2016年第01期

OpenResty Notes

编者简介

覃冠日，北京邮电大学 2013 级硕士，计算机科学与技术专业，OpenResty 粉丝及使用者，目前主要从事高性能服务器的研发。

版权申明

联系方式：403709339@qq.com

卷首语

过去的 2015 年，OpenResty 发生了很多大事：首届 OpenResty 大会成功在北京举办，成立了专门的委员会，锤子手机向 OpenResty 软件基金会捐献奖金…这一件件振奋人心的消息背后，都有一群默默无闻为 OpenResty 技术的推广而努力的人们。作为一位 OpenResty 的粉丝及使用者，自己感谢 OpenResty 技术的贡献者为开源世界提供 OpenResty 这么优秀的作品，同时也琢磨着自己能为 OpenResty 的推广做些什么。因此，这个电子文档诞生了。希望这个电子文档能给 OpenResty 爱好者带来更多的乐趣。

这一期的《OpenResty Notes》是该系列的第一期，主要收录网络上与 OpenResty 相关的优秀文章，计划每月发布一期。目前该系列主要由四个部分构成：要闻、专栏、特写、架构。要闻主要收录最近的重要新闻事件，专栏主要收录优秀的技术博客，特写主要收录某个话题的分享，系统架构主要收录服务端开发、系统架构有关的精选文章。该系列适用于 Ipad 平板上阅读，希望它能成为大家在地铁、公交上细细品味的精神食粮。

最后，祝 OpenResty 的爱好者们新年快乐！

覃冠日

2016.1.1

目录

罗永浩正式发布锤子 T2,奖金将授予 OpenResty

文章来自：http://news.yesky.com/prnews/321/99622821.shtml

2015 年 12 月 29 日，罗永浩在北京正式发布了 Smartisan T2，前几天锤子 T2 代工厂中天信宣布破产的事件，以及发布会前 Smartisan T2 外观和硬件配置的再次曝光，虽然没有阻止这款年度封箱之作的发布，却使得罗永浩蒙上了一层单刀赴会式的时代苍凉感。不过，罗永浩和他的锤子向来不缺乏这种英雄主义的创业情怀。

Smartisan T2 其实早就已经曝光，不管是在外观还是硬件配置上都被扒的一清二楚，有人称这是一次情怀主义以及理想主义者的围炉取暖，不论如何，这几次戏剧性的新闻反转都使得行业内外的人士广泛聚焦于此次发布会。

19 点 30 分发布会正式开始，罗永浩上场简单回顾了去年的发布会，直接开始介绍 T2 的功能以及设计上的更新，称“也许是中国第一部全金属中框无断点智能手机”，接着罗永浩再次回顾了创业的初衷与最新创业感悟，直言“我请你们忘记左边的牌子(Apple)，也忘记右边的牌子(锤子)，好好看一下工业设计，你可以嘲笑我幻觉，如果他们活着(乔布斯等)，我相信他们不会喜欢左边的这些手机”。

Smartisan T2 软件上的更新十分有亮点，如锤子手机原创性的做了一个被称为远程协助的功能，使得两部手机之间在获得允许的情况下可以相互操作，旨在使老年人无障碍使用锤子手机。这项功能引发了现场一阵阵的欢呼声。

当罗永浩介绍了售价和发售时间后，会议已经接近尾声，一如之前锤子手机的发布会门票收入将用于公益，本次发布会的所有门票收入都将用于开源项目 OpenResty， 这是一个由中国人章亦春发起的全功能的 Web 平台，它打包了标准的 Nginx 核心，很多高质量的 Lua 库，很多高质量的第三方模块，以及它们的大多数依赖项。360，UPYUN，阿里云，新浪，腾讯网，去哪儿网，酷狗音乐等都是其深度用户。

UPYUN 是国内新一代云 CDN 服务商。记者在采访其首席架构师时，他表示：“UPYUN 是 OpenResty 的深度用户，我们在 CDN，API 等方面都有大规模应用 OpenResty 这个开源项目。”

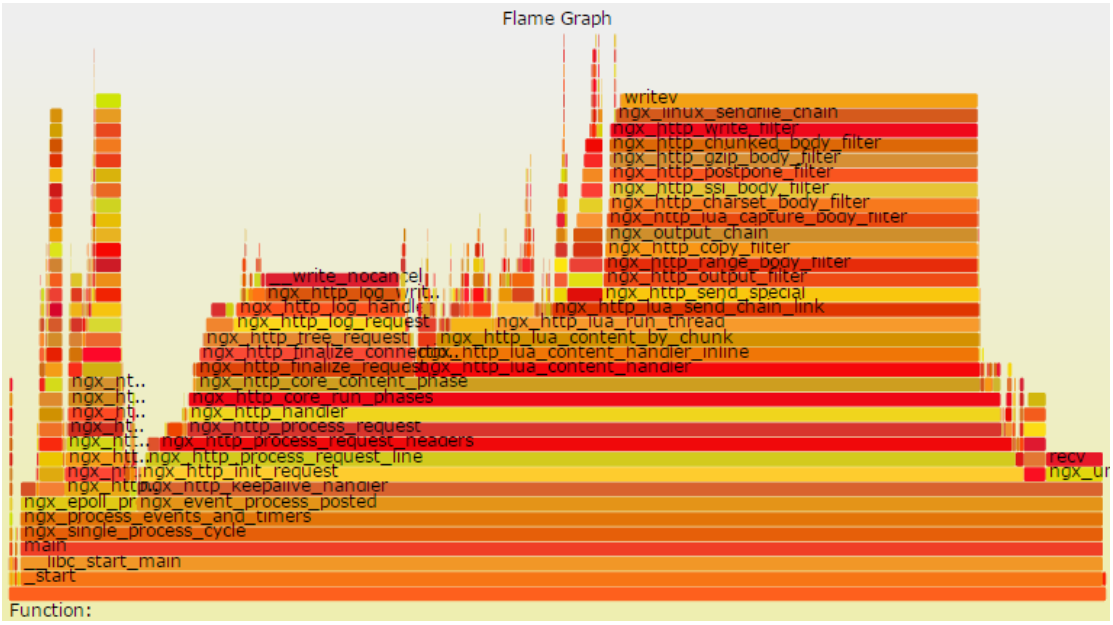
罗永浩的 Smartisan T2 发布会充满了锤粉的狂热和喧嚣，新浪微博上不少用户发状态表示离开发布会现场的地铁上人手一只锤子手机。当情怀的外衣淡去后，Smartisan T2 是否能经受得住市场的考验，令大家拭目以待。

火焰图

文章来自：《OpenResty 最佳实践》

火焰图是定位疑难杂症的神器，比如 CPU 占用高、内存泄漏等问题。特别是 Lua 级别的火焰图，可以定位到函数和代码级别。

下图来自 openresty 的官网，显示的是一个正常运行的 openresty 应用的火焰图，先不用了解细节，有一个直观的了解。



里面的颜色是随机选取的，并没有特殊含义。火焰图的数据来源，是通过 systemtap 定期收集。

什么时候使用

一般来说，当发现 CPU 的占用率和实际业务应该出现的占用率不相符，或者对 nginx worker 的资源使用率（CPU，内存，磁盘 IO）出现怀疑的情况下，都可以使用火焰图进行抓取。另外，对 CPU 占用率低、吐吞量低的情况也可以使用火焰图的方式排查程序中是否有阻塞调用导致整个架构的吞吐量低下。

关于 Github 上提供的由 perl 脚本完成的栈抓取的程序是一个傻瓜化的 stap 脚本，如果有需要可以自行使用 stap 进行栈的抓取并生成火焰图，各位看官可以自行尝试。

如何安装火焰图生成工具

安装 SystemTap

环境 CentOS 6.5 2.6.32-504.23.4.el6.x86_64

SystemTap 是一个诊断 Linux 系统性能或功能问题的开源软件，为了诊断系统问题或性能，开发者或调试人员只需要写一些脚本，然后通过 SystemTap 提供的命令行接口就可以对正在运行的内核进行诊断调试。

首先需要安装内核开发包和调试包（这一步非常重要并且最为繁琐）：

```
# Installaion:

# rpm -ivh kernel-debuginfo-($version).rpm

# rpm -ivh kernel-debuginfo-common-($version).rpm

# rpm -ivh kernel-devel-($version).rpm
```

其中$version 使用 linux 命令 uname -r 查看，需要保证内核版本和上述开发包版本一致才能使用 systemtap。(下载)

安装 systemtap：

```
# yum install systemtap

# ...

# 测试 systemtap 安装成功否：

# stap -v -e 'probe vfs.read {printf("read performed\n"); exit()}'

Pass 1: parsed user script and 103 library script(s) using 201628virt/29508res/3144shr/26860data kb, in 10usr/190sys/219real ms.

Pass 2: analyzed script: 1 probe(s), 1 function(s), 3 embed(s), 0 global(s) using 296120virt/124876res/4120shr/121352data kb, in 660usr/1020sys/1889real ms.

Pass 3: translated to C into "/tmp/stapffFP7E/stap_82c0f95e47d351a956e1587c4dd4cee1_1459_src.c" using 296120virt/125204res/4448shr/121352data kb, in 10usr/50sys/56real ms.

Pass 4: compiled C into "stap_82c0f95e47d351a956e1587c4dd4cee1_1459.ko" in 620usr/620sys/1379real ms.
```

```
Pass 5: starting run.

read performed

Pass 5: run completed in 20usr/30sys/354real ms.
```

如果出现如上输出表示安装成功。

火焰图绘制

首先，需要下载 ngx 工具包：Github 地址，该工具包即是用 perl 生成 stap 探测脚本并运行的脚本，如果是要抓 Lua 级别的情况，请使用工具 ngx-sample-lua-bt

```
# ps -ef | grep nginx （ps：得到类似这样的输出，其中 15010 即使 worker 进程的 pid，后面需要用到）

hippo 14857 1 0 Jul01 ? 00:00:00 nginx: master process /opt/openresty/nginx/sbin/nginx -p /home/hippo/skylar_server_code/nginx/main_server/ -c conf/nginx.conf

hippo 15010 14857 0 Jul01 ? 00:00:12 nginx: worker process

# ./ngx-sample-lua-bt -p 15010 --luajit20 -t 5 > tmp.bt （-p 是要抓的进程的 pid --luajit20|--luajit51 是 LuaJIT 的版本 -t 是探测的时间，单位是秒， 探测结果输出到 tmp.bt）

# ./fix-lua-bt tmp.bt > flame.bt (处理 ngx-sample-lua-bt 的输出，使其可读性更佳)
```

其次，下载 Flame-Graphic 生成包：Github 地址,该工具包中包含多个火焰图生成工具，其中，stackcollapse-stap.pl 才是为 SystemTap 抓取的栈信息的生成工具

```
# stackcollapse-stap.pl flame.bt > flame.cbt

# flamegraph.pl flame.cbt > flame.svg
```

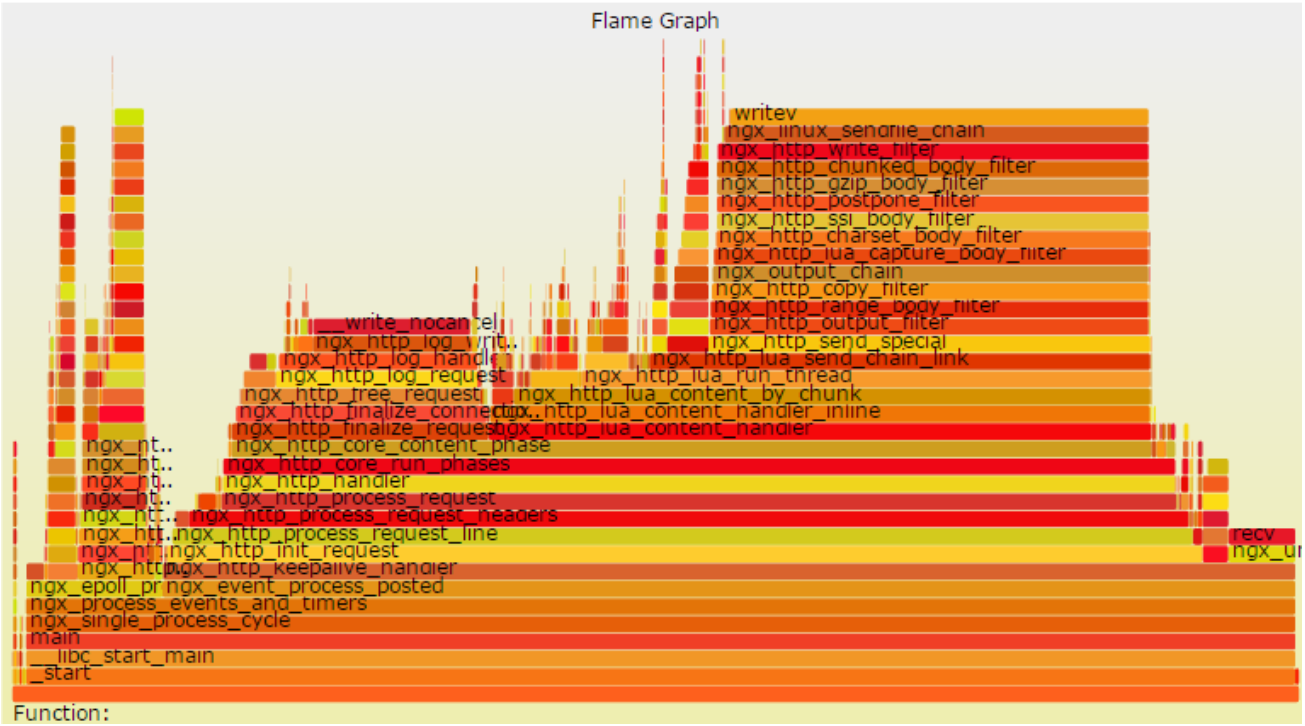
如果一切正常，那么会生成 flame.svg，这便是火焰图，用浏览器打开即可。

问题回顾

在整个安装部署过程中，遇到的最大问题便是内核开发包和调试信息包的安装，找不到和内核版本对应的，好不容易找到了又不能下载，@！￥#@……%@#，于是升级了内核，在后面的过程便没遇到什么问题。 ps：如果在执行 ngx-sample-lua-bt 的时间周期内（上面的命令是 5 秒）,抓取的 worker 没有任何业务在跑，那么生成的火焰图便没有业务内容，不要惊讶哦~

如何定位问题

一个正常的火焰图，应该呈现出如官网给出的样例（官网的火焰图是抓 C 级别函数）：



从上图可以看出，正常业务下的火焰图形状类似的“山脉”，“山脉”的“海拔”表示 worker 中业务函数的调用深度，“山脉”的“长度”表示 worker 中业务函数占用 cpu 的比例。

下面将用一个实际应用中遇到问题抽象出来的示例（CPU 占用过高）来说明如何通过火焰图定位问题。

问题表现，nginx worker 运行一段时间后出现 CPU 占用 100%的情况，reload 后一段时间后复现，当出现 CPU 占用率高情况的时候是某个 worker 占用率高。

问题分析，单 worker cpu 高的情况一定是某个 input 中包含的信息不能被 Lua 函数以正确地方式处理导致的，因此上火焰图找出具体的函数，抓取的过程需要抓取 C 级别的函数和 Lua 级别的函数，抓取相同的时间，两张图一起分析才能得到准确的结果。

抓取步骤：

安装 SystemTap;

获取 CPU 异常的 worker 的进程 ID；

```
ps -ef | grep nginx
```

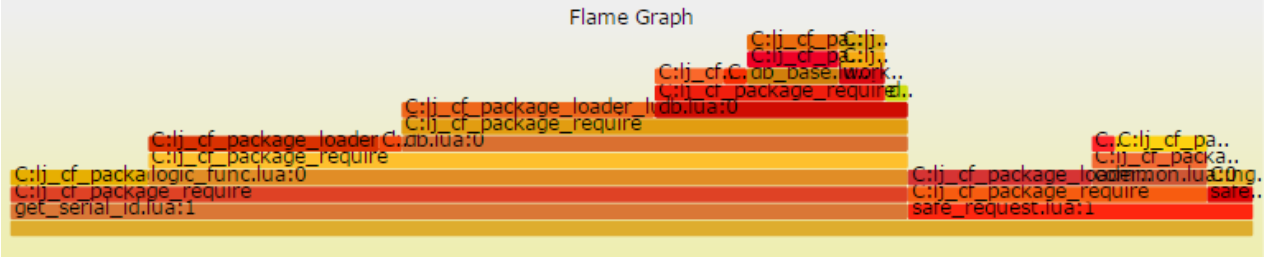
使用 ngx-sample-lua-bt 抓取栈信息,并用 fix-lua-bt 工具处理；

```
./ngx-sample-lua-bt -p 9768 --luajit20 -t 5 > tmp.bt

./fix-lua-bt tmp.bt > a.bt
```

使用 stackcollapse-stap.pl 和；

```
./stackcollapse-stap.pl a.bt > a.cbt

./flamegraph.pl a.cbt > a.svg
```

a.svg 即是火焰图，拖入浏览器即可：



从上图可以清楚的看到 get_serial_id 这个函数占用了绝大部分的 CPU 比例，问题的排查可以从这里入手，找到其调用栈中异常的函数。

ps：一般来说一个正常的火焰图看起来像一座座连绵起伏的“山峰”，而一个异常的火焰图看起来像一座“平顶山”。

OpenResty 反向代理的用法与技巧

文章来自：http://segmentfault.com/a/1190000004128807

导语：

Nginx 最开始是作为反向代理被熟知的，基于它的 OpenResty 的自然也是支持反向代理的，下面我们就来看看它的一些基本用法以及在使用过程中的一些技巧。

一、基本用法

在业务环境中，可能会将 OpenResty（以后简称 OR）作为反向代理，根据不同的 location 定位到不同的后端，在这样的架构下，对应的反向代理配置可能是这样的：

```
location /upstream_A {
    proxy_pass http://192.168.1.100:8080;
    proxy_connect_timeout 2s;
    ...
}
location /upstream_B {
    proxy_pass http://192.168.1.110:8080;
    proxy_connect_timeout 2s;
    ...
}
```

可以看到这部分的配置和 Nginx 并没有太大差别，但是这样的配置会有一些问题，比如我们需要切换后端服务器，将 upstream_A 这个 location 的流量打到 192.168.1.120 这个上游地址而不用修改配置或者重启 Nginx 服务，大部分人可能会选择通过域名的方式去定位上游比如用如下配置：

```
location /upstream_A {
    proxy_pass http://domain_for_upstream_A:8080;
    proxy_connect_timeout 2s;
    ...
}
location /upstream_B {
    proxy_pass http://domain_for_upstream_B:8080;
    proxy_connect_timeout 2s;
    ...
}
```

当需要做_上游切换_的时候，通过修改 /etc/host 文件来将域名定向到新的 Ip，但不幸的是，

Ngixn 并不会使用/etc/host 而是使用命令 resolver 来指定 DNS 服务器,那么在 OR 里面有没有一些高阶的用法可以让上游漂移变得简单呢？答案是肯定的，下面就来看看更加方便的用法，以及里面的坑。

二、进阶用法

上面呢我们讲到如何使用 OR（其实就用到了 Nginx 的配置啦）来完成反向代理，但是由于 Ip 或者域名写死，而 Nginx 又不支持 host，所以在做上游平滑迁移的时候不是很方便，所以我们可以通过在上游配置中用 Nginx 变量来代替上游的地址，变可以避免上述问题。

```
location /internet_prxoy {
    internal;
    set_by_lua $query_url 'return ngx.unescape_uri(ngx.var.arg_url);'
            proxy_pass $query_url;
}
location /upstream_A {
    content_by_lua '
        local redis_op    = require "lua.redis_op"
        local upstream_addr = redis_op.get_upstream_from_redis() --
            从redis中获取上游地址
        local url  = 'http://'..  upstream_addr .. '/foo/bar'
        local res = ngx.location.capture('/internet_proxy',
            { args = {url = url}}
          )
          --容错判断
        ngx.print(res.body)
    ';
}
```

这样，当请求访问到/upstream_A 的时候，会在 redis 当中读取到上游服务器的真实地址并通过/internet_prxoy 转发到上游。

看到这里有人可能会问，为什么需要配置一个额外的跳转 location，而不直接在 set_by_lua 阶段访问 redis 并对 Nginx 变量进行赋值?

