**导航**

博客园
首 页
新随笔
联 系
订 阅 XML
管 理

< 2018年6月 >

| 日 | 一 | 二 | 三 | 四 | 五 | 六 |
|---|---|---|---|---|---|---|
| 27 | 28 | 29 | 30 | 31 | 1 | 2 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |



**搜索**

找找看
谷歌搜索

**常用链接**

我的随笔
我的评论
我的参与
最新评论
我的标签

**最新随笔**

1. PHP学习
2. Go 开发
3. Go 协程编程感悟
4. beego 遇到的一些问题
5. Fiddler 502问题
6. SourceTree
7. Trait
8. PHP PSR 标准
9. 解决MySQL联表时出现字符集不一样
10. Git 代码管理命令

**随笔分类**

Apache(19)
Bootstrap(1)
C++(2)
CentOS(2)
CSS(7)
Docker(1)
Eclipse(8)
Golang(3)
Highcharts(2)

# Nginx工作原理和优化

1. Nginx的模块与工作原理

Nginx由内核和模块组成，其中，内核的设计非常微小和简洁，完成的工作也非常简单，仅仅通过查找配置文件将客户端请求映射到一个location block（location是Nginx配置中的一个指令，用于URL匹配），而在这个location中所配置的每个指令将会启动不同的模块去完成相应的工作。

Nginx的模块从结构上分为核心模块、基础模块和第三方模块：

核心模块：HTTP模块、EVENT模块和MAIL模块

基础模块：HTTP Access模块、HTTP FastCGI模块、HTTP Proxy模块和HTTP Rewrite模块，

第三方模块：HTTP Upstream Request Hash模块、Notice模块和HTTP Access Key模块。

用户根据自己的需要开发的模块都属于第三方模块。正是有了这么多模块的支撑，Nginx的功能才会如此强大。

Nginx的模块从功能上分为如下三类。

Handlers（处理器模块）。此类模块直接处理请求，并进行输出内容和修改headers信息等操作。Handlers处理器模块一般只能有一个。

Filters （过滤器模块）。此类模块主要对其他处理器模块输出的内容进行修改操作，最后由Nginx输出。

Proxies （代理类模块）。此类模块是Nginx的HTTP Upstream之类的模块，这些模块主要与后端一些服务比如FastCGI等进行交互，实现服务代理和负载均衡等功能。

图1-1展示了Nginx模块常规的HTTP请求和响应的过程。

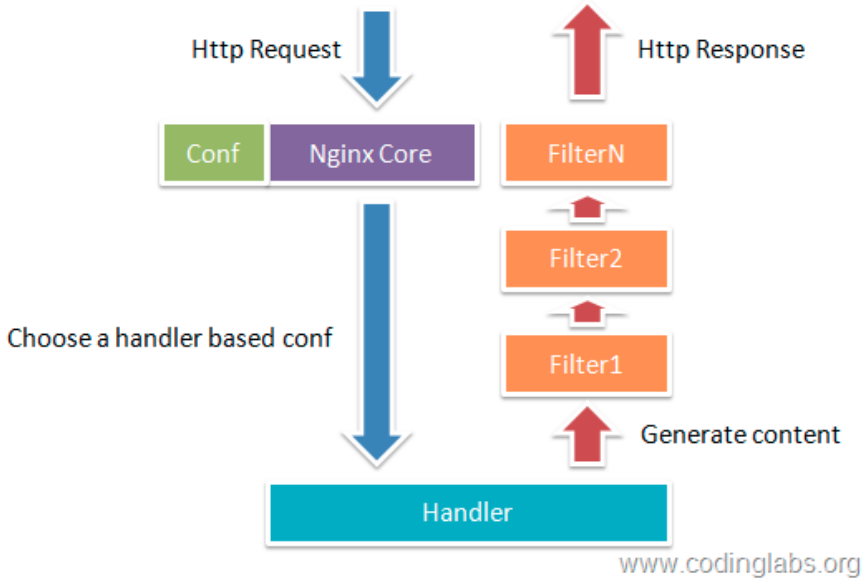



图1-1展示了Nginx模块常规的HTTP请求和响应的过程。

HTML(10)
Javascript(10)
jqGrid(4)
Laravel(23)
Linux(19)
Linux C(5)
Linux命令行(7)
Memcache(6)
MongoDB(1)
MySQL(37)
Nginx(16)
OAuth(1)
PHP(34)
Redis(9)
Session(3)
Ubuntu(13)
Vagrant(1)
Web(7)
Yii
技术(8)
正则表达式(4)

**随笔档案**

2018年4月 (2)
2018年3月 (6)
2017年9月 (1)
2017年6月 (2)
2017年5月 (1)
2017年3月 (1)
2016年12月 (3)
2016年11月 (3)
2016年10月 (1)
2016年9月 (1)
2016年7月 (1)
2016年6月 (15)
2016年5月 (28)
2016年4月 (11)
2016年3月 (6)
2016年2月 (4)
2016年1月 (26)
2015年12月 (10)
2015年11月 (3)
2015年10月 (4)
2015年9月 (4)
2015年8月 (2)
2015年7月 (10)
2015年6月 (1)
2015年5月 (2)
2015年4月 (6)
2015年3月 (8)
2015年2月 (2)
2015年1月 (14)
2014年12月 (1)
2014年11月 (8)
2014年10月 (8)
2014年9月 (4)
2014年8月 (14)
2013年9月 (1)
2013年8月 (4)

**文章分类**

Golang

**积分与排名**

积分 - 69568
排名 - 5382

Nginx本身做的工作实际很少，当它接到一个HTTP请求时，它仅仅是通过查找配置文件将此次请求映射到一个location block，而此location中所配置的各个指令则会启动不同的模块去完成工作，因此模块可以看做Nginx真正的劳动工作者。通常一个location中的指令会涉及一个handler模块和多个filter模块（当然，多个location可以复用同一个模块）。handler模块负责处理请求，完成响应内容的生成，而filter模块对响应内容进行处理。

Nginx的模块直接被编译进Nginx，因此属于静态编译方式。启动Nginx后，Nginx的模块被自动加载，不像Apache，首先将模块编译为一个so文件，然后在配置文件中指定是否进行加载。在解析配置文件时，Nginx的每个模块都有可能去处理某个请求，但是同一个处理请求只能由一个模块来完成。

2. Nginx的进程模型

在工作方式上，Nginx分为单工作进程和多工作进程两种模式。在单工作进程模式下，除主进程外，还有一个工作进程，工作进程是单线程的；在多工作进程模式下，每个工作进程包含多个线程。Nginx默认为单工作进程模式。

Nginx在启动后，会有一个master进程和多个worker进程。

master进程

主要用来管理worker进程，包含：接收来自外界的信号，向各worker进程发送信号，监控worker进程的运行状态，当worker进程退出后(异常情况下)，会自动重新启动新的worker进程。

master进程充当整个进程组与用户的交互接口，同时对进程进行监护。它不需要处理网络事件，不负责业务的执行，只会通过管理worker进程来实现重启服务、平滑升级、更换日志文件、配置文件实时生效等功能。

