

目 录

致谢

入门篇

Socket 编程发展

OpenResty 简介

Lua 入门

Lua 简介

Lua 环境搭建

基础数据类型

表达式

控制结构

if/else

while

repeat

for

break , return 和 goto

Lua函数

函数的定义

函数的参数

函数返回值

全动态函数调用

模块

String 库

Table 库

日期时间函数

数学库函数

文件操作

Nginx

Nginx 新手起步

location 匹配规则

静态文件服务

日志

反向代理

负载均衡

[陷阱和常见错误](#)

[LuaCjsonLibrary](#)

[json解析的异常捕获](#)

[稀疏数组](#)

[空table编码为array还是object](#)

[PostgresNginxModule](#)

[调用方式简介](#)

[不支持事务](#)

[超时](#)

[健康监测](#)

[SQL注入](#)

[LuaNginxModule](#)

[执行阶段概念](#)

[正确的记录日志](#)

[热装载代码](#)

[阻塞操作](#)

[缓存](#)

[sleep](#)

[定时任务](#)

[禁止某些终端访问](#)

[请求返回后继续执行](#)

[调试](#)

[请求中断后的处理](#)

[我的 lua 代码需要调优么](#)

[变量的共享范围](#)

[动态限速](#)

[shared.dict 非队列性质](#)

[正确使用长链接](#)

[如何引用第三方 resty 库](#)

[典型应用场景](#)

[怎样理解 cosocket](#)

[如何安全启动唯一实例的 timer](#)

[如何正确的解析域名](#)

[LuaRestyDNSLibrary](#)

[使用动态 DNS 来完成 HTTP 请求](#)

[LuaRestyLock](#)

[缓存失效风暴](#)

[测试](#)

[代码静态分析](#)

[单元测试](#)

[代码覆盖率](#)

[API 测试](#)

[性能测试](#)

[持续集成](#)

[灰度发布](#)

[Web 服务](#)

[API的设计](#)

[数据合法性检测](#)

[协议无痛升级](#)

[代码规范](#)

[连接池](#)

[C10K 编程](#)

[TIME_WAIT 问题](#)

[与 Docker 使用的网络瓶颈](#)

[火焰图](#)

[什么是火焰图](#)

[什么时候使用](#)

[如何安装火焰图生成工具](#)

[如何定位问题](#)

致谢

当前文档《OpenResty 最佳实践》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-02-21。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN)，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/openresty-best-practices>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

入门篇

- [OpenResty 最佳实践](#)

OpenResty 最佳实践

在 2012 年的时候，加入到奇虎 360 公司，为新的产品做技术选型。由于之前一直混迹在 Python 圈子里面，也接触过 Nginx C 模块的高性能开发，一直想找到一个兼备 Python 快速开发和 Nginx C 模块高性能的产品。看到 OpenResty 后，有发现新大陆的感觉。

于是在新产品里面力推 OpenResty，团队里面几乎没人支持，经过几轮性能测试，虽然轻松击败所有的其他方案，但是其他开发人员并不愿意参与到基于 OpenResty 这个“陌生”框架的开发中来。于是我开始了一个人的 OpenResty 之旅，刚开始经历了各种技术挑战，庆幸有详细的文档，以及春哥和邮件列表里面热情的帮助，成了团队里面 bug 最少和几乎不用加班的同学。

2014 年，团队进来了一批新鲜血液，很有技术品味，先后选择 OpenResty 来作为技术方向。不再是一个人在战斗，而另外一个新问题摆在团队面前，如何保证大家都能写出高质量的代码，都能对 OpenResty 有深入的了解？知识的沉淀和升华，成为一个迫在眉睫的问题。

我们选择把这几年的一些浅薄甚至可能是错误的实践，通过 gitbook 的方式公开出来，一方面有利于团队自身的技术积累，另一方面，也能让更多的高手一起加入，让 OpenResty 的使用变得更加简单，更多的应用到服务端开发中，毕竟人生苦短，少一些加班，多一些陪家人。

这本书的定位是最佳实践，同时会对 OpenResty 做简单的基础介绍。但是我们对初学者的建议是，在看书的同时下载并安装 OpenResty，把[官方网站](#)的 Presentations 浏览和实践几遍。

请 一直 使用最新的 OpenResty 版本来运行本书的代码。

希望你能 enjoy OpenResty 之旅！

点我看书

本书源码在 GitHub 上维护，欢迎参与：[我要写书](#)。也可以加入 QQ 群来和我们交流：

- 34782325（技术交流 ①群 已满）
- 481213820（技术交流 ②群 已满）
- 124613000（技术交流 ③群）
- 679145170（技术交流 ④群）

Socket 编程发展

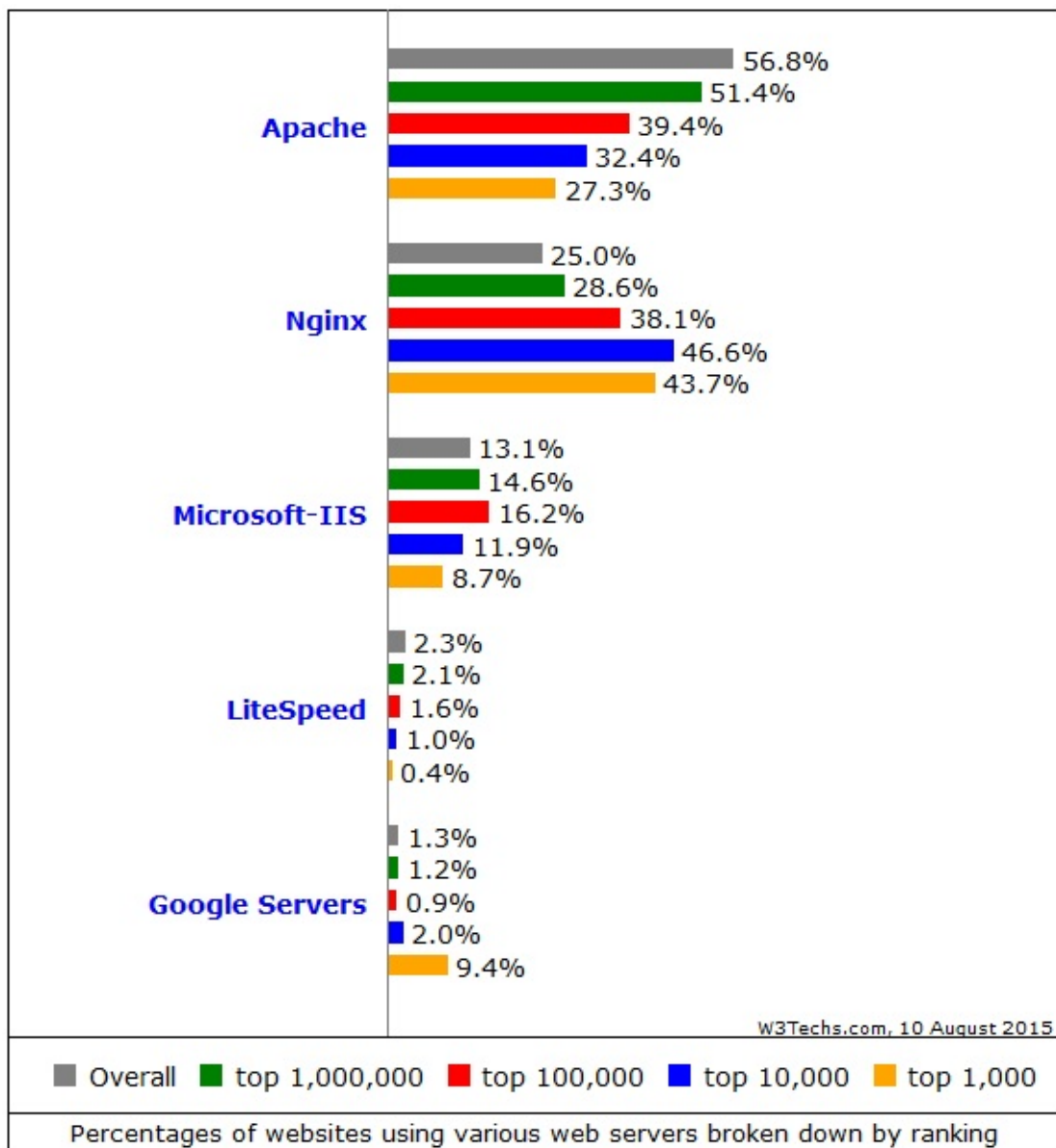
- [Socket 编程发展](#)
 - [select 模型](#)
 - [poll 模型](#)
 - [epoll 模型](#)

Socket 编程发展

Linux Socket 编程领域，为了处理大量连接请求场景，需要使用非阻塞 I/O 和复用。select、poll 和 epoll 是 Linux API 提供的 I/O 复用方式，自从 Linux 2.6 中加入了 epoll 之后，高性能服务器领域得到广泛的应用，现在比较出名的 Nginx 就是使用 epoll 来实现 I/O 复用支持高并发，目前在高并发的场景下，Nginx 越来越受到欢迎。

据 w3techs 在 2015 年 8 月 10 日的统计数据表明，在全球 Top 1000 的网站中，有 43.7% 的网站在使用 Nginx，这使得 Nginx 超越了 Apache，成为了高流量网站最信任的 Web 服务器足足有两年时间。已经确定在使用 Nginx 的站点有：Wikipedia, WordPress, Reddit, Tumblr, Pinterest, Dropbox, Slideshare, Stackexchange 等，可以持续罗列好几个小时，他们太多了。

下图是统计数据：



select 模型

下面是 select 函数接口：

```
1. int select (int n, fd_set *readfds, fd_set *writefds,
2.            fd_set *exceptfds, struct timeval *timeout);
```

select 函数监视的文件描述符分 3 类，分别是 writefds、readfds 和 exceptfds。调用后 select 函数会阻塞，直到有描述符就绪（有数据 可读、可写、或者有 except），或者超时（timeout 指定等待时间，如果立即返回设为 null 即可）。当 select 函数返回后，通过遍历 fd_set，来找到就绪的描述符。

select 目前几乎在所有的平台上支持，其良好跨平台支持是它的一大优点。select 的一个缺点在

于单个进程能够监视的文件描述符的数量存在最大限制，在 Linux 上一般为 1024，可以通过修改宏定义甚至重新编译内核的方式提升这一限制，但是这样也会造成效率的降低。

poll 模型

```
1. int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

不同于 select 使用三个位图来表示三个 fdset 的方式，poll 使用一个 pollfd 的指针实现。

```
1. struct pollfd {
2.     int fd; /* file descriptor */
3.     short events; /* requested events to watch */
4.     short revents; /* returned events witnessed */
5. };
```

pollfd 结构包含了要监视的 event 和发生的 event，不再使用 select “参数-值”传递的方式。同时，pollfd 并没有最大数量限制（但是数量过大后性能也是会下降）。和 select 函数一样，poll 返回后，需要轮询 pollfd 来获取就绪的描述符。

从上面看，select 和 poll 都需要在返回后，通过遍历文件描述符来获取已经就绪的 socket。事实上，同时连接的大量客户端在一时刻可能只有很少的处于就绪状态，因此随着监视的描述符数量的增长，其效率也会线性下降。

epoll 模型

epoll 的接口如下：

```
1. int epoll_create(int size);
2. int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
3.     typedef union epoll_data {
4.         void *ptr;
5.         int fd;
6.         __uint32_t u32;
7.         __uint64_t u64;
8.     } epoll_data_t;
9.
10.    struct epoll_event {
11.        __uint32_t events; /* Epoll events */
12.        epoll_data_t data; /* User data variable */
13.    };
14.
15. int epoll_wait(int epfd, struct epoll_event * events,
16.               int maxevents, int timeout);
```

主要是 `epoll_create`, `epoll_ctl` 和 `epoll_wait` 三个函数。`epoll_create` 函数创建 `epoll` 文件描述符, 参数 `size` 并不是限制了 `epoll` 所能监听的描述符最大个数, 只是对内核初始分配内部数据结构的一个建议。`epoll_ctl` 完成对指定描述符 `fd` 执行 `op` 操作控制, `event` 是与 `fd` 关联的监听事件。`op` 操作有三种: 添加 `EPOLL_CTL_ADD`, 删除 `EPOLL_CTL_DEL`, 修改 `EPOLL_CTL_MOD`。分别添加、删除和修改对 `fd` 的监听事件。`epoll_wait` 等待 `epfd` 上的 `IO` 事件, 最多返回 `maxevents` 个事件。

在 `select/poll` 中, 进程只有在调用一定的方法后, 内核才对所有监视的文件描述符进行扫描, 而 `epoll` 事先通过 `epoll_ctl()` 来注册一个文件描述符, 一旦基于某个文件描述符就绪时, 内核会采用类似 `callback` 的回调机制, 迅速激活这个文件描述符, 当进程调用 `epoll_wait` 时便得到通知。

`epoll` 的优点主要是一下几个方面:

1. 监视的描述符数量不受限制, 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。`select` 的最大缺点就是进程打开的 `fd` 是有数量限制的。这对于连接数量比较大的服务器来说根本不能满足。虽然也可以选择多进程的解决方案(`Apache` 就是这样实现的), 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。
2. `IO` 的效率不会随着监视 `fd` 的数量的增长而下降。`epoll` 不同于 `select` 和 `poll` 轮询的方式, 而是通过每个 `fd` 定义的回调函数来实现的。只有就绪的 `fd` 才会执行回调函数。
3. 支持水平触发和边沿触发两种模式:
 - 水平触发模式, 文件描述符状态发生变化后, 如果没有采取行动, 它将后面反复通知, 这种情况下编程相对简单, `libevent` 等开源库很多都是使用的这种模式。
 - 边沿触发模式, 只告诉进程哪些文件描述符刚刚变为就绪状态, 只说一遍, 如果没有采取行动, 那么它将不会再次告知。理论上边缘触发的性能要更高一些, 但是代码实现相当复杂(`Nginx` 使用的边缘触发)。
4. `mmap` 加速内核与用户空间的信息传递。`epoll` 是通过内核与用户空间 `mmap` 同一块内存, 避免了无谓的内存拷贝。

OpenResty 简介

- [简介](#)

简介

OpenResty (也称为 `ngx_openresty`) 是一个全功能的 web 应用服务器。它打包了标准的 Nginx 核心, 很多的常用的第三方模块, 以及它们的大多数依赖项。

通过揉和众多设计良好的 Nginx 模块, OpenResty 有效地把 Nginx 服务器转变为一个强大的 web 应用服务器, 基于它开发人员可以使用 Lua 编程语言对 Nginx 核心以及现有的各种 Nginx C 模块进行脚本编程, 构建出可以处理一万以上并发请求的极端高性能的 web 应用。

OpenResty 致力于将你的服务器端应用完全运行于 Nginx 服务器中, 充分利用 Nginx 的事件模型来进行非阻塞 I/O 通信。不仅仅是和 HTTP 客户端间的网络通信是非阻塞的, 与 MySQL、PostgreSQL、Memcached 以及 Redis 等众多远方后端之间的网络通信也是非阻塞的。

因为 OpenResty 软件包的维护者也是其中打包的许多 Nginx 模块的作者, 所以 OpenResty 可以确保所包含的所有组件可以可靠地协同工作。

OpenResty 最早是雅虎中国的一个公司项目, 起步于 2007 年 10 月。当时兴起了 OpenAPI 的热潮, 用于满足各种 Web Service 的需求, 就诞生了 OpenResty。在公司领导的支持下, 最早的 OpenResty 实现从一开始就开源了。最初的定位是服务于公司外的开发者, 像其他的 OpenAPI 那样, 但后来越来越多地是为雅虎中国的搜索产品提供内部服务。这是第一代的 OpenResty, 当时的想法是, 提供一套抽象的 web service, 能够让用户利用这些 web service 构造出新的符合他们具体业务需求的 web service 出来, 所以有些“meta web service”的意味, 包括数据模型、查询、安全策略都可以通过这种 meta web service 来表达和配置。同时这种 web service 也有意保持 REST 风格。与这种概念相对应的是纯 AJAX 的 web 应用, 即 web 应用几乎都使用客户端 JavaScript 来编写, 然后完全由 web service 让 web 应用“活”起来。用户把 .html, .js, .css, .jpg 等静态文件下载到 web browser 中, 然后 js 开始运行, 跨域请求雅虎提供的经过站长定制过的 web service, 然后应用就可以运行起来。不过随着后来的发展, 公司外的用户毕竟还是少数, 于是应用的重点是为公司内部的其他团队提供 web service, 比如雅虎中国的全能搜索产品, 及其外围的一些产品。从那以后, 开发的重点便放在了性能优化上面。章亦春在加入淘宝数据部门的量子团队之后, 决定对 OpenResty 进行重新设计和彻底重写, 并把应用重点放在支持像量子统计这样的 web 产品上面, 所以量子统计 3.0 开始也几乎完全是 web service 驱动纯 AJAX 应用。

这是第二代的 OpenResty, 一般称之为 `ngx_openresty`, 以便和第一代基于 Perl 和 Haskell 实现的 OpenResty 加以区别。章亦春和他的同事王晓哲一起设计了第二代的 OpenResty。在王晓哲的提议下, 选择基于 nginx 和 lua 进行开发。

为什么要取 OpenResty 这个名字呢? OpenResty 最早是顺应 OpenAPI 的潮流做的, 所以

Open 取自“开放”之意，而Resty便是 REST 风格的意思。虽然后来也可以基于 ngx_openresty 实现任何形式的 web service 或者传统的 web 应用。

也就是说 Nginx 不再是一个简单的静态网页服务器，也不再是一个简单的反向代理了。第二代的 openresty 致力于通过一系列 nginx 模块，把nginx扩展为全功能的 web 应用服务器。

ngx_openresty 是用户驱动的项目，后来也有不少国内用户的参与，从 openresty.org 的点击量分布上看，国内和国外的点击量基本持平。

ngx_openresty 目前有两大应用目标：

1. 通用目的的 web 应用服务器。在这个目标下，现有的 web 应用技术都可以算是和 OpenResty 或多或少有些类似，比如 Nodejs, PHP 等等。ngx_openresty 的性能（包括内存使用和 CPU 效率）算是最大的卖点之一。
2. Nginx 的脚本扩展编程，用于构建灵活的 Web 应用网关和 Web 应用防火墙。有些类似的是 NetScaler。其优势在于 Lua 编程带来的巨大灵活性。

ngx_openresty 从一开始就是公司实际的业务需求的产物。在过去的几年中的大部分开发工作也是由国内外许多公司和个人的实际业务需求驱动的。这种模型在实践中工作得非常好，可以确保我们做的就是大家最迫切需要的。在此过程中，慢慢形成了 ngx_openresty 的两大应用方向，也就是前面提到的那两大方向。是我们的用户帮助我们确认了这两个方向，事实上，这并不等同于第一代 OpenResty 的方向，而是变得更加底层和更加通用了。

开源精神的核心是分享而非追求流行。毕竟开源界不是娱乐圈，也不是时尚圈。如果我们的开源项目有越来越多的人开始使用，只是一个“happy accident”，我们自然会很高兴，但这并不是我们真正追求的。

开放源码只是开源项目生命周期中的“万里长征第一步”，国内的许多开源项目止步于开放源码，而没有后续投入长期的时间和精力去跟进响应用户的各种需求和反馈，但不免夭折。这种现象在国外的不少开源项目中也常见。

国外成功的开源项目比较多，或许跟许多发达国家的程序员们的精神状态有关系。比如我认识的一些国外的黑客都非常心思单纯，热情似火。他们在精神上的束缚非常少，做起事来多是不拘一格。有的人即便长期没有工作单纯靠抵押和捐赠过活，也会不遗余力地投身于开源项目。而我接触到的国内许多程序员的精神负担一般比较重，经济上的压力也比较大，自然难有“玩开源”的心思。

不过，国内也是有一些程序员拥有国外优秀黑客的素质的，而且他们通过网络和全球的黑客紧密联系在一起，所以我们完全可以期待他们未来有振奋人心的产出。在互联网时代的今天，或许按国界的划分来讨论这样的问题会变得越来越不合时宜。

摘自：[OpenResty 作者章亦春访谈实录](#)

Lua 入门

- [Lua 入门](#)

Lua 入门

Lua 是一个小巧的脚本语言。是巴西里约热内卢天主教大学 (Pontifical Catholic University of Rio de Janeiro) 里的一个研究小组, 由 Roberto Ierusalimschy、Waldemar Celes 和 Luiz Henrique de Figueiredo 所组成并于 1993 年开发。其设计目的是为了嵌入应用程序中, 从而为应用程序提供灵活的扩展和定制功能。Lua 由标准 C 编写而成, 几乎在所有操作系统和平台上都可以编译、运行。Lua 并没有提供强大的库, 这是由它的定位决定的。所以 Lua 不适合作为开发独立应用程序的语言。Lua 有一个同时进行的 JIT 项目, 提供在特定平台上的即时编译功能。

Lua 脚本可以很容易的被 C/C++ 代码调用, 也可以反过来调用 C/C++ 的函数, 这使得 Lua 在应用程序中可以被广泛应用。不仅仅作为扩展脚本, 也可以作为普通的配置文件, 代替 XML、ini 等文件格式, 并且更容易理解和维护。标准 Lua 5.1 解释器由标准 C 编写而成, 代码简洁优美, 几乎在所有操作系统和平台上都可以编译和运行; 一个完整的标准 Lua 5.1 解释器不足 200KB。而本书推荐使用的 LuaJIT 2 的代码大小也只有不足 500KB, 同时也支持大部分常见的体系结构。在目前所有脚本语言引擎中, LuaJIT 2 实现的速度应该算是最快的之一。这一切都决定了 Lua 是作为嵌入式脚本的最佳选择。

Lua 语言的各个版本是不相兼容的。因此本书只介绍 Lua 5.1 语言, 这是为标准 Lua 5.1 解释器和 LuaJIT 2 所共同支持的。LuaJIT 支持的对 Lua 5.1 向后兼容的 Lua 5.2 和 Lua 5.3 的特性, 我们也会在方便的时候予以介绍。

Lua 简介

- [Lua 简介](#)
 - [Lua 是什么?](#)
 - [Lua 和 LuaJIT 的区别](#)

Lua 简介

这一章我们简要地介绍 Lua 语言的基础知识，特别地，我们会将讨论放置于 OpenResty 的上下文中。同时，我们并不会回避 LuaJIT 独有的新特性；当然，在遇到这样的独有特性时，我们都会予以说明。我们会关注各个语言结构和标准库函数对性能的潜在影响。在讨论性能相关的问题时，我们只会关心 LuaJIT 实现。

Lua 是什么?

1993 年在巴西里约热内卢天主教大学(Pontifical Catholic University of Rio de Janeiro in Brazil)诞生了一门编程语言，发明者是该校的三位研究人员，他们给这门语言取了个浪漫的名字——`Lua`，在葡萄牙语里代表美丽的月亮。事实证明她没有糟蹋这个优美的单词，Lua 语言正如它名字所预示的那样成长为一门简洁、优雅且富有乐趣的语言。

Lua 从一开始就是作为一门方便嵌入(其它应用程序)并可扩展的轻量级脚本语言来设计的，因此她一直遵从着简单、小巧、可移植、快速的原则，官方实现完全采用 ANSI C 编写，能以 C 程序库的形式嵌入到宿主程序中。LuaJIT 2 和标准 Lua 5.1 解释器采用的是著名的 MIT 许可协议。正由于上述特点，所以 Lua 在游戏开发、机器人控制、分布式应用、图像处理、生物信息学等各种各样的领域中得到了越来越广泛的应用。其中尤以游戏开发为最，许多著名的游戏，比如 Escape from Monkey Island、World of Warcraft、大话西游，都采用了 Lua 来配合引擎完成数据描述、配置管理和逻辑控制等任务。即使像 Redis 这样中性的内存键值数据库也提供了内嵌用户 Lua 脚本的官方支持。

作为一门过程型动态语言，Lua 有着如下的特性：

1. 变量名没有类型，值才有类型，变量名在运行时可与任何类型的值绑定；
2. 语言只提供唯一一种数据结构，称为表(table)，它混合了数组、哈希，可以用任何类型的值作为 key 和 value。提供了一致且富有表达力的表构造语法，使得 Lua 很适合描述复杂的数据；
3. 函数是一等类型，支持匿名函数和正则尾递归(proper tail recursion)；
4. 支持词法定界(lexical scoping)和闭包(closure)；
5. 提供 thread 类型和结构化的协程(coroutine)机制，在此基础上可方便实现协作式多任务；
6. 运行期能编译字符串形式的程序文本并载入虚拟机执行；
7. 通过元表(metatable)和元方法(metamethod)提供动态元机制(dynamic meta-mechanism)，从而允许程序运行时根据需要改变或扩充语法设施的内在语义；

8. 能方便地利用表和动态元机制实现基于原型(prototype-based)的面向对象模型;
9. 从 5.1 版开始提供了完善的模块机制,从而更好地支持开发大型的应用程序;

Lua 的语法类似 PASCAL 和 Modula 但更加简洁,所有的语法产生式规则(EBNF)不过才 60 几个。熟悉 C 和 PASCAL 的程序员一般只需半个小时便可将其完全掌握。而在语义上 Lua 则与 Scheme 极为相似,她们完全共享上述的 1、3、4、6 点特性, Scheme 的 continuation 与协程也基本相同只是自由度更高。最引人注目的是,两种语言都只提供唯一一种数据结构: Lua 的表和 Scheme 的列表(list)。正因为如此,有人甚至称 Lua 为“只用表的 Scheme”。

Lua 和 LuaJIT 的区别

Lua 非常高效,它运行得比许多其它脚本(如 Perl、Python、Ruby)都快,这点在第三方的独立测评中得到了证实。尽管如此,仍然会有人不满足,他们总觉得“嗯,还不够快!”。LuaJIT 就是一个为了再榨出一些速度的尝试,它利用即时编译(Just-in Time)技术把 Lua 代码编译成本地机器码后交由 CPU 直接执行。LuaJIT 2 的测评报告表明,在数值运算、循环与函数调用、协程切换、字符串操作等许多方面它的加速效果都很显著。凭借着 FFI 特性, LuaJIT 2 在那些需要频繁地调用外部 C/C++ 代码的场景,也要比标准 Lua 解释器快很多。目前 LuaJIT 2 已经支持包括 i386、x86_64、ARM、PowerPC 以及 MIPS 等多种不同的体系结构。

LuaJIT 是采用 C 和汇编语言编写的 Lua 解释器与即时编译器。LuaJIT 被设计成全兼容标准的 Lua 5.1 语言,同时可选地支持 Lua 5.2 和 Lua 5.3 中的一些不破坏向后兼容性的有用特性。因此,标准 Lua 语言的代码可以不加修改地运行在 LuaJIT 之上。LuaJIT 和标准 Lua 解释器的一大区别是, LuaJIT 的执行速度,即使是其汇编编写的 Lua 解释器,也要比标准 Lua 5.1 解释器快很多,可以说是一个高效的 Lua 实现。另一个区别是, LuaJIT 支持比标准 Lua 5.1 语言更多的基本原语和特性,因此功能上也要更加强大。

若无特殊说明,我们接下来的章节都是基于 LuaJIT 进行介绍的。

- Lua 官网链接: <http://www.lua.org>
- LuaJIT 官网链接: <http://luajit.org>

Lua 环境搭建

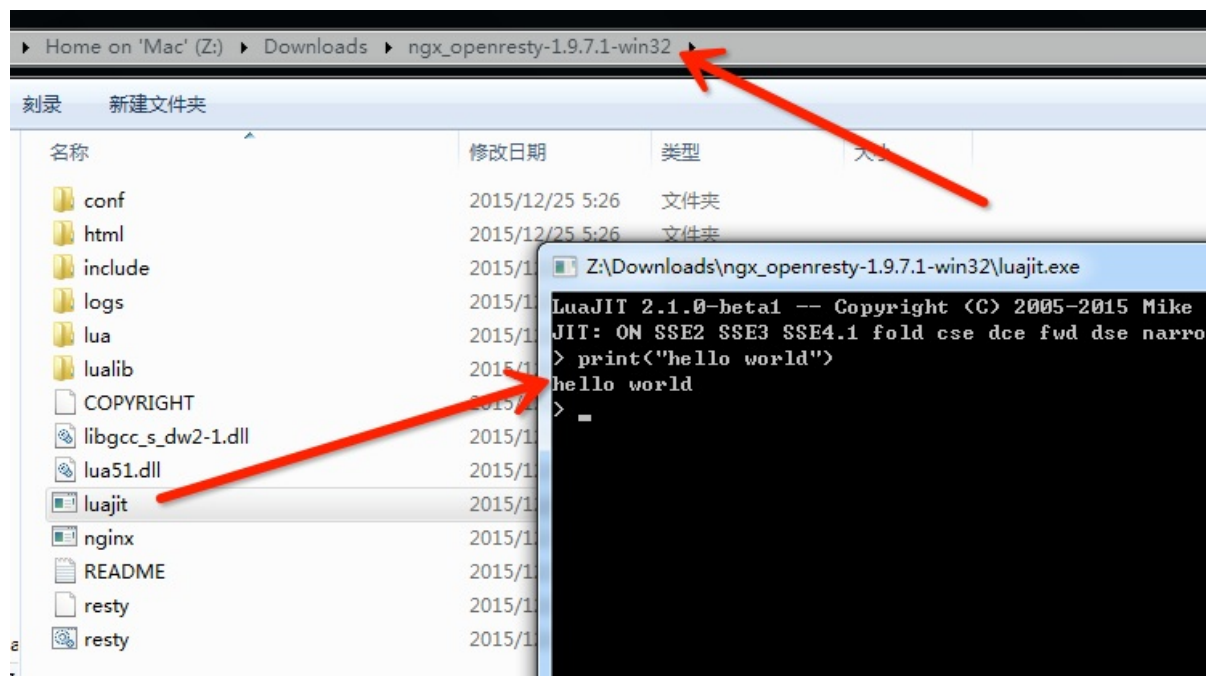
- Lua 环境搭建
 - 在 Windows 上搭建环境
 - 在 Linux、Mac OS X 上搭建环境
 - 验证 LuaJIT 是否安装成功
- 第一个“Hello World”

Lua 环境搭建

在 Windows 上搭建环境

从 1.9.3.2 版本开始，OpenResty 正式对外同时公布维护了 Windows 版本，其中直接包含了编译好的最新版本 LuaJIT。由于 Windows 操作系统自身相对良好的二进制兼容性，使用者只需要下载、解压两个步骤即可。

打开 <http://openresty.org>，选择左侧的 [Download](#) 连接，这时候我们就可以下载最新版本的 OpenResty 版本（例如笔者写书时的最新版本：[ngx_openresty-1.9.7.1-win32.zip](#)）。下载本地成功后，执行解压缩，就能看到下图所示目录结构：



双击图中的 LuaJIT.exe，即可进入命令行模式，在这里我们就可以直接完成简单的 Lua 语法交互了。

在 Linux、Mac OS X 上搭建环境

到 LuaJIT 官网 <http://luajit.org/download.html>，查看当前最新开发版本，例如笔者写书时的最新版本：<http://luajit.org/download/LuaJIT-2.1.0-beta1.tar.gz>。

```
1. # wget http://luajit.org/download/LuaJIT-2.1.0-beta1.tar.gz
2. # tar -xvf LuaJIT-2.1.0-beta1.tar.gz
3. # cd LuaJIT-2.1.0-beta1
4. # make
5. # sudo make install
```

大家都知道，在不同平台，可能都有不同的安装工具来简化我们的安装。为什么我们这给大家推荐的是源码这么原始的方式？笔者为了偷懒么？答案：是的。当然还有另外一个原因，就是我们安装的是 LuaJIT 2.1 版本。

从实际应用性能表现来看，LuaJIT 2.1 虽然目前还是 beta 版本，但是生产运行稳定性已经很不错，并且在运行效率上要比 LuaJIT 2.0 好很多（大家可自行爬文了解一下），所以作为 OpenResty 的默认搭档，已经是 LuaJIT 2.1 很久了。但是针对不同系统的工具包安装工具，他们当前默认绑定推送的都还是 LuaJIT 2.0，所以这里就直接给出最符合我们最终方向的安装方法了。

验证 LuaJIT 是否安装成功

```
1. # luajit -v
2. LuaJIT 2.1.0-beta1 -- Copyright (C) 2005-2015 Mike Pall.
3. http://luajit.org/
```

如果了解其他系统安装 LuaJIT 的步骤，或者安装过程中遇到问题，可以到 LuaJIT 官网查看：<http://luajit.org/install.html>

第一个“Hello World”

安装好 LuaJIT 后，开始我们的第一个 hello world 小程序。首先编写一个 hello.lua 文件，写入内容后，使用 LuaJIT 运行即可。

```
1. # cat hello.lua
2. print("hello world")
3. # luajit hello.lua
4. hello world
```

基础数据类型

- [Lua 基础数据类型](#)
 - [nil \(空\)](#)
 - [boolean \(布尔\)](#)
 - [number \(数字\)](#)
 - [string \(字符串\)](#)
 - [table \(表\)](#)
 - [function \(函数\)](#)

Lua 基础数据类型

函数 `type` 能够返回一个值或一个变量所属的类型。

```
1. print(type("hello world")) -->output:string
2. print(type(print))         -->output:function
3. print(type(true))          -->output:boolean
4. print(type(360.0))         -->output:number
5. print(type(nil))           -->output:nil
```

nil (空)

`nil` 是一种类型，Lua 将 `nil` 用于表示“无效值”。一个变量在第一次赋值前的默认值是 `nil`，将 `nil` 赋予给一个全局变量就等同于删除它。

```
1. local num
2. print(num)         -->output:nil
3.
4. num = 100
5. print(num)         -->output:100
```

值得一提的是，OpenResty 的 Lua 接口还提供了一种特殊的空值，即 `ngx.null`，用来表示不同于 `nil` 的“空值”。后面在讨论 OpenResty 的 Redis 库的时候，我们还会遇到它。

boolean (布尔)

布尔类型，可选值 `true/false`；Lua 中 `nil` 和 `false` 为“假”，其它所有值均为“真”。比如 `0` 和空字符串就是“真”；C 或者 Perl 程序员或许会对此感到惊讶。

```
1. local a = true
2. local b = 0
```

```

3. local c = nil
4. if a then
5.     print("a")      -->output:a
6. else
7.     print("not a")  --这个没有执行
8. end
9.
10. if b then
11.     print("b")      -->output:b
12. else
13.     print("not b")  --这个没有执行
14. end
15.
16. if c then
17.     print("c")      --这个没有执行
18. else
19.     print("not c")  -->output:not c
20. end

```

number (数字)

Number 类型用于表示实数，和 C/C++ 里面的 double 类型很类似。可以使用数学函数 `math.floor` (向下取整) 和 `math.ceil` (向上取整) 进行取整操作。

```

1. local order = 3.99
2. local score = 98.01
3. print(math.floor(order))  -->output:3
4. print(math.ceil(score))  -->output:99

```

一般地，Lua 的 number 类型就是用双精度浮点数来实现的。值得一提的是，LuaJIT 支持所谓的“dual-number” (双数) 模式，即 LuaJIT 会根据上下文用整型来存储整数，而用双精度浮点数来存放浮点数。

另外，LuaJIT 还支持“长长整型”的大整数 (在 x86_64 体系结构上则是 64 位整数)。例如

```

1. print(9223372036854775807LL - 1)  -->output:9223372036854775806LL

```

string (字符串)

Lua 中有三种方式表示字符串：

- 1、使用一对匹配的单引号。例：'hello'。
- 2、使用一对匹配的双引号。例："abclua"。

3、字符串还可以用一种长括号（即[[]]）括起来的方式定义。我们把两个正的方括号（即[]）间插入 n 个等号定义为第 n 级正长括号。就是说，0 级正的长括号写作 []，一级正的长括号写作 [=[]，如此等等。反的长括号也作类似定义；举个例子，4 级反的长括号写作]====]。一个长字符串可以由任何一级的正的长括号开始，而由第一个碰到的同级反的长括号结束。整个词法分析过程将不受分行限制，不处理任何转义符，并且忽略掉任何不同级别的长括号。这种方式描述的字符串可以包含任何东西，当然本级别的反长括号除外。例：[[abc\nbc]]，里面的 “\n” 不会被转义。

另外，Lua 的字符串是不可改变的值，不能像在 c 语言中那样直接修改字符串的某个字符，而是根据修改要求来创建一个新的字符串。Lua 也不能通过下标来访问字符串的某个字符。想了解更多关于字符串的操作，请查看[String 库](#)章节。

```
1. local str1 = 'hello world'
2. local str2 = "hello lua"
3. local str3 = [[ "add\nname", 'hello' ]]
4. local str4 = [= [string have a [[]].]=]
5.
6. print(str1)      -->output:hello world
7. print(str2)      -->output:hello lua
8. print(str3)      -->output:"add\nname", 'hello'
9. print(str4)      -->output:string have a [[]].
```

在 Lua 实现中，Lua 字符串一般都会经历一个“内化”（intern）的过程，即两个完全一样的 Lua 字符串在

Lua 虚拟机中只会存储一份。每一个 Lua 字符串在创建时都会插入到 Lua 虚拟机内部的一个全局的哈希表中。

这意味着

1. 创建相同的 Lua 字符串并不会引入新的动态内存分配操作，所以相对便宜（但仍有全局哈希表查询的开销），
2. 内容相同的 Lua 字符串不会占用多份存储空间，
3. 已经创建好的 Lua 字符串之间进行相等性比较时是 $O(1)$ 时间度的开销，而不是通常见到的 $O(n)$ 。

table（表）

Table 类型实现了一种抽象的“关联数组”。“关联数组”是一种具有特殊索引方式的数组，索引通常是字符串（string）或者 number 类型，但也可以是除 `nil` 以外的任意类型的值。

```
1. local corp = {
2.     web = "www.google.com",    --索引为字符串, key = "web",
3.                                     -- value = "www.google.com"
4.     telephone = "12345678",    --索引为字符串
5.     staff = {"Jack", "Scott", "Gary"}, --索引为字符串, 值也是一个表
6.     100876,                    --相当于 [1] = 100876, 此时索引为数字
```

```

7.          --      key = 1, value = 100876
8.      100191,      --相当于 [2] = 100191, 此时索引为数字
9.      [10] = 360,   --直接把数字索引给出
10.     ["city"] = "Beijing" --索引为字符串
11. }
12.
13. print(corp.web)      -->output:www.google.com
14. print(corp["telephone"]) -->output:12345678
15. print(corp[2])      -->output:100191
16. print(corp["city"]) -->output:"Beijing"
17. print(corp.staff[1]) -->output:Jack
18. print(corp[10])     -->output:360

```

在内部实现上，table 通常实现为一个哈希表、一个数组、或者两者的混合。具体的实现为何种形式，动态依赖于具体的 table 的键分布特点。

想了解更多关于 table 的操作，请查看 [Table 库](#) 章节。

function (函数)

在 Lua 中，函数 也是一种数据类型，函数可以存储在变量中，可以通过参数传递给其他函数，还可以作为其他函数的返回值。

示例

```

1. local function foo()
2.     print("in the function")
3.     --dosomething()
4.     local x = 10
5.     local y = 20
6.     return x + y
7. end
8.
9. local a = foo    --把函数赋给变量
10.
11. print(a())
12.
13. --output:
14. in the function
15. 30

```

有名函数的定义本质上是匿名函数对变量的赋值。为说明这一点，考虑

```

1. function foo()
2. end

```

等价于

```
1. foo = function ()  
2. end
```

类似地，

```
1. local function foo()  
2. end
```

等价于

```
1. local foo = function ()  
2. end
```

表达式

- 表达式
 - 算术运算符
 - 关系运算符
 - 逻辑运算符
 - 字符串连接
 - 优先级

表达式

算术运算符

Lua 的算术运算符如下表所示：

算术运算符	说明
+	加法
-	减法
*	乘法
/	除法
^	指数
%	取模

示例代码：test1.lua

```
1. print(1 + 2)      -->打印 3
2. print(5 / 10)     -->打印 0.5。 这是Lua不同于c语言的
3. print(5.0 / 10)   -->打印 0.5。 浮点数相除的结果是浮点数
4. -- print(10 / 0)  -->注意除数不能为0，计算的结果会出错
5. print(2 ^ 10)     -->打印 1024。 求2的10次方
6.
7. local num = 1357
8. print(num % 2)     -->打印 1
9. print((num % 2) == 1) -->打印 true。 判断num是否为奇数
10. print((num % 5) == 0) -->打印 false。判断num是否能被5整数
```

关系运算符

关系运算符	说明
<	小于

>	大于
<=	小于等于
>=	大于等于
==	等于
~=	不等于

示例代码: `test2.lua`

```
1. print(1 < 2)    -->打印 true
2. print(1 == 2)   -->打印 false
3. print(1 ~= 2)   -->打印 true
4. local a, b = true, false
5. print(a == b)   -->打印 false
```

注意: Lua 语言中“不等于”运算符的写法为: `~=`

在使用“==”做等于判断时,要注意对于 `table`, `userdata` 和函数, `Lua` 是作引用比较的。也就是说,只有当两个变量引用同一个对象时,才认为它们相等。可以看下面的例子:

```
1. local a = { x = 1, y = 0}
2. local b = { x = 1, y = 0}
3. if a == b then
4.     print("a==b")
5. else
6.     print("a~=b")
7. end
8.
9. ---output:
10. a~=b
```

由于 `Lua` 字符串总是会被“内化”,即相同内容的字符串只会被保存一份,因此 `Lua` 字符串之间的相等性比较可以简化为其内部存储地址的比较。这意味着 `Lua` 字符串的相等性比较总是为 $O(1)$ 。而在其他编程语言中,字符串的相等性比较则通常为 $O(n)$,即需要逐个字节(或按若干个连续字节)进行比较。

逻辑运算符

逻辑运算符	说明
and	逻辑与
or	逻辑或
not	逻辑非

Lua 中的 `and` 和 `or` 是不同于 C 语言的。在 C 语言中，`and` 和 `or` 只得到两个值 1 和 0，其中 1 表示真，0 表示假。而 Lua 中 `and` 的执行过程是这样的：

- `a and b` 如果 `a` 为 `nil`，则返回 `a`，否则返回 `b`；
- `a or b` 如果 `a` 为 `nil`，则返回 `b`，否则返回 `a`。

示例代码：test3.lua

```
1. local c = nil
2. local d = 0
3. local e = 100
4. print(c and d) -->打印 nil
5. print(c and e) -->打印 nil
6. print(d and e) -->打印 100
7. print(c or d)  -->打印 0
8. print(c or e)  -->打印 100
9. print(not c)   -->打印 true
10. print(not d)  -->打印 false
```

注意：所有逻辑操作符将 `false` 和 `nil` 视作假，其他任何值视作真，对于 `and` 和 `or`，“短路求值”，对于 `not`，永远只返回 `true` 或者 `false`。

字符串连接

在 Lua 中连接两个字符串，可以使用操作符“`..`”（两个点）。如果其任意一个操作数是数字的话，Lua 会将这个数字转换成字符串。注意，连接操作符只会创建一个新字符串，而不会改变原操作数。也可以使用 `string` 库函数 `string.format` 连接字符串。