原因是函数 get_upstream_from_redis 会涉及到 redis 的访问，而 lua-resty-redis 使用了 ngx.socket.tcp 这个函数，这个函数所支持的执行阶段不包括_set_阶段,需要我们需要一次跳转，通过第一次 location 的 content 阶段从 redis 中将数据读取出来，在第二个 location 的 set 阶段利用刚才所读取的数据完成反向代理。

需要注意的是如果第一次 location，（也就是上述代码中的/upstream_A）的流量非常高，那么可以在 redis 的访问函数，也就是上述代码中的 get_upstream_from_redis()函数中用 shared.dict 来做一次缓存，减少对 redis 的访问量也是可行的。

UPYUN 的 Ngx_lua 最佳技术实践

文章来自：http://blog.sina.com.cn/s/blog_94c587f40102vm7k.html

ngx_lua 是一个 NGINX 的第三方扩展模块，它能够将 Lua 代码嵌入到 NGINX 中来执行。

UPYUN 的 CDN 大量使用了 NGINX 作为反向代理服务器，其中绝大部分的业务逻辑已经由 Lua 来驱动了。

关于这个主题，之前在 OSC 源创会 2014 北京站 和 SegmentFault D-Day 2015 南京站 有做过简单分享，Slide 在【阅读原文】中可以看到。不过两次分享都由于个人时间安排上的不足，对 Keynote 后半部分偏实践的内容并没有做过多地展开，未免有些遗憾，因此，本文作为一个补充将尝试以文字的形式来谈谈这块内容。

ngx_lua 和 Openresty

Openresty 是一套基于 NGINX 核心的相对完整的 Web 应用开发框架，包含了 ngx_lua 在内的众多第三方优秀的 NGINX C 模块，同时也集成了一系列常用的 lua-resty-* 类库，例如 redis, mysql 等，特别地，Openresty 依赖的 NGINX 核心和 LuaJIT 版本都是经过非常充分的测试的，也打了不少必要的补丁。

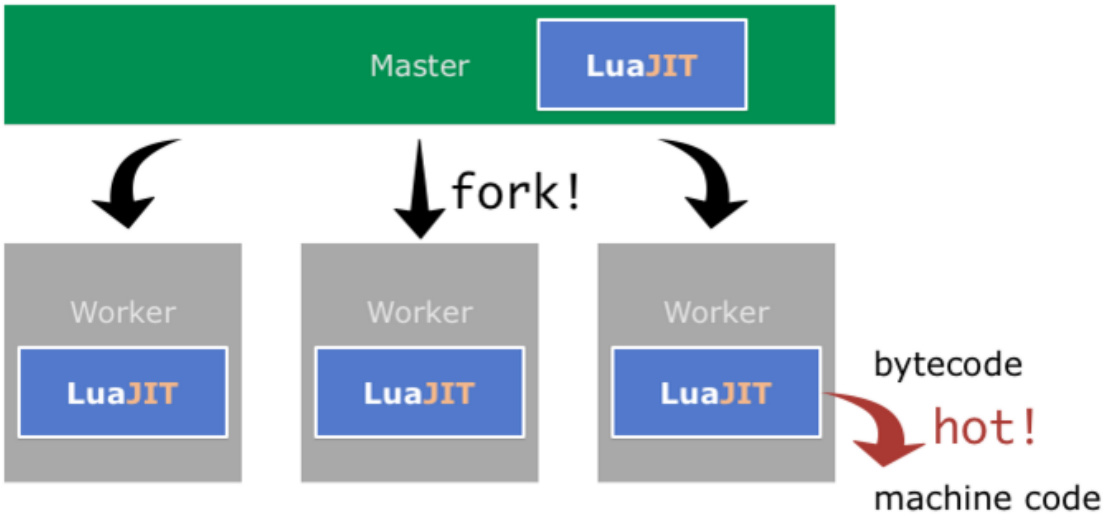
UPYUN CDN 并没有直接基于 Openresty 来开发，而是借鉴了 Openresty 的组织方式，把 ngx_lua 以及我们需要用到的 lua-resty-* 类库直接集成进来自己维护。这样做的原因是因为我们自身也有不少 C 模块存在，同时对 NGINX 核心偶尔也会有一些二次开发的需求，反而直接用 Openresty 会觉得有点不方便。除此之外，需要 ngx_lua 的地方，还是强烈推荐直接用 Openresty。

Lua 的性能

相比 C 模块，Lua 模块在开发效率上有着天然的优势，语言表达能力也更强些，我们目前除了一些业务无关的基础模块首选用 C 来实现外，其它能用 Lua 的基本上都用 Lua 了。这里大家可能比较关心的是脚本语言性能问题，关于这一点，从我们的实践来看，其实不必过于担心的，我们几个比较大的业务模块例如防盗链等用 Lua 重写后，在线下压测和线上运行过程中，均没有发现任何明显的性能衰退迹象。当然，这里很大一部分功劳要归于 LuaJIT，

相比 Lua 官方的 VM，LuaJIT 在性能上有着非常大的提升，另外，还可以利用 LuaJIT FFI 直接调用 C 级别的函数来优化 Lua 代码中可能存在的性能热点。

我们目前线上用的就是 LuaJIT 最新的 2.1 开发版，性能相比稳定版又有不少提升，具体可参考 LuaJIT 这个 NYI 列表。特别地，这里推荐用 Openresty 维护的 Fork 版本，兼容性更加有保障。



如上图所示，LuaJIT 在运行时会将热的 Lua 字节码直接翻译成机器码缓存起来执行。

另外，通过 techempower 网站对 Openresty 的性能评测来看，相比 node.js, cowboy, beego 等，NGINX ngx_lua LuaJIT 这个组合的性能表现还是非常强劲的。

元数据同步与缓存

UPYUN CDN 线上通过 Redis 主从复制的方式由中心节点向外围节点同步用户配置，另外，由于 Redis 本身不支持加密传输，我们还在此基础上利用 stunnel 对传输通道进行了加密，保障数据传输的安全性。

1）缓存是万金油！

当然，不是说节点上有了 Redis 就能直接把它当做主要的缓存层来用了，要知道从 NGINX 到 Redis 获取数据是要消耗一次网络请求的，而这个毫秒级别的网络请求对于外围节点巨大的访问量来说是不可接受的。所以，在这里 Redis 更多地承担着数据存储的角色，而主要的缓存层则是在 NGINX 的共享内存上。

根据业务特点，我们的缓存内容与数据源是不需要严格保持一致的，既能够容忍一定程度的延迟，因此这里简单采用被动更新缓存的策略即可。ngx_lua 提供了一系列共享内存相关的 API (ngx.shared.DICT)，可以很方便地通过设置过期时间来使得缓存被动过期，值得一提的是，当缓存的容量超过预先申请的内存池大小的时候，ngx.shared.DICT.set 方法则会尝试以 LRU 的形式淘汰一部分内容。

以下代码片段给出了一个简陋的实现，当然我们下面会提到这个实现其实存在不少问题，但基本结构大致上是一样的，可以看到下面区分了 4 种状态，分别是：HIT 和 MISS, HIT_NEGATIVE 和 NO_DATA，前两者是对于有数据的情况而言的，后两者则是对于数据不存在的情况而言的，一般来说，对于 NO_DATA 我们会给一个相对更短的过期时间，因为数据不存在这种情况是没有一个固定的边界的，容易把容量撑满。

```lua
local metadata = ngx.shared.metadata
-- local key, bucket = ...
local value = metadata:get(key)
if value ~= nil then
    if value == "404" then
        return    -- HIT_NEGATIVE
    else
        return value -- HIT
    end
end
local rds = redis:new()
local ok, err = rds:connect("127.0.0.1", 6379)
if not ok then
    metadata:set(key, "404", 120) -- expires 2 minutes
    return -- NO_DATA
end
res, err = rds:hget("upyun:" .. bucket, ":something")
if not res or res == ngx.null then
    metadata:set(key, "404", 120)
    return -- NO_DATA
end
metadata:set(key, res, 300) -- expires 5 minutes
rds:set_keepalive()
return res -- MISS
```

2）什么是 Dog-Pile 效应？

在缓存系统中，当缓存过期失效的时候，如果此时正好有大量并发请求进来，那么这些请求将会同时落到后端数据库上，可能造成服务器卡顿甚至宕机。

很明显上面的代码也存在这个问题，当大量请求进来查询同一个 key 的缓存返回 nil 的时候，所有请求就会去连接 Redis，直到其中有一个请求再次将这个 key 的值缓存起来为止，而这两个操作之间是存在时间窗口的，无法确保原子性：

```lua
local value = metadata:get(key)
if value ~= nil then
    -- HIT or HIT_NEGATIVE
end
-- Fetch from Redis
```

避免 Dog-Pile 效应一种常用的方法是采用主动更新缓存的策略，用一个定时任务主动去更新需要变更的缓存值，这样就不会出现某个缓存过期的情况了，数据会永远存在，不过，这个不适合我们这里的场景；另一种常用的方法就是加锁了，每次只允许一个请求去更新缓存，其它请求在更新完之前都会等待在锁上，这样一来就确保了查询和更新缓存这两个操作的原子性，没有时间窗口也就不会产生该效应了。

lua-resty-lock – 基于共享内存的非阻塞锁实现

首先，我们先来消除下大家对锁的抗拒，事实上这把共享内存锁非常轻量。第一，它是非阻塞的，也就是说锁的等待并不会导致 NGINX Worker 进程阻塞；第二，由于锁的实现是基于共享内存的，且创建时总会设置一个过期时间，因此这里不用担心会发生死锁，哪怕是持有这把锁的 NGINX Worker Crash 了。

那么，接下来我们只要利用这把锁按如下步骤来更新缓存即可：

1、检查某个 Key 的缓存是否命中，如果 MISS，则进入步骤 2。

2、初始化 resty.lock 对象，调用 lock 方法将对应的 Key 锁住，检查第一个返回值（即等待锁的时间），如果返回 nil，按相应错误处理；反之则进入步骤 3。

3、再次检查这个 Key 的缓存是否命中，如果依然 MISS，则进入步骤 4；反之，则通过调用 unlock 方法释放掉这把锁。

4、通过数据源（这里特是 Redis）查询数据，把查询到的结果缓存起来，最后通过调用 unlock 方法释放当前 Hold 住的这把锁。

具体代码实现请参考：https://github.com/openresty/lua-resty-lock#for-cache-locks

当数据源故障的时候怎么办？NO_DATA？

同样，我们以上面的代码片段为例，当 Redis 返回出现 err 的时候，此时的状态即不是 MISS 也不是 NO_DATA，而这里统一把它归类到 NO_DATA 了，这就可能会引发一个严重的问题，假设线上这么一台 Redis 挂了，此时，所有更新缓存的操作都会被标记为 NO_DATA 状态，原本旧的拷贝可能还能用的，只是可能不是最新的罢了，而现在却都变成空数据缓存起来了。

那么如果我们能在这种情况下让缓存不过期是不是就能解决问题了？答案是 yes。

lua-resty-shcache – 基于 ngx.shared.DICT 实现了一个完整的缓存状态机，并提供了适配接口

恩，这个库几乎解决了我们上面提到的所有问题：1. 内置缓存锁实现 2. 故障时使用陈旧的拷贝 - STALE

所以，不想折腾的话，直接用它就是的。另外，它还提供了序列化、反序列化的接口，以 UPYUN 为例，我们的元数据原始格式是 JSON，为了减少内存大小，我们又引入了 MessagePack，所以最终缓存在 NGINX 共享内存上是被 MessagePack 进一步压缩过的二进制字节流。

当然，我们在这基础上还增加了一些东西，例如 shcache 无法区分数据源中数据不存在和数据源连接不上两种状态，因此我们额外新增了一个 NET_ERR 状态来标记连接不上这种情况。

序列化、反序列化太耗时？！

由于 ngx.shared.DICT 只能存放字符串形式的值（Lua 里面字符串和字节流是一回事），所以即使缓存命中，那么在使用前，还是需要将其反序列化为 Lua Table 才行。而无论是 JSON 还是 MessagePack，序列化、反序列操作都是需要消耗一些 CPU 的。

如果你的业务场景无法忍受这种程度的消耗，那么不妨可以尝试下这个库：https://github.com/openresty/lua-resty-lrucache 。它直接基于 LuaJIT FFI 实现，能直接将 Lua Table 缓存起来，这样就不需要额外的序列化反序列化过程了。当然，我们目前还没尝试这么做，如果要做的话，建议在 shcache 共享内存缓存层之上再加一层 lrucache，也就是多一级缓存层出来，且这层缓存层是 Worker 独立的，当然缓存过期时间也应该设置得更短些。

节点健康检查

被动健康检查与主动健康检查

我们先来看下 NGINX 基本的被动健康检查机制：

```
upstream api.com {
    server 127.0.0.1:12354 max_fails=15 fail_timeout=30s;
    server 127.0.0.1:12355 max_fails=15 fail_timeout=30s;
    server 127.0.0.1:12356 max_fails=15 fail_timeout=30s;
    proxy_next_upstream error timeout http_500;
    proxy_next_upstream_tries 2;
}
```

主要由 max_failes 和 fail_timeout 两个配置项来控制，表示在 fail_timeout 时间内如果该 server 异常次数累计达到 max_failes 次，那么在下一个 fail_timeout 时间内，我们就认为这台 server 宕机了，即在这段时间内不会再将请求转发给它。

其中判断某次转发到后端的请求是否异常是由 proxy_next_upstream 这个指令决定的，默认只有 error 和 timeout，这里新增了 http_500 这种情况，即当后端响应 500 的时候我们也认为异常。

proxy_next_upstream_tries 是 NGINX 1.7.5 版本后才引入的指令，可以允许自定义重试次数，原本默认重试次数等于 upstream 内配置的 server 个数（当然标记为 down 的除外）。

但只有被动健康检查的话，我们始终无法回避一个问题，即我们始终要将真实的线上请求转发到可能已经宕机的后端去，否则我们就无法及时感知到这台宕机的机器当前是不是已经恢复了。当然，NGINX PLUS 商业版是有主动监控检查功能的，它通过 health_check 这个指令来实现，当然我们这里就不展开说了，说多了都是泪。另外 Taobao 开源的 Tengine 也支持这个特性，建议大家也可以尝试下。

lua-resty-checkups - 纯 Lua 实现的节点健康检查模块

这个模块目前是根据我们自身业务特点高度定制化的，因此暂时就没有开源出来了。agentzh 维护的 lua-resty-upstream-healthcheck 模块跟我们这个很像但很多地方使用习惯都不太一样，当然，如果当初就有这样一个模块的话，说不定就不会重造轮子了 :-)

```lua
-- app/etc/config.lua
_M.global = {
    checkup_timer_interval = 5,
    checkup_timer_overtime = 60,
}
_M.api = {
    timeout = 2,
    typ = "general", -- http, redis, mysql etc.
    cluster = {
        {   -- level 1
            try = 2,
            servers = {
                { host = "127.0.0.1", port = 12354 },
                { host = "127.0.0.1", port = 12355 },
                { host = "127.0.0.1", port = 12356 },
            }
        },
        {   -- level 2
            servers = {
                { host = "127.0.0.1", port = 12360 },
                { host = "127.0.0.1", port = 12361 },
            }
        },
    },
}
```

上面简单给出了这个模块的一个配置示例，checkups 同时包括了主动和被动健康检查两种机制，我们看到上面 checkup_timer_interval 的配置项，就是用来设置主动健康检查间隔时间的。

特别地，我们会在 NGINX Worker 初始阶段创建一个全局唯一的 timer 定时器，它会根据设置的间隔时间进行轮询，对所监控的后端节点进行心跳检查，如果发现异常就会主动将此节点暂时从可用列表中剔除掉；反之，就会重新加入进来。checkup_timer_overtime 配置项，跟我们使用了共享内存锁有关，它用来确保即使 timer 所在的 Worker 由于某种异常 Crash 了，其它 Worker 也能在这个时间过期后新起一个新的 timer，当然存活的 timer 会始终去更新这个共享内存锁的状态。

其它被动健康检查方面，跟 NGINX 核心提供的机制差不多，我们也是仿照他们设计的，唯一区别比较大的是，我们提供了多级 server 的配置策略，例如上面就配置了两个 server 层级，默认始终使用 level 1，当且仅当 level 1 的节点全部宕机的时候，此时才会切换使用 level 2，特别地，每层 level 多个节点默认都是轮询的，当然我们也提供配置项可以特殊设置为一致性哈希的均衡策略。这样一来，同时满足了负载均衡和主备切换两种模式。

另外，基于 lua-upstream-nginx-module 模块，checkups 还能直接访问 nginx.conf 中的 upstream 配置，也可以修改某个 server 的状态，这样主动健康检查就能使用在 NGINX 核心的 upstream 模块上了。

其它

当然，ngx_lua 在 UPYUN 还有很多方面的应用，例如流式上传、跨多个 NGINX 实例的访问速率控制等，这里就不一一展开了，这次 Keynote 中也没有提到，以后有机会我们再谈谈。

基于 OpenResty 的 Web 服务框架 Vanilla 实战

文章来自：http://www.tuicool.com/articles/vM7NNva

周晶

大家好，很高兴能在这里跟大家探讨 OpenResty Web 服务开发的话题，感谢大家的参与，也感谢同学的辛苦组织，我今晚的分享主题是《基于的 Web 服务框架 Vanilla 实战》，在正式开始之 前，我先简单介绍下我和我的团队。

我叫周晶，来自新浪移动事业部系统架构组，我们团队主要负责移动这边所有的后端服务架构和系统调优的工作，战时为线上服务保驾护航，和平时期则进行技术调研、预言、选型、改造升级等。我们对一 切新事物感兴趣，喜欢技术热爱分享，欢迎大家有空过来坐坐：）

接下来我们正式开始

下面是今天分享的提纲：

1. 新浪移动的 OpenResty 技术选型
2. 如何从零开始做 OpenResty 开发
3. Vanilla/ 香草是什么 ？
4. 为什么需要 Vanilla 部署
5. 使用

新浪移动的 OpenResty 技术选型

- Apache 时代（PHP 处理大量 CPU 密集型计算）
- Nginx 时代（数据来源大多走 HTTP 接口）
- OpenResty 时代（H5 的时代，大量异步接口请求）

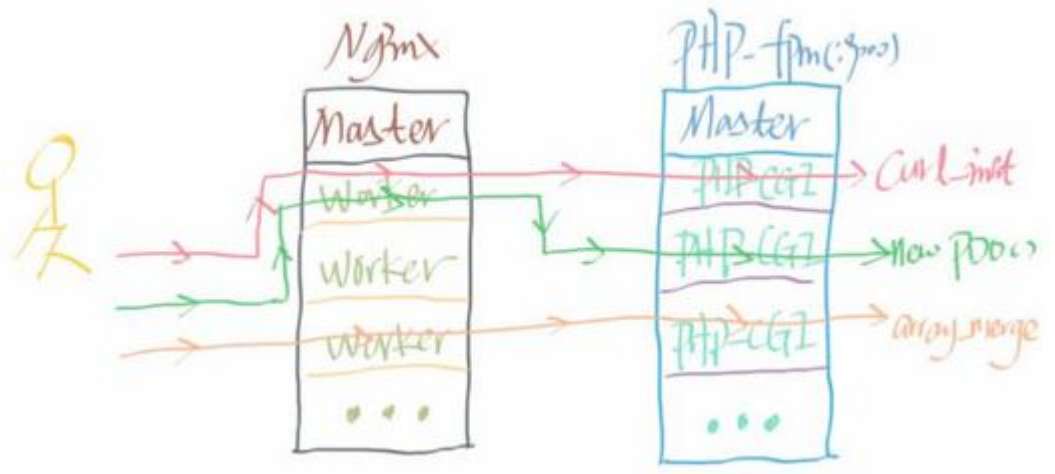
新浪无线时期的手机端业务比较幸富，各种 PC 业务部在出手机版，而很多内容部直接抓取的 PC 页面， 这个时期业务中充斥着很多正则匹配、数据组装、拼台等密集型 CPU 计算，并发请求不大，单机并发在以下，服务很 OK。

Apache 是同步多进程模型，一个连接对应一个进程，并发低的情况下，很稳定。

随着业务线的爆增,Apache 时代并没有持续太久，虽然整体架构勉强满足需求，但是流量暴涨的时候就吃不消，比如某个缓存实例宕机，导致进程处理缓慢，拥堵，连接数吃紧随即宕机，而且一宕一片。

14 年月份马航失联、文章出轨前后一个周我们十几亿的 hits 宕了两次机。因为流量创新高我们很开心，但是面对这样突发的事件、突高的流量如何应对？现有的架构肯定不能满足，我们不得不思考新的解决方案。

