基于动态策略的灰度发布系统

基于ngx_lua的一次实践

手机微博技术保障团队

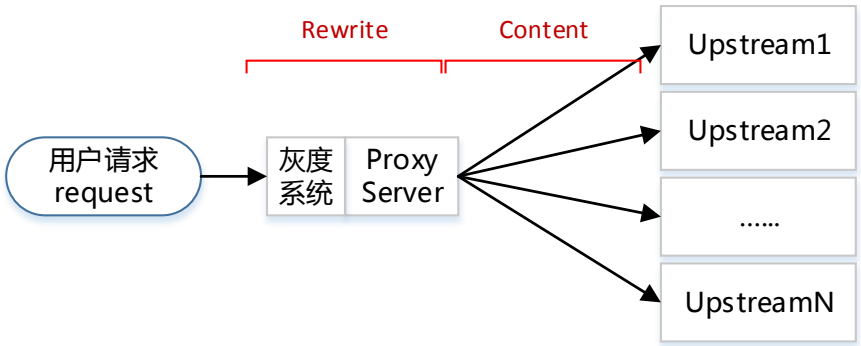
为什么需要动态的灰度发布系统

- 灰度发布系统
  - 需求大，方案多
  - 分流功能是关键，动态分流是**痛点**

- 应用场景：
  - 灰度上线、版本迭代（灰度的量灵活切换）
  - 特殊用户、特别版本（灰度的方式多样化）
  - 即时生效，无需重启

现有的灰度系统解决方案

- 现有方案基于nginx实现：
  - 基于proxy和upstream模块实现
  - nginx.conf中实现分流逻辑




```
location / {
        set $tag v;
        if ( $host ~* "(api|mapi)\.meibo\.(cn|com)") {
           set $tag "${tag}5";
        }
        if ( $request_uri ~* "^\/interface\/(i|f)\/") {
           set $tag "${tag}1";
        }
        if ( $arg_gsid != '' ) {
           set $tag "${tag}2";
        }
        if ( $tag = "v501" ) {
              proxy_pass http://v4;
              break;
        }
        if ( $arg_gsid = '' ) {
              proxy_pass http://v9-no-gsid;
              break;
        }
              proxy_pass http://v9-gsid;
        }
}
```

现有的灰度系统解决方案

- 方案优点：
  - 效率高
  - 维护简单、运维友好
  - 快速部署

- 存在问题：
  - 配置文件里各种if、set和rewrite容易出错
  - 重启生效
  - 不能实现太复杂的逻辑
  - 不能实现一些特殊分流方式

```
location / {
        set $tag v;
        if ( $host ~* "(api|mapi)\.meibo\.(cn|com)") {
           set $tag "${tag}5";
        }
        if ( $request_uri ~* "^\/interface\/(i|f)\/") {
           set $tag "${tag}1";
        }
        if ( $arg_gsid != '' ) {
           set $tag "${tag}2";
        }
        if ( $tag = "v501" ) {
              proxy_pass http://sv4;
              break;
        }
        if ( $arg_gsid = '' ) {
              proxy_pass http://sv9-no-gsid;
              break;
        }
              proxy_pass http://sv9-gsid;
        }
}
```

动态灰度系统的方案选型

- 针对上述缺点，我们采用ngx_lua来逐个解决
- ngx_lua简介
  - Openresty项目
  - 维护者：章亦春（@agentzh）

- ngx_lua原理
  - 将lua VM嵌入到nginx worker里，nginx各阶段lua都可以参与
  - nginx IO原语封装注入lua VM，lua可以发起IO请求
  - ngx_lua采用协程处理每个请求，协程间互相隔离，数据安全
  - lua调用IO操作不能立即完成时，将自身协程挂起，不阻塞worker
    - （by 陈于喆）

动态灰度系统的方案选型

- 选型ngx_lua的依据
  - ngx_lua完全非阻塞，高性能
  - lua表达能力强、实现复杂逻辑
  - ngx_lua得到了较为广泛的应用
  - Luajit的效率很高（by chaoslawful）

测试	Lua	LuaJIT	Java	PHP
fasta	7.02	0.8	0.42	27.96
nbody	58.6	1.34	0.96	143.42
spectral-norm	113.3	2.59	2.98	705.54
binary-trees	29.29	2.95	0.46	110.95
mandelbrot	59.71	1.8	1.05	219.55
fannkuchredux	193.31	5.4	2.66	639.50

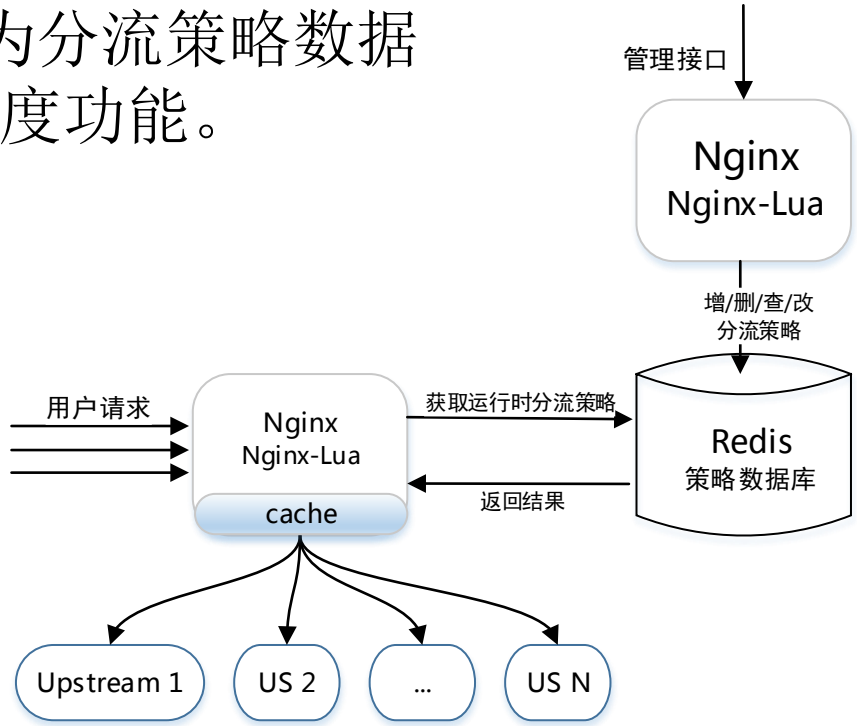
动态灰度系统的方案选型

- ngx_lua实现相同逻辑
  - access_by_lua
  - access_by_lua_file
  - rewrite_by_lua
  - …
  - content_by_lua
  - log_by_lua

```
upstream A_ups {
        server IP:PORT;
}
location /div {
        access_by_lua_file “/path/lualib/access.lua”

        set    $redis_conf      redis_conf;
        set    $server_conf     server_conf;
        set    $upstream        default_srv;

        rewrite_by_lua_block{
                lua code
                …
                local divModule =
                        require(“divModuleName”)

                ngx.var.upstream =
                        divModule.getUpstream(userInfo,
                                                divPolicy)
        };
        proxy_pass http://$upstream;
}
```

基于动态策略灰度发布系统

- ABTestingGateway 是一个可以动态设置分流策略的灰度发布系统，工作在7层，基于ngx_lua开发，使用 redis 作为分流策略数据库，可以实现动态分流和调度功能。



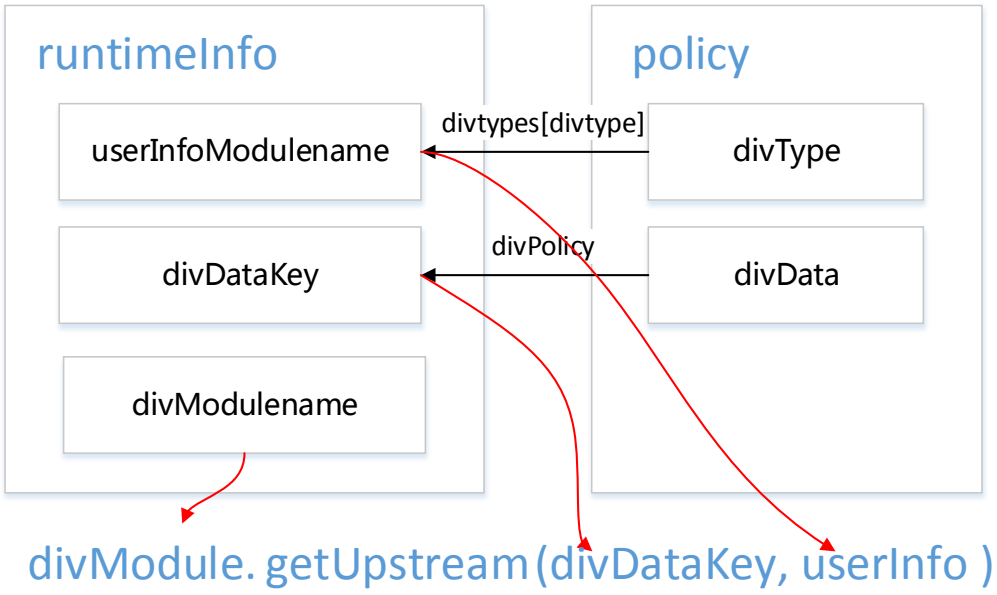
设计的关键

- 如何动态分流
  - **运行时** 设置分流方式
  - 分流方式
    - 分流依据
      - 用户请求特征userinfo，比如uid、ip或arg等
    - 分流策略
      - 分流类型 divtype，比如iprange分流类型、uid尾数类型…
      - 策略内容 divdata，以iprange分流类型为例：

```
{
"divtype":"iprange",
"divdata":[
{"range":{"start":1111, "end":2222}, "upstream":"beta1"},
{"range":{"start":3333, "end":4444}, "upstream":"beta2"},
{"range":{"start":5555, "end":6666}, "upstream":"beta1"},
{"range":{"start":7777, "end":8888}, "upstream":"beta3"}]
}
```

