus中，可以配置如下采集：

```
scrape_configs:
- job_name: 'alb-vts-status'
  honor_timestamps: true
  scrape_interval: 1m
  scrape_timeout: 10s
  metrics_path: /status/format/prometheus # ALB vts监控指标采集url
  scheme: http
  enable_compression: true
  follow_redirects: true
  enable_http2: true
  static_configs:
  - targets:
      - 192.168.1.40:8080                 # ALB VTS 监控指标采集地址
    labels:
      group: 'alb'
      instance: 'alb'
```

其中grafana Dashboard的样例配置文件路径:

`ALB安装目录/utils/grafana/alb-grafana.json`

样例配置文件的展示效果如下：



可以在已部署的grafana中，导入该配置文件即可实现上述的监控仪表板。

6 TCP代理

ALB的stream模块可以用来做TCP代理，通过在alb配置文件中配置stream模块即可实现tcp代理。

如下的代理tcp示例：

```
# TCP代理
stream {
    upstream backend {
        server 127.0.0.1:8080;
    }

    server {
        listen 443;    # 监听端口

        proxy_pass backend;   # 转发到backend
    }
}

# 以下为HTTP配置内容
http {
  # 略
}
```

注： TCP代理配置只能写在最外层配置中，不能在写在http块或者在http块中通过include导入。

7 TCP代理加速

ALB TCP代理加速基于内核eBPF功能，在数据平面实现了高效的包处理和转发机制，显著提升了TCP代理的性能表现。

7.1 内核eBPF加速TCP转发优势

- 零拷贝转发：通过eBPF程序直接在内核态处理网络包，避免了传统方式中用户态和内核态之间的频繁数据拷贝。
- 智能路由决策：利用eBPF的高效匹配能力，快速确定数据包的转发路径，启用TCP代理加速。

7.1.1 配置要求

- 用户权限：
  - user 指令必须使用具有特权的用户（如 root、admin）
  - 或者如果内核支持，可将 /proc/sys/kernel/unprivileged_bpf_disabled 设置为 0 来使用非特权用户
  - **生产环境建议使用具有CAP_BPF能力的非特权用户，或配置内核参数允许非特权eBPF。**
- 连接数限制：
  - 连接总数不超过100000（10万）
- 硬件配置：
  - 内存： >= 16G
- 内核版本：
  - `kernel` 版本 >= 5.15
- ALB产品包要求：
  - 如 `ALB-V2.0.6-SE-20260421-amd64-ebpf.tar.gz` ，后缀携带ebpf特性。

7.1.2 使用示例

```
user root;  # 示例使用root，生产环境建议使用具有CAP_BPF能力的非特权用户


stream {
    upstream backend_servers {
```

```
        server 127.0.0.1:20001;
    }

    # 全局配置
    ebpf_proxy_timeout 600;   # 上下游之间无数据转发时的空闲超时时间(秒)
    ebpf_enable on;           # 全局启用 eBPF 加速

    server {
        listen 10000;
        ebpf_enable on;           # 在特定 server 中启用 eBPF (可覆盖全局设置)
        proxy_pass backend_servers;
    }

    server {
        listen 10001;
        ebpf_enable off;          # 禁用 eBPF 加速
        proxy_pass backend_servers;
    }
}
```

7.1.3 配置参数说明

7.1.3.1 ebpf_status_return

- 语法: `ebpf_status_return;`
- 上下文: server
- 作用: 返回简单的状态信息，用于调试目的
- 示例:

```
    server {
        listen 30001;
        ebpf_status_return;
    }
```

7.1.3.2 ebpf_enable

- 语法: `ebpf_enable on | off;`

- 默认值: off
- 上下文: main, server
- 作用: 启用或禁用 eBPF 加速功能
- 示例:

```
# 全局启用
ebpf_enable on;

server {
    listen 10000;
    # 在特定 server 中启用
    ebpf_enable on;
    proxy_pass backend;
}
```

7.1.3.3 ebpf_proxy_timeout

- 语法: `ebpf_proxy_timeout milliseconds;`
- 默认值: 600000 (单位毫秒，= 600秒)
- 上下文: main, server
- 作用: eBPF连接空闲超时时间，超过此时间无数据转发则断开连接。长连接场景请设置为0（永不超时）。
- 示例:

```
# 全局设置
ebpf_proxy_timeout 300000;

server {
    listen 10000;
    # 在 server 级别覆盖
    ebpf_proxy_timeout 60000;
    proxy_pass backend;
}
```

7.1.3.4 ebpf_proxy_buffer_size

- 语法: `ebpf_proxy_buffer_size size;`
- 默认值: 16384 (16KB)
- 上下文: main, server

- 作用: 设置用于存储从上游接收数据的缓冲区大小
- 示例:

```
# 全局设置
ebpf_proxy_buffer_size 32k;

server {
    listen 10000;
    # 在 server 级别设置
    ebpf_proxy_buffer_size 64k;
    proxy_pass backend;
}
```

7.1.3.5 ebpf_timer_period

- 语法: `ebpf_timer_period milliseconds;`
- 默认值: 2000 (2秒)
- 上下文: main, server
- 作用: 设置定时器周期，用于更新流量统计信息
- 示例:

```
# 全局设置
ebpf_timer_period 1000;

server {
    listen 10000;
    # 在 server 级别设置
    ebpf_timer_period 5000;
    proxy_pass backend;
}
```

8 TCP代理ADMQ

ALB支持使用TCP协议代理ADMQ的服务，下面是使用ALB标准版代理ADMQ服务的样例。

8.1 配置过程

1. 已部署ADMQ（略）
2. 部署ALB(略)
3. 修改ALB配置，支持ADMQ代理
4. 验证ADMQ服务代理

8.2 ADMQ环境和代理需求

- 假定目标ADMQ集群的三个服务节点为: 192.168.3.10、192.168.3.11、192.168.3.12，MQTT服务端口是5682。
- 假定ALB也同样开放5682端口对外提供ADMQ服务。

8.3 修改ALB配置，支持TCP代理ADMQ

编辑`ALB安装目录/conf/alb.conf`文件，在`http`配置之上最外层添加如下配置：

```
stream {
    upstream mq_server {
        server 192.168.3.10:5682 max_fails=3 fail_timeout=10s;
        server 192.168.3.11:5682 max_fails=3 fail_timeout=10s;
        server 192.168.3.12:5682 max_fails=3 fail_timeout=10s;
    }
    server {
        listen 5682 so_keepalive=on;    # ALB监听5682端口
        proxy_connect_timeout 15s;     # 连接超时时间
        proxy_timeout 300s;
        proxy_pass mq_server;           # ALB反向代理到三个mq节点
    }
}
```

```
# 以下是http配置内容，不需要改动
http {
    .... #略
}
```

8.4 验证ALB代理ADMQ服务

- 查看ALB监听端口5682是否存在。
- 若对应端口存在，使用MQ客户端通过ALB的IP和5682端口进行验证。

9 流式响应代理

ALB支持流式响应代理，即ALB不缓存响应，而是直接将响应转发给客户端，常用于各种AI/ChatGPT和对话机器人场景，下面是配置样例

```
location /streaming {
    proxy_pass http://backend_server;

    proxy_cache off; # 关闭缓存
    proxy_buffering off; # 关闭代理缓冲
    chunked_transfer_encoding on; # 开启分块传输编码
    tcp_nopush off; # 禁用TCP NOPUSH选项，允许快速发送小数据包，以减少延迟
    tcp_nodelay on; # 开启TCP NODELAY选项，禁止延迟ACK算法
    keepalive_timeout 300; # 设定keep-alive超时时间为300秒
}
```

9.0.1 配置详解

- `proxy_cache off`

> *关闭缓存功能，防止代理服务器缓存流式响应内容，确保客户端能够接收到实时、完整的响应。*

- `proxy_buffering off`

> *关闭代理服务器的响应缓冲，防止其缓冲整个响应后再发送给客户端，从而实现真正的流式传输效果。*

- `chunked_transfer_encoding on`

> *开启分块传输编码，允许将响应分成多个块进行传输。这是实现流式传输的关键。*

- `tcp_nopush off`

> *禁用TCP NOPUSH选项，允许快速发送小数据包。*

- `tcp_nodelay on`

> *开启TCP NODELAY选项，禁用延迟ACK算法。这有助于减少ACK包的延迟，确保数据及时发送。*

- `keepalive_timeout 300`

> *增加keepalive超时时间，防止在流式响应未完成时，代理与源服务器的连接就被关闭。这里设置为300秒，根据实际需求可以调整。*

10 日志配置

ALB支持两种类型的日志：访问日志（access logs）和错误日志（error logs）。下面详细介绍这两种日志的配置方法。

10.0.1 访问日志

访问日志记录了每个 HTTP 请求的信息，包括请求方法、请求 URI、响应状态码、响应大小、客户端 IP 地址等。

10.0.1.1 访问日志配置

访问日志可以通过 access_log 指令来配置。该指令可以出现在 http, server 或 location 块中。

10.0.1.2 基本配置样例

```
http {
    log_format main '$remote_addr - $remote_user [$time_local]
"$request" '
                    '$status $body_bytes_sent "$http_referer" '
                    '"$http_user_agent" "$http_x_forwarded_for"';

    server {
        listen 80;
        server_name example.com;

        access_log /var/log/alb/example.access.log main;
    }
}
```

- log_format: 定义日志条目的格式。上面的例子定义了一个名为 main 的格式
- access_log: 指定日志文件的位置和使用的格式。在这个例子中，日志文件被保存在 /var/log/alb/example.access.log，并且使用 main 格式。

10.0.1.3 日志格式详解

- $remote_addr: 客户端 IP 地址。
- $remote_user: 经过认证的用户名。
- $time_local: 本地时间戳。
- $request: 完整的请求行。
- $status: HTTP 状态码。
- $body_bytes_sent: 发送给客户端的响应主体大小。

- $http_referer: 请求的 Referer 请求头。
- $http_user_agent: 客户端的 User-Agent 请求头。
- $http_x_forwarded_for: X-Forwarded-For 请求头，通常用于反向代理环境。

10.0.1.4 关闭访问日志

如果你不想记录访问日志，可以将 access_log 设置为 off：

```
access_log off;
```

10.0.2 错误日志

错误日志记录了ALB运行过程中遇到的错误信息，这对于调试和维护非常重要。

10.0.2.1 配置错误日志

错误日志通过 error_log 指令来配置。该指令可以出现在 http, server 或 location 块中。

10.0.2.2 基本配置

```
http {
    error_log /var/log/alb/error.log warn;

    server {
        listen 80;
        server_name example.com;
    }
}
```

- error_log: 指定错误日志文件的位置和最小记录级别。在这个例子中，错误日志被保存在 /var/log/alb/error.log，并且只记录 warn 级别及以上的错误。

10.0.2.3 日志级别

错误日志的级别包括：

- debug: 调试信息。
- info: 一般信息。
- notice: 重要信息。
- warn: 警告信息。
- error: 错误信息。
- crit: 临界错误信息。

- alert: 警报信息。
- emerg: 紧急信息。

11 流量控制

ALB提供了多种方式来实现流量控制，包括限流、限速和限制连接。

11.0.1 使用 limit_req 模块进行限流

limit_req 模块允许你根据请求频率限制客户端的请求。

1、指令名称：`limit_req_zone`

- 语法：limit_req_zone key zone=name:size rate= number r/s
- 默认值：no
- 区域：http
- 使用示例：limit_req_zone $binary_remote_addr zone=addr:10m rate=1r/s
- 描述：定义一个限流区域，其中 $binary_remote_addr 是请求的客户端 IP 地址，zone=addr:10m 表示该区域的内存大小为 10MB，rate=1r/s 表示限制每个 IP 地址每秒只能发出一个请求。

