[使用ngx\_lua构建高并发应用（1）](http://blog.csdn.net/chosen0ne/article/details/7304192)

标签： [lua](http://www.csdn.net/tag/lua)[nginx](http://www.csdn.net/tag/nginx)[access](http://www.csdn.net/tag/access)[io](http://www.csdn.net/tag/io)[memcached](http://www.csdn.net/tag/memcached)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)

2012-02-29 14:18 41611人阅读 [评论](http://blog.csdn.net/chosen0ne/article/details/7304192#comments)(5) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/chosen0ne/article/details/7304192#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

Nginx（14） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/chosen0ne/article/details/7304192)

**一. 概述**

        Nginx是一个高性能，支持高并发的，轻量级的web服务器。目前，Apache依然web服务器中的老大，但是在全球前1000大的web服务器中，Nginx的份额为22.4%。Nginx采用模块化的[**架构**](http://lib.csdn.net/base/architecture)，官方版本的Nginx中大部分功能都是通过模块方式提供的，比如Http模块、Mail模块等。通过开发模块扩展Nginx，可以将Nginx打造成一个全能的应用服务器，这样可以将一些功能在前端Nginx反向代理层解决，比如登录校验、js合并、甚至[**数据库**](http://lib.csdn.net/base/mysql)访问等等。

        但是，Nginx模块需要用C开发，而且必须符合一系列复杂的规则，最重要的用C开发模块必须要熟悉Nginx的源代码，使得开发者对其望而生畏。淘宝的agentzh和chaoslawful开发的ngx\_lua模块通过将lua解释器集成进Nginx，可以采用lua脚本实现业务逻辑，由于lua的紧凑、快速以及内建协程，所以在保证高并发服务能力的同时极大地降低了业务逻辑实现成本。

        本文向大家介绍ngx\_lua，以及我在使用它开发项目的过程中遇到的一些问题。

**二. 准备**

        首先，介绍一下Nginx的一些特性，便于后文介绍ngx\_lua的相关特性。

**1. Nginx进程模型**

        Nginx采用多进程模型，单Master—多Worker，由Master处理外部信号、配置文件的读取及Worker的初始化，Worker进程采用单线程、非阻塞的事件模型（Event Loop，事件循环）来实现端口的监听及客户端请求的处理和响应，同时Worker还要处理来自Master的信号。由于Worker使用单线程处理各种事件，所以一定要保证主循环是非阻塞的，否则会大大降低Worker的响应能力。

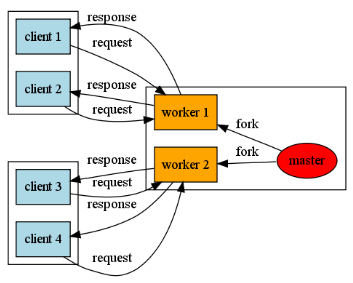


图1

**2. Nginx处理Http请求的过程**

        表面上看，当Nginx处理一个来自客户端的请求时，先根据请求头的host、ip和port来确定由哪个server处理，确定了server之后，再根据请求的uri找到对应的location，这个请求就由这个location处理。实际Nginx将一个请求的处理划分为若干个不同阶段（phase），这些阶段按照前后顺序依次执行，也就是说NGX\_HTTP\_POST\_READ\_PHASE在第一个，NGX\_HTTP\_LOG\_PHASE在最后一个。

