**KS Nginx线上故障洗冤录**

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 发布日期 | 修订内容 | 主要修订人 |
| v 0.1 | 2014-04-02 | 创建版本。  包含：  rewrite规则；  break和last的区别；  location寻址；  导致的线上经典案例分析和总结。 | 王伟 |
| v 0.2 | 2015-01-12 | 新增如下内容：  （1）bfe对304请求的支持不完美导致na集群304请求访问失败案例；  （2）nginx版本升级导致php的304响应失效案例。 | 王伟 |
| v 0.3 | 2015-02-28 | 新增如下内容：  （1）bfe不支持向域名post数据案例；  （2）dna模块bug导致nginx频繁出core | 王伟 |

目 录

前言 1

1 从简单的break说起 2

1.1 知识储备 2

1.2 break之变形一 2

1.3 break之变形2 4

1.4 break之变形3 4

2 到break的亲戚last 7

2.1 nginx中内部路由—源码面前了无秘密 7

2.2 last标记导致的路由死循环 8

2.3 何时使用last 9

3 再到url的简洁之道 10

3.1 何谓简洁url 10

3.2 简洁url的优点 10

3.3 简洁URL都具备的性质 10

3.4 bad url 11

3.5 nginx中的rewrite简洁之道 11

4 静态文件类型显示异常 13

4.1 事故现场还原 13

4.2 nginx的location匹配规则 14

4.3 原因分析 15

4.4 如何解决 15

4.5 总结 16

5 一个回车引发的血案之bfe对304的支持问题分析 17

5.1 引蛇出洞 17

5.2 迷雾重重，楼高不见章台路 17

5.3 旭日东升，掀开层层雾 18

5.4 冲破云霄，终⻅见光明 19

5.5 最终的圆满 20

5.6 总结 20

6 nginx升级引发的na集群304缓存失效 21

6.1 问题描述 21

6.2 问题追查 21

6.3 总结 23

7 bfe不支持向域名post数据案例 24

7.1 案例描述 24

7.2 问题追查 24

7.3 深入分析 24

7.4 总结 25

8 简单的snprintf函数导致的线上core 27

8.1 问题描述 27

8.2 问题定位 27

8.3 深入代码 28

8.4 总结 29

9 再话http的408状态码—Webserver中的DOS攻击 30

9.1 发现408 30

9.2 了解408 30

9.2.1 http协议中对408的解释 30

9.2.2 http协议报文 31

9.2.3 webserver中得DOS攻击 32

9.3 认识nginx对408的处理 33

9.4 测试408 33

9.5 408进阶 33

9.6 总结 33

前言

目前，nginx已经在知识社区甚至公司内外内广泛使用，在过去的使用中，我们出现了各种各样的问题。总结在过去的问题，我们认为这主要是大家对nginx的各种规则了解不足导致的。

目前大家使用最多的是nginx的路由(rewrite)功能，本文对围绕着nginx对过去线上出现的和nginx相关的问题进行了归纳和总结，通过案例、深入代码，由浅入深的为大家讲解nginx的运行原理，目的就是希望通过实际的案例提高工程师对nginx的了解，帮助工程师写出优高效的nginx配置，提高nginx的性能以及nginx配置的可维护性。

# 1 从简单的break说起

这个案例可以说是我碰到过的关于break用法的的一个经典案例。这个案例的各种变形完美的诠释了break的应用。

## 知识储备

在NGINX中，break指令会以两种形态存在：

* 在rewrite规则中作为flag标记用；
* 作为单独的配置指令用。

官方文档对break指令的解释如下所示：

Stops processing the current set of ngx\_http\_rewrite\_module directives. If a directive is specified inside the location, further processing of the request continues in this location.

--www.nginx.org

也就是说如果在配置中遇到了break指令，则NGINX会停止处理该break指令之后的ngx\_http\_rewrite\_module模块的指令集。但是对于该location中的其它非ngx\_http\_rewrite\_module模块的指令，NGINX会继续往下处理。

那么ngx\_http\_rewrite\_module模块的指令集都有哪写呢？从源码中，我们发现，该模块中最重要的指令就是有：rewrite, if, return。

## break之变形一

案例描述：NGINX流控库测试中发现开启流控库功能后，如果命中小流量（$rule\_all\_or变量的值为1）所有的请求都会失败。为了搞清这个问题的始末，我们先来看一下当时的NGINX的配置，如图 1所示。

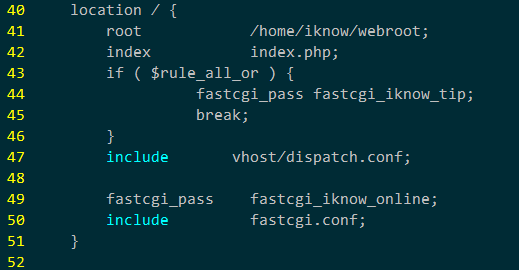


图 1‑1 开启流控功能后的NGINX配置

图 1中的配置文件中，引入了vhost/dispatch.conf文件，这个文件中包含了所有的url路由规则，摘取部分规则，如所示。

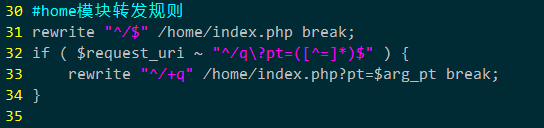


图 1‑2 dispatch.conf中的rewrite规则

根据1.1节中的介绍，你有发现问题吗？

实际上NGINX的处理流程如下：

STEP1）图 1的第43行中，$rule\_all\_or的值为1，因此if指令判断条件成立，此时nginx会把请求转发给fastcgi\_iknow\_tip指定的php。

STEP2）接下来的第45行中出现了一个break，根据1.1中的介绍可以知道，该break语句之后的所有的rewrite指令都不会执行。

STEP3）虽然接下来的47行，引入了url的路由转发规则，但是由于45行的break指令的缘故，所有的url都不会去重写。

STEP4）因此，所有的命中小流量的url都会因为没有进行url重写而访问失败。

现在怎么解决？最容易想到的办法就是把图 1中第47行的语句提到if 指令之前。如果你也是这么想的，那么我们就会得到1.3节中看到的配置。

## break之变形2

对图 1的配置稍作修改，则得到如图 3所示的配置。

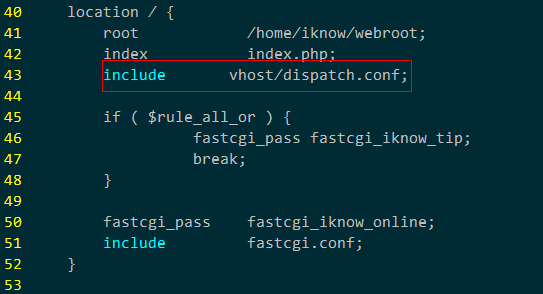


图 1‑3 break之变形2

如果现在$rule\_all\_or = 1的话，又会出现什么情况呢？我们来分析一下这份配置时如何处理的：

STEP1）首先在43行，引入了url重写规则的配置文件dispatch.conf，此时会对请求的url进行重写。值得注意的是，如图 2所示，每条rewrite规则都会有一个break标记，此处的break和break的用法是一致的。会结束该break之后所有的ngx\_http\_rewrite\_module模块的指令。

