ngx_lua 内幕

王晓哲 (清无)[1]

2012-06-30

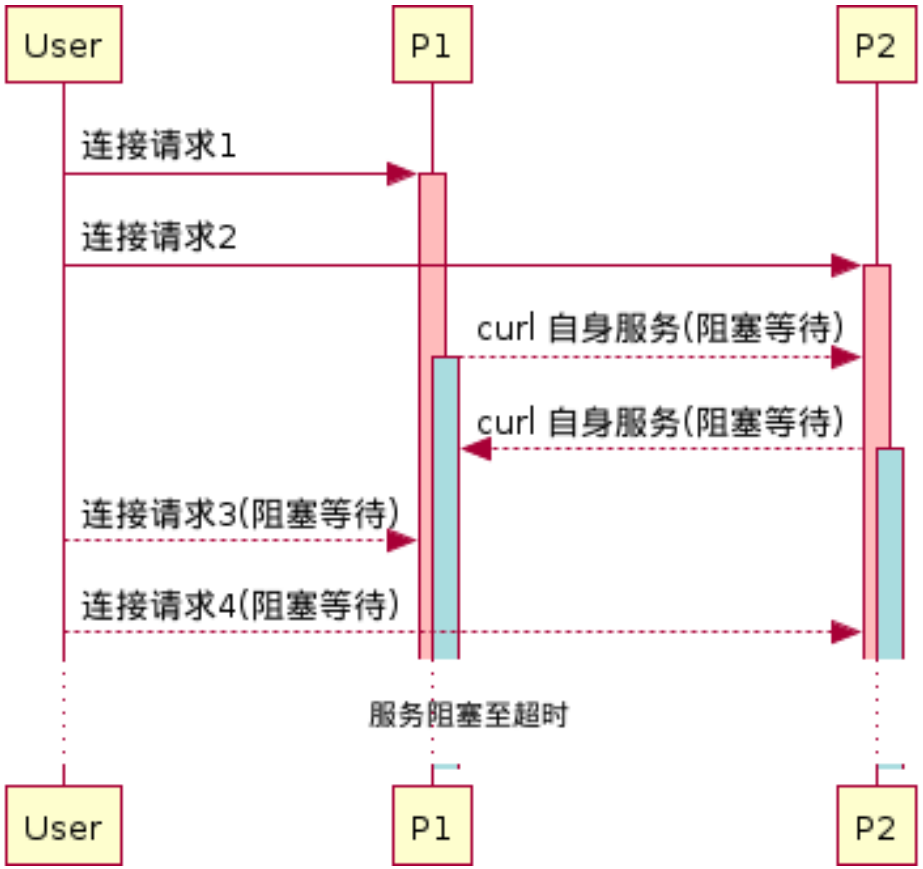
[1] chaoslawful@gmail.com, @chaoslawful

Outline

一个实际场景

Apache+mod_php 一直用的很爽，直到膝盖中了一箭：

- PHP 代码用 curl 访问自己的服务，在生产环境中经常阻塞直至客户端超时



本质问题

- 每连接一进程 (线程) 服务模型的并发天花板太低:例如 Apache prefork/worker
  - 进程/线程数量即最大并发数
  - 随进程/线程数量增加,上下文切换/缓存污染的耗时占比快速上升
  - 每进程 (线程) 的固定内存开销较大:生产环境中每个 Apache+mod_php 进程的 RSS 至少 5 MB
  - 慢连接攻击安全隐患
- 每请求一进程 (线程) 服务模型的并发天花板没有多少提升:例如反向代理 +FastCGI、HSHA/LF 模式
  - 仅消除了慢连接攻击隐患
- I/O 多路复用 + 显式状态机模型可大幅度提升并发能力上限:例如 Nginx C 模块,但
  - 开发复杂业务逻辑成本很高,状态数同 I/O 操作数量成正比
  - 调试、维护均比较麻烦

问题关键词

并发度低、内存开销大、开发成本高、难维护

— ngx_lua 要解决这些问题！

Why Lua?

- 内存开销小
  - 编译后完整功能 VM 尺寸 <100 KB
  - 数据结构附加开销小
    - 每个唯一字符串 ~16 B
    - 每个闭包 ~24 B
    - LuaJIT 附加开销更小

Why Lua?