我们要控制nginx，只需要通过kill向master进程发送信号就行了。比如kill -HUP pid，则是告诉nginx，从容地重启nginx，我们一般用这个信号来重启nginx，或重新加载配置，因为是从容地重启，因此服务是不中断的。master进程在接收到HUP信号后是怎么做的呢？首先master进程在接到信号后，会先重新加载配置文件，然后再启动新的worker进程，并向所有老的worker进程发送信号，告诉他们可以光荣退休了。新的worker在启动后，就开始接收新的请求，而老的worker在收到来自master的信号后，就不再接收新的请求，并且在当前进程中的所有未处理完的请求处理完成后，再退出。当然，直接给master进程发送信号，这是比较老的操作方式，nginx在0.8版本之后，引入了一系列命令行参数，来方便我们管理。比如，./nginx -s reload，就是来重启nginx，./nginx -s stop，就是来停止nginx的运行。如何做到的呢？我们还是拿reload来说，我们看到，执行命令时，我们是启动一个新的nginx进程，而新的nginx进程在解析到reload参数后，就知道我们的目的是控制nginx来重新加载配置文件了，它会向master进程发送信号，然后接下来的动作，就和我们直接向master进程发送信号一样了。

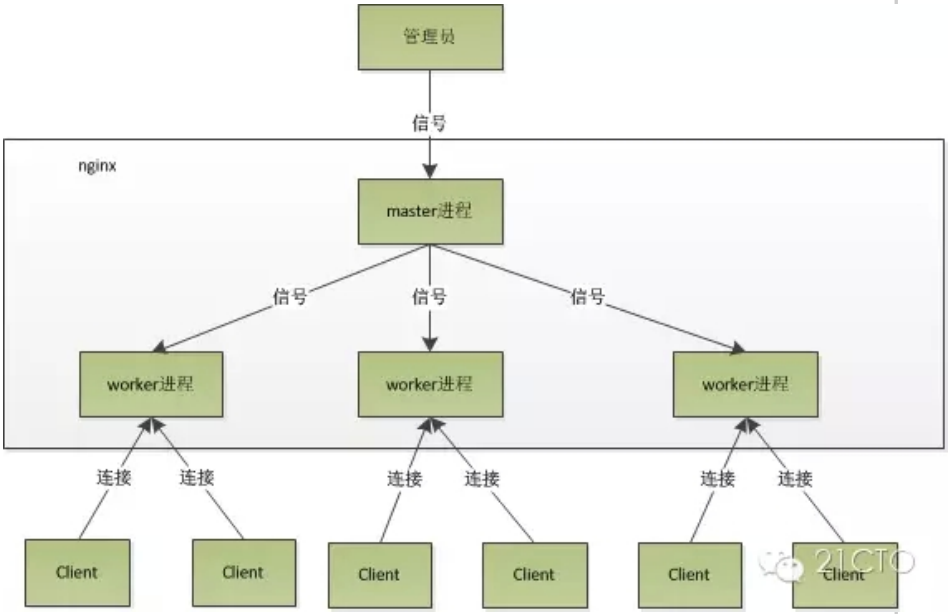
**worker进程：**

而基本的网络事件，则是放在worker进程中来处理了。多个worker进程之间是对等的，他们同等竞争来自客户端的请求，各进程互相之间是独立的。一个请求，只可能在一个worker进程中处理，一个worker进程，不可能处理其它进程的请求。worker进程的个数是可以设置的，一般我们会设置与机器cpu核数一致，这里面的原因与nginx的进程模型以及事件处理模型是分不开的。

worker进程之间是平等的，每个进程，处理请求的机会也是一样的。当我们提供80端口的http服务时，一个连接请求过来，每个进程都有可能处理这个连接，怎么做到的呢？首先，每个worker进程都是从master进程fork过来，在master进程里面，先建立好需要listen的socket（listenfd）之后，然后再fork出多个worker进程。所有worker进程的listenfd会在新连接到来时变得可读，为保证只有一个进程处理该连接，所有worker进程

在注册listenfd读事件前抢accept_mutex，抢到互斥锁的那个进程注册listenfd读事件，在读事件里调用accept接受该连接。当一个worker进程在accept这个连接之后，就开始读取请求，解析请求，处理请求，产生数据后，再返回给客户端，最后才断开连接，这样一个完整的请求就是这样的了。我们可以看到，一个请求，完全由worker进程来处理，而且只在一个worker进程中处理。worker进程之间是平等的，每个进程，处理请求的机会也是一样的。当我们提供80端口的http服务时，一个连接请求过来，每个进程都有可能处理这个连接，怎么做到的呢？首先，每个worker进程都是从master进程fork过来，在master进程里面，先建立好需要listen的socket（listenfd）之后，然后再fork出多个worker进程。所有worker进程的listenfd会在新连接到来时变得可读，为保证只有一个进程处理该连接，所有worker进程在注册listenfd读事件前抢accept_mutex，抢到互斥锁的那个进程注册listenfd读事件，在读事件里调用accept接受该连接。当一个worker进程在accept这个连接之后，就开始读取请求，解析请求，处理请求，产生数据后，再返回给客户端，最后才断开连接，这样一个完整的请求就是这样的了。我们可以看到，一个请求，完全由worker进程来处理，而且只在一个worker进程中处理。

nginx的进程模型，可以由下图来表示：




3. Nginx+FastCGI运行原理

## 1、什么是 FastCGI

FastCGI是一个可伸缩地、高速地在HTTP server和动态脚本语言间通信的接口。多数流行的HTTP server都支持FastCGI，包括Apache、Nginx和lighttpd等。同时，FastCGI也被许多脚本语言支持，其中就有PHP。

FastCGI是从CGI发展改进而来的。传统CGI接口方式的主要缺点是性能很差，因为每次HTTP服务器遇到动态程序时都需要重新启动脚本解析器来执行解析，然后将结果返回给HTTP服务器。这在处理高并发访问时几乎是不可用的。另外传统的CGI接口方式安全性也很差，现在已经很少使用了。

FastCGI接口方式采用C/S结构，可以将HTTP服务器和脚本解析服务器分开，同时在脚本解析服务器上启动一个或者多个脚本解析守护进程。当HTTP服务器每次遇到动态程序时，可以将其直接交付给FastCGI进程来执行，然后将得到的结果返回给浏览器。这种方式可以让HTTP服务器专一地处理静态请求或者将动态脚本服务器的结果返回给客户端，这在很大程度上提高了整个应用系统的性能。

## 2、Nginx+FastCGI运行原理

Nginx不支持对外部程序的直接调用或者解析，所有的外部程序（包括PHP）必须通过FastCGI接口来调用。FastCGI接口在Linux下是socket（这个socket可以是文件socket，也可以是ip socket）。

wrapper：为了调用CGI程序，还需要一个FastCGI的wrapper（wrapper可以理解为用于启动另一个程序的程序），这个wrapper绑定在某个固定socket上，如端口或者文件socket。当Nginx将CGI请求发送给这个socket的时候，通过FastCGI接口，wrapper接收到请求，然后Fork(派生）出一个新的线程，这个线程调用解释器或者外部程序处理脚本并读取返回数据；接着，wrapper再将返回的数据通过FastCGI接口，沿着固定的socket传递给Nginx；最后，Nginx将返回的数据（html页面或者图片）发送给客户端。这就是Nginx+FastCGI的整个运作过程，如图1-3所示。