2、指令名称：`limit_req`

- 语法：limit_req zone=name [burst=number] [nodelay]
- 默认值：no
- 区域：http, server, location
- 使用示例：limit_req zone=mylimit burst=5 nodelay
- 描述：在指定的区域中限制请求速率，其中 burst=5 表示允许的突发请求数量为 5 个，nodelay 表示不延迟处理超过 burst 的请求。

11.0.1.1 示例配置

假设你想限制每个 IP 地址每秒只能发出 1个请求：

```
http {
    limit_req_zone $binary_remote_addr zone=mylimit:10m rate=1r/s;

    server {
        listen 80;
        server_name example.com;

        location / {
            limit_req zone=mylimit burst=5 nodelay;
            proxy_pass http://backend;
        }
```

```
    }
}
```

- limit_req_zone: 定义一个限流区域，其中 $binary_remote_addr 是客户端 IP 地址，mylimit 是区域名称，10m 是内存大小（用于存储限流信息），rate=1r/s 表示每秒最多 1 个请求。
- limit_req: 在 location 块中应用限流规则。zone=mylimit 指定使用前面定义的限流区域，burst=5 允许突发请求，即在短时间内可以发送多于 1 个请求，但不能超过 burst 值；nodelay 表示不允许延迟请求。

11.0.2 使用 limit_conn 模块进行连接数限制

limit_conn 模块可以限制每个客户端的连接数。

1、指令名称：`limit_conn_zone`

- 语法：limit_conn_zone key zone=name:size;
- 默认值：no
- 区域：http
- 使用示例：limit_conn_zone $binary_remote_addr zone=addr:10m;
- 描述：定义一个限流区域，其中 key 是用于标识客户端的键（例如 IP 地址），zone=name 是区域名称，size 是内存大小（用于存储连接信息）。

2、指令名称：`limit_conn`

- 语法：limit_conn zone number;
- 默认值：no
- 区域：http, server, location
- 使用示例：limit_conn addr 10;
- 描述：在指定的区域中限制请求速率，其中 zone 是前面定义的区域名称，number 是允许的最大连接数。

11.0.2.1 示例配置

假设你想限制每个 IP 地址最多只能有 10 个并发连接：

```
http {
    limit_conn_zone $binary_remote_addr zone=myconn:10m;

    server {
        listen 80;
        server_name example.com;

        location / {
```

```
            limit_conn myconn 10;
            proxy_pass http://backend;
        }
    }
}
```

- limit_conn_zone: 定义一个连接数限制区域，其中 $binary_remote_addr 是客户端 IP 地址，myconn 是区域名称，10m 是内存大小（用于存储连接信息）。
- limit_conn: 在 location 块中应用连接数限制规则。zone=myconn 指定使用前面定义的连接数限制区域，10 表示最多允许 10 个并发连接。

11.0.3 使用 limit_rate 指令进行限速

limit_rate 指令可以限制客户端下载速度。

1、指令名称：`limit_rate`

- 语法：limit_rate rate;
- 默认值：no limit
- 区域：http, server, location
- 使用示例：limit_rate 100k;
- 描述：设置每个连接的下载速度限制，其中 rate 是速率（例如 100k 表示每秒传输 100KB 数据）。

11.0.3.1 示例配置

假设你想将每个客户端的下载速度限制为 100k/s：

```
server {
    listen 80;
    server_name example.com;

    location / {
        limit_rate 100k;
        proxy_pass http://backend;
    }
}
```

- limit_rate: 在 location 块中应用限速规则。100k 表示每秒最多传输 100KB 数据。

12 灰度发布和灰度测试

ALB支持灰度发布和灰度测试的功能，可以方便地对新版本进行测试，并平滑升级到生产环境。

12.1 场景介绍

场景	需求	典型例子
新功能灰度	只让内部/白名单用户体验	新下单接口只让 `uid=1000~2000` 的用户访问
AB 实验	按流量比例对比两套后端	50% 用户走 `v1` 服务，50% 走 `v2` 服务
地域/渠道灰度	按请求特征分流	广东用户走新机房，其他走旧机房
金丝雀发布	先放 5% 流量，错误率 >1% 自动回滚	监控 `5xx` 比例，超过阈值一键切回
UI/UX 改版测试	A/B 两组用户看到不同界面，评估点击率等指标	
故障回滚演练	快速切回旧版本，保障系统可用性	

12.2 ALB关键指令速查

ALB 本身不直接提供“灰度”功能，但可通过以下模块和指令组合实现：

- `map` 指令：根据请求特征（如 Cookie、Header、IP）生成变量
- `split_clients` 指令：基于哈希值按比例分流
- `upstream + server`：定义多个后端服务组

指令/变量	作用	示例
`split_clients`	按变量做**一致性哈希**分流	`split_clients "${remote_addr}AAA" $group { 5% canary; 95% stable; }`
`map`	读 header、cookie、arg 做**条件映射**	`map $cookie_ab_version $backend { default v1; canary v2; }`
`set $var` / `rewrite`	运行时改变量	`set $backend ''; rewrite ^/api/(.*) /$1 break;`
`proxy_pass`	把流量打到对应 upstream	`proxy_pass http://$backend;`

access_log	打日志方便回滚	access_log /var/log/alb/canary.log main if=$log_canary;

12.3 关键指令详解

12.3.1 1. split_clients —— 按比例分流（推荐）

```
split_clients ${variable} $backend {
    10%     backend_v2;
    *       backend_v1;
}
```

- `${variable}`：用于哈希的变量，如 $request_id、$remote_addr、$http_user_agent
- `$backend`：输出变量，用于 proxy_pass
- `10%`：流量比例，支持小数（如 0.5%）
- `*`：兜底，匹配剩余流量

✅ **优点**：简单、高效、无状态

❌ **缺点**：无法基于用户身份精准控制（除非变量是用户ID）

12.3.2 2. map —— 基于规则路由

```
map $http_x_version_flag $target_backend {
    default          backend_v1;
    "v2"             backend_v2;
    ~*beta           backend_v2;
}
```

- 根据请求头 `X-Version-Flag: v2` 路由到 v2
- 支持正则匹配（`~*` 表示不区分大小写）

✅ **优点**：灵活，可结合 Cookie、Header、IP 等

❌ **缺点**：需客户端配合传标识

12.3.3 3. upstream —— 定义后端服务组

```
upstream backend_v1 {
    server 192.168.1.100:8080;
}

upstream backend_v2 {
    server 192.168.1.101:8081;
}
```

每个 upstream 可包含多个 server，支持负载均衡 可配合 weight、backup 等参数

12.4 配置样例： 按IP地址和比例灰度发布

根据客户端 IP（$remote_addr）进行哈希计算，并将 约 5% 的 IP 分配到灰度组。

```
http {
    # 1. 定义 upstream
    upstream stable { server 10.0.0.100:8080; }
    upstream canary { server 10.0.0.200:8080; }

    # 2. 按 IP 做 5% 灰度
    split_clients "${remote_addr}AAA" $is_canary {
        5%      1;      # 变量值为 1 表示进灰度
        *       0;
    }

    # 3. 选后端
    map $is_canary $backend {
        1       canary;
        0       stable;
    }

    server {
        listen 80;
        server_name api.example.com;
```

```
        location / {
            proxy_set_header Host $host;
            proxy_pass http://$backend;
            # 打日志，方便回滚
            access_log /var/log/alb/canary.log combined
if=$is_canary;
        }
    }
}
```

12.5 基于Cookie的灰度

用户访问时带上 Cookie：gray=true，即可进入灰度环境。

```
# 若 Cookie 中有 gray=true，则走新版本
map $http_cookie $backend {
    default           old_app;
    ~*gray=true       new_app;
}

upstream old_app { server 10.0.0.10:8000; }
upstream new_app { server 10.0.0.11:8001; }

server {
    listen 80;
    location / {
        proxy_pass http://$backend;
    }
}
```

12.6 基于 IP 白名单强制灰度（内部测试解决方案）

```
upstream old_app { server 10.0.0.10:8000; }
```

```
upstream new_app { server 10.0.0.11:8001; }

map $remote_addr $backend {
    default           old_app;
    192.168.1.100     new_app;   # 内部测试 IP
    10.0.0.0/16       new_app;   # 整个内网段
}
```

12.7 动态控制灰度比例或基于数据库用户标签分流

- 从请求参数（如 ?user_id=123）或请求头（如 X-User-ID: 123）中读取用户 ID；
- 调用 Lua 脚本判断该用户是否属于灰度用户（例如：用户 ID 为偶数则进入新版本）；
- 根据判断结果，将请求代理到 old_app 或 new_app 后端服务；
- 支持日志记录灰度路由结果，便于监控和排查。

```
http {

  # 使用 map 在 http 块级别定义变量，确保 log_format 可访问
  map $arg_user_id $user_id_from_arg {
      default $arg_user_id;
      "" "";
  }

  map $http_x_user_id $user_id_from_header {
      default $http_x_user_id;
      "" "";
  }

  log_format gray_log '$remote_addr - [$time_local] '
                      '"$request" $status $body_bytes_sent '
                      '"$http_referer" "$http_user_agent" '
                      'user_id=$gray_user_id backend=$gray_backend';


  upstream old_app { server 10.0.0.10:8000; }
```

```
upstream new_app { server 10.0.0.11:8001; }

server {
  listen 80;
  server_name localhost;

  # 初始化变量，供 Lua 和日志使用
  set $gray_user_id "";
  set $gray_backend "";

  location / {
    # 使用 access_by_lua_block 在 access 阶段执行 Lua 逻辑
    access_by_lua_block {
        -- 1. 从 query string 或 header 获取 user_id
        local user_id = ngx.var.arg_user_id
        if not user_id or user_id == "" then
            user_id = ngx.var.http_x_user_id
        end

        -- 2. 设置日志变量
        ngx.var.gray_user_id = user_id or ""

        -- 3. 判断是否灰度用户（user_id 为偶数），无用户ID默认走旧版
        local backend = "old_app"
        if user_id and user_id ~= "" then
            local num = tonumber(user_id)
            if num and num % 2 == 0 then
                backend = "new_app"
            end
        end

        ngx.var.gray_backend = backend
    }

    # 4. 代理到对应后端
    proxy_pass http://$gray_backend;
```

```
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    # 访问日志包含灰度信息
    access_log logs/access.log gray_log;
  }
}
```

13 动态DNS解析

ALB 支持动态 DNS 解析，从而实现对域名的实时解析。该功能的核心是通过 resolver 指令配合变量来完成，示例如下：

```
http {
    server {
        listen 80;
        # 声明 DNS 服务器地址及解析结果的有效期（建议在 server 块级别配置，共享 DNS 缓存）
        resolver 8.8.8.8 valid=10s;   # 示例DNS服务器
        # 定义变量，用于动态解析 DNS
        set $test private.server1.com.cn;
        location / {
            # 使用变量实现动态 DNS 解析
            proxy_pass http://$test;
        }
    }
}
```

在上述配置中，当访问服务器的根路径 / 时，请求会被转发到 `$test` 变量所定义的后端地址 `http://private.server1.com.cn` 。由于使用了变量 `$test` ，ALB会在每次请求（或缓存过期后）通过 resolver 指定的 DNS 服务器（如 8.8.8.8）对该域名进行重新解析。

解析结果的有效期由 valid=10s 指定，即缓存 10 秒。超过有效期后，下次访问将触发新的 DNS 查询。

重要说明：如果 `proxy_pass` 后直接写的是域名（而非变量），例如 `proxy_pass http://private.server1.com.cn;` ，那么ALB仅在启动或重载配置时解析一次该域名，无法实现动态更新，因此不能达到动态 DNS 解析的效果。

