灰度系统压测报告

2015-07-24

版本：1.0

目录

一、灰度系统简介

灰度发布系统的主要功能是对用户请求的分流转发，工作在 7 层，根据用户请求特征，如 UID、IP 等，将请求转发至 upstream server，实现分流。nginx 是目前使用较多的 7 层服务器，可以实现高性能的转发和响应；灰度发布系统的主要工作是在 nginx 转发的框架内，在转向 upstream 前，根据用户请求特征和分流策略，计算出目标 upstream 进而分流。灰度系统需要灵活而高效的实现这部分逻辑，使得对原生 nginx 转发的影响降到最低。

本系统基于淘宝的 tengine 开发，采用 ngx-lua 实现系统功能，采用 redis 作为分流策略数据库，通过启用 ngx-lua-sharedDict 和 lua-resty-lock 作为系统缓存和缓存锁，系统获得了较为接近原生 nginx 转发的性能。

本灰度系统与原生 nginx 转发实现的灰度发布系统的特点对比如下表所示，灰度系统的架构简图如图 1.1 所示。

	原生 nginx 系统	灰度发布系统
分流方式	由配置文件和 upstream 模块决定	在系统框架内可以灵活添加各种分流方式，只需实现相应的接口。目前支持按 ip、uid 分流。
更新方式	修改配置文件，重启系统后更新分流方式	通过 http 接口动态更新分流策略及运行时配置，无需重启，快速生效
分流后端	固定在配置文件中，重启后更新	通过 http 接口，添加分流策略实现后端服务器的更新

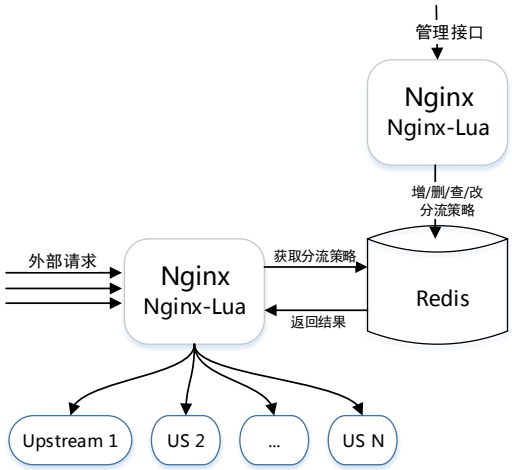


图 1.1 灰度系统架构简图

由于主要测试的是灰度系统的转发性能，我们主要关注压测过程中，proxy server 的 CPU 情况、网络设备使用情况和 nginx 的运行情况，而不监控 upstream server。压测过程中，nginx 的每秒处理能力 qps 和响应时间 response time（rt，ms）是我们最关心的；CPU 情况表示系统负载，主要关注 CPU 使用率和软硬件中断率，其中软件中断 si 主要用来响应网络事务，可以认为软中断的大小表示网络设备工作情况。

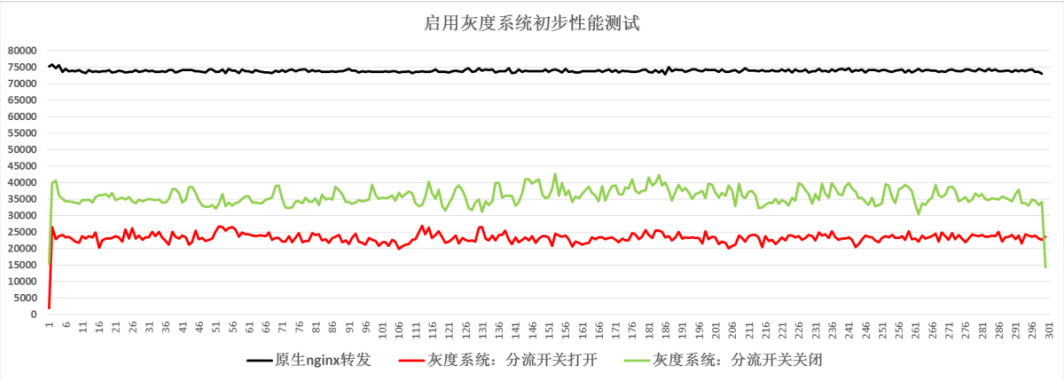
二、灰度系统的开发与优化过程

2.0 压测结果的图表说明

压测过程中由淘宝的 Tsar 来监控灰度系统的运行状况，每秒取样并输出，每个压测场景持续压测 300s，在高压力（并发量 20000）下持续压测 500s。压测结果中的折线图用来

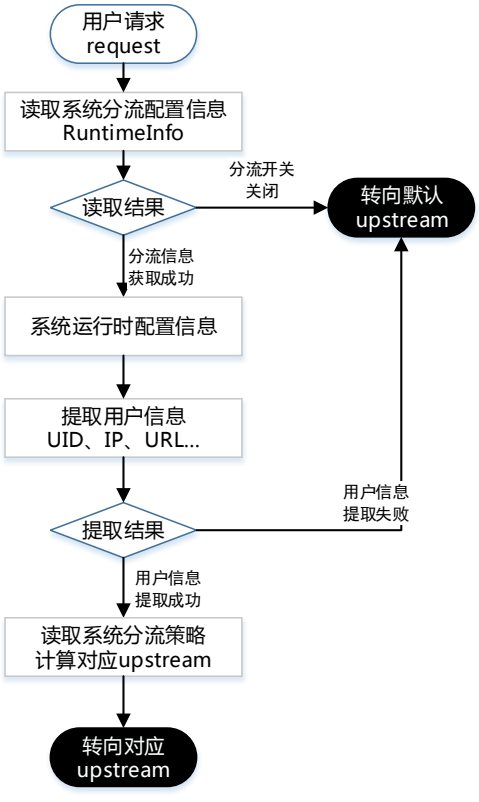
表示压测过程中 nginx 的运行情况，横轴是时间轴，纵轴是结果；柱状图表示压测平均值，纵轴定义与折线图相同。

以下图为例，显示对未经优化的灰度系统进行压测的过程中，qps 的折线图，横轴以秒为单位，纵轴是 qps，黑线、红线和绿线分别是原生 nginx 转发、分流开关打开时的灰度系统和分流开关关闭时的灰度系统。（**注：图表中的原生 nginx 转发都指的是 tengine**）



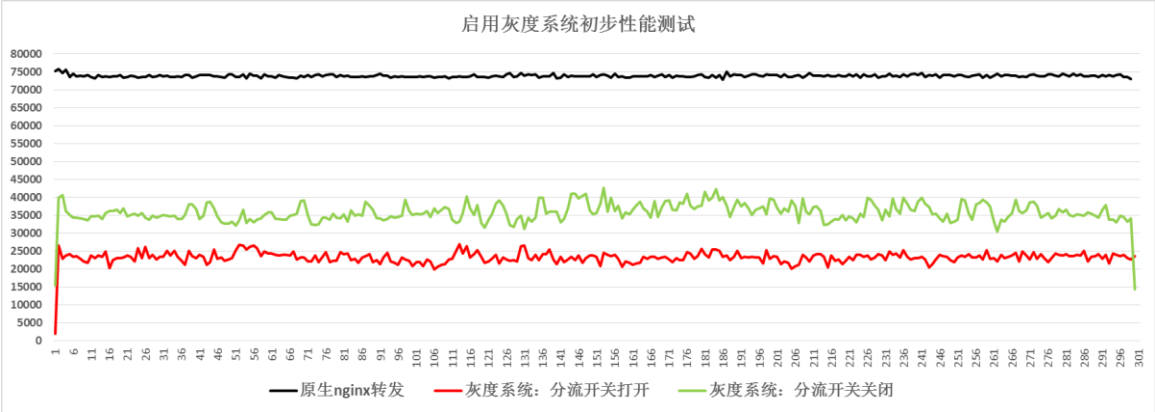
2.1 灰度系统初步性能测试

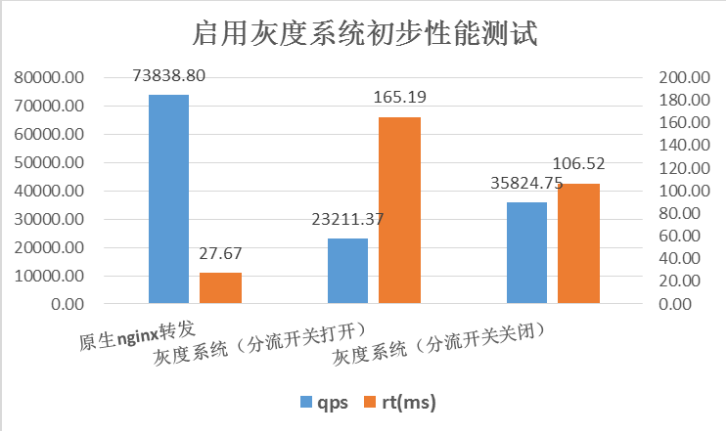
灰度系统采用 redis 作为数据库，系统的运行时配置信息和分流策略都存储在 redis 中，因此需要读取两次 redis 数据库。系统工作时，当用户请求接入后，首先读取系统运行时信息 runtimeInfo，该信息包括灰度系统执行分流所必须的三个模块名：分流模块名 divModulename、用户信息提取模块名 userInfoModulename 和分流策略库名 divDataKey；然后用户信息模块提取请求中的用户信息，比如 UID、IP 等；最后根据分流策略库名和用户特征信息从 redis 中读取分流策略，计算用户信息所对应的 upstream。整个分流过程的流程图如下：