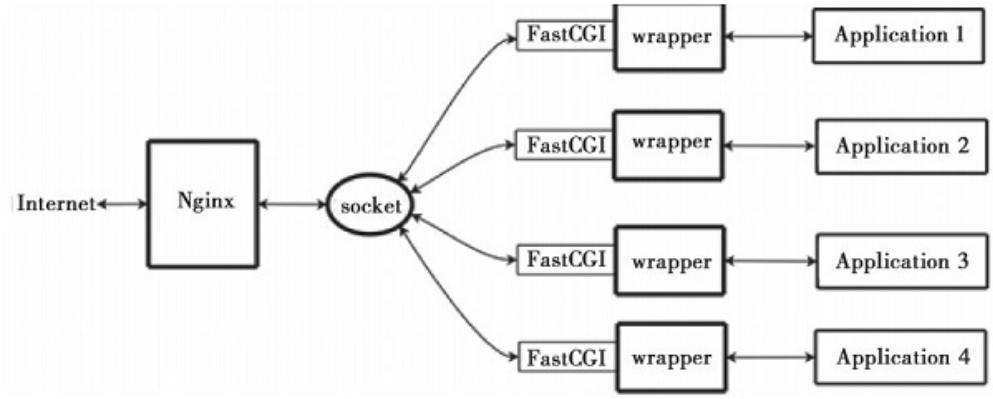



图 1-3 Nginx+FastCGI 运行过程

所以，我们首先需要一个wrapper，这个wrapper需要完成的工作：

1. 通过调用fastcgi（库）的函数通过socket和ningx通信（读写socket是fastcgi内部实现的功能，对wrapper是非透明的）
2. 调度thread，进行fork和kill
3. 和application（php）进行通信

## 3、spawn-fcgi与PHP-FPM

FastCGI接口方式在脚本解析服务器上启动一个或者多个守护进程对动态脚本进行解析，这些进程就是FastCGI进程管理器，或者称为FastCGI引擎。 spawn-fcgi与PHP-FPM就是支持PHP的两个FastCGI进程管理器。**因此HTTPServer完全解放出来，可以更好地进行响应和并发处理。**

spawn-fcgi与PHP-FPM的异同：

1）spawn-fcgi是HTTP服务器lighttpd的一部分，目前已经独立成为一个项目，一般与lighttpd配合使用来支持PHP。但是ligttpd的spwan-fcgi在高并发访问的时候，会出现内存泄漏甚至自动重启FastCGI的问题。即：PHP脚本处理器当机，这个时候如果用户访问的话，可能就会出现白页(即PHP不能被解析或者出错)。

2）Nginx是个轻量级的HTTP server，必须借助第三方的FastCGI处理器才可以对PHP进行解析，因此其实这样看来nginx是非常灵活的，它可以和任何第三方提供解析的处理器实现连接从而实现对PHP的解析(在nginx.conf中很容易设置)。nginx也可以使用spwan-fcgi(需要一同安装lighttpd，但是需要为nginx避开端口，一些较早的blog有这方面安装的教程)，但是由于spawn-fcgi具有上面所述的用户逐渐发现的缺陷，现在慢慢减少用nginx+spawn-fcgi组合了。

由于spawn-fcgi的缺陷，现在出现了第三方(目前已经加入到PHP core中)的PHP的FastCGI处理器PHP-FPM，它和spawn-fcgi比较起来有如下优点：

由于它是作为PHP的patch补丁来开发的，安装的时候需要和php源码一起编译，也就是说编译到php core中了，因此在性能方面要优秀一些；

同时它在处理高并发方面也优于spawn-fcgi，至少不会自动重启fastcgi处理器。因此，推荐使用Nginx+PHP/PHP-FPM这个组合对PHP进行解析。

相对Spawn-FCGI，PHP-FPM在CPU和内存方面的控制都更胜一筹，而且前者很容易崩溃，必须用crontab进行监控，而PHP-FPM则没有这种烦恼。

FastCGI 的主要优点是把动态语言和HTTP Server分离开来，所以Nginx与PHP/PHP-FPM经常被部署在不同的服务器上，以分担前端Nginx服务器的压力，使Nginx专一处理静态请求和转发动态请求，而PHP/PHP-FPM服务器专一解析PHP动态请求。

## 4、Nginx+PHP-FPM

PHP-FPM是管理FastCGI的一个管理器，它作为PHP的插件存在，在安装PHP要想使用PHP-FPM时在老php的老版本（php5.3.3之前）就需要把PHP-FPM以补丁的形式安装到PHP中，而且PHP要与PHP-FPM版本一致，这是必须的）

PHP-FPM其实是PHP源代码的一个补丁，旨在将FastCGI进程管理整合进PHP包中。必须将它patch到你的PHP源代码中，在编译安装PHP后才可以使用。

PHP5.3.3已经集成php-fpm了，不再是第三方的包了。PHP-FPM提供了更好的PHP进程管理方式，可以有效控制内存和进程、可以平滑重载PHP配置，比spawn-fcgi具有更多优点，所以被PHP官方收录了。在./configure的时候带 –enable-fpm参数即可开启PHP-FPM。

fastcgi已经在php5.3.5的core中了，不必在configure时添加 --enable-fastcgi了。老版本如php5.2的需要加此项。

当我们安装Nginx和PHP-FPM完后，配置信息：

PHP-FPM的默认配置php-fpm.conf：
listen_address  127.0.0.1:9000 #这个表示php的fastcgi进程监听的ip地址以及端口
start_servers
min_spare_servers
max_spare_servers

Nginx配置运行php： 编辑nginx.conf加入如下语句：

```
location ~ \.php$ {
    root html;
    fastcgi_pass 127.0.0.1:9000; 指定了fastcgi进程侦听的端口,nginx就是通过这里与php交互的
    fastcgi_index index.php;
    include fastcgi_params;
     fastcgi_param SCRIPT_FILENAME /usr/local/nginx/html$fastcgi_script_name;
}
```

Nginx通过location指令，将所有以php为后缀的文件都交给127.0.0.1:9000来处理，而这里的IP地址和端口就是FastCGI进程监听的IP地址和端口。

其整体工作流程：

1)、FastCGI进程管理器php-fpm自身初始化，启动主进程php-fpm和启动start_servers个CGI 子进程。

主进程php-fpm主要是管理fastcgi子进程，监听9000端口。

fastcgi子进程等待来自Web Server的连接。

2)、当客户端请求到达Web Server Nginx是时，Nginx通过location指令，将所有以php为后缀的文件都交给127.0.0.1:9000来处理，即Nginx通过location指令，将所有以php为后缀的文件都交给127.0.0.1:9000来处理。

3）FastCGI进程管理器PHP-FPM选择并连接到一个子进程CGI解释器。Web server将CGI环境变量和标准输入发送到FastCGI子进程。

4)、FastCGI子进程完成处理后将标准输出和错误信息从同一连接返回Web Server。当FastCGI子进程关闭连接时，请求便告处理完成。

5)、FastCGI子进程接着等待并处理来自FastCGI进程管理器（运行在 WebServer 中）的下一个连接。

**4. Nginx+PHP正确配置**

一般web都做统一入口：把PHP请求都发送到同一个文件上，然后在此文件里通过解析「REQUEST_URI」实现路由。

Nginx配置文件分为好多块，常见的从外到内依次是「http」、「server」、「location」等等，缺省的继承关系是从外到内，也就是说内层块会自动获取外层块的值作为缺省值。