13.1 resolver 配置详解

动态域名解析依赖于 `resolver` 指令与变量的结合使用。 `resolver` 可以在 `http` 、 `server` 块中配置，作用范围相应不同。建议将 resolver 放在 `server` 块级别，共享 DNS 缓存，避免每个 location 单独维护。

```
http {
    server {
```

```
        listen 80;
        # 建议在 server 块级别配置 resolver，共享 DNS 缓存
        resolver 8.8.8.8 valid=10s;  # 示例DNS服务器
        set   $test  private.server1.com.cn;
        location / {
            proxy_pass http://$test;
        }
        location /duplicate/ {
            # 复用 server 块的 resolver 配置，无需重复声明
            proxy_pass http://$test;
        }
    }
}
```

在此配置中，访问 `/` 和 `/duplicate/` 路径时，虽然代理的目标域名相同（均为 private.server1.com.cn），但它们分别使用不同的 DNS 服务器进行解析：

- `/` 使用 `8.8.8.8`
- `/duplicate/` 使用 `114.114.114.114`

并且，这两个路径的 DNS 解析结果相互独立，不会共享缓存。

参数说明：

- `valid=10s`：指定 DNS 解析结果的缓存有效期；
- `ipv6=on|off`：控制是否接受 IPv6 地址。默认为 on，即当域名同时解析出 IPv4 和 IPv6 地址时，两者都会被使用。可通过设置 ipv6=off 禁用 IPv6 解析。

14 HTTPS与国密

ALB通过设置SSL证书和私钥来实现安全的HTTP通信。

14.1 前置准备

- 获取SSL证书和私钥，支持国密SSL证书。
  - 1、服务端加密证书文件
  - 2、服务端加密证书私钥文件
  - 3、服务端签名证书（**国密**必须）
  - 4、服务端签名证书私钥文件（**国密**必须）

14.2 编辑ALB配置文件，开启SSL

- 配置说明：`conf/alb.conf` 文件中的server块内容。
- 核心配置：`listen`、`server_name`、`ssl_certificate`、`ssl_certificate_key`
- 示例配置：

```
server {
    listen 443 ssl;    # 使用SSL标记当前端口开启https
    server_name your_domain.com;    #  SSL证书对应的域名

    ssl_certificate /path/to/your_domain_name.enc.crt.pem;       # 服务端加密证书文件
    ssl_certificate_key /path/to/your_domain_name.enc.key.pem;   # 服务端加密证书私钥文件

    ssl_protocols TLSv1.2 TLSv1.3; # 使用更安全的协议版本
    ssl_prefer_server_ciphers on;   # 优先使用服务器定义的加密套件
    ssl_ciphers HIGH:!aNULL:!MD5;   # 使用更安全的加密算法

    location / {
        root /var/www/html;
        index index.html index.htm;
    }
}
```

- listen：指定监听的端口号，这里设置为 443，表示使用 HTTPS 协议。
- server_name：指定服务器的名称，这里设置为 your_domain.com，表示该服务器对应的域名。
- ssl_certificate：指定 SSL 证书的路径，这里设置为 /path/to/your_domain_name.crt，表示证书文件。
- ssl_certificate_key：指定 SSL 私钥的路径，这里设置为 /path/to/your_domain_name.key，表示私钥文件。
- ssl_protocols：指定支持的 SSL 协议版本，这里设置为 TLSv1.2 和 TLSv1.3，表示使用更安全的协议版本。
- ssl_prefer_server_ciphers：指定使用 stronger ciphers，表示使用更安全的加密算法。
- ssl_ciphers：指定支持的加密算法，这里设置为 HIGH:!aNULL:!MD5，表示使用更安全的加密算法。

14.2.1 配置指令说明

14.2.1.1 listen

- 描述：指定服务器监听的端口和协议。
- 示例： listen 443 ssl;

14.2.1.2 ssl_certificate 和 ssl_certificate_key

- 描述：指定 SSL 证书的路径和私钥的路径。
- 示例： ssl_certificate /path/to/your_domain_name.crt;

14.2.1.3 ssl_protocols

- 描述：指定支持的 SSL 协议版本。
- 示例： ssl_protocols TLSv1.2 TLSv1.3;（建议使用TLS1.2和TLS1.3）

14.2.1.4 ssl_ciphers

- 描述：指定 stronger ciphers 和支持的加密算法。
- 示例： ssl_ciphers HIGH:!aNULL:!MD5;（建议使用 stronger ciphers）

14.2.1.5 ssl_prefer_server_ciphers

- 描述：优先使用服务器定义的加密套件。
- 示例： ssl_prefer_server_ciphers on;

14.2.1.6 ssl_session_cache

- 描述：启用 SSL 会话缓存，以减少 SSL 连接的握手次数。
- 示例： ssl_session_cache shared:SSL:10m;

14.2.1.7 ssl_session_timeout

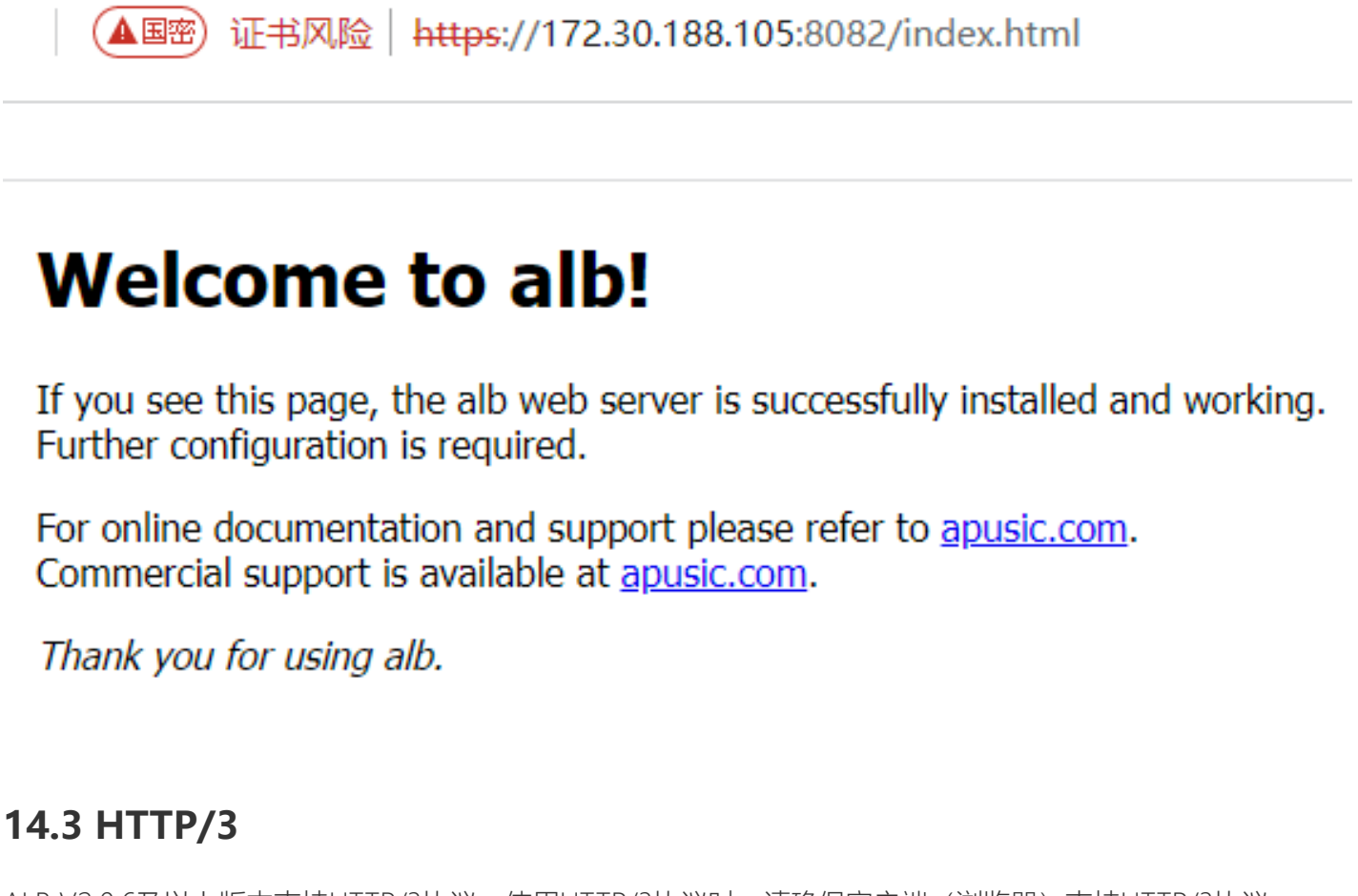
- 描述：设置 SSL 会话缓存的过期时间。
- 示例： ssl_session_timeout 10m;

14.2.2 国密配置样例

注： 国密证书和密钥文件命名中必须包含签名证书（sig）和服务端证书（enc），否则将无法识别。同时sig证书必须配置在前面，如下配置：

```
server {
    listen 443 ssl;    # 使用SSL标记当前端口开启https
    server_name your_domain.com;    #  SSL证书对应的域名

    # 国密服务端签名证书和密钥，一般名称中携带sig字样（sig证书必须先配置）
    ssl_certificate /path/to/your_domain_name.sig.crt.pem;       # 服务端签名证书文件
    ssl_certificate_key /path/to/your_domain_name.sig.key.pem;   # 服务端签名证书私钥文件

    # 国密服务端证书和密钥，一般名字中携带enc（enc证书放在sig证书后面）
    ssl_certificate /path/to/your_domain_name.enc.crt.pem;       # 服务端加密证书文件
    ssl_certificate_key /path/to/your_domain_name.enc.key.pem;   # 服务端加密证书私钥文件

    ssl_protocols TLSv1.2 TLSv1.3; # 使用更安全的协议版本
    ssl_prefer_server_ciphers on;   # 优先使用服务器定义的加密套件
    ssl_ciphers HIGH:!aNULL:!MD5;   # 使用更安全的加密算法

    location / {
        root /var/www/html;
        index index.html index.htm;
    }
}
```

使用国密浏览器访问，在地址栏标记**国密**。



14.3 HTTP/3

ALB V2.0.6及以上版本支持HTTP/3协议，使用HTTP/3协议时，请确保客户端（浏览器）支持HTTP/3协议。

为了保证和原有协议兼容，可以参考下面的HTTP/3配置：

```
server {
    listen 443 ssl;    # 使用SSL标记当前端口开启https
    listen 443 quic reuseport;    # 同一端口可同时监听SSL和QUIC
 (HTTP/3) , 需要ALB 1.25+支持

    server_name your_domain.com;    #  SSL证书对应的域名

    ssl_certificate /path/to/your_domain_name.crt;        # 证书文件
    ssl_certificate_key /path/to/your_domain_name.key;   # 私钥文件

    ssl_protocols TLSv1.3; # 使用更安全的协议版本，HTTP/3需要TLSv1.3
    ssl_prefer_server_ciphers on;   # 优先使用服务器定义的加密套件
    ssl_ciphers HIGH:!aNULL:!MD5;   # 使用更安全的加密算法
```

```
    # 告诉浏览器：同端口支持 HTTP/3，若不是443端口，请求修改和监听端口一致。
    add_header Alt-Svc 'h3=":443"; ma=86400';

    location / {
        root /var/www/html;
        index index.html index.htm;
    }
}
```