灰度系统分流流程图

读取系统运行时信息 runtimeInfo 时，如果取到的三个模块名都合法，则认为灰度**系统分流开关打开**，正在执行分流，进而进行下一步处理；如果未能从数据库中取得同时合法的这三个参数，则认为当前**系统分流开关关闭**，将直接转发至默认 upstream。



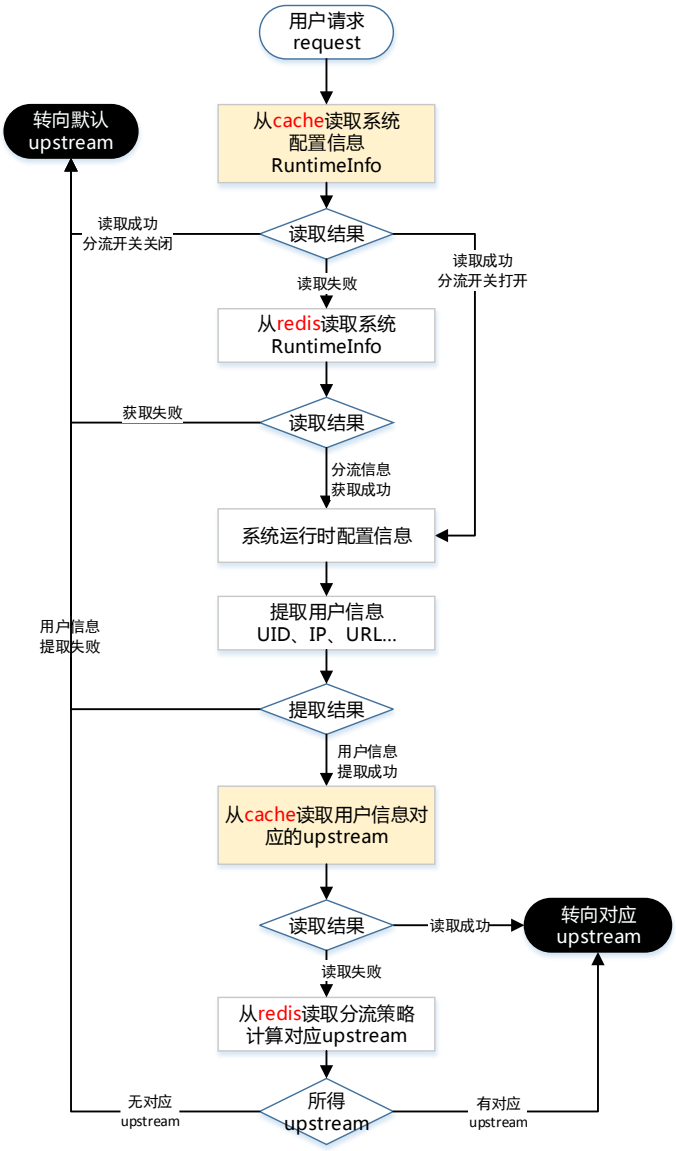


从图中可以看到，灰度系统与原生 nginx 转发之间存在较大的性能差异，原生 nginx 转发的 qps 可以达到平均值 73838，而在灰度系统启动后，在开启分流开关时，由于系统需要两次 redis 读取，所以 qps 低至平均 23211；在关闭分流开关时，由于系统需要读 redis 一次，所以 qps 比开关打开时稍高，平均为 35824。原生 nginx 转发的平均 rt 时间（response time）为 27.67ms，相应的分流开关打开时的灰度系统为 165.19ms，而分流开关关闭时灰度系统为 106.52ms，与 qps 存在负相关关系。

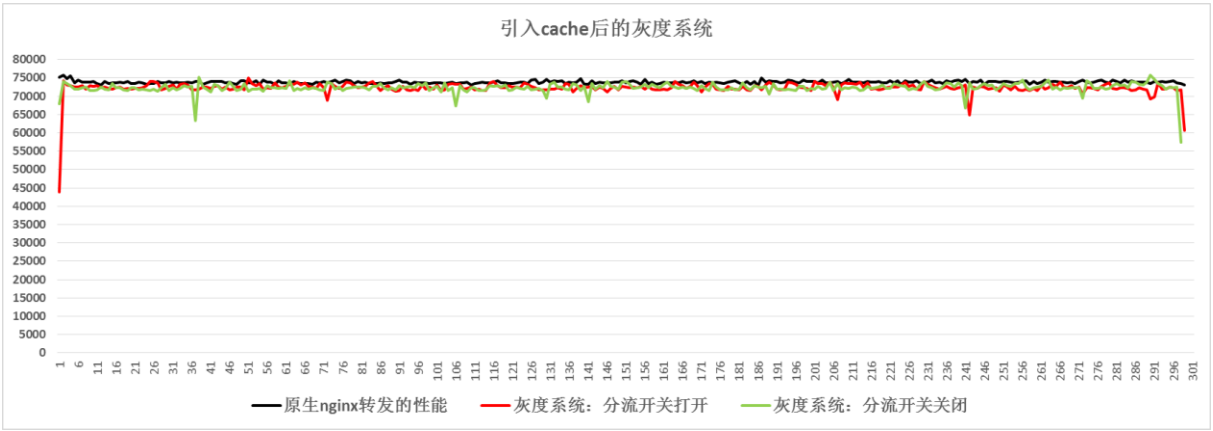
2.2 引入缓存后的灰度系统性能

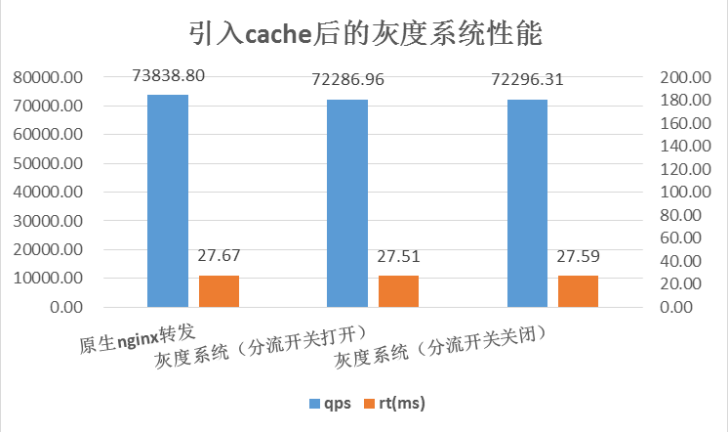
灰度系统工作时，处理每次请求都要读取 redis 数据库，代价很高，而所要读取的信息，比如系统运行时配置信息，在运行时的一段时间内都不会变化，且同一用户发出的请求将被转发至同一个 upstream，所以需要引入缓存机制，将系统运行时信息和用户信息所对应的 upstream 缓存起来，以减少对 redis 数据库的读取次数。

sharedDict 是 nginx-lua 的一个组件，可以实现跨 worker 共享数据，读写是原子实现的，且提供超时机制。对于灰度系统而言一个很好的 cache 方案，将**系统运行时信息**和**用户请求对应的 upstream** 缓存在 sharedDict 中。引入 sharedDict 后，灰度系统分流流程图如下图所示：



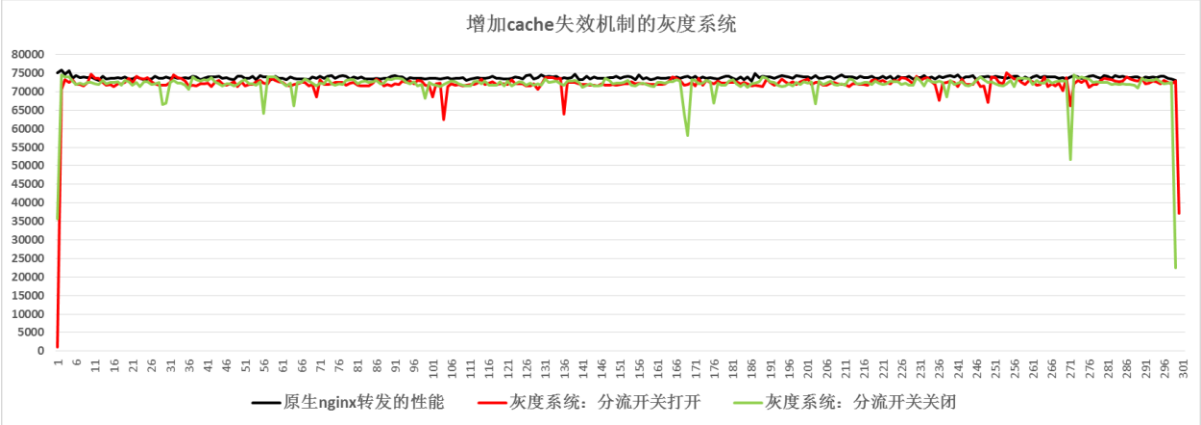
图：引入缓存后的灰度系统分流流程图

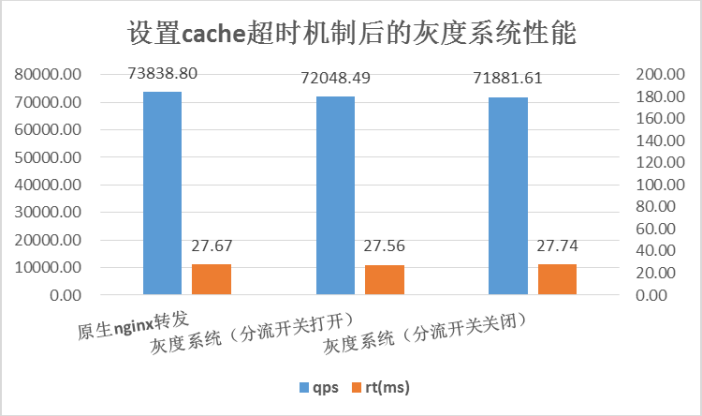




从图中看到，引入 cache 后的灰度系统获得了与原生 nginx 转发接近的性能，这得益于 nginx-lua 的高性能非阻塞设计机制。可以认为这是基于 nginx-lua 的灰度发布系统的理论性能上界。考虑到数据一致性，加入缓存后需要设置缓存失效时间，使系统功能更加完备。