例如：

[plain] view plain copy print ?

```
server {
    listen 80;
    server_name foo.com;
    root /path;
    location / {
        index index.html index.htm index.php;
        if (!-e $request_filename) {
            rewrite . /index.php last;
        }
    }
    location ~ \.php$ {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME /path$fastcgi_script_name;
        fastcgi_pass 127.0.0.1:9000;
        fastcgi_index index.php;
    }
}
```

**1) 不应该在location 模块定义index**

一旦未来需要加入新的「location」，必然会出现重复定义的「index」指令，这是因为多个「location」是平级的关系，不存在继承，此时应该在「server」里定义「index」，借助继承关系，「index」指令在所有的「location」中都能生效。

**2) 使用try_files**

接下来看看「if」指令，说它是大家误解最深的Nginx指令毫不为过：

if (!-e $request_filename) {

rewrite . /index.php last;

}

很多人喜欢用「if」指令做一系列的检查，不过这实际上是「try_files」指令的职责：

try_files $uri $uri/ /index.php;

除此以外，初学者往往会认为「if」指令是内核级的指令，但是实际上它是rewrite模块的一部分，加上Nginx配置实际上是声明式的，而非过程式的，所以当其和非rewrite模块的指令混用时，结果可能会非你所愿。

**3）fastcgi_params」配置文件**：

include fastcgi_params;

Nginx有两份fastcgi配置文件，分别是「fastcgi_params」和「fastcgi.conf」，它们没有太大的差异，唯一的区别是后者比前者多了一行「SCRIPT_FILENAME」的定义：

fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;

注意：$document_root 和 $fastcgi_script_name 之间没有 /。

原本Nginx只有「fastcgi_params」，后来发现很多人在定义「SCRIPT_FILENAME」时使用了硬编码的方式，于是为了规范用法便引入了「fastcgi.conf」。

不过这样的话就产生一个疑问：为什么一定要引入一个新的配置文件，而不是修改旧的配置文件？这是因为「fastcgi_param」指令是数组型的，和普通指令相同的是：内层替换外层；和普通指令不同的是：当在同级多次使用的时候，是新增而不是替换。换句话说，如果在同级定义两次「SCRIPT_FILENAME」，那么它们都会被发送到后端，这可能会导致一些潜在的问题，为了避免此类情况，便引入了一个新的配置文件。

此外，我们还需要考虑一个安全问题：在PHP开启「cgi.fix_pathinfo」的情况下，PHP可能会把错误的文件类型当作PHP文件来解析。如果Nginx和PHP安装在同一台服务器上的话，那么最简单的解决方法是用「try_files」指令做一次过滤：

```
try_files $uri =404;
```

依照前面的分析，给出一份改良后的版本，是不是比开始的版本清爽了很多：

[plain] view plain copy print ?

```
server {
    listen 80;
    server_name foo.com;
    root /path;
    index index.html index.htm index.php;
    location / {
        try_files $uri $uri/ /index.php;
    }
    location ~ \.php$ {
       try_files $uri =404;
       include fastcgi.conf;
       fastcgi_pass 127.0.0.1:9000;
   }
}
```

**5. Nginx为啥性能高－多进程IO模型**

参考http://mp.weixin.qq.com/s?__biz=MjM5NTg2NTU0Ng==&mid=407889757&idx=3&sn=cfa8a70a5fd2a674a91076f67808273c&scene=23&srcid=0401aeJQEraSG6uvLj69Hfve#rd

**1、nginx采用多进程模型好处**

首先，对于每个worker进程来说，独立的进程，不需要加锁，所以省掉了锁带来的开销，同时在编程以及问题查找时，也会方便很多。

其次，采用独立的进程，可以让互相之间不会影响，一个进程退出后，其它进程还在工作，服务不会中断，master进程则很快启动新的worker进程。当然，worker进程的异常退出，肯定是程序有bug了，异常退出，会导致当前worker上的所有请求失败，不过不会影响到所有请求，所以降低了风险。

**1、nginx多进程事件模型：异步非阻塞**

虽然nginx采用多worker的方式来处理请求，每个worker里面只有一个主线程，那能够处理的并发数很有限啊，多少个worker就能处理多少个并发，何来高并发呢？非也，这就是nginx的高明之处，**nginx采用了异步非阻塞的方式来处理请求**，也就是说，nginx是可以同时处理成千上万个请求的。一个worker进程可以同时处理的请求数只受限于内存大小，而且在架构设计上，不同的worker进程之间处理并发请求时几乎没有同步锁的限制，worker进程通常不会进入睡眠状态，因此，当Nginx上的进程数与CPU核心数相等时（最好每一个worker进程都绑定特定的CPU核心），进程间切换的代价是最小的。

而apache的常用工作方式（apache也有异步非阻塞版本，但因其与自带某些模块冲突，所以不常用），每个进程在一个时刻只处理一个请求，因此，当并发数上到几千时，就同时有几千的进程在处理请求了。这对操作系统来说，是个不小的挑战，进程带来的内存占用非常大，进程的上下文切换带来的cpu开销很大，自然性能就上不去了，而这些开销完全是没有意义的。

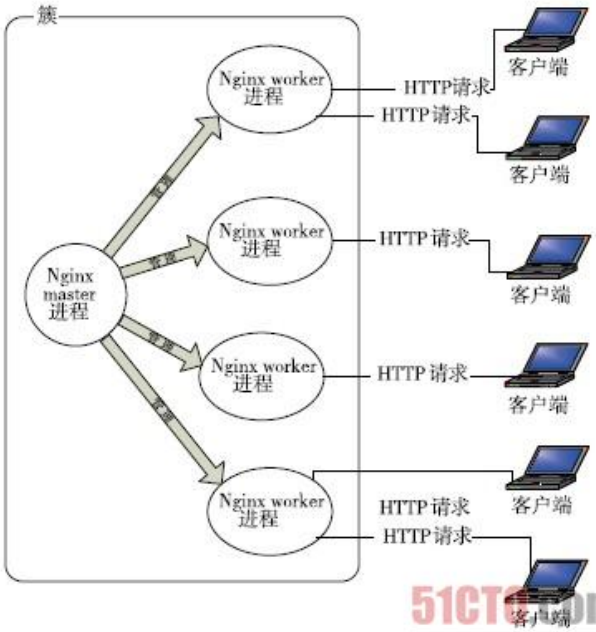


图 2-1 部署后 Nginx 进程间的关系

为什么nginx可以采用异步非阻塞的方式来处理呢，或者异步非阻塞到底是怎么回事呢？

我们先回到原点，看看一个请求的完整过程:首先，请求过来，要建立连接，然后再接收数据，接收数据后，再发送数据。

具体到系统底层，就是读写事件，而当读写事件没有准备好时，必然不可操作，如果不用非阻塞的方式来调用，那就得阻塞调用了，事件没有准备好，那就只能等了，等事件准备好了，你再继续吧。阻塞调用会进入内核等待，cpu就会让出去给别人用了，对单线程的worker来说，显然不合适，当网络事件越多时，大家都在等待呢，cpu空闲下来没人用，cpu利用率自然上不去了，更别谈高并发了。好吧，你说加进程数，这跟apache的线程模型有什么区别，注意，别增加无谓的上下文切换。所以，在nginx里面，最忌讳阻塞的系统调用了。不要阻塞，那就非阻塞喽。非阻塞就是，事件没有准备好，马上返回EAGAIN，告诉你，事件还没准备好呢，你慌什么，过会再来吧。好吧，你过一会，再来检查一下事件，直到事件准备好了为止，在这期间，你就可以先去做其它事情，然后再来看看事件好了没。虽然不阻塞了，但你得不时地过来检查一下事件的状态，你可以做更多的事情了，但带来的开销也是不小的。