```
1. print("Hello " .. "World") -->打印 Hello World
2. print(0 .. 1)             -->打印 01
3.
4. str1 = string.format("%s-%s", "hello", "world")
5. print(str1)                -->打印 hello-world
6.
7. str2 = string.format("%d-%s-%.2f", 123, "world", 1.21)
8. print(str2)                -->打印 123-world-1.21
```

由于 Lua 字符串本质上是只读的，因此字符串连接运算符几乎总会创建一个新的（更大的）字符串。这意味着如果有很多这样的连接操作（比如在循环中使用 `..` 来拼接最终结果），则性能损耗会非常大。在这种情况下，推荐使用 `table` 和 `table.concat()` 来进行很多字符串的拼接，例如：

```
1. local pieces = {}
2. for i, elem in ipairs(my_list) do
3.     pieces[i] = my_process(elem)
4. end
```

```
5. local res = table.concat(pieces)
```

当然，上面的例子还可以使用 LuaJIT 独有的 `table.new` 来恰当地初始化 `pieces` 表的空
间，以避免该表的动态生长。这个特性我们在后面还会详细讨论。

优先级

Lua 操作符的优先级如下表所示(从高到低)：

优先级					
^					
not # -					
* / %					
+ -					
..					
< > <= >= == ~=					
and					
or					

示例：

```
1. local a, b = 1, 2
2. local x, y = 3, 4
3. local i = 10
4. local res = 0
5. res = a + i < b/2 + 1 -->等价于res = (a + i) < ((b/2) + 1)
6. res = 5 + x^2*8       -->等价于res = 5 + ((x^2) * 8)
7. res = a < y and y <=x -->等价于res = (a < y) and (y <= x)
```

若不确定某些操作符的优先级，就应显示地用括号来指定运算顺序。这样做还可以提高代码的可读性。

控制结构

- [控制结构](#)

控制结构

流程控制语句对于程序设计来说特别重要，它可以用于设定程序的逻辑结构。一般需要与条件判断语句结合使用。Lua 语言提供的控制结构有 `if`，`while`，`repeat`，`for`，并提供 `break` 关键字来满足更丰富的需求。本章主要介绍 Lua 语言的控制结构的使用。

if/else

- 控制结构 if-else
 - 单个 if 分支 型
 - 两个分支 if-else 型
 - 多个分支 if-elseif-else 型

控制结构 if-else

if-else 是我们熟知的一种控制结构。Lua 跟其他语言一样，提供了 if-else 的控制结构。因为是大家熟悉的语法，本节只简单介绍一下它的使用方法。

单个 if 分支 型

```
1. x = 10
2. if x > 0 then
3.     print("x is a positive number")
4. end
```

运行输出: *x is a positive number*

两个分支 if-else 型

```
1. x = 10
2. if x > 0 then
3.     print("x is a positive number")
4. else
5.     print("x is a non-positive number")
6. end
```

运行输出: *x is a positive number*

多个分支 if-elseif-else 型

```
1. score = 90
2. if score == 100 then
3.     print("Very good!Your score is 100")
4. elseif score >= 60 then
5.     print("Congratulations, you have passed it,your score greater or equal to 60")
6. --此处可以添加多个elseif
7. else
8.     print("Sorry, you do not pass the exam! ")
```

```
9. end
```

运行输出: *Congratulations, you have passed it,your score greater or equal to 60*

与 C 语言的不同之处是 else 与 if 是连在一起的, 若将 else 与 if 写成 “else if” 则相当于在 else 里嵌套另一个 if 语句, 如下代码:

```
1. score = 0
2. if score == 100 then
3.     print("Very good!Your score is 100")
4. elseif score >= 60 then
5.     print("Congratulations, you have passed it,your score greater or equal to 60")
6. else
7.     if score > 0 then
8.         print("Your score is better than 0")
9.     else
10.        print("My God, your score turned out to be 0")
11.    end --与上一示例代码不同的是, 此处要添加一个end
12. end
```

运行输出: *My God, your score turned out to be 0*

while

- `while` 型控制结构

while 型控制结构

Lua 跟其他常见语言一样，提供了 `while` 控制结构，语法上也没有什么特别的。但是没有提供 `do-while` 型的控制结构，但是提供了功能相当的 `repeat`。

`while` 型控制结构语法如下，当表达式值为假（即 `false` 或 `nil`）时结束循环。也可以使用 `break` 语言提前跳出循环。

```
1. while 表达式 do
2.   --body
3. end
```

示例代码，求 $1 + 2 + 3 + 4 + 5$ 的结果

```
1. x = 1
2. sum = 0
3.
4. while x <= 5 do
5.     sum = sum + x
6.     x = x + 1
7. end
8. print(sum) -->output 15
```

值得一提的是，Lua 并没有像许多其他语言那样提供类似 `continue` 这样的控制语句用来立即进入下一个循环迭代（如果有的话）。因此，我们需要仔细地安排循环体里的分支，以避免这样的需求。

没有提供 `continue`，却也提供了另外一个标准控制语句 `break`，可以跳出当前循环。例如我们遍历 `table`，查找值为 11 的数组下标索引：

```
1. local t = {1, 3, 5, 8, 11, 18, 21}
2.
3. local i
4. for i, v in ipairs(t) do
5.     if 11 == v then
6.         print("index[" .. i .. "] have right value[11]")
7.         break
8.     end
9. end
```

while



repeat

- [repeat 控制结构](#)

repeat 控制结构

Lua 中的 `repeat` 控制结构类似于其他语言（如：C++ 语言）中的 `do-while`，但是控制方式是刚好相反的。简单点说，执行 `repeat` 循环体后，直到 `until` 的条件为真时才结束，而其他语言（如：C++ 语言）的 `do-while` 则是当条件为假时就结束循环。

以下代码将会形成死循环：

```
1. x = 10
2. repeat
3.     print(x)
4. until false
```

该代码将导致死循环，因为`until`的条件一直为假，循环不会结束

除此之外，`repeat` 与其他语言的 `do-while` 基本是一样的。同样，Lua 中的 `repeat` 也可以使用 `break` 退出。

for

- [for 控制结构](#)
 - [for 数字型](#)
 - [for 泛型](#)

for 控制结构

Lua 提供了一组传统的、小巧的控制结构，包括用于条件判断的 `if` 用于迭代的 `while`、`repeat` 和 `for`，本章节主要介绍 `for` 的使用。

for 数字型

`for` 语句有两种形式：数字 `for` (numeric for) 和范型 `for` (generic for)。

数字型 `for` 的语法如下：

```
1. for var = begin, finish, step do
2.     --body
3. end
```

关于数字 `for` 需要关注以下几点：

1. `var` 从 `begin` 变化到 `finish`，每次变化都以 `step` 作为步长递增 `var`
2. `begin`、`finish`、`step` 三个表达式只会在循环开始时执行一次
3. 第三个表达式 `step` 是可选的，默认为 1
4. 控制变量 `var` 的作用域仅在 `for` 循环内，需要在外边控制，则需将值赋给一个新的变量
5. 循环过程中不要改变控制变量的值，那样会带来不可预知的影响

示例

```
1. for i = 1, 5 do
2.     print(i)
3. end
4.
5. -- output:
6. 1
7. 2
8. 3
9. 4
10. 5
```

...

```

1. for i = 1, 10, 2 do
2.   print(i)
3. end
4.
5. -- output:
6. 1
7. 3
8. 5
9. 7
10. 9

```

以下是这种循环的一个典型示例：

```

1. for i = 10, 1, -1 do
2.   print(i)
3. end
4.
5. -- output:
6. ...

```

如果不想给循环设置上限的话，可以使用常量 `math.huge`：

```

1. for i = 1, math.huge do
2.   if (0.3*i^3 - 20*i^2 - 500 >= 0) then
3.     print(i)
4.     break
5.   end
6. end

```

for 泛型

泛型 `for` 循环通过一个迭代器（`iterator`）函数来遍历所有值：

```

1. -- 打印数组a的所有值
2. local a = {"a", "b", "c", "d"}
3. for i, v in ipairs(a) do
4.   print("index:", i, " value:", v)
5. end
6.
7. -- output:
8. index: 1 value: a
9. index: 2 value: b
10. index: 3 value: c
11. index: 4 value: d

```

Lua 的基础库提供了 `ipairs`，这是一个用于遍历数组的迭代器函数。在每次循环中，`i` 会被赋予一个索引值，同时 `v` 被赋予一个对应于该索引的数组元素值。

下面是另一个类似的示例，演示了如何遍历一个 `table` 中所有的 `key`

```
1. -- 打印table t中所有的key
2. for k in pairs(t) do
3.     print(k)
4. end
```

从外观上看泛型 `for` 比较简单，但其实它是非常强大的。通过不同的迭代器，几乎可以遍历所有的东西，

而且写出的代码极具可读性。标准库提供了几种迭代器，包括用于迭代文件中每行的

(`io.lines`)、

迭代 `table` 元素的 (`pairs`)、迭代数组元素的 (`ipairs`)、迭代字符串中单词的

(`string.gmatch`) 等。

泛型 `for` 循环与数字型 `for` 循环有两个相同点：

- (1) 循环变量是循环体的局部变量；
- (2) 决不应该对循环变量作任何赋值。

对于泛型 `for` 的使用，再来看一个更具体的示例。假设有这样一个 `table`，它的内容是一周中每天的名称：

```
1. local days = {
2.     "Sunday", "Monday", "Tuesday", "Wednesday",
3.     "Thursday", "Friday", "Saturday"
4. }
```

现在要将一个名称转换成它在一周中的位置。为此，需要根据给定的名称来搜索这个 `table`。然而在 Lua 中，通常更有效的方法是创建一个“逆向 `table`”。例如这个逆向 `table` 叫 `revDays`，它以

一周中每天的名称作为索引，位置数字作为值：

```
1. local revDays = {
2.     ["Sunday"] = 1,
3.     ["Monday"] = 2,
4.     ["Tuesday"] = 3,
5.     ["Wednesday"] = 4,
6.     ["Thursday"] = 5,
7.     ["Friday"] = 6,
8.     ["Saturday"] = 7
9. }
```

for

接下来，要找出一个名称所对应的需要，只需用名字来索引这个 reverse table 即可：

```
1. local x = "Tuesday"
2. print(revDays[x]) -->3
```

当然，不必手动声明这个逆向 table，而是通过原来的 table 自动地构造出这个逆向 table：

```
1. local days = {
2.     "Monday", "Tuesday", "Wednesday", "Thursday",
3.     "Friday", "Saturday", "Sunday"
4. }
5.
6. local revDays = {}
7. for k, v in pairs(days) do
8.     revDays[v] = k
9. end
10.
11. -- print value
12. for k, v in pairs(revDays) do
13.     print("k:", k, " v:", v)
14. end
15.
16. -- output:
17. k: Tuesday v: 2
18. k: Monday v: 1
19. k: Sunday v: 7
20. k: Thursday v: 4
21. k: Friday v: 5
22. k: Wednesday v: 3
23. k: Saturday v: 6
```

这个循环会为每个元素进行赋值，其中变量 `k` 为 key(1、2、...)，变量 `v` 为 value("Sunday"、"Monday"、...)。

值得一提的是，在 LuaJIT 2.1 中，`ipairs()` 内建函数是可以被 JIT 编译的，而 `pairs()` 则只能被解释执行。因此在性能敏感的场景，应当合理安排数据结构，避免对哈希表进行遍历。事实上，即使未来 `pairs` 可以被 JIT 编译，哈希表的遍历本身也不会有数组遍历那么高效，毕竟哈希表就不是为遍历而设计的数据结构。

break, return 和 goto

- break, return 和 goto
 - break
 - return
 - goto

break, return 和 goto

break

语句 `break` 用来终止 `while`、`repeat` 和 `for` 三种循环的执行，并跳出当前循环体，继续执行当前循环之后的语句。下面举一个 `while` 循环中的 `break` 的例子来说明：

```
1. -- 计算最小的x,使从1到x的所有数相加和大于100
2. sum = 0
3. i = 1
4. while true do
5.     sum = sum + i
6.     if sum > 100 then
7.         break
8.     end
9.     i = i + 1
10. end
11. print("The result is " .. i) -->output:The result is 14
```

在实际应用中，`break` 经常用于嵌套循环中。

return

`return` 主要用于从函数中返回结果，或者用于简单的结束一个函数的执行。

关于函数返回值的细节可以参考 [函数的返回值](#) 章节。`return` 只能写在语句块的最后，一旦执行了 `return` 语句，该语句之后的所有语句都不会再执行。若要写在函数中间，则只能写在一个显式的语句块内，参见示例代码：

```
1. local function add(x, y)
2.     return x + y
3.     --print("add: I will return the result " .. (x + y))
4.     --因为前面有个return, 若不注释该语句, 则会报错
5. end
6.
7. local function is_positive(x)
```

```

8.   if x > 0 then
9.       return x .. " is positive"
10.  else
11.       return x .. " is non-positive"
12.  end
13.
14.  --由于return只出现在前面显式的语句块，所以此语句不注释也不会报错
15.  --，但是不会被执行，此处不会产生输出
16.  print("function end!")
17. end
18.
19. local sum = add(10, 20)
20. print("The sum is " .. sum) -->output:The sum is 30
21. local answer = is_positive(-10)
22. print(answer)             -->output:-10 is non-positive

```

有时候，为了调试方便，我们可以想在某个函数的中间提前 `return`，以进行控制流的短路。此时我们可以将 `return` 放在一个 `do ... end` 代码块中，例如：

```

1. local function foo()
2.     print("before")
3.     do return end
4.     print("after") -- 这一行语句永远不会执行到
5. end

```

goto

LuaJIT 一开始对标的是 Lua 5.1，但渐渐地也开始加入部分 Lua 5.2 甚至 Lua 5.3 的有用特性。

`goto` 就是其中一个不得不提的例子。

有了 `goto`，我们可以实现 `continue` 的功能：

```

1. for i=1, 3 do
2.     if i <= 2 then
3.         print(i, "yes continue")
4.         goto continue
5.     end
6.
7.     print(i, " no continue")
8.
9.     ::continue::
10.    print([[i'm end]])
11. end

```

输出结果：

```

1. $ luajit test.lua
2. 1   yes continue
3. i'm end
4. 2   yes continue
5. i'm end
6. 3   no continue
7. i'm end

```

在 [GotoStatement](#) 这个页面上，你能看到更多用 `goto` 玩转控制流的脑洞。

`goto` 的另外一项用途，就是简化错误处理的流程。有些时候你会发现，直接 `goto` 到函数末尾统一的错误处理过程，是更为清晰的写法。

```

1. local function process(input)
2.     print("the input is", input)
3.     if input < 2 then
4.         goto failed
5.     end
6.     -- 更多处理流程和 goto err
7.
8.     print("processing...")
9.     do return end
10.    ::failed::
11.    print("handle error with input", input)
12. end
13.
14. process(1)
15. process(3)

```

PS：推荐大家多关注：<http://luajit.org/extensions.html#lua52>，说不定你就发现了新大陆。

Lua函数

- [Lua 函数](#)

Lua 函数

在 Lua 中，函数是一种对语句和表达式进行抽象的主要机制。函数既可以完成某项特定的任务，也可以只做一些计算并返回结果。在第一种情况中，一句函数调用被视为一条语句；而在第二种情况中，则将其视为一句表达式。

示例代码：

```
1. print("hello world!")      -- 用 print() 函数输出 hello world !
2. local m = math.max(1, 5)   -- 调用数学库函数 max,
3.                             -- 用来求 1,5 中的最大值，并返回赋给变量 m
```

使用函数的好处：

1. 降低程序的复杂性：把函数作为一个独立的模块，写完函数后，只关心它的功能，而不再考虑函数里面的细节。
2. 增加程序的可读性：当我们调用 `math.max()` 函数时，很明显函数是用于求最大值的，实现细节就不关心了。
3. 避免重复代码：当程序中有相同的代码部分时，可以把这部分写成一个函数，通过调用函数来实现这部分代码的功能，节约空间，减少代码长度。
4. 隐含局部变量：在函数中使用局部变量，变量的作用范围不会超出函数，这样它就不会给外界带来干扰。

函数的定义

- [函数定义](#)

函数定义

Lua 使用关键字 `function` 定义函数，语法如下：

```
1. function function_name (arc)  -- arc 表示参数列表，函数的参数列表可以为空
2.     -- body
3. end
```

上面的语法定义了一个全局函数，名为 `function_name`。全局函数本质上就是函数类型的值赋给了一个全局变量，即上面的语法等价于

```
1. function_name = function (arc)
2.     -- body
3. end
```

由于全局变量一般会污染全局名字空间，同时也有性能损耗（即查询全局环境表的开销），因此我们应当尽量使用“局部函数”，其记法是类似的，只是开头加上 `local` 修饰符：

```
1. local function function_name (arc)
2.     -- body
3. end
```

由于函数定义本质上就是变量赋值，而变量的定义总是应放置在变量使用之前，所以函数的定义也需要放置在函数调用之前。

示例代码：

```
1. local function max(a, b)  --定义函数 max，用来求两个数的最大值，并返回
2.     local temp = nil      --使用局部变量 temp，保存最大值
3.     if(a > b) then
4.         temp = a
5.     else
6.         temp = b
7.     end
8.     return temp           --返回最大值
9. end
10.
11. local m = max(-12, 20)    --调用函数 max，找出 -12 和 20 中的最大值
```

```
12. print(m)           --> output 20
```

如果参数列表为空，必须使用 `()` 表明是函数调用。

示例代码：

```
1. local function func()  --形参为空
2.     print("no parameter")
3. end
4.
5. func()                 --函数调用，圆扩号不能省
6.
7. --> output :
8. no parameter
```

在定义函数要注意几点：

1. 利用名字来解释函数、变量的目的，使人通过名字就能看出来函数、变量的作用。
2. 每个函数的长度要尽量控制在一个屏幕内，一眼可以看明白。
3. 让代码自己说话，不需要注释最好。

由于函数定义等价于变量赋值，我们也可以把函数名替换为某个 Lua 表的某个字段，例如

```
1. function foo.bar(a, b, c)
2.     -- body ...
3. end
```

此时我们是把一个函数类型的值赋给了 `foo` 表的 `bar` 字段。换言之，上面的定义等价于

```
1. foo.bar = function (a, b, c)
2.     print(a, b, c)
3. end
```

对于此种形式的函数定义，不能再使用 `local` 修饰符了，因为不存在定义新的局部变量了。

函数的参数

- 函数的参数
 - 按值传递
 - 变长参数
 - 具名参数
 - 按引用传递

函数的参数

按值传递

Lua 函数的参数大部分是按值传递的。值传递就是调用函数时，实参把它的值通过赋值运算传递给形参，然后形参的改变和实参就没有关系了。在这个过程中，实参是通过它在参数表中的位置与形参匹配起来的。

示例代码：

```
1. local function swap(a, b) --定义函数swap, 函数内部进行交换两个变量的值
2.     local temp = a
3.     a = b
4.     b = temp
5.     print(a, b)
6. end
7.
8. local x = "hello"
9. local y = 20
10. print(x, y)
11. swap(x, y)      --调用swap函数
12. print(x, y)     --调用swap函数后, x和y的值并没有交换
13.
14. -->output
15. hello 20
16. 20  hello
17. hello 20
```

在调用函数的时候，若形参个数和实参个数不同时，Lua 会自动调整实参个数。调整规则：若实参个数大于形参个数，从左向右，多余的实参被忽略；若实参个数小于形参个数，从左向右，没有被实参初始化的形参会被初始化为 `nil`。

示例代码：

```
1. local function fun1(a, b)      --两个形参, 多余的实参被忽略掉
```

```

2.     print(a, b)
3. end
4.
5. local function fun2(a, b, c, d) --四个形参, 没有被实参初始化的形参, 用nil初始化
6.     print(a, b, c, d)
7. end
8.
9. local x = 1
10. local y = 2
11. local z = 3
12.
13. fun1(x, y, z)           -- z被函数fun1忽略掉了, 参数变成 x, y
14. fun2(x, y, z)          -- 后面自动加上一个nil, 参数变成 x, y, z, nil
15.
16. -->output
17. 1  2
18. 1  2  3  nil

```

变长参数

上面函数的参数都是固定的, 其实 Lua 还支持变长参数。若形参为 `...`, 表示该函数可以接收不同长度的参数。访问参数的时候也要使用 `...`。

示例代码:

```

1. local function func( ... )           -- 形参为 ... ,表示函数采用变长参数
2.
3.     local temp = {...}               -- 访问的时候也要使用 ...
4.     local ans = table.concat(temp, " ") -- 使用 table.concat 库函数对数
5.                                         -- 组内容使用 " " 拼接成字符串。
6.     print(ans)
7. end
8.
9. func(1, 2)                          -- 传递了两个参数
10. func(1, 2, 3, 4)                   -- 传递了四个参数
11.
12. -->output
13. 1 2
14.
15. 1 2 3 4

```

值得一提的是, LuaJIT 2 尚不能 JIT 编译这种变长参数的用法, 只能解释执行。所以对性能敏感的代码, 应当避免使用此种形式。

具名参数

Lua 还支持通过名称来指定实参，这时候要把所有的实参组织到一个 table 中，并将这个 table 作为唯一的实参传给函数。

示例代码：

```

1. local function change(arg) -- change 函数，改变长方形的长和宽，使其各增长一倍
2.   arg.width = arg.width * 2
3.   arg.height = arg.height * 2
4.   return arg
5. end
6.
7. local rectangle = { width = 20, height = 15 }
8. print("before change:", "width =", rectangle.width,
9.       "height =", rectangle.height)
10. rectangle = change(rectangle)
11. print("after change:", "width =", rectangle.width,
12.       "height =", rectangle.height)
13.
14. -->output
15. before change: width = 20 height = 15
16. after change: width = 40 height = 30

```

按引用传递

当函数参数是 table 类型时，传递进来的是 实际参数的引用，此时在函数内部对该 table 所做的修改，会直接对调用者所传递的实际参数生效，而无需自己返回结果和让调用者进行赋值。我们把上面改变长方形长和宽的例子修改一下。

示例代码：

```

1. function change(arg) --change函数，改变长方形的长和宽，使其各增长一倍
2.   arg.width = arg.width * 2 --表arg不是表rectangle的拷贝，他们是同一个表
3.   arg.height = arg.height * 2
4. end -- 没有return语句了
5.
6. local rectangle = { width = 20, height = 15 }
7. print("before change:", "width = ", rectangle.width,
8.       " height = ", rectangle.height)
9. change(rectangle)
10. print("after change:", "width = ", rectangle.width,
11.       " height =", rectangle.height)
12.
13. --> output
14. before change: width = 20 height = 15
15. after change: width = 40 height = 30

```

在常用基本类型中，除了 `table` 是按址传递类型外，其它的都是按值传递参数。
用全局变量来代替函数参数的不好编程习惯应该被抵制，良好的编程习惯应该是减少全局变量的使用。

函数返回值

- [函数返回值](#)

函数返回值

Lua 具有一项与众不同的特性，允许函数返回多个值。Lua 的库函数中，有一些就是返回多个值。

示例代码：使用库函数 `string.find`，在源字符串中查找目标字符串，若查找成功，则返回目标字符串在源字符串中的起始位置和结束位置的下标。

```
1. local s, e = string.find("hello world", "llo")
2. print(s, e) -->output 3 5
```

返回多个值时，值之间用“,”隔开。

示例代码：定义一个函数，实现两个变量交换值

```
1. local function swap(a, b)    -- 定义函数 swap，实现两个变量交换值
2.     return b, a              -- 按相反顺序返回变量的值
3. end
4.
5. local x = 1
6. local y = 20
7. x, y = swap(x, y)            -- 调用 swap 函数
8. print(x, y)                  --> output 20 1
```

当函数返回值的个数和接收返回值的变量的个数不一致时，Lua 也会自动调整参数个数。

调整规则：

若返回值个数大于接收变量的个数，多余的返回值会被忽略掉；

若返回值个数小于参数个数，从左向右，没有被返回值初始化的变量会被初始化为 nil。

示例代码：

```
1. function init()              --init 函数 返回两个值 1 和 "lua"
2.     return 1, "lua"
3. end
4.
5. x = init()
6. print(x)
7.
8. x, y, z = init()
9. print(x, y, z)
```



```

10.
11. --output
12. 1
13. 1 lua nil

```

当一个函数有一个以上返回值，且函数调用不是一个列表表达式的最后一个元素，那么函数调用只会产生一个返回值，也就是第一个返回值。

示例代码：

```

1. local function init()      -- init 函数 返回两个值 1 和 "lua"
2.     return 1, "lua"
3. end
4.
5. local x, y, z = init(), 2  -- init 函数的位置不在最后，此时只返回 1
6. print(x, y, z)            -->output 1 2 nil
7.
8. local a, b, c = 2, init()  -- init 函数的位置在最后，此时返回 1 和 "lua"
9. print(a, b, c)            -->output 2 1 lua

```

函数调用的实参列表也是一个列表表达式。考虑下面的例子：

```

1. local function init()
2.     return 1, "lua"
3. end
4.
5. print(init(), 2)  -->output 1 2
6. print(2, init()) -->output 2 1 lua

```

如果你确保只取函数返回值的第一个值，可以使用括号运算符，例如

```

1. local function init()
2.     return 1, "lua"
3. end
4.
5. print((init()), 2)  -->output 1 2
6. print(2, (init())) -->output 2 1

```

值得一提的是，如果实参列表中某个函数会返回多个值，同时调用者又没有显式地使用括号运算符来筛选和过滤，则这样的表达式是不能被 **LuaJIT 2** 所 **JIT** 编译的，而只能被解释执行。

全动态函数调用

- [全动态函数调用](#)
 - [使用场景](#)
 - [小试牛刀](#)

全动态函数调用

调用回调函数，并把一个数组参数作为回调函数的参数。

```
1. local args = {...} or {}
2. method_name(unpack(args, 1, table.maxn(args)))
```

使用场景

如果你的实参 `table` 中确定没有 `nil` 空洞，则可以简化为

```
1. method_name(unpack(args))
```

1. 你要调用的函数参数是未知的；
2. 函数的实际参数的类型和数目也都是未知的。

伪代码

```
1. add_task(end_time, callback, params)
2.
3. if os.time() >= endTime then
4.     callback(unpack(params, 1, table.maxn(params)))
5. end
```

值得一提的是，`unpack` 内建函数还不能为 LuaJIT 所 JIT 编译，因此这种用法总是会被解释执行。对性能敏感的代码路径应避免这种用法。

小试牛刀

```
1. local function run(x, y)
2.     print('run', x, y)
3. end
4.
5. local function attack(targetId)
6.     print('targetId', targetId)
```

```
7. end
8.
9. local function do_action(method, ...)
10.     local args = {...} or {}
11.     method(unpack(args, 1, table.maxn(args)))
12. end
13.
14. do_action(run, 1, 2)           -- output: run 1 2
15. do_action(attack, 1111)       -- output: targetId 1111
```

模块

- [模块](#)
 - [require 函数](#)

模块

从 Lua 5.1 语言添加了对模块和包的支持。一个 Lua 模块的数据结构是用一个 Lua 值（通常是一个 Lua 表或者 Lua 函数）。一个 Lua 模块代码就是一个会返回这个 Lua 值的代码块。

可以使用内建函数 `require()` 来加载和缓存模块。简单的说，一个代码模块就是一个程序库，可以通过 `require` 来加载。模块加载后的结果通过是一个 Lua table，这个表就像是一个命名空间，其内容就是模块中导出的所有东西，比如函数和变量。`require` 函数会返回 Lua 模块加载后的结果，即用于表示该 Lua 模块的 Lua 值。

require 函数

Lua 提供了一个名为 `require` 的函数用来加载模块。要加载一个模块，只需要简单地调用 `require` “file” 就可以了，file 指模块所在的文件名。这个调用会返回一个由模块函数组成的 table，并且还会定义一个包含该 table 的全局变量。

在 Lua 中创建一个模块最简单的方法是：创建一个 table，并将所有需要导出的函数放入其中，最后返回这个 table 就可以了。相当于将导出的函数作为 table 的一个字段，在 Lua 中函数是第一类值，提供了天然的优势。

把下面的代码保存在文件 `my.lua` 中

```
1. local _M = {}
2.
3. local function get_name()
4.     return "Lucy"
5. end
6.
7. function _M.greeting()
8.     print("hello " .. get_name())
9. end
10.
11. return _M
```

把下面代码保存在文件 `main.lua` 中，然后执行 `main.lua`，调用上述模块。

```
1. local my_module = require("my")
2. my_module.greeting()    -->output: hello Lucy
```

注：对于需要导出给外部使用的公共模块，处于安全考虑，是要避免全局变量的出现。
我们可以使用 `lj-releng` 或 `luacheck` 工具完成全局变量的检测。至于如何做，到后面再讲。

另一个要注意的是，由于在 `LuaJIT` 中，`require` 函数内不能进行上下文切换，所以不能够在模块的顶级上下文中调用 `cosocket` 一类的 API。

否则会报 `attempt to yield across C-call boundary` 错误。

String 库

- String 库
 - `string.byte(s [, i [, j]])`
 - `string.char (...)`
 - `string.upper(s)`
 - `string.lower(s)`
 - `string.len(s)`
 - `string.find(s, p [, init [, plain]])`
 - `string.format(formatstring, ...)`
 - `string.match(s, p [, init])`
 - `string.gmatch(s, p)`
 - `string.rep(s, n)`
 - `string.sub(s, i [, j])`
 - `string.gsub(s, p, r [, n])`
 - `string.reverse (s)`

String 库

Lua 字符串库包含很多强大的字符操作函数。字符串库中的所有函数都导出在模块 `string` 中。在 Lua 5.1 中，它还将这些函数导出作为 `string` 类型的方法。这样假设要返回一个字符串转的大写形式，可以写成 `ans = string.upper(s)`，也能写成 `ans = s:upper()`。为了避免与之前版本不兼容，此处使用前者。

Lua 字符串总是由字节构成的。Lua 核心并不尝试理解具体的字符集编码（比如 GBK 和 UTF-8 这样的多字节字符编码）。

需要特别注意的一点是，Lua 字符串内部用来标识各个组成字节的下标是从 1 开始的，这不同于像 C 和 Perl 这样的编程语言。这样数字串位置的时候再也不用调整，对于非专业的开发者来说可能也是一个好事情，`string.sub(str, 3, 7)` 直接表示从第三个字符开始到第七个字符（含）为止的子串。

`string.byte(s [, i [, j]])`

返回字符 `s[i]`、`s[i + 1]`、`s[i + 2]`、……、`s[j]` 所对应的 ASCII 码。`i` 的默认值为 1，即第一个字节，`j` 的默认值为 `i`。

示例代码

```
1. print(string.byte("abc", 1, 3))
2. print(string.byte("abc", 3)) -- 缺少第三个参数，第三个参数默认与第二个相同，此时为 3
3. print(string.byte("abc"))    -- 缺少第二个和第三个参数，此时这两个参数都默认为 1
```

```

4.
5. -->output
6. 97    98    99
7. 99
8. 97

```

由于 `string.byte` 只返回整数，而并不像 `string.sub` 等函数那样（尝试）创建新的 Lua 字符串，

因此使用 `string.byte` 来进行字符串相关的扫描和分析是最为高效的，尤其是在被 LuaJIT 2 所 JIT 编译之后。

string.char (...)

接收 0 个或更多的整数（整数范围：0~255），返回这些整数所对应的 ASCII 码字符组成的字符串。当参数为空时，默认是一个 0。

示例代码

```

1. print(string.char(96, 97, 98))
2. print(string.char())          -- 参数为空，默认是一个0，
3.                               -- 你可以用string.byte(string.char())测试一下
4. print(string.char(65, 66))
5.
6. --> output
7. `ab
8.
9. AB

```

此函数特别适合从具体的字节构造出二进制字符串。这经常比使用 `table.concat` 函数和 `..` 连接运算符更加高效。

string.upper(s)

接收一个字符串 `s`，返回一个把所有小写字母变成大写字母的字符串。

示例代码

```

1. print(string.upper("Hello Lua")) -->output  HELLO  LUA

```

string.lower(s)

接收一个字符串 `s`，返回一个把所有大写字母变成小写字母的字符串。

示例代码


```
1. print(string.lower("Hello Lua")) -->output    hello lua
```

string.len(s)

接收一个字符串，返回它的长度。

示例代码

```
1. print(string.len("hello lua")) -->output    9
```

使用此函数是不推荐的。应当总是使用 `#` 运算符来获取 Lua 字符串的长度。

由于 Lua 字符串的长度是专门存放的，并不需要像 C 字符串那样即时计算，因此获取字符串长度的操作总是 $O(1)$ 的时间复杂度。

string.find(s, p [, init [, plain]])

在 s 字符串中第一次匹配 p 字符串。若匹配成功，则返回 p 字符串在 s 字符串中出现的开始位置和结束位置；若匹配失败，则返回 nil。

第三个参数 init 默认为 1，并且可以为负整数，当 init 为负数时，表示从 s 字符串的 `string.len(s) + init + 1` 索引处开始向后匹配字符串 p。

第四个参数默认为 false，当其为 true 时，只会把 p 看成一个字符串对待。

示例代码

```
1. local find = string.find
2. print(find("abc cba", "ab"))
3. print(find("abc cba", "ab", 2))    -- 从索引为2的位置开始匹配字符串：ab
4. print(find("abc cba", "ba", -1))    -- 从索引为7的位置开始匹配字符串：ba
5. print(find("abc cba", "ba", -3))    -- 从索引为5的位置开始匹配字符串：ba
6. print(find("abc cba", "(%a+)", 1))  -- 从索引为1处匹配最长连续且只含字母的字符串
7. print(find("abc cba", "(%a+)", 1, true)) -- 从索引为1的位置开始匹配字符串：(%a+)
8.
9. -->output
10. 1    2
11. nil
12. nil
13. 6    7
14. 1    3    abc
15. nil
```

对于 LuaJIT 这里有个性能优化点，对于 string.find 方法，当只有字符串查找匹配时，是可以被 JIT 编译器优化的，有关 JIT 可以编译优化清单，大家可以参考

<http://wiki.luajit.org/NYI>，性能提升是非常明显的，通常是 100 倍量级。这里有个的例

子, 大家可以参考 <https://groups.google.com/forum/m/#!topic/openresty-en/rwS88FGRsUI>。

string.format(formatstring, ...)

按照格式化参数 `formatstring`, 返回后面 `...` 内容的格式化版本。编写格式化字符串的规则与标准 `c` 语言中 `printf` 函数的规则基本相同: 它由常规文本和指示组成, 这些指示控制了每个参数应放到格式化结果的什么位置, 及如何放入它们。一个指示由字符 `%` 加上一个字母组成, 这些字母指定了如何格式化参数, 例如 `d` 用于十进制数、`x` 用于十六进制数、`o` 用于八进制数、`f` 用于浮点数、`s` 用于字符串等。在字符 `%` 和字母之间可以再指定一些其他选项, 用于控制格式的细节。

示例代码

```
1. print(string.format("%.4f", 3.1415926))      -- 保留4位小数
2. print(string.format("%d %x %o", 31, 31, 31))-- 十进制数31转换成不同进制
3. d = 29; m = 7; y = 2015                      -- 一行包含几个语句, 用 ; 分开
4. print(string.format("%s %02d/%02d/%d", "today is:", d, m, y))
5.
6. -->output
7. 3.1416
8. 31 1f 37
9. today is: 29/07/2015
```

string.match(s, p [, init])

在字符串 `s` 中匹配 (模式) 字符串 `p`, 若匹配成功, 则返回目标字符串中与模式匹配的子串; 否则返回 `nil`。第三个参数 `init` 默认为 1, 并且可以为负整数, 当 `init` 为负数时, 表示从 `s` 字符串的 `string.len(s) + init + 1` 索引处开始向后匹配字符串 `p`。

示例代码

```
1. print(string.match("hello lua", "lua"))
2. print(string.match("lua lua", "lua", 2)) -- 匹配后面那个lua
3. print(string.match("lua lua", "hello"))
4. print(string.match("today is 27/7/2015", "%d+/%d+/%d+"))
5.
6. -->output
7. lua
8. lua
9. nil
10. 27/7/2015
```

`string.match` 目前并不能被 JIT 编译, 应尽量使用 `ngx_lua` 模块提供的 `ngx.re.match` 等接口。

string.gmatch(s, p)

返回一个迭代器函数，通过这个迭代器函数可以遍历到在字符串 `s` 中出现模式串 `p` 的所有地方。

示例代码

```
1. s = "hello world from Lua"
2. for w in string.gmatch(s, "%a+") do --匹配最长连续且只含字母的字符串
3.     print(w)
4. end
5.
6. -->output
7. hello
8. world
9. from
10. Lua
11.
12.
13. t = {}
14. s = "from=world, to=Lua"
15. for k, v in string.gmatch(s, "(%a+)=(%a+)") do --匹配两个最长连续且只含字母的
16.     t[k] = v --字符串，它们之间用等号连接
17. end
18. for k, v in pairs(t) do
19.     print (k,v)
20. end
21.
22. -->output
23. to      Lua
24. from    world
```

此函数目前并不能被 LuaJIT 所 JIT 编译，而只能被解释执行。应尽量使用 `ngx_lua` 模块提供的 `ngx.re.gmatch` 等接口。

string.rep(s, n)

返回字符串 `s` 的 `n` 次拷贝。

示例代码

```
1. print(string.rep("abc", 3)) --拷贝3次"abc"
2.
3. -->output  abcabcab
```

string.sub(s, i [, j])

返回字符串 `s` 中，索引 `i` 到索引 `j` 之间的子字符串。当 `j` 缺省时，默认为 `-1`，也就是字符串 `s` 的最后位置。`i` 可以为负数。当索引 `i` 在字符串 `s` 的位置在索引 `j` 的后面时，将返回一个空字符串。

示例代码

```
1. print(string.sub("Hello Lua", 4, 7))
2. print(string.sub("Hello Lua", 2))
3. print(string.sub("Hello Lua", 2, 1))    --看到返回什么了吗
4. print(string.sub("Hello Lua", -3, -1))
5.
6. -->output
7. lo L
8. ello Lua
9.
10. Lua
```

如果你只是想对字符串中的单个字节进行检查，使用 `string.char` 函数通常会更为高效。

string.gsub(s, p, r [, n])

将目标字符串 `s` 中所有的子串 `p` 替换成字符串 `r`。可选参数 `n`，表示限制替换次数。返回值有两个，第一个是被替换后的字符串，第二个是替换了多少次。

示例代码

```
1. print(string.gsub("Lua Lua Lua", "Lua", "hello"))
2. print(string.gsub("Lua Lua Lua", "Lua", "hello", 2)) --指明第四个参数
3.
4. -->output
5. hello hello hello    3
6. hello hello Lua      2
```

此函数不能为 LuaJIT 所 JIT 编译，而只能被解释执行。一般我们推荐使用 `ngx_lua` 模块提供的 `ngx.re.gsub` 函数。

string.reverse (s)

接收一个字符串 `s`，返回这个字符串的反转。

示例代码

```
1. print(string.reverse("Hello Lua")) --> output: auL olleH
```


Table 库

- `table` 库
 - 下标从 1 开始
 - `table.getn` 获取长度
 - `table.concat (table [, sep [, i [, j]]])`
 - `table.insert (table, [pos [,] value)`
 - `table.maxn (table)`
 - `table.remove (table [, pos])`
 - `table.sort (table [, comp])`
 - `table` 其他非常有用的函数

table 库

`table` 库是由一些辅助函数构成的，这些函数将 `table` 作为数组来操作。

下标从 1 开始

在 *Lua* 中，数组下标从 1 开始计数。

官方解释: Lua lists have a base index of 1 because it was thought to be most friendly for non-programmers, as it makes indices correspond to ordinal element positions.

确实，对于我们数数来说，总是从 1 开始数的，而从 0 开始对于描述偏移量这样的东西有利。而 *Lua* 最初设计是一种类似 XML 的数据描述语言，所以索引 (index) 反应的是数据在里面的位置，而不是偏移量。

在初始化一个数组的时候，若不显式地用键值对方式赋值，则会默认用数字作为下标，从 1 开始。由于在 *Lua* 内部实际采用哈希表和数组分别保存键值对、普通值，所以不推荐混合使用这两种赋值方式。

```
1. local color={first="red", "blue", third="green", "yellow"}
2. print(color["first"])           --> output: red
3. print(color[1])                 --> output: blue
4. print(color["third"])          --> output: green
5. print(color[2])                 --> output: yellow
6. print(color[3])                 --> output: nil
```

从其他语言过来的开发者会觉得比较坑的一点是，当我们把 `table` 当作栈或者队列使用的时候，容易犯错，追加到 `table` 的末尾用的是 `s[#s+1] = something`，而不是 `s[#s] = something`，而且如果这个 `something` 是一个 `nil` 的话，会导致这一次压栈（或者入队列）没有存入任何东西，

#s 的值没有变。如果 `s = { 1, 2, 3, 4, 5, 6 }`，你令 `s[4] = nil`，#s 会令你“匪夷所思”地变成 3。

table.getn 获取长度

取长度操作符写作一元操作 #。字符串的长度是它的字节数（就是以 一个字符一个字节计算的字符串长度）。

对于常规的数组，里面从 1 到 n 放着一些非空的值的时候，它的长度就精确的为 n，即最后一个值的下标。如果数组有一个“空洞”（就是说，nil 值被夹在非空值之间），那么 #t 可能是指向任何一个 nil 值的前一个位置的下标（就是说，任何一个 nil 值都有可能被当成数组的结束）。这也就说明对于有“空洞”的情况，table 的长度存在一定的不可确定性。

```
1. local tblTest1 = { 1, a = 2, 3 }
2. print("Test1 " .. table.getn(tblTest1))
3.
4. local tblTest2 = { 1, nil }
5. print("Test2 " .. table.getn(tblTest2))
6.
7. local tblTest3 = { 1, nil, 2 }
8. print("Test3 " .. table.getn(tblTest3))
9.
10. local tblTest4 = { 1, nil, 2, nil }
11. print("Test4 " .. table.getn(tblTest4))
12.
13. local tblTest5 = { 1, nil, 2, nil, 3, nil }
14. print("Test5 " .. table.getn(tblTest5))
15.
16. local tblTest6 = { 1, nil, 2, nil, 3, nil, 4, nil }
17. print("Test6 " .. table.getn(tblTest6))
```

我们使用 Lua 5.1 和 LuaJIT 2.1 分别执行这个用例，结果如下：