2.3 引入缓存超时失效机制后的灰度系统性能

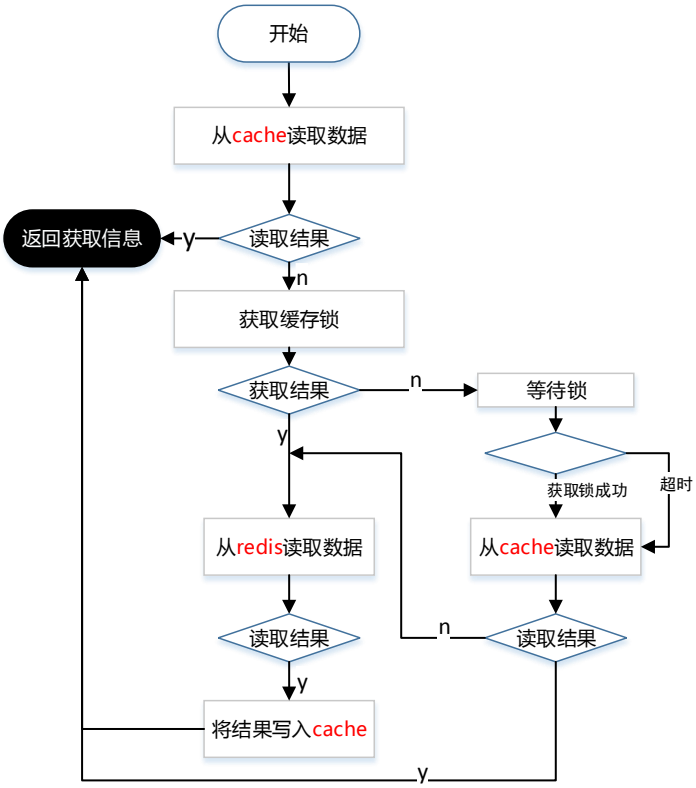




从图中看到，虽然平均 qps 和 rt 没有变化，但是引入 cache 失效机制后，还是引起了一个问题。当缓存失效时，大量并发请求从 cache 中得不到数据时，都会同时从 redis 读取数据。单线程的 redis 处理并发请求的能力较弱，造成灰度系统的性能急剧下降。为避免 cache 失效造成的性能下降，需要引入互斥锁，在从缓存读取**系统运行时信息**和**用户请求对应的 upstream** 失败时，需要先获取互斥锁，然后再读 redis，以降低对 redis 的压力。

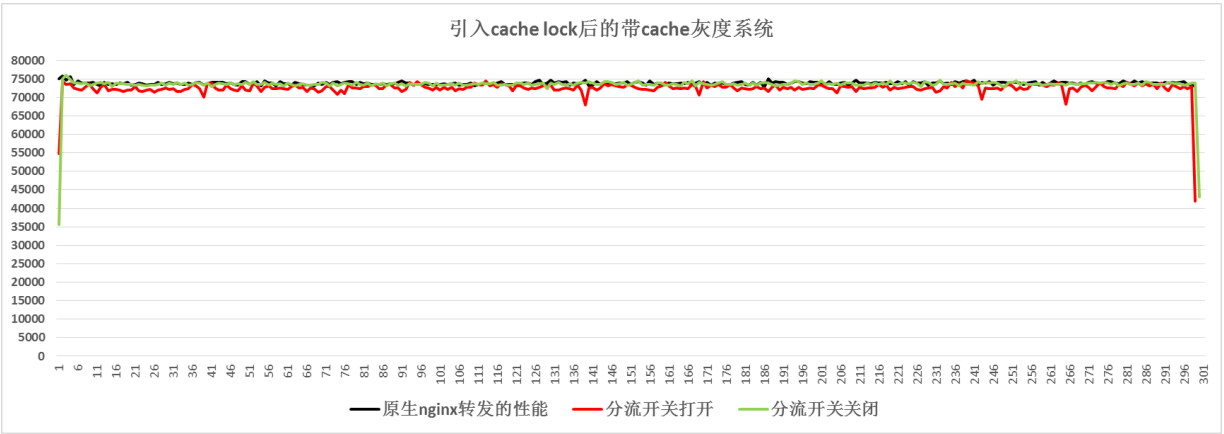
2.4 基于上述工作，引入 cache lock 后的灰度系统性能

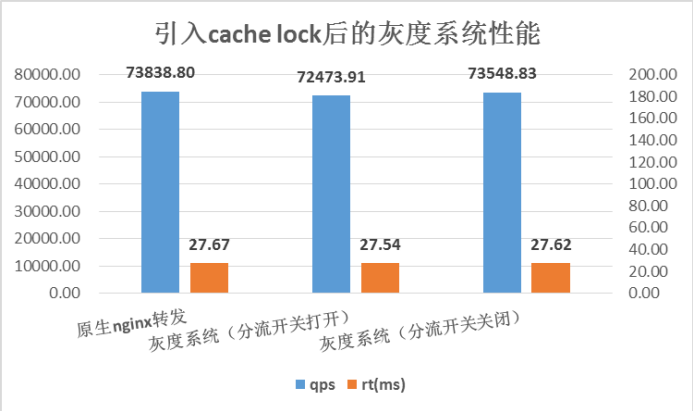
Lua-resty-lock 是 nginx-lua 的一个互斥锁实现，具有非阻塞、跨 worker 共享等特性，与 sharedDict 结合可以实现缓存锁功能。缓存锁工作流程图如图所示，当缓存失效时，首先需要获得锁才能访问 redis；如果不能获得锁，在一定时间（比如 1ms）后自动解除锁定；随后需要再次从 cache 中读取数据，因为在等待锁期间可能其他请求已经更新缓存，无需从 redis 中重新获取；如果 cache 中不能获取数据，则再次读取 redis。最终得到结果。



cache lock 缓存锁工作流程图

缓存锁机制可以避免因大量请求访问 redis 造成性能下降的问题，但是引入了单一锁，同样会造成性能下降，只是与 redis 访问相比会好些。（缓存锁机制可以缓解性能下降问题，而不能彻底解决。彻底解决该问题可以有多种方法：实现类似于信号量的机制，可以锁定一定数量；或更改架构设计，不使用后端数据库。）





从图中可以看到，灰度系统在 cache 失效时仍有一定性能下降问题，但相对于没有 cache lock 机制的情况有所缓解。（在系统中加入了两个 cache 锁，但在实际上线使用中将只对**系统运行时信息缓存**使用锁。这是因为压测场景中的用户请求比较单一，在 upstream 缓存失效时会对 redis 造成较大压力，而真实场景中不会有那么大的压力，同时互斥锁会造成不相关的干扰，对不用加锁的请求加锁。所以上线版本中将只对**读取系统运行时信息**加锁，**用户请求对应的 upstream 缓存**则不加锁。）

三、影响灰度系统性能的参数及配置

在系统开发和压力测试的过程中， nginx 参数、nginx 模块配置、系统 sysctl 参数和灰度系统的组成部分如 redis 的配置将会对系统性能产生较大影响。

3.1 nginx 参数配置

3.1.1 events 配置

```
events {
    worker_connections  32768;
    accept_mutex off;
    multi_accept on;
}
```

参数	配置	作用
worker_connections	32768	一个 worker 可以同时处理 32768 个连接。原理是预先分配 32768 大小的连接池以从 epoll 中获取新的连接。
accept_mutex	off	accept_mutex 设置锁用以避免“惊群效应”，让一个拿到 mutex 的 worker 去 accept 用户请求。 开启这个选项，将导致大部分情况只有一个 worker 去处理所有请求（并发量低于 32768），不能充分利用多核性能，限制了吞吐量。 关闭这个选项，所有 worker 将竞争“抢夺“用户请求，使得用户请求均摊在所有 worker 上，将大大提高系统吞吐量。系统的 load average 也将相应增大。
multi_accept	on	关闭选项，worker 一次只能取一个新连接； 开启选项，将允许 worker 一次将所有新连接。

3.1.2 listen 配置

```
listen PORT backlog=16384;
listen /tmp/nginx-tsar.sock backlog=16384;
```

关于 listen，重要的是：

1、 不论是 listen tcp socket，还是 unix domain socket，都需要设置 backlog，当处理请求速度跟不上的时候，backlog 长度体现 listen 的容纳能力；

2、 不论是 nginx，还是 redis，或者其他 C/S 模式服务，都需要设置 backlog，且通常 backlog 都很小。

3.1.3 nginx sever 对外设置长连接 keepalive

```
设置 keepalive（默认）                不设置 keepalive
    keepalive_timeout   75;              keepalive_timeout   0;
    keepalive_requests  100;             keepalive_requests  0;
```

Nginx server 对外的长连接设置由 keepalive_requests 和 keepalive_timeut 两个参数决定，其中前者默认值为 100，表示一条长连接可以处理的请求数，当超过这个值时连接将关闭；后者默认值时 75，表示一条长连接可以保持的时间为 75s。当不配置这两个参数时，nginx 是默认保持长连接的。当配置 keepalive_requests 和 keepalive_timeut 均为 0 时，则为事实上的关闭长连接设置。