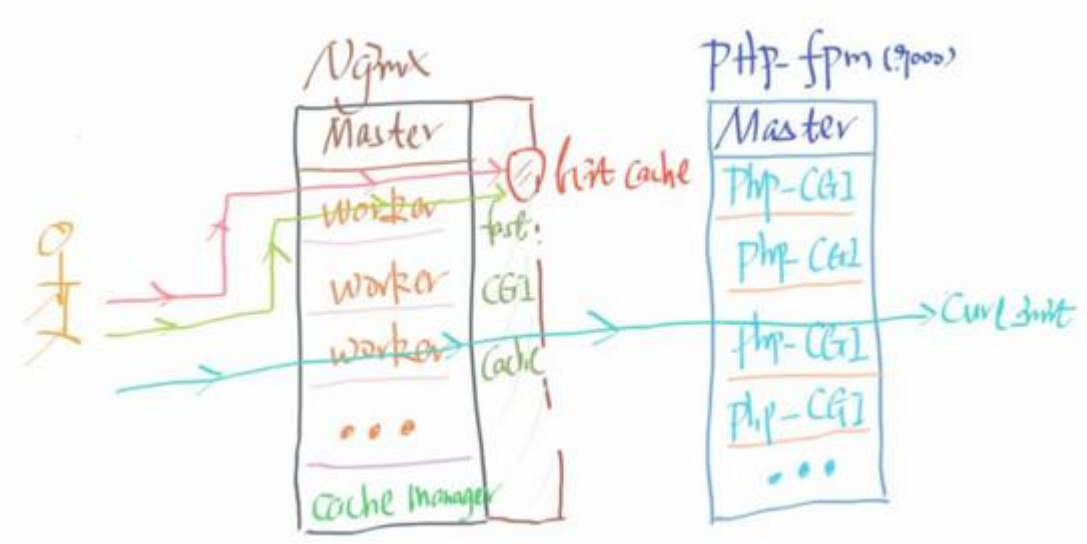
我们重新梳理了业务、资源配置、服务流量分布等信息，本着数据资源离用户最近、服务切换机动性更 高的原则，我们细化了流量分布到各省运营商，调整了数据资源配备，关键一点我们硬着头皮把所有的 Apache 换成了 。



在现实资源紧俏、诸多业务性能亟需优化、业务不断疯涨的前提下，最短平快的解决方案就是在架构层面统一处理，我们看中 Nginx 的异步 I/O 多路复用，而且 Nginx 的 Fast_Cgi_Cache 压测 QPS 提升至少一个 数量级。

我们用了几个月时间将所有的服务部迁移到 Nginx，并在核心业务上添加了 Fast_Cgi_Cache，几百个 Vhost 配置需要从 Apache 转换到 ，绝对是一个大工程，当时负责运维同学的比较忙，基本部是架构组来主导这个事情。

好在我们趟过了一道道坎，14 年的世界杯，我们平稳的扛住了近 20 亿的 hits，相比 14 年初的马航、文章 时期的两次连续君机，我们成长了很多，尤其是在 Nginx 的使用方面。



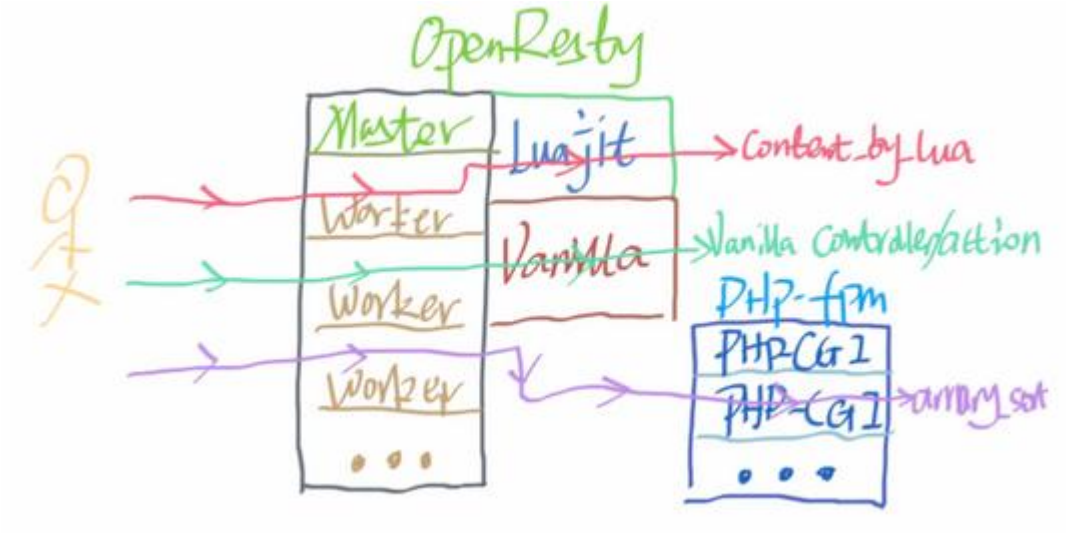
新浪移动有多年的 PHP 积累，但是想要追求更高性能和并发，PHP 显得有些力不从心。为此我们添加了 Fast_Cgi 缓存，但这种文件缓存存在本机，更新、清理等操作不便捷且数据一致性也难以保证。

而发展迅速的新浪移动在业务线方面也慢慢有更多自己特有的特色业务，需求更精细化，数据一致性也 提出更高的要求。如何满足？

我们尝试过 Nodejs 、 等方案，Node 基于事件编程，PHP 程序员转变起来有一定曲折，而 Go 的开发调试相对复杂，对我们业务运营需求快速响应是个问题，于是我们把目光投向了 OpenResty。

移动端的页面结构相对简单、小巧，业务逻辑相对简单，计算量小，反倒是对服务吞吐量、性能有更高 要求。OpenResty 核心是 Nginx，非阻塞的异步 I/O 天然支持高并发，基于 Lua 实现移动端相对简单的业务逻辑又非常轻松。

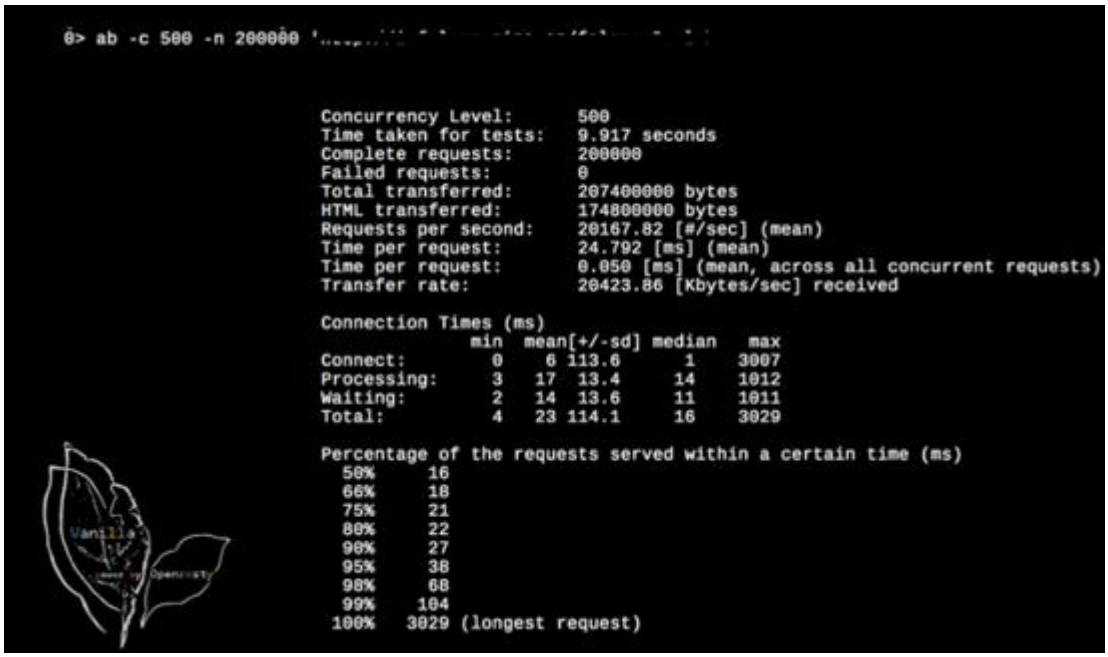
我们可以把我们的业务逻辑部分甚至全部迁移到 Lua，来解决之前文件缓存的问题，保证性能的同时又保证了数据一致性。一系列的测试验证了我们的观点，新浪移动从此走上了 OpenResty 之路。

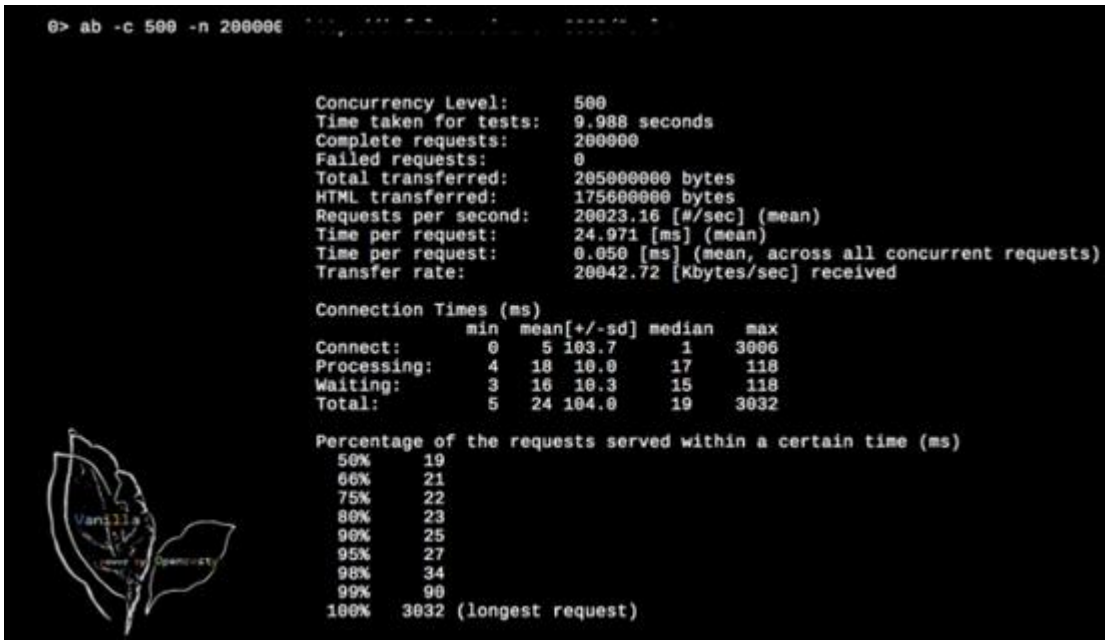


上图也从侧面大致反映了 OpenResty 在新浪移动服务的情况。

当前我们线上全部 Nginx 服务已经全部迁移至 OpenResty，部分核心接口已经使用 Lua 改写，web 服务正在向 Vanilla 迁移过程中， 目前 Lua 提供了过亿的 hits 的服务。

下面是迁移时我们核心接口使用框架前后在测试环境压测的数据对比图，可以看到使用框架并没有带来什么性能损耗：





如何从零开始做 OpenResty 开发

在探讨如何从零开始 OpenResty 开发之前，我们先来重温下什么是 OpenResty，先看一张 OpenResty 官网的截图。看看 OpenResty 作者章亦春(agentzh)对它的定义。

Openresty = Nginx + ngx http_lua_module + lua_resty *； 台上一个 HTTP_LUA 模块，再加上一系列 Lua_resty 模块组成的一个 Ngx_Lua 高性能服务生态。

在没有 OpenResty 的时候，基于 Nginx 开发高性能后端服务这是一件高大上的事情，没有很深的 C 语言开发功底，是做不了的。

要是想做好那更是得对 Nginx 整体架构有深入细致的了解，Nginx 源码里那些三星级（***）四星级（****）的 C 指针让多少英雄好汉望而却步。而且 C 模块的开发调试真是不简单。

下图是 Python 社区大名鼎鼎的“大妈总结的 Lua 开发和 C 开发 Nginx 模块的流程，多么走心的领悟。

应该已经体验到 OpenResty 的核心爽直了?!

具体的:

- 不用跟 c 死磕,想给 Nginx 扩展功能,不用以往麻烦的调试过程循环

```
改c代码
^  `->编译
|   `->替换老.so
|     `->重启nginx (加载模块,有时,还必须整个重新编译 nginx)
|       `-> curl 请求测试
|                |
+----------------/
```

- 而是,非常直接的:

```
改lua代码
 ^ `-> curl 请求测试
 |       |
 +-------/
```

而且, 性能几乎没有下降!

基于 OpenResty 的服务端开发，实际是在一个 Nginx 环境中扩展的 Lua 引擎运行 Lua 代码。需要至少两方 面的基础知识：Nginx 配置、Lua 语法。

而基于 Vanilla 做 OpenResty 开发更简单到只需要了解最简单的 Lua 语法甚至部可以先不了解 Lua 语法，咱们先跑个 HelloWorld 看看，觉得感兴趣咱们再往下继续学习 Lua 语法。

原生的OpenResty开发调试helloworld:

```
server { #在nginx配置文件中引用Lua代码，然后保存重启服务器
  listen       9110;
  server_name  localhost;
  error_log   logs/error.my_openresty.log info;
  location / {
      content_by_lua "ngx.say('Hello,world!')";
  }
}
```

保存重启服务器看效果

```
// ./openresty.server reload
// curl 'http://localhost:9110'
Hello,world!
```

基于的 helloworld（只需要你会写 return 'helloworld'开发）

```
function IndexController:helloworld()
    return 'helloworld'
end
```

开发环境，lua_code_cache默认off，保存即可看到效果

```
// curl 'http://localhost:9110/index/helloworld'
helloworld
```

其实 Vanilla new app 命令会自动生成一个代码骨架，里面的实例已经足够你 Lua 入门了，接下来只需要很方便的在 helloworld 里面去修改、验证、体会即可。

关于 Vanilla 的使用自们后面细说。这里给大家推荐一些入门的经典资料，作为如何从零开始的答案，欢 迎大家尝试，更欢迎大家与我交流。

openresty 官网：http://openresty.org/（中文版：http://openresty.org/cn/）

Nginx 教程：https://openresty.org/download/agentzh-nginx-tutorials-zhcn.html

lua5.1 文档：http://www.lua.org/manual/5.1/

openresty 最佳实践：https://www.gitbook.com/ book/moonbingbing /openresty-best-practices/ details

Nginx-lua 模块文档：https://github.com/openresty/lua-nginx-module

香草是什么?

上面说到基于 OpenResty 使用 Lua 开发 Nginx 模块应用的方便之处时，没把持住已经展示了 Vanilla 的方便之处，那么 Vanilla 到底是什么呢？在这里正式跟大家做一个介绍。

选型初期，我们使用简单的几个 Lua 文件垒出了一部分高并发的核心接口，虽然基于

OpenResty 的开发相对于 PHP 开发有些许不同，调试报错不像 PHP 那样所见即所得，Lua 语法也需要一段时间适应，但是改造后获得比之前有数量级的性能提升，给了我们极大的鼓舞。一切部是值得的。

但是面对现实，如何使用 Lua 能像 PHP 那样快速高效的开发调试满足业务需求？我们的目标不止是几个简单的文件重写一部分接口，我们希望更多的业务能迁移到 OpenResty，更多的享受随之带来的高性能。

我们希望更多的同学能很轻松的加入其中，共同享受。有次在 OpenResty 技术群里面看到一个同学说：

前辈们：新生刚开始学习 openresty，可是无从下手！看了一些零散的资料，就是不能系统的学一下！ 而且好多配置的变量不知道去那个模块去什么地方去找！官网也是一些散乱的模块介绍……不像之前看 的开源框架似的有一套系统的文档……求学习思路或学习路线……如何起步如何系统的学习这 个框架……谢谢!!!

对这句话我有很大共鸣，我想，OpenResty 这么好的东西，为什么没有被更广泛的用起来呢，对于那些 做 C 模块开发的同学，熟悉 Nginx 的同学来说这是小菜一碟，我是做业务出身，回忆起自己走过的这条 OpenResty 学习之路，也很不易。

我在团队做过很多相关 OpenResty 和 Lua 的分享，想以此让大家更多的了解 Nginx、学习 Lua 进而能用起来，但是收效甚微。

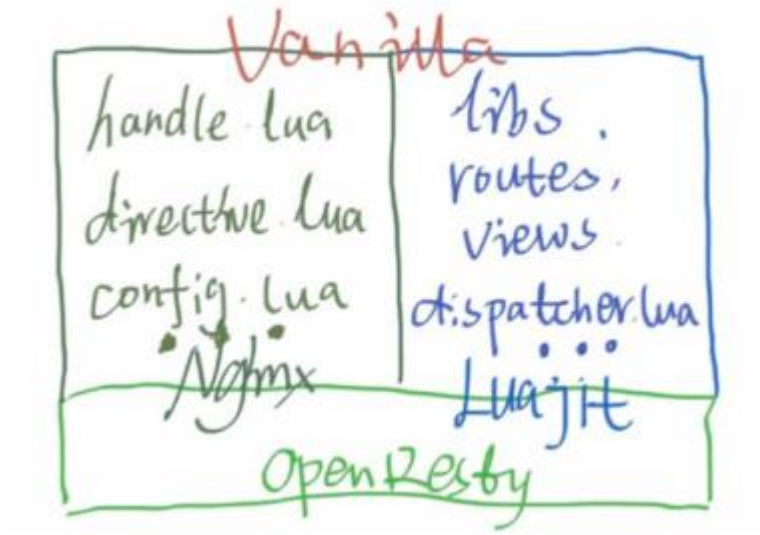
所以我决定为改变这一切，做点什么。

我们结台自身的实际业务场景，分析调研了市面上现有的几个较受欢迎的 Lua 框架，发现离我们的预期 多少有些远，而我们团队自己研发的 PHP 框架 Be 和乌哥的 Yaf 等部是我们同学所熟练使用的，基于此， 我结台这些年自己对 Web 应用开发的理解和众多框架的优点，开发了 Vanilla。

Vanilla 就是为了让大家更好的从零开始做 OpenResty 开发，整台一些优秀的开源库，加速 Web 服务的开发速度，让开发调试更便捷，让大家更舒心的体验 OpenResty 的美。

Nginx 方面 Vanilla 将繁杂的 Nginx 配置和指令集台做了统一封装，使用户不再需要了解 Nginx 相关的配置 和服务的启动管理，而专注于业务开发。采用不同的环境变量加载不同环境的配置。开发、测试、生产 环境相隔离。

方面， 将业务处理整体封装在 Content_by_lua_file 这个 phase，定义了专门的 ErrorHandle 使 用 Pcall 捕获了运行时异常，使开发调试所见即所得，更便捷、高效。如果刚刚入门还不会元表、面向 对象这些高级货，不用担心，你只需要在 Controller 里面写自己的业务逻辑，边写边学习体会 Lua 的内在美。



下面这句是我提炼 Vanilla 的愿景：

存在的意义在于降低基于 OpenResty 开发高性能后端服务的门槛，我们希望能探索一种快乐、便捷的高性能后端服务开发模式。

然，开源永不止步，Vanilla 要走的路还很长，我希望能有更多有想法的同学一起来把 Vanilla 做的更完善、更好用。

部署 Tips

看到上面我描述了那么多 Vanilla 的好，小伙伴们有没有激动的想牛刀小试啊。不着急，自先看下如何拥 有 Vanilla。这里总结几条 Vanilla 部署的 Tips。

1.安装环境，并保证 nginx 命令可直接运行（nginx 可执行文件放置于环境变量 path 下）。

2.安装 Lua 包管理工具 Luarocks 管理 Vanilla 包依赖(Luarocks 依赖的 Lua 开发包指定为 Lua5.1，为何使用 Lua5.1 ？因为 运行时默认使用 Luajit 来优化 Lua 代码的执行效率，而 Luajit 基于 Lua5.1ABI 开发，今后也不会兼容 Lua5.2 和 Lua5.3 ）。

3 .如果使用 luarocks install vanilla 命令无法成功安装 vanilla，请在后面添加—verbose 参数，查看执行过程（luarocks install vanilla —verbose）。

4 .如果你有环境，也可以直接 docker pull zhoujing/vanilla。 关于安装部署有和问题，欢迎与我交流，也可以查看我之前写的两篇文档：

如何配置一套优雅的 Lua 开发环境：http://www.jianshu.com/p/196b5dad4e78

Lua 包管理工具 Luarocks 详解：http://my.oschina.net/idevz/blog/519598

使用

真实前面几个部分已经大致说了 Vanilla 的使用入门，下面具体讲解细节使用，大致按照以下页序进行：

创建应用

请求处理过程

所在的 Phase ： content_by_lua_file ./pub/index.lua; Vanilla 配置

的路由器和路由协议的面件错误处理内建库

成功部署Vanilla后，在命令行运行vanilla命令可见如下信息：

```
 ~/ vanilla
Vanilla v0.1.0-rc2, A MVC web framework for Lua powered by OpenResty.

Usage: vanilla COMMAND [ARGS] [OPTIONS]

The available vanilla commands are:
 new [name]              Create a new Vanilla application
 start                   Starts the Vanilla server
 stop                    Stops the Vanilla server

Options:
 --trace                 Shows additional logs
```