使用支持http3的curl工具测试：

```
 root@root:~$ /snap/bin/curl --http3-only -I
https://localhost:443/index.html -v -k
* Host localhost:443 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:443...
* error:8000006F:system library::Connection refused
* QUIC connect to ::1 port 443 failed: Could not connect to server
*   Trying 127.0.0.1:443...
* Server certificate:
*  subject: CN=localhost
*  start date: Aug 25 10:23:08 2025 GMT
*  expire date: Aug 25 10:23:08 2026 GMT
*  issuer: CN=localhost
*  SSL certificate verify result: self-signed certificate (18),
continuing anyway.
*   Certificate level 0: Public key type RSA (2048/112
Bits/secBits), signed using sha256WithRSAEncryption
* Connected to localhost (127.0.0.1) port 443
* using HTTP/3
* [HTTP/3] [0] OPENED stream for https://localhost:443/index.html
* [HTTP/3] [0] [:method: HEAD]
* [HTTP/3] [0] [:scheme: https]
* [HTTP/3] [0] [:authority: localhost:443]
* [HTTP/3] [0] [:path: /index.html]
```

```
* [HTTP/3] [0] [user-agent: curl/8.15.0]
* [HTTP/3] [0] [accept: */*]
> HEAD /index.html HTTP/3
> Host: localhost:443
> User-Agent: curl/8.15.0
> Accept: */*
>
* Request completely sent off
< HTTP/3 200
HTTP/3 200
< server: alb/2.0.6
server: alb/2.0.6
< date: Tue, 26 Aug 2025 08:37:04 GMT
date: Tue, 26 Aug 2025 08:37:04 GMT
< content-type: text/html
content-type: text/html
< content-length: 617
content-length: 617
< last-modified: Fri, 22 Aug 2025 08:05:21 GMT
last-modified: Fri, 22 Aug 2025 08:05:21 GMT
< etag: "68a824c1-269"
etag: "68a824c1-269"
< alt-svc: h3=":443"; ma=86400
alt-svc: h3=":443"; ma=86400
< accept-ranges: bytes
accept-ranges: bytes
<

* Connection #0 to host localhost left intact
```

15 正向代理

ALB可以用作正向代理服务器，它可以帮助客户端访问其他服务器或服务，同时隐藏客户端的真实 IP 地址。正向代理的主要用途包括缓存、负载均衡、安全过滤等。

15.1 示例配置

以下是一个简单的 ALB 配置示例，支持http和https正向代理，用于将请求转发到另一个服务器：

```
server {
    listen                          8080;
    server_name                     localhost;
    resolver                        114.114.114.114 ipv6=off;
    proxy_connect on;
    proxy_connect_allow             443 80;
    location / {
        proxy_pass $scheme://$http_host$request_uri;
    }
}
```

- resolver: 设置 DNS 解析服务器，这里使用的是 114.114.114.114。
- proxy_connect: 启用代理连接功能。
- proxy_connect_allow: 允许的端口号，这里是 443 和 80。

在 location 块中，使用 $scheme、$http_host 和 $request_uri 变量来构造目标服务器的地址。

- location /: 配置处理所有请求的路径，将请求转发到目标服务器。
- $scheme: 表示请求使用的协议（http 或 https）。
- $http_host: 表示请求的主机名和端口号。
- $request_uri: 表示请求的 URI。

16 缓存配置

ALB 支持代理缓存（Proxy Cache）功能，将后端响应内容缓存在本地磁盘，缓存有效期内由 ALB 直接返回响应，可显著降低后端压力、缩短响应延迟。常用于商品详情页、新闻文章、API 查询结果等"读多写少"的场景。

16.1 基础配置

```
http {
    # 1. 定义缓存存储路径与共享内存区（必须位于 http 上下文）
    proxy_cache_path /var/cache/alb/proxy levels=1:2
keys_zone=my_cache:10m
                     max_size=10240m inactive=60m use_temp_path=off;

    upstream backend {
        server 10.0.0.10:8080;
    }

    server {
        listen 80;
        server_name cache.example.com;

        location / {
            # 2. 在 location 中启用缓存
            proxy_cache my_cache;
            proxy_cache_valid 200 302 10m;
            proxy_cache_valid 404 1m;
            proxy_cache_key "$scheme$proxy_host$request_uri";
            proxy_cache_bypass $http_cache_control;
            proxy_cache_use_stale error timeout http_500 http_502
http_503 http_504;
            add_header X-Cache-Status $upstream_cache_status;

            proxy_pass http://backend;
        }
```

```
    }
}
```

16.2 关键指令说明

指令	上下文	作用
`proxy_cache_path`	http	定义缓存目录、目录层级、共享内存区名称与大小
`levels=1:2`	proxy_cache_path	目录层级，影响单目录文件数
`keys_zone=name:size`	proxy_cache_path	共享内存区名称与大小，1m 可存储约 8000 个 key
`max_size=10g`	proxy_cache_path	缓存总容量上限，超出时按 LRU 淘汰
`inactive=60m`	proxy_cache_path	缓存项在指定时间内未被访问则删除
`proxy_cache`	http/server/location	启用缓存并指定 keys_zone 名称
`proxy_cache_valid`	http/server/location	针对不同响应状态码设置缓存时间
`proxy_cache_key`	http/server/location	缓存键，默认 `$scheme$proxy_host$request_uri`
`proxy_cache_bypass`	http/server/location	指定条件下跳过读缓存
`proxy_cache_use_stale`	http/server/location	后端出错时使用过期缓存
`proxy_cache_min_uses`	http/server/location	同一资源被请求多少次后才缓存
`proxy_cache_lock`	http/server/location	并发请求同一资源时只允许一个请求回源
`proxy_cache_revalidate`	http/server/location	启用 If-Modified-Since 条件请求更新缓存
`$upstream_cache_status`	—	缓存状态变量：HIT/MISS/EXPIRED/STALE/BYPASS/UPDATING/REVALIDATED

16.3 真实使用场景

16.3.1 场景一：商品详情页缓存

电商商品详情页 80% 的访问集中在"商品已上架"状态，内容变动频率低：

```
proxy_cache_path /cache/goods levels=1:2 keys_zone=GOODS:50m
max_size=20g inactive=30m;
```

```
location /goods/detail {
    proxy_cache GOODS;
    proxy_cache_valid 200 5m;
    proxy_cache_key "$scheme$proxy_host$request_uri";   # 建议包含多维度信息，避免仅按商品ID缓存导致冲突
    proxy_cache_min_uses 2;
    proxy_cache_bypass $arg_nocache;
    proxy_cache_use_stale error timeout;
    add_header X-Cache $upstream_cache_status;
    proxy_pass http://goods_service;
}
```

16.3.2 场景二：API 网关响应缓存 + 后端降级

对外提供只读查询接口，后端故障时仍可返回过期数据：

```
proxy_cache_path /cache/api levels=1:2 keys_zone=API:20m max_size=5g inactive=10m;

location /api/v1/products {
    proxy_cache API;
    proxy_cache_valid 200 1m;
    proxy_cache_valid any 5s;
    proxy_cache_use_stale error timeout invalid_header updating http_500 http_502 http_503 http_504;
    proxy_cache_lock on;
    proxy_cache_revalidate on;
    proxy_pass http://api_backend;
}
```

16.4 注意事项

- 缓存目录建议使用 SSD，避免 IO 瓶颈；
- 开启 `use_temp_path=off` 可减少文件复制开销；
- 含 `Set-Cookie` 的响应不会被缓存；
- 涉及用户个人数据（购物车、订单、登录态）的接口**禁止**使用代理缓存。

17 会话保持

会话保持（Session Persistence）指将同一客户端的请求始终路由到同一台后端服务器，常用于购物车、登录态、WebSocket 长连接等有状态场景。ALB 提供多种会话保持方式，可按业务需求选择。

17.1 方式一：基于 IP 哈希

基于客户端 IP 做哈希计算，相同 IP 始终分配到同一后端。配置最简单，但无法应对 NAT/代理环境（多用户共享同一 IP）。

```
upstream backend {
    ip_hash;
    server 10.0.0.10:8080;
    server 10.0.0.11:8080;
}
```

17.2 方式二：基于 Cookie 一致性哈希（推荐）

使用 `hash $cookie_xxx consistent` 指令，按用户 Cookie 做一致性哈希。即使客户端 IP 变化（如切换网络），只要 Cookie 不变，会话一致性就得以保持。

```
upstream backend {
    hash $cookie_session_id consistent;
    server 10.0.0.10:8080;
    server 10.0.0.11:8080;
    server 10.0.0.12:8080;
}
```

17.3 方式三：基于 Lua + Redis 自定义会话保持

通过 Lua 脚本访问外部存储（Redis）查询用户的会话绑定关系，适合需要跨节点共享会话状态、或需要结合业务规则的场景。

```
upstream backend_pool {
    server 10.0.0.10:8080;
```

```
    server 10.0.0.11:8080;
    server 10.0.0.12:8080;
}

server {
    listen 80;
    location / {
        set $backend "";
        access_by_lua_block {
            local redis = require "resty.redis"
            local red = redis:new()
            red:set_timeout(1000)
            local ok, err = red:connect("127.0.0.1", 6379)
            if not ok then
                local hash = ngx.var.remote_addr
                local servers = {"10.0.0.10:8080", "10.0.0.11:8080",
"10.0.0.12:8080"}
                ngx.var.backend = servers[(string.byte(hash, 1) %
#servers) + 1]
                return
            end
            local sid = ngx.var.cookie_session_id
            if not sid or sid == "" then
                red:close()
                ngx.var.backend = "10.0.0.10:8080"
                return
            end
            local backend = red:get("session:" .. sid)
            if not backend or backend == ngx.null then
                local servers = {"10.0.0.10:8080", "10.0.0.11:8080",
"10.0.0.12:8080"}
                backend = servers[(ngx.var.remote_addr:byte() %
#servers) + 1]
                red:set("session:" .. sid, backend, "EX", 3600)
            end
            red:close()
```

```
            ngx.var.backend = backend
        }
        proxy_pass http://$backend;
    }
}
```

17.4 方式四：基于 sticky 模块（推荐）

sticky 是 ALB 内置的会话保持模块，随产品一起发布，无需额外部署即可使用。sticky 支持 cookie、route、learn 三种工作模式，在 `upstream` 块内通过 `sticky` 指令启用。

17.4.1 1. cookie 模式

由 ALB 主动创建 Cookie 记录后端节点，客户端首次访问时分配后端并下发 Cookie，后续请求携带 Cookie 即可路由到相同节点。

```
upstream backend {
    sticky cookie srv_id expires=1h domain=.example.com path=/
httponly secure;
    server 10.0.0.10:8080;
    server 10.0.0.11:8080;
}
```

参数说明：

参数	必选	说明
`srv_id`	是	Cookie 名称，可自定义为业务相关名称（如 `JSESSIONID`、`route_id`）
`expires=1h`	否	Cookie 有效期，浏览器关闭后失效（会话级）
`domain`	否	Cookie 生效域名（如 `.example.com` 覆盖子域）
`path`	否	Cookie 生效路径，默认 `/`
`secure`	否	仅在 HTTPS 连接下传输，防止明文泄露
`httponly`	否	禁止 JavaScript 读取，缓解 XSS 攻击

17.4.2 2. route 模式

按顺序读取请求中的变量（如 Cookie、Header、参数），使用第一个非空变量作为路由键。适合由上游应用在登录时下发业务标识 Cookie 的场景。

```
upstream backend {
    sticky route $cookie_session_id $cookie_user_id $http_x_session;
    server 10.0.0.10:8080;
    server 10.0.0.11:8080;
}
```

按 `$cookie_session_id` → `$cookie_user_id` → `$http_x_session` 顺序查找，找到第一个非空值作为路由键。