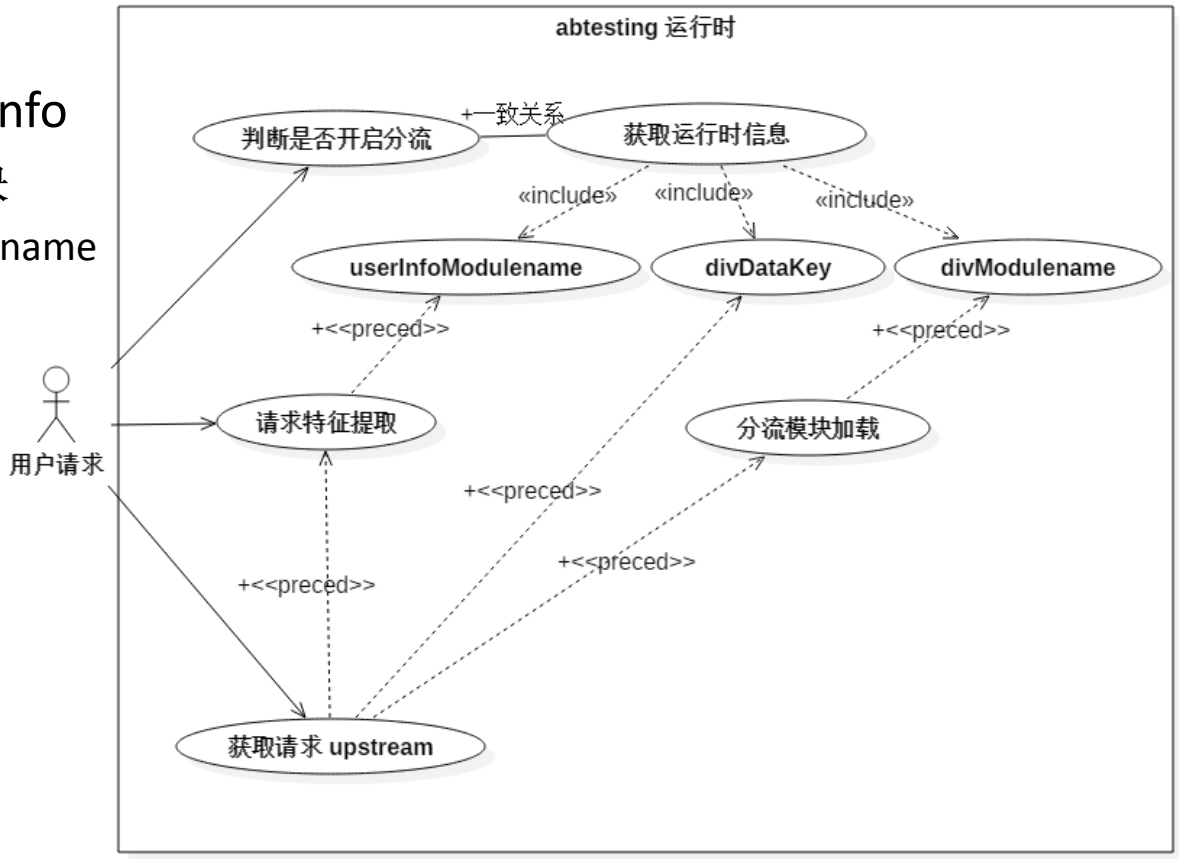
设计的关键

- 运行时设置：分流三要素
  - 分流模块　　divModule　　<- lua require(divModulename)
  - 用户请求特征　　userInfo　　<- 由用户特征提取模块获得
  - 分流策略　　divData　　<- 根据divDataKey从数据库获得



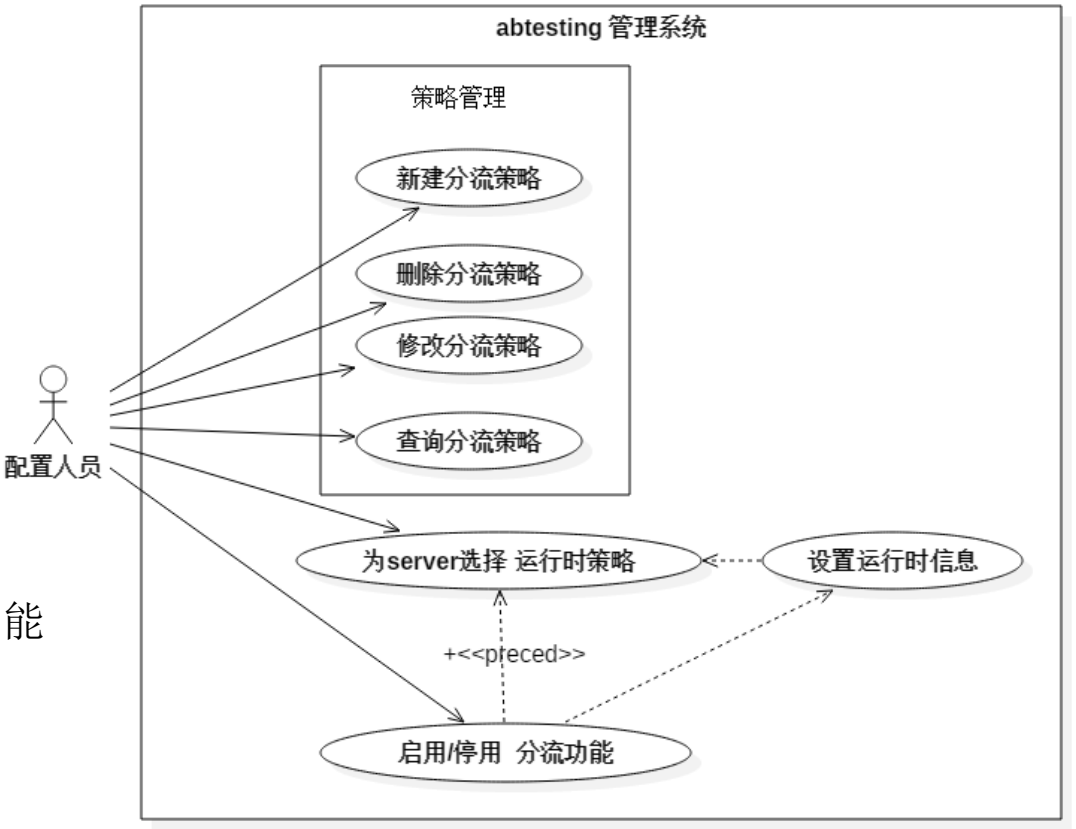
设计的关键

- 分流功能 用例图
  - 运行时信息runtimeInfo
    - 用户信息提取模块
      - userInfoModulename
    - 分流策略名
      - divDataKey
    - 分流模块
      - divModule
  - 提取用户信息
  - 加载分流模块
  - 获取分流后端upstream