目前 vanilla 命令行支持三种操作：

创建新应用

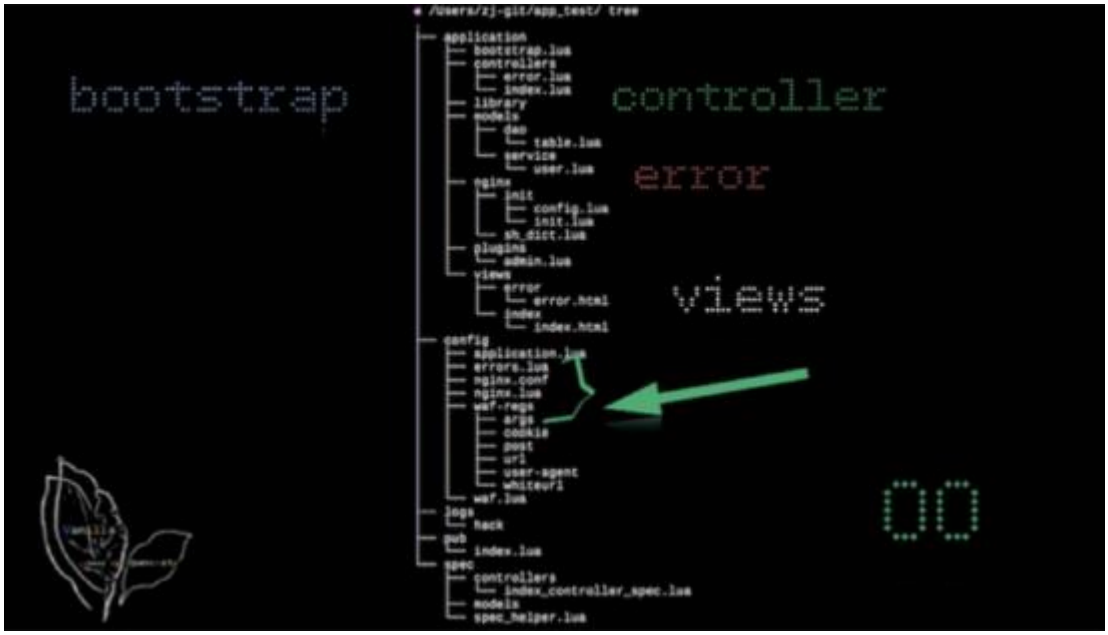
启动应用服务

停止应用服务

创建名为的应用我们只需要运行 vanilla new app_test 即可。启动/停止该服务则需要进入到应用目录 cd app_test 下运行 vanilla start/stop，在 vanilla start/stop 命令后添加—trace 参数，可以显示启动/停止服务所执行的命令。

```
/Users/zj-git/ vanilla new app_test
Creating app app_test...
  created file app_test/logs/hack/.gitkeep
  created file app_test/config/application.lua
  created file app_test/config/waf-regs/url
  created file app_test/application/nginx/init/config.lua
  created file app_test/application/nginx/sh_dict.lua
  created file app_test/spec/models/.gitkeep
  created file app_test/application/nginx/init/init.lua
  created file app_test/spec/controllers/index_controller_spec.lua
  created file app_test/config/waf.lua
  created file app_test/config/waf-regs/whiteurl
  created file app_test/config/nginx.lua
  created file app_test/.gitignore
  created file app_test/application/bootstrap.lua
  created file app_test/config/waf-regs/user-agent
  created file app_test/application/controllers/error.lua
  created file app_test/spec/spec_helper.lua
  created file app_test/config/waf-regs/post
  created file app_test/config/waf-regs/cookie
  created file app_test/application/models/service/user.lua
  created file app_test/application/views/index/index.html
  created file app_test/config/nginx.conf
  created file app_test/application/library/.gitkeep
  created file app_test/config/errors.lua
  created file app_test/application/controllers/index.lua
  created file app_test/config/waf-regs/args
  created file app_test/pub/index.lua
  created file app_test/application/plugins/admin.lua
  created file app_test/application/views/error/error.html
  created file app_test/application/models/dao/table.lua
```

不必为这么多 created file 所下倒,这只是个架子，我们使用 Vanilla 做业务开发，只需要关注 Bootstrap、 Controller、View 和 Error 即可。



进到 new 的应用（自们上面 new 的是 app_test）目录下运行 vanilla start 即可启动服务，默认以开发环境启动，我们只需要到 controller 下面写相应的业务即可。 默认运行在 9110

端口，helloworld 的例子上面已经有了。

上最新版本的 还支持 vanilla-console 命令，运行 vanilla-console 可以进入 Lua 交互命令行，切可以调用相关的 Vanilla 包。目的方便 Lua 学习以及 Vanilla 调试。

目前这个命令还在不断完善中，感兴趣的同学可以先尝试，也欢迎跟我一起完善。

```
 // vanilla-console
Lua 5.1.5  Copyright (C) 1994-2012 Lua.org, PUC-Rio
>
```

目前这个命令还在不断完善中，感兴趣的同学可以先尝试，也欢迎跟我一起完善。

Vanilla 请求处理过程

我们先看服务是如何启动的，服务不起何谈请求。

app_test 这个应用的服务，添加--tracevanilla start 命令

背后实际执行了什么命令以及命令运行输出：

```
 /Users/zj-git/app_test/ vanilla start --trace
nginx  -g "env VA_ENV=development;" -p `pwd`/ -c tmp/development-nginx.conf
nginx: [alert] lua_code_cache is off; this will hurt performance in /Users/zj-
git/app_test/tmp/development-nginx.conf:23
Vanilla app in development was succesfully started on port 9110.
```

命令根据当前运行环境和 nginx 配置文件骨架（./app_test/config/nginx.conf）以当前目录 app_test 为启动目录生成一个对应运行环境的 nginx 配置文件（./app_test/tmp/development-nginx.conf）和 nginx 服务启动命令。默认的环境为 VA_ENV=development。 关键在于这两个 nginx 配置文件。

nginx 配置文件骨架

```
http {
    # use sendfile
    sendfile on;

    # Va initialization
    {{LUA_PACKAGE_PATH}}
    {{LUA_PACKAGE_CPATH}}
    {{LUA_CODE_CACHE}}
    {{LUA_SHARED_DICT}}


    {{INIT_BY_LUA}}
    {{INIT_BY_LUA_FILE}}
    {{INIT_WORKER_BY_LUA}}
    {{INIT_WORKER_BY_LUA_FILE}}

    server {
        server_name v.com;
        include /usr/local/nginx_x/nginx/conf/mime.types;
        # List port
        listen {{PORT}};
        set $template_root '';

        location /static {
            alias pub/static;
        }

        # Access log with buffer, or disable it completetely if unn
        access_log logs/{{VA_ENV}}-access.log combined buffer=16k;
        # access_log off;

        # Error log
        error_log logs/{{VA_ENV}}-error.log debug;
        # error_log logs/{{VA_ENV}}-error.log;

        # Va runtime
        {{CONTENT_BY_LUA_FILE}}

        location ~ .*\.(php|php5)?$
        {
            root /Users/zj-git/yaf/phppub/;
            #fastcgi_pass    unix:/tmp/php-cgi.sock;
            fastcgi_pass    127.0.0.1:9000;
            fastcgi_index index.php;
            include fastcgi.conf;
        }
    }
```

运行时 nginx 配置文件

```
http {
    # use sendfile
    sendfile on;

    # Va initialization
    lua_package_path "./application/?.lua;./application/library/?.lu
        /share/lua/5.1/?.lua;/Users/zhoujing/.luarocks/share/lua/5.1
        usr/local/luarocks-2.2.2/share/lua/5.1/?/init.lua;./?.lua;/u
        .lua;/usr/local/lib/lua/5.1/?.lua;/usr/local/lib/lua/5.1/?/i
    lua_package_cpath "./application/library/?.so;./?.so;/Users/zhou
        lib/lua/5.1/?.so;./?.so;/usr/local/lib/lua/5.1/?.so;/usr/loc
    lua_code_cache off;
    lua_shared_dict zhou 10m;lua_shared_dict jing 2m;


    init_by_lua require('nginx.init'):run();



    server {
        # List port
        listen 9110;
        set $template_root '';

        location /static {
            alias pub/static;
        }

        # Access log with buffer, or disable it completetely if unne
        access_log logs/development-access.log combined buffer=16k;
        # access_log off;

        # Error log
        error_log logs/development-error.log;

        # Va runtime
        location / {
            content_by_lua_file ./pub/index.lua;
        }
    }
```

然后我们就可以看到关键的请求入口在于：

```
# Va runtime
        location / {
            content_by_lua_file ./pub/index.lua;
        }
```

我们来看下这个文件./pub/index.lua

local config = require('config.application')

local app = require('vanilla.v.application'):new(config)

app:bootstrap():run()

vanilla 将所有的业务请求打到这个入口，通过加载运行时配置实例化 app，然后通过初始化 bootstrap 操作装载相关的路由、面件、视图、错误处理甚至是 Waf 等，最后基于此运行。

下面是 bootstrap 的代码示例：

```
local Bootstrap = require('vanilla.v.bootstrap'):new(dispatcher)

function Bootstrap:initRoute()
    local router = self.dispatcher:getRouter()
    local simple_route = require('vanilla.v.routes.simple')
    :new(self.dispatcher:getRequest())
    router:addRoute(simple_route, true)
end

function Bootstrap:initPlugin()
    local admin_plugin = require('plugins.admin'):new()
    self.dispatcher:registerPlugin(admin_plugin);
end

function Bootstrap:boot_list()
    return {
        Bootstrap.initRoute,
        Bootstrap.initPlugin,
    }
end

return Bootstrap
```

我们再看看关键的run方法：

```
function Application:run()
    self:lpcall(function() return self.dispatcher:dispatch() end)
end
```

在 后，开始执行 run，运行具体业务逻辑。而 bootstrap 是可选项，因为 Vanilla 框架本身默认指定了路由、视图等，bootstrap 和面件等存在的意义在于保证 Vanilla 有足够强的可扩展和定制化特性。

所在的 Phase ： content_by_lua_file ./pub/index.lua;

谈到 Vanilla 请求处理的阶段，不得不提 Openresty 处理 HTTP 请求的七个阶段。

处理 请求的执行阶段来自于 Nginx，Nginx 的 HTTP 框架依据常见的处理流程将处理阶 段划分为 11 个阶段，真中每个处理阶段可以由任意多个 HTTP 模块流水式地处理请求。

Openresty 通过 ngx_http_lua_module 将 Lua 特性嵌入 Nginx，ngx_http_lua_module 属于一个 Nginx 的 HTTP 模块，并为高性能服务开发封装了 7 个相应 HTTP 请求处理阶段如下：

set_by_lua

content_by_lua

rewrite_by_lua

access_by_lua

header_filter_by_lua

body_filter_by_lua

log_by_lua

这一系列 phase 真实就是一堆钩子，Vanilla 默认将请求处理放在了 Content 这个 phase，是希望将主要业务处理集中处理，减少开启真他 phase 执行的消耗。

内部实现的面件机制可以很好的满足分阶段处理业务的需求。另外也实现了对所有 OpenResty 指令的封装(新版本的也会跟进加入),只需要在 nginx 配置骨架中配置即可使用。

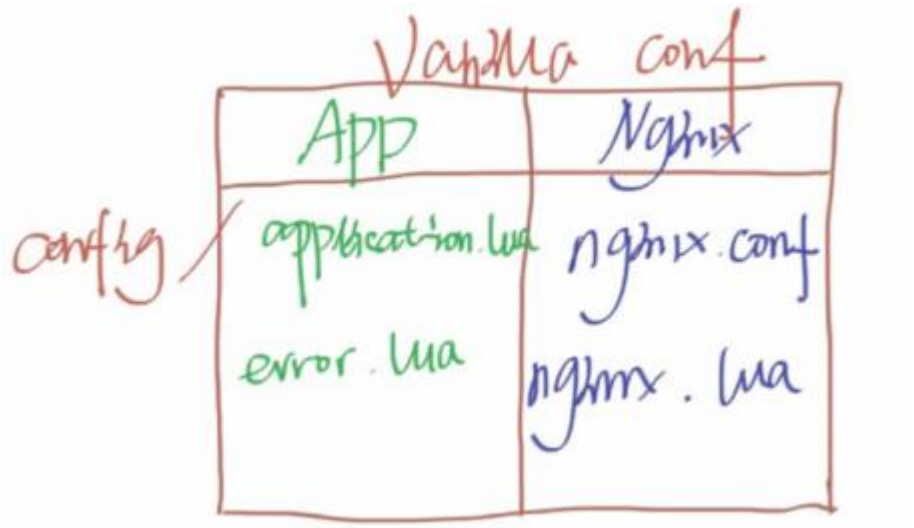
有同学会发现，Openresty 所提供的 LUA 解析指令还有以下两个：

init_by_lua 作用在配置加载阶段，init_worker_by_lua 作用在 worker 进程初始化阶段，并非请求处 理阶段。随着 OpenResty 的发展，今后必然会支持更多的指令，这属于 OpenResty 和 ngx_lua 模块使 用范畴，内容不少，在这里先不展开说了。

另外，需要重点提一下的是，Nginx 输出过滤器是流式处理模型，一个数据块 body filter 就被调用一次。 所以 Vanilla 将所有的结果渲染封装到 response 中，一次性输入，减少多次调用 body_filter 带来的消耗。

Vanilla 配置

App 配置 Nginx 配置



APP 方面：

config/application.lua 应用相关配置（指定路由、bootstrap 等） config/error.lua 定义用户级别错误信息。

Nginx 方面：

config/nginx.conf 前面说过这是 配置骨架，有多指令支持的需求可在此配置（Vanilla 默认只使用 content_by_lua） config/nginx.lua 分成 common 和 env 两个部分，common 段配置了对 nginx 指令封装 的调用，env 段配置了不同运行环境所使用的端口，缓存开关等。

下面截取 nginx.lua 片段（据此生成的 配置文件见上文）：

```
local ngx_conf = {}

ngx_conf.common = {
    INIT_BY_LUA = 'nginx.init',
    LUA_SHARED_DICT = 'nginx.sh_dict',
    CONTENT_BY_LUA_FILE = './pub/index.lua'
}

ngx_conf.env = {}
ngx_conf.env.development = {
    LUA_CODE_CACHE = false,
    PORT = 9110
}

ngx_conf.env.test = {
    LUA_CODE_CACHE = true,
    PORT = 9111
}

ngx_conf.env.production = {
    LUA_CODE_CACHE = true,
    PORT = 80
}

return ngx_conf
```

其他数据或者信息的配置也可在此目录下用 return table 的方式简单实现。详细的 Vanilla 配置使用，请查阅相关文档： https://idevz.gitbooks.io/vanilla-zh/content/cconfig.html

Vanilla 的路由器和路由协议

Vanilla 的请求是依靠一个路由器和一系列路由协议来作为分发依据的，Vanilla 实现了一个路由器，这个路由器里面有个路由协议枝，可以方便的装载和卸下路由协议，vanilla.v.routes.simple 是 Vanilla 默认使用的路由协议，比如访问 http://localhost:9110/index/home 则是路由到 index 这个 controller 下面的 home 这个 action 。

的路由协议实现的也相当简单，只需下面两步即可：

实现一个 方法，这个方法目的很简单，根据 URL 返回相应的 controller 和 action。

2.在路由协议中有个 route_name 字段，并实现 tostring 原方法，将 route_name 返回即可。

3.还有个更简单的方式就是仿造 vanilla.v.routes.simple 重写一个：）

Vanilla 的视图

类似的思想，Vanilla 的视图引擎也是可以替换，方法同路由协议的实现相同。

Vanilla 的插件

的面件来源于 af,同样支持 6 个,在钩子方法内可以获取到当前请求的 request 和 response 实例,很方便在不同的时机去做相应的操作。而比如 Waf 或者 rewrite 等操作也可以在 bootstrap 中在请求处理的最开始部分去实现。

```lua
local AdminPlugin = require('vanilla.v.plugin'):new()

function AdminPlugin:routerStartup(request, response)
end

function AdminPlugin:routerShutdown(request, response)
end

function AdminPlugin:dispatchLoopStartup(request, response)
end

function AdminPlugin:preDispatch(request, response)
end

function AdminPlugin:postDispatch(request, response)
end

function AdminPlugin:dispatchLoopShutdown(request, response)
end

return AdminPlugin
```

错误处理

Vanilla 的错误处理分为两个部分：系统和应用，系统的错误处理由框架本身定义，直接使用即可，每个应用可以在自己的 config 目录下 error.lua 中配置自定义的错误处理。

使用 的方式捕获了所有运行时的异常，并调用系统定义的错误进行处理，开发环境下直接输出到页面。APP 运行中的错误也能被全部捕获，下面举例说明：

./app_test/config/error.lua（自定义一个编号为1000的500错误，错误信息为"Err Test."）

```lua
local Errors = {
    [1000] = { status = 500, message = "Err Test." },
}

return Errors
```

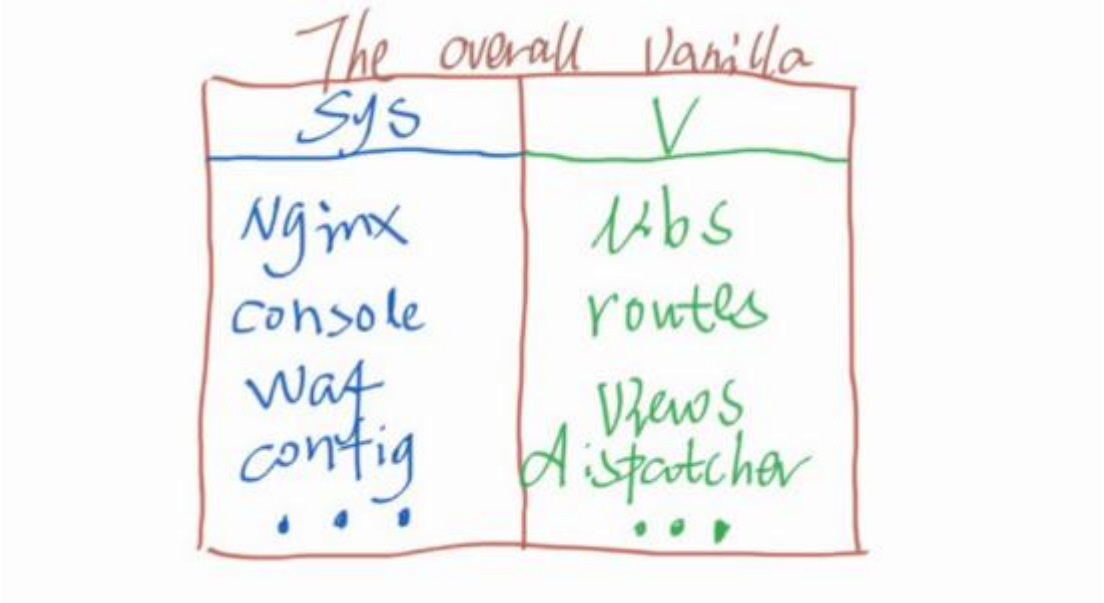
我们可以在业务中这样使用这个错误：

```
local IndexController = {}

function IndexController:index()
    error({ code = 1000, msg = {error_controller='IndexController'}})
    return 'test...error'
end

return IndexController
```

浏览器访问： http://localhost:9110/ 可以看到效果

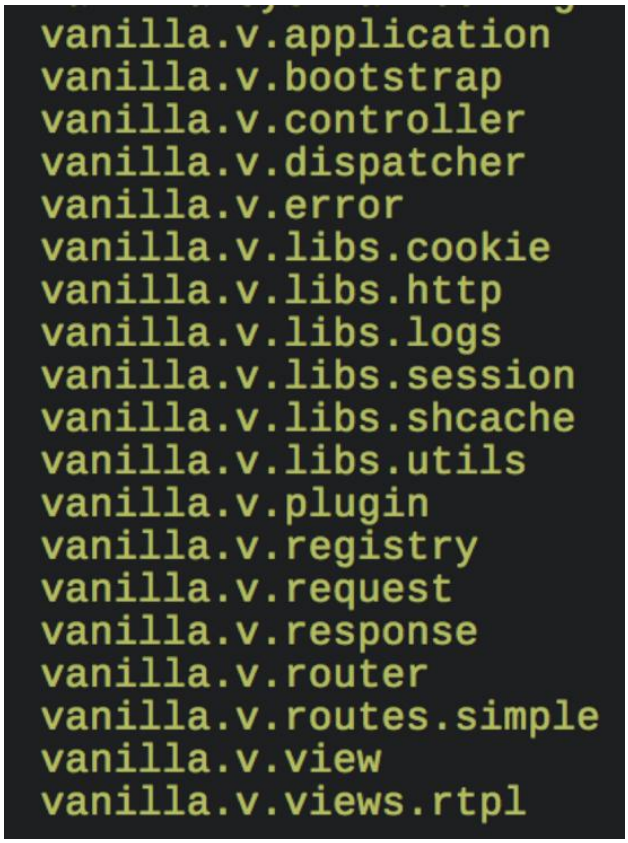
内建库

Vanilla 实现了一系列内建库，分为系统支持和业务处理，如下图



系统支持包括交互式命令行的实现、Waf、Nginx 相关封装，目的为了 Vanilla 使用更便捷，业务开发不需要涉及。

而上图中 V 的一列则是业务开发中，需要用到的包，详细列表如下：



详细使用请查阅文档： https://idevz.gitbooks.io/vanilla-zh/content/overview.html

再发送一遍 Vanilla 的项目地址：https://github.com/idevz/vanilla 欢迎各种、watch、fork 更期待大家的。

以上就是我今天分享的全部内容，感谢大家的参与。有任何问题欢迎随时交流，再次感谢大家。

浅谈 OpenResty 未来发展--章亦春

文章来自：
http://mp.weixin.qq.com/s?__biz=MzI2MDAyOTIyNA==&mid=400642479&idx=1&sn=157c008bdb0290286d5b85cdbf0bed3f&scene=2&srcid=1120jdwydiuq6PBssAsVW2ts&from=timeline&isappinstalled=0#wechat_redirect