- 运行效率高
  - 标准 Lua 快于 Python、Ruby、Perl 和 PHP
  - 对于 CPU 密集型程序，LuaJIT 速度同 C++/Java 在同一量级

实测结果 (数值为耗时，越低越好)[2, 3]：

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

[2]测试环境：i7 3.9 GHz，Lua 5.1.4，LuaJIT 2.0b10，OpenJDK6，PHP 5.3.10

[3]源码在 https://github.com/chaoslawful/shootout

Why Lua?

- 可中断/重入的 VM
  - 原生支持协程 (coroutine)
  - 依靠协程支持来中断/重入 VM 无需操心现场保护问题，切换开销很小
  - I/O 密集型应用的福音，稍作利用即可避免 I/O 阻塞等待浪费进程 (线程) 资源

ngx_lua 设计指导思想

- 基于 Nginx 快速开发高性能、大并发的网络服务
- 提供“同步非阻塞”的 I/O 访问接口简化 I/O 多路复用体系中的业务逻辑开发：
  - “同步”的主体是用户代码与其发起的 I/O 请求处理流程之间的时序关系，意即 I/O 请求处理完成前用户代码将一直挂起
  - “非阻塞”的主体是服务进程，意即 I/O 请求的处理不会导致服务进程阻塞等待，而是可以继续处理其他请求的用户逻辑

ngx_lua 协程化实现思路

- 每 worker 一个 Lua VM 实例，worker 上的每个请求由独立的协程处理，所有协程共享 VM
- Nginx I/O 操作封装为原语注入 Lua VM，允许用户代码直接访问
- 请求处理时发出的 I/O 操作若无法立刻完成，就打断相关协程的运行，待 I/O 操作完成事件触发后再恢复其运行
- 协程中运行的用户代码互相隔离，某个请求的代码异常不影响其他请求的处理过程

最终效果：实现 Proactor 模式，以自然顺序书写业务逻辑，自动获得高并发服务能力！

测试环境

- CPU Intel i7 3.9 GHz
- Mem 16 GB
- 全部使用 4 工作线程

CPU 密集型测试用例 - ngx_lua

Nginx 配置如下：

```
worker_processes 4;
events { worker_connections 4096; }
location /cpu_intense {
   default_type "text/plain";
   content_by_file html/cpu_intense.lua;
}
```

Lua 代码如下：

```
-- cpu_intense.lua
local random = math.random
local say = ngx.say
local n = tonumber(ngx.var.arg_n)
local dat = {}
math.randomseed(os.time())
for i=1, n do
   dat[i] = random(0, 10000)
end
table.sort(dat)
local sum = 0
for i=1, n do
   sum = sum + dat[i]
end
say("sum: " .. sum .. ", cnt: " .. n .. ", avg: " .. sum/n)
```

CPU 密集型测试用例 - Node.js

JS 代码如下：

```
var cluster = require('cluster'),
      http = require('http');
var numCPUs = 4;

if (cluster.isMaster) {
      for(var i = 0; i < numCPUs; i++) {
            cluster.fork();
      }
} else {
      http.createServer(function (req, res) {
            var args = require('url').parse(req.url, true);
            var n = args.query.n;
            var dat = new Array();
            for(i=0; i<n; i++) {
                  dat[i] = Math.floor(Math.random()*10000);
            }
            dat.sort();
            var sum = 0;
            for(i=0; i<n; i++) {
                  sum += dat[i]
            }
            res.writeHead(200, {'-ContentType': 'text/plain'});
            res.end('sum: ' + sum + ', cnt: ' + n + ', avg: ' + sum/n);
      }).listen(8080);
}
```

CPU 密集型测试用例 - HipHop

HipHop 配置如下：

```
Server {
      ThreadCount = 4
}
MySQL {
      ConnectTimeout = 1000
      SlowQueryThreshold = 60000
      ReadTimeout = 60000
}
```

PHP 代码如下：

```
<?php
$n = $_GET['n'];
$dat = array();
srand();
for($i = 0; $i < $n; $i++) {
   $dat[] = rand(0, 10000);
}
sort($dat);
$sum = 0;
for($i = 0; $i < $n; $i++) {
   $sum += $dat[$i];
}
echo "sum: ", $sum, ", cnt: ", $n, ", avg: ", $sum/$n;
?>
```