**[cpp]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. NGX\_HTTP\_POST\_READ\_PHASE,     //0读取请求phase
2. NGX\_HTTP\_SERVER\_REWRITE\_PHASE,//1这个阶段主要是处理全局的(server block)的rewrite
3. NGX\_HTTP\_FIND\_CONFIG\_PHASE,   //2这个阶段主要是通过uri来查找对应的location，然后根据loc\_conf设置r的相应变量
4. NGX\_HTTP\_REWRITE\_PHASE,       //3这个主要处理location的rewrite
5. NGX\_HTTP\_POST\_REWRITE\_PHASE,  //4postrewrite，这个主要是进行一些校验以及收尾工作，以便于交给后面的模块。
6. NGX\_HTTP\_PREACCESS\_PHASE,     //5比如流控这种类型的access就放在这个phase，也就是说它主要是进行一些比较粗粒度的access。
7. NGX\_HTTP\_ACCESS\_PHASE,        //6这个比如存取控制，权限验证就放在这个phase，一般来说处理动作是交给下面的模块做的.这个主要是做一些细粒度的access
8. NGX\_HTTP\_POST\_ACCESS\_PHASE,   //7一般来说当上面的access模块得到access\_code之后就会由这个模块根据access\_code来进行操作
9. NGX\_HTTP\_TRY\_FILES\_PHASE,     //8try\_file模块，就是对应配置文件中的try\_files指令，可接收多个路径作为参数，当前一个路径的资源无法找到，则自动查找下一个路径
10. NGX\_HTTP\_CONTENT\_PHASE,       //9内容处理模块
11. NGX\_HTTP\_LOG\_PHASE            //10log模块

        每个阶段上可以注册handler，处理请求就是运行每个阶段上注册的handler。Nginx模块提供的配置指令只会一般只会注册并运行在其中的某一个处理阶段。比如，set指令属于rewrite模块的，运行在rewrite阶段，deny和allow运行在access阶段。

**3. 子请求（subrequest）**

        其实在Nginx 世界里有两种类型的“请求”，一种叫做“主请求”（main request），而另一种则叫做“子请求”（subrequest）。

        所谓“主请求”，就是由 HTTP 客户端从 Nginx 外部发起的请求。比如，从浏览器访问Nginx就是一个“主请求”。

        而“子请求”则是由 Nginx 正在处理的请求在 Nginx 内部发起的一种级联请求。“子请求”在外观上很像 HTTP 请求，但实现上却和 HTTP 协议乃至网络通信一点儿关系都没有。它是 Nginx 内部的一种抽象调用，目的是为了方便用户把“主请求”的任务分解为多个较小粒度的“内部请求”，并发或串行地访问多个 location 接口，然后由这些 location 接口通力协作，共同完成整个“主请求”。当然，“子请求”的概念是相对的，任何一个“子请求”也可以再发起更多的“子子请求”，甚至可以玩递归调用（即自己调用自己）。当一个请求发起一个“子请求”的时候，按照 Nginx 的术语，习惯把前者称为后者的“父请求”（parent request）。

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. location /main {
2. echo\_location /foo;     # echo\_location发送子请求到指定的location
3. echo\_location /bar;
4. }
5. location /foo {
6. echo foo;
7. }
8. location /bar {
9. echo bar;
10. }

       输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. $ curl location/main
2. $ foo
3. bar

这里，main location就是发送2个子请求，分别到foo和bar，这就类似一种函数调用。

        “子请求”方式的通信是在同一个虚拟主机内部进行的，所以 Nginx 核心在实现“子请求”的时候，就只调用了若干个 C 函数，完全不涉及任何网络或者 UNIX 套接字（socket）通信。我们由此可以看出“子请求”的执行效率是极高的。

**4. 协程（Coroutine）**

         协程类似一种多线程，与多线程的区别有：

        1. 协程并非os线程，所以创建、切换开销比线程相对要小。

        2. 协程与线程一样有自己的栈、局部变量等，但是协程的栈是在用户进程空间模拟的，所以创建、切换开销很小。

        3. 多线程程序是多个线程并发执行，也就是说在一瞬间有多个控制流在执行。而协程强调的是一种多个协程间协作的关系，只有当一个协程主动放弃执行权，另一个协程才能获得执行权，所以在某一瞬间，多个协程间只有一个在运行。

        4. 由于多个协程时只有一个在运行，所以对于临界区的访问不需要加锁，而多线程的情况则必须加锁。

        5. 多线程程序由于有多个控制流，所以程序的行为不可控，而多个协程的执行是由开发者定义的所以是可控的。

        Nginx的每个Worker进程都是在epoll或kqueue这样的事件模型之上，封装成协程，每个请求都有一个协程进行处理。这正好与Lua内建协程的模型是一致的，所以即使ngx\_lua需要执行Lua，相对C有一定的开销，但依然能保证高并发能力。