```
1. # lua test.lua
2. Test1 2
3. Test2 1
4. Test3 3
5. Test4 1
6. Test5 3
7. Test6 1
8. # luajit test.lua
9. Test1 2
10. Test2 1
11. Test3 1
12. Test4 1
13. Test5 1
```

14. Test6 1

这一段的输出结果，就是这么 匪夷所思。请问，你以后还敢在 Lua 的 table 中用 nil 值吗？如果你继续往后面加 nil，你可能会发现点什么。你可能认为你发现的是个规律。但是，你千万不要认为这是个规律，因为这是错误的。

不要在 Lua 的 table 中使用 nil 值，如果一个元素要删除，直接 **remove**，不要用 **nil** 去代替。

table.concat (table [, sep [, i [, j]]])

对于元素是 string 或者 number 类型的表 table，返回 `table[i]..sep..table[i+1] ... sep..table[j]` 连接成的字符串。填充字符串 sep 默认为空白字符串。起始索引位置 i 默认为 1，结束索引位置 j 默认是 table 的长度。如果 i 大于 j，返回一个空字符串。

示例代码

```
1. local a = {1, 3, 5, "hello" }
2. print(table.concat(a))           -- output: 135hello
3. print(table.concat(a, "|"))      -- output: 1|3|5|hello
4. print(table.concat(a, " ", 4, 2)) -- output:
5. print(table.concat(a, " ", 2, 4)) -- output: 3 5 hello
```

table.insert (table, [pos ,] value)

在 (数组型) 表 table 的 pos 索引位置插入 value，其它元素向后移动到空的地方。pos 的默认值是表的长度加一，即默认是插在表的最后。

示例代码

```
1. local a = {1, 8}           -- a[1] = 1, a[2] = 8
2. table.insert(a, 1, 3)      -- 在表索引为1处插入3
3. print(a[1], a[2], a[3])
4. table.insert(a, 10)        -- 在表的最后插入10
5. print(a[1], a[2], a[3], a[4])
6.
7. -->output
8. 3    1    8
9. 3    1    8    10
```

table.maxn (table)

返回 (数组型) 表 table 的最大索引编号；如果此表没有正的索引编号，返回 0。

当长度省略时，此函数通常需要 `O(n)` 的时间复杂度来计算 table 的末尾。因此用这个函数省

略索引位置的调用形式来作 table 元素的末尾追加，是高代价操作。

示例代码

```
1. local a = {}
2. a[-1] = 10
3. print(table.maxn(a))
4. a[5] = 10
5. print(table.maxn(a))
6.
7. -->output
8. 0
9. 5
```

此函数的行为不同于 `#` 运算符，因为 `#` 可以返回数组中任意一个 nil 空洞或最后一个 nil 之前的元素索引。当然，该函数的开销相比 `#` 运算符也会更大一些。

table.remove (table [, pos])

在表 table 中删除索引为 pos (pos 只能是 number 型) 的元素，并返回这个被删除的元素，它后面所有元素的索引值都会减一。pos 的默认值是表的长度，即默认是删除表的最后一个元素。

示例代码

```
1. local a = { 1, 2, 3, 4}
2. print(table.remove(a, 1)) --删除速索引为1的元素
3. print(a[1], a[2], a[3], a[4])
4.
5. print(table.remove(a)) --删除最后一个元素
6. print(a[1], a[2], a[3], a[4])
7.
8. -->output
9. 1
10. 2 3 4 nil
11. 4
12. 2 3 nil nil
```

table.sort (table [, comp])

按照给定的比较函数 comp 给表 table 排序，也就是从 table[1] 到 table[n]，这里 n 表示 table 的长度。

比较函数有两个参数，如果希望第一个参数排在第二个的前面，就应该返回 true，否则返回 false。

如果比较函数 comp 没有给出，默认从小到大排序。

示例代码

```
1. local function compare(x, y) --从大到小排序
2.     return x > y           --如果第一个参数大于第二个就返回true, 否则返回false
3. end
4.
5. local a = { 1, 7, 3, 4, 25}
6. table.sort(a)              --默认从小到大排序
7. print(a[1], a[2], a[3], a[4], a[5])
8. table.sort(a, compare) --使用比较函数进行排序
9. print(a[1], a[2], a[3], a[4], a[5])
10.
11. -->output
12. 1    3    4    7    25
13. 25   7    4    3    1
```

table 其他非常有用的函数

LuaJIT 2.1 新增加的 `table.new` 和 `table.clear` 函数是非常有用的。前者主要用来预分配 Lua table 空间，后者主要用来高效的释放 table 空间，并且它们都是可以被 JIT 编译的。具体可以参考一下 OpenResty 捆绑的 `lua-resty-*` 库，里面有些实例可以作为参考。

日期时间函数

- 日期时间函数
 - `os.time ([table])`
 - `os.difftime (t2, t1)`
 - `os.date ([format [, time]])`

日期时间函数

在 Lua 中，函数 `time`、`date` 和 `difftime` 提供了所有的日期和时间功能。

在 OpenResty 的世界里，不推荐使用这里的标准时间函数，因为这些函数通常会引发不止一个昂贵的系统调用，同时无法为 LuaJIT JIT 编译，对性能造成较大影响。推荐使用 `ngx_lua` 模块提供的带缓存的时间接口，如 `ngx.today` , `ngx.time` , `ngx.utctime` , `ngx.localtime` , `ngx.now` , `ngx.http_time` , 以及 `ngx.cookie_time` 等。

所以下面的部分函数，简单了解一下即可。

os.time ([table])

如果不使用参数 `table` 调用 `time` 函数，它会返回当前的时间和日期（它表示从某一时刻到现在的秒数）。如果用 `table` 参数，它会返回一个数字，表示该 `table` 中所描述的日期和时间（它表示从某一时刻到 `table` 中描述日期和时间的秒数）。`table` 的字段如下：

字段名称	取值范围
<code>year</code>	四位数字
<code>month</code>	1–12
<code>day</code>	1–31
<code>hour</code>	0–23
<code>min</code>	0–59
<code>sec</code>	0–61
<code>isdst</code>	<code>boolean</code> (<code>true</code> 表示夏令时)

对于 `time` 函数，如果参数为 `table`，那么 `table` 中必须含有 `year`、`month`、`day` 字段。其他字缺省时段默认为中午（12:00:00）。

示例代码：（地点为北京）

```
1. print(os.time())    -->output 1438243393
2. a = { year = 1970, month = 1, day = 1, hour = 8, min = 1 }
```

```
3. print(os.time(a)) -->output 60
```

os.difftime (t2, t1)

返回 t1 到 t2 的时间差，单位为秒。

示例代码：

```
1. local day1 = { year = 2015, month = 7, day = 30 }
2. local t1 = os.time(day1)
3.
4. local day2 = { year = 2015, month = 7, day = 31 }
5. local t2 = os.time(day2)
6. print(os.difftime(t2, t1)) -->output 86400
```

os.date ([format [, time]])

把一个表示日期和时间的数值，转换成更高级的表现形式。其第一个参数 `format` 是一个格式化字符串，描述了要返回的时间形式。第二个参数 `time` 就是日期和时间的数字表示，缺省时默认为当前的时间。使用格式字符 `"*t"`，创建一个时间表。

示例代码：

```
1. local tab1 = os.date("*t") --返回一个描述当前日期和时间的表
2. local ans1 = "{"
3. for k, v in pairs(tab1) do --把tab1转换成一个字符串
4.     ans1 = string.format("%s %s = %s,", ans1, k, tostring(v))
5. end
6.
7. ans1 = ans1 .. "}"
8. print("tab1 = ", ans1)
9.
10.
11. local tab2 = os.date("*t", 360) --返回一个描述日期和时间数为360秒的表
12. local ans2 = "{"
13. for k, v in pairs(tab2) do --把tab2转换成一个字符串
14.     ans2 = string.format("%s %s = %s,", ans2, k, tostring(v))
15. end
16.
17. ans2 = ans2 .. "}"
18. print("tab2 = ", ans2)
19.
20. -->output
21. tab1 = { hour = 17, min = 28, wday = 5, day = 30, month = 7, year = 2015, sec = 10, yday = 211,
    isdst = false,}
22. tab2 = { hour = 8, min = 6, wday = 5, day = 1, month = 1, year = 1970, sec = 0, yday = 1, isdst
    = false,}
```

该表中除了使用到了 `time` 函数参数 `table` 的字段外，这还提供了星期（`wday`，星期天为 1）和一年中的第几天（`yday`，一月一日为 1）。

除了使用 “*t” 格式字符串外，如果使用带标记（见下表）的特殊字符串，`os.date` 函数会将相应的标记位以时间信息进行填充，得到一个包含时间的字符串。

表如下：

格式字符	含义
%a	一星期中天数的简写（例如：Wed）
%A	一星期中天数的全称（例如：Wednesday）
%b	月份的简写（例如：Sep）
%B	月份的全称（例如：September）
%c	日期和时间（例如：07/30/15 16:57:24）
%d	一个月中的第几天[01 ~ 31]
%H	24小时制中的小时数[00 ~ 23]
%I	12小时制中的小时数[01 ~ 12]
%j	一年中的第几天[001 ~ 366]
%M	分钟数[00 ~ 59]
%m	月份数[01 ~ 12]
%p	“上午（am）”或“下午（pm）”
%S	秒数[00 ~ 59]
%w	一星期中的第几天[1 ~ 7 = 星期天 ~ 星期六]
%x	日期（例如：07/30/15）
%X	时间（例如：16:57:24）
%y	两位数的年份[00 ~ 99]
%Y	完整的年份（例如：2015）
%%	字符‘%’

示例代码：

```
1. print(os.date("today is %A, in %B"))
2. print(os.date("now is %x %X"))
3.
4. -->output
5. today is Thursday, in July
6. now is 07/30/15 17:39:22
```


数学库函数

- [数学库](#)

数学库

Lua 数学库由一组标准的数学函数构成。数学库的引入丰富了 Lua 编程语言的功能，同时也方便了程序的编写。常用数学函数见下表：

函数名	函数功能
<code>math.rad(x)</code>	角度x转换成弧度
<code>math.deg(x)</code>	弧度x转换成角度
<code>math.max(x, ...)</code>	返回参数中值最大的那个数，参数必须是number型
<code>math.min(x, ...)</code>	返回参数中值最小的那个数，参数必须是number型
<code>math.random ([m, n])</code>	不传入参数时，返回 一个在区间[0, 1)内均匀分布的伪随机实数；只使用一个整数参数m时，返回一个在区间[1, m]内均匀分布的伪随机整数；使用两个整数参数时，返回一个在区间[m, n]内均匀分布的伪随机整数
<code>math.randomseed (x)</code>	为伪随机数生成器设置一个种子x，相同的种子将会生成相同的数字序列
<code>math.abs(x)</code>	返回x的绝对值
<code>math.fmod(x, y)</code>	返回 x对y取余数
<code>math.pow(x, y)</code>	返回x的y次方
<code>math.sqrt(x)</code>	返回x的算术平方根
<code>math.exp(x)</code>	返回自然数e的x次方
<code>math.log(x)</code>	返回x的自然对数
<code>math.log10(x)</code>	返回以10为底，x的对数
<code>math.floor(x)</code>	返回最大且不大于x的整数
<code>math.ceil(x)</code>	返回最小且不小于x的整数
<code>math.pi</code>	圆周率
<code>math.sin(x)</code>	求弧度x的正弦值
<code>math.cos(x)</code>	求弧度x的余弦值
<code>math.tan(x)</code>	求弧度x的正切值
<code>math.asin(x)</code>	求x的反正弦值
<code>math.acos(x)</code>	求x的反余弦值
<code>math.atan(x)</code>	求x的反正切值

示例代码：

```

1. print(math.pi)           -->output  3.1415926535898
2. print(math.rad(180))     -->output  3.1415926535898
3. print(math.deg(math.pi)) -->output  180
4.
5. print(math.sin(1))        -->output  0.8414709848079
6. print(math.cos(math.pi)) -->output  -1
7. print(math.tan(math.pi / 4)) -->output  1
8.
9. print(math.atan(1))       -->output  0.78539816339745
10. print(math.asin(0))      -->output  0
11.
12. print(math.max(-1, 2, 0, 3.6, 9.1)) -->output  9.1
13. print(math.min(-1, 2, 0, 3.6, 9.1)) -->output  -1
14.
15. print(math.fmod(10.1, 3)) -->output  1.1
16. print(math.sqrt(360))    -->output  18.97366596101
17.
18. print(math.exp(1))        -->output  2.718281828459
19. print(math.log(10))       -->output  2.302585092994
20. print(math.log10(10))     -->output  1
21.
22. print(math.floor(3.1415)) -->output  3
23. print(math.ceil(7.998))   -->output  8

```

另外使用 `math.random()` 函数获得伪随机数时，如果不使用 `math.randomseed()` 设置伪随机数生成种子或者设置相同的伪随机数生成种子，那么得到的伪随机数序列是一样的。

示例代码：

```

1. math.randomseed (100) --把种子设置为100
2. print(math.random())   -->output  0.0012512588885159
3. print(math.random(100)) -->output  57
4. print(math.random(100, 360)) -->output  150

```

稍等片刻，再次运行上面的代码。

```

1. math.randomseed (100) --把种子设置为100
2. print(math.random())   -->output  0.0012512588885159
3. print(math.random(100)) -->output  57
4. print(math.random(100, 360)) -->output  150

```

两次运行的结果一样。为了避免每次程序启动时得到的都是相同的伪随机数序列，通常是使用当前时间作为种子。

修改上例中的代码：

```
1.  
2. math.randomseed (os.time())  --把100换成os.time()  
3. print(math.random())          -->output 0.88369396038697  
4. print(math.random(100))       -->output 66  
5. print(math.random(100, 360))  -->output 228
```

稍等片刻，再次运行上面的代码。

```
1. math.randomseed (os.time())  --把100换成os.time()  
2. print(math.random())          -->output 0.88946195867794  
3. print(math.random(100))       -->output 68  
4. print(math.random(100, 360))  -->output 129
```

文件操作

- 文件操作
 - 隐式文件描述
 - 显式文件描述
 - 文件操作函数
 - `io.open (filename [, mode])`
 - `file:close ()`
 - `io.close ([file])`
 - `file:flush ()`
 - `io.flush ()`
 - `io.input ([file])`
 - `file:lines ()`
 - `io.lines ([filename])`
 - `io.output ([file])`
 - `file:read (...)`
 - `io.read (...)`
 - `io.type (obj)`
 - `file:write (...)`
 - `io.write (...)`
 - `file:seek ([whence] [, offset])`
 - `file:setvbuf (mode [, size])`

文件操作

Lua I/O 库提供两种不同的方式处理文件：隐式文件描述，显式文件描述。

这些文件 I/O 操作，在 OpenResty 的上下文中对事件循环是会产生阻塞效应。OpenResty 比较擅长的是高并发网络处理，在这个环境中，任何文件的操作，都将阻塞其他并行执行的请求。实际中的应用，在 OpenResty 项目中应尽可能让网络处理部分、文件 I/O 操作部分相互独立，不要揉和在一起。

隐式文件描述

设置一个默认的输出或输入文件，然后在这个文件上进行所有的输入或输出操作。所有的操作函数由 `io` 表提供。

打开已经存在的 `test1.txt` 文件，并读取里面的内容

```
1. file = io.input("test1.txt")    -- 使用 io.input() 函数打开文件
2.
```

```

3. repeat
4.     line = io.read()          -- 逐行读取内容, 文件结束时返回nil
5.     if nil == line then
6.         break
7.     end
8.     print(line)
9. until (false)
10.
11. io.close(file)              -- 关闭文件
12.
13. --> output
14. my test file
15. hello
16. lua

```

在 `test1.txt` 文件的最后添加一行 `"hello world"`

```

1. file = io.open("test1.txt", "a")  -- 使用 io.open() 函数, 以添加模式打开文件
2. io.output(file)                  -- 使用 io.output() 函数, 设置默认输出文件
3. io.write("\nhello world")        -- 使用 io.write() 函数, 把内容写到文件
4. io.close(file)

```

在相应目录下打开 `test1.txt` 文件, 查看文件内容发生的变化。

显式文件描述

使用 `file:XXX()` 函数方式进行操作, 其中 `file` 为 `io.open()` 返回的文件句柄。

打开已经存在的 `test2.txt` 文件, 并读取里面的内容

```

1. file = io.open("test2.txt", "r")  -- 使用 io.open() 函数, 以只读模式打开文件
2.
3. for line in file:lines() do        -- 使用 file:lines() 函数逐行读取文件
4.     print(line)
5. end
6.
7. file:close()
8.
9. -->output
10. my test2
11. hello lua

```

在 `test2.txt` 文件的最后添加一行 `"hello world"`

```

1. file = io.open("test2.txt", "a")  -- 使用 io.open() 函数, 以添加模式打开文件
2. file:write("\nhello world")      -- 使用 file:write() 函数, 在文件末尾追加内容

```

```
3. file:close()
```

在相应目录下打开 `test2.txt` 文件，查看文件内容发生的变化。

文件操作函数

`io.open (filename [, mode])`

按指定的模式 `mode`，打开一个文件名为 `filename` 的文件，成功则返回文件句柄，失败则返回 `nil` 加错误信息。模式：

模式	含义	文件不存在时
"r"	读模式（默认）	返回nil加错误信息
"w"	写模式	创建文件
"a"	添加模式	创建文件
"r+"	更新模式，保存之前的数据	返回nil加错误信息
"w+"	更新模式，清除之前的数据	创建文件
"a+"	添加更新模式，保存之前的数据，在文件尾进行添加	创建文件

模式字符串后面可以有一个 'b'，用于在某些系统中打开二进制文件。

注意 "w" 和 "wb" 的区别

- "w" 表示文本文件。某些文件系统(如 Linux 的文件系统)认为 `0x0A` 为文本文件的换行符，Windows 的文件系统认为 `0x0D0A` 为文本文件的换行符。为了兼容其他文件系统（如从 Linux 拷贝来的文件），Windows 的文件系统在写文件时，会在文件中 `0x0A` 的前面加上 `0x0D`。使用 "w"，其属性要看所在的平台。
- "wb" 表示二进制文件。文件系统会按纯粹的二进制格式进行写操作，因此也就不存在格式转换的问题。（Linux 文件系统下 "w" 和 "wb" 没有区别）

`file:close ()`

关闭文件。注意：当文件句柄被垃圾收集后，文件将自动关闭。句柄将变为一个不可预知的值。

`io.close ([file])`

关闭文件，和 `file:close()` 的作用相同。没有参数 `file` 时，关闭默认输出文件。

`file:flush ()`

把写入缓冲区的所有数据写入到文件 `file` 中。

io.flush ()

相当于 `file:flush()`，把写入缓冲区的所有数据写入到默认输出文件。

io.input ([file])

当使用一个文件名调用时，打开这个文件（以文本模式），并设置文件句柄为默认输入文件；

当使用一个文件句柄调用时，设置此文件句柄为默认输入文件；

当不使用参数调用时，返回默认输入文件句柄。

file:lines ()

返回一个迭代函数，每次调用将获得文件中的一行内容，当到文件尾时，将返回 `nil`，但不关闭文件。

io.lines ([filename])

打开指定的文件 `filename` 为读模式并返回一个迭代函数，每次调用将获得文件中的一行内容，当到文件尾时，将返回 `nil`，并自动关闭文件。若不带参数时 `io.lines()` 等价于

`io.input():lines()` 读取默认输入设备的内容，结束时不关闭文件。

io.output ([file])

类似于 `io.input`，但操作在默认输出文件上。

file:read (...)

按指定的格式读取一个文件。按每个格式将返回一个字符串或数字，如果不能正确读取将返回 `nil`，若没有指定格式将指默认按行方式进行读取。格式：

格式	含义
"*n"	读取一个数字
"*a"	从当前位置读取整个文件。若当前位置为文件尾，则返回空字符串
"*l"	读取下一行的内容。若为文件尾，则返回 <code>nil</code> 。（默认）
number	读取指定字节数的字符。若为文件尾，则返回 <code>nil</code> 。如果number为0,则返回空字符串，若为文件尾,则返回 <code>nil</code>

io.read (...)

相当于 `io.input():read`

io.type (obj)

检测 `obj` 是否一个可用的文件句柄。如果 `obj` 是一个打开的文件句柄，则返回 `"file"` 如果 `obj` 是一个已关闭的文件句柄，则返回 `"closed file"` 如果 `obj` 不是一个文件句柄，则返回 `nil`。

`file:write (...)`

把每一个参数的值写入文件。参数必须为字符串或数字，若要输出其它值，则需通过 `tostring` 或 `string.format` 进行转换。

`io.write (...)`

相当于 `io.output():write`。

`file:seek ([whence] [, offset])`

设置和获取当前文件位置，成功则返回最终的文件位置(按字节，相对于文件开头)，失败则返回 `nil` 加错误信息。缺省时，`whence` 默认为 `"cur"`，`offset` 默认为 `0`。

参数 `whence`：

whence	含义
<code>"set"</code>	文件开始
<code>"cur"</code>	文件当前位置(默认)
<code>"end"</code>	文件结束

`file:setvbuf (mode [, size])`

设置输出文件的缓冲模式。模式：

模式	含义
<code>"no"</code>	没有缓冲，即直接输出
<code>"full"</code>	全缓冲，即当缓冲满后才进行输出操作(也可调用 <code>flush</code> 马上输出)
<code>"line"</code>	以行为单位，进行输出

最后两种模式，`size` 可以指定缓冲的大小（按字节），忽略 `size` 将自动调整为最佳的大小。

Nginx

- [Nginx](#)

Nginx

Nginx (“engine x”) 是一个高性能的 HTTP 和反向代理服务器，也是一个 IMAP/POP3/SMTP 代理服务器。Nginx 是由 Igor Sysoev 为俄罗斯著名的 Rambler.ru 站点开发的，第一个公开版本 0.1.0 发布于 2004 年 10 月 4 日。其将源代码以类 BSD 许可证的形式发布，因它的稳定性、丰富的功能集、示例配置文件和低系统资源的消耗而闻名。

由于 Nginx 使用基于事件驱动的架构，能够并发处理百万级别的 TCP 连接，高度模块化的设计和自由的许可证使得扩展 Nginx 功能的第三方模块层出不穷。因此其作为 Web 服务器被广泛应用到大流量的网站上，包括淘宝、腾讯、新浪、京东等访问量巨大的网站。

2015 年 6 月，Netcraft 收到的调查网站有 8 亿多家，主流 web 服务器市场份额（前四名）如下表：

Web服务器	市场占有率
Apache	49.53%
Nginx	13.52%
Microsoft IIS	12.32%
Google Web Server	7.72%

其中在访问量最多的一万个网站中，Nginx 的占有率已超过 Apache。

Nginx 新手起步

- [Nginx 新手起步](#)
 - [为什么选择 Nginx](#)
 - [如何使用 Nginx](#)

Nginx 新手起步

为什么选择 Nginx

为什么选择 Nginx？因为它具有以下特点：

1、处理响应请求很快

在正常的情况下，单次请求会得到更快的响应。在高峰期，Nginx 可以比其它的 Web 服务器更快的响应请求。

2、高并发连接

在互联网快速发展，互联网用户数量不断增加的今天，一些大公司、网站都需要面对高并发请求，如果有一个能够在峰值顶住 10 万以上并发请求的 Server，肯定会得到大家的青睐。理论上，Nginx 支持的并发连接上限取决于你的内存，10 万远未封顶。

3、低的内存消耗

在一般的情况下，10000 个非活跃的 HTTP Keep-Alive 连接在 Nginx 中仅消耗 2.5MB 的内存，这也是 Nginx 支持高并发连接的基础。

4、具有很高的可靠性：

Nginx 是一个高可靠性的 Web 服务器，这也是我们为什么选择 Nginx 的基本条件，现在很多的网站都在使用 Nginx，足以说明 Nginx 的可靠性。高可靠性来自其核心框架代码的优秀设计、模块设计的简单性，并且这些模块都非常的稳定。

5、高扩展性

Nginx 的设计极具扩展性，它完全是由多个不同功能、不同层次、不同类型且耦合度极低的模块组成。这种设计造就了 Nginx 庞大的第三方模块。

6、热部署

master 管理进程与 worker 工作进程的分离设计，使得 Nginx 具有热部署的功能，可以在 7 × 24 小时不间断服务的前提下，升级 Nginx 的可执行文件。也可以在不停止服务的情况下修改配置文件，更换日志文件等功能。

7、自由的 BSD 许可协议

BSD 许可协议不只是允许用户免费使用 Nginx，也允许用户修改 Nginx 源码，还允许用户用于商业用途。

如何使用 Nginx

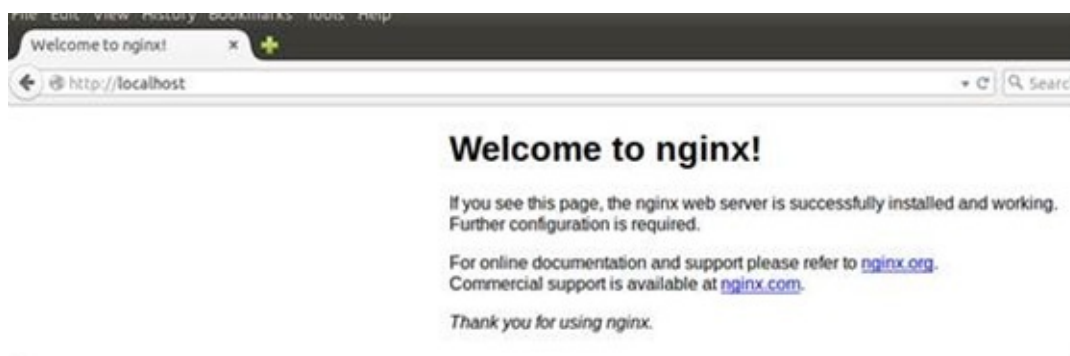
Nginx 安装：

不同系统依赖包可能不同，例如 pcre, zlib, openssl 等。

1. 获取 Nginx，在 <http://nginx.org/en/download.html> 上可以获取当前最新的版本。
2. 解压缩 nginx-xx.tar.gz 包。
3. 进入解压缩目录，执行 ./configure
4. make & make install

若安装时找不到上述依赖模块，使用 `--with-openssl=` `<openssl_dir>`、`--with-pcre=` `<pcre_dir>`、`--with-zlib=` `<zlib_dir>` 指定依赖的模块目录。如已安装过，此处的路径为安装目录；若未安装，则此路径为编译安装包路径，Nginx 将执行模块的默认编译安装。

启动 Nginx 之后，浏览器中输入 <http://localhost> 可以验证是否安装启动成功。



Nginx 配置示例：

安装完成之后，配置目录 conf 下有以下配置文件，过滤掉了 xx.default 配置：

```
1. ubuntu: /opt/nginx-1.7.7/conf$ tree |grep -v default
2. .
3. └─ fastcgi.conf
4. └─ fastcgi_params
5. └─ koi-utf
6. └─ koi-win
7. └─ mime.types
8. └─ nginx.conf
9. └─ scgi_params
10. └─ uwsgi_params
```

```
11. win-utf
```

除了 `nginx.conf`，其余配置文件，一般只需要使用默认提供即可。

`nginx.conf` 是主配置文件，默认配置去掉注释之后的内容如下图所示：

```
1. worker_process      # 表示工作进程的数量，一般设置为cpu的核数
2.
3. worker_connections  # 表示每个工作进程的最大连接数
4.
5. server{}            # 块定义了虚拟主机
6.
7.     listen           # 监听端口
8.
9.     server_name       # 监听域名
10.
11.     location {}      # 是用来为匹配的 URI 进行配置，URI 即语法中的"/uri/"
12.
13.     location /{}     # 匹配任何查询，因为所有请求都以 / 开头
14.
15.         root          # 指定对应uri的资源查找路径，这里html为相对路径，完整路径为
16.                        # /opt/nginx-1.7.7/html/
17.
18.         index          # 指定首页index文件的名称，可以配置多个，以空格分开。如有多
19.                        # 个，按配置顺序查找。
```

真实用例

```
worker_processes 1;

events {
    worker_connections 1024;
}

http {
    include mime.types;
    default_type application/octet-stream;

    sendfile on;
    keepalive_timeout 65;

    server {
        listen 80;
        server_name localhost;

        location / {
            root html;
            index index.html index.htm;
        }

        # redirect server error pages to the static page /50x.html
        error_page 500 502 503 504 /50x.html;
        location = /50x.html {
            root html;
        }
    }
}
```

从配置可以看出，Nginx 监听了 80 端口、域名为 localhost、根路径为 html 文件夹（我的安装路径为 /opt/nginx-1.7.7，所以 /opt/nginx-1.7.7/html）、默认 index 文件为 index.html，index.htm 服务器错误重定向到 50x.html 页面。

可以看到 /opt/nginx-1.7.7/html/ 有以下文件：

```
1. ubuntu:/opt/nginx-1.7.7/html$ ls
2. 50x.html index.html
```

这也是上面在浏览器中输入 `http://localhost`，能够显示欢迎页面的原因。实际上访问的是 `/opt/nginx-1.7.7/html/index.html` 文件。

location 匹配规则

- location 匹配规则
 - 语法规则
 - rewrite 语法
 - redirect 语法
 - 防盗链
 - 根据文件类型设置过期时间
 - 禁止访问某个目录

location 匹配规则

语法规则

```
location [=|~|~*|^~] /uri/ { ... }
```

模式	含义
location = /uri	= 表示精确匹配，只有完全匹配上才能生效
location ^~ /uri	^~ 开头对URL路径进行前缀匹配，并且在正则之前。
location ~ pattern	开头表示区分大小写的正则匹配
location ~* pattern	开头表示不区分大小写的正则匹配
location /uri	不带任何修饰符，也表示前缀匹配，但是在正则匹配之后
location /	通用匹配，任何未匹配到其它location的请求都会匹配到，相当于switch中的default

前缀匹配时，Nginx 不对 url 做编码，因此请求为 `/static/20%/aa`，可以被规则 `^~ /static/`
`/aa` 匹配到（注意是空格）

多个 location 配置的情况下匹配顺序为（参考资料而来，还未实际验证，试试就知道了，不必拘泥，仅供参考）：

- 首先精确匹配 `=`
- 其次前缀匹配 `^~`
- 其次是按文件中顺序的正则匹配
- 然后匹配不带任何修饰的前缀匹配。
- 最后是交给 `/` 通用匹配
- 当有匹配成功时候，停止匹配，按当前匹配规则处理请求

注意：前缀匹配，如果有包含关系时，按最大匹配原则进行匹配。比如在前缀匹配：`location /dir01` 与 `location /dir01/dir02`，如有请求 `http://localhost/dir01/dir02/file` 将最终匹配到 `location /dir01/dir02`

例子，有如下匹配规则：

```
1. location = / {
2.     echo "规则A";
3. }
4. location = /login {
5.     echo "规则B";
6. }
7. location ^~ /static/ {
8.     echo "规则C";
9. }
10. location ^~ /static/files {
11.     echo "规则X";
12. }
13. location ~ \.(gif|jpg|png|js|css)$ {
14.     echo "规则D";
15. }
16. location ~* \.png$ {
17.     echo "规则E";
18. }
19. location /img {
20.     echo "规则Y";
21. }
22. location / {
23.     echo "规则F";
24. }
```

那么产生的效果如下：

- 访问根目录 `/`，比如 `http://localhost/` 将匹配 规则A
- 访问 `http://localhost/login` 将匹配 规则B，`http://localhost/register` 则匹配 规则F
- 访问 `http://localhost/static/a.html` 将匹配 规则C
- 访问 `http://localhost/static/files/a.exe` 将匹配 规则X，虽然 规则C 也能匹配到，但因为最大匹配原则，最终选中了 规则X。你可以测试下，去掉规则 X，则当前 URL 会匹配上 规则C。
- 访问 `http://localhost/a.gif`，`http://localhost/b.jpg` 将匹配 规则D 和 规则 E，但是 规则 D 顺序优先，规则 E 不起作用，而 `http://localhost/static/c.png` 则优先匹配到 规则C
- 访问 `http://localhost/a.PNG` 则匹配 规则 E，而不会匹配 规则 D，因为 规则 E 不区分大小写。
- 访问 `http://localhost/img/a.gif` 会匹配上 规则D，虽然 规则Y 也可以匹配上，但是因为正

则匹配优先，而忽略了 规则Y。

- 访问 `http://localhost/img/a.tiff` 会匹配上 规则Y。

访问 `http://localhost/category/id/1111` 则最终匹配到规则 F，因为以上规则都不匹配，这个时候应该是 Nginx 转发请求给后端应用服务器，比如 FastCGI (php)，tomcat (jsp)，Nginx 作为反向代理服务器存在。

所以实际使用中，笔者觉得至少有三个匹配规则定义，如下：

```
1. # 直接匹配网站根，通过域名访问网站首页比较频繁，使用这个会加速处理，官网如是说。
2. # 这里是直接转发给后端应用服务器了，也可以是一个静态首页
3. # 第一个必选规则
4. location = / {
5.     proxy_pass http://tomcat:8080/index
6. }
7.
8. # 第二个必选规则是处理静态文件请求，这是 nginx 作为 http 服务器的强项
9. # 有两种配置模式，目录匹配或后缀匹配，任选其一或搭配使用
10. location ^~ /static/ {
11.     root /webroot/static/;
12. }
13. location ~* \.(gif|jpg|jpeg|png|css|js|ico)$ {
14.     root /webroot/res/;
15. }
16.
17. # 第三个规则就是通用规则，用来转发动态请求到后端应用服务器
18. # 非静态文件请求就默认是动态请求，自己根据实际把握
19. # 毕竟目前的一些框架的流行，带.php、.jsp后缀的情况很少了
20. location / {
21.     proxy_pass http://tomcat:8080/
22. }
```

rewrite 语法

- last - 基本上都用这个 flag
- break - 中止 rewrite，不再继续匹配
- redirect - 返回临时重定向的 HTTP 状态 302
- permanent - 返回永久重定向的 HTTP 状态 301

1、下面是可以用来判断的表达式：

- ```
1. -f 和 !-f 用来判断是否存在文件
2. -d 和 !-d 用来判断是否存在目录
3. -e 和 !-e 用来判断是否存在文件或目录
4. -x 和 !-x 用来判断文件是否可执行
```

## 2、下面是可以用作判断的全局变量

```

1. 例: http://localhost:88/test1/test2/test.php?k=v
2. $host : localhost
3. $server_port : 88
4. $request_uri : /test1/test2/test.php?k=v
5. $document_uri : /test1/test2/test.php
6. $document_root : D:\nginx/html
7. $request_filename : D:\nginx/html/test1/test2/test.php

```

## redirect 语法

```

1. server {
2. listen 80;
3. server_name start.igrow.cn;
4. index index.html index.php;
5. root html;
6. if ($http_host !~ "^star\.igrow\.cn$") {
7. rewrite ^(.*) http://star.igrow.cn$1 redirect;
8. }
9. }

```

## 防盗链

```

1. location ~* \.(gif|jpg|swf)$ {
2. valid_referers none blocked start.igrow.cn sta.igrow.cn;
3. if ($invalid_referer) {
4. rewrite ^/ http://$host/logo.png;
5. }
6. }

```

## 根据文件类型设置过期时间

```

1. location ~* \.(js|css|jpg|jpeg|gif|png|swf)$ {
2. if (-f $request_filename) {
3. expires 1h;
4. break;
5. }
6. }

```

## 禁止访问某个目录

```

1. location ~* \.(txt|doc){

```

```
2. root /data/www/wwwroot/linuxtone/test;
3. deny all;
4. }
```

一些可用的全局变量，可以参考[获取 Nginx 内置绑定变量](#)章节。



# 静态文件服务

- [Nginx 静态文件服务](#)
  - [文件缓存漫谈](#)
  - [如何安装和配置基础缓存](#)
  - [陈旧总比没有强](#)
  - [缓存微调](#)
  - [跨多硬盘分割缓存](#)

## Nginx 静态文件服务

我们先来看看最简单的本地静态文件服务配置示例：

```
1. server {
2. listen 80;
3. server_name www.test.com;
4. charset utf-8;
5. root /data/www.test.com;
6. index index.html index.htm;
7. }
```

就这些？恩，就这些！如果只是提供简单的对外静态文件，它真的就可以用了。可是他不完美，远远没有发挥 Nginx 的半成功力，为什么这么说呢，看看下面的配置吧，为了大家看着方便，我们把每一项的作用都做了注释。

```
1. http {
2. # 这个将为打开文件指定缓存，默认是没有启用的，max 指定缓存数量，
3. # 建议和打开文件数一致，inactive 是指经过多长时间文件没被请求后删除缓存。
4. open_file_cache max=204800 inactive=20s;
5.
6. # open_file_cache 指令中的inactive 参数时间内文件的最少使用次数，
7. # 如果超过这个数字，文件描述符一直是在缓存中打开的，如上例，如果有一个
8. # 文件在inactive 时间内一次没被使用，它将被移除。
9. open_file_cache_min_uses 1;
10.
11. # 这个是指多长时间检查一次缓存的有效信息
12. open_file_cache_valid 30s;
13.
14. # 默认情况下，Nginx的gzip压缩是关闭的，gzip压缩功能就是可以让你节省不
15. # 少带宽，但是会增加服务器CPU的开销哦，Nginx默认只对text/html进行压缩，
16. # 如果要对html之外的内容进行压缩传输，我们需要手动来设置。
17. gzip on;
18. gzip_min_length 1k;
```

```
19. gzip_buffers 4 16k;
20. gzip_http_version 1.0;
21. gzip_comp_level 2;
22. gzip_types text/plain application/x-javascript text/css application/xml;
23.
24. server {
25. listen 80;
26. server_name www.test.com;
27. charset utf-8;
28. root /data/www.test.com;
29. index index.html index.htm;
30. }
31. }
```

我们都知道，应用程序和网站一样，其性能关乎生存。但如何使你的应用程序或者网站性能更好，并没有一个明确的答案。代码质量和架构是其中的一个原因，但是在很多例子中我们看到，你可以通过关注一些十分基础的应用内容分发技术（basic application delivery techniques），来提高终端用户的体验。其中一个例子就是实现和调整应用栈（application stack）的缓存。

## 文件缓存漫谈

一个 web 缓存坐落于客户端和原始服务器（origin server）中间，它保留了所有可见内容的拷贝。如果一个客户端请求的内容在缓存中存储，则可以直接在缓存中获得该内容而不需要与服务器通信。这样一来，由于 web 缓存距离客户端“更近”，就可以提高响应性能，并更有效率的使用应用服务器，因为服务器不用每次请求都进行页面生成工作。

在浏览器和应用服务器之间，存在多种潜在缓存，如：客户端浏览器缓存、中间缓存、内容分发网络（CDN）和服务器上的负载均衡和反向代理。缓存，仅在反向代理和负载均衡的层面，就对性能提高有很大的帮助。

举个例子说明，去年，我接手了一项任务，这项任务的内容是对一个加载缓慢的网站进行性能优化。首先引起我注意的事情是，这个网站差不多花费了超过 1 秒钟才生成了主页。经过一系列调试，我发现加载缓慢的原因在于页面被标记为不可缓存，即为了响应每一个请求，页面都是动态生成的。由于页面本身并不需要经常性的变更，并且不涉及个性化，那么这样做其实并没有必要。为了验证一下我的结论，我将页面标记为每 5 秒缓存一次，仅仅做了这一个调整，就能明显的感受到性能的提升。第一个字节到达的时间降低到几毫秒，同时页面的加载明显要更快。

并不是只有大规模的内容分发网络（CDN）可以在使用缓存中受益——缓存还可以提高负载均衡器、反向代理和应用服务器前端 web 服务的性能。通过上面的例子，我们看到，缓存内容结果，可以更高效的使用应用服务器，因为不需要每次都去做重复的页面生成工作。此外，web 缓存还可以用来提高网站可靠性。当服务器宕机或者繁忙时，比起返回错误信息给用户，不如通过配置 Nginx 将已经缓存下来的内容发送给用户。这意味着，网站在应用服务器或者数据库故障的情况下，可以保持部分甚至全部的功能运转。

下面讨论如何安装和配置 Nginx 的基础缓存（Basic Caching）。

## 如何安装和配置基础缓存

我们只需要两个命令就可以启用基础缓存：`proxy_cache_path` 和 `proxy_cache`。  
`proxy_cache_path` 用来设置缓存的路径和配置，`proxy_cache` 用来启用缓存。

```
1. proxy_cache_path/path/to/cache levels=1:2 keys_zone=my_cache:10m max_size=10g inactive=60m
2. use_temp_path=off;
3.
4. server {
5. ...
6. location / {
7. proxy_cache my_cache;
8. proxy_pass http://my_upstream;
9. }
10.
11. }
```

`proxy_cache_path` 命令中的参数及对应配置说明如下：

1. 用于缓存的本地磁盘目录是 `/path/to/cache/`
2. `levels` 在 `/path/to/cache/` 设置了一个两级层次结构的目录。将大量的文件放置在单个目录中会导致文件访问缓慢，所以针对大多数部署，我们推荐使用两级目录层次结构。如果 `levels` 参数没有配置，则 Nginx 会将所有的文件放到同一个目录中。
3. `keys_zone` 设置一个共享内存区，该内存区用于存储缓存键和元数据，有些类似计时器的用途。将键的拷贝放入内存可以使 Nginx 在不检索磁盘的情况下快速决定一个请求是 `HIT` 还是 `MISS`，这样大大提高了检索速度。一个 1MB 的内存空间可以存储大约 8000 个 key，那么上面配置的 10MB 内存空间可以存储差不多 80000 个 key。
4. `max_size` 设置了缓存的上限（在上面的例子中是 10G）。这是一个可选项；如果不指定具体值，那就是允许缓存不断增长，占用所有可用的磁盘空间。当缓存达到这个上限，处理器便调用 cache manager 来移除最近最少被使用的文件，这样把缓存的空间降低至这个限制之下。
5. `inactive` 指定了项目在不被访问的情况下能够在内存中保持的时间。在上面的例子中，如果一个文件在 60 分钟之内没有被请求，则缓存管理将会自动将其在内存中删除，不管该文件是否过期。该参数默认值为 10 分钟（10m）。注意，非活动内容有别于过期内容。Nginx 不会自动删除由缓存控制头部指定的过期内容（本例中 `Cache-Control:max-age=120`）。过期内容只有在 `inactive` 指定时间内没有被访问的情况下才会被删除。如果过期内容被访问了，那么 Nginx 就会将其从原服务器上刷新，并更新对应的 `inactive` 计时器。
6. Nginx 最初会将注定写入缓存的文件先放入一个临时存储区域，`use_temp_path=off` 命令指示 Nginx 将在缓存这些文件时将它们写入同一个目录下。我们强烈建议你参数设置为 `off` 来避免在文件系统中不必要的数据拷贝。`use_temp_path` 在 Nginx 1.7 版本和 Nginx Plus R6 中有所介绍。

最终，`proxy_cache` 命令启动缓存那些 URL 与 `location` 部分匹配的内容（本例中，为 `/`）。你同样可以将 `proxy_cache` 命令添加到 `server` 部分，这将会将缓存应用到所有的那些 `location` 中未指定自己的 `proxy_cache` 命令的服务中。

## 陈旧总比没有强

Nginx 内容缓存的一个非常强大的特性是：当无法从原始服务器获取最新的内容时，Nginx 可以分发缓存中的陈旧（stale，编者注：即过期内容）内容。这种情况一般发生在关联缓存内容的原始服务器宕机或者繁忙时。比起对客户端传达错误信息，Nginx 可发送在其内存中的陈旧的文件。Nginx 的这种代理方式，为服务器提供额外级别的容错能力，并确保了在服务器故障或流量峰值的情况下的正常运行。为了开启该功能，只需要添加 `proxy_cache_use_stale` 命令即可：