STEP2）if指令属于ngx\_http\_rewrite\_module模块的指令。因此，由于STEP1）的原因，该指令并不会执行；

STEP3）继续执行第50行的fastcgi\_pass指令，把请求转发到fastcgi\_iknow\_online指定的php机器。因此，无论是否命中小流量，流控功能都不会生效。

## break之变形3

OK，至此，已经讲到了break的概念在nginx中的两个经典案例。相信大家对break的理解更深了一层。那么回到我们这一章最先的问题：如何使流控可以生效呢？

这里涉及到nginx对请求的处理流程，简化版的流程如下图所示：

图 1‑4 nginx对请求的处理阶段

这里面关于rewrite会涉及到三个重要的阶段：server\_rewrite阶段，find\_config阶段，rewrite阶段，关于这三个阶段，下面做详细说明。

**（1）server\_rewrite阶段**

SERVER\_REWRITE阶段是nginx中第一个必须经历的重要phase，请求进入此阶段时已经找到对应的虚拟主机（server）配置（关于如何找到虚拟主机，后面再详细介绍）。nginx的rewrite模块在这个阶段注册了一个handler，可以把一些前缀满足特定模式的uri重定向到一个固定的url，还可以根据请求的属性来决定是否需要重写或者给用户发送特定的返回码。

该阶段和后面的REWRITE阶段作用大不相同：在SERVER\_REWRITE阶段中，请求还未被匹配到一个具体的location中。该阶段执行的结果（比如改写后的uri）会影响后面FIND\_CONFIG阶段的执行。

**（2）find\_config阶段**

FIND\_CONFIG阶段顾名思义就是寻找配置阶段，具体一点就是根据uri查找location配置，找到location配置后，nginx则进入该location去执行该location中相应的配置。最后，由于有REWRITE阶段的存在，FIND\_CONFIG阶段可能会被执行多次。

**（3）rewrite阶段**

REWRITE阶段为location级别的重写，这个阶段的checker和SERVER\_REWRITE阶段的是同一个函数，而且Nginx的rewrite模块对这2个阶段注册的是同一个handler，2者唯一区别就是执行时机不一样，REWRITE阶段为location级别的重写，SERVER\_REWRITE执行之后是FIND\_CONFIG阶段，REWRITE阶段执行之后是POST\_REWRITE阶段。

在上述基础之上，我们做如下分析：

问题的原因：location 中的fastcgi选择的if语句必须是放置在location中的，但是这就会导致location阶段的rewrite失效；

解决方案：把location级别的rewrite提前到server级别的rewrite，这样就不会导致rewrite规则和流控库选择fastcgi的判断冲突。因为server级别的rewrite重写之后的uri都还是会进入到location /{} 中，因此，并不会对线上的服务造成影响。

# 2 到break的亲戚last

借用刀郎的一句歌词来形容last的地位：如果那天你不知道我踩了多少坑，你就不会知道last有多么复杂。

官方文档对last指令的解释如下所示：

last: stops processing the current set of ngx\_http\_rewrite\_module directives and starts a search for a new location matching the changed URI.

--www.nginx.org

由此可知，last提供给nginx一个从rewrite阶段调回到find\_config阶段的机会。这种设计在某些时候会给webserver的运维人员带来很大的便利。但是利弊相随，如果掌握不好，也会带来极大的不利影响。

nginx中rewrite的last标示，类似于C/C++语言中的goto语句，其利弊也类似于goto语句。

last给了nginx一个从跳出当前location域，进入另一个location域的机会，但是随着带来的弊端有如下3点：

（1）跳来跳去，降低nginx的性能；

（2）容易导致路由死循环；

（3）升级维护代价太大，因为单看一条rewrite规则，无法知道某个url最终路由到什么地方；

## nginx中内部路由—源码面前了无秘密

在ngx\_http\_request.h文件中的ngx\_http\_request\_s结构体中存在一个接口体成员：uri\_changes，该成员表示一个http请求在nginx内部的rewrite次数。

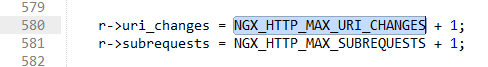


可见uri\_changes是一个4位表示的整数，也就是最大可表示到16。

在ngx\_http\_request.h中，还定义了一个常量，NGX\_HTTP\_MAX\_URI\_CHANGES，用来限制内部路由的次数。从而避免某个请求在nginx内部陷入死循环。

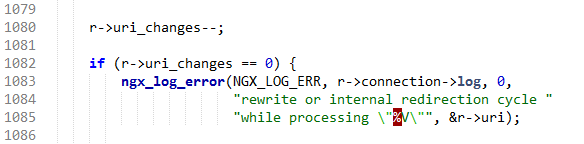


在ngx\_http\_request.c文件的ngx\_http\_create\_request(){}函数中，nginx对uri\_changes变量进行了初始化。



接下来我们想一下，什么时候r->uri\_changes会变化呢？大致有如下几种情况：

在nginx的rewrite post 阶段绑定的（ngx\_http\_core\_post\_rewrite\_phase）函数中，r->uri\_changes会做如下操作：

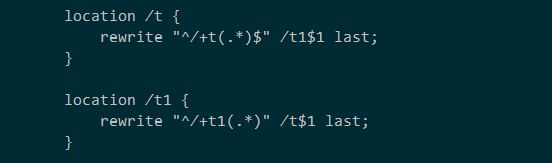


在rewrite post阶段，last字段会使得某个请求有机会跳出该location，从而重新从find config阶段开始进入一个新的location，这也使得ngx\_http\_core\_post\_rewrite\_phase函数有机会执行多次，从而使得r->uri\_changes的值降到0。这从某种程度上讲，也是nginx为了避免rewrite循环带来的服务器性能的空耗而进行的优化。从另一种程度上讲，也是last表示的弊端。

如果大家想看到更有趣的事情，那么可以对ngx\_http\_core\_post\_rewrite\_phase函数做一个小小的改动，把，第1080行的r->uri\_changes--;改成r->uri\_changes= NGX\_HTTP\_MAX\_URI\_CHANGES，从而屏蔽nginx对路由循环的控制。如此，一个存在路由循环的请求就会一直处于uri重写阶段的循环中，直到超时。