**三. ngx\_lua**

**1. 原理**

        ngx\_lua将Lua嵌入Nginx，可以让Nginx执行Lua脚本，并且高并发、非阻塞的处理各种请求。Lua内建协程，这样就可以很好的将异步回调转换成顺序调用的形式。ngx\_lua在Lua中进行的IO操作都会委托给Nginx的事件模型，从而实现非阻塞调用。开发者可以采用串行的方式编写程序，ngx\_lua会自动的在进行阻塞的IO操作时中断，保存上下文；然后将IO操作委托给Nginx事件处理机制，在IO操作完成后，ngx\_lua会恢复上下文，程序继续执行，这些操作都是对用户程序透明的。

        每个NginxWorker进程持有一个Lua解释器或者LuaJIT实例，被这个Worker处理的所有请求共享这个实例。每个请求的Context会被Lua轻量级的协程分割，从而保证各个请求是独立的。

        ngx\_lua采用“one-coroutine-per-request”的处理模型，对于每个用户请求，ngx\_lua会唤醒一个协程用于执行用户代码处理请求，当请求处理完成这个协程会被销毁。每个协程都有一个独立的全局环境（变量空间），继承于全局共享的、只读的“comman data”。所以，被用户代码注入全局空间的任何变量都不会影响其他请求的处理，并且这些变量在请求处理完成后会被释放，这样就保证所有的用户代码都运行在一个“sandbox”（沙箱），这个沙箱与请求具有相同的生命周期。

        得益于Lua协程的支持，ngx\_lua在处理10000个并发请求时只需要很少的内存。根据[**测试**](http://lib.csdn.net/base/softwaretest)，ngx\_lua处理每个请求只需要2KB的内存，如果使用LuaJIT则会更少。所以ngx\_lua非常适合用于实现可扩展的、高并发的服务。

**2. 典型应用**

        官网上列出：

·  Mashup'ing and processing outputs of various nginx upstream outputs(proxy, drizzle, postgres, [**Redis**](http://lib.csdn.net/base/redis), memcached, and etc) in Lua,

·  doing arbitrarily complex access control and security checks in Luabefore requests actually reach the upstream backends,

·  manipulating response headers in an arbitrary way (by Lua)

·  fetching backend information from external storage backends (likeredis, memcached, [**MySQL**](http://lib.csdn.net/base/mysql), postgresql) and use that information to choose whichupstream backend to access on-the-fly,

·  coding up arbitrarily complex web applications in a content handlerusing synchronous but still non-blocking access to the database backends andother storage,

·  doing very complex URL dispatch in Lua at rewrite phase,

·  using Lua to implement advanced caching mechanism for nginxsubrequests and arbitrary locations.

**3. Hello Lua！**

        配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. # nginx.conf
2. worker\_processes 4;
4. events {
5. worker\_connections 1024;
6. }
7. http {
9. server {
10. listen 80;
11. server\_name localhost;
13. location = /lua {
14. content\_by\_lua ‘
15. ngx.say("Hello, Lua!")
16. ';
17. }
18. }
19. }

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. $ curl 'localhost/lua'
2. Hello，Lua！

        这样就实现了一个很简单的ngx\_lua应用，如果这么简单的模块要是用C来开发的话，代码量估计得有100行左右，从这就可以看出ngx\_lua的开发效率。

**4. Benchmark**

        通过和nginx访问静态文件还有nodejs比较，来看一下ngx\_lua提供的高并发能力。

        返回的内容都是”Hello World!”，151bytes

        通过.ab -n 60000   取10次平均

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1000 | 3000 | 5000 | 7000 | 10000 |
| nginx 静态文件 | 11351 | 9653 | 8929 | 8997 | 9722 |
| nodejs | 10846 | 9510 | 8898 | 8387 | 7820 |
| ngx\_lua | 13839 | 10174 | 9523 | 10309 | 10711 |