```
1. location / {
2. ...
3. proxy_cache_use_stale error timeout http_500 http_502 http_503 http_504;
4. }
```

按照上面例子中的配置，当 Nginx 收到服务器返回的 `error`，`timeout` 或者其他指定的 5xx 错误，并且在其缓存中有请求文件的陈旧版本，则会将这些陈旧版本的文件而不是错误信息发送给客户端。

## 缓存微调

Nginx 提供了丰富的可选项配置用于缓存性能的微调。下面是使用了几个配置的例子：

```
1. proxy_cache_path /path/to/cache levels=1:2 keys_zone=my_cache:10m max_size=10g inactive=60m
2. use_temp_path=off;
3. server {
4. ...
5. location / {
6. proxy_cache my_cache;
7. proxy_cache_revalidate on;
8. proxy_cache_min_uses 3;
9. proxy_cache_use_stale error timeout updating http_500 http_502 http_503 http_504;
10. proxy_cache_lock on;
11. proxy_pass http://my_upstream;
12. }
13. }
```

这些命令配置了下列的行为：

1. `proxy_cache_revalidate` 指示 Nginx 在刷新来自服务器的内容时使用 GET 请求。如果客户端的请求项已经被缓存过了，但是在缓存控制头部中定义为过期，那么 Nginx 就会在 GET 请求中包含 `If-Modified-Since` 字段，发送至服务器端。这项配置可以节约带宽，因为对于 Nginx 已经缓存过的文件，服务器只会在该文件请求头中 `Last-Modified` 记录的时间内被修改时才将全部文件一起发送。
2. `proxy_cache_min_uses` 该指令设置同一链接请求达到几次即被缓存，默认值为 1。当缓存

- 不断被填满时，这项设置便十分有用，因为这确保了只有那些被经常访问的内容会被缓存。
3. `proxy_cache_use_stale` 中的 `updating` 参数告知 Nginx 在客户端请求的项目的更新正在原服务器中下载时发送旧内容，而不是向服务器转发重复的请求。第一个请求陈旧文件的用户不得等待文件在原服务器中更新完毕。陈旧的文件会返回给随后的请求直到更新后的文件被全部下载。
  4. 当 `proxy_cache_lock` 被启用时，当多个客户端请求一个缓存中不存在的文件（或称之为一个 MISS），只有这些请求中的第一个被允许发送至服务器。其他请求在第一个请求得到满意结果之后在缓存中得到文件。如果不启用 `proxy_cache_lock`，则所有在缓存中找不到文件的请求都会直接与服务器通信。

## 跨多硬盘分割缓存

使用 Nginx 不需要建立一个 RAID（磁盘阵列）。如果有多个硬盘，Nginx 可以用来在多个硬盘之间分割缓存。下面是一个基于请求 URI 跨越两个硬盘之间均分缓存的例子：

```

1. proxy_cache_path /path/to/hdd1 levels=1:2 keys_zone=my_cache_hdd1:10m max_size=10g
2.
3. inactive=60m use_temp_path=off;
4. proxy_cache_path /path/to/hdd2 levels=1:2 keys_zone=my_cache_hdd2:10m max_size=10g inactive=60m
 use_temp_path=off;
5. split_clients $request_uri $my_cache {
6. 50% "my_cache_hdd1";
7. 50% "my_cache_hdd2";
8. }
9.
10. server {
11. ...
12. location / {
13. proxy_cache $my_cache;
14. proxy_pass http://my_upstream;
15. }
16. }
```

# 日志

- 日志
  - access\_log 访问日志
- error\_log 错误日志

# 日志

Nginx 日志主要有两种：access\_log(访问日志) 和 error\_log(错误日志)。

## access\_log 访问日志

access\_log 主要记录客户端访问 Nginx 的每一个请求，格式可以自定义。通过 access\_log 你可以得到用户地域来源、跳转来源、使用终端、某个 URL 访问量等相关信息。

log\_format 指令用于定义日志的格式，语法：`log_format name string;` 其中 name 表示格式名称，string 表示定义的格式字符串。log\_format 有一个默认的无需设置的组合日志格式。

默认的无需设置的组合日志格式

```
1. log_format combined '$remote_addr - $remote_user [$time_local] '
2. ' "$request" $status $body_bytes_sent '
3. ' "$http_referer" "$http_user_agent" ';
```

access\_log 指令用来指定访问日志文件的存放路径（包含日志文件名）、格式和缓存大小，语法：`access_log path [format_name [buffer=size | off]];` 其中 path 表示访问日志存放路径，format\_name 表示访问日志格式名称，buffer 表示缓存大小，off 表示关闭访问日志。

log\_format 使用示例：在 access.log 中记录客户端 IP 地址、请求状态和请求时间

```
1. log_format myformat '$remote_addr $status $time_local';
2. access_log logs/access.log myformat;
```

需要注意的是：log\_format 配置必须放在 http 内，否则会出现警告。Nginx 进程设置的用户和组必须对日志路径有创建文件的权限，否则，会报错。

定义日志使用的字段及其作用：

| 字段                                       | 作用        |
|------------------------------------------|-----------|
| \$remote_addr与<br>\$http_x_forwarded_for | 记录客户端IP地址 |
| \$remote_user                            | 记录客户端用户名称 |

|                       |                                              |
|-----------------------|----------------------------------------------|
| \$request             | 记录请求的URI和HTTP协议                              |
| \$status              | 记录请求状态                                       |
| \$body_bytes_sent     | 发送给客户端的字节数，不包括响应头的大小                         |
| \$bytes_sent          | 发送给客户端的总字节数                                  |
| \$connection          | 连接的序列号                                       |
| \$connection_requests | 当前通过一个连接获得的请求数量                              |
| \$msec                | 日志写入时间。单位为秒，精度是毫秒                            |
| \$pipe                | 如果请求是通过HTTP流水线(pipelined)发送，pipe值为“p”，否则为“.” |
| \$http_referer        | 记录从哪个页面链接访问过来的                               |
| \$http_user_agent     | 记录客户端浏览器相关信息                                 |
| \$request_length      | 请求的长度（包括请求行，请求头和请求正文）                        |
| \$request_time        | 请求处理时间，单位为秒，精度毫秒                             |
| \$time_iso8601        | ISO8601标准格式下的本地时间                            |
| \$time_local          | 记录访问时间与时区                                    |

## error\_log 错误日志

error\_log 主要记录客户端访问 Nginx 出错时的日志，格式不支持自定义。通过查看错误日志，你可以得到系统某个服务或 server 的性能瓶颈等。因此，将日志利用好，你可以得到很多有价值的信息。

error\_log 指令用来指定错误日志，语法：`error_log path [level]`；其中 path 表示错误日志存放路径，level 表示错误日志等级，日志等级包括 debug、info、notice、warn、error、crit、alert、emerg，从左至右，日志详细程度逐级递减，即 debug 最详细，emerg 最少，默认为 error。

注意：`error_log off` 并不能关闭错误日志记录，此时日志信息会被写入到文件名为 off 的文件当中。如果要关闭错误日志记录，可以使用如下配置：

Linux 系统把存储位置设置为空设备

```
1.
2. error_log /dev/null;
3.
4. http {
5. # ...
6. }
```

Windows 系统把存储位置设置为空设备

```
1.
2. error_log nul;
3.
4. http {
5. # ...
6. }
```

另外 Linux 系统可以使用 `tail` 命令方便的查阅正在改变的文件, `tail -f filename` 会把 `filename` 里最尾部的内容显示在屏幕上, 并且不断刷新, 使你看到最新的文件内容。Windows 系统没有这个命令, 你可以在网上找到动态查看文件的工具。



# 反向代理

- 反向代理
  - 什么是反向代理
  - 反向代理典型应用场景
  - 正向代理

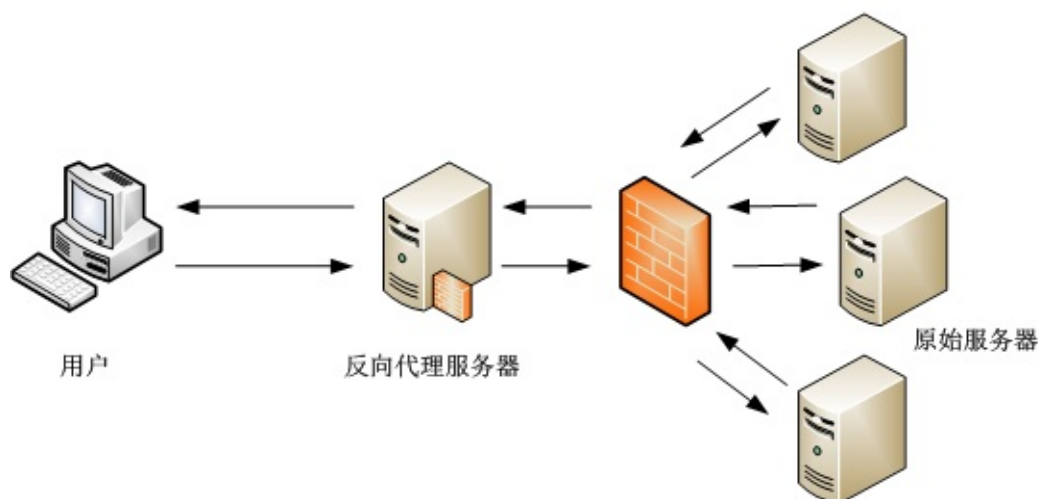
## 反向代理

### 什么是反向代理

反向代理 (Reverse Proxy) 方式是指用代理服务器来接受 internet 上的连接请求, 然后将请求转发给内部网络上的服务器, 并将从服务器上得到的结果返回给 internet 上请求连接的客户端, 此时代理服务器对外就表现为一个反向代理服务器。

举个例子, 一个用户访问 <http://www.example.com/readme>, 但是 [www.example.com](http://www.example.com) 上并不存在 readme 页面, 它是偷偷从另外一台服务器上取回来, 然后作为自己的内容返回给用户。但是用户并不知情这个过程。对用户来说, 就像是直接从 [www.example.com](http://www.example.com) 获取 readme 页面一样。这里所提到的 [www.example.com](http://www.example.com) 这个域名对应的服务器就设置了反向代理功能。

反向代理服务器, 对于客户端而言它就像是原始服务器, 并且客户端不需要进行任何特别的设置。客户端向反向代理的命名空间(name-space)中的内容发送普通请求, 接着反向代理将判断向何处(原始服务器)转交请求, 并将获得的内容返回给客户端, 就像这些内容原本就是它自己的一样。如下图所示:



### 反向代理典型应用场景

反向代理的典型用途是将防火墙后面的服务器提供给 Internet 用户访问, 加强安全防护。反向代

理还可以为后端的多台服务器提供负载均衡，或为后端较慢的服务器提供 缓冲 服务。另外，反向代理还可以启用高级 URL 策略和管理技术，从而使处于不同 web 服务器系统的 web 页面同时存在于同一个 URL 空间下。

Nginx 的其中一个用途是做 HTTP 反向代理，下面简单介绍 Nginx 作为反向代理服务器的方法。

场景描述：访问本地服务器上的 README.md 文件 <http://localhost/README.md>，本地服务器进行反向代理，从 <https://github.com/moonbingbing/openresty-best-practices/blob/master/README.md> 获取页面内容。

nginx.conf 配置示例：

```

1. worker_processes 1;
2.
3. pid logs/nginx.pid;
4. error_log logs/error.log warn;
5.
6. events {
7. worker_connections 3000;
8. }
9.
10. http {
11. include mime.types;
12. server_tokens off;
13.
14. ## 下面配置反向代理的参数
15. server {
16. listen 8866;
17.
18. ## 1. 用户访问 http://ip:port, 则反向代理到 https://github.com
19. location / {
20. proxy_pass https://github.com;
21. proxy_redirect off;
22. proxy_set_header Host $host;
23. proxy_set_header X-Real-IP $remote_addr;
24. proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
25. }
26.
27. ## 2. 用户访问 http://ip:port/README.md, 则反向代理到
28. ## https://github.com/.../README.md
29. location /README.md {
30. proxy_set_header X-Real-IP $remote_addr;
31. proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
32. proxy_pass https://github.com/moonbingbing/openresty-best-
33. practices/blob/master/README.md;
34. }
35. }

```

成功启动 Nginx 后，我们打开浏览器，验证下反向代理的效果。在浏览器地址栏中输入 `localhost/README.md`，返回的结果是我们 GitHub 源代码的 README 页面。如下图：



我们只需要配置一下 `nginx.conf` 文件，不用写任何 web 页面，就可以偷偷地从别的服务器上读取一个页面返回给用户。

下面我们来看一下 `nginx.conf` 里用到的配置项：

### (1) location

location 项对请求 URI 进行匹配，location 后面配置了匹配规则。例如上面的例子中，如果请求的 URI 是 `localhost/`，则会匹配 `location /` 这一项；如果请求的 URI 是 `localhost/README.md`，则会匹配 `location /README.md` 这项。

上面这个例子只是针对一个确定的 URI 做了反向代理，有的读者会有疑惑：如果对每个页面都进行这样的配置，那将会大量重复，能否做 批量 配置呢？此时需要配合使用 location 的正则匹配功能。具体实现方法可参考 Nginx 文档中 [关于 location 的描述](#)。

### (2) proxy\_pass

`proxy_pass` 后面跟着一个 URL，用来将请求反向代理到 URL 参数指定的服务器上。例如我们上面例子中的 `proxy_pass https://github.com`，则将匹配的请求反向代理到 `https://github.com`。

### (3) `proxy_set_header`

默认情况下，反向代理不会转发原始请求中的 `Host` 头部，如果需要转发，就需要加上这句：`proxy_set_header Host $host;`

除了上面提到的常用配置项，还有 `proxy_redirect`、`proxy_set_body`、`proxy_limit_rate` 等参数，具体用法可以到[Nginx 官网](#)查看。

## 正向代理

既然有反向代理，自然也有正向代理。简单来说，正向代理就像一个跳板，例如一个用户访问不了某网站（例如 `www.google.com`），但是他能访问一个代理服务器，这个代理服务器能访问 `www.google.com`，于是用户可以先连上代理服务器，告诉它需要访问的内容，代理服务器去取回来返回给用户。例如一些常见的翻墙工具、游戏代理就是利用正向代理的原理工作的，我们需要在这些正向代理工具上配置服务器的 IP 地址等信息。

# 负载均衡

- [负载均衡](#)
  - [upstream 负载均衡概要](#)
    - [upstream 支持的负载均衡算法](#)
    - [upstream 支持的状态参数](#)
    - [配置 Nginx 负载均衡](#)
    - [配置 Nginx 进行健康状态检查](#)
    - [配置 backup 服务器](#)
    - [配置 ip\\_hash 负载均衡](#)

## 负载均衡

负载均衡 (Load balancing) 是一种计算机网络技术, 用来在多个计算机 (计算机集群)、网络连接、CPU、磁盘驱动器或其他资源中分配负载, 以达到最佳化资源使用、最大化吞吐率、最小化响应时间、同时避免过载的目的。

使用带有负载均衡的多个服务器组件, 取代单一的组件, 可以通过冗余提高可靠性。负载均衡服务通常是由专用软体和硬件来完成。

负载均衡最重要的一个应用是利用多台服务器提供单一服务, 这种方案有时也称之为服务器农场。通常, 负载均衡主要应用于 Web 网站, 大型的 Internet Relay Chat 网络, 高流量的文件下载网站, NNTP (Network News Transfer Protocol) 服务和 DNS 服务。现在负载均衡器也开始支持数据库服务, 称之为数据库负载均衡器。

对于互联网服务, 负载均衡器通常是一个软体程序, 这个程序侦听一个外部端口, 互联网用户可以通过这个端口来访问服务, 而作为负载均衡器的软体会将用户的请求转发给后台内网服务器, 内网服务器将请求的响应返回给负载均衡器, 负载均衡器再将响应发送到用户, 这样就向互联网用户隐藏了内网结构, 阻止了用户直接访问后台 (内网) 服务器, 使得服务器更加安全, 可以阻止对核心网络栈和运行在其它端口服务的攻击。

当所有后台服务器出现故障时, 有些负载均衡器会提供一些特殊的功能来处理这种情况。例如转发请求到一个备用的负载均衡器、显示一条关于服务中断的消息等。负载均衡器使得 IT 团队可以显著提高容错能力。它可以自动提供大量的容量以处理任何应用程序流量的增加或减少。

负载均衡在互联网世界中的作用如此重要, 本章我们一起了解一下 Nginx 是如何帮我们完成 HTTP 协议负载均衡的。

## upstream 负载均衡概要

配置示例, 如下:

```

1. upstream test.net{
2. ip_hash;
3. server 192.168.10.13:80;
4. server 192.168.10.14:80 down;
5. server 192.168.10.15:8009 max_fails=3 fail_timeout=20s;
6. server 192.168.10.16:8080;
7. }
8. server {
9. location / {
10. proxy_pass http://test.net;
11. }
12. }

```

upstream 是 Nginx 的 HTTP Upstream 模块，这个模块通过一个简单的调度算法来实现客户端 IP 到后端服务器的负载均衡。在上面的设定中，通过 upstream 指令指定了一个负载均衡器的名称 test.net。这个名称可以任意指定，在后面需要用到的地方直接调用即可。

### upstream 支持的负载均衡算法

Nginx 的负载均衡模块目前支持 6 种调度算法，下面进行分别介绍，其中后两项属于第三方调度算法。

- 轮询（默认）：每个请求按时间顺序逐一分配到不同的后端服务器，如果后端某台服务器宕机，故障系统被自动剔除，使用户访问不受影响。Weight 指定轮询权值，Weight 值越大，分配到的访问机率越高，主要用于后端每个服务器性能不均的情况下。
- ip\_hash：每个请求按访问 IP 的 hash 结果分配，这样来自同一个 IP 的访客固定访问一个后端服务器，有效解决了动态网页存在的 session 共享问题。
- fair：这是比上面两个更加智能的负载均衡算法。此种算法可以依据页面大小和加载时间长短智能地进行负载均衡，也就是根据后端服务器的响应时间来分配请求，响应时间短的优先分配。Nginx 本身是不支持 fair 的，如果需要使用这种调度算法，必须下载 Nginx 的 upstream\_fair 模块。
- url\_hash：此方法按访问 url 的 hash 结果来分配请求，使每个 url 定向到同一个后端服务器，可以进一步提高后端缓存服务器的效率。Nginx 本身是不支持 url\_hash 的，如果需要使用这种调度算法，必须安装 Nginx 的 hash 软件包。
- least\_conn：最少连接负载均衡算法，简单来说就是每次选择的后端都是当前最少连接的一个 server(这个最少连接不是共享的，是每个 worker 都有自己的一个数组进行记录后端 server 的连接数)。
- hash：这个 hash 模块又支持两种模式 hash，一种是普通的 hash，另一种是一致性 hash(consistent)。

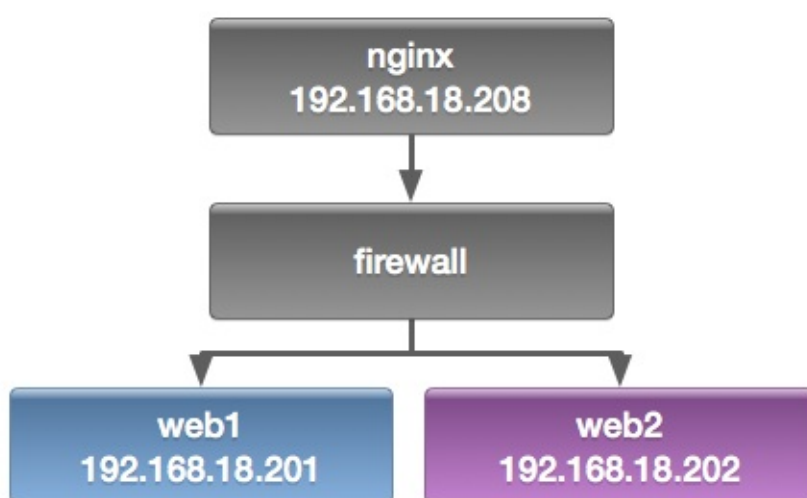
### upstream 支持的状态参数

在 HTTP Upstream 模块中，可以通过 server 指令指定后端服务器的 IP 地址和端口，同时还可以设定每个后端服务器在负载均衡调度中的状态。常用的状态有：

- down: 表示当前的 server 暂时不参与负载均衡。
- backup: 预留的备份机器。当其他所有的非 backup 机器出现故障或者忙的时候，才会请求 backup 机器，因此这台机器的压力最轻。
- max\_fails: 允许请求失败的次数，默认为 1。当超过最大次数时，返回 proxy\_next\_upstream 模块定义的错误。
- fail\_timeout: 在经历了 max\_fails 次失败后，暂停服务的时间。max\_fails 可以和 fail\_timeout 一起使用。

当负载调度算法为 ip\_hash 时，后端服务器在负载均衡调度中的状态不能是 backup。

配置 Nginx 负载均衡



Nginx 配置负载均衡

```

1. upstream webservers {
2. server 192.168.18.201 weight=1;
3. server 192.168.18.202 weight=1;
4. }
5. server {
6. listen 80;
7. server_name localhost;
8. #charset koi8-r;
9. #access_log logs/host.access.log main;
10. location / {
11. proxy_pass http://webservers;
12. proxy_set_header X-Real-IP $remote_addr;
13. }
14. }

```

注，upstream 是定义在 `server{ }` 之外的，不能定义在 `server{ }` 内部。定义好



upstream 之后，用 proxy\_pass 引用一下即可。

重新加载一下配置文件

```
1. # service nginx reload
2. nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
3. nginx: configuration file /etc/nginx/nginx.conf test is successful
```

```
1. # curl http://192.168.18.208
2. web1.test.com
3. # curl http://192.168.18.208
4. web2.test.com
5. # curl http://192.168.18.208
6. web1.test.com
7. # curl http://192.168.18.208
8. web2.test.com
```

注，大家可以不断的刷新浏览的内容，可以发现 web1 与 web2 是交替出现的，达到了负载均衡的效果。

查看一下Web访问服务器日志

Web1:

```
1. # tail /var/log/nginx/access_log
2. 192.168.18.138 - - [04/Sep/2013:09:41:58 +0800] "GET / HTTP/1.0" 200 23 "-"
3. 192.168.18.138 - - [04/Sep/2013:09:41:58 +0800] "GET / HTTP/1.0" 200 23 "-"
4. 192.168.18.138 - - [04/Sep/2013:09:41:59 +0800] "GET / HTTP/1.0" 200 23 "-"
5. 192.168.18.138 - - [04/Sep/2013:09:41:59 +0800] "GET / HTTP/1.0" 200 23 "-"
6. 192.168.18.138 - - [04/Sep/2013:09:42:00 +0800] "GET / HTTP/1.0" 200 23 "-"
7. 192.168.18.138 - - [04/Sep/2013:09:42:00 +0800] "GET / HTTP/1.0" 200 23 "-"
8. 192.168.18.138 - - [04/Sep/2013:09:42:00 +0800] "GET / HTTP/1.0" 200 23 "-"
9. 192.168.18.138 - - [04/Sep/2013:09:44:21 +0800] "GET / HTTP/1.0" 200 23 "-"
10. 192.168.18.138 - - [04/Sep/2013:09:44:22 +0800] "GET / HTTP/1.0" 200 23 "-"
11. 192.168.18.138 - - [04/Sep/2013:09:44:22 +0800] "GET / HTTP/1.0" 200 23 "-"
```

Web2:

先修改一下，web 服务器记录日志的格式。

```
1. # LogFormat "%[X-Real-IP]i %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" combined
2. # tail /var/log/nginx/access_log
3. 192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
4. 192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
5. 192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
6. 192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
```



```

7. 192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
8. 192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
9. 192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
10. 192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
11. 192.168.18.138 - - [04/Sep/2013:09:50:29 +0800] "GET / HTTP/1.0" 200 23 "-"
12. 192.168.18.138 - - [04/Sep/2013:09:50:29 +0800] "GET / HTTP/1.0" 200 23 "-"

```

注，大家可以看到，两台服务器日志都记录是 192.168.18.138 访问的日志，也说明了负载均衡配置成功。

### 配置 Nginx 进行健康状态检查

利用 max\_fails、fail\_timeout 参数，控制异常情况，示例配置如下：

```

1. upstream webservers {
2. server 192.168.18.201 weight=1 max_fails=2 fail_timeout=2;
3. server 192.168.18.202 weight=1 max_fails=2 fail_timeout=2;
4. }

```

重新加载一下配置文件：

```

1. # service nginx reload
2. nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
3. nginx: configuration file /etc/nginx/nginx.conf test is successful
4. 重新载入 nginx : [确定]

```

先停止 Web1，进行测试：

```

1. # service nginx stop
2. 停止 nginx : [确定]

```

```

1. # curl http://192.168.18.208
2. web2.test.com
3. # curl http://192.168.18.208
4. web2.test.com
5. # curl http://192.168.18.208
6. web2.test.com

```

注，大家可以看到，现在只能访问 Web2，再重新启动 Web1，再次访问一下。

```

1. # service nginx start
2. 正在启动 nginx : [确定]

```

```

1. # curl http://192.168.18.208
2. web1.test.com
3. # curl http://192.168.18.208
4. web2.test.com
5. # curl http://192.168.18.208
6. web1.test.com
7. # curl http://192.168.18.208
8. web2.test.com

```

PS: 大家可以看到, 现在又可以重新访问, 说明 Nginx 的健康状态查检配置成功。但大家想一下, 如果不幸的是所有服务器都不能提供服务了怎么办, 用户打开页面就会出现出错页面, 那么会带来用户体验的降低, 所以我们能不能像配置 LVS 是配置 `sorry_server` 呢, 答案是可以的, 但这里不是配置 `sorry_server` 而是配置 `backup`。

### 配置 backup 服务器

备份服务器配置:

```

1. server {
2. listen 8080;
3. server_name localhost;
4. root /data/www/errorpage;
5. index index.html;
6. }

```

`index.html` 文件内容:

```

1. # cat index.html
2. <h1>Sorry.....</h1>

```

负载均衡配置:

```

1. upstream webservers {
2. server 192.168.18.201 weight=1 max_fails=2 fail_timeout=2;
3. server 192.168.18.202 weight=1 max_fails=2 fail_timeout=2;
4. server 127.0.0.1:8080 backup;
5. }

```

重新加载配置文件:

```

1. # service nginx reload
2. nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
3. nginx: configuration file /etc/nginx/nginx.conf test is successful
4. 重新载入 nginx : [确定]

```

关闭 Web 服务器并进行测试：

```
1. # service nginx stop
2. 停止 nginx : [确定]
```

进行测试：

```
1. # curl http://192.168.18.208
2. <h1>Sorry.....</h1>
3. # curl http://192.168.18.208
4. <h1>Sorry.....</h1>
5. # curl http://192.168.18.208
6. <h1>Sorry.....</h1>
```

注，大家可以看到，当所有服务器都不能工作时，就会启动备份服务器。好了，backup 服务器就配置到这里，下面我们来配置 ip\_hash 负载均衡。

配置 ip\_hash 负载均衡

ip\_hash: 每个请求按访问 IP 的 hash 结果分配，这样来自同一个 IP 的访客固定访问一个后端服务器，有效解决了动态网页存在的 session 共享问题，电子商务网站用的比较多。

```
1. # vim /etc/nginx/nginx.conf
2. upstream webservers {
3. ip_hash;
4. server 192.168.18.201 weight=1 max_fails=2 fail_timeout=2;
5. server 192.168.18.202 weight=1 max_fails=2 fail_timeout=2;
6. #server 127.0.0.1:8080 backup;
7. }
```

注，当负载调度算法为 ip\_hash 时，后端服务器在负载均衡调度中的状态不能有 backup。有人可能会问，为什么呢？大家想啊，如果负载均衡把你分配到 backup 服务器上，你能访问到页面吗？不能，所以不能配置 backup 服务器。

重新加载一下服务器：

```
1. # service nginx reload
2. nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
3. nginx: configuration file /etc/nginx/nginx.conf test is successful
4. 重新载入 nginx : [确定]
```

测试一下：

```
1. # curl http://192.168.18.208
```

```
2. web2.test.com
3. # curl http://192.168.18.208
4. web2.test.com
5. # curl http://192.168.18.208
6. web2.test.com
```

注，大家可以看到，你不断的刷新页面一直会显示 web2，说明 ip\_hash 负载均衡配置成功。

## 陷阱和常见错误

- [Nginx 陷阱和常见错误](#)
  - [警告：](#)
  - [本指南说](#)
  - [我的问题没有被列出来](#)
  - [chmod 777](#)
  - [把 root 放在 location 区域内](#)
  - [重复的 index 指令](#)
  - [使用 if](#)
    - [用 if 判断 Server Name](#)
    - [用 if 检查文件是否存在](#)
  - [前端控制器模式的 web 应用](#)
  - [把不可控制的请求发给 PHP](#)
  - [脚本文件名里面的 FastCGI 路径](#)
  - [费力的 rewrites](#)
  - [忽略 http:// 的 rewrite](#)
  - [代理所有东西](#)
  - [配置的修改没有起效](#)
  - [VirtualBox](#)
  - [丢失（消失）的 HTTP 头](#)
  - [没有使用标准的 Document Root Location](#)
  - [使用默认的 Document Root](#)
  - [使用主机名来解析地址](#)
  - [在 HTTPS 中使用 SSLv3](#)

## Nginx 陷阱和常见错误

翻译

自：[https://www.nginx.com/resources/wiki/start/topics/tutorials/config\\_pitfalls/](https://www.nginx.com/resources/wiki/start/topics/tutorials/config_pitfalls/)

**警告：**

请阅读下面所有的内容！是所有的！

不管是新手还是老用户，都可能会掉到一个陷阱中去。下面我们会列出一些我们经常看到，和经常需要解释如何解决的问题。在 Freenode 的 Nginx IRC 频道中，我们频繁的看到这些问题出现。

## 本指南说

最经常看到的是，有人从一些其他的指南中，尝试拷贝、粘贴一个配置片段。并不是说其他所有的指南都是错的，但是里面错误的比例很可怕。即使是在 Linode 库中也有质量较差的信息，一些 Nginx 社区成员曾经徒劳的试图去纠正。

本指南的文档，是社区成员所创建和审查，他们直接和所有类型的 Nginx 用户在一起工作。这个特定的文档之所以存在，是因为社区成员看到有大量普遍和重复出现的问题。

## 我的问题没有被列出来

在这里你没有看到和你具体问题相关的东西。也许我们并没有解决你经历的具体问题。不要只是大概浏览下这个网页，也不要假设你是无意才找到这里的。你找到这里，是因为这里列出了你做错的一些东西。

在许多问题上，当涉及到支持很多用户，社区成员不希望去支持破碎的配置。所以在提问求助前，先修复你的配置。

通读这个文档来修复你的配置，不要只是走马观花。

## chmod 777

永远不要 使用 777，这可能是一个漂亮的数字，有时候可以懒惰的解决权限问题，但是它同样也表示你没有线索去解决权限问题，你只是在碰运气。你应该检查整个路径的权限，并思考发生了什么事情。

要轻松的显示一个路径的所有权限，你可以使用：

```
1. namei -om /path/to/check
```

## 把 root 放在 location 区块内

糟糕的配置：

```
1. server {
2. server_name www.example.com;
3. location / {
4. root /var/www/Nginx -default/;
5. # [...]
6. }
7. location /foo {
8. root /var/www/Nginx -default/;
9. # [...]
10. }
11. location /bar {
12. root /var/www/Nginx -default/;
```

```

13. # [...]
14. }
15. }

```

这个是能工作的。把 `root` 放在 `location` 区块里面会工作，但并不是完全有效的。错就错在只要你开始增加其他的 `location` 区块，就需要给每一个 `location` 区块增加一个 `root`。如果没有添加，就会没有 `root`。让我们看下正确的配置。

推荐的配置：

```

1. server {
2. server_name www.example.com;
3. root /var/www/Nginx -default/;
4. location / {
5. # [...]
6. }
7. location /foo {
8. # [...]
9. }
10. location /bar {
11. # [...]
12. }
13. }

```

## 重复的 `index` 指令

糟糕的配置：

```

1. http {
2. index index.php index.htm index.html;
3. server {
4. server_name www.example.com;
5. location / {
6. index index.php index.htm index.html;
7. # [...]
8. }
9. }
10. server {
11. server_name example.com;
12. location / {
13. index index.php index.htm index.html;
14. # [...]
15. }
16. location /foo {

```

```

17. index index.php;
18. # [...]
19. }
20. }
21. }

```

为什么重复了这么多行不需要的配置呢？简单的使用“`index`”指令一次就够了。只需要把它放到 `http {}` 区块里面，下面的就会继承这个配置。

推荐的配置：

```

1. http {
2. index index.php index.htm index.html;
3. server {
4. server_name www.example.com;
5. location / {
6. # [...]
7. }
8. }
9. server {
10. server_name example.com;
11. location / {
12. # [...]
13. }
14. location /foo {
15. # [...]
16. }
17. }
18. }

```

## 使用 `if`

这里篇幅有限，只介绍一部分使用 `if` 指令的陷阱。更多陷阱你应该点击查看邪恶的 `if` 指令。我们看下 `if` 指令的几个邪恶的用法。

注意看这里：

邪恶的 `if` 指令

## 用 `if` 判断 Server Name

糟糕的配置：

```

1. server {
2. server_name example.com *.example.com;
3. if ($host ~* ^www\.(.+)) {

```



```

4. set $raw_domain $1;
5. rewrite ^/(.*)$ $raw_domain/$1 permanent;
6. }
7. # [...]
8. }
9. }

```

这个配置有三个问题。首先是 `if` 的使用，为啥它这么糟糕呢？你有阅读邪恶的 `if` 指令吗？当 Nginx 收到无论来自哪个子域名的何种请求，不管域名是 `www.example.com` 还是 `example.com`，这个 `if` 指令 总是 会被执行。因此 Nginx 会检查 每个请求 的 Host header，这是十分低效的。你应该避免这种情况，而是使用下面配置里面的两个 `server` 指令。

推荐的配置：

```

1. server {
2. server_name www.example.com;
3. return 301 $scheme://example.com$request_uri;
4. }
5. server {
6. server_name example.com;
7. # [...]
8. }

```

除了增强了配置的可读性，这种方法还降低了 Nginx 的处理要求；我们摆脱了不必要的 `if` 指令；我们用了 `$scheme` 来表示 URI 中是 `http` 还是 `https` 协议，避免了硬编码。

## 用 `if` 检查文件是否存在

使用 `if` 指令来判断文件是否存在是很可怕的，如果你在使用新版本的 Nginx，你应该看看 `try_files`，这会让你的生活变得更轻松。

糟糕的配置：

```

1. server {
2. root /var/www/example.com;
3. location / {
4. if (!-f $request_filename) {
5. break;
6. }
7. }
8. }

```

推荐的配置：

```

1. server {
2. root /var/www/example.com;
3. location / {
4. try_files $uri $uri/ /index.html;
5. }
6. }

```

我们不再尝试使用 `if` 来判断 `$uri` 是否存在，用 `try_files` 意味着你可以测试一个序列。如果 `$uri` 不存在，就会尝试 `$uri/`，还不存在的话，在尝试一个回调 `location`。

在上面配置的例子中，如果 `$uri` 这个文件存在，就正常服务；如果不存在就检测 `$uri/` 这个目录是否存在；如果不存在就按照 `index.html` 来处理，你需要保证 `index.html` 是存在的。  
`try_files` 的加载是如此简单。这是另外一个你可以完全消除 `if` 指令的实例。

## 前端控制器模式的 web 应用

“前端控制器模式”是流行的设计，被用在很多非常流行的 PHP 软件包里面。里面的很多示例配置都过于复杂。想要 Drupal, Joomla 等运行起来，只用这样做就可以了：

```

1. try_files $uri $uri/ /index.php?q=$uri&$args;

```

注意：你实际使用的软件包，在参数名字上会有差异。比如：

- “q” 参数用在 Drupal, Joomla, WordPress
- “page” 用在 CMS Made Simple

一些软件甚至不需要查询字符串，它们可以从 `REQUEST_URI` 中读取。比如 WordPress 就支持这样的配置：

```

1. try_files $uri $uri/ /index.php;

```

当然在你的开发中可能会有变化，你可能需要基于你的需要设置更复杂的配置。但是对于一个基础的网站来说，这个配置可以工作得很完美。你应该永远从简单开始来搭建你的系统。

如果你不关心目录是否存在这个检测的话，你也可以决定忽略这个目录的检测，去掉 “`$uri/`” 这个配置。

## 把不可控制的请求发给 PHP

很多网络上面推荐的和 PHP 相关的 Nginx 配置，都是把每一个 `.php` 结尾的 URI 传递给 PHP 解释器。

请注意，大部分这样的 PHP 设置都有严重的安全问题，因为它可能允许执行任意第三方代码。

有问题的配置通常如下：

```
1. location ~* \.php$ {
2. fastcgi_pass backend;
3. # [...]
4. }
```

在这里，每一个 .php 结尾的请求，都会传递给 FastCGI 的后台处理程序。这样做的问题是，当完整的路径未能指向文件系统里面一个确切的文件时，默认的 PHP 配置试图是猜测你想执行的是哪个文件。

举个例子，如果一个请求中的 `/forum/avatar/1232.jpg/file.php` 文件不存在，但是 `/forum/avatar/1232.jpg` 存在，那么 PHP 解释器就会取而代之，使用 `/forum/avatar/1232.jpg` 来解释。如果这里面嵌入了 PHP 代码，这段代码就会被执行起来。

有几个避免这种情况的选择：

- 在 `php.ini` 中设置 `cgi.fix_pathinfo=0`。这会令 PHP 解释器只尝试给定的文件路径，如果没有找到这个文件就停止处理。
- 确保 Nginx 只传递指定的 PHP 文件去执行

```
1. location ~* (file_a|file_b|file_c)\.php$ {
2. fastcgi_pass backend;
3. # [...]
4. }
```

- 对于任何用户可以上传的目录，特别的关闭 PHP 文件的执行权限

```
1. location /uploaddir {
2. location ~ \.php$ {return 403;}
3. # [...]
4. }
```

- 使用 `try_files` 指令过滤出文件不存在的情况

```
1. location ~* \.php$ {
2. try_files $uri =404;
3. fastcgi_pass backend;
4. # [...]
5. }
```

- 使用嵌套的 `location` 过滤出文件不存在的情况

```
1. location ~* \.php$ {
2. location ~ \..*/.*\.php$ {return 404;}
3. fastcgi_pass backend;
4. # [...]
5. }
```

## 脚本文件名里面的 FastCGI 路径

很多外部指南喜欢依赖绝对路径来获取你的信息。这在 PHP 的配置块里面很常见。

当你从仓库安装 Nginx，通常都是以在配置里面折腾好“`include fastcgi_params;`”来收尾。这个配置文件位于你的 Nginx 根目录下，通常在 `/etc/nginx/` 里面。

推荐的配置：

```
1. fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
```

糟糕的配置：

```
1. fastcgi_param SCRIPT_FILENAME /var/www/your-site.com/$fastcgi_script_name;
```

`$document_root$` 在哪里设置呢？它是 `server` 块里面的 `root` 指令来设置的。

你的 `root` 指令不在 `server` 块内？请看前面关于 `root` 指令的陷阱。

## 费力的 rewrites

不要知难而退，`rewrite` 很容易和正则表达式混为一谈。

实际上，`rewrite` 是很容易的，我们应该努力去保持它们的整洁。

很简单，不添加冗余代码就行了。

糟糕的配置：

```
1. rewrite ^/(.*)$ http://example.com/$1 permanent;
```

好点儿的配置：

```
1. rewrite ^ http://example.com$request_uri? permanent;
```

更好的配置：

```
1. return 301 http://example.com$request_uri;
```

反复对比下这几个配置。

第一个 `rewrite` 捕获不包含第一个斜杠的完整 URI。

使用内置的变量 `$request_uri`，我们可以有效的完全避免任何捕获和匹配。

## 忽略 `http://` 的 `rewrite`

这个非常简单，`rewrites` 是用相对路径的，除非你告诉 Nginx 不是相对路径。

生成绝对路径的 `rewrite` 也很简单，加上 `scheme` 就行了。

糟糕的配置：

```
1. rewrite ^ example.com permanent;
```

推荐的配置：

```
1. rewrite ^ http://example.com permanent;
```

你可以看到我们做的只是在 `rewrite` 里面增加了 `http://`。这个很简单而且有效。

## 代理所有东西

糟糕的配置：

```
1. server {
2. server_name _;
3. root /var/www/site;
4. location / {
5. include fastcgi_params;
6. fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
7. fastcgi_pass unix:/tmp/phpcggi.socket;
8. }
9. }
```

这个是令人讨厌的配置，你把 所有东西 都丢给了 PHP。

为什么呢？Apache 可能要这样做，但在 Nginx 里你不必这样。

换个思路，`try_files` 有一个神奇之处，它是按照特定顺序去尝试文件的。

这意味着 Nginx 可以先尝试下静态文件，如果没有才继续往后走。

这样 PHP 就不用参与到这个处理中，会快很多。

特别是如果你提供一个 1MB 图片数千次请求的服务，通过 PHP 处理还是直接返回静态文件呢？

让我们看下怎么做到吧。

推荐的配置：

```

1. server {
2. server_name _;
3. root /var/www/site;
4. location / {
5. try_files $uri $uri/ @proxy;
6. }
7. location @proxy {
8. include fastcgi_params;
9. fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
10. fastcgi_pass unix:/tmp/phpcgsocket;
11. }
12. }
```

另外一个推荐的配置：

```

1. server {
2. server_name _;
3. root /var/www/site;
4. location / {
5. try_files $uri $uri/ /index.php;
6. }
7. location ~ \.php$ {
8. include fastcgi_params;
9. fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
10. fastcgi_pass unix:/tmp/phpcgsocket;
11. }
12. }
```

这个很容易，不是吗？你看，如果请求的 URI 存在，Nginx 会处理掉；  
 如果不存在，检查下目录是不是存在，是的话也可以被 Nginx 处理；  
 只有在 Nginx 不能直接处理请求的 URI 的时候，才会进入 proxy 这个 location 来处理。

现在，考虑下你的请求中有多少静态内容，比如图片、css、javascript 等。这可能会帮你节省很多开销。

## 配置的修改没有起效

浏览器缓存。你的配置可能是对的，但怎么尝试结果总是不对，百思不得其解。  
 罪魁祸首是你的浏览器缓存。当你下载东西的时候，浏览器做了缓存。

怎么修复：

- 在 Firefox 里面 Ctrl + Shift + Delete，检查缓存，点击立即清理。可以用你喜欢的搜

索引找到其他浏览器清理缓存的方法。

每次更改配置后，都需要清理下缓存（除非你知道这个不必要），这会省很多事儿。

- 使用 curl。

## VirtualBox

如果你在 VirtualBox 的虚拟机中运行 Nginx，而它不工作，可能是因为 `sendfile()` 引起的麻烦。

只用简单的注释掉 `sendfile` 指令，或者设置为 `off`。该指令大都会写在 Nginx `.conf` 文件中：