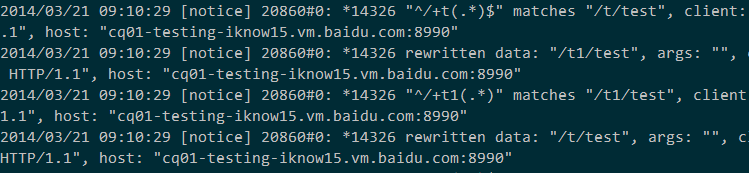
## last标记导致的路由死循环

空说无凭，实例为证。为了更好的让大家了解last的机制，看一下如下的配置：



现在，一眼就能看出来，/t和t1之间存在一个循环。因为例子中的业务关系并不复杂，还能一眼看出这种环的存在。但是，对于业务复杂，动则上百条的路由规则的情形下，隐含于茫茫路由规则之中的环很可能会被忽略，从而导致问题。

现在访问一下<http://cq01-testing-iknow15.vm.baidu.com:8990/t/test>，看下nginx的日志，会发现如下的情形：



## 何时使用last

何时使用last和c语言中何时使用goto语句有着相同的原则，但是却比goto的使用更棘手，更麻烦。更好的使用条件是：除非必须使用last，否则一律使用break。

# 3 再到url的简洁之道

## 何谓简洁url

Clean URL: Clean URLs are purely structural URLs that do not contain a query string [e.g., action=delete&id=91] and instead contain only the path of the resource (after the scheme [e.g., http] and the authority [e.g., example.org]). This is often done for aesthetic, usability, or search engine optimization (SEO) purposes. Other reasons for designing a clean URL structure for a website or web service include ensuring that individual web resources remain under the same URL for years, which makes the World Wide Web a more stable and useful system, and to make them memorable, logical, easy to type, human-centric, and long-lived.

--www.wikipedia.org

从如上的描述可以看到，所谓的简洁url即url中不存在参数的url。

## 简洁url的优点

(1)美观。如果说上帝按照自己的样子创造了人类，那么工程师按照自己的灵魂创造了代码。而这份代码总归会有一个名字，对于互联网产品而言，这个名字就是URL。取名是一种艺术，是阅历，思维，见识的较量。美观的url更容易让人记住，让人赏心悦目，流连忘返。

(2)易用，便于维护、升级。简洁的url能够做到见名知意。另外，简洁url好比好的路标，无论是哪个司机，都能很容易的到达目的地。

看看如下的url，小伙伴们还记得吗？能知道是什么东西吗？

^/+q\?(.\*)?(ct=17&(.\*)?tn=ikfsjson|tn=ikfsjson&(.\*)?ct=17)(&.\*)?$

(3)方便搜索优化。

## 简洁URL都具备的性质

(1)简洁的URL中的每个部分都很清晰，URI部分是一个没有请求参数的类似文件系统的层级结构。

(2)独立于webserver依赖的环境（php，jsp，asp…），隐藏webserver的详细信息（比如index.php）。一方面，这些信息和用户无关，另一方面，如果应用升级到其它环境，则url也必须做升级。

(3)不会经常发生改变。

## bad url

接下给给大家列举几个bad url的例子。

<http://zhidao.baidu.com/q?ct=24&cm=18&tn=uiframework&un=tdliling&t=1357278183721#info>

<http://example.com/index.php?page=consulting/marketing>

<http://example.com/index.php?mod=profiles&id=193>

## nginx中的rewrite简洁之道

在简洁URL之后，需要考虑的另一个问题就是在nginx中rewrite规则的简洁之道啦。

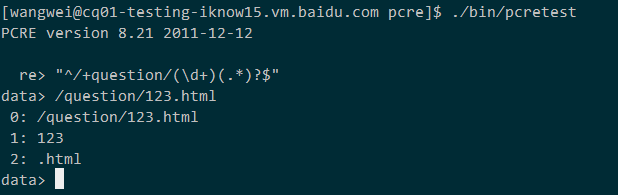
首先看一下nginx中rewrite指令的用法：rewrite regex replacement [flag]

对如上的规则做如下说明：

rewrite的regex参数仅是用来匹配请求中的uri部分的（路径部分），请求中的参数部分是不会参与匹配过程的。

nginx中的rewrite规则按照其在配置文件中出现的先后顺序，依次执行，直到遇到某个标识停止处理。

在rewrite指令中给，会大量的用到正则表达式捕获组的概念，nginx中的正则表达式使用的是pcre库。因此，如果要测试rewrite规则，可以使用pcre自带的pcretest工具，具体使用如下图所示。



如果请求中的url的参数部分，确实包含特定含义，则单独使用rewrite指定无法完成转发的功能。此时需要使用if指令并配合rewrite规则进行使用。例如：<http://zhidao.baidu.com/q?ct=24&cm=18&tn=uiframework&un=tdliling&t=1357278183721#info>，可以配置为：

if ($request\_uri ~\* “/q(.\*)ct=24”) {

rewrite “..” “…” break;

}

如下的转发规则则是错误的：

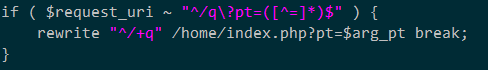
rewrite ^/bbs/board.php?bo=([a-zA-Z0-9]+)&wr=([0-9]+)$ /show-$1-$2.html break;

请求中的参数部分，会原封不动的追加到跳转之后的replacement之后。因此，在编写rewrite规则的时候，如果不需要对请求的参数部分作特殊处理，是不需要显示配置请求参数的。

为了避免原请求中的参数追加到新请求之后，可以在replacement的最后增加?以做标示。如：

rewrite ^/users/(.\*)$ /show?user=$1? break;