3.2 nginx 的 proxy 和 upstream 模块配置

在 nginx 作为 proxy sever 的配置中，proxy server 和 upstream server 间的长连接配置由 proxy 模块和 upstream 模块共同决定。

```
upstream beta1 {
            keepalive 1000;
            server localhost:8020;
    }
location = /proxybeta1 {
                proxy_pass "http://beta1/";
                proxy_http_version 1.1;
                proxy_set_header Connection "";
    }
```

其中 keepalive 参数表示 proxy 与 upstream 间每个 worker 维持的长连接数，而 location 中需要将外部请求的 Connection 头部清空，并设置请求的 http 版本为 1.1 版。这些配置才能使能 proxy 和 upstream 间的长连接。

3.3 系统 sysctl 参数配置

net.core.somaxconn = 655360	系统设定的 backlog 值，若 listen 时 backlog 大于此值，则不会生效
net.core.netdev_max_backlog = 6553600	网卡设备的请求队列长度（硬件 backlog）
net.ipv4.tcp_max_tw_buckets = 50000 net.ipv4.tcp_tw_timeout = 5 net.ipv4.tcp_tw_recycle = 1 net.ipv4.tcp_tw_reuse = 1	端口回收，以及限制 time_wait 状态的 tcp 连接
net.ipv4.ip_local_port_range = 1025　65535	可用端口范围

net.unix.max_dgram_qlen = 655360	unix domain socket 的数据包队列

3.4 redis 配置参数

timeout 0	当一个连接空闲 N 秒后，主动关闭连接；N 为 0 时，redis 不主动关闭连接。
maxclients 262144	Redis 可以保持的最大连接数，默认 10000，超过后将拒绝服务
tcp-keepalive 120	主动保持 tcp 长连接，时长 120s
tcp-backlog 20000	listen 队列，backlog 为 20000。这个 backlog 不仅用于 tcp 连接，还用于 unix domain socket，尤其是在 redis 有 uds 客户端连接的时候

3.5 nginx-lua 与 redis 的长连接设置

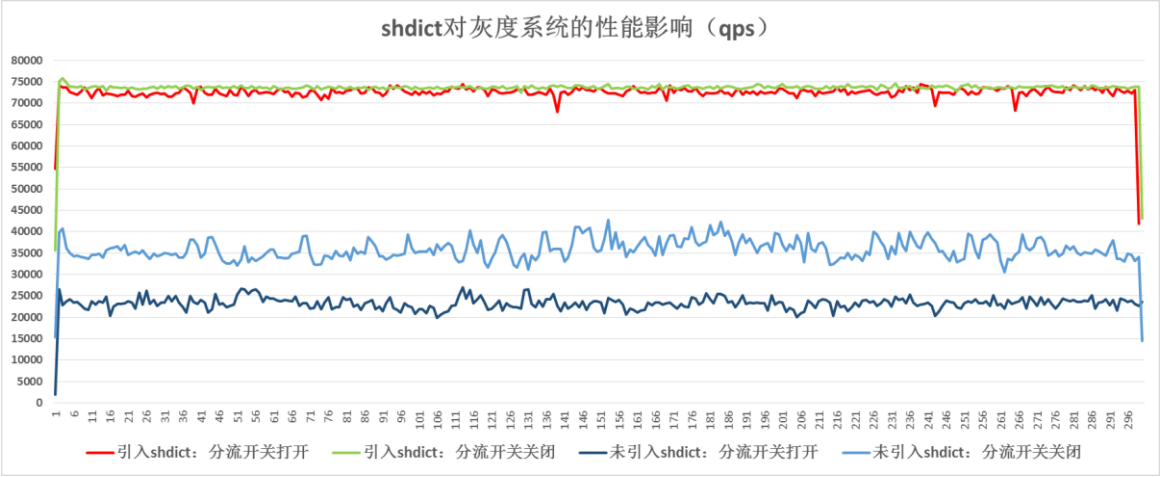
Ngx-lua 基于 cosocket 向 redis 发起 tcp/uds 连接，ngx-lua 的 set_keepalive 将使该连接保持长连接，时间为 90000ms，连接池大小为 1000。当 nginx 需要频繁访问 redis 时，保持与 redis 的长连接能够提高效率，减少不必要的建连任务。

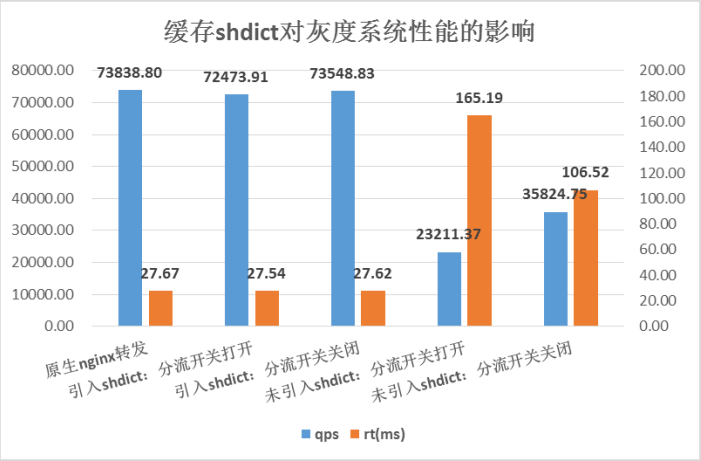
```
local pool_max_idle_time = 90000 --毫秒
local pool_size = 1000 --连接池大小
local ok, err = redis:set_keepalive(pool_max_idle_time, pool_size)

if not ok then
    ngx.log(ngx.ERR, 'failed set_keepalive, ', err)
    return
end
```

四、各参数对灰度系统性能的影响

4.1 缓存（sharedDict）对灰度系统性能的影响

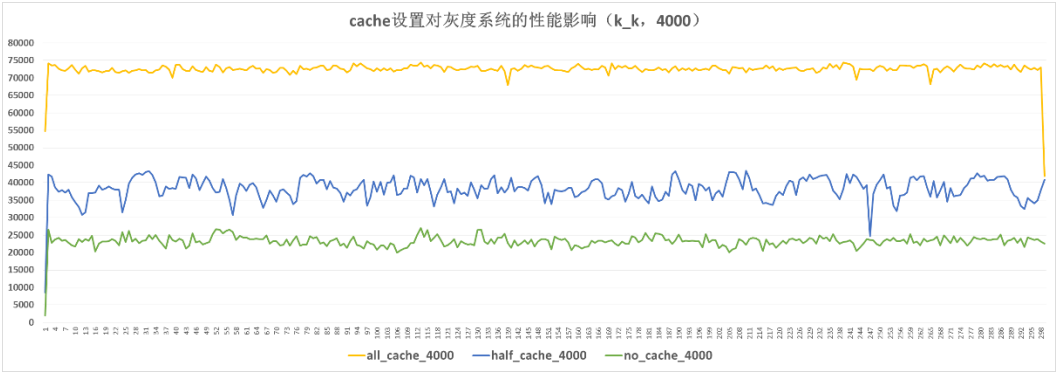


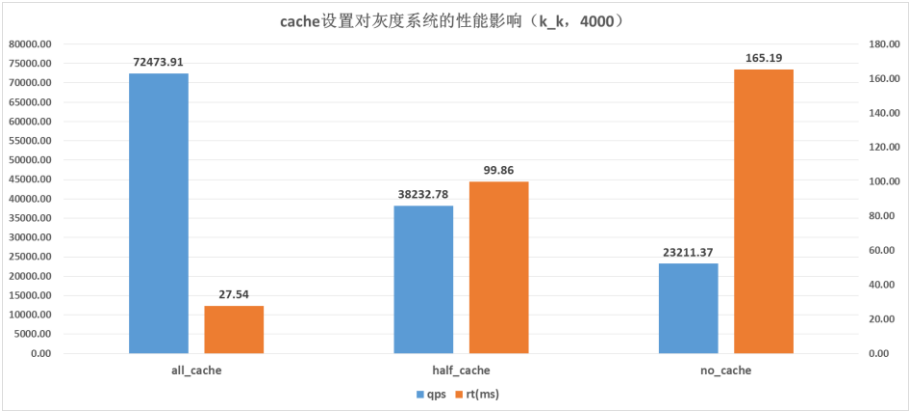


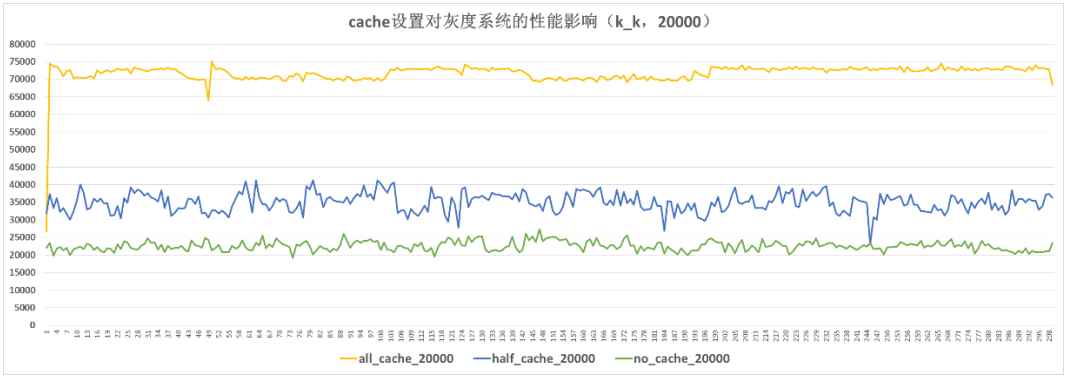
从图中可以看到，引入缓存后的灰度系统在 qps 和 rt 两个参数上非常接近，且性能稳定；而未引入缓存时灰度系统的性能则降低很多。可见缓存的引入对于目前灰度系统有非常重要的影响。如果未引入缓存，灰度系统需要**两次读取数据库 redis**，第一次是读取运行时分流配置信息，如果得到分流开关打开，则提取用户信息后再次从数据库 redis 读取分流策略来计算 upstream，否则直接转向默认 upstream。因此在图中看到，未引入 shdict 的情况下，分流开关打开时的 qps 比分流开关关闭时低很多，因为每次请求都多读 1 次 redis。