17.4.3 3. learn 模式

从上游响应的 `Set-Cookie` 中学习会话标识。ALB 不主动创建 Cookie，而是透传上游下发的 Session Cookie。适合已经使用 Session 机制的 Java/PHP 等后端应用。

```
upstream backend {
    sticky learn create=$upstream_cookie_JSESSIONID
lookup=$cookie_JSESSIONID
                expires=1h;
    server 10.0.0.10:8080;
    server 10.0.0.11:8080;
}
```

- `create=$upstream_cookie_JSESSIONID`：从上游响应的 `Set-Cookie` 头中提取 `JSESSIONID` 写入变量
- `lookup=$cookie_JSESSIONID`：从客户端请求 Cookie 中读取 `JSESSIONID` 作为路由键
- 上游应用负责登录时写入 Session Cookie（如 Java 的 `JSESSIONID`），ALB 自动识别并保持会话

17.5 关键指令说明

指令/变量	作用
`ip_hash`	启用基于客户端 IP 的哈希分配
`hash $key consistent`	基于任意变量（Cookie、Header、参数）的一致性哈希
`sticky cookie`	由 ALB 创建会话 Cookie 进行粘性会话

`sticky route`	按请求变量（Cookie、Header）作为路由键
`sticky learn`	从上游 Set-Cookie 中学习会话标识
`$cookie_xxx`	读取指定名称的 Cookie 值
`access_by_lua_block`	在请求访问阶段执行 Lua 代码，常用于动态选择后端
`lua-resty-redis`	ALB 内置的 Redis 客户端库（Lua 模块），可用于会话存储

17.6 真实使用场景

17.6.1 场景一：电商登录态保持

用户登录后写入 `session_id` Cookie，后续请求根据该 Cookie 路由到同一后端，避免购物车丢失：

```
upstream user_backend {
    hash $cookie_session_id consistent;
    server 10.0.0.10:8080;
    server 10.0.0.11:8080;
}

server {
    listen 80;
    server_name shop.example.com;
    location / {
        proxy_pass http://user_backend;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

17.6.2 场景二：WebSocket 长连接保持

WebSocket 连接建立后会被绑定到特定后端，必须保证同一用户始终连接到同一后端，否则需要重建连接：

```
upstream ws_backend {
    ip_hash;
    server 10.0.0.10:8080;
    server 10.0.0.11:8080;
```

```
}

server {
    listen 80;
    location /ws/ {
        proxy_pass http://ws_backend;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
}
```

17.6.3 场景三：Java 后端 sticky 模式无缝接入

后端为 Java 应用（Spring Boot 等）已使用 `JSESSIONID`，ALB 启用 sticky learn 模式即可自动识别上游 Session Cookie，无需改造业务：

```
upstream java_backend {
    sticky learn create=$upstream_cookie_JSESSIONID
lookup=$cookie_JSESSIONID
                expires=1h;
    server 10.0.0.10:8080;
    server 10.0.0.11:8080;
    server 10.0.0.12:8080;
}

server {
    listen 80;
    server_name java.example.com;
    location / {
        proxy_pass http://java_backend;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

17.7 注意事项

- 启用 `consistent` 后端扩容/缩容时，仅影响部分用户的会话，迁移成本低；
- 基于 Cookie 的方式要求后端应用在登录成功时设置统一的 session_id Cookie；
- Lua + Redis 方案需要额外部署 Redis 服务并保证其高可用；
- sticky cookie 模式下，Cookie 名称应避免与后端应用 Session Cookie 冲突（例如 Java 应用同时使用 `JSESSIONID` 时不要将 sticky Cookie 也命名为 `JSESSIONID`）；
- sticky learn 模式要求上游响应中 `Set-Cookie` 头必须携带会话标识，常见 Java 框架默认即符合；
- sticky 模块通过 Cookie 工作，需保证客户端启用 Cookie。

18 压缩

ALB 支持对响应内容进行压缩，减小传输体积、加快页面加载速度。支持 Gzip 与 Brotli 两种算法：Gzip 兼容性最好（所有浏览器都支持），Brotli 压缩率更高但需要 HTTPS 协议且仅 Chrome/Firefox/Edge 等现代浏览器支持。

18.1 Gzip 压缩

```
http {
    gzip on;
    gzip_min_length 1024;
    gzip_comp_level 6;
    gzip_vary on;
    gzip_proxied any;
    gzip_types
        text/plain
        text/css
        text/xml
        application/json
        application/javascript
        application/xml
        application/xml+rss
        image/svg+xml
        font/ttf
        font/otf;

    server {
        listen 80;
        server_name www.example.com;
        root /var/www/html;
    }
}
```

18.2 关键指令说明

指令	默认值	作用
`gzip`	off	是否启用 Gzip 压缩
`gzip_min_length`	20	响应体小于该值不压缩（字节）
`gzip_comp_level`	1	压缩级别 1-9，建议 4-6
`gzip_vary`	off	是否在响应头添加 `Vary: Accept-Encoding`
`gzip_proxied`	off	对上游响应启用压缩的条件
`gzip_types`	text/html	需要压缩的 MIME 类型（text/html 始终压缩）
`gzip_disable`	—	通过正则匹配禁用压缩
`gzip_buffers`	32 4k	压缩缓冲区数量与大小
`gzip_http_version`	1.1	启用压缩的最低 HTTP 版本

18.3 Brotli 压缩（需 ALB 编译 alb_brotli 模块）

Brotli 算法相比 Gzip 压缩率可提升 15-25%，特别适合 JS/CSS/HTML 等文本资源。

```
http {
    brotli on;
    brotli_comp_level 6;
    brotli_min_length 1024;
    brotli_types
        text/plain
        text/css
        text/xml
        application/json
        application/javascript
        application/xml
        image/svg+xml;

    server {
        listen 443 ssl;
```

```
    }
}
```

18.4 预压缩文件（gzip_static / brotli_static)

如果后端在构建阶段已经生成了 `.gz` / `.br` 预压缩文件，可使用 `gzip_static on` 让 ALB 直接发送预压缩文件，跳过实时压缩过程，可显著降低 CPU 占用。

```
http {
    gzip_static on;
    brotli_static on;
    gzip_proxied any;
    gzip on;
}
```

预压缩文件命名规则（以 `index.html` 为例）：

- 原始文件：`index.html`
- Gzip 预压缩：`index.html.gz`
- Brotli 预压缩：`index.html.br`

18.5 真实使用场景

18.5.1 场景一：API 响应压缩

JSON 接口在移动端网络下体积较大，开启 Gzip 后通常可减少 60-80% 体积：

```
http {
    gzip on;
    gzip_min_length 256;
    gzip_types application/json application/javascript text/css;

    server {
        location /api/ {
            gzip on;
            gzip_types application/json;
            proxy_pass http://api_backend;
```

```
        }
    }
}
```

18.5.2 场景二：静态资源双压缩（Gzip + Brotli）

同时启用 Gzip 与 Brotli，通过 `Vary: Accept-Encoding` 头让浏览器和 CDN 缓存按压缩算法分别存储：

```
http {
    gzip on;
    gzip_static on;
    brotli on;
    brotli_static on;
    gzip_vary on;

    server {
        listen 80;
        root /var/www/static;
        location / {
            add_header Vary "Accept-Encoding";
        }
    }
}
```

18.6 注意事项

- 已被压缩的资源（jpg、png、mp4、zip 等）开启压缩会浪费 CPU，应避免在 `gzip_types` 中列出；
- CPU 资源紧张时优先使用 `gzip_static` 跳过实时压缩；
- 启用 Brotli 必须使用 HTTPS（浏览器仅在 TLS 下发送 `br` 标记）；
- 高并发场景下建议将 `gzip_comp_level` 设置为 4-5，避免 CPU 成为瓶颈。

19 gRPC 代理

ALB 支持 gRPC 代理，可作为微服务架构中 gRPC 服务集群的统一入口。gRPC 基于 HTTP/2 协议传输，ALB 通过 `grpc_pass` 指令实现 gRPC 请求的反向代理。

19.1 基础配置

```
upstream grpc_backend {
    server 10.0.0.10:50051;
    server 10.0.0.11:50051;
}

server {
    listen 80 http2;          # 必须启用 http2（生产环境建议使用 443 端口并启用 TLS）
    server_name grpc.example.com;

    location / {
        grpc_pass grpc://grpc_backend;

        # 将上游错误转换为 gRPC 状态码
        error_page 502 = /grpc_error;
        error_page 503 = /grpc_error;
        error_page 504 = /grpc_error;

        # gRPC 健康检查见 "健康检查 -> 七层HTTP代理健康检查" 章节
    }

    location = /grpc_error {
        internal;
        default_type application/grpc;
        add_header grpc-status 14;          # 14 = UNAVAILABLE
        add_header content-length 0;
        return 204;
```

```
    }
}
```

19.1.1 HTTPS 加密代理

生产环境推荐使用 HTTPS 加密传输 gRPC 流量：

```
server {
    listen 443 ssl http2;
    server_name grpc.example.com;

    ssl_certificate     /path/to/server.crt;
    ssl_certificate_key /path/to/server.key;
    ssl_protocols       TLSv1.2 TLSv1.3;

    location / {
        grpc_pass grpcs://grpc_backend;      # grpcs 表示 HTTPS 加密

        # 透传客户端元数据
        grpc_set_header X-Request-Id $request_id;
        grpc_set_header X-Real-IP    $remote_addr;
    }
}

upstream grpc_backend {
    server 10.0.0.10:50051;
    server 10.0.0.11:50051;
}
```

19.2 关键指令说明

指令	上下文	作用
`grpc_pass grpc://upstream`	location	代理 gRPC 流量到上游（明文）
`grpc_pass grpcs://upstream`	location	代理 gRPC 流量到上游（TLS 加密）

`grpc_set_header`	location	设置/重写传递给上游的 gRPC 元数据头
`error_page 502 = /grpc_error`	http/server/location	将 HTTP 错误转换为 gRPC 状态码
`grpc-status 14`	—	gRPC 状态码：14 = UNAVAILABLE，常用于后端不可达
`listen ... http2`	server	启用 HTTP/2，gRPC 必需

19.3 真实使用场景

19.3.1 场景一：微服务 gRPC 集群统一入口

微服务架构中，多个 gRPC 服务由 ALB 统一对外暴露入口：

```
upstream user_service {
    server 10.0.0.10:50051;
    server 10.0.0.11:50051;
}

upstream order_service {
    server 10.0.0.20:50051;
    server 10.0.0.21:50051;
}

server {
    listen 443 ssl http2;
    server_name api.example.com;

    ssl_certificate     /etc/alb/certs/api.crt;
    ssl_certificate_key /etc/alb/certs/api.key;

    # 用户服务
    location /user.UserService/ {
        grpc_pass grpcs://user_service;
    }
```

```
    # 订单服务
    location /order.OrderService/ {
        grpc_pass grpcs://order_service;
    }
}
```

19.3.2 场景二：gRPC 反射与健康检查

上游 gRPC 服务启用 gRPC Reflection 与 Health Checking 协议后，可结合 `check` 指令做主动健康检查：

```
upstream grpc_backend {
    server 10.0.0.10:50051;
    server 10.0.0.11:50051;

    # type=http 即可，gRPC 复用 HTTP 健康检查机制
    check interval=3000 rise=2 fall=3 timeout=2000 type=http;

    # gRPC 健康检查路径（使用 HTTP/1.1，ALB 健康检查模块当前不支持 HTTP/2.0 请求）
    check_http_send "POST /grpc.health.v1.Health/Check HTTP/1.1\r\nHost: grpc.example.com\r\nContent-Type: application/grpc\r\n\r\n";
    check_http_expect_alive http_2xx;
}
```