```
1. sendfile off;
```

## 丢失（消失）的 HTTP 头

如果你没有明确的设置 `underscores_in_headers on;`，

Nginx 将会自动丢弃带有下划线的 HTTP 头（根据 HTTP 标准，这样做是完全正当的）。

这样做是为了防止头信息映射到 CGI 变量时产生歧义，因为破折号和下划线都会被映射为下划线。

## 没有使用标准的 Document Root Location

在所有的文件系统中，一些目录永远也不应该被用做数据的托管。这些目录包括 `/` 和 `/root`。

你永远不应该使用这些目录作为你的 `document root`。

使用这些目录的话，等于打开了潘多拉魔盒，请求会超出你的预期获取到隐私的数据。

永远也不要这样做！！！（对，我们还是要看下飞蛾扑火的配置长什么样子）

```
1. server {
2. root /;
3.
4. location / {
5. try_files /web/$uri $uri @php;
6. }
7.
8. location @php {
9. [...]
10. }
11. }
```

当一个对 `/foo` 的请求，会传递给 PHP 处理，因为文件没有找到。

这可能没有问题，直到遇到 `/etc/passwd` 这个请求。没错，你刚才给了我们这台服务器的所有用

户列表。

在某些情况下，Nginx 的 workers 甚至是 root 用户运行的。那么，我们现在有你的用户列表，以及密码哈希值，我们也知道哈希的方法。这台服务器已经变成我们的肉鸡了。

Filesystem Hierarchy Standard (FHS) 定义了数据应该如何存在。你一定要去阅读下。简单点儿说，你应该把 web 的内容 放在 `/var/www/`，`/srv` 或者 `/usr/share/www` 里面。

## 使用默认的 Document Root

在 Ubuntu、Debian 等操作系统中，Nginx 会被封装成一个易于安装的包，里面通常会提供一个“默认”的配置文件作为范例，也通常包含一个 `document root` 来保存基础的 HTML 文件。

大部分这些打包系统，并没有检查默认的 `document root` 里面的文件是否修改或者存在。在包升级的时候，可能会导致代码失效。有经验的系统管理员都知道，不要假设默认的 `document root` 里面的数据在升级的时候会原封不动。

你不应该使用默认的 `document root` 做网站的任何关键文件的目录。并没有默认的 `document root` 目录会保持不变这样的约定，你网站的关键数据，很可能在更新和升级系统提供的 Nginx 包时丢失。

## 使用主机名来解析地址

糟糕的配置：

```
1. upstream {
2. server http://someserver;
3. }
4.
5. server {
6. listen myhostname:80;
7. # [...]
8. }
```

你不应该在 `listen` 指令里面使用主机名。虽然这样可能是有效的，但它会带来层出不穷的问题。其中一个问题是，这个主机名在启动时或者服务重启中不能解析。这会导致 Nginx 不能绑定所需的 TCP socket 而启动失败。

一个更安全的做法是使用主机名对应 IP 地址，而不是主机名。这可以防止 Nginx 去查找 IP 地址，也去掉了去内部、外部解析程序的依赖。

例子中的 `upstream location` 也有同样的问题，虽然有时候在 `upstream` 里面不可避免要使用



到主机名，  
但这是一个不好的实践，需要仔细考虑以防出现问题。

推荐的配置：

```
1. upstream {
2. server http://10.48.41.12;
3. }
4.
5. server {
6. listen 127.0.0.16:80;
7. # [...]
8. }
```

## 在 HTTPS 中使用 SSLv3

由于 SSLv3 的 [POODLE 漏洞](#)，  
建议不要在开启 SSL 的网站使用 SSLv3。  
你可以简单粗暴的直接禁用 SSLv3，用 TLS 来替代：

```
1. ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
```

# LuaCjsonLibrary

- [LuaCjsonLibrary](#)

## LuaCjsonLibrary

---

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它基于 ECMAScript 的一个子集。JSON 采用完全独立于语言的文本格式，但是也使用了类似于 C 语言家族的习惯（包括 C、C++、C#、Java、JavaScript、Perl、Python 等）。这些特性使 JSON 成为理想的数据交换语言。易于人阅读和编写，同时也易于机器解析和生成(网络传输速率)。

JSON 格式被广泛的使用于各类不同应用场景，有些是 REST + JSON API，还有大部分不同应用、组件之间沟通的中间数据也是有 JSON 来完成的。由于他可读性、体积、编解码效率相比 XML 有很大优势，非常值得推荐。

# json解析的异常捕获

- json 解析的异常捕获

## json 解析的异常捕获

首先来看最最普通的一个 json 解析的例子（被解析的 json 字符串是错误的，缺少一个双引号）：

```
1. -- http://www.kyne.com.au/~mark/software/lua-cjson.php
2. -- version: 2.1 devel
3.
4. local json = require("cjson")
5. local str = [[{"key:"value"}]]
6.
7. local t = json.decode(str)
8. ngx.say(" --> ", type(t))
9.
10.
11. -- ... do the other things
12. ngx.say("all fine")
```

代码执行错误日志如下：

```
1. 2015/06/27 00:01:42 [error] 2714#0: *25 lua entry thread aborted: runtime error:
...ork/git/github.com/lua-resty-memcached-server/t/test.lua:8: Expected colon but found invalid
token at character 9
2. stack traceback:
3. coroutine 0:
4. [C]: in function 'decode'
5. ...ork/git/github.com/lua-resty-memcached-server/t/test.lua:8: in function
<...ork/git/github.com/lua-resty-memcached-server/t/test.lua:1>, client: 127.0.0.1, server:
localhost, request: "GET /test HTTP/1.1", host: "127.0.0.1:8001"
```

这可不是期望结果：decode 失败，500 错误直接退了。改良了一下代码：

```
1. local json = require("cjson")
2.
3. local function _json_decode(str)
4. return json.decode(str)
5. end
6.
7. function json_decode(str)
8. local ok, t = pcall(_json_decode, str)
```

```
9. if not ok then
10. return nil
11. end
12.
13. return t
14. end
```

如果需要在 Lua 中处理错误，必须使用函数 `pcall (protected call)` 来包装需要执行的代码。

`pcall` 接收一个函数和要传递给后者的参数，并执行，执行结果：有错误、无错误；返回值 `true` 或者 `false`, `errorinfo`。`pcall` 以一种“保护模式”来调用第一个参数，因此 `pcall` 可以捕获函数执行中的任何错误。有兴趣的同学，请更多了解下 Lua 中的异常处理。

另外，可以使用 `CJSON 2.1.0`，该版本新增一个 `cjson.safe` 模块接口，该接口兼容 `cjson` 模块，并且在解析错误时不抛出异常，而是返回 `nil`。

```
1. local json = require("cjson.safe")
2. local str = [[{"key":"value"}]]
3.
4. local t = json.decode(str)
5. if t then
6. ngx.say(" --> ", type(t))
7. end
```

# 稀疏数组

- 稀疏数组

## 稀疏数组

请看示例代码（注意 data 的数组下标）：

```
1. -- http://www.kyne.com.au/~mark/software/lua-cjson.php
2. -- version: 2.1 devel
3.
4. local json = require("cjson")
5.
6. local data = {1, 2}
7. data[1000] = 99
8.
9. -- ... do the other things
10. ngx.say(json.encode(data))
```

运行日志报错结果：

```
1. 2015/06/27 00:23:13 [error] 2714#0: *40 lua entry thread aborted: runtime error:
...ork/git/github.com/lua-resty-memcached-server/t/test.lua:13: Cannot serialise table:
excessively sparse array
2. stack traceback:
3. coroutine 0:
4. [C]: in function 'encode'
5. ...ork/git/github.com/lua-resty-memcached-server/t/test.lua:13: in function
<...ork/git/github.com/lua-resty-memcached-server/t/test.lua:1>, client: 127.0.0.1, server:
localhost, request: "GET /test HTTP/1.1", host: "127.0.0.1:8001"
```

如果把 data 的数组下标修改成 5，那么这个 json.encode 就会是成功的。

结果是：[1, 2, null, null, 99]

为什么下标是 1000 就失败呢？实际上这么做是 cjson 想保护你的内存资源。她担心这个下标过大直接撑爆内存（贴心小棉袄啊）。如果我们一定要让这种情况下可以 encode，就要尝试 `encode_sparse_array` API 了。有兴趣的同学可以自己试一试。我相信你看过有关 cjson 的代码后，就知道 cjson 的一个简单危险防范应该怎样完成的。

# 空table编码为array还是object

- 编码为 array 还是 object

## 编码为 array 还是 object

首先大家请看这段源码：

```
1. -- http://www.kyne.com.au/~mark/software/lua-cjson.php
2. -- version: 2.1 devel
3.
4. local json = require("cjson")
5. ngx.say("value --> ", json.encode({dogs={}}))
```

输出结果

```
value -> {"dogs":{}}
```

注意看下 encode 后 key 的值类型，“{}”代表 key 的值是个 object，“[]”则代表 key 的值是个数组。对于强类型语言(C/C++，Java 等)，这时候就有点不爽。因为类型不是他期望的要兼容。对于 Lua 本身，是把数组和字典融合到一起了，所以他是无法区分空数组和空字典的。

参考 [openresty/lua-cjson](#) 中额外贴出测试案例，我们就很容易找到思路了。

```
1. -- 内容节选lua-cjson-2.1.0.2/tests/agentzh.t
2. === TEST 1: empty tables as objects
3. --- lua
4. local cjson = require "cjson"
5. print(cjson.encode({}))
6. print(cjson.encode({dogs = {}}))
7. --- out
8. {}
9. {"dogs":{}}
10.
11.
12. === TEST 2: empty tables as arrays
13. --- lua
14. local cjson = require "cjson"
15. cjson.encode_empty_table_as_object(false)
16. print(cjson.encode({}))
17. print(cjson.encode({dogs = {}}))
18. --- out
19. []
20. {"dogs":[]}
```

综合本章节提到的各种问题，我们可以封装一个 `json_encode` 的示例函数：

```

1. local json = require("cjson")
2. --稀疏数组会被处理成object
3. json.encode_sparse_array(true)
4.
5. local function _json_encode(data)
6. return json.encode(data)
7. end
8.
9. function json_encode(data, empty_table_as_object)
10. --Lua的数据类型里面，array和dict是同一个东西。对应到json encode的时候，就会有不同的判断
11. --cjson对于空的table，就会被处理为object，也就是{}
12. --处理方法：对于cjson，使用encode_empty_table_as_object这个方法。
13. json.encode_empty_table_as_object(empty_table_as_object or false) -- 空的table默认为array
14. local ok, json_value = pcall(_json_encode, data)
15. if not ok then
16. return nil
17. end
18. return json_value
19. end

```

另一种思路是，使用 `setmetatable(data, json.empty_array_mt)`，来标记特定的 table，让 cjson 在编码这个空 table 时把它处理成 array：

```

1. local data = {}
2. setmetatable(data, json.empty_array_mt)
3. ngx.say("empty array: ", json.encode(data)) -- empty array: []

```

# PostgresNginxModule

- [PostgresNginxModule](#)

## PostgresNginxModule

---

PostgreSQL 是加州大学伯克利分校计算机系开发的对象关系型数据库管理系统(ORDBMS)，目前是免费开源的，且是全功能的自由软件数据库。

PostgreSQL 支持大部分 SQL 标准，其特性覆盖了 SQL-2/SQL-92 和 SQL-3/SQL-99，并且提供了许多其他现代特点，如复杂查询、外键、触发器、视图、事务完整性、多版本并行控制系统（MVCC）等。

PostgreSQL 可以使用许多方法扩展，比如，通过增加新的数据类型、函数、操作符、聚集函数、索引方法、过程语言等。

PostgreSQL 在灵活的 BSD 风格许可证下发行，任何人都可以根据自己的需要免费使用、修改和分发 PostgreSQL，不管是用于私人、商业、还是学术研究。

ngx\_postgres 是一个提供 Nginx 与 PostgreSQL 直接通讯的 upstream 模块。应答数据采用了 rds 格式，所以模块与 ngx\_rds\_json 和 ngx\_drizzle 模块是兼容的。



## 调用方式简介

- [PostgresNginxModule 模块的调用方式](#)
  - [ngx\\_postgres 模块使用方法](#)
  - [与 resty-mysql 调用方式的不同](#)

## PostgresNginxModule 模块的调用方式

### ngx\_postgres 模块使用方法

```
1. location /postgres {
2. internal;
3.
4. default_type text/html;
5. set_by_lua_block $query_sql {return ngx.unescape_uri(ngx.var.arg_sql)}
6.
7. postgres_pass pg_server;
8. rds_json on;
9. rds_json_buffer_size 16k;
10. postgres_query $query_sql;
11. postgres_connect_timeout 1s;
12. postgres_result_timeout 2s;
13. }
```

这里有很多指令要素：

- `internal` 这个指令指定所在的 `location` 只允许使用于处理内部请求，否则返回 404。
- `set_by_lua` 这一段内嵌的 Lua 代码用于计算出 `$query_sql` 变量的值，即后续通过指令 `postgres_query` 发送给 PostgreSQL 处理的 SQL 语句。这里使用了 GET 请求的 `query` 参数作为 SQL 语句输入。
- `postgres_pass` 这个指令可以指定一组提供后台服务的 PostgreSQL 数据库的 `upstream` 块。
- `rds_json` 这个指令是 `ngx_rds_json` 提供的，用于指定 `ngx_rds_json` 的 `output` 过滤器的开关状态，其模块作用就是一个用于把 `rds` 格式数据转换成 `json` 格式的 `output filter`。这个指令在这里出现意思是让 `ngx_rds_json` 模块帮助 `ngx_postgres` 模块把模块输出数据转换成 `json` 格式的数据。
- `rds_json_buffer_size` 这个指令指定 `ngx_rds_json` 用于每个连接的数据转换的内存大小。默认是 4/8k，适当加大此参数，有利于减少 CPU 消耗。
- `postgres_query` 指定 SQL 查询语句，查询语句将会直接发送给 PostgreSQL 数据库。
- `postgres_connect_timeout` 设置连接超时时间。
- `postgres_result_timeout` 设置结果返回超时时间。

这样的配置就完成了初步的可以提供其他 location 调用的 location 了。但这里还差一个配置没说明白，就是这一行：

```
1. postgres_pass pg_server;
```

其实这一行引入了 名叫 pg\_server 的 upstream 块，其定义应该像如下：

```
1. upstream pg_server {
2. postgres_server 192.168.1.2:5432 dbname=pg_database
3. user=postgres password=postgres;
4. postgres_keepalive max=800 mode=single overflow=reject;
5. }
```

这里有一些指令要素：

- postgres\_server 这个指令是必须带的，但可以配置多个，用于配置服务器连接参数，可以分解成若干参数：
  - 直接跟在后面的应该是服务器的 IP:Port
  - dbname 是服务器要连接的 PostgreSQL 的数据库名称。
  - user 是用于连接 PostgreSQL 服务器的账号名称。
  - password 是账号名称对应的密码。
- postgres\_keepalive 这个指令用于配置长连接连接池参数，长连接连接池有利于提高通讯效率，可以分解为若干参数：
  - max 是工作进程可以维护的连接池最大长连接数量。
  - mode 是后端匹配模式，在postgres\_server 配置了多个的时候发挥作用，有 single 和 multi 两种值，一般使用 single 即可。
  - overflow 是当长连接数量到达 max 之后的处理方案，有 ignore 和 reject 两种值。
    - ignore 允许创建新的连接与数据库通信，但完成通信后马上关闭此连接。
    - reject 拒绝访问并返回 503 Service Unavailable

这样就构成了我们 PostgreSQL 后端通讯的通用 location，在使用 Lua 业务编码的过程中可以直接使用如下代码连接数据库（折腾了这么老半天）：

```
1. local json = require "cjson"
2.
3. function test()
4. local res = ngx.location.capture('/postgres',
5. { args = {sql = "SELECT * FROM test" } }
6.)
7. }
```

```

8. local status = res.status
9. local body = json.decode(res.body)
10.
11. if status == 200 then
12. status = true
13. else
14. status = false
15. end
16. return status, body
17. end

```

## 与 resty-mysql 调用方式的不同

先来看一下 `lua-resty-mysql` 模块的调用示例代码。

```

1. # you do not need the following line if you are using
2. # the ngx_openresty bundle:
3. lua_package_path "/path/to/lua-resty-mysql/lib/?.lua;";
4.
5. server {
6. location /test {
7. content_by_lua_block {
8. local mysql = require "resty.mysql"
9. local db, err = mysql:new()
10. if not db then
11. ngx.say("failed to instantiate mysql: ", err)
12. return
13. end
14.
15. db:set_timeout(1000) -- 1 sec
16.
17. local ok, err, errno, sqlstate = db:connect{
18. host = "127.0.0.1",
19. port = 3306,
20. database = "ngx_test",
21. user = "ngx_test",
22. password = "ngx_test",
23. max_packet_size = 1024 * 1024 }
24.
25. if not ok then
26. ngx.say("failed to connect: ", err, ": ", errno, " ", sqlstate)
27. return
28. end
29.
30. ngx.say("connected to mysql.")
31.
32. -- run a select query, expected about 10 rows in

```

```

33. -- the result set:
34. res, err, errno, sqlstate =
35. db:query("select * from cats order by id asc", 10)
36. if not res then
37. ngx.say("bad result: ", err, ": ", errno, ": ", sqlstate, ".")
38. return
39. end
40.
41. local cjson = require "cjson"
42. ngx.say("result: ", cjson.encode(res))
43.
44. -- put it into the connection pool of size 100,
45. -- with 10 seconds max idle timeout
46. local ok, err = db:set_keepalive(10000, 100)
47. if not ok then
48. ngx.say("failed to set keepalive: ", err)
49. return
50. end
51. }
52. }
53. }

```

看过这段代码，大家肯定会说：这才是我熟悉的，我想要的。为什么刚刚 `ngx_postgres` 模块的调用这么诡异，配置那么复杂，其实这是发展历史造成的。`ngx_postgres` 起步比较早，当时 `OpenResty` 也还没开始流行，所以更多的 Nginx 数据库都是以 `ngx_c_module` 方式存在。有了 `OpenResty`，才让我们具有了使用完整的语言来描述我们业务能力。

后面我们会单独说一说使用 `ngx_c_module` 的各种不方便，也就是我们所踩过的坑。希望能给大家一个警示，能转到 `lua-resty-***` 这个方向的，就千万不要和 `ngx_c_module` 玩，`ngx_c_module` 的扩展性、可维护性、升级等各方面都没有 `lua-resty-***` 好。

这绝对是经验的总结。不服来辩！

# 不支持事务

- 不支持事务

## 不支持事务

我们继续上一章节的内容，大家应该记得我们 `Lua` 代码中是如何完成的 `ngx_postgres` 模块调用的。我们把他简单改造一下，让他更接近真实代码。

```
1. local json = require "cjson"
2.
3. function db_exec(sql_str)
4. local res = ngx.location.capture('/postgres',
5. { args = {sql = sql_str } }
6.)
7.
8. local status = res.status
9. local body = json.decode(res.body)
10.
11. if status == 200 then
12. status = true
13. else
14. status = false
15. end
16. return status, body
17. end
18.
19. -- 转账操作，对ID=100的用户加10，同时对ID=200的用户减10。
20. ? local status
21. ? status = db_exec("BEGIN")
22. ? if status then
23. ? db_exec("ROLLBACK")
24. ? end
25. ?
26. ? status = db_exec("UPDATE ACCOUNT SET MONEY=MONEY+10 WHERE ID = 100")
27. ? if status then
28. ? db_exec("ROLLBACK")
29. ? end
30. ?
31. ? status = db_exec("UPDATE ACCOUNT SET MONEY=MONEY-10 WHERE ID = 200")
32. ? if status then
33. ? db_exec("ROLLBACK")
34. ? end
35. ?
36. ? db_exec("COMMIT")
```

后面这部分有问题的代码，在没有并发的场景下使用，是不会有问题的。但是这段代码在高并发应用场景下，错误百出。你会发现最后执行结果完全摸不清楚。明明是个转账逻辑，一个收入，一直支出，最后却发现总收入比支出要大。如果这个错误发生在金融领域，那不知道要赔多少钱。

如果你能靠自己很快明白错误的原因，那么恭喜你你对数据库连接 `Nginx` 机理都是比较清楚的。如果你想不明白，那就听我给你掰一掰这面的小知识。

数据库的事物成功执行，事物相关的所有操作是必须执行在一条连接上的。`SQL` 的执行情况类似这样：

```
1. 连接：`BEGIN` -> `SQL(UPDATE、DELETE... ..)` -> `COMMIT`。
```

但如果你创建了两条连接，每条连接提交的 `SQL` 语句是下面这样：

```
1. 连接1：`BEGIN` -> `SQL(UPDATE、DELETE... ..)`
2. 连接2：`COMMIT`
```

这时就会出现连接 1 的内容没有被提交，行锁产生。连接 2 提交了一个空的 `COMMIT`。

说到这里你可能开始鄙视我了，谁疯了非要创建两条连接来这么用 `SQL` 啊。又麻烦，又不好看，貌似从来没听说过还有人在一次请求中创建多个数据库连接，简直就是非人类嘛。

或许你不会主动、显示的创建多个连接，但是刚刚的示例代码，高并发下这个事物的每个 `SQL` 语句都可能落在不同的连接上。为什么呢？这是因为通过 `ngx.location.capture` 跳转到 `/postgres` 小节后，`Nginx` 每次都会从连接池中挑选一条空闲连接，而当时哪条连接是空闲的，完全没法预估。所以上面的第二个例子，就这么静悄悄的发生了。如果你不了解 `Nginx` 的机理，那么他肯定会一直困扰你。为什么一会儿好，一会儿不好。

同样的道理，我们推理到 `DrizzleNginxModule`、`RedisNginxModule`、`Redis2NginxModule`，他们都是无法做到在两次连续请求落到同一个连接上的。

由于这个 `Bug` 藏得比较深，并且不太好讲解，所以我觉得生产中最好用 `lua-resty-*` 这类的库，更符合标准调用习惯，直接可以绕过这些坑。不要为了一点点的性能，牺牲了更大的蛋糕。看得见的，看不见的，都要了解用用，最后再做决定，肯定不吃亏。

# 超时

- 超时

## 超时

当我们所有数据库的 `SQL` 语句是通过子查询方式完成，对于超时的控制往往很容易被大家忽略。因为大家在代码里看不到任何调用 `set_timeout` 的地方。实际上 `PostgreSQL` 已经为我们预留好了两个设置。

请参考下面这段配置：

```
1. location /postgres {
2. internal;
3.
4. default_type text/html;
5. set_by_lua_block $query_sql {return ngx.unescape_uri(ngx.var.arg_sql)}
6.
7. postgres_pass pg_server;
8. rds_json on;
9. rds_json_buffer_size 16k;
10. postgres_query $query_sql;
11. postgres_connect_timeout 1s;
12. postgres_result_timeout 2s;
13. }
```

生产中使用这段配置，遇到了一个不大不小的坑。在我们的开发机、测试环境上都没有任何问题的安装包，到了用户那边出现所有数据库操作异常，而且是数据库连接失败，但手工连接本地数据库，发现没有任何问题。同样的执行程序再次 copy 回来后，公司内环境不能复现问题。考虑到我们当次升级刚好修改了 `postgres_connect_timeout` 和 `postgres_result_timeout` 的默认值，所以我们尝试去掉了这两行个性设置，重启服务后一切都好了。

起初我们也很怀疑出了什么诡异问题，要知道我们的 `nginx` 和 `PostgreSQL` 可是安装在本机，都是使用 `127.0.0.1` 这样的 IP 来完成通信的，难道客户的机器在这个时间内还不能完成连接建立？

经过后期排插问题，发现是客户的机器上安装了一些趋势科技的杀毒客户端，而趋势科技为了防止无效连接，对所有连接的建立均阻塞了一秒钟。就是这一秒钟，让我们的服务彻底歇菜。

本以为是一次比较好的优化，没想到因为这个原因没能保留下来，反而给大家带来麻烦。只能说企业版环境复杂，边界比较多。但也好在我们一直使用最常见的技术、最常见的配置解决各种问题，让我们的经验可以复用到其他公司里。

超时



# 健康监测

- [健康监测](#)

# 健康监测

---

# SQL注入

- SQL 注入

## SQL 注入

有使用 SQL 语句操作数据库的经验朋友，应该都知道使用 SQL 过程中有一个安全问题叫 SQL 注入。所谓 SQL 注入，就是通过把 SQL 命令插入到 Web 表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的 SQL 命令。

为了防止 SQL 注入，在生产环境中使用 OpenResty 的时候就要注意添加防范代码。

延续之前的 ngx\_postgres 调用代码的使用，

```
1. local sql_normal = [[select id, name from user where name=']] .. ngx.var.arg_name .. [[' and
 password=']] .. ngx.var.arg_password .. [[' limit 1;]]
2.
3. local res = ngx.location.capture('/postgres',
4. { args = {sql = sql } }
5.)
6.
7. local body = json.decode(res.body)
8.
9. if (table.getn(res) > 0) {
10. return res[1];
11. }
12.
13. return nil;
```

假设我们在用户登录使用上 SQL 语句查询账号是否账号密码正确，用户可以通过 GET 方式请求并发送登录信息比如：

```
1. # http://localhost/login?name=person&password=12345
```

那么我们上面的代码通过 ngx.var.arg\_name 和 ngx.var.arg\_password 获取查询参数，并且与 SQL 语句格式进行字符串拼接，最终 sql\_normal 会是这个样子的：

```
1. local sql_normal = [[select id, name from user where name='person' and password='12345'
 limit 1;]]
```

正常情况下，如果 person 账号存在并且 password 是 12345，那么 sql 执行结果就应该是能返回 id 号的。这个接口如果暴露在攻击者面前，那么攻击者很可能会让参数这样传入：

```

1. name="' or '='"
2. password="' or '='"

```

那么这个 `sql_normal` 就会变成一个永远都能执行成功的语句了。

```

1. local sql_normal = [[select id, name from user where name="' or '='" and password="' or
 '='" limit 1;]]

```

这就是一个简单的 `sql inject`（注入）的案例，那么问题来了，面对这么凶猛的攻击者，我们有什么办法防止这种 SQL 注入呢？

很简单，我们只要把传入参数的变量做一次字符转义，把不该作为破坏 SQL 查询语句结构的双引号或者单引号等做转义，把 `'` 转义成 `\'`，那么变量 `name` 和 `password` 的内容还是乖乖的作为查询条件传入，他们再也不能为非作歹了。

那么怎么做到字符转义呢？要知道每个数据库支持的 SQL 语句格式都不太一样啊，尤其是双引号和单引号的应用上。有几个选择：

```

1. ndk.set_var.set_quote_sql_str()
2. ndk.set_var.set_quote_pgsql_str()
3. ngx.quote_sql_str()

```

这三个函数，前面两个是 `ndk.set_var` 跳转调用，其实是 `HttpSetMiscModule` 这个模块提供的函数，是一个 C 模块实现的函数，`ndk.set_var.set_quote_sql_str()` 是用于 MySQL 格式的 SQL 语句字符转义，而 `set_quote_pgsql_str` 是用于 PostgreSQL 格式的 SQL 语句字符转义。最后 `ngx.quote_sql_str` 是一个 `ngx_lua` 模块中实现的函数，也是用于 MySQL 格式的 SQL 语句字符转义。

让我们看看代码怎么写：

```

1. local name = ngx.quote_sql_str(ngx.var.arg_name)
2. local password = ngx.quote_sql_str(ngx.var.arg_password)
3. local sql_normal = [[select id, name from user where name=]] .. name .. [[and password=]]
 .. password .. [[limit 1;]]
4.
5. local res = ngx.location.capture('/postgres',
6. { args = {sql = sql } }
7.)
8.
9. local body = json.decode(res.body)
10.
11. if (table.getn(res) > 0) {
12. return res[1];
13. }
14.

```

```
15. return nil;
```

注意上述代码有两个变化：

1. \* 用 `ngx.quote_sql_str` 把 `ngx.var.arg_name` 和 `ngx.var.arg_password` 包了一层，把返回值作为 `sql` 语句拼凑起来。
2. \* 原本在 `sql` 语句中添加的单引号去掉了，因为 `ngx.quote_sql_str` 的返回值正确的带上引号了。

这样已经可以抵御 SQL 注入的攻击手段了，但开发过程中需要不断的产生新功能新代码，这时候也一定注意不要忽视 SQL 注入的防护，安全防御代码就想织网一样，只要有一处漏洞，鱼儿可就游走了。

# LuaNginxModule

- [LuaNginxModule](#)
  - [链接](#)

## LuaNginxModule

---

### 链接

---

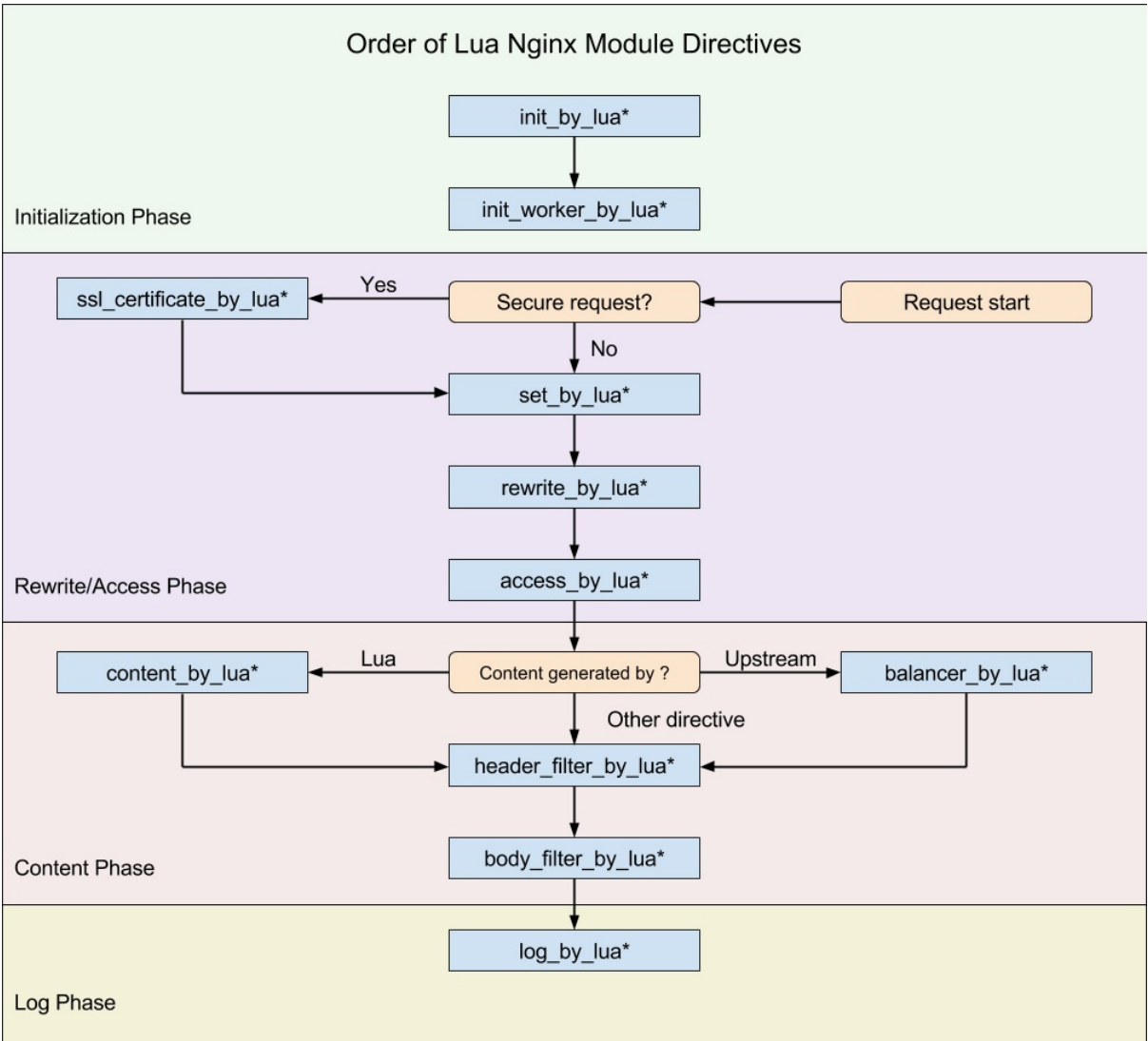
- [执行阶段概念](#)
- [正确的记录日志](#)
- [热装载代码](#)
- [阻塞操作](#)
- [缓存](#)
- [sleep](#)
- [定时任务](#)
- [禁止某些终端访问](#)
- [请求返回后继续执行](#)
- [调试](#)
- [请求中断后的处理](#)
- [我的 lua 代码需要调优么](#)
- [变量的共享范围](#)
- [动态限速](#)
- [shared.dict 非队列性质](#)
- [正确使用长链接](#)
- [如何引用第三方 resty 库](#)
- [典型应用场景](#)
- [怎样理解 cosocket](#)
- [如何安全启动唯一实例的 timer](#)
- [如何正确的解析域名](#)

# 执行阶段概念

- [执行阶段概念](#)

# 执行阶段概念

OpenResty 处理一个请求，它的处理流程请参考下图（从 Request start 开始）：



我们在这里做个测试，示例代码如下：

```
1. location /mixed {
2. set_by_lua_block $a {
3. ngx.log(ngx.ERR, "set_by_lua*")
4. }
5. rewrite_by_lua_block {
```

```

6. ngx.log(ngx.ERR, "rewrite_by_lua*")
7. }
8. access_by_lua_block {
9. ngx.log(ngx.ERR, "access_by_lua*")
10. }
11. content_by_lua_block {
12. ngx.log(ngx.ERR, "content_by_lua*")
13. }
14. header_filter_by_lua_block {
15. ngx.log(ngx.ERR, "header_filter_by_lua*")
16. }
17. body_filter_by_lua_block {
18. ngx.log(ngx.ERR, "body_filter_by_lua*")
19. }
20. log_by_lua_block {
21. ngx.log(ngx.ERR, "log_by_lua*")
22. }
23. }

```

执行结果日志 (截取了一下):

```

1. set_by_lua*
2. rewrite_by_lua*
3. access_by_lua*
4. content_by_lua*
5. header_filter_by_lua*
6. body_filter_by_lua*
7. log_by_lua*

```

这几个阶段的存在, 应该是 OpenResty 不同于其他多数 Web 平台编程的最明显特征了。由于 Nginx 把一个请求分成了很多阶段, 这样第三方模块就可以根据自己行为, 挂载到不同阶段进行处理达到目的。OpenResty 也应用了同样的特性。所不同的是, OpenResty 挂载的是我们编写的 Lua 代码。

这样我们就可以根据我们的需要, 在不同的阶段直接完成大部分典型处理了。

- `set_by_lua*`: 流程分支处理判断变量初始化
- `rewrite_by_lua*`: 转发、重定向、缓存等功能 (例如特定请求代理到外网)
- `access_by_lua*`: IP 准入、接口权限等情况集中处理 (例如配合 iptable 完成简单防火墙)
- `content_by_lua*`: 内容生成
- `header_filter_by_lua*`: 响应头部过滤处理 (例如添加头部信息)
- `body_filter_by_lua*`: 响应体过滤处理 (例如完成应答内容统一成大写)
- `log_by_lua*`: 会话完成后本地异步完成日志记录 (日志可以记录在本地, 还可以同步到其他机器)

实际上我们只使用其中一个阶段 `content_by_lua*`，也可以完成所有的处理。但这样做，会让我们的代码比较臃肿，越到后期越发难以维护。把我们的逻辑放在不同阶段，分工明确，代码独立，后期发力可以有很多有意思的玩法。

举一个例子，如果在最开始的开发中，请求体和响应体都是通过 HTTP 明文传输，后面需要使用 aes 加密，利用不同的执行阶段，我们可以非常简单的实现：

```
1. # 明文协议版本
2. location /mixed {
3. content_by_lua_file ...; # 请求处理
4. }
5.
6. # 加密协议版本
7. location /mixed {
8. access_by_lua_file ...; # 请求加密解码
9. content_by_lua_file ...; # 请求处理，不需要关心通信协议
10. body_filter_by_lua_file ...; # 应答加密编码
11. }
```

内容处理部分都是在 `content_by_lua*` 阶段完成，第一版本 API 接口开发都是基于明文。为了传输体积、安全等要求，我们设计了支持压缩、加密的密文协议(上下行)，痛点就来了，我们要更改所有 API 的入口、出口么？

最后我们是在 `access_by_lua*` 完成密文协议解码，`body_filter_by_lua*` 完成应答加密编码。如此一来世界都宁静了，我们没有更改已实现功能的一行代码，只是利用 OpenResty 的阶段处理特性，非常优雅的解决了这个问题。

不同的阶段，有不同的处理行为，这是 OpenResty 的一大特色。学会它，适应它，会给你打开新的一扇门。这些东西不是 OpenResty 自身所创，而是 Nginx module 对外开放的处理阶段。理解了他，也能更好的理解 Nginx 的设计思维。



## 正确的记录日志

- [正确的记录日志](#)

## 正确的记录日志

看过本章第一节的同学应该还记得，`log_by_lua*` 是一个请求经历的最后阶段。由于记日志跟应答内容无关，Nginx 通常在结束请求之后才更新访问日志。由此可见，如果有日志输出的情况，最好统一到 `log_by_lua*` 阶段。如果我们把记日志的操作放在 `content_by_lua*` 阶段，那么将线性的增加请求处理时间。

在公司某个定制化项目中，Nginx 上的日志内容都要输送到 syslog 日志服务器。我们使用了 [lua-resty-logger-socket](#) 这个库。

调用示例代码如下（有问题的）：

```
1. -- lua_package_path "/path/to/lua-resty-logger-socket/lib/?.lua;";
2. --
3. -- server {
4. -- location / {
5. -- content_by_lua_file lua/log.lua;
6. -- }
7. -- }
8.
9. -- lua/log.lua
10. local logger = require "resty.logger.socket"
11. if not logger.initted() then
12. local ok, err = logger.init{
13. host = 'xxx',
14. port = 1234,
15. flush_limit = 1, --日志长度大于flush_limit的时候会将msg信息推送一次
16. drop_limit = 99999,
17. }
18. if not ok then
19. ngx.log(ngx.ERR, "failed to initialize the logger: ",err)
20. return
21. end
22. end
23.
24. local msg = string.format(....)
25. local bytes, err = logger.log(msg)
26. if err then
27. ngx.log(ngx.ERR, "failed to log message: ", err)
28. return
29. end
```

在实测过程中我们发现了些问题：

- 缓存无效：如果 `flush_limit` 的值稍大一些（例如 2000），会导致某些体积比较小的日志出现莫名其妙的丢失，所以我们只能把 `flush_limit` 调整的很小
- 自己拼写 `msg` 所有内容，比较辛苦

那么我们来查看[lua-resty-logger-socket](#)这个库的 `log` 函数是如何实现的呢，代码如下：

```

1. function _M.log(msg)
2. ...
3.
4. if (debug) then
5. ngx.update_time()
6. ngx_log(DEBUG, ngx.now(), ":log message length: " .. #msg)
7. end
8.
9. local msg_len = #msg
10.
11. if (is_exiting()) then
12. exiting = true
13. _write_buffer(msg)
14. _flush_buffer()
15. if (debug) then
16. ngx_log(DEBUG, "Nginx worker is exiting")
17. end
18. bytes = 0
19. elseif (msg_len + buffer_size < flush_limit) then -- 历史日志大小+本地日志大小小于推送上限
20. _write_buffer(msg)
21. bytes = msg_len
22. elseif (msg_len + buffer_size <= drop_limit) then
23. _write_buffer(msg)
24. _flush_buffer()
25. bytes = msg_len
26. else
27. _flush_buffer()
28. if (debug) then
29. ngx_log(DEBUG, "logger buffer is full, this log message will be "
30. .. "dropped")
31. end
32. bytes = 0
33. --- this log message doesn't fit in buffer, drop it
34.
35. ...

```

由于在 `content_by_lua*` 阶段变量的生命周期会随着请求的终结而终结，所以当日志量小于

`flush_limit` 的情况下这些日志就不能被累积，也不会触发 `_flush_buffer` 函数，所以小日志会丢失。

这些坑回头看来这么明显，所有的问题都是因为我们把 `lua/log.lua` 用错阶段了，应该放到 `log_by_lua*` 阶段，所有的问题都不复存在。

修正后：

```
1. lua_package_path "/path/to/lua-resty-logger-socket/lib/?.lua;;";
2.
3. server {
4. location / {
5. content_by_lua_file lua/content.lua;
6. log_by_lua_file lua/log.lua;
7. }
8. }
```

这里有个新问题，如果我的 `log` 里面需要输出一些 `content` 的临时变量，两阶段之间如何传递参数呢？

方法肯定有，推荐下面这个：

```
1. location /test {
2. rewrite_by_lua_block {
3. ngx.say("foo = ", ngx.ctx.foo)
4. ngx.ctx.foo = 76
5. }
6. access_by_lua_block {
7. ngx.ctx.foo = ngx.ctx.foo + 3
8. }
9. content_by_lua_block {
10. ngx.say(ngx.ctx.foo)
11. }
12. }
```

更多有关 `ngx.ctx` 信息，请看[这里](#)。

## 热装载代码

- [热装载代码](#)
  - [自定义 module 的动态装载](#)
  - [自定义 Lua script 的动态装载实现](#)

## 热装载代码

在 OpenResty 中，提及热加载代码，估计大家的第一反应是 `lua_code_cache` 这个开关。但 `lua_code_cache off` 的工作原理，是给每个请求创建一个独立的 Lua VM。即使抛去性能因素不谈，考虑到程序的正确性，也不应该在生产环境中关闭 `lua_code_cache`。

那么我们是否可以在生产环境中完成热加载呢？

- 代码有变动时，自动加载最新 Lua 代码，但是 Nginx 本身，不做任何 reload。
- 自动加载后的代码，享用 `lua_code_cache on` 带来的高效特性。

这里有多种玩法（[引自 OpenResty 讨论组](#)）：

- 使用 HUP reload 或者 binary upgrade 方式动态加载 Nginx 配置或重启 Nginx。这不会导致中间有请求被 drop 掉。
- 当 `content_by_lua_file` 里使用 Nginx 变量时，是可以动态加载新的 Lua 脚本的，不过要记得对 Nginx 变量的值进行基本的合法性验证，以免被注入攻击。