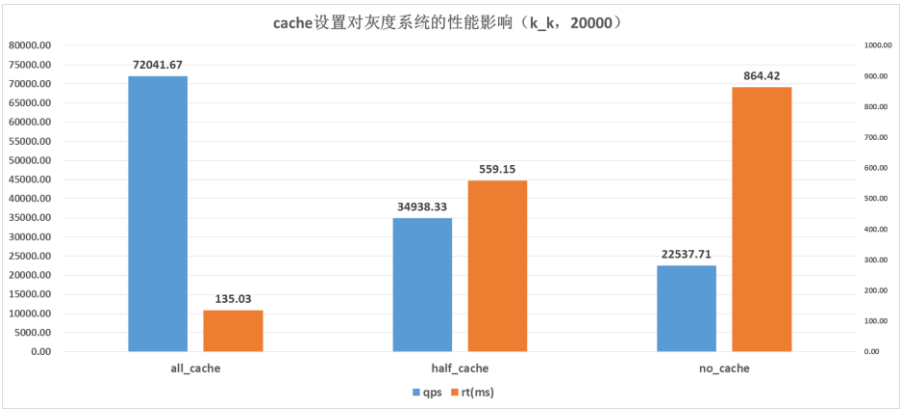
在真实场景中，运行时配置信息在一段时间内不会发生变化，因此将其缓存起来，使得分流过程中几乎可以不用从数据库中读取它；而从数据库中读取分流策略这一步，由于真实场景中的用户不会同时发出大规模的连接，所以将用户信息对应的 upstream 缓存起来意义不大，不缓存这个信息更加接近真实场景。因此在这里提出一种更偏向于真实情况的**压测场景**：只缓存运行时配置信息的**半 cache 场景**。这个场景仅仅是一种压测场景，上线版本中还是要对 upstream 使用缓存的。

4.1.1 只缓存运行时配置信息（半 cache）对灰度系统性能的影响



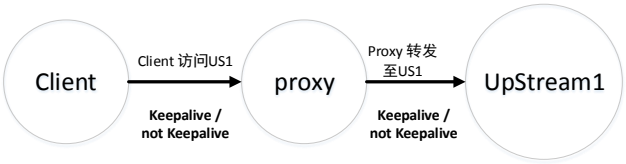






从图中看到，并发 4000 和并发 20000 时，all_cache 场景和 half_cache 场景的性能都比较稳定，而 half_cache 相对于 all_cache 来说，qps 只相当于一半，rt 时间高出很多。真实场景中的灰度系统性能将低于 all_cache，而高于 half_cache，偏向于 half_cache 场景。

4.2 长连接（keepalive）对性能的影响



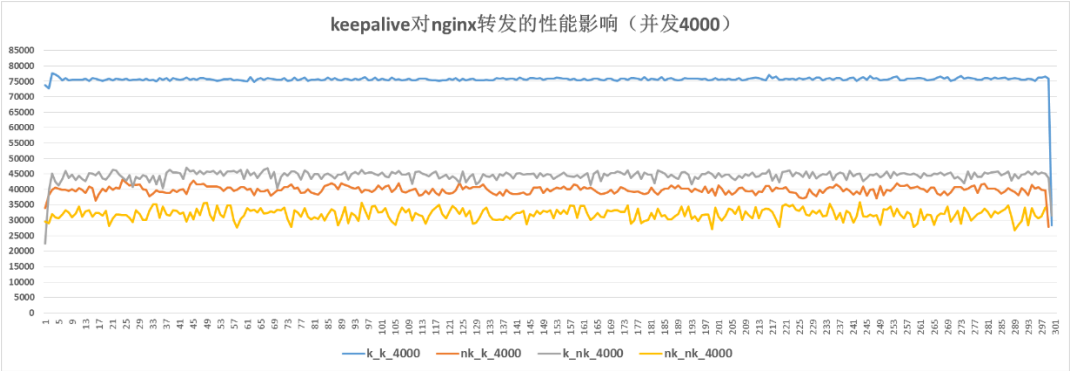
对于 nginx 实现转发的过程中，存在两个**长连接设置**，如图所示，分别是 **client 到 proxy server** 间的长连接设置，以及 **proxy server 到 upstream server 间**的长连接设置。

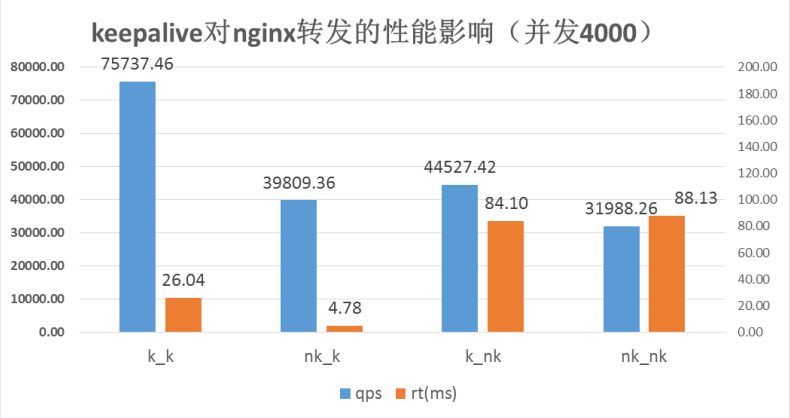
在真实场景中，Client 端一般通过浏览器发起请求，浏览器会启用 keepalive 设置，将一个会话发出的几个 http 请求以 keepalive 方式发出，保持长连接；在 7 层服务中，proxy 与 upstream 间会维持较大量的长连接。因此在 nginx 做 7 层服务时，其 keepalive 的设置可以分为四种情况：

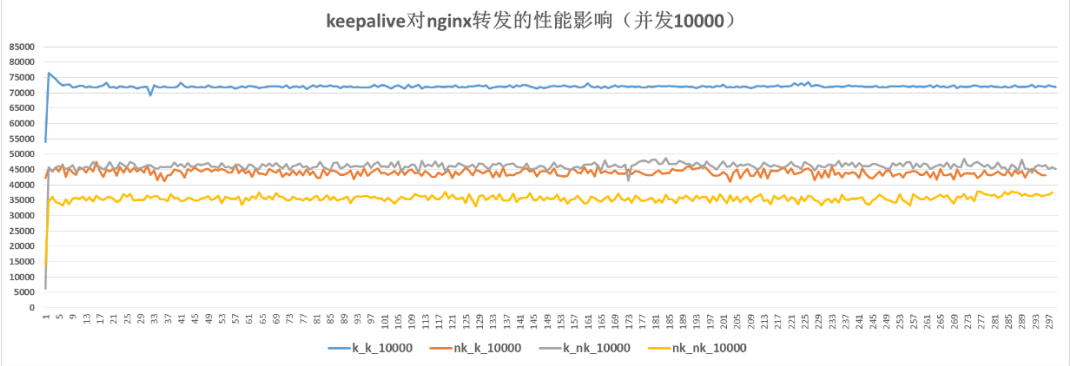
		Proxy 与 upstream	
		设置 keepalive	不设置 keepalive
Client 与 Porxy	设置 keepalive	k_k	k_nk
	不设置 keepalive	nk_k	nk_nk

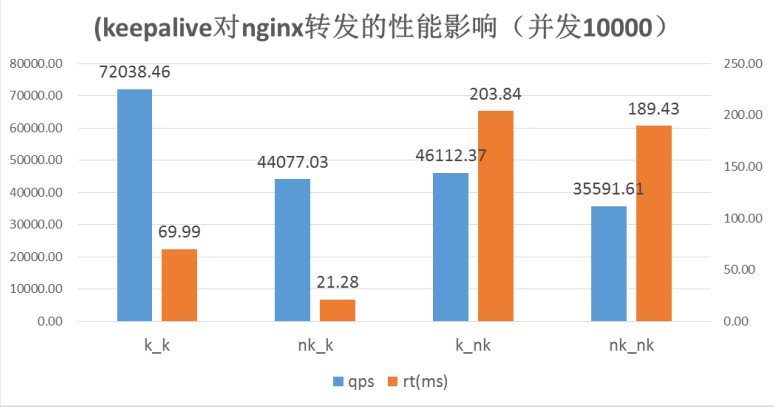
为表述方便，在后续的压测场景中，将以 k_k，nk_k，k_nk 和 nk_nk 来简单表述四种长连接（keepalive）设置，前一个 k 是指 client 与 proxy 间的 keepalive 设置，后一个 k 是指 proxy 和 upstream 间的 keepalive 设置。