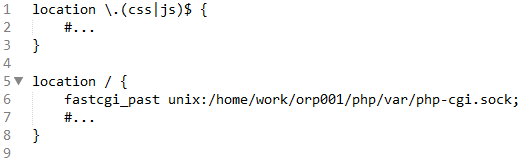
使用$arg\_XXX变量获取原请求中的请求参数值。例如：



# 4 静态文件类型显示异常

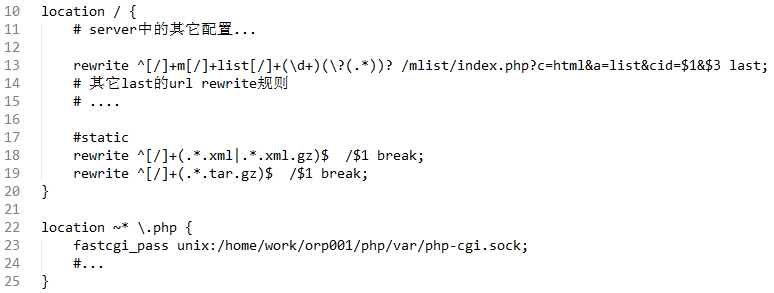
## 事故现场还原

（1）在一次nginx的配置升级中，忽然增加了如下的配置：

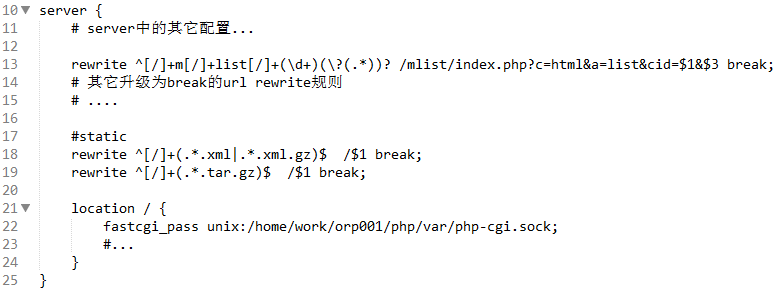


从而导致服务器对所有静态文件的请求的相应类型变为text/html类型。

（2）还有，经验在一次nginx的rewrite规则升级中，把如下的规则：



中的last全部升级为break后，得到如下的配置



从而导致队友xml静态文件的请求，服务器的相应类型全部变为text/html。

不知道其它产品线是否有遇到过这种相应类型错误的情况？那么原因究竟是什么呢？在真正分析原因之前，先讲解一下nginx是如何处理location指令的。

## nginx的location匹配规则

nginx在确定一个请求由哪个server去处理之后，接下来就是要选择处理该请求的location域。那么如何对location进行选择呢？一般会遵循如下的规则：

nginx中的location配置有如下的形式：

1.精确匹配的路径， location = /XX {}

2.抢占式前缀匹配的路径，location ^~ /XX {}

3.普通前缀匹配的路径， location /XX {}

4.正则表达式路径匹配，location ~\* regex {}

nginx在路径(location)搜索时，正则匹配路径和其他的路径分开搜。

搜索时，首先去搜索前缀字符串定义的location，并且以最长的前缀匹配作为匹配结果。例如如果一个请求满足如下的两个location：

A: location / {}

B: location /test {}

则会选择B作为匹配的结果。

在前缀匹配中，如果匹配到location = /XX {}类型的精确匹配，则nginx会停止继续匹配并且使用该location，例如：

A: location = /user {}

B:location / {}

对于/user/test这样的请求，nginx在匹配A之后会停止location的搜索。

这里要注意如上的规则和下面规则的不同：

A: location /user {}

B: location / {}

这种规则，nginx会搜索B路径，只是因为A路径为最长前缀匹配，才选择了A路径而已。

前缀路径搜索完毕之后，nginx会记录最长前缀匹配路径。

继续按出现顺序搜索正则表达式指定的location路径，如果能够匹配，则停止搜索，并且使用该匹配的路径。

以正则表达式方式出现的location规则，必须满足 location ~\* regex {} 或 location ~ regex {}。

如果正则路径匹配没有成功，则使用之前记录的最长前缀匹配。

## 原因分析

在4.2节的基础之上，我们来分析4.1节中出现的事故的具体原因。

可以发现，4.1节第一个例子中，静态文件的loation配置 locatin \.(css|js) {}的配置不符合locatin配置规则6。

(css|js)为一个正则表达式，但是该配置有没有~\* 或 ~修饰符，因此导致nginx永远无法选择该locaiton去处理css，js文件。

最终导致nginx选择了location / {}去处理静态文件，而该location中，包含一个fastcgi\_pass指令，从而把对静态文件的请求交个php模块处理。而php影响的内容，响应类型均为text/html，从而导致静态文件的类型变成text/html类型，导致静态文件失效。

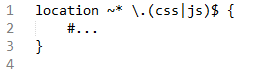
关于4.1节第二个例子，比较复杂。升级前之所以可以正长，是因为升级前，location /{} 中的所有php文件都因为last标志的原因，重新跳转到location ~\* \.php {}中去。而xml静态文件因为break的原因保留在location / {} 中运行。因此升级前运行正常。

但是升级之后，rewrite规则提升到了server级别，并且只存在一个location / {} 域，导致所有的请求都会进入该location并转交给php处理。从而导致对于xml文件的请求，形影格式变为text/html。