根据春哥 OpenResty2015 大会演讲现场视频整理，如有错漏，属于笔者问题，和演讲者无关。最后 10 分钟没有整理成文字，请大家期待下周的视频完整版本。

看了两天的视频，一个字一个字打出来的，深感这个演讲分量很重，值得大家细细体会。

内容相当多，大家慢慢看…

大家好，我叫章亦春，我喜欢用 agentzh 这个名字，我很讨厌别人把我这个首字母 a 大写，我觉得特别丑，所以大家千万不要大写，这是雷区。

今天主要想和大家分享下 or 的过去，现在和未来。如果我在其他的 it 交流会议上面重点介绍我做了什么，或者正在做什么，那么在 or 这个我们自己的会上面，我就可以放心大胆的说我们将要做什么。

由来

其实最初我只是想自己搭一个个人博客，我不想用 wordpress，也不想用其他现成的技术，我想自己用一个轻量级的技术做出来。所以最初只是一个非常单纯的想法。

我的第一份工作是在 yahoo 中国的搜索部门，我当时有很多很碎的业务，比如搜索结果的相关推荐，或者垂直搜索，还有下拉列表的自动提示，很多这种很碎的功能在搜索部门。于是我们老大让我写一个通用的平台，可以方便的构造这样的接口，给前前端来展现。这些接口可能是吐 json 这种很简单的 API。那个年代，API 的概念还是方兴未艾，顺应这个潮流，做了第一版的 or，是由我当时最喜欢的脚本语言 perl 写的。花了很大的精力去优化，因为当时在 yahoo 使用的生成机器是爬虫淘汰下来的机器，就是连爬虫都觉得很慢的机器，拿给我们跑业务跑生产，所以在用 Perl 做这种访问量很大的接口时候，很难进行优化。当然我也不断去把越来越多的 or 功能用 c 去实现，但是还是很难达到让人眼前一亮的性能要求。所以现实是非常残酷的。

我在 yahoo 有一个同事，叫 chaoslawful，大名叫王晓哲，我很有幸拜他为师，学了很多系统编程方面的东西，因为他是一个很了不起的人，也是一个自学成才的工程师。

在 09 年的时候，就一直有一个想法，把之前用 perl 的 or 进行重写，然后达到一个很高的性能要求。在 09 年的时候，我和晓哲转到了淘宝，当时加入了量子统计这个数据平台部门，

做量子统计这个产品，给卖家提供流量统计、销售统计、广告效果报表的数据分析产品。现在这个产品已经不存在了，是很多年前淘宝的一个产品。它有很复杂的业务逻辑，对性能也有比较高的要求，同时后台的数据量是很恐怖的，淘宝大部分商家的数据都会在这里展现，进行各种维度和搜索的分析。当时，我就想，我既要把 or 重写了，得到很高效的框架，同时也要和部门的同事一起，把量子统计在这个平台上面构建出来。所以很难得有这个机会，同时也是非常大的一个挑战。我希望在这里尝试一些从未尝试过的、新奇的玩法。所以大家现在看到的 or，其实就是在那段时间完成的，09 年到 2011 年，我在淘宝的时间中。同时把量子统计这个 web 应用进行了非常高效的实现，相比之前 PHP 实现的老版本，代码量减少了 90%，性能提升了一个数量级，包括延时，包括并发。我们后台的一个数据库工程师和我说，之前老板点击一个黄钻卖家的报表，离开工位倒杯水回来，报表还在转，新版的话，啪一下就出来了。所以效果还是非常明显的。只用非常少的机器，就能处理非常复杂的数据库相关的查询，因为它数据源很多，所以这里面有复杂的关系型的融合还有分表的逻辑。

Why lua

那么，大家可能会觉得，lua 作为一个非常简单的语言，来表达很复杂的业务系统，可能会很吃力，事实上，我们在构建这个业务系统的时候，并没有写多少 lua 代码，但确实是由 lua 来驱动的。技巧就是我经常鼓吹的一种玩法：设计一种针对业务的小语言，或者说 DSL。这个应用背后的数据平台，就是用我自己设计的一种类似关系型的语言来表达的，然后我自己实现的编译器，处理这种业务描述，生成高度优化的 lua 代码。lua 代码生成之后，你会觉得人类是很难写出来的，因为有很多优化，是人很难做对的。对查询的拆分，对查询的优化，所以我又在里面实现了一个中间件，但代码很少，所以我等于把 lua 语言当做虚拟机的机器语言在使用，业务我是用最适合业务模型的一种表达方式来表达的。这块儿我可以讲很多，可以讲一天。这块儿和 or 配合起来去使用的一个优势就在于，这个平台框架很轻，没有什么乱七八糟的东西，同时 lua 语言是一种高度动态的语言，我可以根据需要去做一些魔法般的事情，把它做成我的虚拟机的 CPU，来进行代码生成会非常的方便，同时效率也会非常非常的高。后面我还会讲到类似的例子。

我觉得很多时候不用纠结到底使用什么编程语言，更多的应该考虑用业务语言来表达我们的业务。比如团队里面产品经理用犀利的语言把业务描述的很精确，那么他写的这个文档， 已经是这个业务系统本身了。我们写个编译机让它跑起来就行了。大家可以去想一想这个思路。

应用场景

在 2012 年的时候，我加入了 cloudflare。2011 年的时候，我在福州，我不是福州人，很多人认为我是福州人，回老家，其实不是，我们只是随便挑选了一个南方城市，去过一种半隐居的生活，有更多的时间专注于开源项目，特别是 or。现在大家看到的很多高级功能，都是在那段时间完成的。在 2012 年，福州田园生活过了一年之后，就加入了美国的 cloudflare 公司，这是一个 cdn 公司。所以我之前在搜索行业、在数据分析行业混了几年，然后到了 cdn 行业。机缘巧合，这家公司希望用 or 来构建他们的基础设置。我觉得去美国过类似的田园生活也不错。但我没想到的是 or 在 cdn 行业获取了很大的成功，有很多的公司会去使用。这也是我最初没有想到的，我最初想的是 web 应用，对富客户端或者移动客户端的 API，使用 or 构建是非常合适的。大家之前看到的 web 引用，其实也是由这些 API 驱动的，大量使用了 ajax 这样的技术，像模板渲染、用户交互的控制流逻辑实现都是在浏览器里面完成

的，这是一个胖客户端，当时花了很多精力去搞 IE6，所以又写了很多 javascript，还是很辛苦的。

最开始我们考虑把 lua 嵌入到 nginx 当中，也是在淘宝的时候，晓哲老师给我建议了这样一个技术方案，我觉得靠谱。我最早的改写方案是基于 apache 去写 c module，但是 apache 的代码看得我有些云里雾里，所以在晓哲老师的建议下，开始把工作重心放在 nginx 上面。这时候可能知道 nginx 的同学也不是很多。nginx 的请求有多个不同的处理阶段，在不同的阶段，我们可以插入一些 lua 代码，插入自己的逻辑，进行一些控制或者事件记录。当然最常用的就是 content_by_lua，这个是用 or 做完整外围应用的一种玩法，刚才很多同学已经演示了，可能对于 cdn，对于动态均衡器来说，他就不会用 content_by_lua，而是用 nginx 自己的 proxy_pass 这样的反向代理模块，而且会去使用 access_by_lua 和 rewrite_by_lua 这样一些在更早阶段执行的 hook 去加入自己的一些逻辑。我开始的时候写了一些 resty 的 lua 库，可以在 lua 层面做一些代码的复用。然后社区的热心用户贡献了很多 resty 库，特别是今天上午演讲的 Aapo 同学，贡献了非常多的 resty 库，这些都是让我很振奋的事情。在做这些库的时候，我有一个愿望，就是他们都是相对独立的，尽量之间不要有相互依赖，这样可以确保一个清晰的结果，lua 的优点就在于他的简洁明快。

刚才提到三个应用场景，API Server，还有 HTTP Proxy，这个在 CDN 行业用的比较多。Web Application，这个相对少一些。这个和 nginx 本身的使用方式有关。有些同学可能没法接受把 nginx 本身作为 Web Application 的容器或者 server，但这是我的初衷。

OpenResty 的哲学：包容、高效、简单、实用、有趣

在发展的过程中，or 秉持的是兼容并包的思想，我们并不排他。出于 nginx 在整个 web stack 中位置的特殊性，我们可以很方便的和现有的技术进行融合，比如 PHP、Python、go、nodejs，我们在网关这个层面，所以我们可以同时和其他后端应用并存，虽然我还是更倾向于更纯净的方案，但事实上，在 or 社区里面，我们的用户来自各个社区，Ruby、Python 甚至 java，所以我很高兴看到不同语言社区的同学，把他们自己社区的文化，一些看待问题考虑问题的方法，能够带到我们社区里面来，扩展我们的思路。我们也鼓励各种混合的使用方法，至少对于现存的系统来说，我们的迁移可以一块一块的进行，当然用户认证的这个逻辑需要优先迁移到 nginx 这个层面。这种玩法也是我最初没有想到的。我最初用 nginx，是看中它 http、io 事件处理模型这块的实现，大家最最最为经典的用法是把 nginx 最为前面的网关，后面用 fcgi 去连 API server，或者 proxy_pass 去做反向代理。那么在这种结构下，or 更有机会去做更多的事情，融合现有更多的业务。

我自己也用过这种技巧，在量子的时候，有一个实时统计的引擎，有很复杂的线路协议，当时我在做业务迁移的时候，因为很严格的上线的时间点，所以没有时间去完成一个非阻塞性能很高的实时引擎客户端，所以我简单的用 Perl 实现了一个客户端，虽然我很仔细的实现了，但其实不行，扛不住，成为整个系统的瓶颈。在上线之后，有了更多时间之后，我写了一个纯 c 的 nginx module，来进行非阻塞通信。刚才我讲用 perl，用 c，其实并不是我直接用 perl，用 c 去实现，而是用了一个写程序的程序的技术。这个实时数据库的维护者，也是我的同事，他维护了一篇非常漂亮的文档，用来详细讲解线路协议的每一个主要方面。我一看到这个文档，我灵机一动，这个文档就是我的客户端的实现。我写了一个 Perl 脚本去自动分析这个文档，把里面的数据，里面的结构，全部抽出来，变成一个数据结构，自动生成

Perl 实现，自动生成 c 的实现。事实上，我还让它自动生成了测试集，测试也可以自动生成。在这个过程中，我发现他文档中的一些笔误，因为毕竟是人写的，而我的分析器尝试把它当做程序来运行的时候，就会发现很多细节问题。所以文档真的很重要，重要到你可以用文档来生成任何东西，包括实现，包括测试，中文的，英文的，葡萄牙文的各种人类的文档都可以。这是一种很有意思的思路。这也使得你的业务和具体实现无关，在任何一天，你都可以换掉下面的实现，而不用动上面的实现代码。像刚才我举的那个例子，我可以很惬意的把 Perl 实现换成 c 实现，其实我只是写了一个模板。前段工程师可能很熟悉这种模板的技术，我们一般会使用模板来生成 HTML 和 CSS，但是你有没有想过，我们可以用模板生成任何东西，包括你的程序本身，为什么不呢？所以我使用写程序的程序的技术的时候，我恰恰是用了那种通常是生成 HTML 的模板引擎，当然更复杂的代码生成器，还是需要专门的代码生成器技术。这个说远了，我的意思是说，这种兼容并包的哲学，可以打开我们的思路，我们不必局限于 lua 这个语言，nginx 这个东西，其实我们只是根据需要，把很多我们需要的块拼接起来，而并没有任何的排他或者任何宗教信仰式的极端主义的倾向。

在设计整个 or 的过程中，我们还是有几个比较清晰的目标。第一个目标首先是简单，simple。这也是我为什么不喜欢很多 java 框架的原因，就是一定要简单，不需要的东西一定不能存在。然后要小巧，这点我也是很执着的一个事情，我算是半个 ops，运维人员的背景出身，所以我希望我在机器上面跑的程序能够小巧，以至于能够完全控制整个代码机，在出问题的时候，可以自己去分析，自己去追踪，乃至自己去修复，而不管出问题的组件处于软件系统的哪个位置，所以这个有点偏执，但是这样可以让我晚上睡个好觉，因为我不再惧怕任何诡异问题，因为本来也就没有什么诡异问题。第三个就是要快，这是我在 yahoo 和淘宝工作的最直接的感触。后来到了 cloudflare 也是类似的情况，其实现在的互联网环境变得如此的拥挤，就是看上去很不起眼的东西，往往就能吸引到很多流量，有很高的性能要求，而给我们的机器，通常不尽如人意，yahoo 当时爬虫淘汰下来的机器是一个极端例子，但也能说明问题，所以要快。如果一个程序跑的还不如我收工干活来的要快，那么为什么还要写程序呢？最后一个目标是一定要灵活。这也是我非常看重脚本语言的一个方面，静态语言有很多优点，但是我希望在做业务的时候，我的手是足够灵活的，可以做任何我想做的事情，而不会受到很多不必要的束缚。这一点，在座的脚本程序员都会深有体会。有时候，我并不希望实现语言给我太多的限制。

那么，很重要的一点，就是要实用主义。我们并不追求很花哨的学院派的概念，虽然作为业务爱好还是很有裨益的，但是对于工程的实践来说，实用主义应该是放在第一位的。首先要把活干了，也得得到一个足够高效、足够健壮、足够优美的一个系统。优美是最高境界。那么 or 就是实用主义的产物。我希望我们继续贯彻这个方针，保持实用主义者的称号。

还有一个很重要的，就是开源工作者很看重的：有趣。我们假设开源是一个编译器，它有优化选项，-Ofun，我们是针对乐趣进行优化。这一点看上去是和实用主义原则冲突，其实不然。因为对于一个工程师来说，最有意思的莫过于自己的技术，自己搭建的系统，自己设计的方案，能够在线上跑的非常好，能够服务越来越多的用户，这是非常大的一个乐趣。对开源工作者来讲，他也希望自己的代码能够跑在尽可能多的公司的服务器上，能够收到尽可能多的用户的感谢信。这个你想，在一天干活最痛苦的时候，突然收到一封来自世界另一个角落的用户的感谢信，字里行间洋溢着一种感激，一种欣喜，那你这一天立马就会变得非常美好。所以，要让乐趣变成我们工作的主旋律，而不要让工作变成一种负担。这也是我在做 or 的时候，希望我自己，以及每一位参与开发或者普通使用的用户，都能去做到的一件事

情。我们会看到在一些社区里面，很容易产生一种愤怒，比如说有某个用户问了一个很小白的问题，或者出言不逊，诸如此类的，很容易让人变得毛躁起来。我希望这种消极情绪能够尽量的消极乃至没有，因为我还是希望世界能多一些美好，多一些正面的积极的情绪，对我们每个人也会更好一些。

社区

那么我们有一个正在不断成长的社区，我每天花费你们没法想象的时间，去处理用户的邮件和补丁，有新特性，有 bugfix。会用各种语言去写，有时候会忘了正在用什么语言，可能前半截用的中文，后半截用的英文，有时候也会出现混乱的情况。360 的几位哥们儿，温铭、院生、艾菲他们也自己开了一个 qq 群，我不在那个群里面，因为我很久不用 qq 了。然后 github 上面也会有很多 issue 和 pull request，当然一般的讨论还是希望能够在邮件列表里面，这样可以方便通过搜索引擎进行检索。那么现在我们还有了会议，让原本只知道 web ID 的朋友，能够看到真人，很多朋友只认识他们的 github ID，或者 email 里面的 nick，而不知道他们长什么样子。其实很多人和我想像多年的样子完全不一样，都是很正常的，还好没有性别和我想像的不一样。

我很希望去写一本书，那么 360 的温铭他们的团队已经开始写一本叫做《OpenResty 最佳实践》的书，我自己脑袋里面也会有一些想法，我们可能会合作去合著一本书，这都是可以去讨论的。我脑袋里面那本书叫《programming OpenResty》，希望能够把我这么多年来，做 or 的一些感悟和心得，能够写下来。这样也省的我在邮件里面，翻来覆去的重复一些很基本的东西，也能让更多的人受益。

未来特性

包管理一直是社区里面呼声很高的一个功能。我也希望 or 官方提供包管理这样的东西，我们可以很方便的上传、分享、安装自己或者他人的 lua resty 模块，lua 库，或者基于 or 的应用程序和工具，都可以通过这个统一的工具链。大家熟悉的比如说，nodejs 的 NPM，Perl 的 CPAN，ptyhon 的 PIP， Ruby 的 gems，有很多现有包管理工具，我们可以去参考。我们设计还没有定型，我只是给大家看下我脑海里面很粗糙的想法。这个名字不太好起，我跟温铭，跟院生他们商量，我觉得目前最好的名字是 iresty，但还是稍微有点儿长，像 PIP、NPM 都是三个字母。iresty.org 这个网站，我们可以检索所有 lua resty 的库，或者 driver，比如需要 ZooKeeper、MongoDB 这样的客户端库的时候，就可以先在这个网站上进行搜索。没有的话，或者没有合适的话，你可以去着手开发，这样可以避免重复劳动，也可以鼓励大家的协同。

命令行工具也会叫 iresty，如果大家有更好的想法， 可以在邮件列表或者 qq 群，告诉我们。现在是一个雏形的想法。我们可以去 install 一个库，可以删除卸载这个模块。我比较坚持的一点是，模块名字前面要求加上 github ID，这样可以避免其他社区包管理里面常见的一种抢坑的问题，就是我这个东西还没做出来，我先抢这个坑，别人就不能注册这个名字了。我希望像 github 一样，鼓励大家不在名字空间上面去争夺，像 guthub 一样鼓励大家去 fork，去自己定制，但是还是希望尽量去协作，但我不希望协作是强制的。也可以去 search，这些基本功能是需要的。

我还希望有一个命令行的文档，可以看到标准模块和安装模块的格式化的文档。这样省的我

们在网上搜索，那也挺费事的。在终端里面可以看到，这比较符合我们 UNIX 程序员的习惯。当然不仅能查看库的文档，模块的文档，还可以查看 nginx lua 或者 nginx 里面某个配置指令的文档，我也希望能做到这一点，这样所有的文档查找都能通过一个命令行工具来进行，甚至你终端支持颜色的话，显示出来的文档是彩色的。

resty 这个工具已经实现了有一段时间，可能有些同学还不知道。这是一个很简单的 or 命令行，你可以用-e 选项去执行一行代码，你可以使用几乎所有的 or 里面暴露的标准 API，你也可以指定一个文件，用 or 实现一些命令行的工具。我们注意到这个工具也可以用于一些在线的分析，比如说同一份业务代码，即可以作为服务跑在 nginx 里面，同时也可以在命令行里面，供运维人员去快速检查一些当前生产数据的状态，所以这也会非常有用。还有同学会用这个工具做单元测试，因为本质上这个 resty 工具，自己启动了一个没有脑袋的 nginx，这个 nginx 不会监听任何东西，只是一股脑的从头执行到尾，然后退出。

我们刚才说到 irsety 包管理工具，会同时支持 lua 库和 or 应用，我希望有更多的 or 应用可以通过 iresty install 就可以装上，就可以运行。

我们可能会对 LuaRocks 有限性的兼容支持，而且我个人对 LuaRocks 不是特别的喜欢。所以我希望 or 有自己的包管理系统。同时标准 Lua 世界的库，由于各种原因，没有考虑 or 的阻塞和非阻塞的问题，它没有使用 or 提供底层的原语接口，比如说 socket 接口，所以这些库通常会阻塞 nginx 的事件循环，造成并发的严重下降。