```
1. location ~ '^/lua/(\w+(?:\./\w+)*)$' {
2. content_by_lua_file $1;
3. }
```

- 自己从外部数据源（包括文件系统）加载 Lua 源码或字节码，然后使用 `loadstring()` “eval” 进 Lua VM。可以通过 `package.loaded.* = nil` 自己来做缓存，毕竟频繁地加载源码和调用 `loadstring()`，以及频繁地 JIT 编译还是很昂贵的。比如 CloudFlare 公司采用的方法是从 modsecurity 规则编译出来的 Lua 代码就是通过 KyotoTycoon 动态分发到全球网络中的每一个 Nginx 服务器的。无需 reload 或者 binary upgrade。

## 自定义 module 的动态装载

对于已经装载的 module，我们可以通过 `package.loaded.* = nil` 的方式卸载（注意：如果对应模块是通过本地文件 `require` 加载的，该方式失效，`ngx_lua_module` 里面对以文件加载模块的方式做了特殊处理）。

不过，值得提醒的是，因为 `require` 这个内建函数在标准 Lua 5.1 解释器和 LuaJIT 2 中都被

实现为 C 函数，所以你在自己的 loader 里可能并不能调用 ngx\_lua 那些涉及非阻塞 IO 的 Lua 函数。因为这些 Lua 函数需要 yield 当前的 Lua 协程，而 yield 是无法跨越 Lua 调用栈上的 C 函数帧的。细节见

<https://github.com/openresty/lua-nginx-module#lua-coroutine-yieldingresuming>

所以直接操纵 `package.loaded` 是最简单和最有效的做法。CloudFlare 的 Lua WAF 系统中就是这么做的。

不过，值得提醒的是，从 `package.loaded` 解注册的 Lua 模块会被 GC 掉。而那些使用下列某一个或某几个特性的 Lua 模块是不能被安全的解注册的：

- 使用 FFI 加载了外部动态库
- 使用 FFI 定义了新的 C 类型
- 使用 FFI 定义了新的 C 函数原型

这个限制对于所有的 Lua 上下文都是适用的。

这样的 Lua 模块应避免手动从 `package.loaded` 卸载。当然，如果你永不手工卸载这样的模块，只是动态加载的话，倒也无所谓了。但在我们的 Lua WAF 的场景，已动态加载的一些 Lua 模块还需要被热替换掉（但不重新创建 Lua VM）。

## 自定义 Lua script 的动态装载实现

引自 [OpenResty 讨论组](#)

一方面使用自定义的环境表 [1]，以白名单的形式提供用户脚本能访问的 API；另一方面，（只）为用户脚本禁用 JIT 编译，同时使用 Lua 的 debug hooks [2] 作脚本 CPU 超时保护（debug hooks 对于 JIT 编译的代码是不会执行的，主要是出于性能方面的考虑）。

下面这个小例子演示了这种玩法：

```
1. local user_script = [[
2. local a = 0
3. local rand = math.random
4. for i = 1, 200 do
5. a = a + rand(i)
6. end
7. ngx.say("hi")
8.]]
9.
10. local function handle_timeout(typ)
11. return error("user script too hot")
```

```

12. end
13.
14. local function handle_error(err)
15. return string.format("%s: %s", err or "", debug.traceback())
16. end
17.
18. -- disable JIT in the user script to ensure debug hooks always work:
19. user_script = [[jit.off(true, true)]] .. user_script
20.
21. local f, err = loadstring(user_script, "=user script")
22. if not f then
23. ngx.say("ERROR: failed to load user script: ", err)
24. return
25. end
26.
27. -- only enable math.*, and ngx.say in our sandbox:
28. local env = {
29. math = math,
30. ngx = { say = ngx.say },
31. jit = { off = jit.off },
32. }
33. setfenv(f, env)
34.
35. local instruction_limit = 1000
36. debug.sethook(handle_timeout, "", instruction_limit)
37. local ok, err = xpcall(f, handle_error)
38. if not ok then
39. ngx.say("failed to run user script: ", err)
40. end
41. debug.sethook() -- turn off the hooks

```

这个例子中我们只允许用户脚本调用 `math` 模块的所有函数、`ngx.say()` 以及 `jit.off()`。其中 `jit.off()` 是必需引用的，为的是在用户脚本内部禁用 JIT 编译，否则我们注册的 debug hooks 可能不会被调用。

另外，这个例子中我们设置了脚本最多只能执行 1000 条 VM 指令。你可以根据你自己的场景进行调整。

这里很重要的是，不能向用户脚本暴露 `pcall` 和 `xpcall` 这两个 Lua 指令，否则恶意用户会利用它故意拦截掉我们在 debug hook 里为中断脚本执行而抛出的 Lua 异常。

另外，`require()`、`loadstring()`、`loadfile()`、`dofile()`、`io.`、`os.` 等等 API 是一定不能暴露给不被信任的 Lua 脚本的。



## 阻塞操作

- [阻塞操作](#)
  - [常见语言代表异步框架](#)
  - [异步编程的噩梦](#)
  - [OpenResty 中的阻塞函数](#)

## 阻塞操作

OpenResty 的诞生，一直对外宣传是同步非阻塞(100% non-blocking)的。基于事件通知的 Nginx 给我们带来了足够强悍的高并发支持，但是也对我们的编码有特殊要求。这个特殊要求就是我们的代码，也必须是非阻塞的。如果你的服务端编程生涯一开始就是从异步框架开始的，恭喜你了。但如果你的编程生涯是从同步框架过来的，而且又是刚刚开始深入了解异步框架，那你就要小心了。

Nginx 为了减少系统上下文切换，它的 worker 是用单进程单线程设计的，事实证明这种做法运行效率很高。Nginx 要么是在等待网络讯号，要么就是在处理业务（请求数据解析、过滤、内容应答等），没有任何额外资源消耗。

## 常见语言代表异步框架

- Golang: 使用协程技术实现
- Python: `gevent` 基于协程的 Python 网络库
- Rust: 用的少，只知道语言完备支持异步框架
- OpenResty: 基于 Nginx，使用事件通知机制
- Java: `Netty`，使用网络事件通知机制

## 异步编程的噩梦

异步编程，如果从零开始，难度是非常大的。一个完整的请求，由于网络传输的非连续性，这个请求要被多次挂起、恢复、运行，一旦网络有新数据到达，都需要立刻唤醒恢复原始请求处于运行状态。开发人员不仅仅要考虑异步 API 接口本身的使用规范，还要考虑业务请求的完整处理，稍有不慎，全盘皆输。

最最重要的噩梦是，我们好不容易搞定异步框架和业务请求完整性，但是却在我们的业务请求上使用了阻塞函数。一开始没有任何感知，只有做压力测试的时候才发现我们的并发量上不去，各种卡顿，甚至开始怀疑人生：异步世界也就这样。

## OpenResty 中的阻塞函数



官方有明确说明，OpenResty 的官方 API 绝对 100% non-blocking，所以我们只能在她的外面寻找了。我这里大致归纳总结了一下，包含下面几种情况：

- 高 CPU 的调用（压缩、解压缩、加解密等）
- 高磁盘的调用（所有文件操作）
- 非 OpenResty 提供的网络操作（luasocket 等）
- 系统命令行调用（os.execute 等）

这些都应该是我们尽量要避免的。理想丰满，现实骨感，谁能保证我们的应用中不使用这些类型的 API？没人保证，我们能做的就是把他们的调用数量、频率降低再降低，如果还是不能满足我们需要，那么就考虑把他们封装成独立服务，对外提供 TCP/HTTP 级别的接口调用，这样我们的 OpenResty 就可以同时享受异步编程的好处又能达到我们的目的。

# 缓存

- 缓存
  - 缓存的原则
  - OpenResty 的缓存
    - 使用 Lua shared dict
    - 使用 Lua LRU cache
    - 如何选择？

## 缓存

### 缓存的原则

缓存是一个大型系统中非常重要的一个组成部分。在硬件层面，大部分的计算机硬件都会用缓存来提高速度，比如 CPU 会有多级缓存、RAID 卡也有读写缓存。在软件层面，我们用的数据库就是一个缓存设计非常好的例子，在 SQL 语句的优化、索引设计、磁盘读写的各个地方，都有缓存，建议大家在设计自己的缓存之前，先去了解下 MySQL 里面的各种缓存机制，感兴趣的可以去看下[High Performance MySQL](#)这本非常有价值的书。

一个生产环境的缓存系统，需要根据自己的业务场景和系统瓶颈，来找出最好的方案，这是一门平衡的艺术。

一般来说，缓存有两个原则。一是越靠近用户的请求越好，比如能用本地缓存的就不要发送 HTTP 请求，能用 CDN 缓存的就不要打到 Web 服务器，能用 Nginx 缓存的就不要用数据库的缓存；二是尽量使用本进程和本机的缓存解决，因为跨了进程和机器甚至机房，缓存的网络开销就会非常大，在高并发的时候会非常明显。

### OpenResty 的缓存

我们介绍下在 OpenResty 里面，有哪些缓存的方法。

#### 使用 Lua shared dict

我们看下面这段代码：

```
1. function get_from_cache(key)
2. local cache ngx = ngx.shared.my_cache
3. local value = cache ngx:get(key)
4. return value
5. end
6.
```

```

7. function set_to_cache(key, value, exptime)
8. if not exptime then
9. exptime = 0
10. end
11.
12. local cache ngx = ngx.shared.my_cache
13. local succ, err, forcible = cache ngx:set(key, value, exptime)
14. return succ
15. end

```

这里面用的就是 ngx shared dict cache。你可能会奇怪，ngx.shared.my\_cache 是从哪里冒出来的？没错，少贴了 nginx.conf 里面的修改：

```

1. lua_shared_dict my_cache 128m;

```

如同它的名字一样，这个 cache 是 Nginx 所有 worker 之间共享的，内部使用的 LRU 算法（最近最少使用）来判断缓存是否在内存占满时被清除。

## 使用 Lua LRU cache

直接复制下春哥的示例代码：

```

1. local _M = {}
2.
3. -- alternatively: local lrucache = require "resty.lrucache.pureffi"
4. local lrucache = require "resty.lrucache"
5.
6. -- we need to initialize the cache on the Lua module level so that
7. -- it can be shared by all the requests served by each nginx worker process:
8. local c = lrucache.new(200) -- allow up to 200 items in the cache
9. if not c then
10. return error("failed to create the cache: " .. (err or "unknown"))
11. end
12.
13. function _M.go()
14. c:set("dog", 32)
15. c:set("cat", 56)
16. ngx.say("dog: ", c:get("dog"))
17. ngx.say("cat: ", c:get("cat"))
18.
19. c:set("dog", { age = 10 }, 0.1) -- expire in 0.1 sec
20. c:delete("dog")
21. end
22.
23. return _M

```

可以看出来，这个 cache 是 worker 级别的，不会在 Nginx workers 之间共享。并且，它是预先分配好 key 的数量，而 shared dict 需要自己用 key 和 value 的大小和数量，来估算需要把内存设置为多少。

## 如何选择？

shared.dict 使用的是共享内存，每次操作都是全局锁，如果高并发环境，不同 worker 之间容易引起竞争。所以单个 shared.dict 的体积不能过大。lru\_cache 是 worker 内使用的，由于 Nginx 是单进程方式存在，所以永远不会触发锁，效率上有优势，并且没有 shared.dict 的体积限制，内存上也更弹性，但不同 worker 之间数据不同享，同一缓存数据可能被冗余存储。

你需要考虑的，一个是 Lua lru cache 提供的 API 比较少，现在只有 get、set 和 delete，而 ngx shared dict 还可以 add、replace、incr、get\_stale（在 key 过期时也可以返回之前的值）、get\_keys（获取所有 key，虽然不推荐，但说不定你的业务需要呢）；第二个是内存的占用，由于 ngx shared dict 是 workers 之间共享的，所以在多 worker 的情况下，内存占用比较少。

# sleep

- [sleep](#)
  - [为什么会这样？](#)

## sleep

这是一个比较常见的功能，你会怎么做呢？Google 一下，你会找到[Lua 的官方指南](#)，

里面介绍了 10 种 sleep 不同的方法（操作系统不一样，方法还有区别），选择一个用，然后你就杯具了：（你会发现 Nginx 高并发的特性不见了！

在 OpenResty 里面选择使用库的时候，有一个基本的原则：尽量使用 **OpenResty** 的库函数，尽量不用 **Lua** 的库函数，因为 **Lua** 的库都是同步阻塞的。

```

1. # you do not need the following line if you are using
2. # the ngx_openresty bundle:
3. lua_package_path "/path/to/lua-resty-redis/lib/?.lua;;";
4.
5. server {
6. location /non_block {
7. content_by_lua_block {
8. ngx.sleep(0.1)
9. }
10. }
11. }
```

本章节内容好少，只是想通过一个真实的例子，来提醒大家，做 OpenResty 开发，[lua-nginx-module](#) 的文档是你的首选，Lua 语言的库都是同步阻塞的，用的时候要三思。

再来一个例子来说明阻塞 API 的调用对 Nginx 并发性能的影响

```

1. location /sleep_1 {
2. default_type 'text/plain';
3. content_by_lua_block {
4. ngx.sleep(0.01)
5. ngx.say("ok")
6. }
7. }
8.
9. location /sleep_2 {
10. default_type 'text/plain';
11. content_by_lua_block {
12. function sleep(n)

```

```
13. os.execute("sleep " .. n)
14. end
15. sleep(0.01)
16. ngx.say("ok")
17. }
18. }
```

ab 测试一下

```
1. → nginx git:(master) ab -c 10 -n 20 http://127.0.0.1/sleep_1
2. ...
3. Requests per second: 860.33 [#/sec] (mean)
4. ...
5. → nginx git:(master) ab -c 10 -n 20 http://127.0.0.1/sleep_2
6. ...
7. Requests per second: 56.87 [#/sec] (mean)
8. ...
```

可以看到，如果不使用 ngx\_lua 提供的 sleep 函数，Nginx 并发处理性能会下降 15 倍左右。

## 为什么会这样？

原因是 sleep\_1 接口使用了 OpenResty 提供的非阻塞 API，而 sleep\_2 使用了系统自带的阻塞 API。前者只会引起(进程内)协程的切换，但进程还是处于运行状态(其他协程还在运行)，而后者则会触发进程切换，当前进程会变成睡眠状态，结果 CPU 就进入空闲状态。很明显，非阻塞的 API 的性能会更高。

# 定时任务

- 定时任务

## 定时任务

在[请求返回后继续执行](#)章节中，我们介绍了一种实现的方法，这里我们介绍一种更优雅更通用的方法：`ngx.timer.at()`。

`ngx.timer.at` 会创建一个 Nginx timer。在事件循环中，Nginx 会找出到期的 timer，并在一个独立的协程中执行对应的 Lua 回调函数。

有了这种机制，`ngx_lua` 的功能得到了非常大的扩展，我们有机会做一些更有想象力的功能出来。比如

批量提交和 cron 任务。随便一提，官方的 `resty-cli` 工具，也是基于 `ngx.timer.at` 来运行指定的代码块。

比较典型的用法，如下示例：

```
1. local delay = 5
2. local handler
3. -- do some routine job in Lua just like a cron job
4. handler = function (premature)
5. if premature then
6. return
7. end
8. local ok, err = ngx.timer.at(delay, handler)
9. if not ok then
10. ngx.log(ngx.ERR, "failed to create the timer: ", err)
11. return
12. end
13. end
14.
15. local ok, err = ngx.timer.at(delay, handler)
16. if not ok then
17. ngx.log(ngx.ERR, "failed to create the timer: ", err)
18. return
19. end
```

从示例代码中我们可以看到，`ngx.timer.at` 创建的回调是一次性的。如果要实现“定期”运行，需要在回调函数中重新创建 timer 才行。不过当前主线上的 OpenResty 已经引入了新的

`ngx.timer.every` 接口，允许直接创建定期执行的 timer。

`ngx.timer.at` 的 `delay` 参数，指定的是以秒为单位的延迟触发时间。跟 OpenResty 的其他函数一样，指定的时间最多精确到毫秒。如果你想要的是一个当前阶段结束后立刻执行的回调，可以直

接设置 `delay` 为 0。

handler 回调第一个参数 `premature`，则是用于标识触发该回调的原因是否由于 timer 的到期。Nginx worker 的退出，也会触发当前所有有效的 timer。这时候 `premature` 会被设置为 `true`。回调函数需要正确处理这一参数（通常直接返回即可）。

需要特别注意的是：有一些 `ngx_lua` 的 API 不能在这里调用，比如子请求、`ngx.req.*`和向下游输出的 API(`ngx.print`、`ngx.flush` 之类)，原因是这些调用需要依赖具体的请求。但是

`ngx.timer.at` 自身的运行，与当前的请求并没有关系的。

再说一遍，`ngx.timer.at` 的执行是在独立的协程里完成的。千万不能忽略这一点。有人可能会犯这样的错误：

```
1. local tcpsock = create_tcp_client() -- 创建一个 cosocket 连接
2. local ok, err = ngx.timer.at(delay, function()
3. tcpsock:send() -- bad request!
4. end)
```

`cosocket` 跟某个特定的 `ngx_http_request_t*` 绑定在一起的。虽然由于闭包，在回调函数中我们依旧可以访问 `tcpsock`，但整个上下文已经不一样了。



# 禁止某些终端访问

- [禁止某些终端访问](#)
  - [列举几个限制策略来源](#)

## 禁止某些终端访问

不同的业务应用场景，会有完全不同的非法终端控制策略，常见的限制策略有终端 IP、访问域名端口，这些可以通过防火墙等很多成熟手段完成。可也有一些特定限制策略，例如特定 cookie、url、location，甚至请求 body 包含有特殊内容，这种情况下普通防火墙就比较难限制。

Nginx 是 HTTP 7 层协议的实现者，相对普通防火墙从通讯协议有自己的弱势，同等的配置下的性能表现绝对远不如防火墙，但它的优势胜在价格便宜、调整方便，还可以完成 HTTP 协议上一些更具体的控制策略，与 iptable 的联合使用，让 Nginx 玩出更多花样。

## 列举几个限制策略来源

- IP 地址
- 域名、端口
- Cookie 特定标识
- location
- body 中特定标识

示例配置 (allow、deny)

```
1. location / {
2. deny 192.168.1.1;
3. allow 192.168.1.0/24;
4. allow 10.1.1.0/16;
5. allow 2001:0db8::/32;
6. deny all;
7. }
```

这些规则都是按照顺序解析执行直到某一条匹配成功。在这里示例中，10.1.1.0/16 and

192.168.1.0/24 都是用来限制 IPv4 的，2001  :/32 的配置是用来限制 IPv6。具体有关 allow、deny 配置，请参考[这里](#)。

示例配置 (geo)

```
1. Example:
```

```

2.
3. geo $country {
4. default ZZ;
5. proxy 192.168.100.0/24;
6.
7. 127.0.0.0/24 US;
8. 127.0.0.1/32 RU;
9. 10.1.0.0/16 RU;
10. 192.168.1.0/24 UK;
11. }
12.
13. if ($country == ZZ){
14. return 403;
15. }

```

使用 geo，让我们有更多的分支条件。注意：在 Nginx 的配置中，尽量少用或者不用 if，因为“if is evil”。[点击查看](#)

目前为止所有的控制，都是用 Nginx 模块完成，执行效率、配置明确是它的优点。缺点也比较明显，修改配置代价比较高（reload 服务）。并且无法完成与第三方服务的对接功能交互（例如调用 iptable）。

在 OpenResty 里面，这些问题就都容易解决，还记得 `access_by_lua*` 么？推荐一个第三方库 [lua-resty-iputils](#)。

示例代码：

```

1. init_by_lua_block {
2. local iputils = require("resty.iputils")
3. iputils.enable_lrucache()
4. local whitelist_ips = {
5. "127.0.0.1",
6. "10.10.10.0/24",
7. "192.168.0.0/16",
8. }
9.
10. -- WARNING: Global variable, recommend this is cached at the module level
11. -- https://github.com/openresty/lua-nginx-module#data-sharing-within-an-nginx-worker
12. whitelist = iputils.parse_cidrs(whitelist_ips)
13. }
14.
15. access_by_lua_block {
16. local iputils = require("resty.iputils")
17. if not iputils.ip_in_cidrs(ngx.var.remote_addr, whitelist) then
18. return ngx.exit(ngx.HTTP_FORBIDDEN)
19. end
20. }

```

以次类推，我们想要完成域名、Cookie、location、特定 body 的准入控制，甚至可以做到与本地 iptable 防火墙联动。

我们可以把 IP 规则存到数据库中，这样我们就再也不用 reload Nginx，在有规则变动的时候，刷新下 Nginx 的缓存就行了。

思路打开，大家后面多尝试各种玩法吧。

## 请求返回后继续执行

- [请求返回后继续执行](#)

## 请求返回后继续执行

在一些请求中，我们会做一些日志的推送、用户数据的统计等和返回给终端数据无关的操作。而这些操作，即使你用异步非阻塞的方式，在终端看来，也是会影响速度的。这个和我们的原则：终端请求，需要用最快的速度返回给终端，是冲突的。

这时候，最理想的是，获取完给终端返回的数据后，就断开连接，后面的日志和统计等动作，在断开连接后，后台继续完成即可。

怎么做到呢？我们先看其中的一种方法：

```
1. local response, user_stat = logic_func.get_response(request)
2. ngx.say(response)
3. ngx.eof()
4.
5. if user_stat then
6. local ret = db_redis.update_user_data(user_stat)
7. end
```

没错，最关键的一行代码就是`ngx.eof()`，它可以即时关闭连接，把数据返回给终端，后面的数据库操作还会运行。比如上面代码中的

```
1. local response, user_stat = logic_func.get_response(request)
```

运行了 0.1 秒，而

```
1. db_redis.update_user_data(user_stat)
```

运行了 0.2 秒，在没有使用 `ngx.eof()` 之前，终端感知到的是 0.3 秒，而加上 `ngx.eof()` 之后，终端感知到的只有 0.1 秒。

需要注意的是，你不能任性的把阻塞的操作加入代码，即使在 `ngx.eof()` 之后。虽然已经返回了终端的请求，但是，Nginx 的 worker 还在被你占用。所以在 keep alive 的情况下，本次请求的总时间，会把上一次 `eof()` 之后的时间加上。

如果你加入了阻塞的代码，Nginx 的高并发就是空谈。

有没有其他的方法来解决这个问题呢？我们会在[ngx.timer.at](#)里面给大家介绍更优雅的方案。

请求返回后继续执行

# 调试

- 调试
  - 关闭 code cache
  - 记录日志

# 调试

调试是一个程序猿非常重要的能力，人写的程序总会有 bug，所以需要 debug。如何方便和快速的定位 **bug**，是我们讨论的重点，只要 bug 能定位，解决就不是问题。

对于熟悉用 Visual Studio 和 Eclipse 这些强大的集成开发环境的来做 C++ 和 Java 的同学来说，OpenResty 的 debug 要原始很多，但是对于习惯 Python 开发的同学来说，又是那么的熟悉。

张银奎有本《[软件调试](#)》的书，windows 客户端程序猿应该都看过，大家可以去试读下，看看里面有多复杂：(

对于 OpenResty，坏消息是，没有单步调试这些玩意儿（我们尝试搞出来过 ngx Lua 的单步调试，但是没人用...）；好消息是，它像 Python 一样，非常简单，不用复杂的技术，只靠 print 和 log 就能定位绝大部分问题，难题有[火焰图](#)这个神器。

## • 关闭 code cache

在调试的时候最好关闭 `lua_code_cache` 这个选项。

```
1. lua_code_cache off;
```

关闭 `lua_code_cache` 之后，OpenResty 会给每个请求创建新的 Lua VM。由于没有 Lua module 的缓存，新的 VM 会去加载刚最新的 Lua 文件。

这样，你修改完代码后，不用 reload Nginx 就可以生效了。在生产环境下记得打开这个选项。

当然，由于每个请求运行在独立的 Lua VM 里，`lua_code_cache off` 会带来以下几个问题：

1. 每个请求都会有独立的 module，独立的 lru cache，独立的 timer，独立的线程池。
2. 跟请求无关的模块，由于不会被新的请求加载，并不会主动更新。比如 `init_by_lua_file` 引用的文件就不会被更新。
3. `*_by_lua_block` 里面的代码，由于不在 Lua 文件里面，设置 `lua_code_cache` 对其没有意义。

如果调试的目标涉及以上内容，仍需设置 `lua_code_cache on`，通过 reload 来更新代码。

## • 记录日志

这个看上去谁都会的东西，要想做好也不容易。

你有遇到这样的情况吗？QA 发现了一个 bug，开发说我修改代码加个日志看看，然后 QA 重现这个问题，发现日志不够详细，需要再加，反复几次，然后再给 QA 一个没有日志的版本，继续测试其他功能。

如果产品已经发布到用户那里了呢？如果用户那里是隔离网，不能远程怎么办？

你在写代码的时候，就需要考虑到调试日志。 比如这个代码：

```
1. local response, err = redis_op.finish_client_task(client_mid, task_id)
2. if response then
3. put_job(client_mid, result)
4. ngx.log(ngx.WARN, "put job:", common.json_encode({channel="task_status", mid=client_mid,
 data=result}))
5. end
```

我们在做一个操作后，就把结果记录到 Nginx 的 error.log 里面，等级是 warn。在生产环境下，日志等级默认为 error，在我们需要详细日志的时候，把等级调整为 warn 即可。在我们的实际使用中，我们会把一些很少发生的重要事件，做为 error 级别记录下来，即使它并不是 Nginx 的错误。

与日志配套的，你需要[logrotate](#)来做日志的切分和备份。

## 请求中断后的处理

- [请求中断后的处理](#)

## 请求中断后的处理

---



# 我的 lua 代码需要调优么

- 网上有大量对 Lua 调优的推荐，我们应该如何看待？

## 网上有大量对 Lua 调优的推荐，我们应该如何看待？

Lua 的解析器有官方的 standard Lua 和 LuaJIT，需要明确一点的是目前大量的优化文章都比较陈旧，而且都是针对 standard Lua 解析器的，standard Lua 解析器在性能上需要书写者自己规避，才能写出高性能来。需要各位看官注意的是，OpenResty 最新版默认已经绑定 LuaJIT，优化手段和方法已经略有不同。我们现在的做法是：代码易读是首位，目前还没有碰到同样代码换个写法就有质的提升，如果我们对某个单点功能有性能要求，那么建议用 LuaJIT 的 FFI 方法直接调用 C 接口更直接一点。

代码出处：<http://www.cnblogs.com/lovevivi/p/3284643.html>

```
1. 3.0 避免使用table.insert()
2.
3. 下面来看看4个实现表插入的方法。在4个方法之中table.insert()在效率上不如其他方法，是应该避免使用的。
4. 使用table.insert
5. local a = {}
6. local table_insert = table.insert
7. for i = 1,100 do
8. table_insert(a, i)
9. end
10.
11. 使用循环的计数
12.
13. local a = {}
14. for i = 1,100 do
15. a[i] = i
16. end
17. 使用table的size
18.
19. local a = {}
20. for i = 1,100 do
21. a[#a+1] = i
22. end
23. 使用计数器
24.
25. local a = {}
26. local index = 1
27. for i = 1,100 do
28. a[index] = i
```

```
29. index = index+1
30. end
31.
32. 4.0 减少使用 unpack()函数
33. Lua的unpack()函数不是一个效率很高的函数。你完全可以写一个循环来代替它的作用。
34.
35. 使用unpack()
36.
37. local a = { 100, 200, 300, 400 }
38. for i = 1,100 do
39. print(unpack(a))
40. end
41. 代替方法
42.
43. local a = { 100, 200, 300, 400 }
44. for i = 1,100 do
45. print(a[1],a[2],a[3],a[4])
46. end
```

针对这篇文章内容写了一些测试代码：

```
1. local start = os.clock()
2.
3. local function sum(...)
4. local args = {...}
5. local a = 0
6. for k,v in pairs(args) do
7. a = a + v
8. end
9. return a
10. end
11.
12. local function test_unit()
13. -- t1: 0.340182 s
14. -- local a = {}
15. -- for i = 1,1000 do
16. -- table.insert(a, i)
17. -- end
18.
19. -- t2: 0.332668 s
20. -- local a = {}
21. -- for i = 1,1000 do
22. -- a[#a+1] = i
23. -- end
24.
25. -- t3: 0.054166 s
26. -- local a = {}
```

```

27. -- local index = 1
28. -- for i = 1,1000 do
29. -- a[index] = i
30. -- index = index+1
31. -- end
32.
33. -- p1: 0.708012 s
34. -- local a = 0
35. -- for i=1,1000 do
36. -- local t = { 1, 2, 3, 4 }
37. -- for i,v in ipairs(t) do
38. -- a = a + v
39. -- end
40. -- end
41.
42. -- p2: 0.660426 s
43. -- local a = 0
44. -- for i=1,1000 do
45. -- local t = { 1, 2, 3, 4 }
46. -- for i = 1,#t do
47. -- a = a + t[i]
48. -- end
49. -- end
50.
51. -- u1: 2.121722 s
52. -- local a = { 100, 200, 300, 400 }
53. -- local b = 1
54. -- for i = 1,1000 do
55. -- b = sum(unpack(a))
56. -- end
57.
58. -- u2: 1.701365 s
59. -- local a = { 100, 200, 300, 400 }
60. -- local b = 1
61. -- for i = 1,1000 do
62. -- b = sum(a[1], a[2], a[3], a[4])
63. -- end
64.
65. return b
66. end
67.
68. for i=1,10 do
69. for j=1,1000 do
70. test_unit()
71. end
72. end
73.
74. print(os.clock()-start)

```

从运行结果来看，除了 t3 有本质上的性能提升（六倍性能差距，但是 t3 写法相当丑陋），其他不同的写法都在一个数量级上。你是愿意让代码更易懂还是更牛逼，就看各位看官自己的抉择了。不要盲信，也不要不信，各位要睁开眼自己多做测试。

另外说明：文章提及的使用局部变量、缓存 table 元素，在 LuaJIT 中还是很有用的。

## 变量的共享范围

- 变量的共享范围
  - 问题在哪里？
  - OpenResty 中 Lua 变量的范围
    - 全局变量
    - 模块变量
    - 本地变量

## 变量的共享范围

本章内容来自openresty讨论组 [这里](#)

先看两段代码：

```
1. -- index.lua
2. local uri_args = ngx.req.get_uri_args()
3. local mo = require('mo')
4. mo.args = uri_args
```

```
1. -- mo.lua
2.
3. local showJs = function(callback, data)
4. local cJSON = require('cjson')
5. ngx.say(callback .. '(' .. cJSON.encode(data) .. ')')
6. end
7. local self.jsonp = self.args.jsonp
8. local keyList = string.split(self.args.key_list, ',')
9. for i=1, #keyList do
10. -- do something
11. ngx.say(self.args.kind)
12. end
13. showJs(self.jsonp, valList)
```

大概代码逻辑如上，然后出现这种情况：

生产服务器中，如果没有用户访问，自己几个人测试，一切正常。

同样生产服务器，我将大量的用户请求接入后，我不停刷新页面的时候会出现部分情况（概率也不低，几分之一，大于 10%），输出的 callback（也就是来源于 self.jsonp，即 URL 参数中的 jsonp 变量）和 url 地址中不一致（我自己测试的值是 `?jsonp=jsonp1435220570933`，而用户的请求基本上都是 `?jsonp=jquery....`）。错误的情况都是会出现用户请求才会有的 `jquery....` 这种字符串。另外 URL 参数中的 kind 是 1，我在循环中输出会有“1”或“nil”的情况。不仅这两

种参数，几乎所有 url 中传递的参数，都有可能变成其他请求链接中的参数。

基于以上情况，个人判断会不会是在生产服务器大量用户请求中，不同请求参数串掉了，但是如果这样，是否应该会出现我本次的获取参数是某个其他用户的值，那么 for 循环中的值也应该固定的，而不会是一会儿是我自己请求中的参数值，一会儿是其他用户请求中的参数值。

## 问题在哪里？

Lua module 是 VM 级别共享的，见[这里](#)。

`mo.args` 变量一不留神全局共享了，而这肯定不是作者期望的。所以导致了高并发应用场景下偶尔出现异常错误的情况。

每个请求的数据在传递和存储时须特别小心，只应通过你自己的函数参数来传递，或者通过 `ngx.ctx` 表。前者效率显然较高，而后者胜在能跨阶段使用。

贴一个 `ngx.ctx` 的例子：

```
1. location /test {
2. rewrite_by_lua_block {
3. ngx.ctx.foo = 76
4. }
5. access_by_lua_block {
6. ngx.ctx.foo = ngx.ctx.foo + 3
7. }
8. content_by_lua_block {
9. ngx.say(ngx.ctx.foo)
10. }
11. }
```

## OpenResty 中 Lua 变量的范围

### 全局变量

在 OpenResty 里面，只有在 `init_by_lua*` 和 `init_worker_by_lua*` 阶段才能定义真正的全局变量。

这是因为其他阶段里面，OpenResty 会设置一个隔离的全局变量表，以免在处理过程污染了其他请求。

即使在上述两个可以定义全局变量的阶段，也尽量避免这么做。全局变量能解决的问题，用模块变量也能解决，而且会更清晰、更干净。

### 模块变量

这里把定义在模块里面的变量称为模块变量。无论定义变量时有没有加 `local`，有没有通过 `_M`

把变量引用起来，  
定义在模块里面的变量都是模块变量。

由于 Lua VM 会把 require 进来的模块缓存到 `package.loaded` 表里，除非设置了

`lua_code_cache off`，

模块里定义的变量都会被缓存起来。而且重要的是，模块变量在每个请求中是共享的。

模块变量的跨请求特性，可以有很多用途。比如在变量间共享值，或者在 `init_worker_by_lua*` 中初始化全局用到的数值。

作为硬币的反面，无视这一特性也会带来许多问题。下面让我们看看一个例子。

nginx.conf

```
1. location = /index {
2. content_by_lua_file conf/lua/web/index.lua;
3. }
```

index.lua

```
1. local var = require "var"
2.
3.
4. if var.calc() == 1 then
5. ngx.say("ok")
6. else
7. ngx.status = ngx.HTTP_INTERNAL_SERVER_ERROR
8. ngx.say("error")
9. end
```

var.lua

```
1. local count = 1
2.
3. local _M = {}
4.
5. local function add()
6. count = count + 1
7. end
8.
9. local function sub()
10. count = count - 1
11. end
12.
13. function _M.calc()
14. add()
15. -- 模拟协程调度
```

```

16. ngx.sleep(ngx.time()%0.003)
17. sub()
18. return count
19. end
20.
21. return _M

```

分别用单个客户端和两个客户端请求之：

```

1. ~/test/openresty wrk --timeout 10s -t 1 -c 1 -d 1s http://localhost:6699/index
2. Running 1s test @ http://localhost:6699
3. 1 threads and 1 connections
4. Thread Stats Avg Stdev Max +/- Stdev
5. Latency 1.22ms 291.09us 3.48ms 77.30%
6. Req/Sec 822.64 175.27 1.01k 54.55%
7. 900 requests in 1.10s, 153.76KB read
8. ~/test/openresty wrk --timeout 10s -t 2 -c 2 -d 1s http://localhost:6699/index
9. Running 1s test @ http://localhost:6699
10. 2 threads and 2 connections
11. Thread Stats Avg Stdev Max +/- Stdev
12. Latency 1.18ms 387.03us 7.92ms 85.98%
13. Req/Sec 0.86k 168.12 1.02k 60.00%
14. 1709 requests in 1.00s, 310.29KB read
15. Non-2xx or 3xx responses: 852

```

对于那些返回 500 状态码的请求，其调度过程大概是这样的：

```

1. coroutine A | coroutine B | count
2. add | | 2
3. sleep | | 2
4. | add | 3
5. | sleep | 3
6. sub | | 2
7.
8. (2 != 1) => HTTP_INTERNAL_SERVER_ERROR!

```

同样道理，如果在模块级别共享 TCP/UDP Client，比如在模块开头 `local httpc = http.new()`，高并发下难免

会有奇怪的问题。把 `httpc:send` 看作 `add`；`httpc:receive` 看作 `sub`，几乎就是上述的例子。

运气好的话，你可能只会碰到一个 `bad request` 的异常；运气不好，就是一个潜在的坑。

本地变量

跟全局变量、模块变量相对，这里我们姑且把 `*_by_lua*` 里面定义的变量称之为本地变量。



本地变量仅在当前阶段有效，如果要跨阶段使用，需要借助 `ngx.ctx` 或者附加在模块变量里。

值得注意的是 `ngx.timer.*`。虽然 `timer` 代码占的是别的上下文的位置，但是每个 `timer` 都是运行在自己的协程里面，里面定义的变量都是协程内部的。

举个例子，让我们在 `init_worker_by_lua_block` 里面定义一个 `timer`。

```
1. init_worker_by_lua_block {
2. local delay = 5
3. local handler
4. handler = function()
5. counter = counter or 0
6. counter = counter + 1
7. ngx.log(ngx.ERR, counter)
8. local ok, err = ngx.timer.at(delay, handler)
9. if not ok then
10. ngx.log(ngx.ERR, "failed to create the timer: ", err)
11. return
12. end
13. end
14. local ok, err = ngx.timer.at(delay, handler)
15. if not ok then
16. ngx.log(ngx.ERR, "failed to create the timer: ", err)
17. return
18. end
19. }
```

`counter` 变量初看是定义在 `init_worker_by_lua*` 的全局变量。定义在 `init_worker_by_lua*` 阶段，没有 `local` 修饰，根据前面的讨论，它肯定是个全局变量嘛。

运行一下，你会发现，每次 `counter` 的输出都是 1。

其实 `counter` 实际的定义域是 `handler` 这个函数内部。由于每个 `timer` 都运行在独立的协程里，`timer` 的每次触发，都会重新把 `counter` 定义一遍。如果要在 `timer` 的每次触发中共享变量，你有两个选择：

1. 通过函数参数，把每个变量都传递一遍。
2. 把要共享的变量当作模块变量。

（当然也可以选择 `init_worker_by_lua*` 里面、`ngx.timer.*` 外面定义真正的全局变量，不过不太推荐罢了）



## 动态限速

- [动态限速](#)
  - [限制响应速度](#)
  - [限制连接数和请求数](#)

## 动态限速

在应用开发中，经常会有对请求进行限速的需求。

通常意义上的限速，其实可以分为以下三种：

1. `limit_rate` 限制响应速度
2. `limit_conn` 限制连接数
3. `limit_req` 限制请求数

接下来让我们看看，这三种限速在 OpenResty 中分别怎么实现。

## 限制响应速度

Nginx 有一个 `$limit_rate`，这个变量反映的是当前请求每秒能响应的字节数。该字节数默认为配置文件中 `limit_rate` 指令的设置。

一如既往，通过 OpenResty，我们可以直接在 Lua 代码中动态设置它。

```
1. access_by_lua_block {
2. -- 设定当前请求的响应上限是 每秒 300K 字节
3. ngx.var.limit_rate = "300K"
4. }
```

## 限制连接数和请求数

对于连接数和请求数的限制，我们可以求助于 OpenResty 官方的 [lua-resty-limit-traffic](#)。需要注意的是，`lua-resty-limit-traffic` 要求 OpenResty 版本在 `1.11.2.2` 以上（对应的 `lua-nginx-module` 版本是 `0.10.6`）。

如果要配套更低版本的 OpenResty 使用，需要修改源码。比如把代码中涉及 `incr(key, value, init)` 方法，改成 `incr(key, value)` 和 `set(key, init)` 两步操作。这么改会增大有潜在 race condition 的区间。

`lua-resty-limit-traffic` 这个库是作用于所有 Nginx worker 的。由于数据同步上的局限，在限制请求数的过程中 `lua-resty-limit-traffic` 有一个 race condition 的区间，可能多放过几个请求。误差大小取决于 Nginx worker 数量。

如果要求“宁可拖慢一千，不可放过一个”的精确度，恐怕就不能用这个库了。你可能需要使用 `lua-resty-lock` 或外部的锁服务，只是性能上的代价会更高。

`lua-resty-limit-traffic` 的限速实现基于[漏桶原理](#)。

通俗地说，就是小学数学中，蓄水池一边注水一边放水的问题。

这里注水的速度是新增请求/连接的速度，而放水的速度则是配置的限制速度。

当注水速度快于放水速度（表现为池中出現蓄水），则返回一个数值 `delay`。调用者通过

`ngx.sleep(delay)` 来减慢注水的速度。

当蓄水池满时（表现为当前请求/连接数超过设置的 `burst` 值），则返回错误信息 `rejected`。调用者需要丢掉溢出来的这部份。

下面是限制连接数的示例：

```
1. # nginx.conf
2. lua_code_cache on;
3. # 注意 limit_conn_store 的大小需要足够放置限流所需的键值。
4. # 每个 $binary_remote_addr 大小不会超过 16K，算上 lua_shared_dict 的节点大小，总共不到 64 字节。
5. # 100M 可以放 1.6M 个键值对
6. lua_shared_dict limit_conn_store 100M;
7. server {
8. listen 8080;
9. location / {
10. access_by_lua_file src/access.lua;
11. content_by_lua_file src/content.lua;
12. log_by_lua_file src/log.lua;
13. }
14. }
```

```
1. -- utils/limit_conn.lua
2. local limit_conn = require "resty.limit.conn"
3.
4. -- new 的第四个参数用于估算每个请求会维持多长时间，以便于应用漏桶算法
5. local limit, limit_err = limit_conn.new("limit_conn_store", 10, 2, 0.05)
6. if not limit then
7. error("failed to instantiate a resty.limit.conn object: ", limit_err)
8. end
9.
10. local _M = {}
11.
12. function _M.incoming()
13. local key = ngx.var.binary_remote_addr
14. local delay, err = limit.incoming(key, true)
15. if not delay then
16. if err == "rejected" then
17. return ngx.exit(503)
18. end
19. end
20. }
```

```

19. ngx.log(ngx.ERR, "failed to limit req: ", err)
20. return ngx.exit(500)
21. end
22.
23. if limit:is_committed() then
24. local ctx = ngx.ctx
25. ctx.limit_conn_key = key
26. ctx.limit_conn_delay = delay
27. end
28.
29. if delay >= 0.001 then
30. ngx.log(ngx.WARN, "delaying conn, excess ", delay,
31. "s per binary_remote_addr by limit_conn_store")
32. ngx.sleep(delay)
33. end
34. end
35.
36. function _M.leaving()
37. local ctx = ngx.ctx
38. local key = ctx.limit_conn_key
39. if key then
40. local latency = tonumber(ngx.var.request_time) - ctx.limit_conn_delay
41. local conn, err = limit:leaving(key, latency)
42. if not conn then
43. ngx.log(ngx.ERR,
44. "failed to record the connection leaving ",
45. "request: ", err)
46. end
47. end
48. end
49.
50. return _M

```

```

1. -- src/access.lua
2. local limit_conn = require "utils.limit_conn"
3.
4.
5. -- 对于内部重定向或子请求，不进行限制。因为这些并不是真正对外的请求。
6. if ngx.req.is_internal() then
7. return
8. end
9. limit_conn.incoming()

```