        从图表中可以看到，在各种并发条件下ngx\_lua的rps都是最高的，并且基本维持在10000rps左右，nginx读取静态文件因为会有磁盘io所以性能略差一些，而nodejs是相对最差的。通过这个简单的测试，可以看出ngx\_lua的高并发能力。

        ngx\_lua的开发者也做过一个测试对比nginx+fpm+[**PHP**](http://lib.csdn.net/base/php)和nodejs，他得出的结果是ngx\_lua可以达到28000rps，而nodejs有10000多一点，php则最差只有6000。可能是有些配置我没有配好导致ngx\_lua rps没那么高。

**5. ngx\_lua安装**

        ngx\_lua安装可以通过下载模块源码，编译Nginx，但是推荐采用openresty。Openresty就是一个打包程序，包含大量的第三方Nginx模块，比如HttpLuaModule，HttpRedis2Module，HttpEchoModule等。省去下载模块，并且安装非常方便。

        ngx\_openresty bundle: [openresty](http://openresty.org/)

        ./configure --with-luajit&& make && make install

        默认Openresty中ngx\_lua模块采用的是标准的Lua5.1解释器，通过--with-luajit使用LuaJIT。

**6. ngx\_lua的用法**

        ngx\_lua模块提供了配置指令和Nginx API。

        配置指令：在Nginx中使用，和set指令和pass\_proxy指令使用方法一样，每个指令都有使用的context。

        Nginx API：用于在Lua脚本中访问Nginx变量，调用Nginx提供的函数。

        下面举例说明常见的指令和API。

**7. 配置指令**

**a. set\_by\_lua和set\_by\_lua\_file**

        和set指令一样用于设置Nginx变量并且在rewrite阶段执行，只不过这个变量是由lua脚本计算并返回的。

        语法：set\_by\_lua$res <lua-script-str> [$arg1 $arg2 ...]

        配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. location = /adder {
2. set\_by\_lua $res "
3. local a = tonumber(ngx.arg[1])
4. local b = tonumber(ngx.arg[2])
5. return a + b" $arg\_a $arg\_b;
7. echo $res;
8. }

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. $ curl 'localhost/adder?a=25&b=75'
2. $ 100

        set\_by\_lua\_file执行Nginx外部的lua脚本，可以避免在配置文件中使用大量的转义。

        配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. location = /fib {
2. set\_by\_lua\_file $res "conf/adder.lua" $arg\_n;
4. echo $res;
5. }

        adder.lua:

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. local a = tonumber(ngx.arg[1])
2. local b = tonumber(ngx.arg[2])
3. return a + b

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. $ curl 'localhost/adder?a=25&b=75
2. $ 100

**b. access\_by\_lua和access\_by\_lua\_file**

        运行在access阶段，用于访问控制。Nginx原生的allow和deny是基于ip的，通过access\_by\_lua能完成复杂的访问控制，比如，访问数据库进行用户名、密码验证等。

        配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. location /auth {
2. access\_by\_lua '
3. if ngx.var.arg\_user == "ntes" then
4. return
5. else
6. Ngx.exit(ngx.HTTP\_FORBIDDEN)
7. end
8. ';
9. echo 'welcome ntes';
10. }

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. $ curl 'localhost/auth?user=sohu'
2. $ Welcome ntes

5. $ curl 'localhost/auth?user=ntes'
6. $ <html>
7. <head><title>403 Forbidden</title></heda>
8. <body bgcolor="white">
9. <center><h1>403 Forbidden</h1></center>
10. <hr><center>ngx\_openresty/1.0.10.48</center>
11. </body>
12. </html>

**c. rewrite\_by\_lua和rewrite\_by\_lua\_file**

        实现url重写，在rewrite阶段执行。

        配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. location = /foo {
2. rewrite\_by\_lua 'ngx.exec("/bar")';
3. echo 'in foo';
4. }
6. location = /bar {
7. echo 'in bar';
8. }

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. $ curl 'localhost/lua'
2. $ Hello, Lua!

**d. content\_by\_lua和content\_by\_lua\_file**

        Contenthandler在content阶段执行，生成http响应。由于content阶段只能有一个handler，所以在与echo模块使用时，不能同时生效，我测试的结果是content\_by\_lua会覆盖echo。这和之前的hello world的例子是类似的。

        配置（直接响应）：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. location = /lua {
2. content\_by\_lua 'ngx.say("Hello, Lua!")';
3. }

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. $ curl 'localhost/lua'
2. $ Hello, Lua!