## 如何解决

根据如上的分析：

4.1节中的问题1，可以改为如下配置，进行修复。



4.1节中的问题2，修改为如下配置。



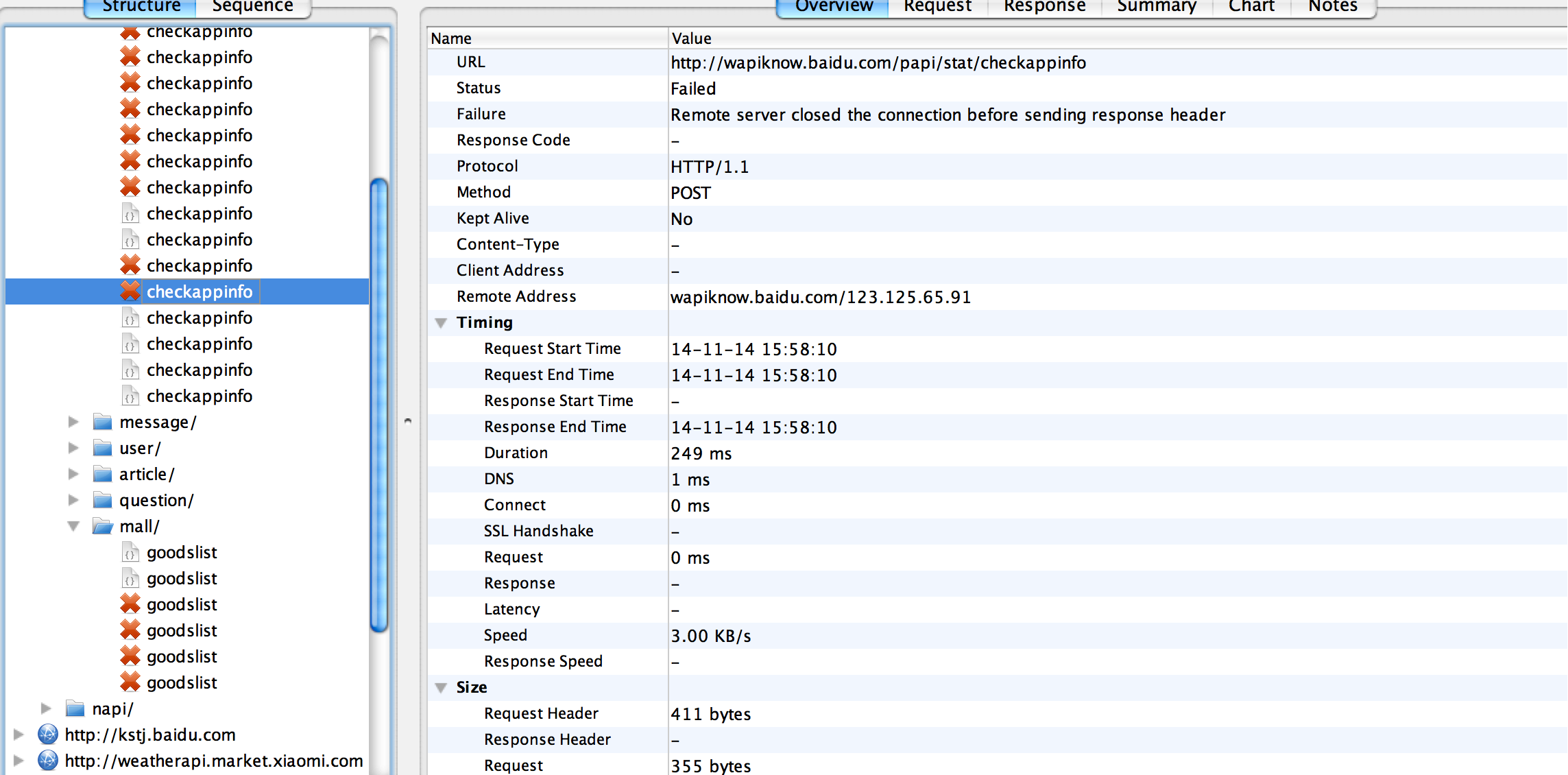
## 总结

对于静态和php文件同机器部署的产品线而言，在进行路由规则配置时，需要确保静态文件的处理不能转发给php模块去处理。

# 5 一个回车引发的血案之bfe对304的支持问题分析

## 引蛇出洞

宝宝知道4.0的server端上线后，发现版本检查，金币列表等接口会时而获取不到数据。开启代理工具进行抓包分析，抓包结果如下所示：



于此同时，PM反馈用户存在无法登录的情况。经分析发现如抓包所示的情况属于同一个问题。diff出问题的几个接口发现新代码没有什么改动，并且直接访问单机请求功能正常，因此断定不是上线引起的问题。但是考虑到用户体验问题, 和RD代亮决定先回滚代码，然后再进行排查。

代码回滚之后，问题得到解决。这个现象引出了我们要讨论的问题。

## 迷雾重重，楼高不见章台路

为了方便描述，我把现象整理成一系列的问题，如下所示：

1. 如果是代码的问题，为什么ios正常，android就异常？

2. 如果是代码的问题，为什么访问单机正常，域名访问就异常？

3. 如果是代码的问题，为什么线下正常，线上就异常？

4. 如果是代码的问题，为什么访问异常的时候，模块的php日志显示是正常的，并且没有报任何错误？

5. 如果是代码的问题，为什么访问异常的时候，nginx,nginx\_trans的日志返回304是正常的？

6. 如果这种情况和客户端相关，为什么用浏览器进行访问也会出现相同的问题？

7. 如果不是代码的问题,为什么代码回滚后就回复正常了？

追查无果时，在tips机器上进行试验，发现了又一个疑点:  对nginx返回的status code为200的请求处理正常，对status code为 304的处理异常，抓包工具会提示：远程服务器断开连接。

面对如上的谜团，真是庭院深深深几许，楼高不见章台路。 但是真相永远只有一个,如果直接看第7点，那么肯定是代码出问题了。但是第2点貌似又和第7点有些矛盾。第8点的存在，让我想起了bfe对100-continue 这个http header不支持引发的惨案，开始怀疑,是否是bfe有升级导致的？但是这个原因很快就被排除，因为作业帮和知道app的请求中的status code为304的请求没有出现这种情况。因此问题肯定和上线的代码有关系，但是具体的原因是什么无法定位。

## 旭日东升，掀开层层雾

既然是只有nginx的status code为304的时候会出问题，那把缓存逻辑去掉之后是否会正常？RD修改代码，去掉代码中的缓存逻辑后，接口请求正常。但是去掉缓存逻辑会导致用户流量的增加，因此去掉缓存逻辑虽然可以解决问题，但不是上上策略。

然而，缓存逻辑是没有改动的，为什么缓存逻辑会出问题？并且如果真是缓存逻辑出问题，则单机访问也会出问题？而实际中，单机访问时，对304的响应是正常的。

此时，RD@代亮在存在混存逻辑代码的同时增加了ob\_clean()函数,  此时发现域名访问已经正常。该函数的功能之一为:防止在浏览器有输出之后再使用setcookie()或者header()造成的错误。那么到此可以认定，肯定是在浏览器有输出之后调用了header函数引起的。但是具体是什么原因导致的呢?

为了确认这个猜测，利用curl对增加ob\_clean()前后的接口进行访问，发现增加前，结果如下图所示：



而增加后，结果如下图所示：



值得关注的是增加ob\_clean()最大的不同就是图中红色标注的地方，也就是那个空行。就是这个空行引起了域名访问时出现的问题。

## 冲破云霄，终⻅见光明

接下来，对所有的代码文件进行排查，发现有一个文件存在如下图所示的内容：



从上图可以明显的看到，该文件在<?php之前多了⼀一个空行，而php- cgi处理之后，这个空行就被当做html的空行输出到output buffer。而这个缓冲区的内容会通过nginx一直往上返，直到bfe的webserver。当bfe的webserver收到这种响应后，发现一个已经输出 内容的请求的http响应状态码为304，此时bfe的webserver就断开了这个请求，因此也就出现了最开始的图中出现的：远程服务器已经断开连接的错误。

至于bfe是如何断开的，无从得知。但是从现实证据可以推断，bfe在收到响应码为304，并且有内容输出的响应时，确实会断开该请求连接。如果想证明这个问题，还得了解我们用户请求整个过程遇到的层层关卡，为了证明推断结果，画了一个简图，如下所示：



针对如上图的描述，对出问题的请求进行的日志分析，请求出问题时，php，odp\_webserver，nginx\_trans的响应都正常，而之所以出现远程服务器断开连接的情况，则只有在bfe层断开连接，否则nginx\_trans或者之下的任何一层肯定会有日志体现。综合日志和请求流程，断定是bfe对于这种特殊的请求不予支持。并且在试验中，单机访问正常也对这一论断进行了佐证。

因此，最总得出结论：bfe不支持有响应数据输出并且响应状态码为304的http的响应请求。bfe一旦收到此种响应，会立即断开这次访问的请求。

## 最终的圆满

到此，我们来回顾下bfe的某些特性对我们的影响：

作为请求接受方，bfe不支持Expect为100-continue的http请求首部，导致的直接向域名(zhidao.baidu.com/ wapiknow.baidu.com)post数据时，nginx\_trans出现的499的响应。

作为响应的接收方，bfe不支持响应状态码为304时存在输出数据, 导致的断开客户端请求连接，进而导致用户请求无响应的情况。

针对如上两点，如果把bfe比作一个两面通⾏行的门，那么从入口方向和出口方向都埋下了坑，也算是一种对称，一种圆满了。其实翻翻HTTP协议的详细文档会发现，HTTP协议文档中，对于出现这些情况时，webserver应该以何种行为进行响应做了明确的说明，但是......