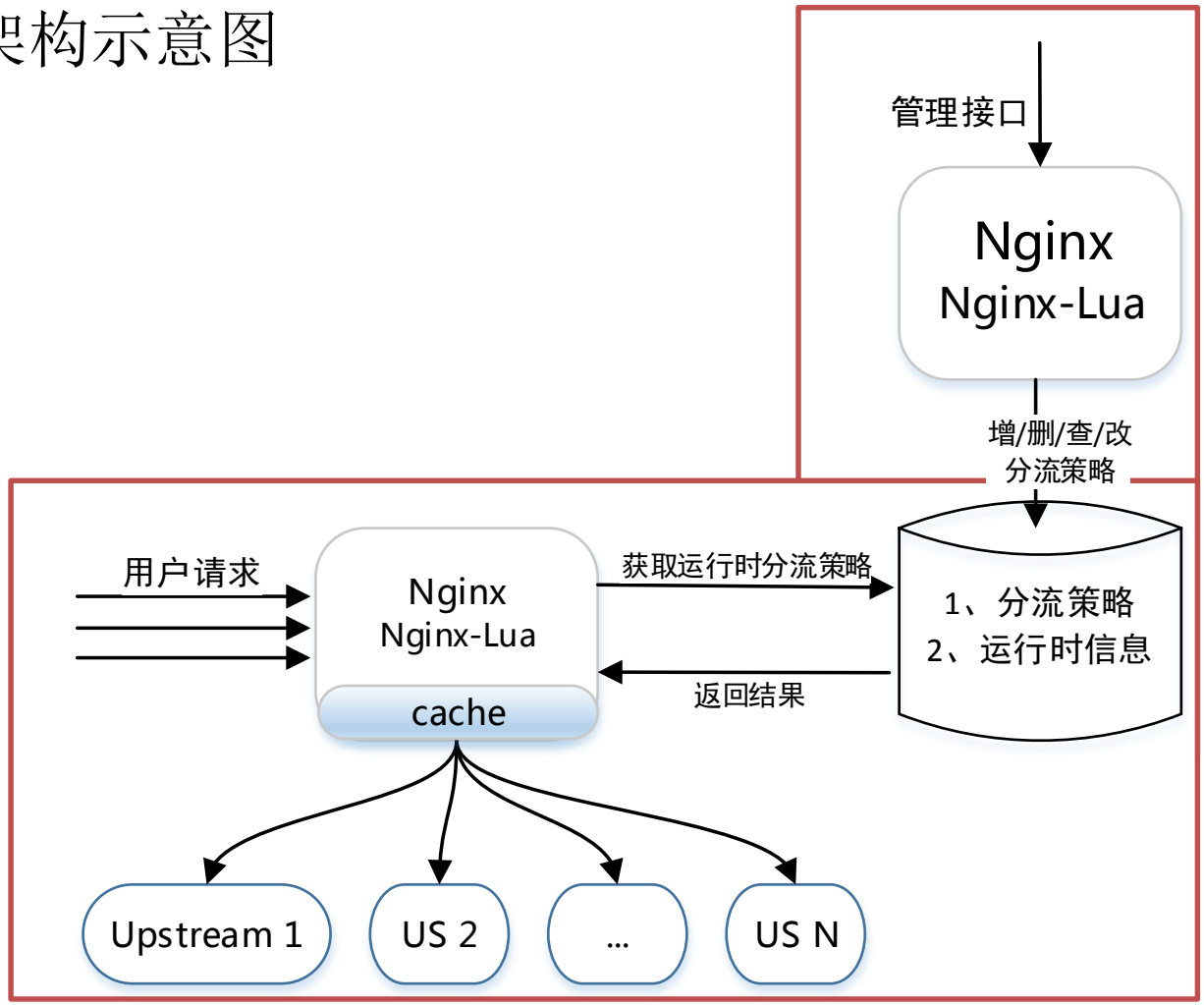
设计的关键

- 管理功能 用例图
  - 策略管理
    - 增/删/改/查
  - 运行时信息管理
    - 指定 运行时策略
    - 设置分流三要素
      - 启用/停用 分流功能
  - 系统配置和运维
    - nginx.conf
    - redis.conf



设计的关键

- 系统架构示意图



实现的细节

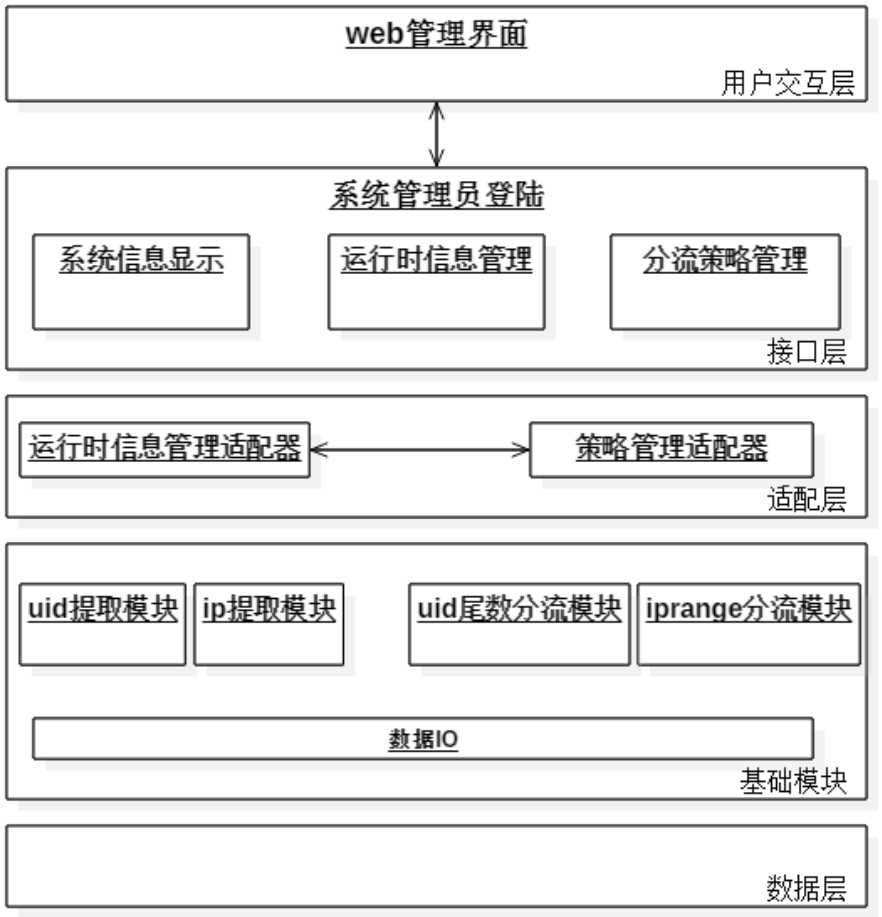
- 架构设计
  - 管理功能
  - 分流功能

实现的细节

- 架构设计——模块化设计
  - 各功能模块的接口
    - 策略管理模块　policyModule　：check/set/get/del
    - 运行时信息模块　runtimeInfoModule　：get/set/del
    - 分流模块　divModule　：getUpstream
    - 用户特征提取模块　userInfoModule　：get
  - 抽象出适配层
    - adapter.policy.set　->　iprange.set
    - divModule.getUpstream　->　iprange.getUpstream
    - userInfoModule.get　->　ipParser.get
  - 适配层向上提供统一接口，向下调用具体模块
    - 向系统添加新的分流方式，只需完整实现各功能接口即可

实现的细节

- 管理功能 架构设计



对外接口（nginx conf）
/admin/policy/check
/admin/policy/set
/admin/policy/get
/admin/policy/del

/admin/runtime/set
/admin/runtime/get
/admin/runtime/del

适配层模块（lua module）
adapter.pocliy.check
adapter.policy.set
adapter.policy.get
adapter.policy.del

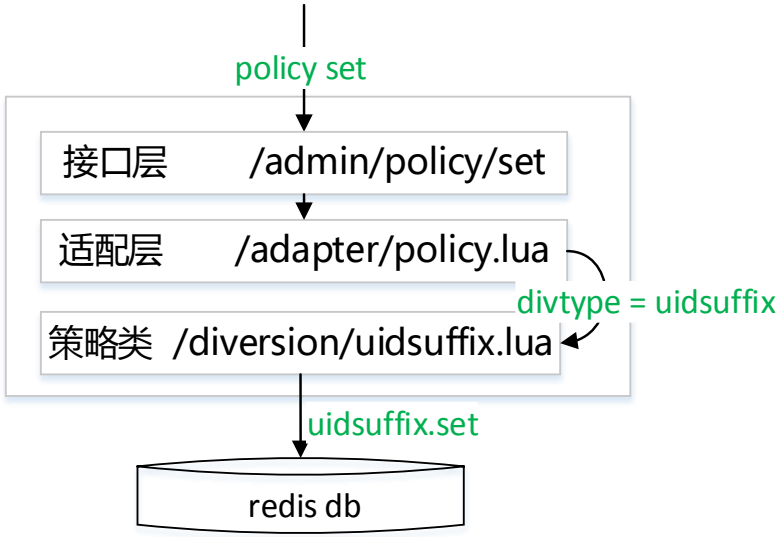
例如：iprange策略模块（lua module）
diversion.iprange.check
diversion.iprange.set
diversion.iprange.get
diversion.iprange.del

Redis数据库
lua-resty-redis

实现的细节

- 接口设计与实现
  - 管理接口
    - 策略管理配置

```
location = /admin/policy/set {
        content_by_lua_file '../admin/policy/set.lua';
}
location = /admin/policy/get {
        content_by_lua_file '../admin/policy/get.lua';
}
location = /admin/policy/del {
        content_by_lua_file '../admin/policy/del.lua';
}
location = /admin/policy/check {
        content_by_lua_file '../admin/policy/check.lua';
}
```




```
130> curl 127.0.0.1:8030/admin/policy/set -d '{"divtype":"uidsuff
ix","divdata":[{"suffix":"1","upstream":"beta1"},{"suffix":"3","u
pstream":"beta2"},{"suffix":"5","upstream":"beta1"},{"suffix":"0"
,"upstream":"beta3"}]}'
{"errcode":200,"errinfo":"success  the id of new policy is 6"}
```