        配置（在Lua中访问Nginx变量）：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. location = /hello {
2. content\_by\_lua 'local who = ngx.var.arg\_who
3. ngx.say("Hello, ", who, "!")';
4. }

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. $ curl 'localhost/hello?who=world
2. $ Hello, world!

**8. Nginx API**

        Nginx API被封装ngx和ndk两个package中。比如ngx.var.NGX\_VAR\_NAME可以访问Nginx变量。这里着重介绍一下ngx.location.capture和ngx.location.capture\_multi。

**a. ngx.location.capture**

        语法：res= ngx.location.capture(uri, options?)

        用于发出一个同步的，非阻塞的Nginxsubrequest（子请求）。可以通过Nginx subrequest向其它location发出非阻塞的内部请求，这些location可以是配置用于读取文件夹的，也可以是其它的C模块，比如ngx\_proxy, ngx\_fastcgi, ngx\_memc, ngx\_postgres, ngx\_drizzle甚至是ngx\_lua自己。

        Subrequest只是模拟Http接口，并没有额外的Http或者Tcp传输开销，它在C层次上运行，非常高效。Subrequest不同于Http 301/302重定向，以及内部重定向（通过ngx.redirection）。

        配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. location = /other {
2. ehco 'Hello, world!';
3. }
5. # Lua非阻塞IO
6. location = /lua {
7. content\_by\_lua '
8. local res = ngx.location.capture("/other")
9. if res.status == 200 then
10. ngx.print(res.body)
11. end
12. ';
13. }

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. $ curl  'http://localhost/lua'
2. $ Hello, world!

        实际上，location可以被外部的Http请求调用，也可以被内部的子请求调用。每个location相当于一个函数，而发送子请求就类似于函数调用，而且这种调用是非阻塞的，这就构造了一个非常强大的变成模型，后面我们会看到如何通过location和后端的memcached、redis进行非阻塞通信。

**b. ngx.location.capture\_multi**

        语法：res1,res2, ... = ngx.location.capture\_multi({ {uri, options?}, {uri, options?}, ...})

        与ngx.location.capture功能一样，可以并行的、非阻塞的发出多个子请求。这个方法在所有子请求处理完成后返回，并且整个方法的运行时间取决于运行时间最长的子请求，并不是所有子请求的运行时间之和。

        配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. # 同时发送多个子请求（subrequest）
2. location = /moon {
3. ehco 'moon';
4. }
5. location = /earth {
6. ehco 'earth';
7. }
9. location = /lua {
10. content\_by\_lua '
11. local res1,res2 = ngx.location.capture\_multi({ {"/moon"}, {"earth"} })
12. if res1.status == 200 then
13. ngx.print(res1.body)
14. end
15. ngx.print(",")
16. if res2.status == 200 then
17. ngx.print(res2.body)
18. end
19. ';
20. }

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. $ curl  'http://localhost/lua'
2. $ moon,earth

**c. 注意**

        在Lua代码中的网络IO操作只能通过Nginx Lua API完成，如果通过标准Lua API会导致Nginx的事件循环被阻塞，这样性能会急剧下降。

        在进行数据量相当小的磁盘IO时可以采用标准Lua io库，但是当读写大文件时这样是不行的，因为会阻塞整个NginxWorker进程。为了获得更大的性能，强烈建议将所有的网络IO和磁盘IO委托给Nginx子请求完成（通过ngx.location.capture）。

        下面通过访问/html/index.html这个文件，来测试将磁盘IO委托给Nginx和通过Lua io直接访问的效率。

        通过ngx.location.capture委托磁盘IO：

        配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. location / {
2. internal;
3. root html;
4. }
6. location /capture {
7. content\_by\_lua '
8. res = ngx.location.capture("/")
9. echo res.body
10. ';
11. }

        通过标准lua io访问磁盘文件：

        配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7304192) [copy](http://blog.csdn.net/chosen0ne/article/details/7304192)

[print?](http://blog.csdn.net/chosen0ne/article/details/7304192)