19.4 注意事项

- gRPC 代理**必须**启用 HTTP/2（`listen 443 ssl http2;` 或 `listen 80 http2;`）；
- HTTP 错误必须通过 `error_page` 转换为 gRPC 状态码，否则客户端将得到无意义的 HTTP 错误；
- gRPC 客户端对代理超时非常敏感，需结合上游 RTT 合理设置 `grpc_connect_timeout`、`grpc_send_timeout`、`grpc_read_timeout`；
- ALB 与上游 gRPC 服务之间建议使用 TLS（`grpcs://`）加密传输。

20 WebSocket 代理

ALB 支持 WebSocket 代理，可将客户端的 WebSocket 连接反向代理到后端 WebSocket 服务。WebSocket 在客户端（浏览器）与 ALB、ALB 与上游服务之间分别建立 HTTP 升级握手，握手成功后切换为双向长连接。

20.1 基础配置

```
upstream ws_backend {
    server 10.0.0.10:8080;
    server 10.0.0.11:8080;
    ip_hash;              # WebSocket 长连接建议保持会话粘性，但生产环境建议使用 sticky cookie 或基于 cookie 的一致性哈希
}

server {
    listen 80;
    server_name ws.example.com;

    location /ws/ {
        proxy_pass http://ws_backend;

        # WebSocket 升级必备头
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";

        # 长连接超时（默认 60s 过短）
        proxy_read_timeout 3600s;         # 1 小时无数据断开
        proxy_send_timeout 3600s;

        # 透传客户端真实信息
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
```

```
    }
}
```

20.2 与 HTTPS 共用 443 端口

WebSocket 协议可通过 `Upgrade: websocket` 在 HTTPS 上承载，无需单独开放端口：

```
server {
    listen 443 ssl;
    server_name ws.example.com;

    ssl_certificate     /etc/alb/certs/ws.crt;
    ssl_certificate_key /etc/alb/certs/ws.key;

    # 同一 server 块下，location 可同时处理 HTTPS 和 WebSocket
    location /ws/ {
        proxy_pass http://ws_backend;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_read_timeout 3600s;
    }

    location / {
        # 普通 HTTPS 请求
        proxy_pass http://backend;
    }
}
```

20.3 关键指令说明

指令	作用
`proxy_http_version 1.1`	必须设置，WebSocket 升级基于 HTTP/1.1
`proxy_set_header Upgrade $http_upgrade`	透传客户端 Upgrade 头（`$http_upgrade` 默认值即为空）

`proxy_set_header Connection "upgrade"`	固定设置为 `upgrade`，通知上游建立长连接
`proxy_read_timeout`	读空闲超时（WebSocket 静默期），默认 60s 极易断开
`proxy_send_timeout`	写空闲超时
`proxy_buffering off`	关闭响应缓冲，适合实时双向通信
`ip_hash` / `sticky`	WebSocket 长连接需配合会话保持，否则节点变更会强制重连

20.4 真实使用场景

20.4.1 场景一：在线客服系统 WebSocket 代理

客服系统 WebSocket 推送用户消息、客服回复、工单状态变更：

```
upstream chat_backend {
    ip_hash;              # 同一用户始终连接同一后端
    server 10.0.0.10:8080;
    server 10.0.0.11:8080;
    server 10.0.0.12:8080;
}

server {
    listen 443 ssl;
    server_name support.example.com;

    ssl_certificate     /etc/alb/certs/support.crt;
    ssl_certificate_key /etc/alb/certs/support.key;

    location /ws/chat {
        proxy_pass http://chat_backend;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_read_timeout 86400s;       # 24 小时静默超时
        proxy_send_timeout 86400s;
        proxy_buffering off;
```

```
    }
}
```

20.4.2 场景二：实时数据推送（股票行情、K 线）

金融行情推送要求毫秒级延迟，关闭所有缓冲并扩大超时：

```
upstream market_backend {
    server 10.0.0.10:8080;
    server 10.0.0.11:8080;
}

server {
    listen 443 ssl;
    server_name market.example.com;

    location /ws/quote {
        proxy_pass http://market_backend;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";

        # 实时性优化
        proxy_buffering off;
        proxy_cache off;
        tcp_nodelay on;
        tcp_nopush off;
        keepalive_timeout 300s;

        # 长连接超时
        proxy_read_timeout 7d;            # 一周，避免长时间静默断开
        proxy_send_timeout 7d;
    }
}
```

20.5 注意事项

- **必须**使用 `ip_hash` 或 `sticky` 实现会话保持，否则 WebSocket 节点切换将导致连接强制重连；
- `proxy_read_timeout` 是连接静默超时，并非总存活时间，**应设置为业务允许的最大间隔**；
- 反向代理时关闭 `proxy_buffering` 与 `proxy_cache` 避免消息延迟；
- 当客户端使用 HTTPS 时，ALB 与上游可使用 HTTP（如 `proxy_pass http://ws_backend;`），无需双层 TLS；
- WebSocket over HTTP/2（RFC 8441）目前兼容性较差，ALB 当前建议使用 HTTP/1.1 升级方式建立 WebSocket 连接。

21 URL 重写与重定向

ALB 提供灵活的 URL 重写与重定向能力，可用于路径美化、旧链接迁移、API 版本路由、HTTPS 强制跳转、路由兜底等场景。

21.1 基础配置

```
# 1. return 强制 HTTPS
server {
    listen 80;
    server_name example.com;

    location / {
        return 301 https://$host$request_uri;
    }
}

# 2. rewrite 路径重写
server {
    listen 80;
    server_name example.com;

    location /old-api/ {
        rewrite ^/old-api/(.*)$ /new-api/v2/$1 permanent;
    }
}

# 3. try_files 路由兜底
server {
    listen 80;
    server_name example.com;

    location / {
        try_files $uri $uri/ /index.html;
```

```
    }
}
```

21.2 关键指令说明

指令	上下文	作用
`return code [text|url]`	server/location	直接返回状态码/重定向，停止后续处理
`rewrite regex replacement [flag]`	server/location/if	按正则重写 URL，可选 `last`/`break`/`redirect`/`permanent` 标志
`try_files file ... =code`	server/location	依次检查文件是否存在，最后一个参数为兜底（URI 或状态码）

21.2.0.1 rewrite 标志位

标志	状态码	是否重新匹配 location	典型用途
（无）	—	否	内部重写，浏览器无感知
`last`	—	是	重写后停止当前 location，重新匹配
`break`	—	否	重写后停止当前规则，继续处理当前 location
`redirect`	302	否	临时重定向，浏览器地址栏更新
`permanent`	301	否	永久重定向，浏览器与 SEO 友好

21.2.0.2 return 常用状态码

状态码	含义
`301`	永久重定向，浏览器会缓存
`302`	临时重定向，浏览器每次重新请求
`307`	临时重定向，保留请求方法（POST → POST）
`308`	永久重定向，保留请求方法
`444`	ALB 特有，不返回响应直接关闭连接（防扫描）
`403` / `404` / `500` 等	错误响应

21.3 真实使用场景

21.3.1 场景一：旧 API 路径迁移

老接口 `/api/v1/users` 迁移至新接口 `/api/v3/members`，使用 301 永久重定向保留 SEO 权重：

```
server {
    listen 80;
    server_name api.example.com;

    location /api/v1/ {
        rewrite ^/api/v1/(.*)$ /api/v3/members/$1 permanent;
    }

    location /api/v3/ {
        proxy_pass http://api_backend;
    }
}
```

21.3.2 场景二：HTTP 强制跳转 HTTPS

所有 HTTP 访问必须重定向至 HTTPS：

```
server {
    listen 80;
    server_name example.com;

    # 全站强制 HTTPS
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl;
    server_name example.com;
    # ... HTTPS 配置
}
```

21.3.2.1 场景三：SPA 前端路由兜底

React/Vue 单页应用，物理路径不存在时回退到 `index.html`，由前端路由处理：

```
server {
    listen 80;
    server_name app.example.com;
    root /var/www/app;

    location / {
        # 优先找物理文件 → 找目录 → 兜底返回 index.html
        try_files $uri $uri/ /index.html;
    }
}
```

21.3.2.2 场景四：API 版本头透传与路径重写

调用方统一访问 `/api/v1/`，但后端服务只暴露 `/internal/v3/`：

```
location /api/v1/ {
    # 去掉 /api/v1 前缀，替换为 /internal/v3/
    rewrite ^/api/v1/(.*)$ /internal/v3/$1 break;
    proxy_pass http://api_backend;
}
```

21.4 注意事项

- `rewrite` 与 `return` 同处一个 `location` 时，`return` 优先；
- `permanent` (301) 一旦下发，浏览器会长期缓存，迁移前确认 URL 永久变更；
- `last` 标志会重新匹配 location，可能导致循环重写（应避免 `last` 后又命中同一规则）；
- 复杂重写建议使用 `map` 指令 + 变量，避免 `if` 中嵌套 `rewrite`（`if` 指令有诸多限制）。

22 跨域配置（CORS）

ALB 可在网关层统一处理 CORS（Cross-Origin Resource Sharing，跨域资源共享），避免在每个后端服务中重复实现跨域逻辑。常见场景：前后端分离项目中，前端通过 Ajax 调用不同域的 API 接口。

22.0.1 基础配置

```
server {
    listen 80;
    server_name api.example.com;

    # 简单请求处理
    location /api/ {
        # 允许所有来源（生产环境应明确指定）
        add_header Access-Control-Allow-Origin $http_origin always;
        add_header Access-Control-Allow-Credentials true always;
        add_header Access-Control-Allow-Methods "GET, POST, PUT,
DELETE, OPTIONS" always;
        add_header Access-Control-Allow-Headers "Authorization,
Content-Type, X-Requested-With" always;
        add_header Access-Control-Max-Age 3600;

        # 处理 OPTIONS 预检请求
        if ($request_method = 'OPTIONS') {
            add_header Access-Control-Allow-Origin $http_origin;
            add_header Access-Control-Allow-Credentials true;
            add_header Access-Control-Allow-Methods "GET, POST, PUT,
DELETE, OPTIONS";
            add_header Access-Control-Allow-Headers "Authorization,
Content-Type, X-Requested-With";
            add_header Access-Control-Max-Age 3600;
            add_header Content-Type "text/plain; charset=utf-8";
            add_header Content-Length 0;
            return 204;
        }
```

```
        proxy_pass http://api_backend;
        proxy_set_header Host $host;
    }
}
```

22.1 关键指令说明

响应头	作用
`Access-Control-Allow-Origin`	允许的来源（`*` 或具体域名，多域名需配合 `$http_origin`）
`Access-Control-Allow-Credentials`	是否允许携带 Cookie（设为 `true` 时 `Allow-Origin` 不可为 `*`）
`Access-Control-Allow-Methods`	允许的 HTTP 方法（OPTIONS 预检响应必备）
`Access-Control-Allow-Headers`	允许的请求头（前端自定义头如 `Authorization`）
`Access-Control-Expose-Headers`	暴露给 JS 的响应头（默认 JS 只能读取 6 个 CORS 安全头）
`Access-Control-Max-Age`	预检结果缓存时间（秒），减少 OPTIONS 请求频率
`$http_origin`	客户端请求的 Origin 头值