实现的细节

- 接口设计与实现
  - 管理接口
    - 运行时管理配置

```
0> curl 127.0.0.1:8030/admin/runtime/set?policyid=6
{"errcode":200,"errinfo":"success "}
```

```
location = /admin/runtime/set {
        content_by_lua_file '../admin/runtime/set.lua';
}
location = /admin/runtime/get {
        content_by_lua_file '../admin/runtime/get.lua';
}
location = /admin/runtime/del {
        content_by_lua_file '../admin/runtime/del.lua';
}
```

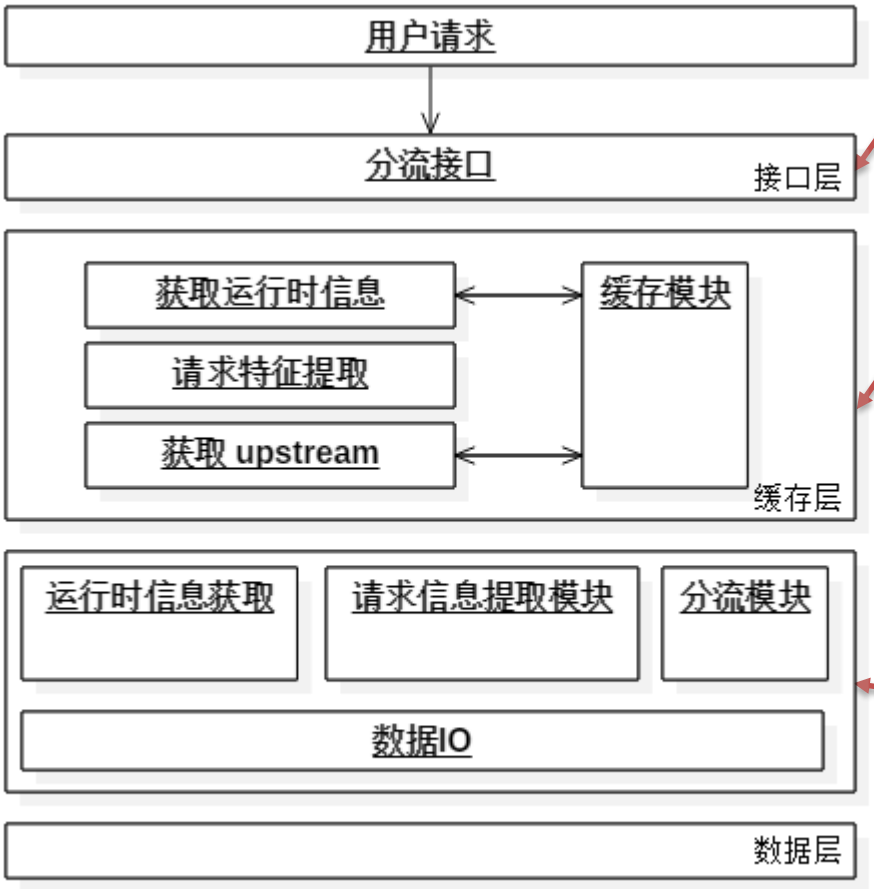
    - 运行时信息：分流三要素

```
127.0.0.1:6379> keys ab:test:runtimeInfo*
1) "ab:test:runtimeInfo:localhost:divModulename"
2) "ab:test:runtimeInfo:localhost:userInfoModulename"
3) "ab:test:runtimeInfo:localhost:divDataKey"
```

```
abtesting.diversion.uidsuffix
abtesting.userinfo.uidParser
ab:test:policies:6:divdata
```

实现的细节

- 分流功能 架构设计



分流接口（nginx conf）
location / {}
或者
location ~ /div {}

缓存模块（ngx-shared-dict）

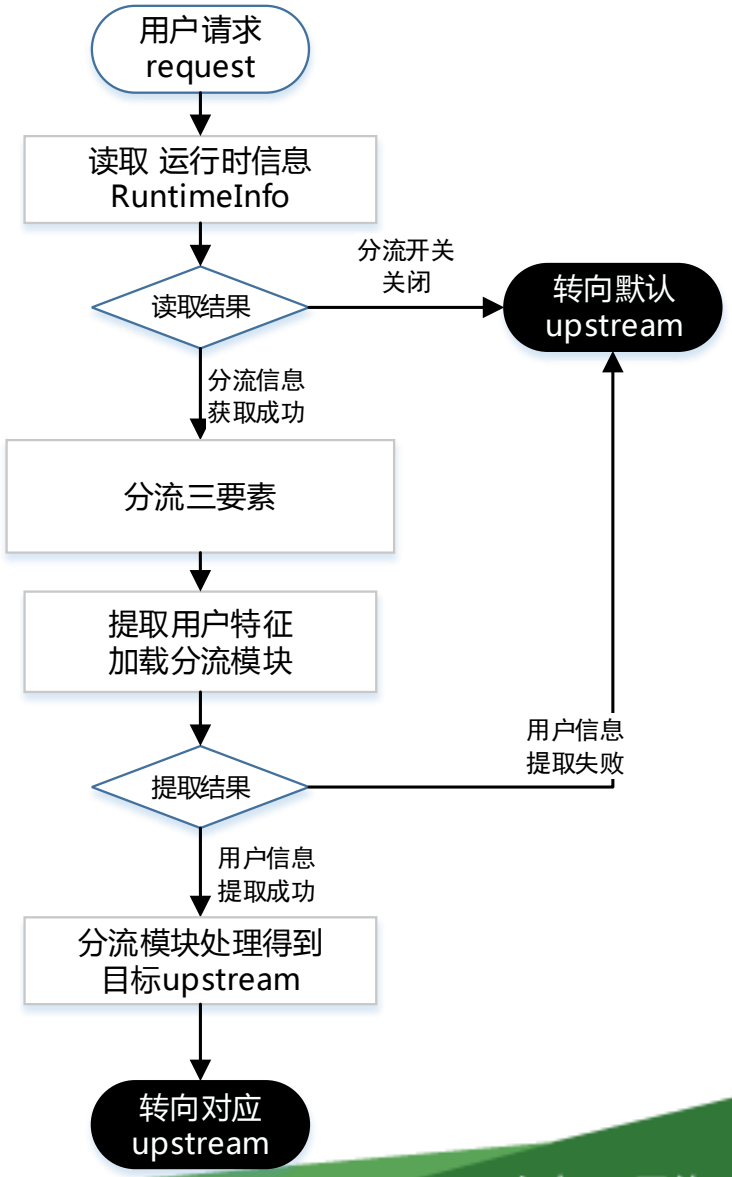
缓存结果
userInfo : upstream

缓存运行时信息
divModuleName : divModule
userInfoModuleName : userInfoMod
divPolicyKey : divPolicy

缓存失效时
1、runtimeModule.getRuntimeInfo()
get_divPolicyKey
require(divModule)
require(userInfoModule)
2、divModule.getUpstream(userInfo)
3、setCache()

实现的细节

- 分流过程流程图
  - 先读取运行时信息，再进行分流工作
  - 分流方式变更后，无需重启，即时生效。



实现的细节

- 功能设计
  - 特征提取
  - 策略查询
  - 错误处理
  - 缓存设计

实现的细节

- 功能设计——特征提取
  - 读写nginx变量（内建变量、自定义变量）
    - ngx.var.variable
      - 灰度系统的配置参数
  - 获取GET参数
    - ngx.var.arg_variable
      - 例如，获取req的tag，ngx.var.arg_tag
  - 获取POST请求体数据
    - ngx.var.request_body
  - 获取请求头部
    - ngx.req.get_headers()
      - 返回table
      - 例如，获取真实IP：X-Real-IP或X-Forwarded-For

实现的细节

- 功能设计——策略查询
  - getUpstream(divPolicy, userInfo)
    - 基于redis设计
    - 策略类型 决定 策略的存储方式
  - 哈希查找
    - uid尾数分流
    - HTTP请求参数分流
  - 区间查找
    - ip段分流
    - uid段分流

实现的细节

- 功能设计——策略查询
  - 哈希类策略，以uid尾数分流为例
    - 左图为uid尾数分流策略
    - 右图为该策略在redis中以hset类型存储
    - getUpstream时以KV方式获取

```
{
"divtype":"uidsuffix",
"divdata":[
{"suffix":"1", "upstream":"beta1"},
{"suffix":"3", "upstream":"beta2"},
{"suffix":"5", "upstream":"beta1"},
{"suffix":"0", "upstream":"beta3"}]
}
```

```
127.0.0.1:6379> get ab:test:policies:0:divtype
"uidsuffix"
127.0.0.1:6379> hgetall  ab:test:policies:0:divdata
1) "1"
2) "beta1"
3) "3"
4) "beta2"
5) "5"
6) "beta1"
7) "0"
8) "beta3"
127.0.0.1:6379>
```