还会有更多官方的二进制发布包。比如说我们现在正在做的 windows 的二进制发布包，里面有一个 32 位的 windows 版本，刚才 360 的同学们也讲到，他们维护了一个 windows 的分支。现在计划是把他们的优化工作，融合到官方的 windows 版本中来。我现在这个是非常简单的，并不适合做生产，只适合在 windows 上面做开发，有人有这样的需求，虽然他们最后的生产环境是 Linux 或者 BSD。 我现在的 windows 版本是用 MinGW GCC 来编译的，我希望最终使用微软的工具链来编译，以获得最好的性能，和最少的依赖项。还会有主流的 Linux 发行版本的仓库，这样大家升级是 apt-get update 一下。这些工作都是近期会重点做的事情。我还希望 iresty.org 网站上传的 lua 模块和应用，能自动生成二进制各个发行版的安装包，这也是个很 cool 的 feature。

还有一个大家呼声很高的特性，就是让 nginx 成为一个一般的 TCP server，而不仅仅是 http server，我们就可以去做更多的事情。那么 nginx 最新提供的 stream 子系统就可以做到这一点。我们会有基于这个子系统的 lua 模块和 echo 模块。echo 模块已经完成开发，大家可以在 github 上面去查看，lua 模块也正在开发过程当中，它会拥有和现有的 nginx http modlue 相同的 lua API，所以你们的很多应用和库，可以不加修改的运行在一个新的子系统上面。

类似的，我们会有 UDP server，其实我很想看到的一点，是基于 or 的 DNS name server，我觉得这是一个很有意思的应用。

下一个，模板。上午 Aapo 老师介绍过他自己写的 lua-resty-template，我自己也有我的一些想法，就是 lua-resty-tt2(jemplate)。作为一个 Perl 程序员，我非常热爱 Perl 里面 tt2 这个模板语言。它其实跟 Perl 没有太多关系，Perl 社区的一个大神，他写了一个 jemplate 这个工具，可以把 tt2 生成为独立的、可以在客户端运行的 JavaScript，这就意味着同一份模板，即可以

跑在服务端，也可以跑在客户端，这是一个非常有趣的想法。这得益于这个模板语言既不是 Perl，也不是 lua，也不是 JavaScript，而是它自己的一种小语言，足够简单但有足够强大。我曾经用一个脚本语言实现了一个比较完整的 x86 反汇编器，就是通过这个模板。所以这块也是我想去尝试的官方模板引擎库。当然，如果大家有兴趣，也可以去移植 Python 里面常见的模板语言到 or 的场景下，那么 Python 原有的应用可以更容易的迁移到 or，模板文件至少不用动。

大家注意我微博的话，可以看到我很多时间花在 Streaming Regex 这个正则引擎上面，这是一个从零实现的正则引擎。web 世界很难离开正则，因为总会遇到各种文本处理，对于 CDN 来说尤为如此，因为我们需要对请求进行很复杂的模式匹配，不管是 WAF，还是 CDN 的规则的调度分发，还包括对响应体的一些正则替换，去掉或者删除一些东西，比如把电子邮件地址替换成它对应的图片，这些都需要很强大的正则引擎，支持流式处理，可以分块来进行匹配，这样的引擎是永远不会往后回溯的，我处理一块扔掉一块，再处理下一块，用 O(1) 的内存来处理任意长度的数据流。这个 libsregex 还在开发当中，我做了一些尝试和实践，最新的是基于确定性有穷自动机（DFA）来实现的。我觉得这里面有一些算法的创新，我也跟 PCRE JIT 的作者有过沟通，他也很关注这个项目，很早就主动和我联系，让我有些受宠若惊。所以我很高兴通过这个项目，吸引了一些正则这个领域举足轻重的人物的注意和帮助。他们会问需不需要帮助啊，或者说你的这个算法好有意思啊，你教教我好不好，我们一起讨论一下。这些互动是非常非常宝贵的，这也会打破公司的边界甚至国界，这是开源带给我们的好处和乐趣。我还写了一个 nginx 的模块，用正则在响应体里面做替换，它现在效率不太好，但是它实现了这个想法，它现在是确定性有穷自动机的算法，所以比较慢，和 RE2 的一般情况一样慢，我需要像 PCRE JIT 一样性能级别的引擎。最新的 libsregex 里面的 DFA 引擎，在平均情况下可以达到 PCRE JIT 甚至更高的效率，最坏的情况下，也能和 RE2 或者比 RE2 更好。Intel 在今年 10 月份开源了一个做了 7 年的正则库，叫 HyperScan，也是值得去参考的，只不过它的语义和 Perl，和 PCRE 的差别更大一些。但它的优点是至少可以处理上万个正则的流式匹配。

另外 lua-resty-pegex 也是我想做的一个库，我其实并不想把社区可以做的事情都做完，我只是捡一些我特别想做的库。一旦有了 iresty 之后，标准库和非标准库的边界已经不是那么清晰了。pegex 是 Perl 里面我非常喜欢的用于语法分析的引擎，可以用于构造编译器，小语言，写程序的程序，我们需要这种分析机构造器去实现小语言的编译器，让计算机认识我们定义的语法，认识我们某个同事写文档的格式或者惯用语，那么让计算机懂得这个文法，我们就需要另一种小语言，来描述这种小语言的语法，那么这些工具就可以把描述变成实现，让计算机能够理解我们想象的语言的语法。语义分析可能会更复杂一些，因为没有这种自动化工具，但实现起来也没有那么复杂。编译原理应该是我计算机专业学习当中，让我受益最大的一门课。在这里领域的很多想法，不仅适用于我们常见的 gcc、java 这样的全功能的工业级别的编译器，也适用于小巧的，只有几百行、几千行自己实现的小语言的编译器。

基于这种小语言的思路，我希望 or 可以提供更多的小语言，让大家可以去接受、去摆弄这样的一些想法。我脑海里面的 Edge 语言就是这样一个例子，它是为 CDN 行业里面典型的业务模型、业务逻辑来设计的，可以很方便的通过基于规则的语法来表示很复杂的过滤规则、分发规则、WAF 的防火墙规则，用一种统一的方式来表达我们的意图，而不是去表达一种实现。那么我们的编译器就有机会在更高的层面上进行业务的语义分析，来写很了不起的全局优化。比如说，我们有 100 个规则，都需要对 URI 进行正则检查，我们自己实现的优化器，

可以把规则里面对 URI 的模式，合并成一个自动机，这样我们只用对 URI 扫描一遍，而不是 100 遍。所以这样就打开了我们优化的可能性。

（这里举了一个例子来说明，大家可以看视频来理解）

我很想看到的一件事情是 or 有一个官方的 WAF 平台，现在开源的一些解决方案，至少我觉得可以做的比它更好。包括 cloudflare 正在线上使用的 WAF，我觉得也可以做的更好。我希望最好的 WAF 应该是开源的，而且是 or 里面的。对于 WAF 来说，是一个很经典的 DSL 的例子，我们使用一种小语言，比如刚才的 Edge 语言，也可以扩展到 WAF 场景，动作可能是 reject、pass、打分，分值达到某个程度之后 reject，返回一个 403，完成一个 capture 来进行验证。

我觉得特别有趣的一种玩法是从 DSL 生成高度优化的 Lua 代码。这样就可以发挥包括 LuaJIT 在内的组件的威力，对于 JIT 编译器来讲，Lua 代码分支越少，越简单，越有可能生成最高效的机器指令，同时我们因为使用了编译的方法，我们抽象本身的开销可以在编译期，在上线之前完成。我们最后上线的时候，是生成的非常小巧的 lua 代码，没有什么运行时开销，也没有什么依赖项。魔法都可以在编译期完成，这是一种脱去抽象本身开销的很好的思路。现在很多 web 框架，是在运行时进行抽象，那么会导致在线时候，内存和 CPU 的损耗很大，因为你引入了很多层次，这些开销是运行时的开销。这一点也可能不是上线之前，也可能是在一些特殊的配置端口，比如说 CDN 的配置界面上面，我们会做编译，通过一些复制的方式，比如 KT，推送到全球各个网络的各个节点，我们可以推送 lua 字节码。在 CDN 的场景下，等价于于我们为每一个客户，高度定制了一个 CDN 软件。这样如果你的 CDN 配置很简单，那么生成的 CDN 软件就非常简单。

我们社区常常会吸引到其他社区，包括 redis 社区的邮件，比如 redis 社区的一个哥们儿就发邮件和我讨论 stateful 这个概念，这个我也是从他那里听说的。所以有一些很有趣的协议和方法，当然每一种协议都有自己的折中和自己的应用场景。我希望后面 or 能在这方面多做一些尝试，各个 or 之间形成更加默契配合的关系。同时能给请求引入状态，这个状态可以在同一机房的不同节点传递，甚至是跨机房，在全球节点之间传递。这些都是可以玩的一些东西，也会有它们各自的应用场景。

semaphore 是一个很重要的特性，它可以用于 ngx_lua 轻量级线程之间的同步，其实就是线程同步里面的信号量。酷狗音乐的同学一直在开发这个功能，已经有一些很不错的成品了。我还在 review 过程当中，希望能尽快融合进主干，这个可以避免使用 sleep 的方法进行同步，损耗可以下降非常多。这个朱德江同学的分享里面也会提到。

酷狗音乐的朱德江同学还实现了基于 list 的共享内存的类型支持，可以用 redis 风格的接口对共享内存进行队列操作，这个也会有非常有趣的应用场景。

我自己也很想拿 Shm-based databases 来练手，在 or 里面来实现一个内存数据库，没有持久化，或者有限持久化支持的数据库。关系类型的，或者其他类型的，时间序列类型的。

init_by_lua 一直不支持 cosockets，我希望这个能增加支持。在 nginx 启动的时候，在外部的 tcp 或者 udp 服务里面拿一些数据，这个还是非常有用的。那目前需要在每个 worker 里面做

一些比较恶心的初始化同步工作。

我们需要有更好的 cosockets。cosockets 仍然像经典的 socket 同步的使用，但它是非阻塞的，不会阻塞任何操作系统线程。我们有一些内存是分配在 nginx 的内存池里面，我希望可以避免这种分配，更适用于推送的场景，很多下游长连接保持非常长时间的场景。

UDP 或者 gram socket 需要一个 bind()方法，这样我们可以进行双向的服务发现，服务推送之类的东西。

cosockets 连接池，我希望能够基于它做一些后端的并发控制，当超过这个连接池容量的时候，可以对 connect 请求进行排队，而确保后端不会因为并发数太多而损失吞吐量，即使是 redis，如果给他并发太高的话，吞吐量也会直线下降。

ngx.connection 这是一个比较新的东西，和 cosockets 平行。我希望能够提供一个 fd，也就是文件描述符，这个 fd 可能是其他 c 库实现的，我们可以把 fd 注册到 nginx 的事件循环里面，然后可以去同步非阻塞的等待 nginx 事件循环里面的事件，比如说 read、write。这种模式的好处在于我们可以结合 luajit 的 ffi，很方便的整合现有的第三方的支持非阻塞 IO 的 c 库，比如 postgres 的 libpq，而不用 lua 的 cosockets 去重写这样的库。这个也是能够改变 or 生态系统的一个特性。幸运的是，这个特性实现起来很简单。而且 github 上面已经有了一个 pull request，叫做 wait for fd，有兴趣的同学可以一起参与讨论，一起研究。

balancer_by_lua 这个刚刚开源，你可以用 lua 来定义自己的负载均衡器，可以在每个请求的级别上去定义，当前访问的后端的节点地址、端口，还可以定制很细力度的访问失败之后的重试策略。如果我是重试还是不重试，重试失败后去重试哪个节点，都可以在每个请求的力度上进行 lua 编程。这对于很多反向代理风格的业务来讲，是一个很重要的特性。

（这里又是一个例子，建议大家看视频）

SSL 现在也很热门，大家为了安全起见，都往 HTTPS 迁移。 ssl_certificate_by_lua 可以让你通过 lua 严格去控制下游的 ssl 握手的过程。比如使用什么 ssl，使用什么证书，让不让它握手，都可以在这个环节插入自己的代码来完成。比如 cloudflare 的 ssl 网关就是利用这个特性来实现的。

OpenResty 的现状、趋势、使用及学习方法--温铭

温铭，奇虎 360 企业安全服务端架构师，OpenResty 社区咨询委员会成员。一直在互联网安全公司从事服务端的开发和架构工作，致力于用互联网技术帮助企业提高安全防护。曾负责开发过木马云查杀和反钓鱼系统。

我个人之前主要是用 Python 来完成开发工作，包括云查杀和反钓鱼系统，都是 Python 完成的。在 2011 年左右接触到 nginx 的 C Module 开发，被异步的高性能颠覆了三观，只是门槛太高，一直想找一个像 Python 一样简单，像 nginx C Module 一样高效的技术。

所以在 2012 年，得知 OpenResty 这个项目的时候，我就在企业安全一个新项目里面，使用它作为服务端的主要技术。

在今年上半年开始在 Github 上面，把积累的经验写成一本电子书《OpenResty 最佳实践》，并在刚过去的 11 月 14 号，以社区名义组织了 OpenResty 的第一次技术大会。

1. OpenResty 是什么，适合什么场景下使用

和大部分知名开源软件诞生在欧美国家不同，OpenResty 自身和依赖的主要组件都是金砖国家的开发者发明的，这点还挺有意思。

Nginx 是俄罗斯人发明的， Lua 是巴西几个教授发明的，中国人章亦春把 LuaJIT VM 嵌入到 Nginx 中，实现了 OpenResty 这个高性能服务端解决方案。

通过 OpenResty，你可以把 nginx 的各种功能进行自由拼接， 更重要的是，开发门槛并不高，这一切都是用强大轻巧的 Lua 语言来操控。

它主要的使用场景主要是：

- 在 Lua 中揉和和处理各种不同的 nginx 上游输出（Proxy，Postgres，Redis，Memcached 等）
- 在请求真正到达上游服务之前，Lua 可以随心所欲的做复杂的访问控制和安全检测
- 随心所欲的操控响应头里面的信息
- 从外部存储服务（比如 Redis，Memcached，MySQL，Postgres）中获取后端信息，并用

这些信息来实时选择哪一个后端来完成业务访问

- 在内容 handler 中随意编写复杂的 Web 应用，使用 同步但依然非阻塞 的方式，访问后端数据库和其他存储
- 在 rewrite 阶段，通过 Lua 完成非常复杂的 URL dispatch
- 用 Lua 可以为 nginx 子请求和任意 location，实现高级缓存机制

组织 OpenResty 技术大会之前，我一直认为自己是一个孤独的 OpenResty 使用者，觉得自己在使用一个冷门的技术。

虽然大家都听说过 OpenResty 或者 ngx_lua，但感觉用在生产环境中使用的却少之又少，除了几个 CDN 公司外，好像没有听说过哪家知名互联网公司在使用。而 CDN 行业之所以使用，很多是受到 cloudflare 技术栈的影响，OpenResty 的作者也在国外这家 CDN 公司。

但办完这个大会，我发现使用者真的挺多，奇虎 360 的所有服务端团队都在使用，京东、百度、魅族、知乎、优酷、新浪这些互联网公司都在使用。有用来写 WAF、有做 CDN 调度、有做广告系统、消息推送系统，还有像我们部门一样，用作 API server 的。有些还用在非常关键的业务上，比如开涛在高可用架构分享的京东商品详情页，是我知道的 ngx_lua 最大规模的应用。

2. 奇虎企业安全服务端技术选型的标准

先说下 3 年多前做架构选型的时候，我为什么会选择 OpenResty？

其实架构如何设计并不重要，因为每家公司，每个团队，他们的公司文化和技术背景各不相同，生搬硬套会适得其反。重要的是当初为什么这么选择，中途为什么调整。

我们的产品要求单机上面，服务端提供高性能的 API 接口， QPS 至少过万，未来需要支撑到 10 万。我们并没有急于去使用 PHP 、 Python 或者其他的语言来实现功能，而是先勾勒出一个理想化的技术模型。

这个模型应该具备：

- 非阻塞的访问网络 IO。在连接 MySQL 、Redis 和发起 HTTP 请求时，工作进程不能傻傻的等待网络 IO 的返回，而是需要支持事件驱动，用协程的方式让 CPU 资源更有效的去处理其他请求。很多语言并不具备这样的能力和周边库。
- 有完备的缓存机制。不仅需要支持 Redis 、Memcached 等外部缓存，也应该在自己的进程内有缓存系统。我们希望大部分的请求都能在一个进程中得到数据并返回，这样是最高效的方法，一旦有了网络 IO 和进程间的交互，性能就会受到很大影响。

- 同步的写代码逻辑，不要让开发者感知到回调和异步。这个也很重要，程序员也是人，代码应该更符合人的思维习惯，显式的回调和异步关键字，会打断思路，也给调试带来困难。

- 最好是站在巨人肩上，基于成熟的技术上搭建。采用一门全新诞生的语言和技术，需要经历语言自身发展期频繁调整的阵痛，还可能站错队。

- 不仅支持 Linux 平台，还需要支持 Windows 平台，这个是我们产品很特别的需求，很多中小企业用户还是习惯 Windows 的操作，不具备 Linux 的维护能力。

基于以上几点的考虑，考察了当时的一些方案，选择了 OpenResty 。

首先，它最大的特点就是用同步的代码逻辑实现非阻塞的调用，其次它有单进程内的 LRU cache 和进程间的 share DICT cache，而且它是揉合 nginx 和 LuaJIT 而产生的。而且 nginx 有 Windows 版本，虽然有非常多的限制，但这些限制都是可以解决的， nginx 官方 Windows 版本中不支持的特性，我们开源出来的版本都解决了。

第一次看到这样的方案，我觉得它肯定会颠覆高性能服务端的开发。为什么呢？在我之前的公司里，每天会有近百亿次的查询请求，而服务器只用了十台。

我们采用了 nginx C 模块 ＋ 内置在 nginx 中的 K-V 数据库（自己开发的），来实现所有的业务逻辑，达到这个目标。听上去很简单，但是过程非常艰辛，两三个十几年工作经验的大牛做了一年多才稳定下来。绝大部分开发能力不足，只能望尘莫及。而且后续的调试和维护，也会花费不少精力。

但是 OpenResty 的出现改变了这一切， OpenResty 非常的 pythonic ，适合人类的正常思维。新手经过一两个月的学习，做出来的 API， 就可以达到 nginx C 模块的性能，而且代码量大大减少，也方便调试。

3. 以奇虎和新浪为例，如何在项目中引入新技术

技术选型只是第一步，如何才能在一个产品或者项目中引入 OpenResty 这个新的技术呢？我拿奇虎企业安全和新浪移动这两家公司真实发生的案例给大家看看。我和新浪移动的周晶，都是在一个有成熟产品的部门，用一两个人的力量，把一个新技术，替换掉了原有的技术架构。但由于企业产品和个人产品的不同，方法有很大的不一样。

先说我所在奇虎企业安全。我在 2012 年初加入这个部门，当时产品主打免费，目标用户是小企业。所以架构设计上面，只考虑了几十点、几百点的终端请求，使用了非常强绑定的 Windows 平台技术，而且倾向于不用开源软件，自己新做一个更适合自己的框架。包括自己用 C++ 开发的 Web server，自己写的 PHP 路由和框架，数据存储在 sqlite 里面。

我帮忙修改了两个月 PHP 的 bug，看明白了技术架构的思路之后，就去新开的一个产品线了。这是一个实验性的产品，主要面对央企和专用网，一个网络中有上百万的终端。

刚开始没有什么人关注，我就直接采用了 Linux + OpenResty + Redis + Postgres 的开源组件，性能测试甩之前的 N 条街。后面这个实验性的产品，和之前的产品，合并为一个产品，技术上面就割裂为两套架构。老功能用老架构，新功能用新架构。

随着越来越多大用户的增加，原有的技术架构开始捉襟见肘，技术债务越积压越多。随着用户的抱怨，sqlite 被抛弃，全面换成 Postgres。但对于自己开发的框架还是有些敝帚自珍。

期间通过对比测试、OpenResty 培训还有多次用户性能问题排查，让开发同学们都知道这门技术的优势。快被加班压垮的开发同学，逐渐开始选择使用 OpenResty 而不是自研的框架，来进行新功能的开发，以及旧功能的迁移，来避免加班。

在产品重构的时候，之前自研的服务端框架被完全抛弃，服务端开发的同学从 8 、9 个人减少到 3 个人。在新技术的引入过程中，我们没有采用强制的举措，因为企业产品需要稳定，用户处部署的版本更新很慢。

而新浪移动周晶的实践，对大家更有参考意义。新浪移动最开始是基于 Apache，用 PHP 来处理用户请求。Apache 是同步多进程模型，在并发请求不多的情况下没有问题。

但是总是会有突发新闻，比如马航失联、文章出轨等，突发的高流量把后台压垮了几次。而且可以预见世界杯的流量也会很大，所以周晶花几个月时间，用 nginx 替换了 Apache，使用 nginx 的 fast_cgi_cache，QPS 提升了一个数量级。