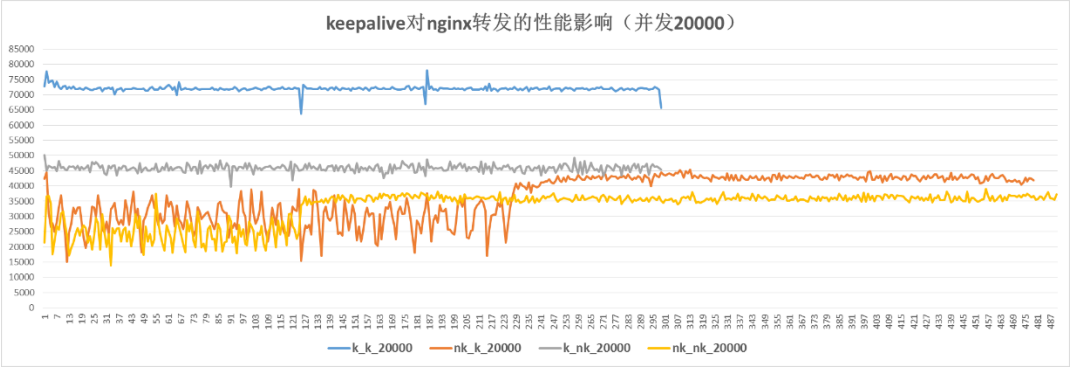
在并发量为 4000、10000 和 20000 的压力下，不同 keepalive 设置的原生 nginx 转发的性能表现分别如下图所示。

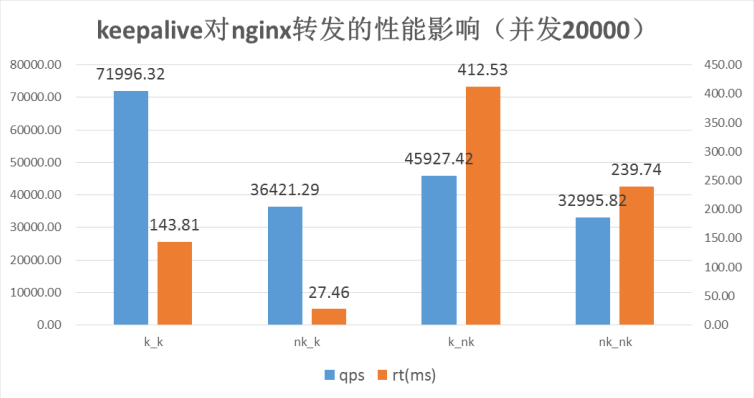

keepalive对nginx转发的性能影响（并发4000）
k_k_4000
nk_k_4000
k_nk_4000
nk_nk_4000


keepalive对nginx转发的性能影响（并发4000）
75737.46
26.04
39809.36
4.78
44527.42
84.10
31988.26
88.13
k_k
nk_k
k_nk
nk_nk
qps
rt(ms)


keepalive对nginx转发的性能影响（并发10000）
k_k_10000
nk_k_10000
k_nk_10000
nk_nk_10000


(keepalive对nginx转发的性能影响（并发10000）
72038.46
69.99
44077.03
21.28
46112.37
203.84
35591.61
189.43
k_k
nk_k
k_nk
nk_nk
qps
rt(ms)

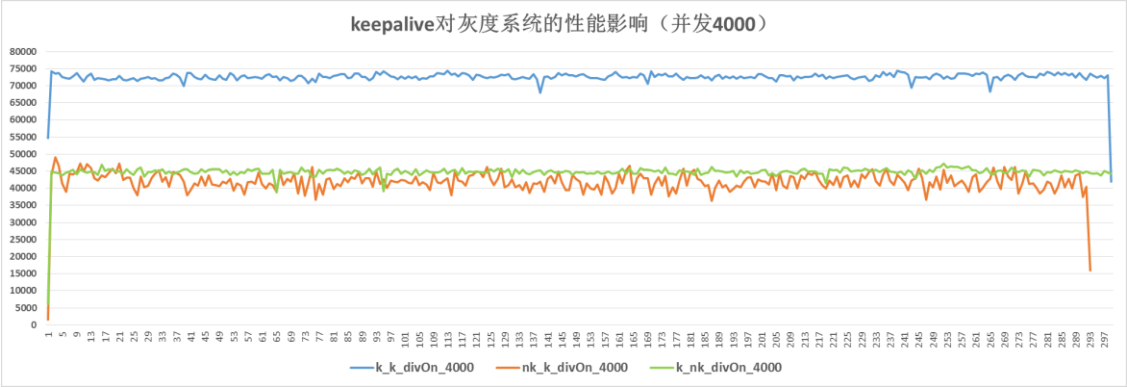


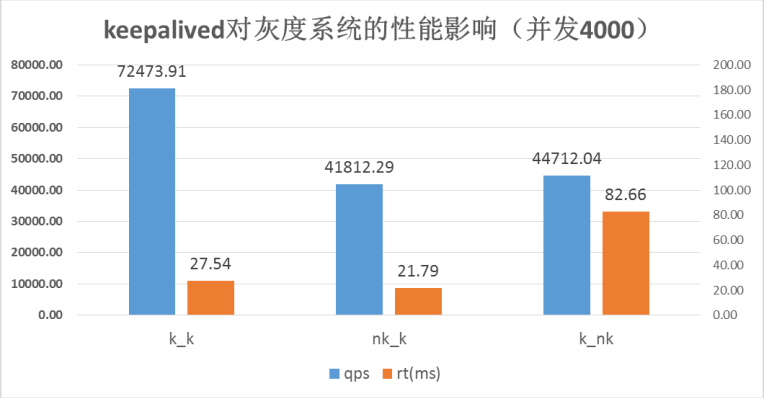


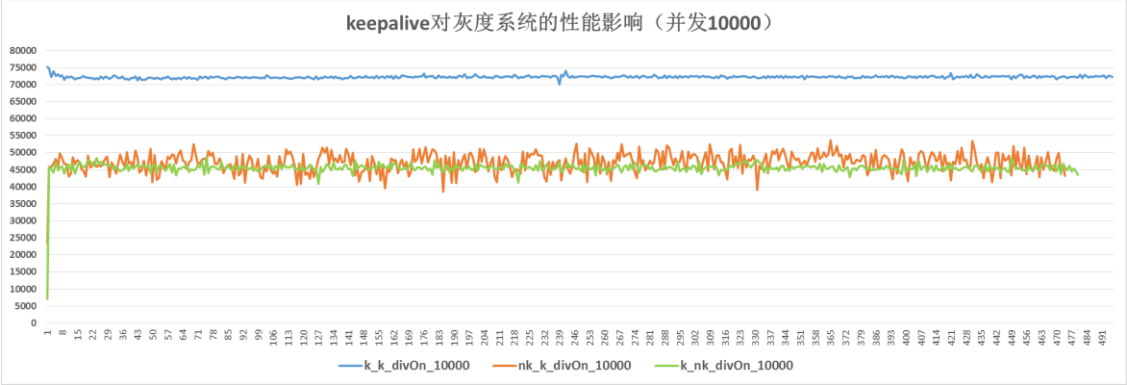
在并发压力较高的情况下，首先看到 k_k 和 k_nk 一直比较平稳，然后 nk_k 和 nk_nk 在刚开始的时候都不很稳定，运行一段时间后 qps 和 rt 趋于稳定。

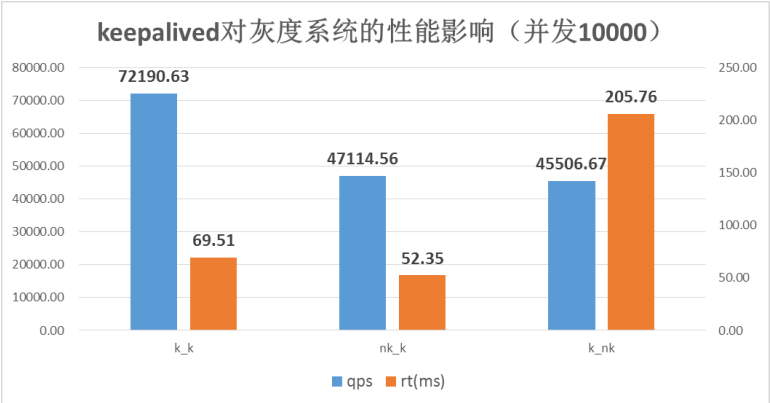
k_k 和 k_nk 的 qps 表现优于 nk_k 和 nk_nk，说明对外设置 keepalive 连接较为重要，因为这决定系统的对外服务能力。nk_k 的 qps 略高于 nk_nk，同时前者的 rt 表现远远好于后者，说明 proxy 与 upstream 间的 keepalive 设置对 7 层转发的响应时间非常重要。在 nginx 的 7 层服务中，proxy 与 upstream 大多处在内网的同一个网段中，所以我们默认是设置为 keepalive 的。（在压测情景中，压测工具发出的请求充分的利用了 proxy 对外的长连接，而实际情景中可能更偏向于 nk_k。）

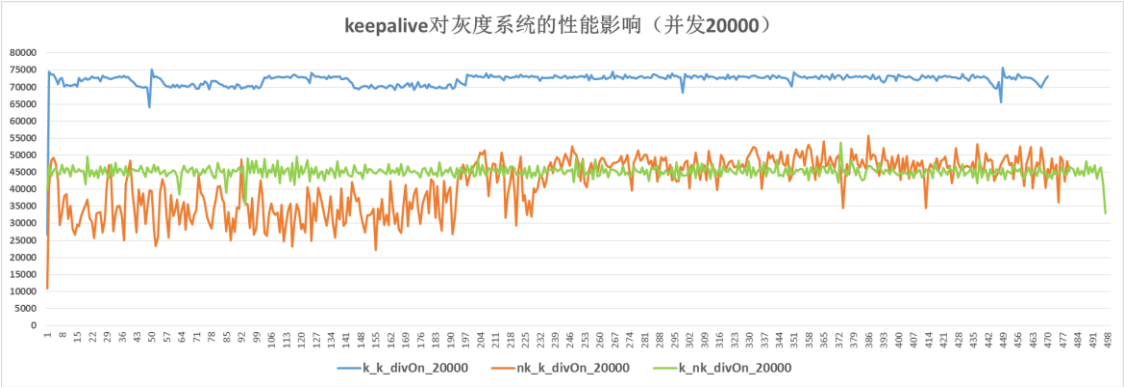
由于 keepalive 对于 nginx 的 7 层功能非常重要，我们关心 keepalive 设置对灰度系统性能的影响。在并发量为 4000、10000 和 20000 的压力下，灰度系统的性能表现分别如下图所示。