实现的细节

- 功能设计——策略查询
  - 区间类策略，以iprange分流为例
    - iprange分流策略

```
{
"divtype":"iprange",
"divdata":[
{"range":{"start":1111, "end":2222}, "upstream":"beta1"},
{"range":{"start":3333, "end":4444}, "upstream":"beta2"},
{"range":{"start":5555, "end":6666}, "upstream":"beta1"},
{"range":{"start":7777, "end":8888}, "upstream":"beta3"}]
}
```

    - 以Redis的zset结构存储
      - Zset的key格式为
        - » [2n]:beta1　偶数为区间开始
        - » [2n+1]:beta1 奇数为区间结束
      - IP采用32位整型表示，作为score
      - zrangebyscore policy queryIP +inf 0 1
      - 当queryIP在某个ip段里时，zrangebyscore的结果一定是[2n+1]:upstream，否则是[2n]:upstream

```
zset
key      score

0:beta1 1111
1:beta1 2222

2:beta2 3333
3:beta2 4444

4:beta1 5555
5:beta1 6666

6:beta3 7777
7:beta3 8888
```

实现的细节

- 功能设计——错误处理
  - xpcall()
    - Lua的代码保护执行机制
    - arg1 pfunc是被保护执行的函数
    - arg2 handler是异常捕捉函数
    - 返回值status表示执行状态
      - 若status为true，则info为pfunc的正常返回结果
      - 若status为false，则info为handler返回的错误信息

```
78 local pfunc = function()
79     policyMod = policyModule:new(red.redis, policyLib)
80     return policyMod:check(policy)
81 end
82
83 local status, info = xpcall(pfunc, handler)
84 if not status then
85     local errinfo  = info[1]
86     local errstack = info[2]
87     local err, desc = errinfo[1], errinfo[2]
88     local response      = doresp(err, desc)
89     dolog(err, desc, nil, errstack)
90     ngx.say(response)
91     return
92 end
```

实现的细节

- 功能设计——错误处理
  - Lua代码的异常
    - Lua runtime error
      - 错误信息
      - 调用栈

```
0> lua
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
> s1 = "hello"
> s2 = nil
> s3 = s1 .. s2
stdin:1: attempt to concatenate global 's2' (a nil value)
stack traceback:
        stdin:1: in main chunk
        [C]: ?
```

    - error主动抛出异常
      - error抛出异常信息
      - 调用栈

```
16 _M.new = function(self, database, policyLib)
17     if not database then
18         error{ERRORINFO.PARAMETER_NONE, 'need avaliable redis db'}
19     end if not policyLib then
20         error{ERRORINFO.PARAMETER_NONE, 'need avaliable policy lib'}
21     end
22
23     self.database = database
24     self.policyLib = policyLib
25     return setmetatable(self, mt)
26 end
```

实现的细节

- 功能设计——错误处理
  - Handler函数设计
    - 配合xpcall()
    - handler捕获系统所有异常
    - handler同时返回异常发生的调用栈
    - errorcode的设计见/path/lib/abtesting/error/errcode.lua
    - handler函数对异常信息的统一收集，可以输出同格式的log和resp

```
--将doresp和dolog，与handler统一起来。
--handler将返回一个table，结构为：
--[[
handler- - - errinfo- - - - errcode- - - - code
    |           |               |
    |           |               |- - - - info
    |           |
    |           |- - - - errdesc
    |
    |
    |
    |- - - errstack
]]--
```

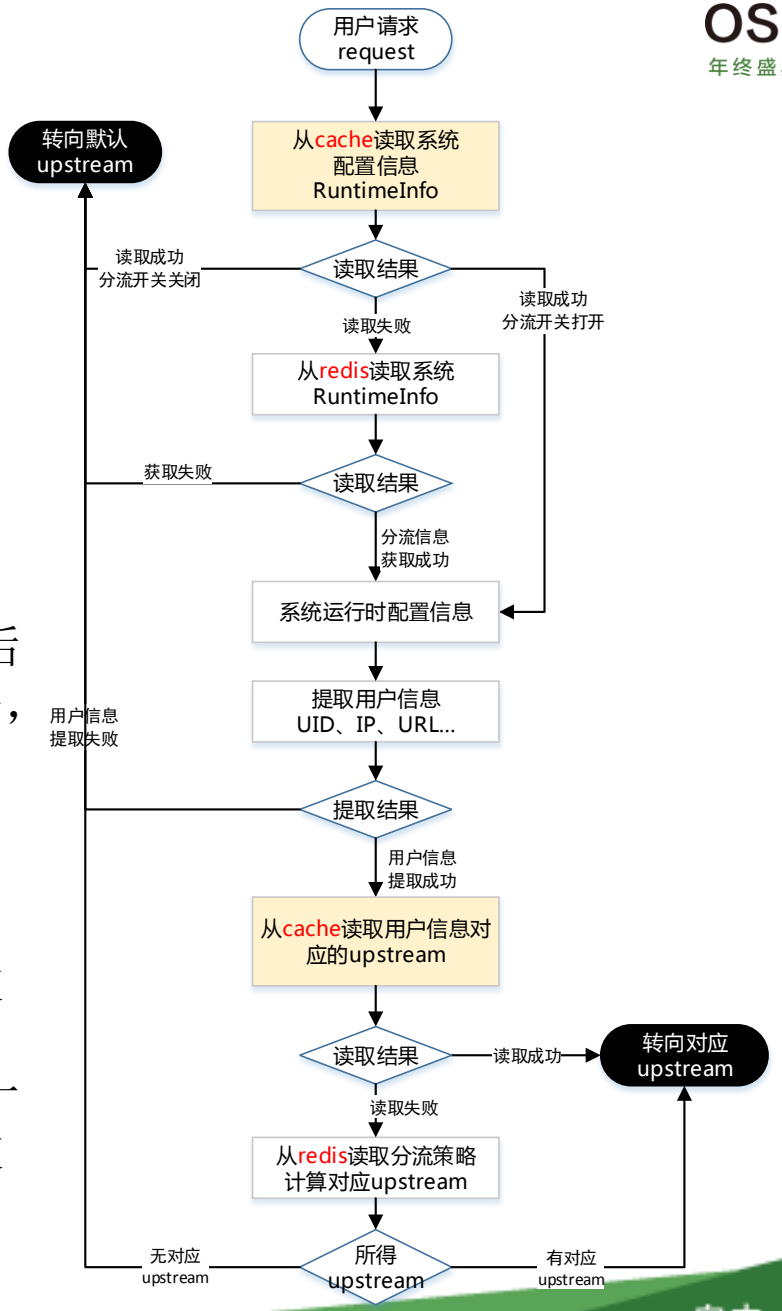
```
_M.handler = function(errinfo)
    local info
    if type(errinfo) == 'table' then
        info = errinfo
    elseif type(errinfo) == 'string' then
        info = {ERRORINFO.LUA_RUNTIME_ERROR, errinfo}
    else
        info = {ERRORINFO.BLANK_INFO_ERROR, }
    end

    local errstack = debug.traceback()
    return {info, errstack}
end
```

实现的细节

- 功能设计——缓存设计
  - 引入缓存
    - 一定时间内，系统分流方式不变
    - 分流方式不变时，同一用户的请求将被分流到同一后端
    - Redis查询涉及网络IO，用时漫长
  - ngx-shared-dict
    - ngx_lua的一个组件
    - 基于ngx-shm实现，KV存储
    - 跨worker共享，内部有锁实现
    - 内部基于ngx_rbtree实现，高效查找

实现的细节

- 引入cache后的流程图
  - 优先从cache中获取结果
  - 缓存失效机制
    - expire 设置为 60s，超时后需要再次从redis中取数据，保证缓存更新状态
  - 改变分流方式时的不一致
    - 当数据库中的分流方式改变时，如果此时60s内的cache未失效，会造成不一致，允许短期少量不一致



系统的优化

- 缓存及缓存锁优化
- Nginx配置优化
- 多队列网卡的中断绑定
- Redis的配置优化

系统的优化

- 1、引入缓存及缓存锁优化
  - ngx-shared-dict
  - lua-resty-lock
    - 避免dog-pile效应，缓存失效时，先拿锁再访问数据库
    - 防止缓存失效时对数据库压力过大
    - lock时间可配置