新浪移动后台的接口都是使用 PHP 来实现的，在高并发下有些力不从心。而 nginx 简单的缓存虽然能满足性能，但不能满足业务精细化和数据一致性的要求，需要找 PHP 之外的解决方案，前提是让 PHP 的开发能够舒适的使用。 node.js 的回调地狱、Go 的调试不方便，都是一个阻碍。

他们最后选择了 OpenResty，而且基于 OpenResty 开源了一个 Web 框架 Vanilla（香草），模仿了 Yaf 的使用习惯，让 PHP 的开发更容易接受和上手。 Vanilla 已经在新浪移动开始使用，一些核心业务，比如高清图和体育直播，正在向这个框架迁移中。

4. 入门痛点，以及学习的正确方法

我和周晶的入门，都是自己摸着石头过河。当时除了 Python 社区「大妈」的那篇使用文章外，找不到其他的资料。

奇虎和新浪都用 OpenResty 成功替换了之前的技术，但问题还是挺明显，就是大家都认为自己是孤独的使用者，同事中基本没有人认同。在关键和支撑业务上，使用 OpenResty 有些不放心，都会在边缘业务上先做尝试和验证。

虽然 OpenResty 的性能做的很棒，比肩或者超过其他所有的高性能解决方案，但是担心没有学习资料、担心招不到人、担心没人交流，可能还担心作者章亦春哪天撂挑子不干了，这个项目就黄了。

高可用架构群里的各位都是架构师，是技术决策者，在引入一门新技术的时候，肯定会考虑到这些风险。比如小米科技马利超在高可用架构的分享，他们在抢购系统中曾经使用过 ngx_lua，虽然性能满足需求，但是团队里面熟悉的人少，最后还是改成了 Go 语言实现。

如何解决这些担忧？ 社区是有过思考和讨论的，我们放在分享最后讲。先从一个尝试使用这门技术的开发者的角度看，OpenResty 不少基础工作没有完善，友好程度不够：

只能从源码安装，没有 apt-get、brew 等软件仓库安装方法；安装第三方库没有 PIP、NPM 之类的包管理工具，需要去先谷歌，然后拷贝代码文件到指定的目录下，才能 require 使用。

代码编写需要修改 nginx.conf 和对应的 lua 代码，即使是 hello world 也是如此。当然你可以把代码写在 nginx 的配置文件里面，但是生产环境肯定是要分离的。这种编写代码的方式，不像是一个编程语言，和常规的编程方式不同。

有独特的执行阶段概念，因为 OpenResty 是基于 nginx 的，所以也继承它的这种概念。你的代码逻辑，可能需要放在不同的阶段里面运行，才能获取你想要的预期。而这些阶段间信息如何传递，以及哪些 API 不能在某些阶段使用，就会经常拦住新手。

遇到问题只有邮件列表这一种方式来沟通，而邮件列表是被墙的。文档也只有英文版本，导致很多新手的问题无法被解决。

没有系统学习 OpenResty 的手段，大都是业务需要实现什么功能，就去文档和 API 里面去找。至于方式对不对，能不能优化，就不知道了。

而 Lua 语言自身也有一些特别的地方：

- 下标从 1 开始，这个是和其他编程语言很大的不同。
- 不区分 array 和 dict ，会导致处理 json 的时候，无法区分 array 和 object。
- 默认全局变量，需要在所有变量前加 local，忘记的话，可能导致各种难查的 bug。
- 自带的字符串正则匹配规则和通常的 PCRE 不同，使用的话，学习成本较高。
- Lua 标准库和周边库，都是阻塞的，需要自己甄别哪些可以和 OpenResty 搭配使用。新手很容易使用了阻塞的库，而导致性能急剧下降。

有没有好的入门方法？

我们团队正在做这方面的努力，尽量在现有的基础上，降低学习的门槛。 对于新手，可以看 StuQ 上面 OpenResty 的系列视频教程 (http://www.stuq.org/course/detail/1015)，我们计划有 4 季，分别是入门、进阶、实战和源码分析。现在第一季已经上线，第二季正在后期制作。看完前两季，基本上就可以在项目里面用了。

对于已经使用了 OpenResty 的开发者，我们把这两三年遇到的坑，都记录在 GitHub 的《OpenResty 最佳实践》上面（https://github.com/moonbingbing/openresty-best-practices），大家可以当做 cookbook 来使用。

5. nginScript 这样的尝试会替代 OpenResty 吗？

nginScript 是今年 nginx 大会上，Nginx 官方推出的一个新的配置语言。它是模仿了 OpenResty 的做法，把 JavaScript VM 嵌入到 nginx 中，提供简单的 nginx 配置功能。

我们看下它的 hello world：

```
location / {
    js_run "
        var res;
        res = $r.response;
        res.status = 200;
        res.send('Hello World!');
        res.finish();
    ";
}
```

再对比下 OpenResty 的 hello world：

```
location / {
    content_by_lua_block {
        ngx.say("hello world")
    }
}
```

看上去差不多，只是 OpenResty 简洁一些。根据 nginx 官方的说明，nginScript 只是想提供一种更方便配置 nginx 的方法，并不想取代 ngx_lua。

考虑到 JavaScript 本身的流行和开发社区的强大，如果未来两三年它从一个简单的 nginx 配置语言，逐渐演变成类似 ngx_lua 这样功能非常完备的开发语言，甚至替代 OpenResty 也是有可能的。

当然，这个前提是 OpenResty 停滞不前。现在 OpenResty 已经有的功能，和计划开发的功能，倾向于覆盖 nginx Plus 的功能。所以 nginx 和 OpenResty 之间，有一个良性的竞争关系，这是大家都乐意看到的。

6. 未来重点解决的问题和新增特性

短期内的目标，是想降低入门的难度：

提供官方二进制发布包。类似于 docker 的安装方法，一行命令，下载一个 sh 脚本，增加一个源地址，不用手工解决依赖，不用源码编译，直接就可以试用。

而且会发布 Windows 的二进制包，方便这个平台的开发者本机做一些测试。

增加包管理。命令行工具叫 iresty，可以从 iresty.org 上面搜索、安装需要的 lua resty 库，避免找错库或者放错目录。

写一本书《 OpenResty 编程》，这本书会成为官方的入门书籍，框架和关键内容由作者春哥直接操刀，我和社区的其他同学帮助一起完成。

做完上面 3 点，OpenResty 的入门难度会降低到和其他编程语言一样。

在功能上面，会增加很多激动人心的新特性：

- 支持 TCP 和 UDP 。Nginx 最新的 stream 子系统已经支持了 TCP，OpenResty 的 ngx_stream_lua 模块正在开发中，会拥有和现有的 nginx http modlue 相同的 lua API，所以很多应用和库，可以不加修改的运行在一个新的子系统上面。

- 更好的支持推送场景。增加 shared list 共享内存的队列，可以用于 worker 间的通讯；增加 semaphore 特性，用于 ngx_lua 轻量级线程间的通讯。酷狗音乐的推送服务就是基于这些实现的，这些改动点会在这个月并入 master。可以邀请酷狗音乐的同学，来给大家详细分享下里面的细节。

- 建立一个开源的 WAF 平台。现在阿里云和 cloudflare 的 WAF 做的都很棒，经受住了很多实际的考验。但是都没有开源，我们希望最好的 WAF 是开源的，而且是基于 OpenResty 的。

- 在 OpenResty 中增加内存数据库。可以有持久化，或者就是全内存的，支持 SQL 的查询。这个也是出于极致性能的考虑，有时候我们还是需要使用 SQL 来做一些复杂的查询，但有不想使用那么重的关系型数据库，而且数据是可以丢失的。那么这个就可以排上用场。

- 实现 PHP、Python 等方言，让 PHP、Python 等程序员可以用自己喜欢的语言写 OpenResty 的代码，底层转换为 LuaJIT 的字节码。

春哥在 OpenResty 技术大会上面说了非常多的新特性，包括 streaming RegEx 正则引擎等等，非常高端，我挑了几个我觉得有意思的做介绍。

7. 开源社区建设

OpenResty 诞生于 2011 年，大多数时间都是春哥主力在维护这个项目，当然也有很多开发者提交 feature 和 bugfix ，但基本上算单打独斗。

社区有 github 和邮件列表，大部分还是提问的。春哥每天会花费很多的时间，来详细的回答各种基础问题。

今年新增了 QQ 群和微信群， QQ 群的质量很高，每天都会有很多提问，非技术问题是被禁止的。而且还有了自己的技术大会，能给大家面对面交流的机会。

我们翻译了 ngx_lua 的英文文档，能让大家更方便的查找资料；我们搭建了一个不用翻墙就能访问的论坛： bbs.iresty.com，用作提问和知识积累的地方。后面会把谷歌邮件列表的内容同步过来。

只有上面这些是不够的，在 OpenResty 技术大会的第二天，我们召集了一个很小规模的闭门会议，决定成立 OpenResty 咨询委员会。

这个委员会，是以个人名字参加的，成员来自奇虎 360、新浪、又拍云、酷狗音乐等公司和社区的开发者，希望把国内社区的核心使用者和开发者团结在一起，促进 OpenResty 的发展。

同时，OpenResty 软件基金会也开始筹备工作，我们希望走规范的非盈利组织的模式，来保证 OpenResty 长期稳定发展。给开发者和使用者信心，敢于在关键业务上面使用 OpenResty。

Q & A

1、请问 OpenResty 的定位是什么，从分享来看似乎全栈了？

定位主要是高性能，所有的新功能和优化，都是针对性能的。 但是也有人拿来做页面，比如京东；也有人拿来替代 PHP 做 Web server，比如新浪。 我觉得它越来越像一个独立的开发语言。

2、请问 Lua 是不是可以实现动态配置 location？比如动态切流量？

balancer_by_lua 可能是你需要的，你可以用 Lua 来定义自己的负载均衡器，可以在每个请求的级别上去定义，当前访问的后端的节点地址、端口，还可以定制很细力度的访问失败之后的重试策略。

3、OpenResty 是可以拿到 nginx 请求里面的所有信息？那是不是可以做一些更复杂的转发操作？能介绍一下 OpenResty 在 cdn 里面的应用场景吗？

可以看下 iresty.com 的分享，又拍的张聪非常详细的介绍了 OpenResty 在又拍 CDN 的使用。

4、OpenResty 是否修改了 nginx 的源码，还是和 nginx 完全可剥离开的？Nginx 版本升级，OpenResty 也跟着升级吗？例如 nginx 修复漏洞 bug 等情况。

OpenResty 不修改 nginx 的源码，可以跟随 nginx 无痛升级。 如果你觉得 OpenResty 升级慢了， 你可以只拿 ngx_lua 出来，当做 nginx 的一个模块来编译。实际上，OpenResty 在测试过程中，发现了很多 nginx 自身的 bug 。

5、软 WAF nginx + Lua 是主流和未来方向么？

我觉得 WAF 应该基于 nginx，不管是性能还是流行程度。而 OpenResty 具有更灵活操控 nginx 的能力，所以我觉得 OpenResty 在 WAF 领域非常合适。cloudflare 的 WAF 就是基于 OpenResty。

6、看样子未来可能有各种 ngx_xx，最有可能的是 js，不知道这方面有什么前沿的动向？

我们组在尝试把 PHP 嵌入到 nginx 中，当然性能肯定不如 LuaJIT，但是会方便很多 PHP 同学，有进展的话，我们会开源出来 :）

7、OpenResty 目前看似乎是一个 proxy 的配置框架（糅合了 nginx + Lua），但以后的发展是什么样子？会不会以后更进一步，比如做一个 API gateway 之类的。

OpenResty 其实是希望大家忽略 nginx 的存在，直接使用 ngx_lua 提供的 API 实现自己的业务逻辑。更像一门独立的开发语言，只不过底层使用 nginx 的网络库而已。你可以按照你的想法搭建任何好玩的服务端应用出来。

实施微服务需要哪些基础框架

文章来源：
http://mp.weixin.qq.com/s?__biz=MjM5MDE0Mjc4MA==&mid=400645575&idx=1&sn=da55d75db55117046c520de88dde1123&scene=2&srcid=1109LBUa6088UHL97qS7dcnp&from=timeline&isappinstalled=0#wechat_redirect

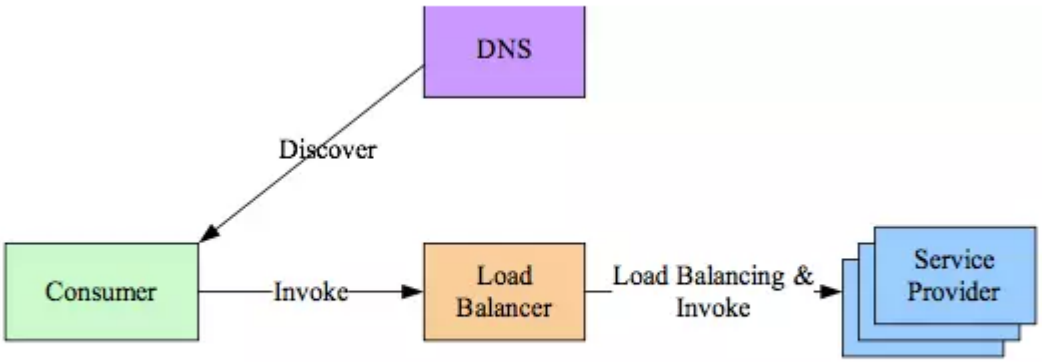
引 言

微服务(MicroServices)架构是当前互联网业界的一个技术热点，圈里有不少同行朋友当前有计划在各自公司开展微服务化体系建设，他们都有相同的疑问：一个微服务架构有哪些技术关注点(technical concerns)？需要哪些基础框架或组件来支持微服务架构？这些框架或组件该如何选型？笔者之前在两家大型互联网公司参与和主导过大型服务化体系和框架建设，同时在这块也投入了很多时间去学习和研究，有一些经验和学习心得，可以和大家一起分享。

服务注册、发现、负载均衡和健康检查

和单块(Monolithic)架构不同，微服务架构是由一系列职责单一的细粒度服务构成的分布式网状结构，服务之间通过轻量机制进行通信，这时候必然引入一个服务注册发现问题，也就是说服务提供方要注册通告服务地址，服务的调用方要能发现目标服务，同时服务提供方一般以集群方式提供服务，也就引入了负载均衡和健康检查问题。根据负载均衡 LB 所在位置的不同，目前主要的服务注册、发现和负载均衡方案有三种：

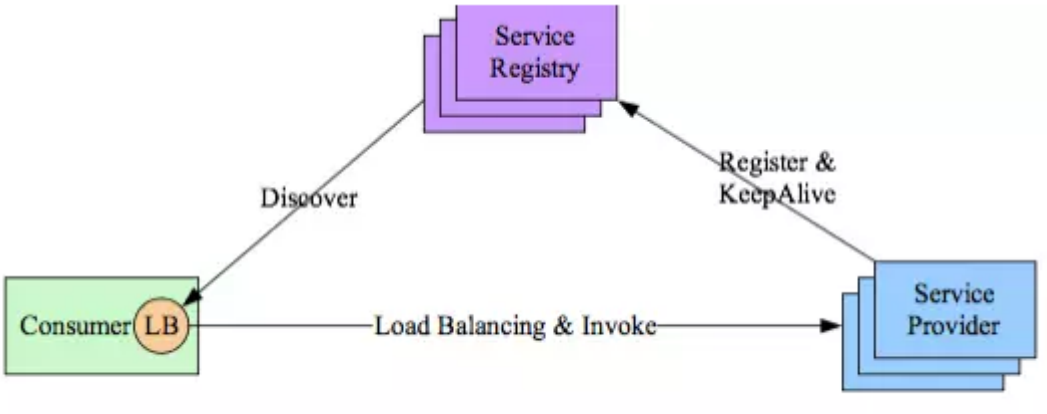
第一种是集中式 LB 方案，如下图 1，在服务消费者和服务提供者之间有一个独立的 LB，LB 通常是专门的硬件设备如 F5，或者基于软件如 LVS，HAproxy 等实现。LB 上有所有服务的地址映射表，通常由运维配置注册，当服务消费方调用某个目标服务时，它向 LB 发起请求，由 LB 以某种策略（比如 Round-Robin）做负载均衡后将请求转发到目标服务。LB 一般具备健康检查能力，能自动摘除不健康的服务实例。服务消费方如何发现 LB 呢？通常的做法是通过 DNS，运维人员为服务配置一个 DNS 域名，这个域名指向 LB。



[图1] 集中式LB方案

集中式 LB 方案实现简单，在 LB 上也容易做集中式的访问控制，这一方案目前还是业界主流。集中式 LB 的主要问题是单点问题，所有服务调用流量都经过 LB，当服务数量和调用量大的时候，LB 容易成为瓶颈，且一旦 LB 发生故障对整个系统的影响是灾难性的。另外，LB 在服务消费方和服务提供方之间增加了一跳(hop)，有一定性能开销。

第二种是进程内 LB 方案，针对集中式 LB 的不足，进程内 LB 方案将 LB 的功能以库的形式集成到服务消费方进程里头，该方案也被称为软负载(Soft Load Balancing)或者客户端负载方案，下图 Fig 2 展示了这种方案的工作原理。这一方案需要一个服务注册表(Service Registry)配合支持服务自注册和自发现，服务提供方启动时，首先将服务地址注册到服务注册表（同时定期报心跳到服务注册表以表明服务的存活状态，相当于健康检查），服务消费方要访问某个服务时，它通过内置的 LB 组件向服务注册表查询（同时缓存并定期刷新）目标服务地址列表，然后以某种负载均衡策略选择一个目标服务地址，最后向目标服务发起请求。这一方案对服务注册表的可用性(Availability)要求很高，一般采用能满足高可用分布式一致的组件（例如 Zookeeper, Consul, Etcd 等）来实现。

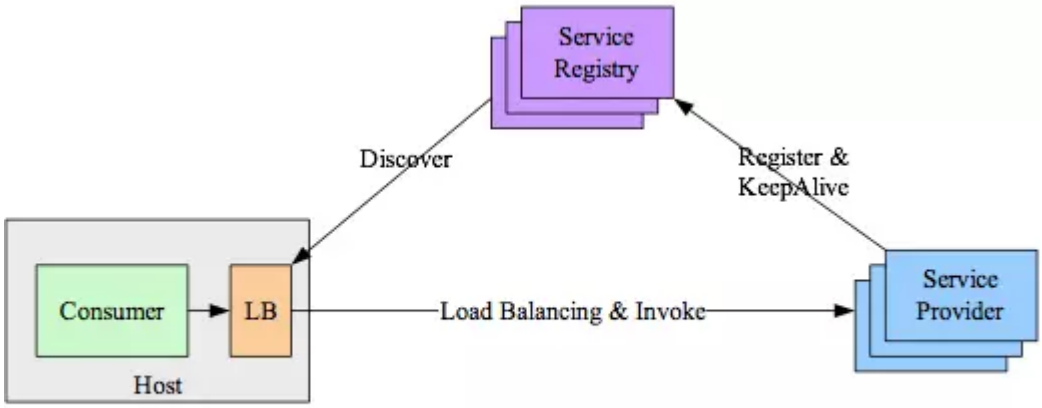


[图2] 进程内LB方案

进程内 LB 方案是一种分布式方案，LB 和服务发现能力被分散到每一个服务消费者的进程内部，同时服务消费方和服务提供方之间是直接调用，没有额外开销，性能比较好。但是，该方案以客户库(Client Library)的方式集成到服务调用方进程里头，如果企业内有多种不同的语言栈，就要配合开发多种不同的客户端，有一定的研发和维护成本。另外，一旦客户端跟随服务调用方发布到生产环境中，后续如果要对客户库进行升级，势必要求服务调用方修改代码并重新发布，所以该方案的升级推广有不小的阻力。

进程内 LB 的案例是 Netflix 的开源服务框架，对应的组件分别是：Eureka 服务注册表，Karyon 服务端框架支持服务自注册和健康检查，Ribbon 客户端框架支持服务自发现和软路由。另外，阿里开源的服务框架 Dubbo 也是采用类似机制。

第三种是主机独立 LB 进程方案，该方案是针对第二种方案的不足而提出的一种折中方案，原理和第二种方案基本类似，不同之处是，他将 LB 和服务发现功能从进程内移出来，变成主机上的一个独立进程，主机上的一个或者多个服务要访问目标服务时，他们都通过同一主机上的独立 LB 进程做服务发现和负载均衡，见下图 Fig 3。



[图3] 主机独立LB进程方案

该方案也是一种分布式方案，没有单点问题，一个 LB 进程挂了只影响该主机上的服务调用方，服务调用方和 LB 之间是进程内调用，性能好，同时，该方案还简化了服务调用方，不需要为不同语言开发客户库，LB 的升级不需要服务调用方改代码。该方案的不足是部署较复杂，环节多，出错调试排查问题不方便。