1. location /luaio{
2. content\_by\_lua '
3. local io = require("io")
4. local chunk\_SIZE = 4096
5. local f = assert(io.open("html/index.html","r"))
6. while true do
7. local chunk = f:read(chunk)
8. if not chunk then
9. break
10. end
11. ngx.print(chunk)
12. ngx.flush(true)
13. end
14. f:close()
15. ';
16. }

        这里通过ab去压，在各种并发条件下，分别返回151bytes、151000bytes的数据，取10次平均，得到两种方式的rps。

        静态文件：151bytes

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1000 | 3000 | 5000 | 7000 | 10000 |
| capture | 11067 | 8880 | 8873 | 8952 | 9023 |
| Lua io | 11379 | 9724 | 8938 | 9705 | 9561 |

        静态文件：151000bytes，在10000并发下内存占用情况太严重，测不出结果        这种情况下，文件较小，通过Nginx访问静态文件需要额外的系统调用，性能略逊于ngx\_lua。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1000 | 3000 | 5000 | 7000 | 10000 |
| capture | 3338 | 3435 | 3178 | 3043 | / |
| Lua io | 3174 | 3094 | 3081 | 2916 | / |

        在大文件的情况，capture就要略好于ngx\_lua。

        这里没有对Nginx读取静态文件进行优化配置，只是采用了sendfile。如果优化一下，可能nginx读取静态文件的性能会更好一些，这个目前还不熟悉。所以，在Lua中进行各种IO时，都要通过ngx.location.capture发送子请求委托给Nginx事件模型，这样可以保证IO是非阻塞的。

**四. 小结**

        这篇文章简单介绍了一下ngx\_lua的基本用法，后一篇会对ngx\_lua访问redis、memcached已经连接池进行详细介绍。

        注： 关于子请求的描述摘自agentzh的博客： <http://blog.sina.com.cn/s/blog_6d579ff40100wqn7.html>

                 进程模型的图摘自Joshua Zhu的ppt： <http://www.slideshare.net/joshzhu/nginx-internals>

# [使用ngx\_lua构建高并发应用（2）](http://blog.csdn.net/chosen0ne/article/details/7309316)

标签： [lua](http://www.csdn.net/tag/lua)[redis](http://www.csdn.net/tag/redis)[memcached](http://www.csdn.net/tag/memcached)[nginx](http://www.csdn.net/tag/nginx)[server](http://www.csdn.net/tag/server)[query](http://www.csdn.net/tag/query)

2012-03-01 13:17 11935人阅读 [评论](http://blog.csdn.net/chosen0ne/article/details/7309316#comments)(2) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/chosen0ne/article/details/7309316#report)

本文章已收录于：

[[](http://lib.csdn.net/base/34?source=blogtop) **Redis知识库**](http://lib.csdn.net/base/34?source=blogtop)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

Nginx（14） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/chosen0ne/article/details/7309316)

        在[之前的文章](http://blog.csdn.net/chosen0ne/article/details/7304192)中，已经介绍了ngx\_lua的一些基本介绍，这篇文章主要着重讨论一下如何通过ngx\_lua同后端的memcached、[**Redis**](http://lib.csdn.net/base/redis)进行非阻塞通信。

#### 1. Memcached

        在Nginx中访问Memcached需要模块的支持，这里选用[HttpMemcModule](http://wiki.nginx.org/HttpMemcModule)，这个模块可以与后端的Memcached进行非阻塞的通信。我们知道官方提供了[Memcached](http://wiki.nginx.org/HttpMemcachedModule)，这个模块只支持get操作，而Memc支持大部分Memcached的命令。

        Memc模块采用入口变量作为参数进行传递，所有以$memc\_为前缀的变量都是Memc的入口变量。memc\_pass指向后端的Memcached Server。

  配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7309316) [copy](http://blog.csdn.net/chosen0ne/article/details/7309316)

[print?](http://blog.csdn.net/chosen0ne/article/details/7309316)

1. #使用HttpMemcModule
2. location = /memc {
3. set $memc\_cmd $arg\_cmd;
4. set $memc\_key  $arg\_key;
5. set $memc\_value $arg\_val;
6. set $memc\_exptime $arg\_exptime;
8. memc\_pass '127.0.0.1:11211';
9. }