22.2 真实使用场景

22.2.0.1 场景一：多来源白名单

企业网关需要同时支持 `https://admin.example.com` 与 `https://ops.example.com` 调用 API：

```
# 通过 map 维护白名单
map $http_origin $cors_origin {
    default "";
    "https://admin.example.com" "https://admin.example.com";
    "https://ops.example.com"   "https://ops.example.com";
    "https://test.example.com"  "https://test.example.com";
}

server {
    listen 80;
    server_name api.example.com;
```

```
    location /api/ {
        add_header Access-Control-Allow-Origin $cors_origin always;
        add_header Access-Control-Allow-Credentials true always;
        add_header Access-Control-Allow-Methods "GET, POST, PUT,
DELETE, OPTIONS" always;
        add_header Access-Control-Allow-Headers "Authorization,
Content-Type" always;

        if ($request_method = 'OPTIONS') {
            add_header Access-Control-Allow-Origin $cors_origin;
            add_header Access-Control-Allow-Credentials true;
            add_header Access-Control-Allow-Methods "GET, POST, PUT,
DELETE, OPTIONS";
            add_header Access-Control-Allow-Headers "Authorization,
Content-Type";
            add_header Access-Control-Max-Age 86400;
            return 204;
        }

        proxy_pass http://api_backend;
    }
}
```

22.2.0.2 场景二：全开放（仅适用于内部系统/开发环境）

```
location /api/ {
    add_header Access-Control-Allow-Origin * always;
    add_header Access-Control-Allow-Methods "GET, POST, OPTIONS"
always;
    add_header Access-Control-Max-Age 3600;

    if ($request_method = 'OPTIONS') {
        return 204;
    }
```

```
    proxy_pass http://api_backend;
}
```

22.3 注意事项

- `add_header` 在 `4xx` / `5xx` 响应中默认不生效，需添加 `always` 参数强制下发；
- 使用 Cookie 时 `Access-Control-Allow-Origin` **不能**为 `*`，必须为具体域名；
- OPTIONS 预检请求必须直接返回 204，**不要**代理到上游（避免上游重复处理）；
- CORS 头应统一在 ALB 处理，避免后端服务与 ALB 同时下发导致重复头冲突；
- 涉及敏感接口时，建议在 CORS 之外额外校验 `Origin` 头或 Referer 头。

23 故障转移与重试

ALB 在上游节点故障时具备自动故障转移与重试能力，可显著提升服务可用性。

23.0.1 基础配置

```
upstream backend {
    server 10.0.0.10:8080 max_fails=3 fail_timeout=30s;
    server 10.0.0.11:8080 max_fails=3 fail_timeout=30s;
    server 10.0.0.12:8080 max_fails=3 fail_timeout=30s;

    # 备机节点：所有主机不可用时启用
    server 10.0.0.100:8080 backup;
}

server {
    listen 80;
    server_name example.com;

    location / {
        proxy_pass http://backend;

        # 失败重试：仅在指定错误时重试下一个上游
        proxy_next_upstream error timeout invalid_header http_502
http_503 http_504;
        proxy_next_upstream_tries 3;        # 最多尝试 3 个节点
        proxy_next_upstream_timeout 10s;    # 整体重试时间不超过 10s

        # 重试时禁用请求体重传（POST 失败时不重试）
        proxy_next_upstream_non_idempotent off;
    }
}
```

23.1 关键指令说明

指令	默认值	作用
max_fails=N	1	失败次数达到 N 后，节点被标记为不可用
fail_timeout=time	10s	失败次数计算窗口；标记为不可用后，节点被排除的时间
backup	—	备机节点，仅当所有主机都不可用时才使用
down	—	永久标记节点不可用（用于维护时摘除节点）
proxy_next_upstream	error timeout	触发重试的条件（error / timeout / invalid_header / http_502 / http_503 / http_504 / http_403 / http_404 / http_500 / off）
proxy_next_upstream_tries	0	最多重试次数（0 表示无限制，受 proxy_next_upstream_timeout 约束）
proxy_next_upstream_timeout	0	整体重试超时（0 表示无限制）
proxy_next_upstream_non_idempotent	off	是否对非幂等请求（POST）启用重试（默认 off）

23.2 真实使用场景

23.2.0.1 场景一：主备灾备

主机房故障时自动切换到备机房：

```
upstream backend {
    # 主机房
    server 10.0.0.10:8080 max_fails=3 fail_timeout=30s;
    server 10.0.0.11:8080 max_fails=3 fail_timeout=30s;
    # 备机房
    server 10.1.0.10:8080 max_fails=3 fail_timeout=30s backup;
    server 10.1.0.11:8080 max_fails=3 fail_timeout=30s backup;
}

server {
    listen 80;
    location / {
        proxy_pass http://backend;
        proxy_next_upstream error timeout http_502 http_503
http_504;
```

```
        proxy_next_upstream_tries 3;
    }
}
```

23.2.0.2 场景二：节点维护摘除

需要对某节点进行维护时，可临时将其标记为 `down`：

```
upstream backend {
    server 10.0.0.10:8080;
    server 10.0.0.11:8080 down;        # 维护中
    server 10.0.0.12:8080;
}
```

或通过 Lua 动态控制（无需重启）：

```
http {
    init_by_lua_block {
        -- 从 Redis 加载节点状态
        local redis = require "resty.redis"
        local red = redis:new()
        red:set_timeout(1000)
        local ok = red:connect("127.0.0.1", 6379)
        if ok then
            ngx.shared.nodes:set("10.0.0.11:8080", "down")
        end
    }
}
```

23.2.0.3 场景三：可重试与不可重试请求区分

GET 等幂等请求可重试，POST 涉及订单/支付等业务应避免重试：

```
map $request_method $retry_enable {
    default 1;
    POST    0;
    PUT     0;
```

```
    DELETE  0;
}

server {
    location /api/ {
        proxy_pass http://backend;
        proxy_next_upstream error timeout http_502 http_503
http_504;
        # 仅幂等请求启用重试
        proxy_next_upstream $retry_enable;
    }
}
```

23.3 注意事项

- `proxy_next_upstream` 在重试时会**完整重传**请求体（除非上游已收到部分响应），POST 等非幂等请求需谨慎开启；
- `max_fails` 与主动健康检查独立工作，建议同时启用主动健康检查以更快摘除故障节点（参见"健康检查"章节）；
- `fail_timeout` 时间设置过短会导致节点在恢复后仍被排除，过长则故障切换缓慢，建议 10-30s；
- `proxy_next_upstream_non_idempotent on` 仅在业务确保上游接口**完全幂等**时开启，否则可能产生重复订单/支付。

24 用户场景配置案例

24.1 概述

ALB 不仅是一个 Web 服务器，更是现代应用架构中的"流量中枢"。它在高并发、安全防护、服务治理等方面发挥着关键作用。本文通过多个贴近真实业务的场景，详细说明企业在不同发展阶段如何借助ALB解决实际问题，并附上可直接复用的配置样例与效果分析。

24.2 场景一：前端单页应用托管与路由支持/静态资源发布

24.2.1 1. 背景

某公司使用 React 开发管理后台，构建后输出静态资源。初期通过 Node.js Express 提供静态服务，部署在单台云服务器上。

24.2.2 2. 遇到的问题与待解决的问题

- 用户直接访问 /settings 等非根路径时返回 404（因服务器无对应物理文件）；
- 静态资源未缓存，每次刷新都重新下载，带宽成本高、加载慢；
- Express 在并发 >500 时 CPU 占用飙升，响应延迟超过 2 秒。

待解决问题：实现 SPA 路由兼容、静态资源高效分发、降低服务器负载。

24.2.3 3. ALB 功能与配置与问题匹配

- `try_files` 指令：实现前端路由 fallback 到 index.html；
- `expires` 与 `add_header`：设置长期缓存策略；
- ALB 高性能事件驱动模型：高效处理静态文件请求。

24.2.4 4. 使用 ALB 解决该场景下的配置与详解

```
server {
    listen 80;
    server_name test.com;

    root /var/www/admin/dist;
    index index.html;

    # SPA 路由支持：若文件不存在，回退到 index.html
    location / {
```

```
        try_files $uri $uri/ /index.html;
    }

    # 静态资源设置 1 年缓存（immutable）
    location ~* \.(js|css|png|jpg|svg|woff2)$ {
        expires 1y;
        add_header Cache-Control "public, immutable";
    }
}
```

- `try_files $uri $uri/ /index.html`：优先找物理文件，找不到则交由前端路由处理；
- `immutable` 缓存：浏览器在缓存有效期内绝不发起条件请求，极大提升复访速度。

24.2.5 5. 效果，前后对比

指标	改造前（Express）	改造后（ALB）
首屏加载时间	2.1s	0.6s
静态资源 QPS	300	5000+
CPU 占用（500 并发）	85%	12%
路由 404 错误	频繁发生	完全消除

24.3 场景二：微服务 API 统一入口与负载均衡

24.3.1 1. 背景

某电商平台完成微服务拆分，拥有 user、order、payment 三个核心服务，分别部署在不同主机或容器中。

24.3.2 2. 遇到的问题与待解决的问题

- 客户端需维护多个服务地址，调用复杂；
- 某实例宕机时无法自动剔除，导致请求失败；
- 缺乏统一日志，排查问题困难。

待解决问题：统一 API 入口、自动负载均衡、集中可观测性。

24.3.3 3. ALB 功能与配置与问题匹配

- `upstream` + `server`：定义后端集群；
- `proxy_pass`：反向代理到对应服务；

- `max_fails/fail_timeout`：健康检查机制；
- `access_log`：统一记录请求日志。

24.3.4 4. 使用 ALB 解决该场景下的配置与详解

```
upstream user_backend {
    server 10.0.0.10:8080 max_fails=3 fail_timeout=30s;
    server 10.0.0.11:8080 max_fails=3 fail_timeout=30s;
}

upstream order_backend {
    server 10.0.0.10:8080 max_fails=3 fail_timeout=30s;
    server 10.0.0.11:8080 max_fails=3 fail_timeout=30s;
}

upstream payment_backend {
    server 10.0.0.10:8080 max_fails=3 fail_timeout=30s;
    server 10.0.0.11:8080 max_fails=3 fail_timeout=30s;
}

server {
    listen 80;
    server_name api.shop.com;

    access_log /var/log/alb/api.log combined;

    location /api/users/ {
        proxy_pass http://user_backend/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    location /api/orders/ {
        proxy_pass http://order_backend/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
```

```
    }

    location /api/payment/ {
        proxy_pass http://payment_backend/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

- `max_fails=3`：连续 3 次失败后，30 秒内不再转发请求到该节点；
- `X-Real-IP`：透传真实客户端 IP，便于后端审计。

24.3.5 5. 效果，前后对比

- 客户端调用地址从 3 个减少为 1 个；
- 服务实例故障时，错误率从 15% 降至 0.2%；
- 运维可通过单一日志文件追踪所有 API 调用链。

24.4 场景三：全站 HTTPS 强制跳转与 SSL 终止

24.4.1 1. 背景

某在线教育平台因 GDPR 合规要求，必须启用 HTTPS。当前同时开放 HTTP 和 HTTPS，证书由 Java 应用管理。

24.4.2 2. 遇到的问题与待解决的问题

- 用户可能通过 HTTP 访问，存在数据泄露风险；
- 后端应用处理 SSL 加解密，CPU 开销大；
- 证书更新需重启应用，影响可用性。

待解决问题：强制 HTTPS、SSL 边缘终止、零停机证书更新。

24.4.3 3. ALB 功能与配置与问题匹配

- return 301：HTTP 强制跳转 HTTPS；
- ssl_certificate：加载 TLS 证书；
- 支持热重载（./bin/reload-alb.sh），无需中断连接。

24.4.4 4. 使用 ALB 解决该场景下的配置与详解

```
server {
    listen 80;
```

```
    server_name learn.edu.com;
    return 301 https://$host$request_uri;   # 强制跳转
}

upstream backend {
    server 10.0.0.10:8080 max_fails=3 fail_timeout=30s;
    server 10.0.0.11:8080 max_fails=3 fail_timeout=30s;
}
server {
    listen 443 ssl http2;
    server_name learn.edu.com;

    ssl_certificate /opt/ALB-V2.0.6-SE-
amd64/conf/learn.edu.com/fullchain.pem;
    ssl_certificate_key /opt/ALB-V2.0.6-SE-
amd64/conf/learn.edu.com/privkey.pem;