最后，也是最重要的，经试验作业帮也会存在这种问题。可以推断，任何导致出现这些行为的情况都会影响我们服务的稳定性。并且这种情况线下无法复现，因此其它端、甚至PC端的RD后续也要引起注意。

## 总结

1. 如果要向域名post数据，请一定关闭100-continue的Expect首部，具体方法：

curl\_setopt($ch, CURLOPT\_HTTPHEADER, array('Expect:')); 具体原因：在使用基于curl为基础的库进行POST时, 当POST的数据大于1024Byte或者post的数据为数组的时候，curl并不会直接就发起POST请求, 而是会分为两步:

1)发送一个请求，包含一个Expect:100-continue，询问Server是否愿意接受数据？

2)接收到Server返回的100-continue应答以后，客户端才把数据真正POST给Server。

2. 尽量保证<?php置于文件的第一行，至于原因，不再多做解释，这是实践的教训。

# 6 nginx升级引发的na集群304缓存失效

## 问题描述

在测试母婴android4.2的时候发现，本来server端应该返回304的请求返回了200。这个问题虽然对功能没有影响，但是却会默默的耗费用户很多流量。以母婴4.2的日刊为例，每进入一次日刊专栏，则会耗费120KB的流量，用户刷个10几次，1MB的流量就没有了。这个在2G，3G网络环境下还是很惊人的。

## 问题追查

查看线上的日志，发现线上还是存在304的请求的，大概单机每小时在2000左右。这个引起了注意，并且感到非常奇怪：为什么当前的请求该返回304的时候返回了200？为什么线上的请求又存在304的情况？如果是代码的问题，那么应该没有304的响应才对。

接下来对缓存的代码进行了分析，发现代码层面是没有任何问题的。具体的代码如下所示。



线上的问题表现为：对于header(‘ETag: ’ . $etag);这句话本来应该向响应头中增加Etag: “….”的实体标签，但是实际上客户端收到的响应中丢失了该标签，导致客户端不能正确的以IF-NONE-MATCH请求头进行请求，今儿导致缓存失效。

为了分析问题出现的原因，在qas环境上进行试验，发现qas环境中可以正常的进行304的响应。此时开始怀疑是否是线上webserver进行了修改。

去线上查看nginx发现nginx在周四下午4点有改动，查看nginx的版本（./nginx -V）发现线上nginx的版本已经升级到1.4.4。此时开始怀疑和nginx的版本升级有关系。把线上的nginx版本拉到线下发现响应头中的确丢失了ETag。在网上查到有文章介绍这个情况：http://forum.nginx.org/read.php?29,233052,233241，然后查看nginx的开发历史发现nginx从1.3.3版本开始支持etag，而之前线上版本是1.2.4，恰好不支持etag。而升级到1.4.4之后，nginx开始支持etag。然后对比了1.2.4和1.4.4的gzip请求头过滤模块发现1.4.4版本之后，如果开了gzip压缩，则nginx会清除响应头中的etag字段。具体的代码diff如下所示：

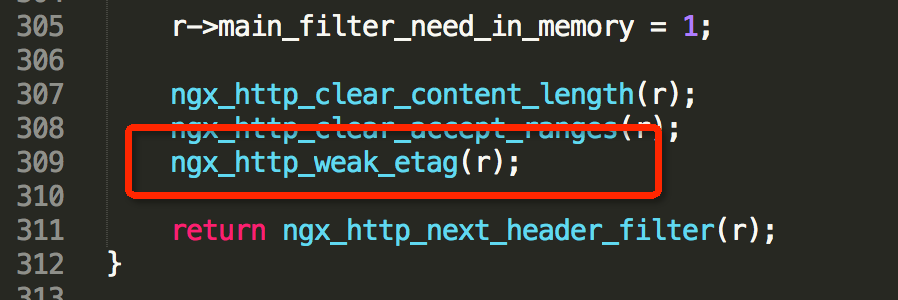


我们来看下这个宏的作用是什么，具体如下所示：



可以看到，如果响应头中存在etag响应实体的话，则ngx\\_http\\_gzip\\_filter\\_module会把etag实体清空。

然后查看了nginx社区最新的1.7.7版本的nginx，发现情况又有一些变化，具体如下图所示：



也就是说，社区已经认识到对于开启gzip进行压缩的请求，暴力的清除etag而拒绝使用缓存有些不妥。因此在最新的nginx版本中，开始使用weak etag来取代直接清除etag的做法。

## 总结

到此，该线上bug的来龙去脉已经清楚了。关于etag，weak etag，strong etag的这些概念有不懂的，可以查看http1.1协议，协议中解释的已经很清楚了（[http://tools.ietf.org/html/rfc2616#section-14.42]）。

发现该问题后，与基础知道RD负责任联系，确认修复方案（采用patch的方式，对1.4.4中的源码进行升级，对于gzip的请求，忽略对etag的清除处理）。

# 7 bfe不支持向域名post数据案例

## 案例描述

PGC项目接入第三方之后，第三方（寻医问药）研发人员反馈访问pgc接口有网络超时现象。

## 问题追查

开始以为是接口的问题，但是排查pgc对外接口的日志信息时，并没有发现有处理错误的情况。

然后想到可能是网络耗时长导致的，但是第三方设置的超时时间为20s，即便再差的网络也不会有20s的请求的。并且和知道OP以及第三方的OP一起查看网络情况，也没有发现网络层面有问题。

追查nginx\_trans的日志发现对addreply接口的访问有499的状态码，继而在odp\_webserver追查499的请求，发现499的请求没有转发到odp的nginx。

在问题即将定位为网络异常的时候，对线上近两天的nginx\_trans线上日志进行了排查，根据第三方接口请求的分布和相应情况，感觉可能不是网络耗时导致的，但是没有找到证据。

仔细观察nginx\_trans的499的请求，发现请求在nginx\_trans中的日志中的响应时间为2s，基本上排除了网络耗时的情况。

## 深入分析

为什么请求在nginx\_trans等待了2s，都没有转发给odp？是否是因为bfe不支持100-continue导致的？

然后，从这一点出发，使用如下命令访问pgc的接口，在nginx\_trans层出现了499的情况。

curl --data '{"questionid":"1234","content":"this is a answer content for test","thirth\_id":"3","thirth\_uid":"1010101"}' --header 'Expect:100-continue' --max-time 20 --write-out '%{time\_total} %{time\_connect}' '<http://zhidao.baidu.com/prof/api/addreply>'

使用了测试中写的第三方桩：cq01-testing-iknow15.vm.baidu.com:8080/testpgc/api/addquestion进行复现；发现接口提交的post数据超过1024B时，会出现499的响应；

上述的原因就是所有基于libcurl实现的curl命令/函数，在发送的post数据（字符串）超过1024B的时候，会先发一条包含Except:100-continue的请求首部的请求，服务器响应该请求后客户端才开始发送请求的请求实体；然而访问zhidao.baidu.com域时，请求会先到bfe，bfe目前不支持对Expect:100-continue的首部，也没有把该首部转发给下游的nginx\_trans，导致nginx\_trans在等待用户的请求实体的提交，而用户由于没有收到bfe的对Expect:100-continue首部的响应，也在等待。双方僵持到客户端超时自动断开连接，此时就出现了nginx\_trans的499的响应。由于nginx\_trans认为第一次请求不是一个完整的请求，因此没有把请求转发给下游的odp\_webserver，于是出现了排查中出现的现象。