关于IO模型：http://blog.csdn.net/hguisu/article/details/7453390

**nginx支持的事件模型如下（nginx的wiki）：**

Nginx支持如下处理连接的方法（I/O复用方法），这些方法可以通过use指令指定。

- **select**– 标准方法。 如果当前平台没有更有效的方法，它是编译时默认的方法。你可以使用配置参数 –with-select_module 和 –without-select_module 来启用或禁用这个模块。
- **poll**– 标准方法。 如果当前平台没有更有效的方法，它是编译时默认的方法。你可以使用配置参数 –with-poll_module 和 –without-poll_module 来启用或禁用这个模块。
- **kqueue**– 高效的方法，使用于 FreeBSD 4.1+, OpenBSD 2.9+, NetBSD 2.0 和 MacOS X. 使用双处理器的MacOS X系统使用kqueue可能会造成内核崩溃。
- **epoll** – 高效的方法，使用于Linux内核2.6版本及以后的系统。在某些发行版本中，如SuSE 8.2, 有让2.4版本的内核支持epoll的补丁。
- **rtsig** – 可执行的实时信号，使用于Linux内核版本2.2.19以后的系统。默认情况下整个系统中不能出现大于1024个POSIX实时(排队)信号。这种情况 对于高负载的服务器来说是低效的；所以有必要通过调节内核参数 /proc/sys/kernel/rtsig-max 来增加队列的大小。可是从Linux内核版本2.6.6-mm2开始， 这个参数就不再使用了，并且对于每个进程有一个独立的信号队列，这个队列的大小可以用 RLIMIT_SIGPENDING 参数调节。当这个队列过于拥塞，nginx就放弃它并且开始使用 poll 方法来处理连接直到恢复正常。
- **/dev/poll** – 高效的方法，使用于 Solaris 7 11/99+, HP/UX 11.22+ (eventport), IRIX 6.5.15+ 和 Tru64 UNIX 5.1A+.
- **eventport** – 高效的方法，使用于 Solaris 10. 为了防止出现内核崩溃的问题， 有必要安装这个 安全补丁。

在linux下面，只有epoll是高效的方法

下面再来看看epoll到底是如何高效的

Epoll是Linux内核为处理大批量句柄而作了改进的poll。 要使用epoll只需要这三个系统调用：epoll_create(2)， epoll_ctl(2)， epoll_wait(2)。它是在2.5.44内核中被引进的(epoll(4) is a new API introduced in Linux kernel 2.5.44)，在2.6内核中得到广泛应用。

epoll的优点

- 支持一个进程打开大数目的socket描述符(FD)

select 最不能忍受的是一个进程所打开的FD是有一定限制的，由FD_SETSIZE设置，默认值是2048。对于那些需要支持的上万连接数目的IM服务器来说显 然太少了。这时候你一是可以选择修改这个宏然后重新编译内核，不过资料也同时指出这样会带来网络效率的下降，二是可以选择多进程的解决方案(传统的 Apache方案)，不过虽然linux上面创建进程的代价比较小，但仍旧是不可忽视的，加上进程间数据同步远比不上线程间同步的高效，所以也不是一种完 美的方案。不过 epoll则没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左 右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

- IO效率不随FD数目增加而线性下降

传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，不过由于网络延时，任一时间只有部分的socket是”活跃”的，但 是select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题，它只会对”活

跃”的socket进行操 作—这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有”活跃”的socket才会主动的去调用 callback函数，其他idle状态socket则不会，在这点上，epoll实现了一个”伪”AIO，因为这时候推动力在os内核。在一些 benchmark中，如果所有的socket基本上都是活跃的—比如一个高速LAN环境，epoll并不比select/poll有什么效率，相 反，如果过多使用epoll_ctl,效率相比还有稍微的下降。但是一旦使用idle connections模拟WAN环境,epoll的效率就远在select/poll之上了。

- 使用mmap加速内核与用户空间的消息传递。

这 点实际上涉及到epoll的具体实现了。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很 重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。而如果你想我一样从2.5内核就关注epoll的话，一定不会忘记手工 mmap这一步的。

- 内核微调

这一点其实不算epoll的优点了，而是整个linux平台的优点。也许你可以怀疑linux平台，但是你无法回避linux平台赋予你微调内核的能力。比如，内核TCP/IP协 议栈使用内存池管理sk_buff结构，那么可以在运行时期动态调整这个内存pool(skb_head_pool)的大小— 通过echo XXXX>/proc/sys/net/core/hot_list_length完成。再比如listen函数的第2个参数(TCP完成3次握手 的数据包队列长度)，也可以根据你平台内存大小动态调整。更甚至在一个数据包面数目巨大但同时每个数据包本身大小却很小的特殊系统上尝试最新的NAPI网卡驱动架构。

**(epoll内容，参考epoll_互动百科)**

推荐设置worker的个数为cpu的核数，在这里就很容易理解了，更多的worker数，只会导致进程来竞争cpu资源了，从而带来不必要的上下文切换。而且，nginx为了更好的利用多核特性，提供了cpu亲缘性的绑定选项，我们可以将某一个进程绑定在某一个核上，这样就不会因为进程的切换带来cache的失效。像这种小的优化在nginx中非常常见，同时也说明了nginx作者的苦心孤诣。比如，nginx在做4个字节的字符串比较时，会将4个字符转换成一个int型，再作比较，以减少cpu的指令数等等。

代码来总结一下nginx的事件处理模型：

[cpp] view plain copy print ?

```cpp
while (true) {
    for t in run_tasks:
        t.handler();
    update_time(&now);
    timeout = ETERNITY;
    for t in wait_tasks: /* sorted already */
        if (t.time <= now) {
            t.timeout_handler();
        } else {
            timeout = t.time - now;
            break;
        }
    nevents = poll_function(events, timeout);
    for i in nevents:
        task t;
        if (events[i].type == READ) {
            t.handler = read_handler;
        } else { /* events[i].type == WRITE */
            t.handler = write_handler;
        }
        run_tasks_add(t);
}
```

6. Nginx优化

## 1. 编译安装过程优化

**1）.减小Nginx编译后的文件大小**

在编译Nginx时，默认以debug模式进行，而在debug模式下会插入很多跟踪和ASSERT之类的信息，编译完成后，一个Nginx要有好几兆字节。而在编译前取消Nginx的debug模式，编译完成后Nginx只有几百千字节。因此可以在编译之前，修改相关源码，取消debug模式。具体方法如下：