CPU 密集型应用测试结果 - ngx_lua

测试参数 n 取 10000

并发数	总 QPS(1/s)	99% 响应时间 (ms)
1	193	5
7	768	9
15	762	19
30	740	40
60	748	80
125	756	165
250	754	331
500	751	665
1000	773	1292

CPU 密集型应用测试结果 - Node.js

测试参数 n 取 10000

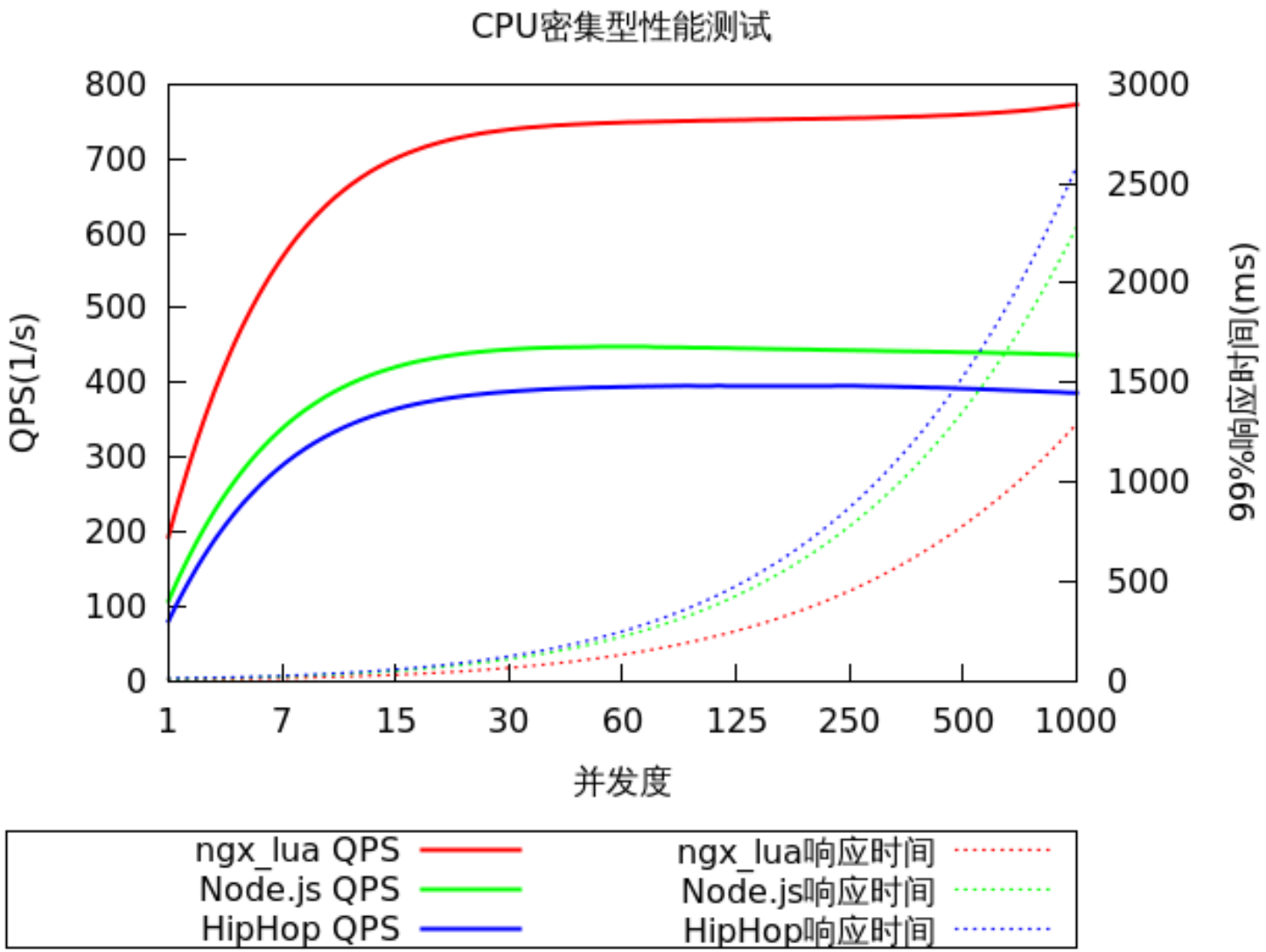
并发数	总 QPS(1/s)	99% 响应时间 (ms)
1	107	9
7	463	15
15	454	33
30	453	66
60	449	133
125	446	280
250	442	565
500	442	1128
1000	437	2283