## 总结

问题到此就结束了吗？

curl在post数据时，什么情况下会自动追加Expect:100-continue呢？

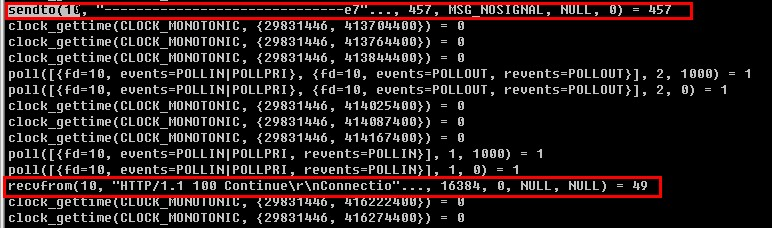
关于这个100状态码，请参考：[http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html#sec8.2.3](http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html%23sec8.2.3)。

网上目前只有一种说法：post数据超过1024B，但是这种说法不能说错，但却也不是正确的。

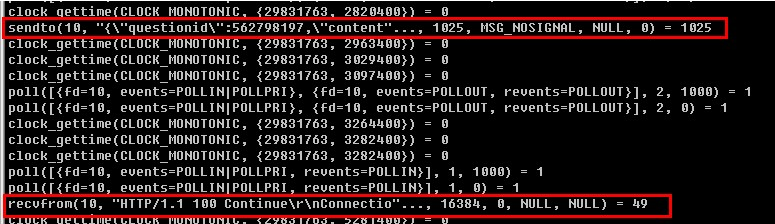
查看php文档对curl\_setopt($ch, CURLOPT\_POSTFIELDS, $data)的说明发现：$data字段有两种类型：数组，字符串；

因此对两种类型的数据分别进行了测试，在测试过程中利用strace命令跟踪php-cgi的系统调用发现curl在post的数据时的如下结论。

（1）如果$data的类型为数组，curl会自动先发送一个包含Expect:100-continue首部的请求（和数组长度无关），如下图所示：



（2）如果$data的类型为字符串（使用json\_encode,http\_build\_query等编码函数对数组编码后的字符串，其它字符串），只有当$data的长度超过1024B时，curl才会自动先发送一个包含Expect:100-continue首部的请求，如下图所示：



# 8 简单的snprintf函数导致的线上core

## 问题描述

?device=%28%23context[\%22xwork.MethodAccessor.denyMethodExecution\%22]=new%20java.lang.Boolean%28false%29%2C%23\_memberAccess[\%22allowStaticMethodAccess\%22]=new%20java.lang.Boolean%28true%29%2C%23\_memberAccess.excludeProperties={}%2C%23a\_str=%27814F60BD-F6DF-4227-%27%2C%23b\_str=%2786F5-8D9FBF26A2EB%27%2C%23a\_resp=%40org.apache.struts2.ServletActionContext%40getResponse%28%29%2C%23a\_resp.getWriter%28%29.println%28%23a\_str%2B%23b\_str%29%2C%23a\_resp.getWriter%28%29.flush%28%29%2C%23a\_resp.getWriter%28%29.close%28%29%29%28meh%29&z[%28[FOO]%29%28%27meh%27%29]=true

如上的请求导致web集群的nginx出core。

作为接入层的nginx，每秒流量在万级别。一旦web集群的nginx出core，将会严重影响用户的访问。

## 问题定位

之前curl请求http服务，当url超过256字符时也遇到过出core。虽然http协议没有规定url的长度，但是一般而言也不建议超过256B。因此，首先怀疑是不是这个地方的问题。然后拿官网原声的nginx做测试，发现原生的nginx并没有出core。排除了这种可能性。

gdb打印堆栈信息，发现出core的地方位于生成logid的模块，因此开始怀疑是不是这个模块有内存泄露，但是最终发现该模块似乎是正常的。

然后开始怀疑是否是有特殊字符，然后对请求的参数做了各种尝试，最终发现只有包含device参数并且device的值足够长时才会触发出core。

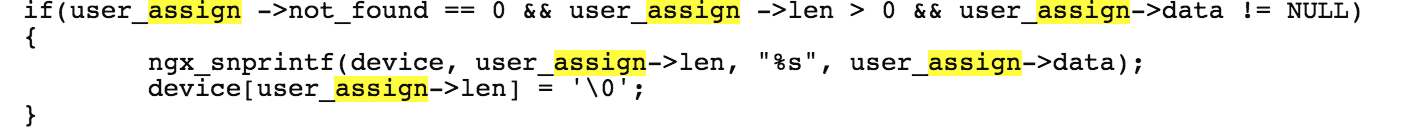
同时，对生成logid的模块代码进行了检查，并且增加日志重新编译了nginx，发现该模块确实没有问题。之所以core打在该模块，是因为该模块中需要从nginx的内存池中申请内存，在申请的时候，内存池已经坏掉了。

考虑到只有url中存在device参数时才会出core，因此，决定从device参数入手。查阅资料发现，dna扩展会使用device参数判断用户访问的终端类型。因此，决定把dna模块去掉。

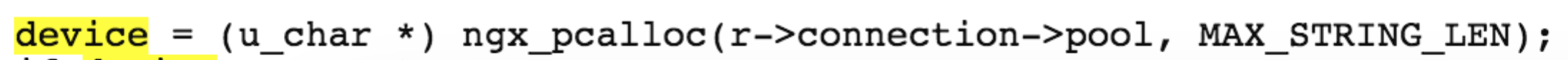
接下来，编译线上版本的nginx，并且把dna功能关闭，发现nginx已经不再出core。基本定位是由于dna模块导致的出core。

## 深入代码

问题已经定位，就是dna模块导致nginx出core，但是究竟是什么原因导致的呢？源码面前了无秘密，接下来开始阅读dna模块的代码，终于发现了如下的bug（https://svn.baidu.com/public/tags/nginx/nginx\_1-0-19-0\_PD\_BL/modules/dna-nginx-module/src/ngx\_dna\_module.cpp）：

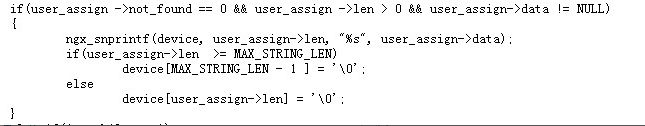


可以看到，第3行对device变量进行了内存拷贝，而device是什么东东呢？我们网上看，发现：



而MAX\_STRING\_LEN为一个宏定义：#define MAX\_STRING\_LEN 64，因此问题就是处在对device的内存拷贝上。显然snprintf函数在此处进行了误用，并没有起到snprint函数应有的作用。把用户传递的字符长度作为拷贝的长度，必然会收到用户数据的影响。