在Nginx源码文件被解压后，找到源码目录下的auto/cc/gcc文件，在其中找到如下几行：

```
1. # debug
2. CFLAGS="$CFLAGS -g"
```

注释掉或删掉这两行，即可取消debug模式。

**2.为特定的CPU指定CPU类型编译优化**

在编译Nginx时，默认的GCC编译参数是"-O"，要优化GCC编译，可以使用以下两个参数：

```
1. --with-cc-opt='-O3'
2. --with-cpu-opt=CPU  #为特定的 CPU 编译，有效的值包括：
   pentium, pentiumpro, pentium3, # pentium4, athlon, opteron, amd64, sparc32
   , sparc64, ppc64
```

要确定CPU类型，可以通过如下命令：

```
1. [root@localhost home]#cat /proc/cpuinfo | grep "model name"
```

## 2. 利用TCMalloc优化Nginx的性能

TCMalloc的全称为Thread-Caching Malloc，是谷歌开发的开源工具google-perftools中的一个成员。与标准的glibc库的Malloc相比，TCMalloc库在内存分配效率和速度上要高很多，这在很大程度上提高了服务器在高并发情况下的性能，从而降低了系统的负载。下面简单介绍如何为Nginx添加TCMalloc库支持。

要安装TCMalloc库，需要安装libunwind（32位操作系统不需要安装）和google-perftools两个软件包，libunwind库为基于64位CPU和操作系统的程序提供了基本函数调用链和函数调用寄存器功能。下面介绍利用TCMalloc优化Nginx的具体操作过程。

**1).安装libunwind库**

可以从http://download.savannah.gnu.org/releases/libunwind下载相应的libunwind版本，这里下载的是libunwind-0.99-alpha.tar.gz。安装过程如下：

```
1. [root@localhost home]#tar zxvf libunwind-0.99-alpha.tar.gz
2. [root@localhost home]# cd libunwind-0.99-alpha/
3. [root@localhost libunwind-0.99-alpha]#CFLAGS=-fPIC ./configure
4. [root@localhost libunwind-0.99-alpha]#make CFLAGS=-fPIC
5. [root@localhost libunwind-0.99-alpha]#make CFLAGS=-fPIC install
```

**2).安装google-perftools**

可以从http://google-perftools.googlecode.com下载相应的google-perftools版本，这里下载的是google-perftools-1.8.tar.gz。安装过程如下：

```
1. [root@localhost home]#tar zxvf google-perftools-1.8.tar.gz
2. [root@localhost home]#cd google-perftools-1.8/
3. [root@localhost google-perftools-1.8]# ./configure
4. [root@localhost google-perftools-1.8]#make && make install
5. [root@localhost google-perftools-1.8]#echo "/usr/
   local/lib" > /etc/ld.so.conf.d/usr_local_lib.conf
6. [root@localhost google-perftools-1.8]# ldconfig
```

至此，google-perftools安装完成。

**3).重新编译Nginx**

为了使Nginx支持google-perftools，需要在安装过程中添加“–with-google_perftools_module”选项重新编译Nginx。安装代码如下：

```
1. [root@localhostnginx-0.7.65]#./configure \
2. >--with-google_perftools_module --with-http_stub_status_module  --
   prefix=/opt/nginx
3. [root@localhost nginx-0.7.65]#make
4. [root@localhost nginx-0.7.65]#make install
```

到这里Nginx安装完成。

**4).为google-perftools添加线程目录**

创建一个线程目录，这里将文件放在/tmp/tcmalloc下。操作如下：

```
1. [root@localhost home]#mkdir /tmp/tcmalloc
2. [root@localhost home]#chmod 0777 /tmp/tcmalloc
```

**5).修改Nginx主配置文件**

修改nginx.conf文件，在pid这行的下面添加如下代码：

```
1. #pid        logs/nginx.pid;
2. google_perftools_profiles /tmp/tcmalloc;
```

接着，重启Nginx即可完成google-perftools的加载。

**6).验证运行状态**

为了验证google-perftools已经正常加载，可通过如下命令查看：

```
1. [root@ localhost home]# lsof -n | grep tcmalloc
2. nginx      2395 nobody   9w  REG    8,8       0    1599440 /tmp/tcmalloc.2395
3. nginx      2396 nobody   11w REG   8,8       0    1599443 /tmp/tcmalloc.2396
4. nginx      2397 nobody   13w REG  8,8        0    1599441  /tmp/tcmalloc.2397
5. nginx     2398 nobody    15w REG  8,8     0    1599442 /tmp/tcmalloc.2398
```

由于在Nginx配置文件中设置worker_processes的值为4，因此开启了4个Nginx线程，每个线程会有一行记录。每个线程文件后面的数字值就是启动的Nginx的pid值。

至此，利用TCMalloc优化Nginx的操作完成。

## 3.Nginx内核参数优化

内核参数的优化，主要是在Linux系统中针对Nginx应用而进行的系统内核参数优化。

下面给出一个优化实例以供参考。

```
1. net.ipv4.tcp_max_tw_buckets = 6000
```

2. net.ipv4.ip_local_port_range = 1024 65000
3. net.ipv4.tcp_tw_recycle = 1
4. net.ipv4.tcp_tw_reuse = 1
5. net.ipv4.tcp_syncookies = 1
6. net.core.somaxconn = 262144
7. net.core.netdev_max_backlog = 262144
8. net.ipv4.tcp_max_orphans = 262144
9. net.ipv4.tcp_max_syn_backlog = 262144
10. net.ipv4.tcp_synack_retries = 1
11. net.ipv4.tcp_syn_retries = 1
12. net.ipv4.tcp_fin_timeout = 1
13. net.ipv4.tcp_keepalive_time = 30

将上面的内核参数值加入/etc/sysctl.conf文件中，然后执行如下命令使之生效：

1. [root@ localhost home]#/sbin/sysctl -p

下面对实例中选项的含义进行介绍：

net.ipv4.tcp_max_tw_buckets ：选项用来设定timewait的数量，默认是180 000，这里设为6000。

net.ipv4.ip_local_port_range:选项用来设定允许系统打开的端口范围。在高并发情况否则端口号会不够用。

net.ipv4.tcp_tw_recycle:选项用于设置启用timewait快速回收.