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7309316) [copy](http://blog.csdn.net/chosen0ne/article/details/7309316)

[print?](http://blog.csdn.net/chosen0ne/article/details/7309316)

1. $ curl  'http://localhost/memc?cmd=set&key=foo&val=Hello'
2. $ STORED
3. $ curl  'http://localhost/memc?cmd=get&key=foo'
4. $ Hello

        这就实现了memcached的访问，下面看一下如何在lua中访问memcached。

  配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7309316) [copy](http://blog.csdn.net/chosen0ne/article/details/7309316)

[print?](http://blog.csdn.net/chosen0ne/article/details/7309316)

1. #在Lua中访问Memcached
2. location = /memc {
3. internal;   #只能内部访问
4. set $memc\_cmd get;
5. set $memc\_key  $arg\_key;
6. memc\_pass '127.0.0.1:11211';
7. }
8. location = /lua\_memc {
9. content\_by\_lua '
10. local res = ngx.location.capture("/memc", {
11. args = { key = ngx.var.arg\_key }
12. })
13. if res.status == 200 then
14. ngx.say(res.body)
15. end
16. ';
17. }

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7309316) [copy](http://blog.csdn.net/chosen0ne/article/details/7309316)

[print?](http://blog.csdn.net/chosen0ne/article/details/7309316)

1. $ curl  'http://localhost/lua\_memc?key=foo'
2. $ Hello

        通过lua访问memcached，主要是通过子请求采用一种类似函数调用的方式实现。首先，定义了一个memc location用于通过后端memcached通信，就相当于memcached storage。由于整个Memc模块时非阻塞的，ngx.location.capture也是非阻塞的，所以整个操作非阻塞。

#### 2. Redis

        访问redis需要[HttpRedis2Module](http://wiki.nginx.org/HttpRedis2Module)的支持，它也可以同redis进行非阻塞通行。不过，redis2的响应是redis的原生响应，所以在lua中使用时，需要解析这个响应。可以采用[LuaRedisModule](http://wiki.nginx.org/LuaRedisParser)，这个模块可以构建redis的原生请求，并解析redis的原生响应。

        配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7309316) [copy](http://blog.csdn.net/chosen0ne/article/details/7309316)

[print?](http://blog.csdn.net/chosen0ne/article/details/7309316)

1. #在Lua中访问Redis
2. location = /redis {
3. internal;   #只能内部访问
4. redis2\_query get $arg\_key;
5. redis2\_pass '127.0.0.1:6379';
6. }
7. location = /lua\_redis { #需要LuaRedisParser
8. content\_by\_lua '
9. local parser = require("redis.parser")
10. local res = ngx.location.capture("/redis", {
11. args = { key = ngx.var.arg\_key }
12. })
13. if res.status == 200 then
14. reply = parser.parse\_reply(res.body)
15. ngx.say(reply)
16. end
17. ';
18. }

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7309316) [copy](http://blog.csdn.net/chosen0ne/article/details/7309316)

[print?](http://blog.csdn.net/chosen0ne/article/details/7309316)

1. $ curl  'http://localhost/lua\_redis?key=foo'
2. $ Hello

        和访问memcached类似，需要提供一个redis storage专门用于查询redis，然后通过子请求去调用redis。

#### 3. Redis Pipeline

        在实际访问redis时，有可能需要同时查询多个key的情况。我们可以采用ngx.location.capture\_multi通过发送多个子请求给redis storage，然后在解析响应内容。但是，这会有个限制，Nginx内核规定一次可以发起的子请求的个数不能超过50个，所以在key个数多于50时，这种方案不再适用。

        幸好redis提供pipeline机制，可以在一次连接中执行多个命令，这样可以减少多次执行命令的往返时延。客户端在通过pipeline发送多个命令后，redis顺序接收这些命令并执行，然后按照顺序把命令的结果输出出去。在lua中使用pipeline需要用到redis2模块的redis2\_raw\_queries进行redis的原生请求查询。

        配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7309316) [copy](http://blog.csdn.net/chosen0ne/article/details/7309316)