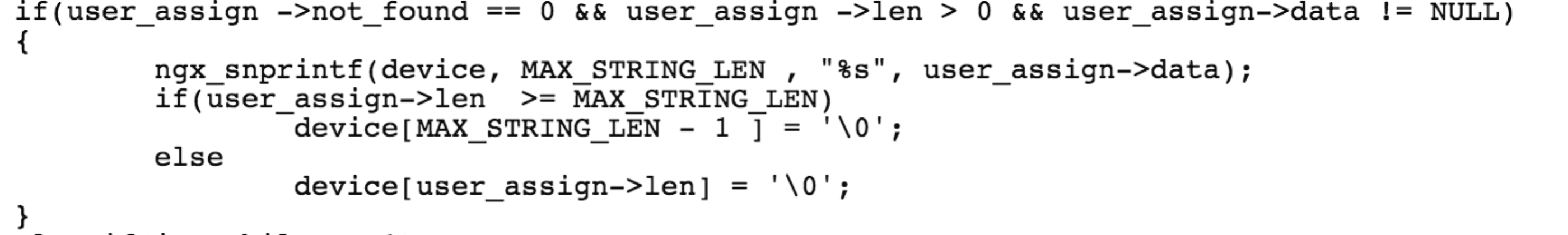
戏剧性的一幕接着出现了，上报该bug后，相关RD进行了初次修复，拿到代码后，发现修复结果如下：



从修复结果页能看出，其实RD对于snprintf函数以及内存拷贝并没有深入了解。If判断位于内存赋值之后，虽然增加了字符串的结束符，但是内存已经被破坏，因此还是会导致出core。

继续反馈bug，并给出解释，最终，得到了如下的修复：

（<https://svn.baidu.com/public/tags/nginx/nginx_1-0-20-2_PD_BL/modules/dna-nginx-module/src/ngx_dna_module.cpp>）



至此，问题得以解决

## 总结

优秀的代码，优秀的框架那都是自然进化的结果。

像如这种错误，很简单，但是追查起来是很麻烦的，尤其是在nginx中，内存泄露追查的时候异常复杂，不单纯是用gdb调试一下就能查出来的。千里之堤，毁于蚁穴。

# 9 再话http的408状态码—Webserver中的DOS攻击

## 发现408

在和某厂某部的朋友聊天时，忽然就聊到了nginx在何时会以408的情况进行响应。该问题源自朋友发现，他们的线上服务中nginx的响应码呈如下的分布：

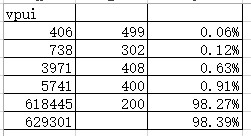


图 9‑1 某服务的http状态统计图

通过如图 9‑1的统计可知，存在0.63%的请求以408的状态响应。

由于自己之前也未曾发现过有408的响应，因此本着求真务实的原则，又捧起了http协议（http://tools.ietf.org/html/rfc2616#section-10.4.9）阅读起来。

## 了解408

### http协议中对408的解释

HTTP协议对408的解释如下：

10.4.9 408 Request Timeout

The client did not produce a request within the time that the server was prepared to wait. The client MAY repeat the request without modifications at any later time.

*-- http://tools.ietf.org/html/rfc2616#section-10.4.9*

说的直白一点就是如下的意思：

如果客户端没能够在webserver可忍受的等待时间内完成某个请求，则webserver需要返回408响应状态码。对于408的请求，客户端可以在稍后的任意时间重新发送之前的请求。

那么什么叫做在webserver可忍受的时间范围内完成请求？要回答这个问题，首先需要理解http协议的流程。因此，接下来我先介绍一下http协议对请求的规范。或许你天天在使用http协议，但是却从未认真去了解它。借此机会，也来深入了解一下它吧，这是我们解决&定位任何nginx问题的基础。

### http协议报文

HTTP报文实在HTTP应用程序之间发送的数据块，HTTP中使用流入，流出（上游、下游）来描述事物处理的方向。请求报文和响应报文都会像水流一样向下游流动。

HTTP报文是简单的格式化数据块，每条报文或者是一条来自客户端的请求，或者是一条来自服务器的响应。所有的报文都由3个部分组成：

1. 对报文进行描述的起始行；
2. 包含属性的首部；
3. 包含数据的主体；

具体如图 9‑2所示。



图 9‑2 HTTP请求报文图

那么对于webserver而言，如何处理用户请求呢？一般而言，会遵循如下的步骤：

1）连：接受和客户端的链接，创建一条tcp链路；

2）收：在该tcp连接上接受数据，然后对流中的数据做如下解析：

2.1）解析请求行：从头读取数据，直到遇到CRLF序列，作为请求行的结束；

2.2）读取以CRLF结尾的数据作为报文首部；

2.3）读取标识首部结束的CRLF，结束首部；

2.4）读取主体；

3）做：处理请求

4）发：发送响应数据

5）记：记录相关日志

### webserver中得DOS攻击

根据9.2.2中基础知识，我们可以得出结论：在webserver处理客户端请求的第二步出现超时问题，会导致webserver产生408响应。

那么什么情况下会出现超时呢？

超时一般会有两种原因：

* 物理网络原因：网速慢；

一般而言，用户请求的数据也就处于几KB，目前而言，即便最慢的2G网络也可以在1S内完成请求的发送。但是当用户的请求数据比较大时：比如cookie太大，请求首部数据太多，再加上网络环境不好，很可能会导致408的出现。本案例中得408即是因此而发生。

如果讲到这里就结束，那就没有必要写到本书中了。实际上，晚上对于408的介绍大多止步于此。但是，我们接下来，将重点关注导致408的第二种原因。能够通过一点点东西认识到事物的全貌以及本质，做到从点到线到面到体，也不失是件乐事。

* 数据原因：构造异常请求报文，导致webserver处于等待状态。

因为http报文的特殊性，我们极容易构造一些非法但是webserver又能处理的请求数据。比如：在发送如下的报文数据后，会出现什么情况？

GET / HTTP/1.1\r\n Host: www.baidu.com\r\n

由于webserver在解析请求首部时，遇到\r\n才认为请求首部已经结束。当出现如上所示的请求时，webserver会认为该请求的数据还没有发送完毕，处于等待状态。对于一个单线程阻塞型的webserver并且没有做任何超时处理的webserver而言，情况会相当糟糕。因为，这意味着该webserver不会处理之后的所有的请求。

说道这里，大家是不是想起了什么？是不是想到了N年之前的DOS攻击？对啦，对于webserver而言，这种情况就是webserver中得DOS攻击。

对于nginx而言，如果我们有足够多的肉机（多到和nginx中配置的worker\_connections\* worker\_processes）时，我们就足以让该站点爆发短暂的DOS攻击。

## 认识nginx对408的处理

## 测试408

## 408进阶

## 总结