CPU 密集型应用测试结果 - HipHop

测试参数 n 取 10000

并发数	总 QPS(1/s)	99% 响应时间 (ms)
1	80	12
7	400	17
15	394	38
30	396	75
60	390	153
125	400	311
250	400	619
500	392	1275
1000	386	2588

CPU 密集型应用测试结果总览



I/O 密集型测试用例 - ngx_lua

Nginx 配置如下：

```
worker_processes 4;
events { worker_connections 4096; }
location /io_intense {
   default_type "text/plain";
   content_by_file html/io_intense.lua;
}
```

Lua 代码如下：

```
local mysql = require 'resty.mysql'
local db = mysql:new()
db:set_timeout(2000)
local ok, err, errno, sqlstate = db:connect{
      host = "...", port = 3306,
      database = "test",
      user = "...", password = "...",
}
if not ok then
   ngx.exit(500)
end
local res, err, errno, sqlstate = db:query("select sleep(1)")
if not res then
   ngx.exit(500)
end
say("ok")
```

I/O 密集型测试用例 - Node.js

JS 代码如下：

```
var cluster = require('cluster'),
      http = require('http'),
      mysql = require('mysql');
var numCPUs = 4;

if (cluster.isMaster) {
      for(var i = 0; i < numCPUs; i++) {
            cluster.fork();
      }
} else {
      http.createServer(function (req, res) {
            var client = mysql.createClient({
                  host: '127.0.0.1',
                  user: 'wxz',
                  password: '',
            });

            client.query('select sleep(1)', function(err, rows, fields) {
                  if(err) throw err;
                  res.writeHead(200, {'-ContentType': 'text/plain'});
                  res.end('ok');
                  client.end();
            });
      }).listen(8080);
}
```

I/O 密集型测试用例 - HipHop

HipHop 配置如下：

```
Server {
      ThreadCount = 4
}
MySQL {
      ConnectTimeout = 1000
      SlowQueryThreshold = 60000
      ReadTimeout = 60000
}
```

PHP 代码如下：

```
<?php
$host = "...";
$user = "...";
$pass = "...";
$db = mysql_connect($host, $user, $pass)
   or die ('Could not connect: ' . mysql_error());
$res = mysql_query("select sleep(1)")
   or die("Query failed: " . mysql_error());
mysql_free_result($res);
mysql_close($db);
?>
```

I/O 密集型应用测试结果 - ngx_lua

并发数	总 QPS(1/s)	99% 响应时间 (ms)
1	1	1023
7	7	1060
15	15	1005
30	27	1091
60	56	1071
125	111	1125
250	214	1167
500	390	1280
1000	605	1651