net.ipv4.tcp_tw_reuse:选项用于设置开启重用，允许将TIME-WAIT sockets重新用于新的TCP连接。

net.ipv4.tcp_syncookies:选项用于设置开启SYN Cookies，当出现SYN等待队列溢出时，启用cookies进行处理。

net.core.somaxconn:选项的默认值是128， 这个参数用于调节系统同时发起的tcp连接数，在高并发的请求中，默认的值可能会导致链接超时或者重传，因此，需要结合并发请求数来调节此值。

net.core.netdev_max_backlog:选项表示当每个网络接口接收数据包的速率比内核处理这些包的速率快时，允许发送到队列的数据包的最大数目。

net.ipv4.tcp_max_orphans:选项用于设定系统中最多有多少个TCP套接字不被关联到任何一个用户文件句柄上。如果超过这个数字，孤立连接将立即被复位并打印出警告信息。这个限制只是为了防止简单的DoS攻击。不能过分依靠这个限制甚至人为减小这个值，更多的情况下应该增加这个值。

net.ipv4.tcp_max_syn_backlog:选项用于记录那些尚未收到客户端确认信息的连接请求的最大值。对于有128MB内存的系统而言，此参数的默认值是1024，对小内存的系统则是128。

net.ipv4.tcp_synack_retries参数的值决定了内核放弃连接之前发送SYN+ACK包的数量。

net.ipv4.tcp_syn_retries选项表示在内核放弃建立连接之前发送SYN包的数量。

net.ipv4.tcp_fin_timeout选项决定了套接字保持在FIN-WAIT-2状态的时间。默认值是60秒。正确设置这个值非常重要，有时即使一个负载很小的Web服务器，也会出现大量的死套接字而产生内存溢出的风险。

net.ipv4.tcp_syn_retries选项表示在内核放弃建立连接之前发送SYN包的数量。

如果发送端要求关闭套接字，net.ipv4.tcp_fin_timeout选项决定了套接字保持在FIN-WAIT-2状态的时间。接收端可以出错并永远不关闭连接，甚至意外宕机。

net.ipv4.tcp_fin_timeout的默认值是60秒。需要注意的是，即使一个负载很小的Web服务器，也会出现因为大量的死套接字而产生内存溢出的风险。FIN-WAIT-2的危险性比FIN-WAIT-1要小，因为它最多只能消耗1.5KB的内存，但是其生存期长些。

net.ipv4.tcp_keepalive_time选项表示当keepalive启用的时候，TCP发送keepalive消息的频度。默认值是2（单位是小时）。

## 4. PHP-FPM的优化

如果您高负载网站使用PHP-FPM管理FastCGI，这些技巧也许对您有用:

**1）增加FastCGI进程数**

把PHP FastCGI子进程数调到100或以上，在4G内存的服务器上200就可以建议通过压力测试获取最佳值。

**2）增加 PHP-FPM打开文件描述符的限制**

标签rlimit_files用于设置PHP-FPM对打开文件描述符的限制，默认值为1024。这个标签的值必须和Linux内核打开文件数关联起来，例如，要将此值设置为65 535，就必须在Linux命令行执行“ulimit -HSn 65536”。

然后 增加 PHP-FPM打开文件描述符的限制:
# vi /path/to/php-fpm.conf
找到“<valuename="rlimit_files">1024</value>”
把1024更改为 4096或者更高.
重启 PHP-FPM.

ulimit -n 要调整为65536甚至更大。如何调这个参数，可以参考网上的一些文章。命令行下执行 ulimit -n 65536即可修改。如果不能修改，需要设置 /etc/security/limits.conf，加入

* hard nofile65536

* soft nofile 65536

**3）适当增加max_requests**

标签max_requests指明了每个children最多处理多少个请求后便会被关闭，默认的设置是500。

<value name="max_requests"> 500 </value>

## 4. nginx.conf的参数优化

nginx要开启的进程数 一般等于cpu的总核数 其实一般情况下开4个或8个就可以。

每个nginx进程消耗的内存10兆的模样

**worker_cpu_affinity**
仅适用于linux，使用该选项可以绑定worker进程和CPU（2.4内核的机器用不了）

假如是8 cpu 分配如下：
worker_cpu_affinity 00000001 00000010 00000100 00001000 00010000

00100000 01000000 10000000

nginx可以使用多个worker进程，原因如下：

to use SMP
to decrease latency when workers blockend on disk I/O

to limit number of connections per process when select()/poll() is

used The worker_processes and worker_connections from the event sections

allows you to calculate maxclients value: k max_clients = worker_processes * worker_connections

worker_rlimit_nofile 102400;

每个nginx进程打开文件描述符最大数目 配置要和系统的单进程打开文件数一致,linux 2.6内核下开启文件打开数为65535，worker_rlimit_nofile就相应应该填写65535 nginx调度时分配请求到进程并不是那么的均衡，假如超过会返回502错误。我这里写的大一点

**use epoll**

Nginx使用了最新的epoll（Linux 2.6内核）和kqueue（freebsd）网络I/O模型，而Apache则使用的是传统的select模型。

处理大量的连接的读写，Apache所采用的select网络I/O模型非常低效。在高并发服务器中，轮询I/O是最耗时间的操作 目前Linux下能够承受高并发

访问的Squid、Memcached都采用的是epoll网络I/O模型。

**worker_connections 65535;**
每个工作进程允许最大的同时连接数 （Maxclient = work_processes * worker_connections）

**keepalive_timeout 75**

keepalive超时时间

这里需要注意官方的一句话：
The parameters can differ from each other. Line Keep-Alive:

timeout=time understands Mozilla and Konqueror. MSIE itself shuts

keep-alive connection approximately after 60 seconds.

**client_header_buffer_size 16k**
**large_client_header_buffers 4 32k**
客户请求头缓冲大小
nginx默认会用client_header_buffer_size这个buffer来读取header值，如果header过大，它会使用large_client_header_buffers来读取

如果设置过小HTTP头/Cookie过大 会报400 错误 nginx 400 bad request
求行如果超过buffer，就会报HTTP 414错误(URI Too Long) nginx接受最长的HTTP头部大小必须比其中一个buffer大，否则就会报400的HTTP错误(Bad Request)。

**open_file_cache max 102400**

使用字段:http, server, location 这个指令指定缓存是否启用,如果启用,将记录文件以下信息: ·打开的文件描述符,大小信息和修改时间. ·存在的目录信息. ·在搜索文件过程中的错误信息 -- 没有这个文件,无法正确读取,参考open_file_cache_errors 指令选项: ·max - 指定缓存的最大数目,如果缓存溢出,最长使用过的文件(LRU)将被移除
例: open_file_cache max=1000 inactive=20s; open_file_cache_valid 30s; open_file_cache_min_uses 2; open_file_cache_errors on;

open_file_cache_errors
语法:open_file_cache_errors on | off 默认值:open_file_cache_errors off 使用字段:http, server, location 这个指令指定是否在搜索一个文件是记录cache错误.

**open_file_cache_min_uses**

语法:open_file_cache_min_uses number 默认值:open_file_cache_min_uses 1 使用字段:http, server, location 这个指令指定了在open_file_cache指令无效的参数中一定的时间范围内可以使用的最小文件数,如 果使用更大的值,文件描述符在cache中总是打开状态.
open_file_cache_valid

语法:open_file_cache_valid time 默认值:open_file_cache_valid 60 使用字段:http, server, location 这个指令指定了何时需要检查open_file_cache中缓存项目的有效信息.
开启gzip
gzip on;
gzip_min_length 1k;
gzip_buffers 4 16k;
gzip_http_version 1.0;
gzip_comp_level 2;
gzip_types text/plain application/x-JavaScript text/css

application/xml;
gzip_vary on;

缓存静态文件：

location ~* ^.+\.(swf|gif|png|jpg|js|css)$ {
root /usr/local/ku6/ktv/show.ku6.com/;
expires 1m;
}

**7. 错误排查**

**1、Nginx 502 Bad Gateway**

php-cgi进程数不够用、php执行时间长（mysql慢）、或者是php-cgi进程死掉，都会出现502错误

一般来说Nginx 502 Bad Gateway和php-fpm.conf的设置有关，而Nginx 504 Gateway Time-out则是与nginx.conf的设置有关