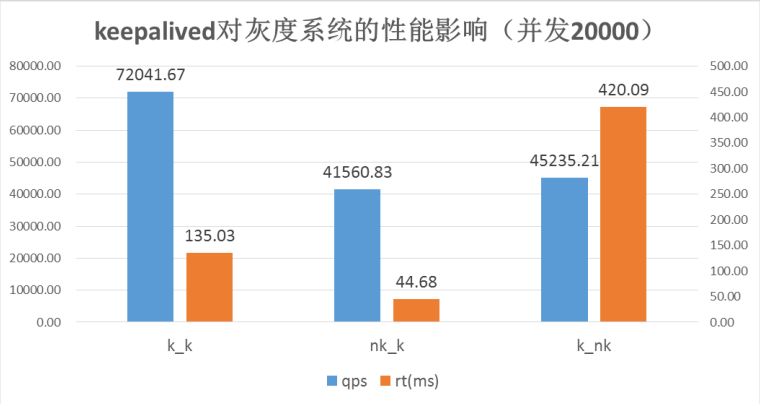

keepalive对灰度系统的性能影响（并发4000）
k_k_divOn_4000
nk_k_divOn_4000
k_nk_divOn_4000


keepalived对灰度系统的性能影响（并发4000）
72473.91
27.54
41812.29
21.79
44712.04
82.66
k_k
nk_k
k_nk
qps
rt(ms)

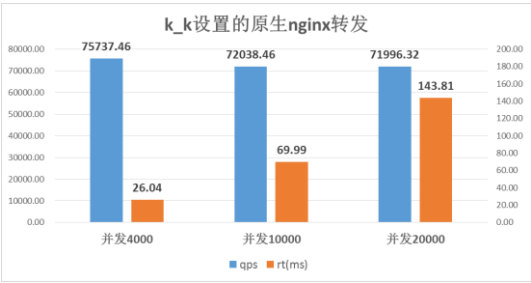

keepalive对灰度系统的性能影响（并发10000）
k_k_divOn_10000
nk_k_divOn_10000
k_nk_divOn_10000

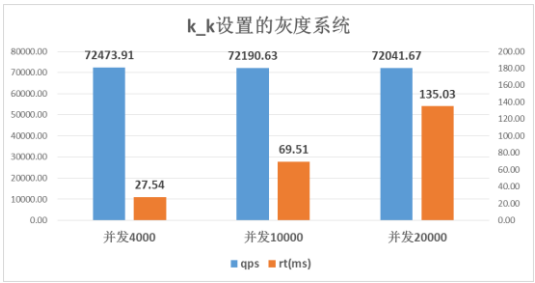

keepalived对灰度系统的性能影响（并发10000）
72190.63
69.51
47114.56
52.35
45506.67
205.76
k_k
nk_k
k_nk
qps
rt(ms)

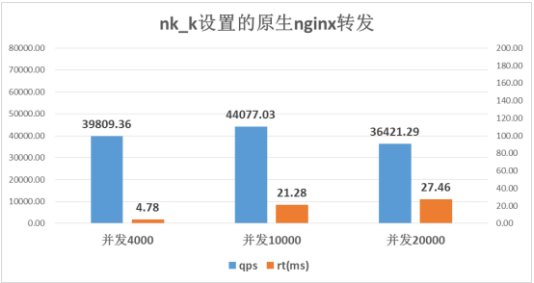


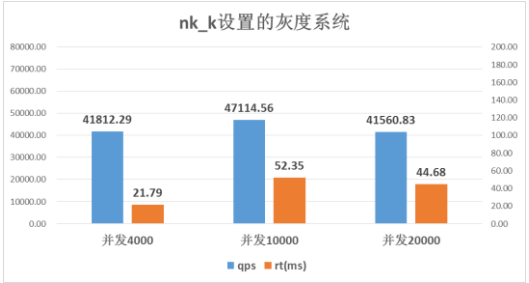


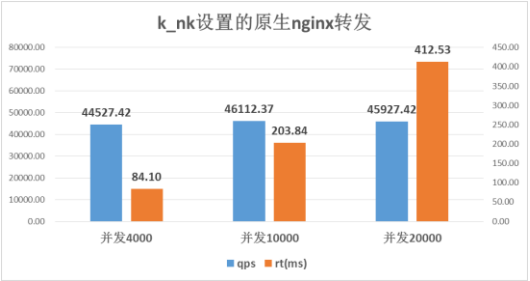
长连接 keepalive 设置对不同压力下的原生 nginx 转发和灰度系统的性能影响，如下图所示，左列为原生 nginx 转发的数据，右列为灰度系统的数据。

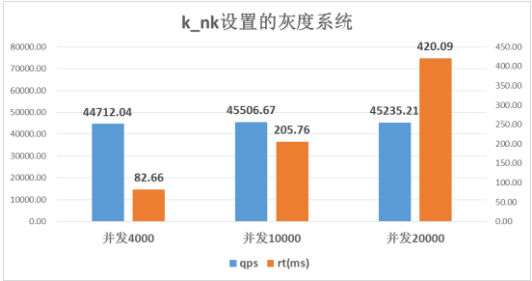








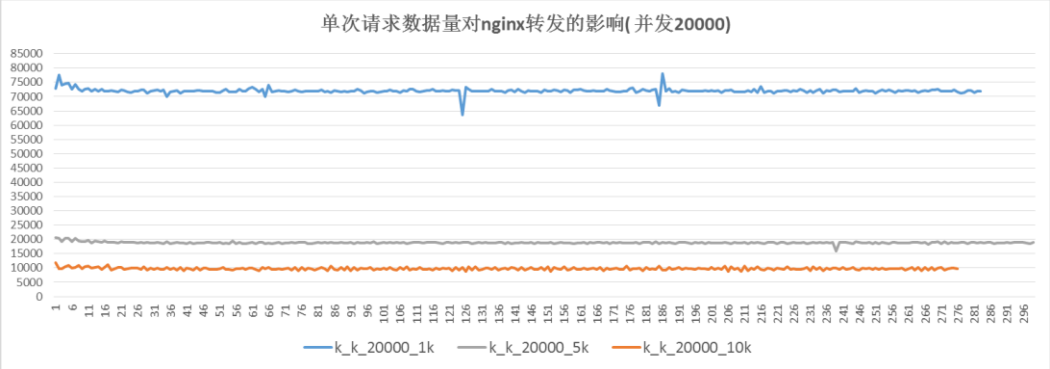


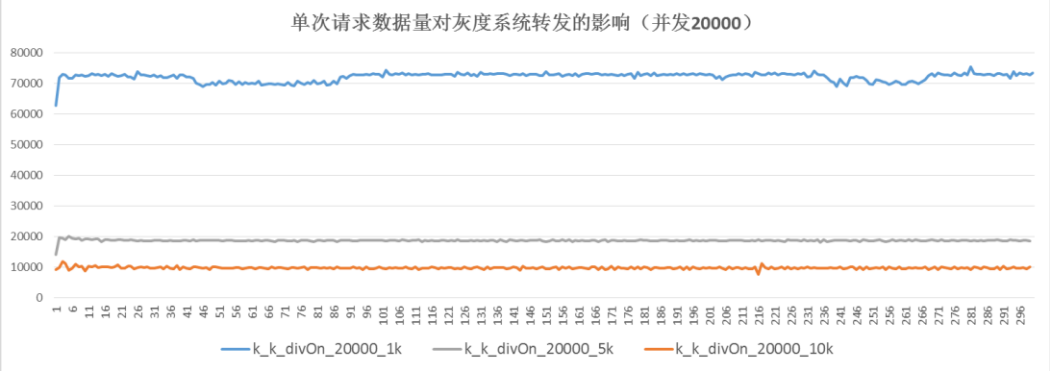


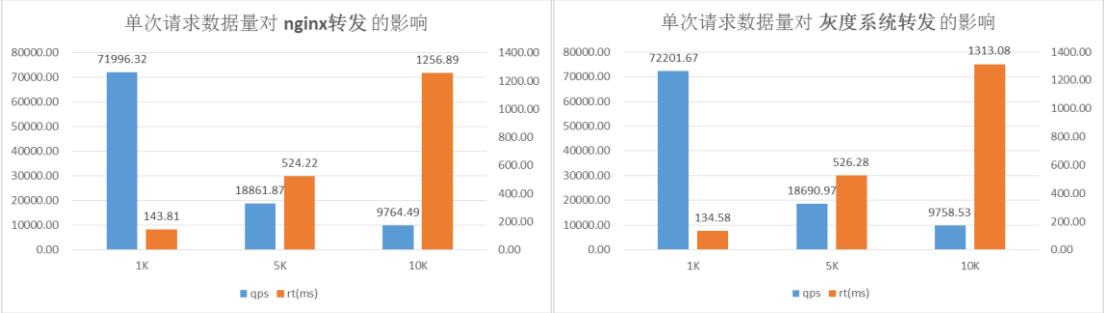
通过对比，可以认为在相同 keepalive 设置、相同并发压力的情况下，灰度系统的 qps 与原生 nginx 转发基本一致，因为灰度系统是在 nginx 转发的基础上增加了根据策略计算选择 upstream 的功能，ngx-lua 的高效率可以将这一功能的影响降到最低。