系统的优化

- 2、Nginx的配置优化

worker_processes auto	tengine的根据系统cpu数适配worker数
accept_mutex off; multi_accept on;	允许多worker竞争处理，忽略“惊群效应” 允许worker一次处理多个连接
lua_code_cache on;	允许缓存lua code，不用每次读取lua文件（对ngx_lua应用来说非常重要）
#keepalive_timeout 0; #keepalive_requests 0;	Nginx对client的keepalive设置，默认分别为75s和100个。其中一个为0则表示不启用keepalive设置

http{
proxy_set_header Connection "";
proxy_http_version 1.1;
}
upstream{
keepalive 16;
}
	Proxy sever与upstream server的keepalive设置，对7层转发很重要，避免频繁的与后端server进行短连接

系统的优化

- 3、网卡软件中断均衡配置
  - 将多队列网卡的中断请求与固定的CPU核心绑定，减少上下文切换，充分利用多核CPU的优势
  - https://github.com/chunshengster/scripts/blob/master/nic_smp_affinity_set.sh
- 4、redis 配置优化

unixsocket unixsocketperm	/tmp/redis.sock 766	redis的unix domain socket连接设置
timeout	0	redis不主动关闭用户连接
tcp-keepalive	120	redis的keepalive设置，作用于tcp连接
tcp-backlog	20000	redis的backlog设置，可作用于**tcp**和**uds**的listen队列

系统的性能

- 灰度系统的压测与性能监控
  - 压测工具
    - 模拟真实场景
    - 输出足够压力
    - 备选方案：Ab、httperf、http_load、wrk
  - 监控工具
    - 反映真实情况
    - 实时性
    - 最小的增加系统负担
    - 方案：stub_status module、Tsar（同时监控linux、nginx、nic，真实和实时）

```
0> tsar -li1
Time              ---cpu-- ---mem-- ---tcp-- -----traffic---- --sda---  ---load- ------------------nginx-----------------
Time                util     util   retran     bytin  bytout     util     load1      qps       rt  sslqps  spdyps  sslhst
04/12/15-11:18:15  24.10    47.12     0.00    698.00  712.00     0.00     0.05     1.00     0.00    0.00    0.00    0.00
04/12/15-11:18:16  23.59    47.12     0.00     70.00  230.00     0.00     0.05     1.00     0.00    0.00    0.00    0.00
```

系统的性能

- 压测工具选择

压测工具	多线程	原理	优点	缺点
Ab	否	epoll	可以持续提供最高20k的并发，能够打印rt的详细分布，性能较高，占用资源较少	功能较单一，适合做快活。
Httperf	否	select	引入session、rate等概念，可以指定规律，模拟用户打开网页的时间规律，模拟真实场景进行压力测试	太慢，性能较差，适合做细活。
http_load	否	select	快捷，功能基本与ab一致。可发出多个不同请求，和绑定多个本机ip	压测结果太简单
Wrk	是	epoll	1. 性能高，采用与redis一样的事件循环机制。 2. 使用多线程充分利用机器性能 3. 引入lua脚本实现丰富功能。	

系统的性能

- 压测工具选择
  - Wrk的{多进程+epoll}可以提供足够的压力
  - Wrk内嵌lua非常有用，我们发出大量http请求时，wrk可以通过lua动态设置请求中的header、arg等参数，对于我们要模拟真实场景非常有用。Lua需要资源非常少。

```
59 request = function()
60     ip = various_ip
61     val = various_val
62     path = "/uri?val=" .. val .. "&ip=".ip
63     return wrk.format(nil, path)
64 end
```

系统的性能

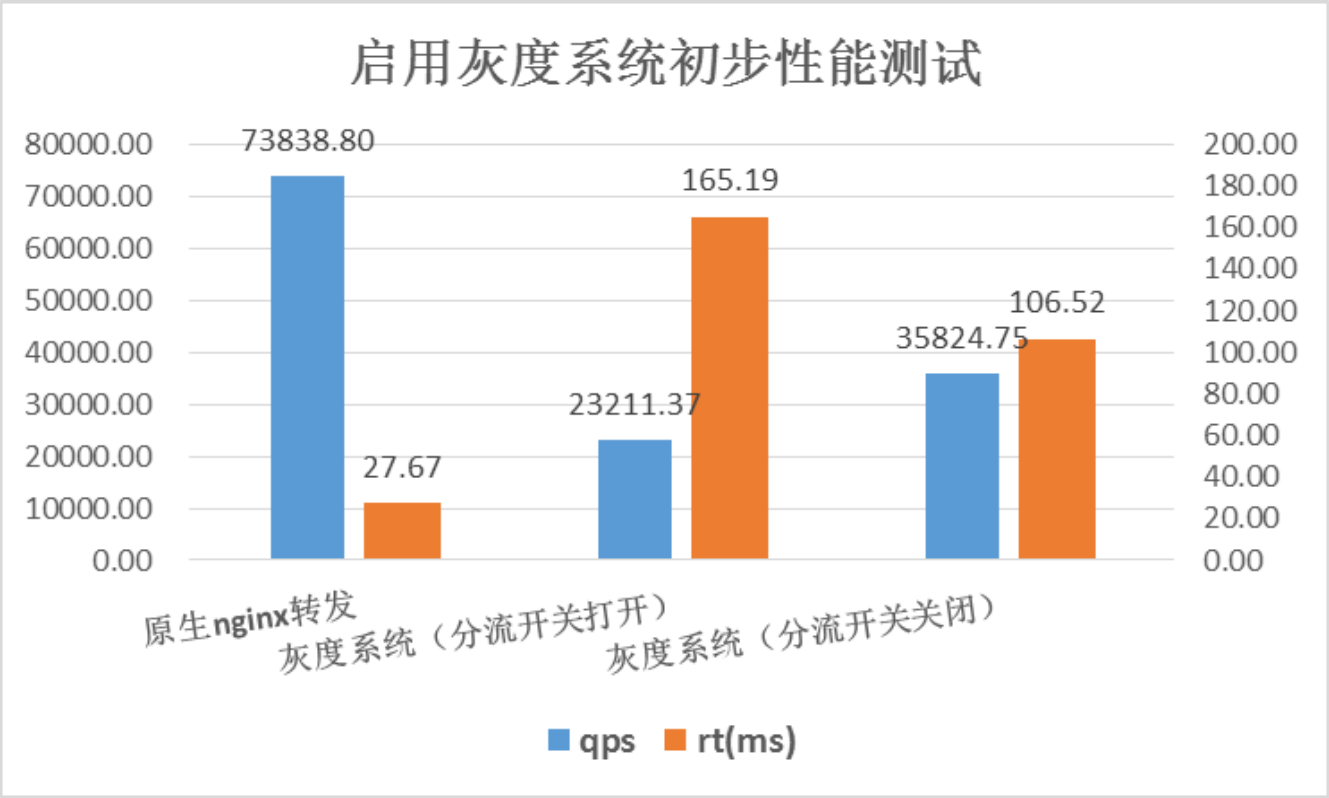
- 硬件环境
  - Intel(R) Xeon(R) CPU E5620 @ 2.40GHz 16核
  - MEM：24GB
  - 千兆网卡
- 软件环境
  - CentOS-6.5 kernel v2.6.32
  - Tengine-2.1.0
  - LuaJIT
  - ngx_lua-0.9.6
- 压测设置
  - 多种并发量（400、10000、20000等）
  - 持续一段时间（200s、500s…）
  - 主要关注qps和rt
  - 相同条件下，与原生nginx转发进行比较，作为参考

系统的性能

- 压测场景
  - 主要影响因素
    - Cache
    - Keepalive
  - 其他影响因素
    - 并发量
    - 请求数据量大小
    - nginx配置
    - Sysctl参数