1）、查看当前的PHP FastCGI进程数是否够用：

netstat -anpo | grep "php-cgi" | wc -l

如果实际使用的“FastCGI进程数”接近预设的“FastCGI进程数”，那么，说明“FastCGI进程数”不够用，需要增大。

2）、部分PHP程序的执行时间超过了Nginx的等待时间，可以适当增加

nginx.conf配置文件中FastCGI的timeout时间，例如：

http {
.....
fastcgi_connect_timeout 300;
fastcgi_send_timeout 300;
fastcgi_read_timeout 300;
.....
}

**2、413 Request Entity Too Large**

解决：增大client_max_body_size

client_max_body_size:指令指定允许客户端连接的最大请求实体大小,它出现在请求头部的Content-Length字段. 如果请求大于指定的值,客户端将收到一个"Request Entity Too Large" (413)错误. 记住,浏览器并不知道怎样显示这个错误.

php.ini中增大
post_max_size 和upload_max_filesize

3 Ngnix error.log出现：upstream sent too big header while reading response header from upstream错误

**1)如果是nginx反向代理**

proxy是nginx作为client转发时使用的，如果header过大，超出了默认的1k，就会引发上述的upstream sent too big header （说白了就是nginx把外部请求给后端server，后端server返回的header 太大nginx处理不过来就导致了。

```
server {
    listen       80;
    server_name  *.xywy.com ;

    large_client_header_buffers 4 16k;

    location / {

     #添加这3行
      proxy_buffer_size 64k;
      proxy_buffers   32 32k;
      proxy_busy_buffers_size 128k;

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

  }

}
```

2) 如果是 nginx+PHPcgi

错误带有 upstream: "fastcgi://127.0.0.1:9000"。就该

多加：

```
fastcgi_buffer_size 128k;
fastcgi_buffers 4 128k;

server {
    listen       80;
    server_name  ddd.com;
    index index.html index.htm index.php;

    client_header_buffer_size 128k;
    large_client_header_buffers 4 128k;
    proxy_buffer_size 64k;
    proxy_buffers 8 64k;
    fastcgi_buffer_size 128k;
    fastcgi_buffers 4 128k;


    location / {
```

```
        ……
      }
}
```

**8. Nginx的php漏洞**

漏洞介绍：nginx是一款高性能的web服务器，使用非常广泛，其不仅经常被用作反向代理，也可以非常好的支持PHP的运行。80sec发现其中存在一个较为严重的安全问题，默认情况下可能导致服务器错误的将任何类型的文件以PHP的方式进行解析，这将导致严重的安全问题，使得恶意的攻击者可能攻陷支持php的nginx服务器。

漏洞分析：nginx默认以cgi的方式支持php的运行，譬如在配置文件当中可以以

```
location ~ .php$ {
root html;
fastcgi_pass 127.0.0.1:9000;
fastcgi_index index.php;
fastcgi_param SCRIPT_FILENAME /scripts$fastcgi_script_name;
include fastcgi_params;
}
```

的方式支持对php的解析，location对请求进行选择的时候会使用URI环境变量进行选择，其中传递到后端Fastcgi的关键变量SCRIPT_FILENAME由nginx生成的$fastcgi_script_name决定，而通过分析可以看到$fastcgi_script_name是直接由URI环境变量控制的，这里就是产生问题的点。而为了较好的支持PATH_INFO的提取，在PHP的配置选项里存在cgi.fix_pathinfo选项，其目的是为了从SCRIPT_FILENAME里取出真正的脚本名。
那么假设存在一个http://www.80sec.com/80sec.jpg，我们以如下的方式去访问

http://www.80sec.com/80sec.jpg/80sec.php

将会得到一个URI

/80sec.jpg/80sec.php

经过location指令，该请求将会交给后端的fastcgi处理，nginx为其设置环境变量SCRIPT_FILENAME，内容为

/scripts/80sec.jpg/80sec.php

而在其他的webserver如lighttpd当中，我们发现其中的SCRIPT_FILENAME被正确的设置为

/scripts/80sec.jpg

所以不存在此问题。
后端的fastcgi在接受到该选项时，会根据fix_pathinfo配置决定是否对SCRIPT_FILENAME进行额外的处理，一般情况下如果不对fix_pathinfo进行设置将影响使用PATH_INFO进行路由选择的应用，所以该选项一般配置开启。Php通过该选项之后将查找其中真正的脚本文件名字，查找的方式也是查看文件是否存在，这个时候将分

离出SCRIPT_FILENAME和PATH_INFO分别为

/scripts/80sec.jpg和80sec.php

最后，以/scripts/80sec.jpg作为此次请求需要执行的脚本，攻击者就可以实现让nginx以php来解析任何类型的文件了。

POC：访问一个nginx来支持php的站点，在一个任何资源的文件如robots.txt后面加上/80sec.php，这个时候你可以看到如下的区别：

访问http://www.80sec.com/robots.txt

```
HTTP/1.1 200 OK
Server: nginx/0.6.32
Date: Thu, 20 May 2010 10:05:30 GMT
Content-Type: text/plain
Content-Length: 18
Last-Modified: Thu, 20 May 2010 06:26:34 GMT
Connection: keep-alive
Keep-Alive: timeout=20
Accept-Ranges: bytes
```

访问访问http://www.80sec.com/robots.txt/80sec.php

```
HTTP/1.1 200 OK
Server: nginx/0.6.32
Date: Thu, 20 May 2010 10:06:49 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive
Keep-Alive: timeout=20
X-Powered-By: PHP/5.2.6
```

其中的Content-Type的变化说明了后端负责解析的变化，该站点就可能存在漏洞。

漏洞厂商：http://www.nginx.org

解决方案：

我们已经尝试联系官方，但是此前你可以通过以下的方式来减少损失

关闭cgi.fix_pathinfo为0

或者

```
if ( $fastcgi_script_name ~ ..*/.*php ) {
return 403;
}
```

分类: Nginx



林锅
关注 - 20
粉丝 - 36
+加关注

1 0

« 上一篇：Apache2 三种MPM对比分析
» 下一篇：Nginx架构解析

posted on 2016-05-20 10:56 林锅 阅读(8670) 评论(2) 编辑 收藏

## 评论

### #1楼

历害了 我的哥，真是到了精通的地步，学习了

支持(0) 反对(0)

2018-04-10 19:49 | mfyang

### #2楼

使用mmap加速内核与用户空间的消息传递。
源码里面epoll没有使用mmap.
https://www.nowcoder.com/discuss/26226

支持(0) 反对(0)

2018-06-07 09:09 | jiesix

刷新评论 刷新页面 返回顶部

**注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。**



**最新IT新闻**:
· 花呗新广告有张魔性的人脸，它具体是怎么做出来的？
· 为何AI也学会了种族和性别歧视？
· FB高管桑德伯格MIT毕业演讲：你们肩负着改变世界的重任
· 中兴解困，华为受伤：关于中兴事件的深度解读
· Google再与军方合作，AI伦理的边界在哪里？
» 更多新闻...

**最新知识库文章**:
· 程序员的那些反模式
· 程序员的宇宙时间线
· 突破程序员思维
· 云、雾和霭计算如何一起工作
· 你可以把编程当做一项托付终身的职业
» 更多知识库文章...