4.3 请求数据量大小对性能的影响

毫无疑问，单次请求数据量的大小对性能有至关重要的影响，主要体现在每次请求的数据发送量，以及硬件网卡的限制。由于目前压测环境中的瓶颈在于网卡是千兆网卡，流量在 120MB/s 多的时候就已经跑满，同时 cpu 相应网卡软中断率已经达到 30%++，基本可以认为达到机器极限。压测场景为每次请求数据为 1k、5k 和 10k，分别用 k_k 设置的 nginx 转发和 k_k 设置的灰度系统进行压测，压测结果如下图所示：



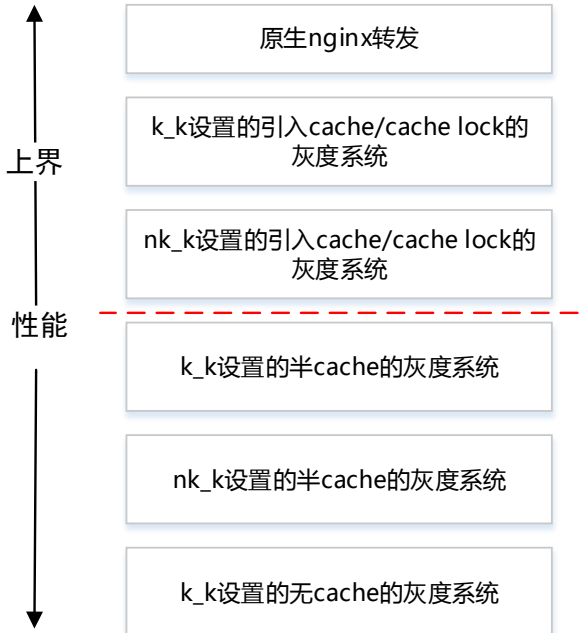




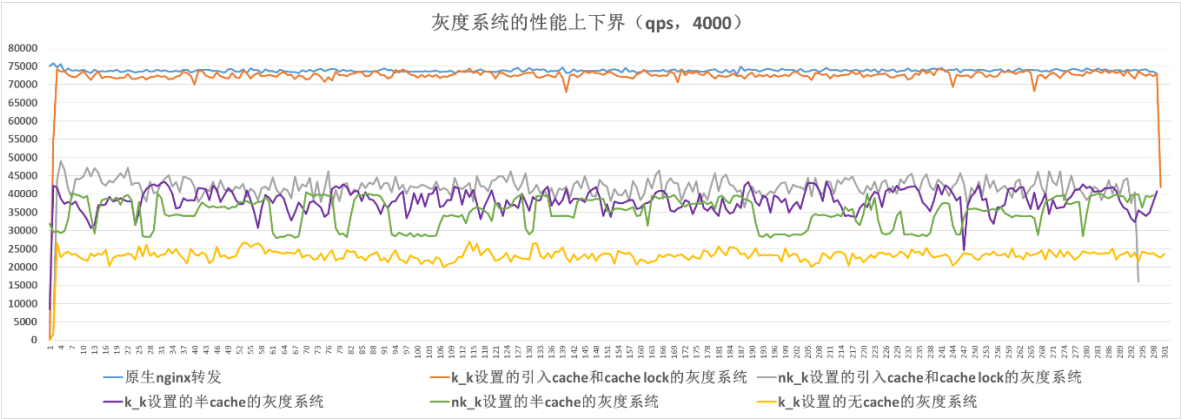
从图中看到由于达到网卡性能极限，请求数据量为 1k、5k 和 10k 时，qps 呈倍数下降的趋势。Nginx 转发与灰度系统在相同条件下表现一致。

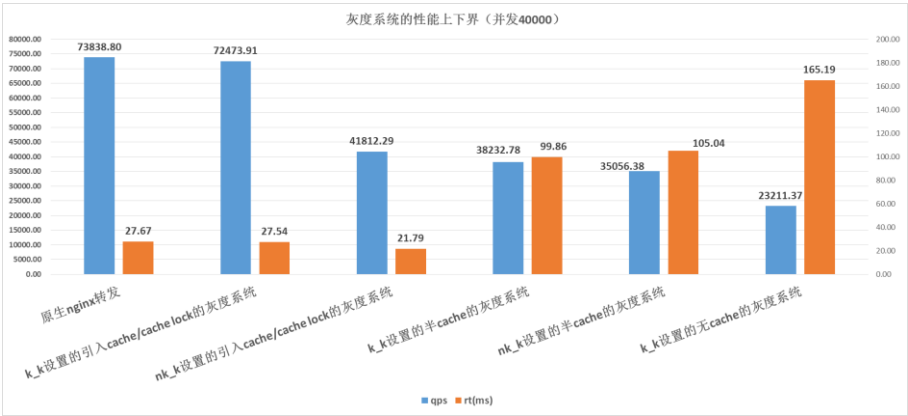
五、灰度系统的性能上下界

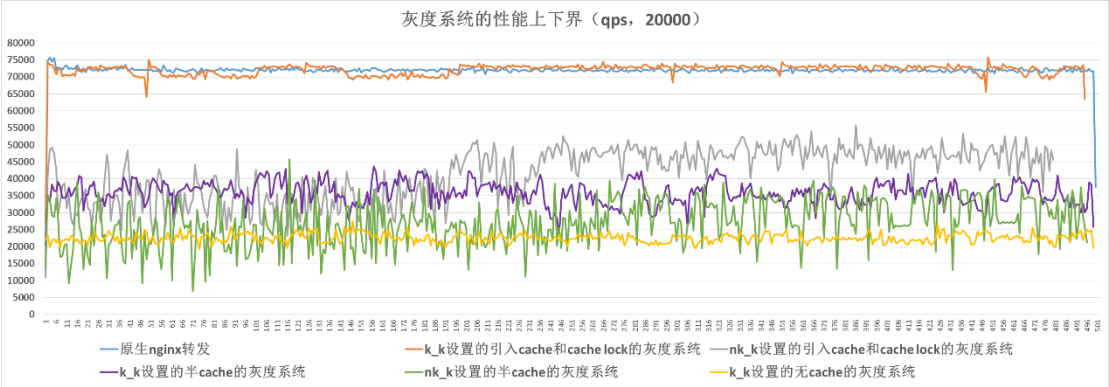
经过引入 cache 和 cache lock 进行优化，结合灰度系统的实际使用场景，我们可以基于当前系统的软硬件配置，设计压测场景，得出灰度系统的性能上下界。在压测过程中得出系统的性能上下界，对于评估和部署工作有很大帮助。

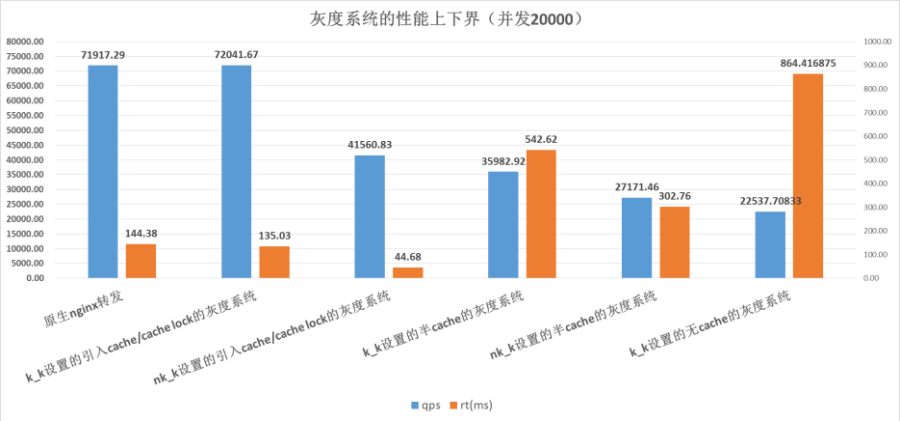


灰度系统的性能上界是原生 nginx 转发和压测场景下的 k_k 设置且引入 cache 的灰度系统，下界是没有 cache 的灰度系统。根据第三章各参数对灰度系统的性能影响，以及灰度系统的应用场景，灰度系统的性能处于 nk_k 设置时，全部使用 cache 和“半 cache”场景的性能之间。其性能表现如下图所示：









六、总结

当前灰度发布系统实现了基本功能，能够按照预期规划实现分流，动态更新策略。压测过程中的数据显示，灰度系统对原生 nginx 转发的影响比较小，启用灰度系统分流和启用原生 nginx 转发分流在相同场景下的性能十分接近。

压测过程测试出不同的因素对系统的影响，目前来看比较大的影响是 1、系统 sysctl 参数；2、nginx 配置参数；3、灰度系统各组件的配置；4、外部请求。

本次测试报告的不足之处在于：

1、 测试出不同因素的影响，但是不同因素之间的区别不能准确分析。尤其是在相同场景下，灰度系统性能比 nginx 直接转发要好的时候，一定是有其他因素导致，如能获得这种场景下的细节信息，应该能够分析得到一定结果。

2、 受限于知识储备，很多点没有关注到，比如 access.log 的 request_time 和 upstream_response_time 没有用到；wrk 压测工具的请求时间分布直方图没有用到；wrk 工具的高级特性和 lua 脚本没有充分用到；tsar 显示结果时可以指定-D 参数，输出具体数字，比如 74321，而不是 74k。

3、 缺少一种“流水线”式的测试和分析方法，跟踪一个用户请求从接入，到开始处理，到请求 upstream，到得到 upstream 响应，到返回给用户这条执行路径的完整情况；目前可以得到请求和 upstream 响应的整体时间，但是精细的方法还需要我们继续探索，比如通过 nginx plus 和 tengine 的未开源的优点，或者自己开发模块，或者充分利用日志。