    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5;   # 使用更安全的加密算法

    location / {
        proxy_pass http://backend/;
        proxy_set_header X-Forwarded-Proto $scheme;   # 告知后端原始协议
    }
}
```

24.4.5 5. 效果，前后对比

- HTTP 流量占比从 40% 降至 0%；
- 后端 CPU 负载下降 18%；
- 证书更新只需替换文件并 reload，服务零中断。

24.5 场景四：API 接口防刷与速率限制

24.5.1 1. 背景

某社交平台开放用户信息查询接口 /api/profile/{id}，近期遭遇爬虫高频请求。

24.5.2 2. 遇到的问题与待解决的问题

- 数据库 QPS 从 2000 暴增至 15000，响应超时；
- 正常用户请求被阻塞；
- 临时封 IP 影响合法开发者。

待解决问题：在不修改业务代码前提下，快速实施限流。

24.5.3 3. ALB 功能与配置与问题匹配

- `limit_req_zone`：基于 IP 定义速率限制区域；
- `limit_req`：在 location 中应用限流策略；
- `burst + nodelay`：允许突发流量平滑处理。

24.5.4 4. 使用 ALB 解决该场景下的配置与详解

```
http {
    limit_req_zone $binary_remote_addr zone=profile_limit:10m
rate=5r/s;

    server {
        location = /api/profile {
            limit_req zone=profile_limit burst=10 nodelay;
            limit_req_status 429;
            proxy_pass http://user_service;
        }
    }
}
```

- `rate=5r/s`：每秒最多 5 次请求；
- `burst=10`：允许瞬时突发 10 次，避免正常用户误伤；
- 返回 429 状态码，符合 RESTful 规范。

24.5.5 5. 效果，前后对比

- 恶意请求拦截率 99.6%；
- 数据库 QPS 回落至 2200；
- 正常用户无感知，开发者可依据 429 实现重试逻辑。

24.6 场景五：动静分离与动态内容缓存

24.6.1 1. 背景

视频平台课程详情页包含动态数据（学习进度、评论），但被频繁访问。

24.6.2 2. 遇到的问题与待解决的问题

- 每次请求都穿透到 Java 后端，即使内容未变；
- Redis 缓存策略复杂，命中率低；
- 高峰期数据库连接池耗尽。

待解决问题：对“相对静态”的动态页面做边缘缓存。

24.6.3 3. ALB 功能与配置与问题匹配

- proxy_cache_path：定义缓存存储；
- proxy_cache_valid：设置缓存有效期；
- proxy_cache_use_stale：故障时返回过期缓存，保障可用性。

24.6.4 4. 使用 ALB 解决该场景下的配置与详解

```
proxy_cache_path /cache/alb levels=1:2 keys_zone=course_cache:20m
inactive=10m;

location = /course/detail {
    proxy_cache course_cache;
    proxy_cache_valid 200 10m;
    proxy_cache_use_stale error timeout updating;
    proxy_pass http://course_service;
    add_header X-Cache $upstream_cache_status;
}
```

24.6.5 5. 效果，前后对比

- 课程页缓存命中率 72%；
- 后端 QPS 下降 65%；
- 故障期间用户仍可看到“稍旧但可用”的页面。

24.7 场景六：WebSocket 长连接代理

24.7.1 1. 背景

官网集成在线客服系统，使用 WebSocket 实现实时聊天。

24.7.2 2. 遇到的问题与待解决的问题

- 连接建立后 60 秒自动断开；
- 消息丢失严重，用户体验差；
- 无法与 HTTP 共用 443 端口。

待解决问题：稳定代理 WebSocket 长连接。

24.7.3 3. ALB 功能与配置与问题匹配

- `Upgrade` 和 `Connection` 头透传；
- `proxy_read_timeout`：延长读超时时间。

24.7.4 4. 使用 ALB 解决该场景下的配置与详解

```
location /ws/chat/ {
    proxy_pass http://chat_service;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_read_timeout 86400s;
}
```

24.7.5 5. 效果，前后对比

- WebSocket 连接可稳定维持 24 小时；
- 消息到达率从 82% 提升至 99.9%；
- 与 HTTPS 共享 443 端口，简化网络策略。

24.8 场景七：防止恶意扫描与目录遍历攻击

24.8.1 1. 背景

某政府信息公开网站部署在公网，近期被自动化工具频繁扫描 `/admin/`、`/.git/`、`/backup.zip` 等敏感路径，尝试获取内部配置或源码。

24.8.2 2. 遇到的问题与待解决的问题

- 攻击请求占用服务器资源；
- 日志中大量 404 请求干扰正常监控；
- 存在因误暴露备份文件导致数据泄露的风险。

待解决问题：在入口层拦截高危路径访问，无需修改应用代码。

24.8.3 3. ALB 功能与配置与问题匹配

- `location` 精确匹配 + `deny all`：禁止访问敏感目录；
- 正则匹配隐藏文件（如 `.env`、`.git`）；
- 返回 444（ALB 特有状态码，直接关闭连接，不响应）。

24.8.4 4. 使用 ALB 解决该场景下的配置与详解

```
server {
    listen 80;
    server_name www.govinfo.gov.cn;

    # 禁止访问隐藏文件
    location ~ /\. {
        deny all;
        return 444;
    }

    # 禁止访问常见敏感目录
    location ~* ^/(admin|phpmyadmin|wp-admin|backup|config)/ {
        deny all;
        return 444;
    }

    # 禁止下载特定扩展名文件
    location ~* \.(sql|bak|tar\.gz|zip)$ {
        deny all;
        return 444;
    }

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

- `return 444`：不返回任何响应，直接断开 TCP 连接，降低攻击者探测效率；
- 正则 ~* 表示不区分大小写，覆盖更多变种路径。

24.8.5 5. 效果，前后对比

- 每日恶意请求从 12,000+ 降至 <50；
- 安全审计报告中"信息泄露风险"项清零；
- 服务器 CPU 在扫描高峰期下降 30%。

24.9 场景八：基于 Cookie 的灰度发布（A/B 测试）

24.9.1 1. 背景

电商平台计划上线新版商品详情页，希望先对 10% 用户开放新版本，验证转化率后再全量。

24.9.2 2. 遇到的问题与待解决的问题

- 后端无灰度能力，需网关层分流；
- 要求同一用户始终看到同一版本（会话一致性）；
- 不能依赖 IP（多用户共享出口）。

待解决问题：基于用户标识（如 Cookie）实现稳定灰度路由。

24.9.3 3. ALB 功能与配置与问题匹配

- map 指令：根据 $cookie_version 值决定后端；
- 自定义变量控制流量比例；
- sticky 会话保持（通过 Cookie 实现）。

24.9.4 4. 使用 ALB 解决该场景下的配置与详解

```
# 若用户无 version cookie，则随机分配（10% 概率设为 new）
map $cookie_version $backend_group {
    default "old";
    ""      "auto";  # 未设置时走自动分配
    "new"   "new";
}

upstream old_backend {
    server 10.0.0.10:8080;
}
```

```
upstream new_backend {
    server 10.0.0.20:8080;
}

server {
    listen 80;
    server_name shop.com;

    location /product/ {
        # 未分配版本的用户：10% 概率打标为 new
        if ($backend_group = "auto") {
            set $rand "";
            set_by_lua_block $rand { return math.random() < 0.1 and
"new" or "old" }
            set $backend_group $rand;
            add_header Set-Cookie "version=$rand; Path=/; Max-
Age=2592000";
        }

        if ($backend_group = "new") {
            proxy_pass http://new_backend;
        }
        if ($backend_group = "old") {
            proxy_pass http://old_backend;
        }
    }
}
```

24.9.5 5. 效果，前后对比

- 新老版本用户隔离清晰，无交叉污染；
- A/B 测试周期从 2 周缩短至 3 天；
- 无需改造业务系统，灰度策略完全由 ALB 控制。

24.10 场景九：大文件上传超时与分片支持优化

24.10.1 1. 背景

在线教育平台允许教师上传课件（PPT、视频），部分文件超过 500MB。

24.10.2 2. 遇到的问题与待解决的问题

- 默认 60 秒超时导致大文件上传失败；
- 用户网络波动时需重新上传整个文件；
- 后端 Java 应用内存溢出（OOM）。

待解决问题：延长上传超时、支持断点续传、减轻后端压力。

24.10.3 3. ALB 功能与配置与问题匹配

- client_max_body_size：允许大请求体；
- client_body_timeout / proxy_send_timeout：延长超时；
- （可选）结合 alb-upload-module 或前端分片实现断点续传。

24.10.4 4. 使用 ALB 解决该场景下的配置与详解

```
server {
    listen 80;
    server_name teach.edu.com;

    # 允许最大 2GB 文件
    client_max_body_size 2G;

    # 上传相关超时设为 1 小时
    client_body_timeout 3600s;
    proxy_send_timeout 3600s;
    proxy_read_timeout 3600s;

    location /api/upload {
        proxy_pass http://file_service;
        proxy_set_header X-Real-IP $remote_addr;
        # 临时文件存储路径（避免内存缓存大文件）
        client_body_temp_path /var/tmp/alb_upload;
    }
}
```

24.10.5 5. 效果，前后对比

- 500MB+ 文件上传成功率从 40% 提升至 99%；
- 后端 OOM 错误归零；
- 用户可在弱网环境下完成上传。

24.11 场景十：静态资源 Gzip 压缩加速

24.11.1 1. 背景

企业官网包含大量 JS/CSS 文件，海外用户反馈加载缓慢。

24.11.2 2. 遇到的问题与待解决的问题

- 未启用压缩，文本资源体积大；
- 后端动态压缩消耗 CPU；
- 移动端流量成本高。

待解决问题：在边缘层对文本资源进行高效 Gzip 压缩。

24.11.3 3. ALB 功能与配置与问题匹配

- `gzip on`：启用压缩；
- `gzip_types`：指定压缩 MIME 类型；
- `gzip_vary on`：通知代理缓存不同版本。

24.11.4 4. 使用 ALB 解决该场景下的配置与详解

```
http {
    gzip on;
    gzip_vary on;
    gzip_min_length 1024;   # 小于 1KB 不压缩
    gzip_comp_level 6;      # 平衡 CPU 与压缩率
    gzip_types
        text/plain
        text/css
        application/json
        application/javascript
        text/xml
        application/xml
        image/svg+xml;

    server {
```

```
        listen 80;
        server_name www.company.com;
        root /var/www/html;
    }
}
```

24.11.5 5. 效果，前后对比

资源类型	原始大小	Gzip 后	压缩率
main.js	850 KB	210 KB	75% ↓
style.css	320 KB	85 KB	73% ↓

- 页面整体加载时间减少 40%；
- 用户跳出率下降 18%。

全国统一服务热线

4008-555-800

金蝶天燕云计算股份有限公司(简称“金蝶天燕云”)成立于2000年，前身为“金蝶中间件公司”，是金蝶集团旗下新一代软件基础云平台服务商，云计算国家标准制定企业，国家信创产业核心软件企业。金蝶天燕是国家863重点研发计划与核高基重大专项承接企业，也是“两网一站四库十二金”国家重点工程的基础平台提供商，产品广泛应用于政府、军工、金融、能源等关键行业，累计服务客户总数超过10万家。

Apusic 金蝶天燕

云计算国家标准制定企业

金蝶集团旗下基础软件企业

信息技术应用创新核心企业

官网：www.apusic.com