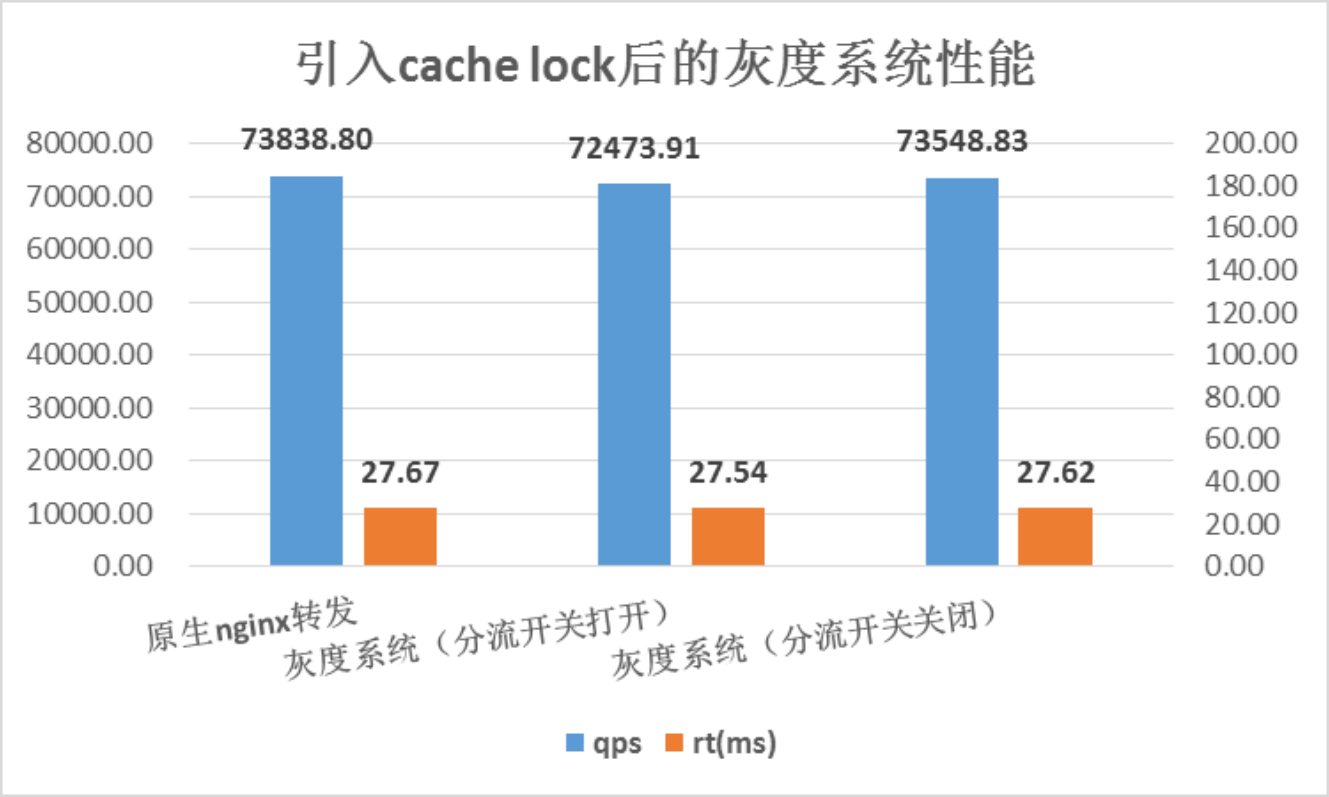
系统的性能

- 1、Cache的性能影响
  - 不启用Cache



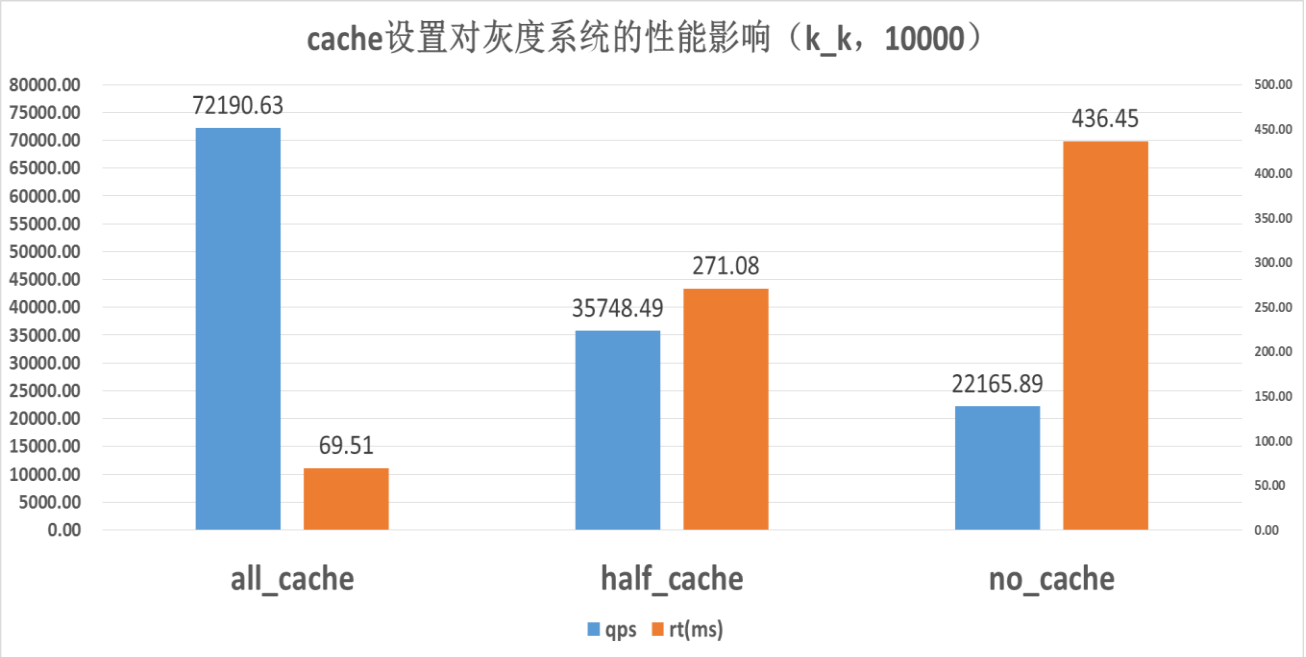
系统的性能

- 1、Cache的性能影响
  - 启用Cache，将**运行时信息**和**userinfo:upstream**都cache起来



系统的性能

- 1、Cache的性能影响
  - 启用Cache，只缓存**运行时信息**，不缓存**userinfo:upstream**
  - 称为“半cache”



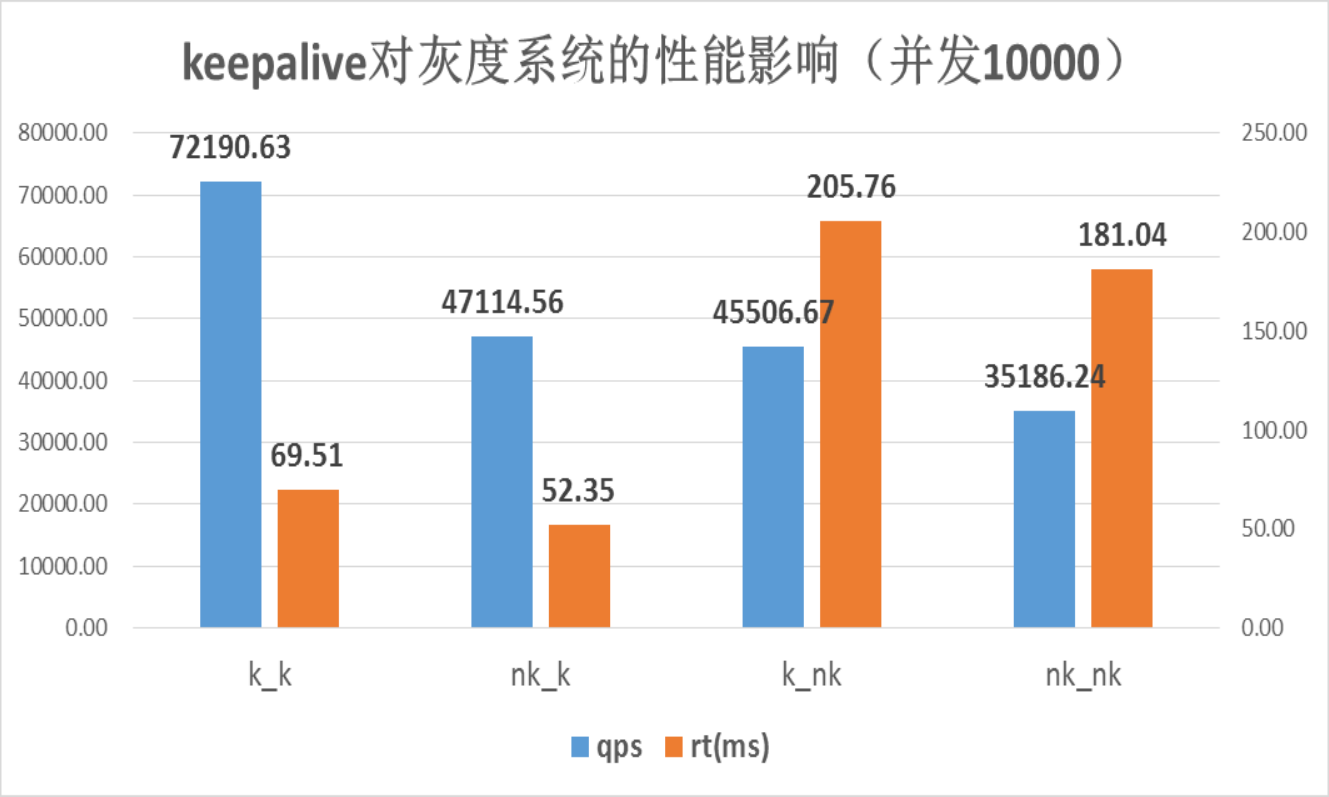
系统的性能

- 2、长连接keepalive的性能影响
  - Keepalive设置对nginx的影响
  - Client 与 proxy server的配置
    - keepalive_requests 100
    - keepalive_timeout 75s
  - Proxy与upstream server配置
    - keepalive 1000