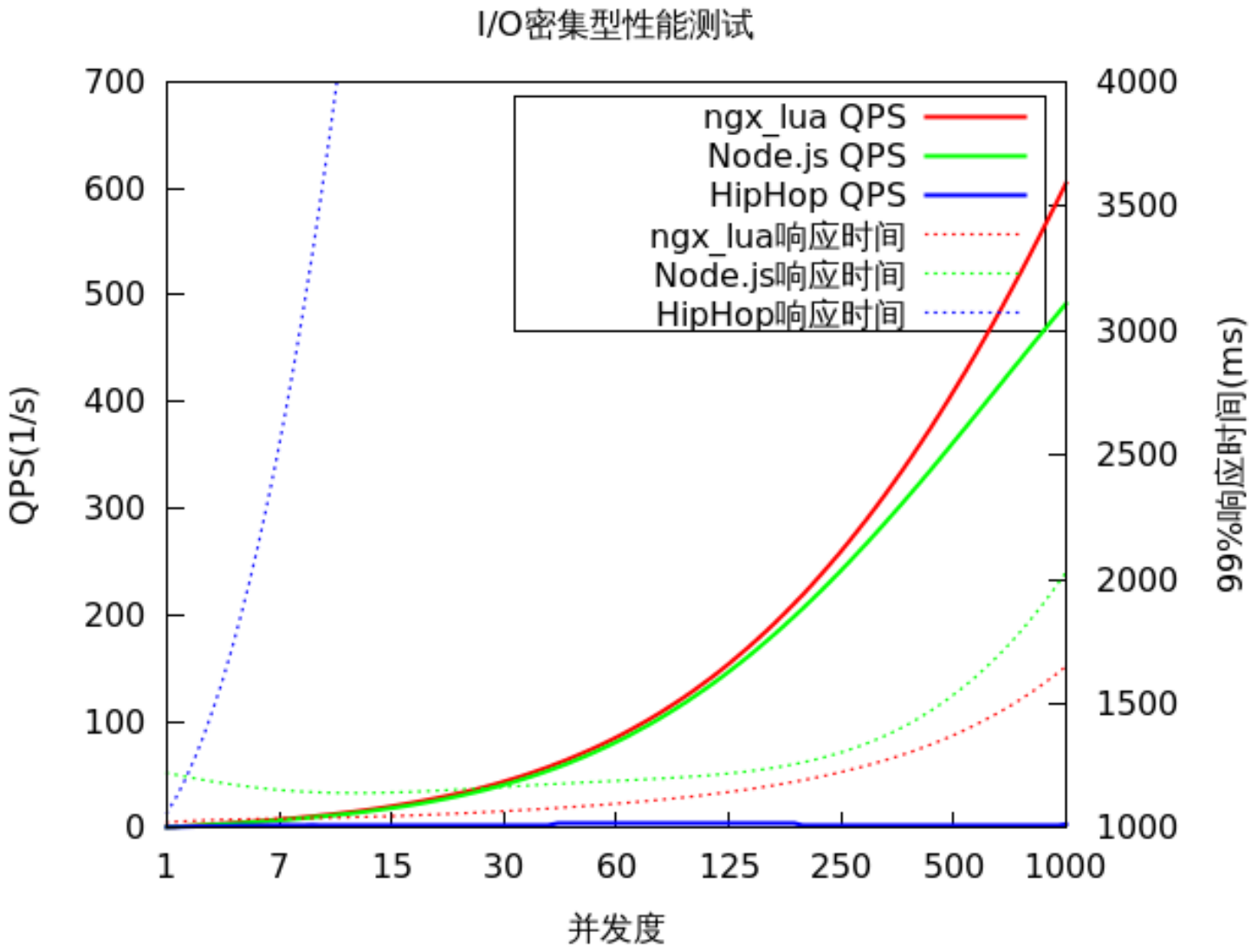
I/O 密集型应用测试结果 - Node.js

并发数	总 QPS(1/s)	99% 响应时间 (ms)
1	1	1222
7	6	1133
15	14	1060
30	25	1194
60	48	1256
125	107	1162
250	218	1142
500	366	1364
1000	492	2030

I/O 密集型应用测试结果 - HipHop

并发数	总 QPS(1/s)	99% 响应时间 (ms)
1	0.95	1061
7	3.98	1760
15	3.57	4203
30	3.65	8208
60	3.83	15672
125	3.98	31433
250	3.65	68496
500	3.41	146515
1000	3.74	267069

I/O 密集型应用测试结果总览



ngx_lua 的优势、劣势

优势：

- 同步非阻塞 I/O 形式直观易懂，并发服务能力强
- CPU、内存运行开销低
- 同 Nginx 结合度高，可方便粘合现有 Nginx 模块功能

劣势：

- 属于新技术方案，Lua 相比于 PHP、Ruby 等广泛使用的开发语言，周边附属设施尚不够健全，需要时间积累

ngx_lua 适用场景

1. 网络 I/O 阻塞时间远高于 CPU 计算占用时间、同时上游资源非瓶颈(可伸缩)的网络应用，如高性能网络中间层、HTTP REST 接口服务等；
2. 期望简化系统架构，让服务向 Nginx 同质化的 Web 站点；

参考

- http://wiki.nginx.org/HttpLuaModuleZh
- http://wiki.nginx.org/HttpLuaModule
- https://github.com/chaoslawful/lua-nginx-module
- https://github.com/agentzh/lua-resty-redis
- https://github.com/agentzh/lua-resty-mysql
- https://github.com/agentzh/lua-resty-memcached

That's all

Thanks!

Q&A