[print?](http://blog.csdn.net/chosen0ne/article/details/7309316)

1. #在Lua中访问Redis
2. location = /redis {
3. internal;   #只能内部访问
5. redis2\_raw\_queries $args $echo\_request\_body;
6. redis2\_pass '127.0.0.1:6379';
7. }
9. location = /pipeline {
10. content\_by\_lua 'conf/pipeline.lua';
11. }

        pipeline.lua

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7309316) [copy](http://blog.csdn.net/chosen0ne/article/details/7309316)

[print?](http://blog.csdn.net/chosen0ne/article/details/7309316)

1. -- conf/pipeline.lua file
2. local parser = require(‘redis.parser’)
3. local reqs = {
4. {‘get’, ‘one’}, {‘get’, ‘two’}
5. }
6. -- 构造原生的redis查询，get one\r\nget two\r\n
7. local raw\_reqs = {}
8. for i, req in ipairs(reqs)  do
9. table.insert(raw\_reqs, parser.build\_query(req))
10. end
11. local res = ngx.location.capture(‘/redis?’..#reqs, { body = table.concat(raw\_reqs, ‘’) })
13. if res.status and res.body then
14. -- 解析redis的原生响应
15. local replies = parser.parse\_replies(res.body, #reqs)
16. for i, reply in ipairs(replies)  do
17. ngx.say(reply[1])
18. end
19. end

        输出：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7309316) [copy](http://blog.csdn.net/chosen0ne/article/details/7309316)

[print?](http://blog.csdn.net/chosen0ne/article/details/7309316)

1. $ curl  'http://localhost/pipeline'
2. $ first
3. second

#### 4. Connection Pool

        前面访问redis和memcached的例子中，在每次处理一个请求时，都会和后端的server建立连接，然后在请求处理完之后这个连接就会被释放。这个过程中，会有3次握手、timewait等一些开销，这对于高并发的应用是不可容忍的。这里引入connection pool来消除这个开销。

        连接池需要[HttpUpstreamKeepaliveModule](http://wiki.nginx.org/HttpUpstreamKeepaliveModule)模块的支持。

        配置：

**[plain]** [view plain](http://blog.csdn.net/chosen0ne/article/details/7309316) [copy](http://blog.csdn.net/chosen0ne/article/details/7309316)

[print?](http://blog.csdn.net/chosen0ne/article/details/7309316)

1. http {
2. # 需要HttpUpstreamKeepaliveModule
3. upstream redis\_pool {
4. server 127.0.0.1:6379;
5. # 可以容纳1024个连接的连接池
6. keepalive 1024 single;
7. }
9. server {
10. location = /redis {
11. …
12. redis2\_pass redis\_pool;
13. }
14. }
15. }

        这个模块提供keepalive指令，它的context是upstream。我们知道upstream在使用Nginx做反向代理时使用，实际upstream是指“上游”，这个“上游”可以是redis、memcached或是mysql等一些server。upstream可以定义一个虚拟server集群，并且这些后端的server可以享受负载均衡。keepalive 1024就是定义连接池的大小，当连接数超过这个大小后，后续的连接自动退化为短连接。连接池的使用很简单，直接替换掉原来的ip和端口号即可。

        有人曾经测过，在没有使用连接池的情况下，访问memcached（使用之前的Memc模块），rps为20000。在使用连接池之后，rps一路飙到140000。在实际情况下，这么大的提升可能达不到，但是基本上100-200%的提高还是可以的。

#### 5. 小结

        这里对memcached、redis的访问做个小结。

        1. Nginx提供了强大的编程模型，location相当于函数，子请求相当于函数调用，并且location还可以向自己发送子请求，这样构成一个递归的模型，所以采用这种模型实现复杂的业务逻辑。

        2. Nginx的IO操作必须是非阻塞的，如果Nginx在那阻着，则会大大降低Nginx的性能。所以在Lua中必须通过ngx.location.capture发出子请求将这些IO操作委托给Nginx的事件模型。

        3. 在需要使用tcp连接时，尽量使用连接池。这样可以消除大量的建立、释放连接的开销。