		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

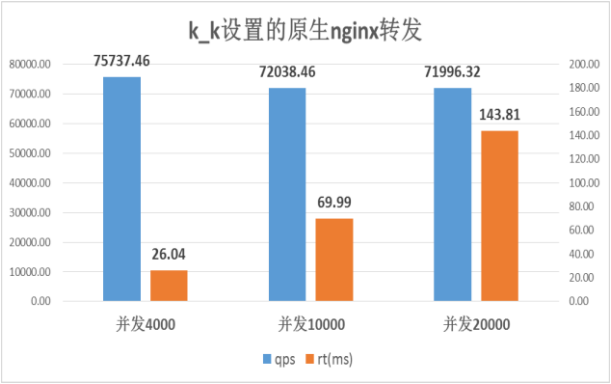
系统的性能

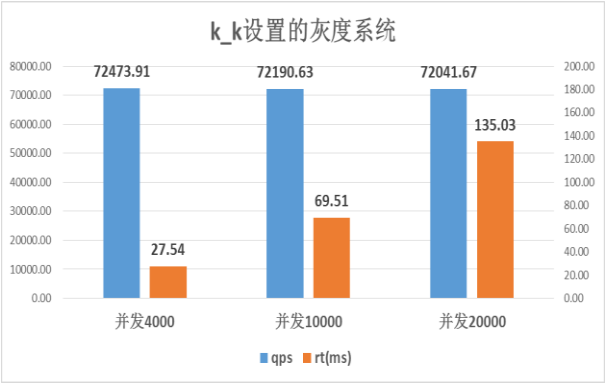
- 2、长连接keepalive的性能影响

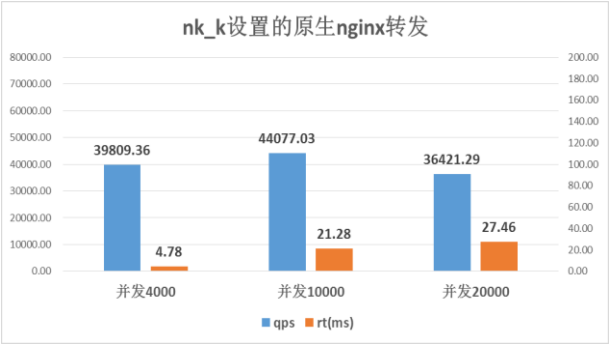


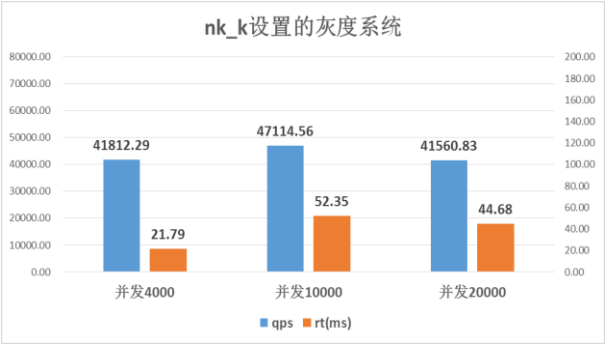
系统的性能

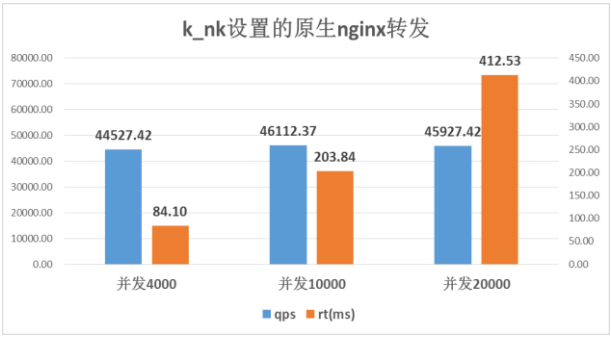
- 2、keepalive的性能影响
  - 原生nginx转发(左列)
  - 灰度系统(右列)
  - 性能差距不大

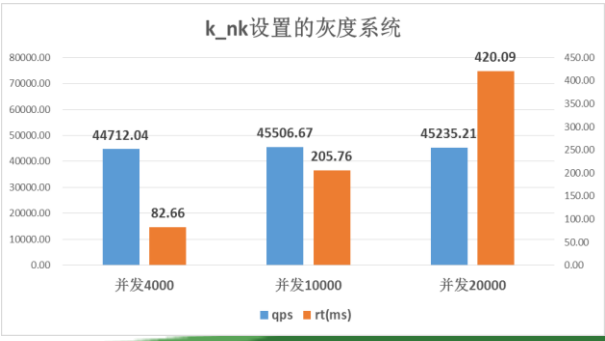












系统的性能

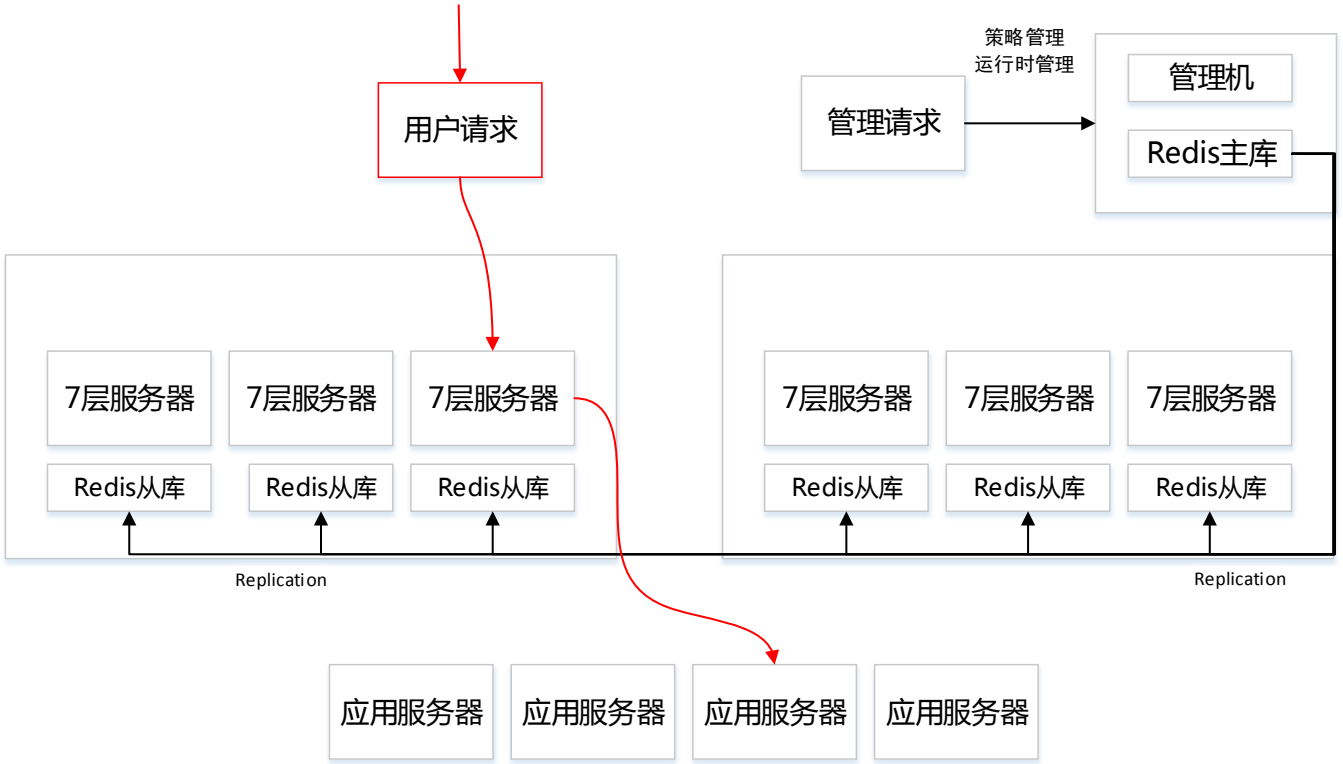
- 压测结果

	qps	rt(ms)	备注
原生nginx转发	73.8k	27.67	
灰度系统（理想情况）	72.4k	27.54	启用cache， 长连接设置
灰度系统（最差情况）	35.2k	181.04	启用cache 无长连接

灰度系统特性

1. 动态设置分流策略，即时生效，无需重启
2. 支持多种分流方式，目前支持iprange、uidrange、uid尾数、指定uid分流、用户请求arg分流等，并且可以灵活添加新的分流方式
3. 高性能，压测数据接近原生nginx转发
4. 灰度系统配置写在nginx配置文件中，运维友好
5. 适用于多种场景：灰度发布、负载均衡、AB测试和API路由等

系统的部署



下一步工作

- 开发和引入适用于自身项目的ngx_lua模块
  - ngx-shared-tree 实现区间查找（已完成）
  - ngx-shared-redis
- 服务监控和报警机制

谢谢大家！