该方案的典型案例是 Airbnb 的 SmartStack 服务发现框架，对应组件分别是：Zookeeper 作为服务注册表，Nerve 独立进程负责服务注册和健康检查，Synapse/HAproxy 独立进程负责服务发现和负载均衡。Google 最新推出的基于容器的 PaaS 平台 Kubernetes，其内部服务发现采用类似的机制。

服务前端路由

微服务除了内部相互之间调用和通信之外，最终要以某种方式暴露出去，才能让外界系统（例如客户的浏览器、移动设备等等）访问到，这就涉及服务的前端路由，对应的组件是服务网关(Service Gateway)，见图 4，网关是连接企业内部和外部系统的一道门，有如下关键作用：

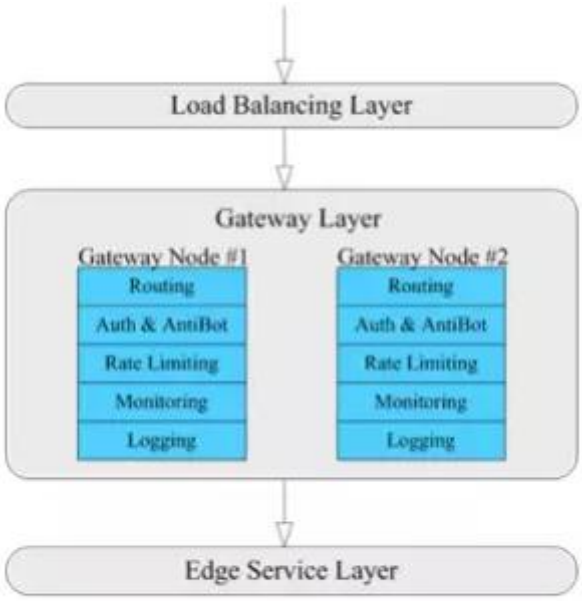
- 服务反向路由，网关要负责将外部请求反向路由到内部具体的微服务，这样虽然企业内部是复杂的分布式微服务结构，但是外部系统从网关上看到的就像是一个统一的完整服务，网关屏蔽了后台服务的复杂性，同时也屏蔽了后台服务的升级和变化。

- 安全认证和防爬虫，所有外部请求必须经过网关，网关可以集中对访问进行安全控制，比如用户认证和授权，同时还可以分析访问模式实现防爬虫功能，网关是连接企业内外系统的安全之门。

- 限流和容错，在流量高峰期，网关可以限制流量，保护后台系统不被大流量冲垮，在内部系统出现故障时，网关可以集中做容错，保持外部良好的用户体验。

- 监控，网关可以集中监控访问量，调用延迟，错误计数和访问模式，为后端的性能优化或者扩容提供数据支持。

- 日志，网关可以收集所有的访问日志，进入后台系统做进一步分析。

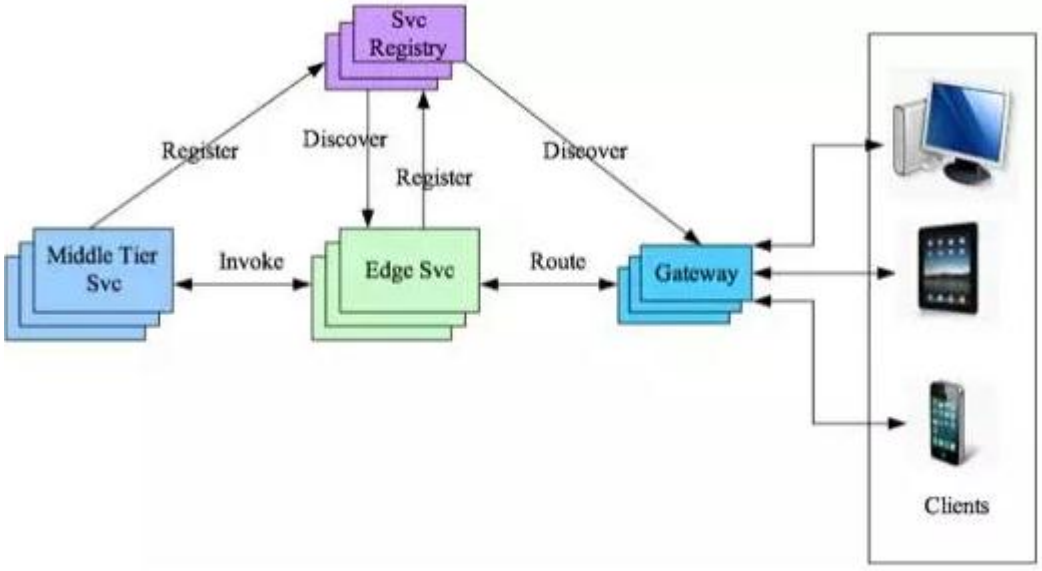


[图4] 服务网关

除以上基本能力外，网关还可以实现线上引流，线上压测，线上调试(Surgical debugging)，金丝雀测试(Canary Testing)，数据中心双活(Active-Active HA)等高级功能。

网关通常工作在 7 层，有一定的计算逻辑，一般以集群方式部署，前置 LB 进行负载均衡。

开源的网关组件有 Netflix 的 Zuul，特点是动态可热部署的过滤器(filter)机制，其它如 HAproxy，Nginx 等都可以扩展作为网关使用。

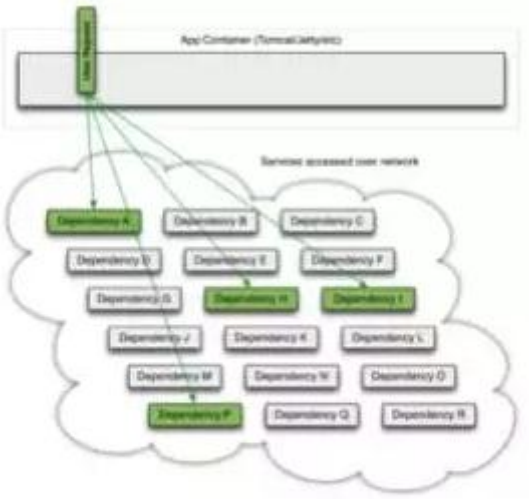


[图5] 简化的微服务架构图

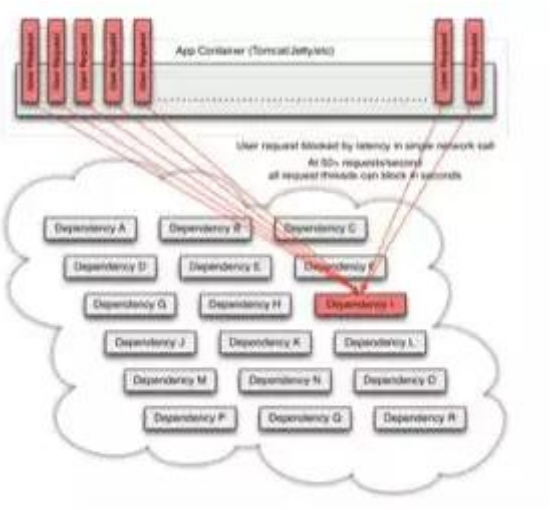
在介绍过服务注册表和网关等组件之后，我们可以通过一个简化的微服务架构图(Fig 5)来更加直观地展示整个微服务体系内的服务注册发现和路由机制，该图假定采用进程内 LB 服务发现和负载均衡机制。在下图 Fig 5 的微服务架构中，服务简化为两层，后端通用服务（也称中间层服务 Middle Tier Service）和前端服务（也称边缘服务 Edge Service，前端服务的作用是对后端服务做必要的聚合和裁剪后暴露给外部不同的设备，如 PC，Pad 或者 Phone）。后端服务启动时会将地址信息注册到服务注册表，前端服务通过查询服务注册表就可以发现然后调用后端服务；前端服务启动时也会将地址信息注册到服务注册表，这样网关通过查询服务注册表就可以将请求路由到目标前端服务，这样整个微服务体系的服务自注册自发现和软路由就通过服务注册表和网关串联起来了。如果以面向对象设计模式的视角来看，网关类似 Proxy 代理或者 Façade 门面模式，而服务注册表和服务自注册自发现类似 IoC 依赖注入模式，微服务可以理解为基于网关代理和注册表 IoC 构建的分布式系统。

服务容错

当企业微服务化以后，服务之间会有错综复杂的依赖关系，例如，一个前端请求一般会依赖于多个后端服务，技术上称为 1 -> N 扇出(见图 Fig 6)。在实际生产环境中，服务往往不是百分百可靠，服务可能会出错或者产生延迟，如果一个应用不能对其依赖的故障进行容错和隔离，那么该应用本身就处在被拖垮的风险中。在一个高流量的网站中，某个单一后端一旦发生延迟，可能在数秒内导致所有应用资源(线程，队列等)被耗尽，造成所谓的雪崩效应(Cascading Failure，见图 7)，严重时可致整个网站瘫痪。



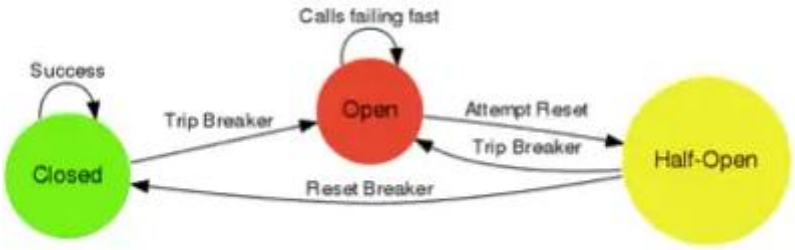
[图6] 服务依赖



[图7] 高峰期单个服务延迟致雪崩效应

经过多年的探索和实践，业界在分布式服务容错一块探索出了一套有效的容错模式和最佳实践，主要包括：

- 电路熔断器模式(Circuit Breaker Patten)，该模式的原理类似于家里的电路熔断器，如果家里的电路发生短路，熔断器能够主动熔断电路，以避免灾难性损失。在分布式系统中应用电路熔断器模式后，当目标服务慢或者大量超时，调用方能够主动熔断，以防止服务被进一步拖垮；如果情况又好转了，电路又能自动恢复，这就是所谓的弹性容错，系统有自恢复能力。下图 Fig 8 是一个典型的具备弹性恢复能力的电路保护器状态图，正常状态下，电路处于关闭状态(Closed)，如果调用持续出错或者超时，电路被打开进入熔断状态(Open)，后续一段时间内的所有调用都会被拒绝(Fail Fast)，一段时间以后，保护器会尝试进入半熔断状态(Half-Open)，允许少量请求进来尝试，如果调用仍然失败，则回到熔断状态，如果调用成功，则回到电路闭合状态。



[图8] 弹性电路保护状态图

- 舱壁隔离模式(Bulkhead Isolation Pattern)，顾名思义，该模式像舱壁一样对资源或失败单元进行隔离，如果一个船舱破了进水，只损失一个船舱，其它船舱可以不受影响 。线程隔离(Thread Isolation)就是舱壁隔离模式的一个例子，假定一个应用程序 A 调用了 Svc1/Svc2/Svc3 三个服务，且部署 A 的容器一共有 120 个工作线程，采用线程隔离机制，可以给对 Svc1/Svc2/Svc3 的调用各分配 40 个线程，当 Svc2 慢了，给 Svc2 分配的 40 个线程因慢而阻塞并最终耗尽，线程隔离可以保证给 Svc1/Svc3 分配的 80 个线程可以不受

影响，如果没有这种隔离机制，当 Svc2 慢的时候，120 个工作线程会很快全部被对 Svc2 的调用吃光，整个应用程序会全部慢下来。

- 限流(Rate Limiting/Load Shedder)，服务总有容量限制，没有限流机制的服务很容易在突发流量(秒杀，双十一)时被冲垮。限流通常指对服务限定并发访问量，比如单位时间只允许 100 个并发调用，对超过这个限制的请求要拒绝并回退。

- 回退(fallback)，在熔断或者限流发生的时候，应用程序的后续处理逻辑是什么？回退是系统的弹性恢复能力，常见的处理策略有，直接抛出异常，也称快速失败(Fail Fast)，也可以返回空值或缺省值，还可以返回备份数据，如果主服务熔断了，可以从备份服务获取数据。

Netflix 将上述容错模式和最佳实践集成到一个称为 Hystrix 的开源组件中，凡是需要容错的依赖点(服务，缓存，数据库访问等)，开发人员只需要将调用封装在 Hystrix Command 里头，则相关调用就自动置于 Hystrix 的弹性容错保护之下。Hystrix 组件已经在 Netflix 经过多年运维验证，是 Netflix 微服务平台稳定性和弹性的基石，正逐渐被社区接受为标准容错组件。

服务框架

微服务化以后，为了让业务开发人员专注于业务逻辑实现，避免冗余和重复劳动，规范研发提升效率，必然要将一些公共关注点推到框架层面。服务框架(Fig 9)主要封装公共关注点逻辑，包括：

[图9] 服务框架

- 服务注册、发现、负载均衡和健康检查，假定采用进程内 LB 方案，那么服务自注册一般统一做在服务器端框架中，健康检查逻辑由具体业务服务定制，框架层提供调用健康检查逻辑的机制，服务发现和负载均衡则集成在服务客户端框架中。

- 监控日志，框架一方面要记录重要的框架层日志、metrics 和调用链数据，还要将日志、metrics 等接口暴露出来，让业务层能根据需要记录业务日志数据。在运行环境中，所有日志数据一般集中落地到企业后台日志系统，做进一步分析和处理。

- REST/RPC 和序列化，框架层要支持将业务逻辑以 HTTP/REST 或者 RPC 方式暴露出来，HTTP/REST 是当前主流 API 暴露方式，在性能要求高的场合则可采用 Binary/RPC 方式。针对当前多样化的设备类型(浏览器、普通 PC、无线设备等)，框架层要支持可定制的序列化机制，例如，对浏览器，框架支持输出 Ajax 友好的 JSON 消息格式，而对无线设备上的 Native App，框架支持输出性能高的 Binary 消息格式。

- 配置，除了支持普通配置文件方式的配置，框架层还可集成动态运行时配置，能够在运行时针对不同环境动态调整服务的参数和配置。

- 限流和容错，框架集成限流容错组件，能够在运行时自动限流和容错，保护服务，如果进一步和动态配置相结合，还可以实现动态限流和熔断。

- 管理接口，框架集成管理接口，一方面可以在线查看框架和服务内部状态，同时还可以动态调整内部状态，对调试、监控和管理能提供快速反馈。Spring Boot 微框架的 Actuator 模块就是一个强大的管理接口。

- 统一错误处理，对于框架层和服务的内部异常，如果框架层能够统一处理并记录日志，对服务监控和快速问题定位有很大帮助。

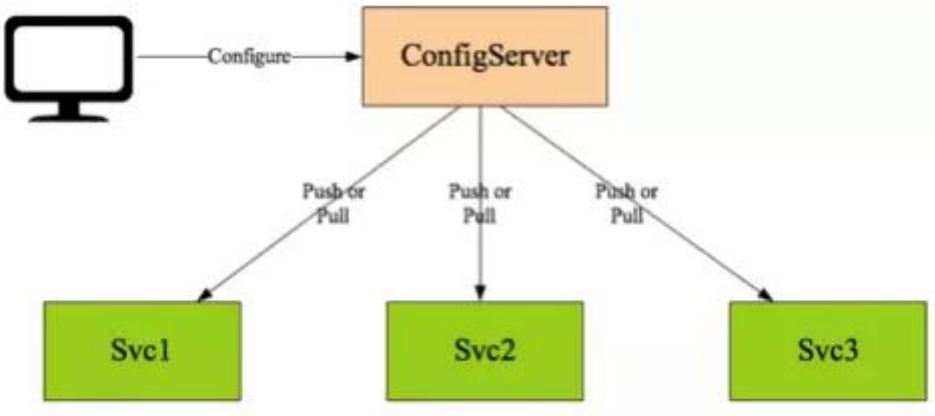
- 安全，安全和访问控制逻辑可以在框架层统一进行封装，可做成插件形式，具体业务服务根据需要加载相关安全插件。

- 文档自动生成，文档的书写和同步一直是一个痛点，框架层如果能支持文档的自动生成和同步，会给使用 API 的开发和测试人员带来极大便利。Swagger 是一种流行 Restful API 的文档方案。

当前业界比较成熟的微服务框架有 Netflix 的 Karyon/Ribbon，Spring 的 Spring Boot/Cloud，阿里的 Dubbo 等。

运行期配置管理

服务一般有很多依赖配置，例如访问数据库有连接字符串配置，连接池大小和连接超时配置，这些配置在不同环境(开发/测试/生产)一般不同，比如生产环境需要配连接池，而开发测试环境可能不配，另外有些参数配置在运行期可能还要动态调整，例如，运行时根据流量状况动态调整限流和熔断阀值。目前比较常见的做法是搭建一个运行时配置中心支持微服务的动态配置，简化架构如下图：



[图10] 服务配置中心

动态配置存放在集中的配置服务器上，用户通过管理界面配置和调整服务配置，具体服务通过定期拉(Scheduled Pull)的方式或者服务器推(Server-side Push)的方式更新动态配置，拉方式比较可靠，但会有延迟同时有无效网络开销(假设配置不常更新)，服务器推方式能及时更新配置，但是实现较复杂，一般在服务和配置服务器之间要建立长连接。配置中心还要解决配置的版本控制和审计问题，对于大规模服务化环境，配置中心还要考虑分布式和高可用问题。

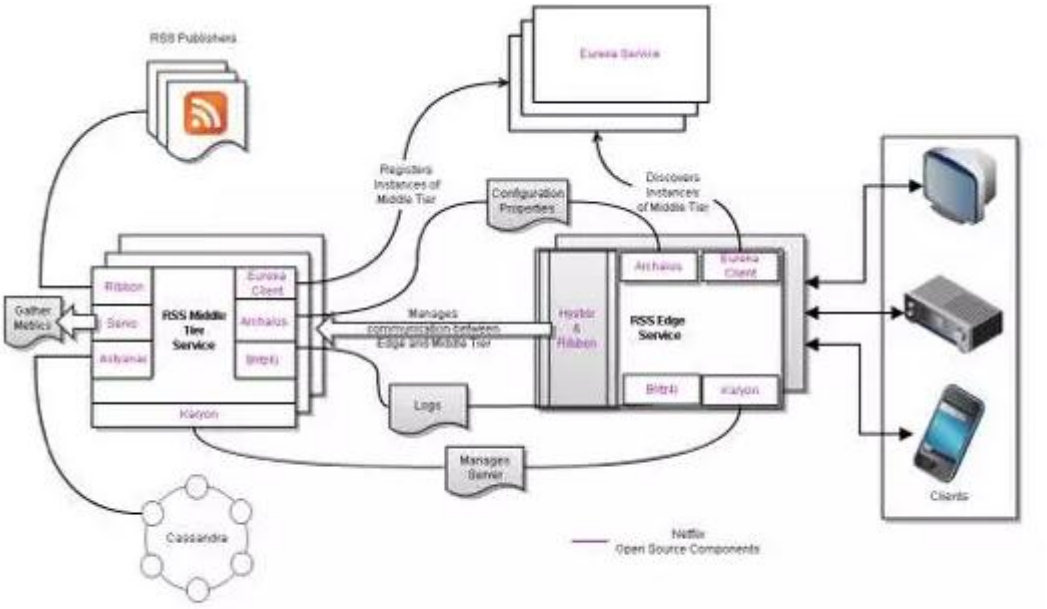
配置中心比较成熟的开源方案有百度的 Disconf，360 的 QConf，Spring 的 Cloud Config 和阿里的 Diamond 等。

Netflix 的微服务框架

Netflix 是一家成功实践微服务架构的互联网公司，几年前，Netflix 就把它的几乎整个微服务框架栈开源贡献给了社区，这些框架和组件包括：

- Eureka: 服务注册发现框架
- Zuul: 服务网关
- Karyon: 服务端框架
- Ribbon: 客户端框架
- Hystrix: 服务容错组件
- Archaius: 服务配置组件
- Servo: Metrics 组件
- Blitz4j: 日志组件

下图 11 展示了基于这些组件构建的一个微服务框架体系，来自 recipes-rss。



[图11] 基于Netflix开源组件的微服务框架

Netflix 的开源框架组件已经在 Netflix 的大规模分布式微服务环境中经过多年的生产实战验证，正逐步被社区接受为构造微服务框架的标准组件。Pivotal 去年推出的 Spring Cloud 开源产品，主要是基于对 Netflix 开源组件的进一步封装，方便 Spring 开发人员构建微服务基础框架。对于一些打算构建微服务框架体系的公司来说，充分利用或参考借鉴 Netflix 的开源微服务组件(或 Spring Cloud)，在此基础上进行必要的企业定制，无疑是通向微服务架构的捷径。