```

1. -- src/log.lua
2. local limit_conn = require "utils.limit_conn"
3.

```

```
4.
5. limit_conn.leaving()
```

注意在限制连接的代码里面，我们用 `ngx.ctx` 来存储 `limit_conn_key`。这里有一个坑。内部重定向（比如调用了 `ngx.exec`）会销毁 `ngx.ctx`，导致 `limit_conn.leaving()` 无法正确调用。如果需要限连业务里有用到 `ngx.exec`，可以考虑改用 `ngx.var` 而不是 `ngx.ctx`，或者另外设计一套存储方式。只要能保证请求结束时能及时调用 `limit:leaving()` 即可。

限制请求数的实现差不多，这里就不赘述了。

# shared.dict 非队列性质

- ngx.shared.DICT 非队列性质
  - 非队列性质
  - 问题解决

ngx.shared.DICT

## 非队列性质

=====

执行阶段和主要函数请参考 [HttpLuaModule#ngx.shared.DICT](http://lua-module#ngx.shared.DICT)

## 非队列性质

ngx.shared.DICT 的实现是采用红黑树实现，当申请的缓存被占用完后如果有新数据需要存储则采用 LRU 算法淘汰掉“多余”数据。

这样数据结构的在带有队列性质的业务逻辑下会出现的一些问题：

我们用 shared 作为缓存，接纳终端输入并存储，然后在另外一个线程中按照固定的速度去处理这些输入，代码如下：

```

1.
2. -- [ngx.thread.spawn](https://github.com/openresty/lua-nginx-module#ngxthreadspawn) #1 存储线程 理
 解为生产者
3.
4.
5. local cache_str = string.format("[%s&%s&%s&%s&%s&%s&%s]", net, name, ip,
6. mac, ngx.var.remote_addr, method, md5)
7. local ok, err = ngx_nf_data:safe_set(mac, cache_str, 60*60) --这些是缓存数据
8. if not ok then
9. ngx.log(ngx.ERR, "stored nf report data error: "..err)
10. end
11.
12.
13.
14. -- [ngx.thread.spawn](https://github.com/openresty/lua-nginx-module#ngxthreadspawn) #2 取线程 理解
 为消费者
15.
16. while not ngx.worker.exiting() do
17. local keys = ngx_share:get_keys(50) -- 一秒处理50个数据
18.
19. for index, key in pairs(keys) do
20. str = ((nil ~= str) and str..[#]..ngx_share:get(key)) or ngx_share:get(key)

```

```
21. ngx_share:delete(key) --干掉这个key
22. end
23. --一些消费过程，看官不要在意
24. ngx.sleep(1)
25. end
```

在上述业务逻辑下会出现由生产者生产的某些 key-val 对永远不会被消费者取出并消费，原因就是 `shared.DICT` 不是队列，`ngx_shared:get_keys(n)` 函数不能保证返回的 `n` 个键值对是满足 FIFO 规则的，从而导致问题发生。

## 问题解决

问题的原因已经找到，解决方案有如下几种：

1. 修改暂存机制，采用 Redis 的队列来做暂存；
2. 调整消费者的消费速度，使其远远大于生产者的速度；
3. 修改 `ngx_shared:get_keys()` 的使用方法，即是不带参数；

方法 3 和 2 本质上都是一样的，由于业务已经上线，方法 1 周期太长，于是采用方法 2 解决，在后续的业务中不再使用 `shared.DICT` 来暂存队列性质的数据



# 正确使用长链接

- [KeepAlive](#)

## KeepAlive

在 OpenResty 中，连接池在使用上如果不加以注意，容易产生数据写错地方，或者得到的应答数据异常以及类似的问题，当然使用短连接可以规避这样的问题，但是在一些企业用户环境下，短连接 + 高并发对企业内部的防火墙是一个巨大的考验，因此，长连接自有其用武之地，使用它的时候要记住，长连接一定要保持其连接池中所有连接的正确性。

```

1. -- 错误的代码
2. local function send()
3. for i = 1, count do
4. local ssdb_db, err = ssdb:new()
5. local ok, err = ssdb_db:connect(SSDB_HOST, SSDB_PORT)
6. if not ok then
7. ngx.log(ngx.ERR, "create new ssdb failed!")
8. else
9. local key, err = ssdb_db:qpop(something)
10. if not key then
11. ngx.log(ngx.ERR, "ssdb qpop err:", err)
12. else
13. local data, err = ssdb_db:get(key[1])
14. -- other operations
15. end
16. end
17. end
18. ssdb_db:set_keepalive(SSDB_KEEP_TIMEOUT, SSDB_KEEP_COUNT)
19. end
20. -- 调用
21. while true do
22. local ths = {}
23. for i=1, THREADS do
24. ths[i] = ngx.thread.spawn(send) ----创建线程
25. end
26. for i = 1, #ths do
27. ngx.thread.wait(ths[i]) ----等待线程执行
28. end
29. ngx.sleep(0.020)
30. end

```

以上代码在测试中发现，应该得到 `get(key)` 的返回值有一定几率为 `key`。

原因即是在 `ssdb` 创建连接时可能会失败，但是当得到失败的结果后依然调用

```
ssdb_db:set_keepalive
```

 将此连接并入连接池中。

正确地做法是如果连接池出现错误，则不要将该连接加入连接池。

```

1. local function send()
2. for i = 1, count do
3. local ssdb_db, err = ssdb:new()
4. local ok, err = ssdb_db:connect(SSDB_HOST, SSDB_PORT)
5. if not ok then
6. ngx.log(ngx.ERR, "create new ssdb failed!")
7. return
8. else
9. local key, err = ssdb_db:qpop(something)
10. if not key then
11. ngx.log(ngx.ERR, "ssdb qpop err:", err)
12. else
13. local data, err = ssdb_db:get(key[1])
14. -- other operations
15. end
16. ssdb_db:set_keepalive(SSDB_KEEP_TIMEOUT, SSDB_KEEP_COUNT)
17. end
18. end
19. end

```

所以，当你使用长连接操作 `db` 出现结果错乱现象时，首先应该检查下是否存在长连接使用不当的情况。

## 如何引用第三方 resty 库

- [如何引用第三方 resty 库](#)

## 如何引用第三方 resty 库

OpenResty 引用第三方 resty 库非常简单，只需要将相应的文件拷贝到 resty 目录下即可。

我们以 `resty.http` ( [pint-sized/lua-resty-http](#) ) 库为例。

只要将 `lua-resty-http/lib/resty/` 目录下的 `http.lua` 和 `http_headers.lua` 两个文件拷贝到 `/usr/local/openresty/lualib/resty` 目录下即可(假设你的 OpenResty 安装目录为 `/usr/local/openresty` )。

验证代码如下：

```
1.
2. server {
3.
4. listen 8080 default_server;
5. server_name _;
6.
7. resolver 8.8.8.8;
8.
9. location /baidu {
10. content_by_lua_block {
11. local http = require "resty.http"
12. local httpc = http.new()
13. local res, err = httpc:request_uri("http://www.baidu.com")
14. if res.status == ngx.HTTP_OK then
15. ngx.say(res.body)
16. else
17. ngx.exit(res.status)
18. end
19. }
20. }
21. }
```

访问 <http://127.0.0.1:8080/baidu> , 如果出现的是百度的首页, 说明你配置成功了。

当然这里也可以自定义 `lua_package_path` 指定 Lua 的查找路径, 这样就可以把第三方代码放到相应的位置下, 这么做更加方便归类文件, 明确什么类型的文件放到什么地方(比如: 公共文件、业务文件)。



# 典型应用场景

- 典型应用场景
  - 不擅长的应用场景

# 典型应用场景

可以这样说，任何一个开发语言、开发框架，都有它存在的明确目的，重心是为了解决什么问题。没有说我们学习一门语言或技术，就可以解决所有的问题。同样的，`OpenResty` 的存在也有其自身适用的应用场景。

其实官网 wiki 已经列了出来：

- 在 Lua 中混合处理不同 Nginx 模块输出 (proxy, drizzle, postgres, Redis, memcached 等)。
- 在请求真正到达上游服务之前，Lua 中处理复杂的准入控制和安全检查。
- 比较随意的控制应答头 (通过 Lua)。
- 从外部存储中获取后端信息，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，同步编写异步访问后端数据库和其他存储。
- 在 rewrite 阶段，通过 Lua 完成非常复杂的处理。
- 在 Nginx 子查询、location 调用中，通过 Lua 实现高级缓存机制。
- 对外暴露强劲的 Lua 语言，允许使用各种 Nginx 模块，自由拼合没有任何限制。该模块的脚本有充分的灵活性，同时提供的性能水平与本地 C 语言程序无论是在 CPU 时间方面以及内存占用差距非常小。所有这些都要求 LuaJIT 2.x 是启用的。其他脚本语言实现通常很难满足这一性能水平。

## 不擅长的应用场景

前面的章节，我们是从它适合的场景出发，`OpenResty` 不适合的场景又有哪些？以及我们在使用中如何规避这些问题呢？

这里官网并没有给出答案，我根据我们的应用场景给大家列举，并简单描述一下原因：

- 有长时间阻塞调用的过程
  - 例如通过 `Lua` 完成系统命令行调用
  - 使用阻塞的 `Lua API` 完成相应操作
- 单个请求处理逻辑复杂，尤其是需要和请求方多次交互的长连接场景
  - `Nginx` 的内存池 pool 是每次新申请内存存放数据
  - 所有的内存释放都是在请求退出的时候统一释放
  - 如果单个请求处理过于复杂，将会有过多内存无法及时释放
- 内存占用高的处理
  - 受制于 `Lua VM` 的最大使用内存 1G 的限制

- 这个限制是单个 `Lua VM`，也就是单个 `Nginx worker`
- 两个请求之间有交流的场景
  - 例如你做个在线聊天，要完成两个用户之间信息的传递
  - 当前 `OpenResty` 还不具备这个通讯能力（后面可能会有所完善）
- 与行业专用的组件对接
  - 最好是 TCP 协议对接，不要是 API 方式对接，防止里面有阻塞 TCP 处理
  - 由于 `OpenResty` 必须要使用非阻塞 API，所以传统的阻塞 API，我们是没法直接使用的
  - 获取 TCP 协议，使用 `cosocket` 重写（重写后的效率还是很赞的）
- 每请求开启的 `light thread` 过多的场景
  - 虽然已经是 `light thread`，但它对系统资源的占用相对是比较大的

这些适合、不适合信息可能在后面随着 `OpenResty` 的发展都会有新的变化，大家拭目以待。

# 怎样理解 cosocket

- 怎样理解 cosocket

## 怎样理解 cosocket

笔者认为，cosocket 是 OpenResty 世界中技术、实用价值最高部分。让我们可以用非常低廉的成本，优雅的姿势，比传统 socket 编程效率高好几倍的方式进行网络编程。无论资源占用、执行效率、并发能力都非常出色。

鲁迅有句名言“其实世界上本没有路，走的人多了便有了路”，其实对于 cosocket 的中文翻译貌似我也碰到了类似的问题。当我想给大家一个正面解释，爬过了官方 wiki 发现，原来作者本人（章亦春）也没有先给出 cosocket 定义。

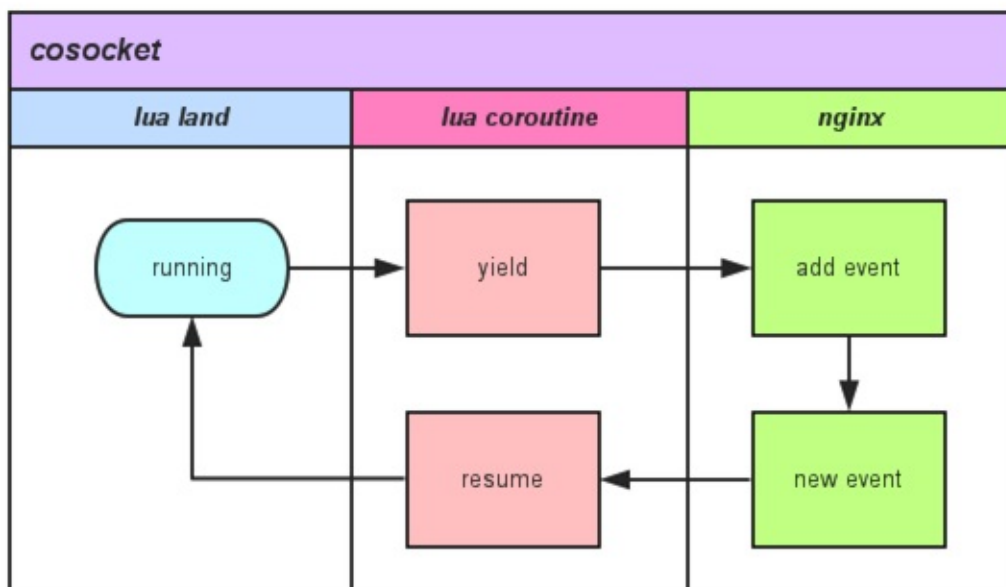
看来只能通过一些侧面信息，从而让这条路逐渐的清晰起来。

```
cosocket = coroutine + socket
```

coroutine: 协同程序（后面简称：协程）

socket: 网络套接字

OpenResty 中的 cosocket 不仅需要协程特性支撑，它还需 Nginx 非常最重要的“事件循环回调机制”，两部分结合在一起最终达到了 cosocket 效果，外加 Nginx 自身对各种资源的“小气”，LuaJIT 的执行效率，最终加分不少。在 Lua 世界中调用任何一个有关 cosocket 网络函数内部关键调用如图所示：



从该图中我们可以看到，用户的 Lua 脚本每触发一个网络操作，都会有协程的 yield 以及

resume，因为请求的 Lua 脚本实际上都运行在独享协程之上，可以在任何需要的时候暂停自己（yield），也可以在任何需要的时候被唤醒（resume）。

暂停自己，把网络事件注册到 Nginx 监听列表中，并把运行权限交给 Nginx。当有 Nginx 注册网络事件达到触发条件时，唤醒对应的协程继续处理。

以此为蓝图，封装实现 connect、read、receive 等操作，形成了大家目前所看到的 cosocket API。

可以看到，cosocket 是依赖 Lua 协程 + Nginx 事件通知两个重要特性拼的。

从 0.9.9 版本开始，cosocket 对象是全双工的，也就是说，一个专门读取的 “light thread”，一个专门写入的 “light thread”，它们可以同时针对同一个 cosocket 对象进行操作（两个 “light threads” 必须运行在同一个 Lua 环境中，原因见上）。但是你不能让两个 “light threads” 对同一个 cosocket 对象都进行读（或者写入、或者连接）操作，否则当调用 cosocket 对象时，你将得到一个类似 “socket busy reading” 的错误。

所以东西总结下来，到底什么是 cosocket，中文应该怎么翻译，笔者本人都开始纠结了。我们不妨从另外一个角度来审视它，它到底给我们带来了什么。

- 它是同步的；
- 它是非阻塞的；
- 它是全双工的；

同步与异步解释：

同步：做完一件事再去做另一件；

异步：同时做多件事情，某个事情有结果了再去处理。

阻塞与非阻塞解释：

阻塞：不等到想要的结果我就不走了；

非阻塞：有结果我就带走，没结果我就空手而回，总之一句话：爷等不起。

异步 / 同步是做事派发方式，阻塞 / 非阻塞是如何处理事情，两组概念不在同一个层面。

无论 ngx.socket.tcp()、ngx.socket.udp()、ngx.socket.stream()、ngx.req.socket()，它们基本流程都是一样的，只是一些细节参数上有区别（比如 TCP 和 UDP 的区别）。下面这些函数，都是用来辅助完成更高级的 socket 行为控制：

- connect
- sslhandshake
- send
- receive
- close
- settimeout
- setoption
- receiveuntil
- setkeepalive



- `getreusedtimes`

它们不仅完整兼容 `LuaSocket` 库的 `TCP API`，而且还是 100% 非阻塞的。

这里给大家 show 一个例子，对 `cosocket` 使用有一个整体认识。

```

1. location /test {
2. resolver 114.114.114.114;
3.
4. content_by_lua_block {
5. local sock = ngx.socket.tcp()
6. local ok, err = sock:connect("www.baidu.com", 80)
7. if not ok then
8. ngx.say("failed to connect to baidu: ", err)
9. return
10. end
11.
12. local req_data = "GET / HTTP/1.1\r\nHost: www.baidu.com\r\n\r\n"
13. local bytes, err = sock:send(req_data)
14. if err then
15. ngx.say("failed to send to baidu: ", err)
16. return
17. end
18.
19. local data, err, partial = sock:receive()
20. if err then
21. ngx.say("failed to receive from baidu: ", err)
22. return
23. end
24.
25. sock:close()
26. ngx.say("successfully talk to baidu! response first line: ", data)
27. }
28. }
```

可以看到，这里的 `socket` 操作都是同步非阻塞的，完全不像 `node.js` 那样充满各种回调，整体看上去非常简洁优雅，效率还非常棒。

对 `cosocket` 做了这么多铺垫，到底他有多么重要呢？直接看一下官方默认绑定包有多少是基于 `cosocket` 的：

- `ngx_stream_lua_module` Nginx “stream” 子系统的官方模块版本（通用的下游 TCP 对话）。
- `lua-resty-memcached` 基于 `ngx_lua cosocket` 的库。
- `lua-resty-redis` 基于 `ngx_lua cosocket` 的库。
- `lua-resty-mysql` 基于 `ngx_lua cosocket` 的库。

- [lua-resty-upload](#) 基于 ngx\_lua cosocket 的库。
- [lua-resty-dns](#) 基于 ngx\_lua cosocket 的库。
- [lua-resty-websocket](#) 提供 WebSocket 的客户端、服务端，基于 ngx\_lua cosocket 的库。

效仿这些基础库的实现方法，可以完成不同系统或组件的对接，例如 syslog、beanstalkd、mongodb 等，直接 copy 这些组件的通讯协议即可。

# 如何安全启动唯一实例的 timer

- [如何只启动一个 timer 工作？](#)

## 如何只启动一个 timer 工作？

### 应用场景

整个 OpenResty 启动后，我们有时候需要后台处理某些动作，比如数据定期清理、同步数据等。而这个后台任务实例我们期望是唯一并且安全，这里的安全指的是所有 Nginx worker 任意 crash 任何一个，有机制合理保证后续 timer 依然可以正常工作。

这里需要给大家介绍一个重要 API `ngx.worker.id()`。

1. 语法: `seq_id = ngx.worker.id()`
- 2.
3. 返回当前 Nginx 工作进程的一个顺序数字（从 0 开始）。
- 4.
5. 所以，如果工作进程总数是 N，那么该方法将返回 0 和 N - 1（包含）的一个数字。
- 6.
7. 该方法只对 Nginx 1.9.1+ 版本返回有意义的值。更早版本的 nginx，将总是返回 nil。

### 解决办法

通过 API 描述可以看到，我们可以用它来确定这个 worker 的内部身份，并且这个身份是相对稳定的。即使当前 Nginx 进程因为某些原因 crash 了，新 fork 出来的 Nginx worker 是会继承这个 worker id 的。

剩下的问题就比较简单了，完全可以把我们的 timer 绑定到某个特定的 worker 上即可。下面的例子，演示如何只在 worker.id 为 0 的进程上运行后台 timer。

```
1. init_worker_by_lua_block {
2. local delay = 3 -- in seconds
3. local new_timer = ngx.timer.at
4. local log = ngx.log
5. local ERR = ngx.ERR
6. local check
7.
8. check = function(premature)
9. if not premature then
10. -- do the health check or other routine work
11. local ok, err = new_timer(delay, check)
12. if not ok then
13. log(ERR, "failed to create timer: ", err)
```

```
14. return
15. end
16. end
17. end
18.
19. if 0 == ngx.worker.id() then
20. local ok, err = new_timer(delay, check)
21. if not ok then
22. log(ERR, "failed to create timer: ", err)
23. return
24. end
25. end
26. }
```

## 如何正确的解析域名

- [为什么我们的域名不能被解析](#)
- [搭配 resolver 指令](#)
- [进阶玩法](#)
- [高阶玩法](#)
- [最终的演示例子如下](#)
- [如何使用 /etc/hosts 自定义域名](#)

## 为什么我们的域名不能被解析

最近经常有朋友在使用一个域名地址时发现无法被正确解析

比如在使用 `MySQL` 实例时某些云会给一个私有的域名搭配自有的 `nameserver` 使用

```
1. local client = mysql:new()
2. client:connect({
3. host = "rdsmxxxxxx.mysql.rds.xxx.com",
4. port = 3306,
5. database = "test",
6. user = "test",
7. password = "123456"
8. })
```

以上代码在直接使用时往往会报一个无法解析的错误。那么怎么在 `openresty` 中使用域名呢

## 搭配 resolver 指令

我们可以直接在 `nginx` 的配置文件中使用 `resolver` 指令直接设置使用的 `nameserver` 地址。

官方文档中是这么描述的

```
1. Syntax: resolver address ... [valid=time] [ipv6=on|off];
2. Default: -
3. Context: http, server, location
```

一个简单的例子

```
1. resolver 8.8.8.8 114.114.114.114 valid=3600s;
```

不过这样的问题在于 `nameserver` 被写死在配置文件中，如果使用场景比较复杂或有内部 `dns` 服务时维护比较麻烦。

## 进阶玩法

我们的代码常常运行在各种云上，为了减少维护成本，我采用了动态读取本机 `/etc/resolv.conf` 的方法来做。

废话不说，让我们一睹为快。

```
1. local pcall = pcall
2. local io_open = io.open
3. local ngx_re_gmatch = ngx.re.gmatch
4.
5. local ok, new_tab = pcall(require, "table.new")
6.
7. if not ok then
8. new_tab = function (narr, nrec) return {} end
9. end
10.
11. local _dns_servers = new_tab(5, 0)
12.
13. local _read_file_data = function(path)
14. local f, err = io_open(path, 'r')
15.
16. if not f or err then
17. return nil, err
18. end
19.
20. local data = f:read('*all')
21. f:close()
22. return data, nil
23. end
24.
25. local _read_dns_servers_from_resolv_file = function()
26. local text = _read_file_data('/etc/resolv.conf')
27.
28. local captures, it, err
29. it, err = ngx_re_gmatch(text, [[^nameserver\s+(\d+?\.\d+?\.\d+?\.\d+?$)]], "jomi")
30.
31. for captures, err in it do
32. if not err then
33. _dns_servers[#_dns_servers + 1] = captures[1]
34. end
35. end
36. end
```

```

37.
38. _read_dns_servers_from_resolv_file()

```

通过上述代码我们成功动态拿到了一组 `nameserver` 的地址，下面就可以通过

`resty.dns.resolver` 大杀四方了

```

1. local require = require
2. local ngx_re_find = ngx.re.find
3. local lrucache = require "resty.lrucache"
4. local resolver = require "resty.dns.resolver"
5. local cache_storage = lrucache.new(200)
6.
7. local _is_addr = function(hostname)
8. return ngx_re_find(hostname, [[\d+?\.\d+?\.\d+?\.\d+$]], "jo")
9. end
10.
11. local _get_addr = function(hostname)
12. if _is_addr(hostname) then
13. return hostname, hostname
14. end
15.
16. local addr = cache_storage:get(hostname)
17.
18. if addr then
19. return addr, hostname
20. end
21.
22. local r, err = resolver:new({
23. nameservers = _dns_servers,
24. retrans = 5, -- 5 retransmissions on receive timeout
25. timeout = 2000, -- 2 sec
26. })
27.
28. if not r then
29. return nil, hostname
30. end
31.
32. local answers, err = r:query(hostname, {qtype = r.TYPE_A})
33.
34. if not answers or answers.errcode then
35. return nil, hostname
36. end
37.
38. for i, ans in ipairs(answers) do
39. if ans.address then
40. cache_storage:set(hostname, ans.address, 300)
41. return ans.address, hostname

```

```
42. end
43. end
44.
45. return nil, hostname
46. end
47.
48. ngx.say(_get_addr("www.baidu.com"))
49. ngx.say(_get_addr("192.168.0.1"))
```

我这边把解析的结果放入了 `lrucache` 缓存了 5 分钟，你们同样可以把结果放入 `shared` 中来减少 `worker` 查询次数。

## 高阶玩法

现在我们已经实现了自缓存体系的 `dns` 查询，如果搭配 `resty.http` 就会达到更好的效果。

一个简单的例子是，通过解析 `uri` 得到 `hostname`、`port`、`path`，把 `hostname` 扔给自缓存 `dns` 获取结果

发起 `request` 请求，`addr` + `port` `connect` 之，设置 `header` 的 `host` 为 `hostname`，`path` 等值来实现 `ip` 直接访问等高阶用法。

这里就不过多的阐述了。

## 最终的演示例子如下

```
1. local client = mysql:new()
2. client:connect({
3. host = _get_addr(conf.mysql_hostname),
4. port = 3306,
5. database = "test",
6. user = "test",
7. password = "123456"
8. })
```

## 如何使用 `/etc/hosts` 自定义域名

还有些同学可能会在 `hosts` 文件中自定义域名和 `ip`，这时候 `resolve` 是无法正常解析的。

这个时候可以借助 `dnsmasq` 这个服务来缓存我们的 `dns` 结果，`hosts` 的定义可以被该服务



识别。

需要在 `nginx` 的配置文件中，设置 `resolver` 为 `dnsmasq` 服务的监听地址即可。

See:

<http://hambut.com/2016/09/09/how-to-resolve-the-domain-name-in-openresty/>

# LuaRestyDNSLibrary

- [LuaRestyDNSLibrary 简介](#)

## LuaRestyDNSLibrary 简介

这个 Lua 库提供了 ngx\_lua 模块的 DNS 解析器：

### lua-resty-dns

这个 Lua 库基于 ngx\_lua 的 cosocket API 实现，可以确定是 100% 非阻塞的。注意，该模块至少需要 ngx\_lua 0.5.12 或 OpenResty 1.2.1.11 版本。Lua bit 模块也是需要的。如果你的 ngx\_lua 绑定的 LuaJIT 的版本是 2.0，就已经默认开启了 Lua bit 模块。注意，这个模块在 OpenResty 集成环境中默认是开启的。

使用代码示例：

```
1. lua_package_path "/path/to/lua-resty-dns/lib/?.lua;;";
2.
3. server {
4. location = /dns {
5. content_by_lua_block {
6. local resolver = require "resty.dns.resolver"
7. local r, err = resolver:new{
8. nameservers = {"8.8.8.8", {"8.8.4.4", 53} },
9. retrans = 5, -- 5 retransmissions on receive timeout
10. timeout = 2000, -- 2 sec
11. }
12.
13. if not r then
14. ngx.say("failed to instantiate the resolver: ", err)
15. return
16. end
17.
18. local answers, err = r:query("www.google.com")
19. if not answers then
20. ngx.say("failed to query the DNS server: ", err)
21. return
22. end
23.
24. if answers.errcode then
25. ngx.say("server returned error code: ", answers.errcode,
26. ": ", answers.errstr)
27. end
28.
```

```
29. for i, ans in ipairs(answers) do
30. ngx.say(ans.name, " ", ans.address or ans.cname,
31. " type:", ans.type, " class:", ans.class,
32. " ttl:", ans.ttl)
33. end
34. }
35. }
36. }
```

# 使用动态 DNS 来完成 HTTP 请求

- 使用动态 DNS 来完成 HTTP 请求

## 使用动态 DNS 来完成 HTTP 请求

其实针对大多应用场景，DNS 是不会频繁变更的，使用 Nginx 默认的 resolver 配置方式就能解决。

对于部分应用场景，可能需要支持的系统众多：win、centos、ubuntu 等，不同的操作系统获取 DNS 的方法都不太一样。再加上我们使用 Docker，导致我们在容器内部获取 DNS 变得更加难以准确。

如何能够让 Nginx 使用随时可以变化的 DNS 源，成为我们急待解决的问题。

当我们需要在某一个请求内部发起这样一个 http 查询，采用 proxy\_pass 是不行的（依赖 resolver 的 DNS，如果 DNS 有变化，必须要重新加载配置），并且由于 proxy\_pass 不能直接设置 keepalive，导致每次请求都是短链接，性能损失严重。

使用 resty.http，目前这个库只支持 ip : port 的方式定义 url，其内部实现并没有支持 domain 解析。resty.http 是支持 set\_keepalive 完成长连接，这样我们只需要让他支持 DNS 解析就能有完美解决方案了。

```
1. local resolver = require "resty.dns.resolver"
2. local http = require "resty.http"
3.
4. function get_domain_ip_by_dns(domain)
5. -- 这里写死了google的域名服务ip, 要根据实际情况做调整（例如放到指定配置或数据库中）
6. local dns = "8.8.8.8"
7.
8. local r, err = resolver:new{
9. nameservers = {dns, {dns, 53} },
10. retrans = 5, -- 5 retransmissions on receive timeout
11. timeout = 2000, -- 2 sec
12. }
13.
14. if not r then
15. return nil, "failed to instantiate the resolver: " .. err
16. end
17.
18. local answers, err = r:query(domain)
19. if not answers then
20. return nil, "failed to query the DNS server: " .. err
21. end
```

```

22.
23. if answers.errcode then
24. return nil, "server returned error code: " .. answers.errcode .. ": " .. answers.errstr
25. end
26.
27. for i, ans in ipairs(answers) do
28. if ans.address then
29. return ans.address
30. end
31. end
32.
33. return nil, "not founded"
34. end
35.
36. function http_request_with_dns(url, param)
37. -- get domain
38. local domain = ngx.re.match(url, [[/([^\s]+?)/]])
39. domain = (domain and 1 == #domain and domain[1]) or nil
40. if not domain then
41. ngx.log(ngx.ERR, "get the domain fail from url:", url)
42. return {status=ngx.HTTP_BAD_REQUEST}
43. end
44.
45. -- add param
46. if not param.headers then
47. param.headers = {}
48. end
49. param.headers.Host = domain
50.
51. -- get domain's ip
52. local domain_ip, err = get_domain_ip_by_dns(domain)
53. if not domain_ip then
54. ngx.log(ngx.ERR, "get the domain[" .. domain .. "] ip by dns failed:", err)
55. return {status=ngx.HTTP_SERVICE_UNAVAILABLE}
56. end
57.
58. -- http request
59. local httpc = http.new()
60. local temp_url = ngx.re.gsub(url, "/[^.domain.]/", string.format("//%s/", domain_ip))
61.
62. local res, err = httpc:request_uri(temp_url, param)
63. if err then
64. return {status=ngx.HTTP_SERVICE_UNAVAILABLE}
65. end
66.
67. -- httpc:request_uri 内部已经调用了keepalive, 默认支持长连接
68. -- httpc:set_keepalive(1000, 100)
69. return res

```

```
70. end
```

动态 DNS，域名访问，长连接，这些都具备了，貌似可以安稳一下。在压力测试中发现这里面有个机制不太好，就是对于指定域名解析，每次都要和 DNS 服务会话询问 IP 地址，实际上这是不需要的。普通的浏览器，都会对 DNS 的结果进行一定的缓存，那么这里也必须要使用了。

对于缓存实现代码，请参考 ngx\_lua 相关章节，肯定会有惊喜等着你挖掘碰撞。

# LuaRestyLock

- [LuaRestyLock](#)
  - [链接](#)

## LuaRestyLock

---

### 链接

---

- [缓存失效风暴](#)

# 缓存失效风暴

- 缓存失效风暴

## 缓存失效风暴

看下这个段伪代码：

```
1. local value = get_from_cache(key)
2. if not value then
3. value = query_db(sql)
4. set_to_cache(value, timeout = 100)
5. end
6. return value
```

看上去没有问题，在单元测试情况下，也不会有异常。

但是，进行压力测试的时候，你会发现，每隔 100 秒，数据库的查询就会出现一次峰值。如果你的 cache 失效时间设置的比较长，那么这个问题被发现的机率就会降低。

为什么会出现峰值呢？想象一下，在 cache 失效的瞬间，如果并发请求有 1000 条同时到了

`query_db(sql)` 这个函数会怎样？没错，会有 1000 个请求打向数据库。这就是缓存失效瞬间引起的风暴。它有一个英文名，叫 **“dog-pile effect”**。

怎么解决？自然的想法是发现缓存失效后，加一把锁来控制数据库的请求。具体的细节，春哥在 `lua-resty-lock` 的文档里面做了详细的说明，我就不重复了，请看[这里](#)。多说一句，`lua-resty-lock` 库本身已经替你完成了 `wait for lock` 的过程，看代码的时候需要注意下这个细节。



# 测试

- [测试](#)
  - [链接](#)

# 测试

---

# 链接

---

- [代码静态分析](#)
- [单元测试](#)
- [代码覆盖率](#)
- [API 测试](#)
- [性能测试](#)
- [持续集成](#)
- [灰度发布](#)

# 代码静态分析

- 代码静态分析

## 代码静态分析

代码静态分析可以在不运行代码的情况下，提前检测代码。

主要可以做两点

1. 语法检测
2. 编码规范检测

作为开发人员，在日常编码中，难免会犯一些低级错误，比如少个括号，少个逗号，使用了未定义变量等等，我们往往会使用编辑器的 lint 插件来检测此类错误。

对于我们 OpenResty 开发中，日常开发的都是 Lua 代码，所以我们可以使用 [luacheck](#) 这款静态代码检测工具来帮助我们检查，比较好的一点是这款工具对 ngx\_lua 做了一些支持，我们使用的 ngx 变量在开启了配置 `--std ngx_lua` 后即可被 luacheck 识别，而不会被认为是未定义的变量。

我们可以通过 luarocks 来安装：

```
1. → luarocks install luacheck
```

使用也很方便，只需要 `luacheck filename or directory` 即可。

```
1. $ luacheck src extra_file.lua another_file.lua
2. Checking src/good_code.lua OK
3. Checking src/bad_code.lua 3 warnings
4.
5. src/bad_code.lua:3:23: unused variable length argument
6. src/bad_code.lua:7:10: setting non-standard global variable embrace
7. src/bad_code.lua:8:10: variable opt was previously defined as an argument on line 7
8.
9. Checking src/python_code.lua 1 error
10.
11. src/python_code.lua:1:6: expected '=' near '__future__'
12.
13. Checking extra_file.lua 5 warnings
14.
15. extra_file.lua:3:18: unused argument baz
16. extra_file.lua:4:8: unused loop variable i
17. extra_file.lua:13:7: accessing uninitialized variable a
```

```

18. extra_file.lua:14:1: value assigned to variable x is unused
19. extra_file.lua:21:7: variable z is never accessed
20.
21. Checking another_file.lua 2 warnings
22.
23. another_file.lua:2:7: unused variable height
24. another_file.lua:3:7: accessing undefined variable heighth
25.
26. Total: 10 warnings / 1 error in 5 files

```

当然你也可以指定一些参数来运行 `luacheck`，常见的有 `std`、`ignore`、`globals` 等，我们一般会必选上 `--std ngx_lua` 来识别 `ngx_lua` 的全局变量，具体的规则可以查看 [官方文档](#)

除了使用命令行参数，`luacheck` 还支持使用配置文件的形式，这也是我们推荐的做法。`luacheck` 使用时会优先查找当前目录下的 `.luacheckrc` 文件，未找到则去上层目录查找，以此类推。所以我们可以项目的根目录下放置一个我们配置好的 `.luacheckrc` 文件以便之后使用。

一个 `.luacheckrc` 大概是这样子的：

```

1. -- .luacheckrc 文件其实就是个 lua 代码文件
2. cache = true
3. std = 'ngx_lua'
4. ignore = {
5. "_", -- 忽略 _ 变量，我们用它来表示没有用到的变量
6. "6..", -- 忽略格式上的warning
7. }
8. -- 这里因为客观原因，定的比较松。如果条件允许，你可以去掉这些豁免条例。
9. unused = false
10. unused_args = false
11. unused_secondaries = false
12. redefined = false
13. -- top-level module name
14. globals = {
15. -- 标记 ngx.header and ngx.status 是可以被写入的
16. "ngx",
17. }
18.
19. -- 因为历史遗留原因，我们代码里有部分采用了旧风格的 module(..., package.seeall)
20. -- 来定义模块。下面一行命令用于找出这一类文件，并添加豁免的规则。
21. -- find -name '*.lua' -exec grep '^module(' -l {} \; | awk '{ print "\"" $0 "\"", " }'
22. local old_style_modules = {
23. -- ...
24. }
25. for _, path in ipairs(old_style_modules) do
26. files[path].module = true
27. files[path].allow_defined_top = true
28. end

```

```

29.
30. -- 对用了 busted 测试框架的测试文件添加额外的标准
31. files["test/*_spec.lua"].std = "+busted"
32.
33. -- 不检查来自第三方的代码库
34. exclude_files = {
35. "nginx/resty",
36. }

```

luacheck 也可以集成进编辑器使用，支持的有 Vim, Sublime Text, Atom, Emacs, Brackets。基本主流的编辑器都有支持。具体可以看相应的 [使用文档](#)，这里就不做说明了。

这里特别说一下的是，我们在项目中使用了 git pre-commit hooks 来进行静态检查，在 git commit 前会检测本次提交修改和新增的代码，判断是否通过了 luacheck 的检测，未通过会给出提示并询问是否退出这次 commit。这一切都是通过 git hooks 来做的，顾名思义我们的钩子是下在 commit 这个动作上的，只要进行了 commit 操作，就会触发我们的钩子。git 内置了一些钩子，不同的 git 操作会触发不同的钩子，这些钩子放在项目文件夹的 `.git/hooks/` 文件夹下，我们这里用到的是 pre-commit。

```

1. #!/usr/bin/env bash
2. lua_files=$(git status -s|awk '{if (($1=="M"||$1=="A") && $2 ~ /\.lua$/){print $2;}}')
3.
4. if [["$lua_files" != ""]]; then
5. result=$(luacheck $lua_files)
6.
7. if [["$result" =~ .*:.*:.*:]]; then
8. echo "$result"
9. echo ""
10. exec < /dev/tty
11. read -p "Abort commit?(Y/n)"
12.
13. if [["$REPLY" == y*]] || [["$REPLY" == Y*]]; then
14. echo "Abort commit"
15. exit 1
16. fi
17. fi
18. fi

```

# 单元测试

- [单元测试](#)

## 单元测试

单元测试 (unit testing)，是指对软件中的最小可测试单元进行检查和验证。对于单元测试中单元的含义，一般来说，要根据实际情况去判定其具体含义，如 C 语言中单元指一个函数，Java 里单元指一个类，图形化的软件中可以指一个窗口或一个菜单等。总的来说，单元就是人为规定的最小的被测功能模块。单元测试是在软件开发过程中要进行的最低级别的测试活动，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。

单元测试的书写、验证，互联网公司几乎都是研发自己完成的，我们要保证代码出手时可交付、符合预期。如果连自己的预期都没达到，后面所有的工作，都将是额外无用功。

Lua 中我们没有找到比较好的测试库，参考了 Golang、Python 等语言的单元测试书写方法以及调用规则，编写了[lua-resty-test](#)测试库，这里给自己的库推广一下，希望这东东也是你们的真爱。

nginx示例配置

```

1. #you do not need the following line if you are using
2. #the ngx_openresty bundle:
3.
4. lua_package_path "/path/to/lua-resty-redis/lib/?.lua;";
5.
6. server {
7. location /test {
8. content_by_lua_file test_case_lua/unit/test_example.lua;
9. }
10. }
```

test\_case\_lua/unit/test\_example.lua:

```

1. local iresty_test = require "resty.iresty_test"
2. local tb = iresty_test.new({unit_name="example"})
3.
4. function tb:init()
5. self:log("init complete")
6. end
7.
8. function tb:test_00001()
9. error("invalid input")
10. end
```

```
11.
12. function tb:atest_00002()
13. self:log("never be called")
14. end
15.
16. function tb:test_00003()
17. self:log("ok")
18. end
19.
20. -- units test
21. tb:run()
```

- init 里面我们可以完成一些基础、公共变量的初始化，例如特定的 url 等
- test\_\*\*\*\*\* 函数中添加我们的单元测试代码
- 搞定测试代码，它即是单元测试，也是成压力测试

输出日志：

```
1. TIME Name Log
2. 0.000 [example] unit test start
3. 0.000 [example] init complete
4. 0.000 _[test_00001] fail ...de/nginx/main_server/test_case_lua/unit/test_example.lua:9:
 invalid input
5. 0.000 _[test_00003] ↓ ok
6. 0.000 _[test_00003] PASS
7. 0.000 [example] unit test complete
```

# 代码覆盖率

- 代码覆盖率

## 代码覆盖率

这是一个重要的可量化指标，如果代码覆盖率很高，你就可以放心的修改代码，在发版本的时候也能睡个安稳觉。否则就是拆东墙补西墙，陷入无尽的 bug 诅咒中。

那么在 OpenResty 里面如何看到代码覆盖率呢？其实很简单，使用 [LuaCov](#) 可以很方便的实现。

我们先了解下 LuaCov，这是一个针对 Lua 脚本的代码覆盖率工具，通过 luarocks 来安装：

```
1. luarocks install luacov
```

如果你的项目比较大，建议安装

```
1. luarocks install cluacov
```

后者通过 C 代码加速了 luacov 的耗时操作。亲测能够快近一倍呢。

安装了 LuaCov 之后，还需要先配置一下。

在 OpenResty 里面使用 LuaCov，只用在 Nginx.conf 中增加 `init_by_lua_block`（只能放在 http 上下文中）既可。

```
1. init_by_lua_block {
2. require 'luacov.tick'
3. jit.off()
4. }
```

这个 `*_bolck` 语法在较新的 OpenResty 版本中新引入，如果提示指令不存在，请使用最新的版本来测试。

抑或在 LuaCov 配置文件 `.luacov` 中加入 `tick = true` 这一行。效果是一样的。

另外，由于 LuaCov 的分析文件默认按照 100 条一批来写入的，如果你的代码量不大，可能就会不准确。

这时候可以设置 `savestepsize = 2` 来提高写入频率。

更多可用配置请访问 LuaCov 的配置文档：

<http://keplerproject.github.io/luacov/doc/modules/luacov.defaults.html>

重新启动 OpenResty 后, LuaCov 就已经生效了。你可以跑下单元测试, 或者访问下 API 接口, 在当前工作目录下, 就会生成 `luacov.stats.out` 这个统计文件。然后 `cd` 到这个目录下, 运行:

```
1. luacov
```

就会生成 `luacov.report.out` 这个可读性比较好的覆盖率报告。需要注意的是, `luacov` 这个命令后面不用加任何的参数, 这个在官方文档里面有说明, 只是比较隐晦。

我们看下 `luacov.report.out` 里面的一个片段:

```
1. 1 function get_config(mid, args)
2. 13 local configs = {}
3. 13 local res, err = red:hmget("client_".. mid, "tpl_id", "gid")
4. 13 if err then
5. ****0 return nil, err
6. end
7. end
```

代码前面的数字, 代表的是运行的次数。而 `****0` 很明确的指出这行代码没有被测试案例覆盖到。

在 `luacov.report.out` 的最后会有一个汇总的覆盖率报告:



| Summary |      |         |           |
|---------|------|---------|-----------|
| 11      | 58   | 15.94%  | /Users/mi |
| 90      | 338  | 21.03%  | /Users/mi |
| 44      | 0    | 100.00% | /Users/mi |
| 18      | 93   | 16.22%  | /Users/mi |
| 9       | 29   | 23.68%  | /Users/mi |
| 14      | 132  | 9.59%   | /Users/mi |
| 167     | 702  | 19.22%  | /Users/mi |
| 59      | 81   | 42.14%  | /Users/mi |
| 33      | 330  | 9.09%   | /Users/mi |
| 28      | 178  | 13.59%  | /Users/mi |
| 48      | 331  | 12.66%  | /Users/mi |
| 6       | 14   | 30.00%  | /Users/mi |
| 71      | 60   | 54.20%  | /Users/mi |
| 104     | 105  | 49.76%  | /Users/mi |
| 70      | 69   | 50.36%  | /Users/mi |
| 5       | 6    | 45.45%  | /Users/mi |
| 19      | 3    | 86.36%  | /Users/mi |
| 15      | 26   | 36.59%  | /Users/mi |
| 223     | 65   | 77.43%  | /Users/mi |
| -----   |      |         |           |
| 1034    | 2620 | 28.30%  |           |

可以看到，在我的这个单元测试里面，一共涉及到近 20 个代码文件。其中倒数第三个是我测试的 API 接口，覆盖到的代码有 19 行，没有覆盖的有 3 行，所以代码覆盖率是  $86.36\% (19.0 / (19 + 3))$ 。

最后有一个总的代码覆盖率是 28.3%，这个值在跑完所有单元测试后是有意义的，单跑一个是没有参考价值的，因为很多基础函数可能并没有运行到。



# API 测试

- [API 测试](#)

## API 测试

API (Application Programming Interface) 测试的自动化是软件测试最基本的一种类型。从本质上来说, API 测试是用来验证组成软件的那些单个方法的正确性, 而不是测试整个系统本身。

API 测试也称为单元测试 (Unit Testing)、模块测试 (Module Testing)、组件测试 (Component Testing) 以及元件测试 (Element Testing)。从技术上来说, 这些术语是有很大的差别的, 但是在日常应用中, 你可以认为它们大致相同的意思。它们背后的思想就是, 必须确定系统中每个单独的模块工作正常, 否则, 这个系统作为一个整体不可能是正确的。毫无疑问, API 测试对于任何重要的软件系统来说都是必不可少的。

我们对 API 测试的定位是服务对外输出的 API 接口测试, 属于黑盒、偏重业务的测试步骤。

看过上一章内容的朋友还记得[lua-resty-test](#), 我们的 API 测试同样是需要它来完成。`get_client_tasks` 是终端用来获取当前可执行任务清单的 API, 我们用它当做例子给大家做个介绍。

*nginx conf:*

```
1. location ~* /api/([\w_]+?)\.json {
2. content_by_lua_file lua/$1.lua;
3. }
4.
5. location ~* /unit_test/([\w_]+?)\.json {
6. lua_check_client_abort on;
7. content_by_lua_file test_case_lua/unit/$1.lua;
8. }
```

*API测试代码:*

```
1. -- unit test for /api/get_client_tasks.json
2. local tb = require "resty.iresty_test"
3. local json = require("cjson")
4. local test = tb.new({unit_name="get_client_tasks"})
5.
6. function tb:init()
7. self.mid = string.rep('0',32)
8. end
9.
10. function tb:test_0000()
11. -- 正常请求
```

```

12. local res = ngx.location.capture(
13. '/api/get_client_tasks.json?mid='..self.mid,
14. { method = ngx.HTTP_POST, body=[[{"type":[1600,1700]}]] }
15.)
16.
17. if 200 ~= res.status then
18. error("failed code:" .. res.status)
19. end
20. end
21.
22. function tb:test_0001()
23. -- 缺少body
24. local res = ngx.location.capture(
25. '/api/get_client_tasks.json?mid='..self.mid,
26. { method = ngx.HTTP_POST }
27.)
28.
29. if 400 ~= res.status then
30. error("failed code:" .. res.status)
31. end
32. end
33.
34. function tb:test_0002()
35. -- 错误的json内容
36. local res = ngx.location.capture(
37. '/api/get_client_tasks.json?mid='..self.mid,
38. { method = ngx.HTTP_POST, body=[[{"type":["1600,1700]}]] }
39.)
40.
41. if 400 ~= res.status then
42. error("failed code:" .. res.status)
43. end
44. end
45.
46. function tb:test_0003()
47. -- 错误的json格式
48. local res = ngx.location.capture(
49. '/api/get_client_tasks.json?mid='..self.mid,
50. { method = ngx.HTTP_POST, body=[[{"type":["1600,1700"]}]] }
51.)
52.
53. if 400 ~= res.status then
54. error("failed code:" .. res.status)
55. end
56. end
57.
58. test:run()

```

Nginx output:

```
1. 0.000 [get_client_tasks] unit test start
2. 0.001 _[test_0000] PASS
3. 0.001 _[test_0001] PASS
4. 0.001 _[test_0002] PASS
5. 0.001 _[test_0003] PASS
6. 0.001 [get_client_tasks] unit test complete
```

使用 `capture` 来模拟请求，其实是不靠谱的。如果我们要完全 100% 模拟客户请求，这时候就要使用第三方 `cosocket` 库，例如[lua-resty-http](#)，这样我们才可以完全指定 `http` 参数。

# 性能测试

- 性能测试
  - [ab 压测](#)
  - [wrk 压测](#)

## 性能测试

---

性能测试应该有两个方向：

- 单接口压力测试
- 生产环境模拟用户操作高压测试

生产环境模拟测试，目前我们都是交给公司的 QA 团队专门完成的。这块我只能粗略列举一下：

- 获取 1000 用户以上生产用户的访问日志（统计学要求 1000 是最小集合）
- 计算指定时间内（例如 10 分钟），所有接口的触发频率
- 使用测试工具（loadrunner, jmeter 等）模拟用户请求接口
- 适当放大压力，就可以模拟 2000、5000 等用户数的情况

## ab 压测

单接口压力测试，我们都是由研发团队自己完成的。传统一点的方法，我们可以使用 ab(apache bench)这样的工具。

```
1. #ab -n10 -c2 http://haosou.com/
2.
3. -- output:
4. ...
5. Complete requests: 10
6. Failed requests: 0
7. Non-2xx responses: 10
8. Total transferred: 3620 bytes
9. HTML transferred: 1780 bytes
10. Requests per second: 22.00 [#/sec] (mean)
11. Time per request: 90.923 [ms] (mean)
12. Time per request: 45.461 [ms] (mean, across all concurrent requests)
13. Transfer rate: 7.78 [Kbytes/sec] received
14. ...
```

大家可以看到 ab 的使用超级简单，简单的有点弱了。在上面的例子中，我们发起了 10 个请求，每个请求都是一样的，如果每个请求有差异，ab 就无能为力。

## wrk 压测

单接口压力测试，为了满足每个请求或部分请求有差异，我们试用过很多不同的工具。最后找到了这个和我们距离最近、表现优异的测试工具 [wrk](#)，这里我们重点介绍一下。

wrk 如果要完成和 ab 一样的压力测试，区别不大，只是命令行参数略有调整。下面给大家举例每个请求都有差异的例子，供大家参考。

`scripts/counter.lua`

```
1. -- example dynamic request script which demonstrates changing
2. -- the request path and a header for each request
3. -----
4. -- NOTE: each wrk thread has an independent Lua scripting
5. -- context and thus there will be one counter per thread
6.
7. counter = 0
8.
9. request = function()
10. path = "/" .. counter
11. wrk.headers["X-Counter"] = counter
12. counter = counter + 1
13. return wrk.format(nil, path)
14. end
```

`shell`执行

```
1. # ./wrk -c10 -d1 -s scripts/counter.lua http://baidu.com
2. Running 1s test @ http://baidu.com
3. 2 threads and 10 connections
4. Thread Stats Avg Stdev Max +/- Stdev
5. Latency 20.44ms 3.74ms 34.87ms 77.48%
6. Req/Sec 226.05 42.13 270.00 70.00%
7. 453 requests in 1.01s, 200.17KB read
8. Socket errors: connect 0, read 9, write 0, timeout 0
9. Requests/sec: 449.85
10. Transfer/sec: 198.78KB
```

`WireShark`抓包印证一下

```
1. GET /228 HTTP/1.1
2. Host: baidu.com
3. X-Counter: 228
4.
5. ...(应答包 省略)
6.
7. GET /232 HTTP/1.1
```

```
8. Host: baidu.com
9. X-Counter: 232
10.
11. ...(应答包 省略)
```

wrk 是个非常成功的作品，它的实现更是从多个开源作品中挖掘优秀的代码融入自身，如果你每天还在用 C/C++，那么 wrk 的成功，对你应该有绝对的借鉴意义，多抬头，多看优秀代码，我们绝对可以创造奇迹。

引用wrk官方结尾：

```
1. wrk contains code from a number of open source projects including the 'ae'
2. event loop from redis, the nginx/joyent/node.js 'http-parser', and Mike
3. Pall's LuaJIT.
```



# 持续集成

- [持续集成](#)

## 持续集成

---

我们做的还不够好，先占个坑。

欢迎贡献章节。

# 灰度发布

- [灰度发布](#)

## 灰度发布

---

我们做的还不够好，先占个坑。

欢迎贡献章节。

# Web 服务

- [Web 服务](#)
  - [链接](#)

## Web 服务

---

## 链接

---

- [API的设计](#)
- [数据合法性检测](#)
- [协议无痛升级](#)
- [代码规范](#)
- [连接池](#)
- [C10K 编程](#)
- [TIME\\_WAIT 问题](#)
- [与 Docker 使用的网络瓶颈](#)

# API的设计

- [API 的设计](#)
  - [什么是 REST](#)
  - [REST 使用举例](#)
  - [还有办法压缩不？](#)
  - [REST 风格的缺点](#)

## API 的设计

OpenResty，最擅长的应用场景之一就是 API Server。如果我们只有简单的几个 API 出口、入口，那么我们可以相对随意简单一些。

举例几个简单API接口输出：

```
1. server {
2. listen 80;
3. server_name localhost;
4.
5. location /app/set {
6. content_by_lua_block {
7. ngx.say('set data')
8. }
9. }
10.
11. location /app/get {
12. content_by_lua_block {
13. ngx.say('get data')
14. }
15. }
16.
17. location /app/del {
18. content_by_lua_block {
19. ngx.say('del data')
20. }
21. }
22. }
```

当你的 API Server 接口服务比较多，那么上面的方法显然不适合我们（太啰嗦）。这里推荐一下 REST 风格。

## 什么是 REST

从资源的角度来观察整个网络，分布在各处的资源由 URI 确定，而客户端的应用通过 URI 来获取资源的表示方式。获得这些表徵致使这些应用程序转变了其状态。随着不断获取资源的表示方式，客户端应用不断地在转变着其状态，所谓表述性状态转移（Representational State Transfer）。

这一观点不是凭空臆造的，而是通过观察当前 Web 互联网的运作方式而抽象出来的。Roy Fielding 认为，

1. 设计良好的网络应用表现为一系列的网页，这些网页可以看作虚拟的状态机，用户选择这些链接
2. 导致下一网页传输到用户端展现给使用的人，而这正代表了状态的转变。

REST是设计风格而不是标准。

REST 通常基于使用 HTTP，URI，和 XML 以及 HTML 这些现有的广泛流行的协议和标准。

- 资源是由 URI 来指定。
- 对资源的操作包括获取、创建、修改和删除资源，这些操作正好对应 HTTP 协议提供的 GET、POST、PUT 和 DELETE 方法。
- 通过操作资源的表现形式来操作资源。
- 资源的表现形式则是 XML 或者 HTML，取决于读者是机器还是人，是消费 Web 服务的客户软件还是 Web 浏览器。当然也可以是任何其他的格式。

REST的要求

- 客户端和服务端结构
- 连接协议具有无状态性
- 能够利用 Cache 机制增进性能
- 层次化的系统

## REST 使用举例

按照 REST 的风格引导，我们有关数据的 API Server 就可以变成这样。

```

1. server {
2. listen 80;
3. server_name localhost;
4.
5. location /app/task01 {
6. content_by_lua_block {
7. ngx.say(ngx.req.get_method() .. ' task01')
8. }
9. }
10. location /app/task02 {
11. content_by_lua_block {
12. ngx.say(ngx.req.get_method() .. ' task02')
13. }

```

```

14. }
15. location /app/task03 {
16. content_by_lua_block {
17. ngx.say(ngx.req.get_method() .. ' task03')
18. }
19. }
20. }

```

对于 `/app/task01` 接口，这时候我们可以用下面的方法，完成对应的方法调用。

```

1. # curl -X GET http://127.0.0.1/app/task01
2. # curl -X PUT http://127.0.0.1/app/task01
3. # curl -X DELETE http://127.0.0.1/app/task01

```

## 还有办法压缩不？

上一个章节，如果 task 类型非常多，那么后面这个配置依然会随着业务调整而调整。其实每个程序员都有一定的洁癖，是否可以以后直接写业务，而不用每次都修改主配置，万一改错了，服务就起不来了。

引用一下 HttpLuaModule 官方示例代码。

```

1. # use nginx var in code path
2. # WARNING: contents in nginx var must be carefully filtered,
3. # otherwise there'll be great security risk!
4. location ~ ^/app/([_a-zA-Z0-9/]+) {
5. set $path $1;
6. content_by_lua_file /path/to/lua/app/root/$path.lua;
7. }

```

这下世界宁静了，每天写 Lua 代码的同学，再也不用去每次修改 Nginx 主配置了。有新业务，直接开工。顺路还强制了入口文件命名规则。对于后期检查维护更容易。

## REST 风格的缺点

需要一定的学习成本，如果你的接口是暴露给运维、售后、测试等不同团队，那么他们经常不去确定当时的 `method`。当他们查看、模拟的时候，具有一定学习难度。

REST 推崇使用 HTTP 返回码来区分返回结果，但最大的问题在于 HTTP 的错误返回码（4xx 系列为主）不够多，而且订得很随意。比如用 API 创建一个用户，那么错误可能有：

- 调用格式错误(一般返回 400, 405)
- 授权错误(一般返回 403)
- “运行期”错误

- 用户名冲突
- 用户名不合法
- email 冲突
- email 不合法

# 数据合法性检测

- [数据合法性检测](#)
  - [JSON 数据格式](#)
  - [关键字段编码为 HEX，长度不定](#)
  - [TABLE 内部字段类型](#)

## 数据合法性检测

对用户输入的数据进行合法性检查，避免错误非法的数据进入服务，这是业务系统最常见的需求。很可惜 Lua 目前没有特别好的数据合法性检查库。

坦诚我们自己做的也不够好，这里只能抛砖引玉，看看大家是否有更好办法。

我们有这么几个主要的合法性检查场景：

- JSON 数据格式
- 关键字段编码为 HEX ( 0-9, a-f, A-F )，长度不定
- TABLE 内部字段类型

### JSON 数据格式

这里主要是 json decode 时，可能抛出异常的问题。我们已经在 [json 解析的异常捕获](#) 一章中详细说明了问题本身以及解决方法，这里就不再重复。

### 关键字段编码为 HEX，长度不定

HEX 编码，最常见的存在有 MD5 值等。他们是由 0-9, A-F ( 或 a-f ) 组成。笔者把使用过的代码版本逐一罗列，并进行性能测试。通过这个测试，我们不仅仅可以收获参数校验的正确写法，以及可以再次印证一下效率最高的匹配，应该注意什么。

```

1. require "resty.core.regex"
2.
3.
4. -- 纯 lua 版本，优点是兼容性好，可以适用任何 lua 语言环境
5. function check_hex_lua(str)
6. if "string" ~= type(str) then
7. return false
8. end
9.
10. for i = 1, #str do
11. local ord = str:byte(i)
12. if not (

```



```

13. (48 <= ord and ord <= 57) or
14. (65 <= ord and ord <= 70) or
15. (97 <= ord and ord <= 102)
16.) then
17. return false
18. end
19. end
20. return true
21. end
22.
23. -- 使用 ngx.re.* 完成, 没有使用任何调优参数
24. function check_hex_default(str)
25. if "string" ~= type(str) then
26. return false
27. end
28.
29. return ngx.re.find(str, "[^0-9a-fA-F]") == nil
30. end
31.
32. -- 使用 ngx.re.* 完成, 使用调优参数 "jo"
33. function check_hex_jo(str)
34. if "string" ~= type(str) then
35. return false
36. end
37.
38. return ngx.re.find(str, "[^0-9a-fA-F]", "jo") == nil
39. end
40.
41.
42. -- 下面就是测试用例部分代码
43. function do_test(name, fun)
44. ngx.update_time()
45. local start = ngx.now()
46.
47. local t = "012345678901234567890123456789abcdefABCDEF"
48. assert(fun(t))
49. for i=1,10000*300 do
50. fun(t)
51. end
52.
53. ngx.update_time()
54. print(name, "\ttimes:", ngx.now() - start)
55. end
56.
57. do_test("check_hex_lua", check_hex_lua)
58. do_test("check_hex_default", check_hex_default)
59. do_test("check_hex_jo", check_hex_jo)

```

把上面的源码在 OpenResty 环境中运行，输出结果如下：

```
1. → resty test.lua
2. check_hex_lua times:1.0390000343323
3. check_hex_default times:5.1929998397827
4. check_hex_jo times:0.4539999961853
```

不知道这个结果大家是否有些意外，`check_hex_default` 的运行效率居然比 `check_hex_lua` 要差。不过所幸的是我们对正则开启了 `jo` 参数优化后，速度上有明显提升。

引用一下 `ngx.re.*` 官方 wiki 的原文：在优化性能时，`o` 选项非常有用，因为正则表达式模板将仅仅被编译一次，之后缓存在 worker 级的缓存中，并被此 Nginx worker 处理的所有请求共享。缓存数量上限可以通过 `lua_regex_cache_max_entries` 指令调整。

课后小作业：为什么测试用例中要使用 `ngx.update_time()` 呢？好好想一想。

课后小作业：在测试用例里面加了一行 `require "resty.core.regex"`。试试去掉这一行，重新跑下程序。结果怎么样？

## TABLE 内部字段类型

当我们接收客户端请求，除了指定字段的特殊校验外，我们最常见的需求就是对指定字段的类型做限制了。比如用户注册接口，我们就要求对方姓名、邮箱等是个字符串，手机号、电话号码等是个数字类型，详细信息可能是个图片又或者是个嵌套的 TABLE。

例如我们接受用户的注册请求，注册接口示例请求 body 如下：

```
1. {
2. "username":"myname",
3. "age":8,
4. "tel":88888888,
5. "mobile_no":13888888888,
6. "email":"**@**.com",
7. "love_things":["football", "music"]
8. }
```

这时候可以用一个简单的字段描述格式来表达限制关系，如下：

```
1. {
2. "username":"","
3. "age":0,
4. "tel":0,
5. "mobile_no":0,
6. "email":"","
7. "love_things":[]
8. }
```

对于有效字段描述格式，数据值是不敏感的，但是数据类型是敏感的，只要数据类型能匹配，就可以让我们轻松不少。

来看下面的参数校验代码以及基本的测试用例：

```

1. function check_args_template(args, template)
2. if type(args) ~= type(template) then
3. return false
4. elseif "table" ~= type(args) then
5. return true
6. end
7.
8. for k,v in pairs(template) do
9. if type(v) ~= type(args[k]) then
10. return false
11. elseif "table" == type(v) then
12. if not check_args_template(args[k], v) then
13. return false
14. end
15. end
16. end
17.
18. return true
19. end
20.
21. local args = {name="myname", tel=888888, age=18,
22. mobile_no=138888888888, love_things = {"football", "music"}}
23.
24. print("valid check: ", check_args_template(args, {name="", tel=0, love_things={}}))
25. print("unvalid check: ", check_args_template(args, {name="", tel=0, love_things={}, email=""}))

```

运行一下上面的代码，结果如下：

```

1. → resty test.lua
2. valid check: true
3. unvalid check: false

```

可以看到，当我们业务层面需要有 email 地址但是请求方没有上送，这时候就能检测出来了。大家看到这里也许会笑，尤其是从其他成熟 web 框架中过来的同学，我们这里的校验可以说是比较粗糙简陋的，很多开源框架中的参数限制，都可以做到更加精确的限制。

如果你有更好更优雅的解决办法，欢迎与我们联系。



# 协议无痛升级

- 协议无痛升级

## 协议无痛升级

使用度最高的通讯协议，一定是 HTTP 了。优点有多少，相信大家肯定有切身体会。我相信每家公司对 HTTP 的使用都有自己的规则，甚至偏好。这东西没有谁对谁错，符合业务需求、量体裁衣是王道。这里我们想通过亲身体会，告诉大家利用好 OpenResty 的一些特性，会给我们带来惊喜。

在产品初期，由于产品初期存在极大不确定性、不稳定性，所以要暴露给开发团队、测试团队完全透明的传输协议，所以我们 1.0 版本就是一个没有任何处理的明文版本 HTTP+JSON。但随着产品功能的丰富，质量的逐步提高，具备一定的交付能力，这时候通讯协议必须要升级了。

为了更好的安全、效率控制，我们需要支持压缩、防篡改、防重复、简单加密等特性，为此我们设计了全新 2.0 通讯协议。如何让这个协议升级无感知、改动少，并且简单呢？

### 1.0 明文协议配置

```
1. location ~ ^/api/([-_a-zA-Z0-9/]+).json {
2. content_by_lua_file /path/to/lua/api/$1.lua;
3. }
```

### 1.0 明文协议引用示例：

```
1. # curl http://ip:port/api/heartbeat.json?key=value -d '...'
```

### 2.0 密文协议引用示例：

```
1. # curl http://ip:port/api/heartbeat.json?key=value&ver=2.0 -d '...'
```

从引用示例中看到，我们的密文协议主要都是在请求 body 中做的处理。最生硬的办法就是我们在每个业务入口、出口分别做协议的解析、编码处理。如果你只有几个 API 接口，那么直来直去的修改所有 API 接口源码，最为直接，容易理解。但如果你需要修改几十个 API 入口，那就要静下来考虑一下，修改的代价是否完全可控。

最后我们使用了 OpenResty 阶段的概念完成协议的转换。

```
1. location ~ ^/api/([-_a-zA-Z0-9/]+).json {
2. access_by_lua_file /path/to/lua/api/protocol_decode.lua;
3. content_by_lua_file /path/to/lua/api/$1.lua;
4. body_filter_by_lua_file /path/to/lua/api/protocol_encode.lua;
```

```
5. }
```

#### 内部处理流程说明

- `Nginx` 中这三个阶段的执行顺序：`access` → `content` → `body_filter`;
- `access_by_lua_file`: 获取协议版本 → 获取 `body` 数据 → 协议解码 → 设置 `body` 数据;
- `content_by_lua_file`: 正常业务逻辑处理, 零修改;
- `body_filter_by_lua_file`: 判断协议版本 → 协议编码。

刚好前些日子春哥公开了一篇 `GitHub` 通过引入了 `OpenResty` 解决 `SSL` 证书的问题, 他们的解决思路和我们差不多。都是利用 `access` 阶段做一些非标准 `HTTP(S)` 上的自定义修改, 但对于已有业务是不需要任何感知的。

我们这个通讯协议的无痛升级, 实际上是有很多玩法可以实现, 如果我们的业务从一开始有个相对稳定的框架, 可能完全不需要操这个心。没有引入框架, 一来是现在没有哪个框架比较成熟, 而是从基础开始更容易摸到细节。对于目前 `OpenResty` 可参考资料少的情况下, 我们更倾向于从最小工作集开始, 减少不确定性、复杂度。

也许在后面, 我们会推出我们的开发框架, 用来统一规避现在碰到的问题, 提供完整、可靠、高效的解决方法, 我们正在努力ing, 请大家拭目以待。

# 代码规范

- [代码规范](#)

## 代码规范

其实选择 `OpenResty` 的同学，应该都是对执行性能、开发效率比较在乎的，而对于代码风格、规范等这些 `小事` 不太在意。作为一个从 Linux C/C++ 转过来的研发，脚本语言的开发速度，接近 C/C++ 的执行速度，在我轻视了代码规范后，一个 BUG 的发生告诉我，没规矩不成方圆。

既然我们玩的是 `OpenResty`，那么很自然的联想到，`OpenResty` 自身组件代码风格是怎样的呢？

*lua-resty-string 的 string.lua*

```
1. local ffi = require "ffi"
2. local ffi_new = ffi.new
3. local ffi_str = ffi.string
4. local C = ffi.C
5. local setmetatable = setmetatable
6. local error = error
7. local tonumber = tonumber
8.
9.
10. local _M = { _VERSION = '0.09' }
11.
12.
13. ffi.cdef[[
14. typedef unsigned char u_char;
15.
16. u_char * ngx_hex_dump(u_char *dst, const u_char *src, size_t len);
17.
18. intptr_t ngx_atoi(const unsigned char *line, size_t n);
19.]]
20.
21. local str_type = ffi.typeof("uint8_t[?]")
22.
23.
24. function _M.to_hex(s)
25. local len = #s * 2
26. local buf = ffi_new(str_type, len)
27. C.ngx_hex_dump(buf, s, #s)
28. return ffi_str(buf, len)
29. end
30.
31.
```

```

32. function _M.atoi(s)
33. return tonumber(C.ngx_atoi(s, #s))
34. end
35.
36.
37. return _M

```

代码虽短，但我们可以从中获取很多信息：

1. 没有全局变量，所有的变量均使用 `local` 限制作用域
2. 提取公共函数到本地变量，使用本地变量缓存函数指针，加速下次使用
3. 函数名称全部小写，使用下划线进行分割
4. 两个函数之间距离两个空行

这里的第 2 条，是有争议的。当你按照这个方式写业务的时候，会有些痛苦。因为我们总是把标准 API 命名成自己的别名，不同开发协作人员，命名结果一定不一样，最后导致同一个标准 API 在不同地方变成不同别名，会给开发造成极大困惑。

因为这个可预期的麻烦，我们没有遵循第 2 条标准，尤其是具体业务上游模块。但对于被调用的次数比较多基础模块，可以使用这个方式进行调优。其实这里最好最完美的方法，应该是 Lua 编译成 Luac 的时候，直接做 Lua Byte Code 的调优，直接隐藏这个简单的处理逻辑。

有关更多代码细节，其实我觉得主要还是多看写的漂亮的代码，一旦看他们看的顺眼、形成习惯，那么就自然能写出风格一致的代码。规定的条条框框死记硬背总是不爽的，所以多去看看春哥开源的 `resty` 系列代码，顺手品一品一下不同组件的玩法也别有一番心得。

说说我上面提及的因为风格问题造出来的坑吧。

```

1. local
2. function test()
3. -- do something
4. end
5.
6. function test2()
7. -- do something
8. end

```

这是我当时不记得从哪里看到的一个 `Lua` 风格，在被引入项目初期，自我感觉良好。可突然从某个时间点开始，新合并进来的代码无法正常工作。查看最后的代码发现原来是 `test()` 函数作废，被删掉，手抖没有把上面的 `local` 也删掉。这个隐形的 `local` 就作用到了下一个函数，最终导致异常。





# 连接池

- [连接池](#)

## 连接池

作为一个专业的服务端开发工程师，我们必须要对连接池、线程池、内存池等有较深理解，并且有自己熟悉的库函数可以让我们轻松驾驭这些不同的 `池子`。既然他们都叫某某池，那么他们从基础概念上讲，原理和目的几乎是一样的，那就是 `复用`。

以连接池做引子，我们说说服务端工程师基础必修课。

从我们应用最多的 HTTP 连接、数据库连接、消息推送、日志存储等，所有点到点之间，都需要花样繁多的各色连接。为了传输数据，我们需要完成创建连接、收发数据、拆除连接。对并发量不高的场景，我们为每个请求都完整走这三步（短连接），开发工作基本只考虑业务即可，基本上也不会有什么問題。一旦挪到高并发应用场景，那么可能我们就要郁闷了。

你将会碰到下面几个常见问题：

- 性能普遍上不去
- CPU 大量资源被系统消耗
- 网络一旦抖动，会有大量 TIME\_WAIT 产生，不得不定期重启服务或定期重启机器
- 服务器工作不稳定，QPS 忽高忽低

这时候我们可以优化的第一件事情就是把短链接改成长连接。也就是改成创建连接、收发数据、收发数据... 拆除连接，这样我们就可以减少大量创建连接、拆除连接的时间。从性能上来说肯定要比短连接好很多。但这里还是有比较大的浪费。

举例：请求进入时，直接分配数据库长连接资源，假设有 80% 时间在与关系型数据库通讯，20% 时间是在与 Nosql 数据库通讯。当有 50K 个并行请求时，后端要分配  $50K * 2 = 100K$  的长连接支撑请求。无疑这时候系统压力是非常大的。数据库再牛也抵不住滥用不是？

连接池终于要出场了，它的解决思路是先把所有长连接存起来，谁需要使用，从这里取走，干完活立马放回来。那么按照这个思路，刚刚的 50K 的并发请求，最多占用后端 50K 的长连接就够了。省了一半啊有木有？

在 OpenResty 中，所有具备 `set_keepalive` 的类、库函数，说明他都是支持连接池的。

来点代码，给大家提提神，看看连接池使用时的一些注意点，麻雀虽小，五脏俱全。

```
1. server {
2. location /test {
3. content_by_lua_block {
```

```

4. local redis = require "resty.redis"
5. local red = redis:new()
6.
7. local ok, err = red:connect("127.0.0.1", 6379)
8. if not ok then
9. ngx.say("failed to connect: ", err)
10. return
11. end
12.
13. -- red:set_keepalive(10000, 100) -- 坑①
14.
15. ok, err = red:set("dog", "an animal")
16. if not ok then
17. -- red:set_keepalive(10000, 100) -- 坑②
18. return
19. end
20.
21. -- 坑③
22. red:set_keepalive(10000, 100)
23. }
24. }
25. }

```

- 坑①：只有数据传输完毕了，才能放到池子里，系统无法帮你自动做这个事情
- 坑②：不能把状态未知的连接放回池子里，设想另一个请求如果获取到一个不能用的连接，他不得哭死啊
- 坑③：逗你玩，这个不是坑，是正确的

尤其是掉进了第二个坑，你一定会莫名抓狂。不信的话，你就自己模拟试试，老带劲了。

理解了连接池，那么线程池、内存池，就应该都明白了，只是存放的东西不一样，思想没有任何区别。

# C10K 编程

- [C10K 编程](#)

## C10K 编程

比较传统的服务端程序（PHP、FAST CGI 等），大多都是通过每产生一个请求，都会有一个进程与之相对应，请求处理完毕后相关进程自动释放。由于进程创建、销毁对资源占用比较高，所以很多语言都通过常驻进程、线程等方式降低资源开销。即使是资源占用最小的线程，当并发数量超过 1K 的时候，操作系统的处理能力就开始出现明显下降，因为有太多的 CPU 时间都消耗在系统上下文切换。

由此催生了 C10K 编程，指的是服务器同时支持成千上万个连接，也就是 concurrent 10 000 connection（这也是 C10K 这个名字的由来）。由于硬件成本的大幅度降低和硬件技术的进步，加上一台服务器同时能够服务更多的客户端，就意味着服务每一个客户端的成本大幅度降低，从这个角度来看，C10K 问题显得非常有意义。

理想情况下，具备 C10K 能力的服务端处理能力是 c1K 的十倍，返回来说我们可以减少 90% 的服务器资源，多么诱人的结果。

C10K 解决了这几个主要问题：

- 单个进程或线程可以服务于多个客户端请求
- 事件触发替代业务轮询
- IO 采用非阻塞方式，减少额外不必要性能损耗

C10K 编程的世界，一定是异步编程的世界，他俩绝对是一对儿好基友。服务端一直都不缺乏新秀，各种语言、框架层出不穷。笔者了解的就有 OpenResty, Golang, Node.js, Rust, Python(gevent)等。每个语言或解决方案，都有自己完全不同的定位和表现，甚至设计哲学。但是他们从系统底层 API 应用、基本结构，都是相差不大。这些语言自身的实现机理、运行方式可能差别很大，但只要没有严重的代码错误，他们的性能指标都应该是在同一个级别的。

如果你用了这些解决方案，发现自己的性能非常低，就要好好看看自己是不是姿势有问题。

c1K → C10K → C100K → ???

人类前进的步伐，没有尽头的，总是在不停的往前跑。C10K 的问题，早就被解决，而且方法还不止一个。目前方案优化手段给力，做到 C100K 也是可以达到的。后面还有世界么？我们还能走么？

告诉你肯定是有，那就是 C10M。推荐大家了解一下[dpdk](#)这个项目，并搜索一些相关领域的知识。要做到 C10M，可以说系统网络内核、内存管理，都成为瓶颈了。所以要揭竿起义，统统推到重来。直接操作网卡绕过内核对网络的封装，直接使用用户态内存，再次绕过系统内核。

C10M 这个动作比较大，而且还需要特定的硬件型号支持（主要是网卡，网络处理嘛），所以目前这个项目进展还比较缓慢。不过对于有追求的人，可能就要两眼放光了。

前些日子 dpdk 组织国内 CDN 厂商开了一个小会，阿里的朋友说已经用这个开发出了 C10M 级别的产品。小伙伴们，你们怎么看？心动了，行动不？

# TIME\_WAIT 问题

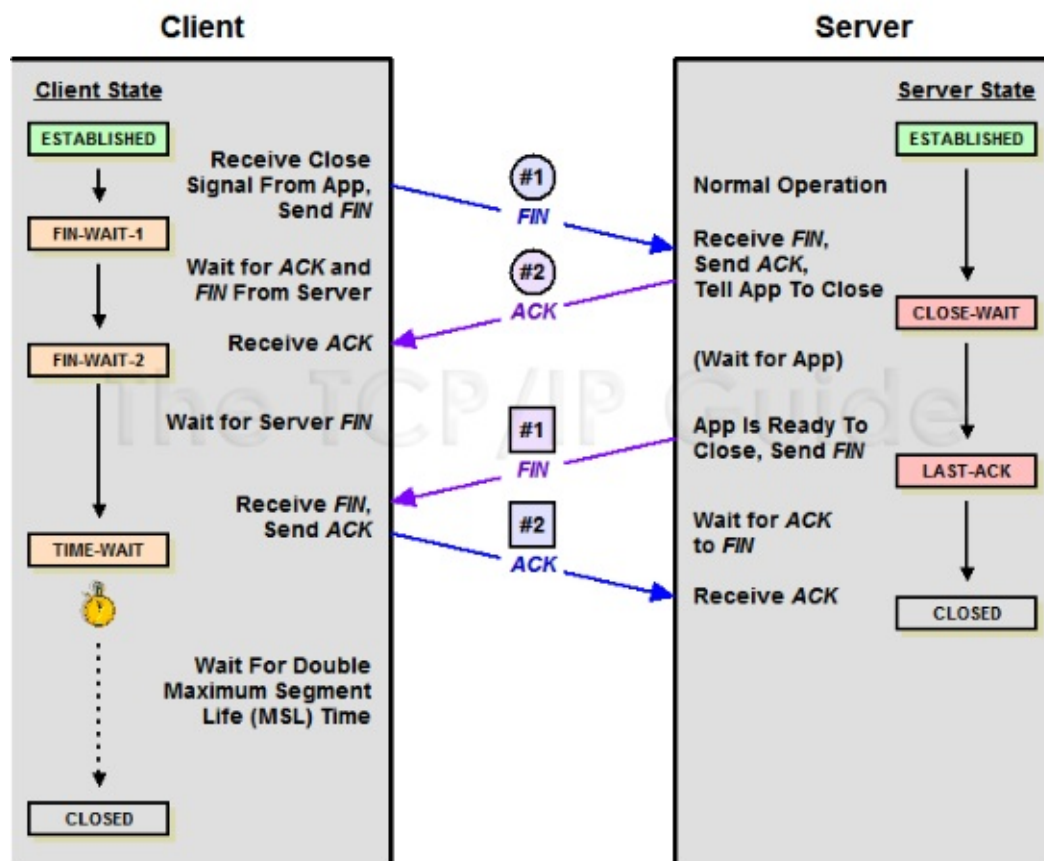
- `TIME_WAIT`
  - `HTTP 1.0` 协议
  - `user agent`

## TIME\_WAIT

这个是高并发服务端常见的一个问题，一般的做法是修改 `sysctl` 的参数来解决。

但是，做为一个有追求的程序员，你需要多问几个为什么，为什么会出现 `TIME_WAIT`？出现这个合理吗？

我们需要先回顾下 `tcp` 的知识，请看下面的状态转换图（图片来自「[The TCP/IP Guide](#)」）：



因为 `TCP` 连接是双向的，所以在关闭连接的时候，两个方向各自都需要关闭。

先发 `FIN` 包的一方执行的是主动关闭；后发 `FIN` 包的一方执行的是被动关闭。

主动关闭的一方会进入 `TIME_WAIT` 状态，并且在此状态停留两倍的 `MSL` 时长。

修改 `sysctl` 的参数，只是控制 `TIME_WAIT` 的数量。你需要很明确的知道，在你的应用场景里面，你预期是服务端还是客户端来主动关闭连接的。一般来说，都是客户端来主动关闭的。

Nginx 在某些情况下，会主动关闭客户端的请求，这个时候，返回值的 `connection` 为 `close`。我们看两个例子：

## HTTP 1.0 协议

请求包：

```
1. GET /hello HTTP/1.0
2. User-Agent: curl/7.37.1
3. Host: 127.0.0.1
4. Accept: */*
5. Accept-Encoding: deflate, gzip
```

应答包：

```
1. HTTP/1.1 200 OK
2. Date: Wed, 08 Jul 2015 02:53:54 GMT
3. Content-Type: text/plain
4. Connection: close
5.
6. hello world
```

对于 HTTP 1.0 协议，如果请求头里面没有包含 `connection`，那么应答默认是返回 `Connection: close`，也就是说 Nginx 会主动关闭连接。

## user agent

请求包：

```
1. POST /api/heartbeat.json HTTP/1.1
2.
3. Content-Type: application/x-www-form-urlencoded
4. Cache-Control: no-cache
5. User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT)
6. Accept-Encoding: gzip, deflate
7. Accept: */*
8. Connection: Keep-Alive
9. Content-Length: 0
```

应答包：

```
1. HTTP/1.1 200 OK
2. Date: Mon, 06 Jul 2015 09:35:34 GMT
```

```
3. Content-Type: text/plain
4. Transfer-Encoding: chunked
5. Connection: close
6. Content-Encoding: gzip
```

这个请求包是 HTTP 1.1 的协议，也声明了 Connection: Keep-Alive，为什么还会被 Nginx 主动关闭呢？

问题出在 **User-Agent**，Nginx 认为终端的浏览器版本太低，不支持 keep alive，所以直接 close 了。

在我们应用的场景下，终端不是通过浏览器而是后台请求的，而我们也无法控制终端的 User-Agent，那有什么方法不让 Nginx 主动去关闭连接呢？可以用 `keepalive_disable` 这个参数来解决。这个参数并不是字面的意思，用来关闭 keepalive，而是用来定义哪些古代的浏览器不支持 keepalive 的，默认值是 MSIE6。

```
1. keepalive_disable none;
```

修改为 none，就是认为不再通过 User-Agent 中的浏览器信息，来决定是否 keepalive。

注：本文内容参考了[火丁笔记](#)和[Nginx 开发从入门到精通](#)，感谢大牛分享。



## 与 Docker 使用的网络瓶颈

- 与 Docker 使用的网络瓶颈
  - None 模式
  - Bridge 模式
  - Host 模式
  - Container 模式

## 与 Docker 使用的网络瓶颈

Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）。几乎没有性能开销，可以很容易地在机器和数据中心中运行。最重要的是，他们不依赖于任何语言、框架包括系统。

Docker 自 2013 年以来非常火热，无论是从 GitHub 上的代码活跃度，还是 Redhat 在 RHEL6.5 中集成对 Docker 的支持，就连 Google 的 Compute Engine 也支持 Docker 在其之上运行。

笔者使用 Docker 的原因和目的，可能与其他公司不太一样。我们一直存在“分发”需求，Docker 主要是用来屏蔽企业用户平台的 inconsistency。我们的企业用户使用的系统比较杂，仅仅主流系统就有 Ubuntu, Centos, RedHat, AIX 还有一些定制裁减系统等，可谓百花齐放。

虽然 OpenResty 具有良好的跨平台特性，无奈我们的安全项目比较重，组件比较多，是不可能逐一适配不同平台的，工作量、稳定性等，难度和后期维护复杂度是难以想象的。如果您的应用和我们一样需要二次分发，非常建议考虑使用 Docker。这个年代是云的时代，二次分发其实成本很高，后期维护成本也很高，所以尽量做到云端。

说说 Docker 与 OpenResty 之间的“坑”吧，你们肯定对这个更感兴趣。

我们刚开始使用的时候，是这样启动的：

```
1. docker run -d -p 80:80 openresty
```

首次压测过程中发现 Docker 进程 CPU 占用率 100%，单机接口 4-5 万的 QPS 就上不去了。经过我们多方探讨交流，终于明白原来是网络瓶颈所致（OpenResty 太彪悍，Docker 默认的虚拟网卡受不了了 ^\_^）。

最终我们绕过这个默认的桥接网卡，使用 `--net` 参数即可完成。

```
1. docker run -d --net=host openresty
```

多么简单，就这么一个参数，居然困扰了我们好几天。一度怀疑我们是否要忍受引入 Docker 带来的低效率网络。所以有时候多出来交流、学习，真的可以让我们学到好多。虽然这个点是我们自己挖出来的，但是在交流过程中还学到了很多好东西。

Docker Network settings, 引自: <http://www.lupaworld.com/article-250439-1.html>

1. 默认情况下，所有的容器都开启了网络接口，同时可以接受任何外部的数据请求。
2. `--dns=[]` : Set custom dns servers for the container
3. `--net="bridge"` : Set the Network mode for the container
4. `'bridge'`: creates a new network stack for the container on the docker bridge
5. `'none'`: no networking for this container
6. `'container:<name|id>'`: reuses another container network stack
7. `'host'`: use the host network stack inside the container
8. `--add-host=""` : Add a line to /etc/hosts (host:IP)
9. `--mac-address=""` : Sets the container's Ethernet device's MAC address

你可以通过 `docker run --net none` 来关闭网络接口，此时将关闭所有网络数据的输入输出，你只能通过 STDIN、STDOUT 或者 files 来完成 I/O 操作。默认情况下，容器使用主机的 DNS 设置，你也可以通过 `--dns` 来覆盖容器内的 DNS 设置。同时 Docker 为容器默认生成一个 MAC

地址，你可以通过 `--mac-address 12:34:56 9a:bc` 来设置你自己的 MAC 地址。

Docker 支持的网络模式有：

- none。关闭容器内的网络连接
- bridge。通过 veth 接口来连接容器，默认配置。
- host。允许容器使用 host 的网络堆栈信息。注意：这种方式将允许容器访问 host 中类似 D-BUS 之类的系统服务，所以认为是不安全的。
- container。使用另外一个容器的网络堆栈信息。

## None 模式

将网络模式设置为 none 时，这个容器将不允许访问任何外部 router。这个容器内部只会有一个 loopback 接口，而且不存在任何可以访问外部网络的 router。

## Bridge 模式

Docker 默认会将容器设置为 bridge 模式。此时在主机上面将会存在一个 docker0 的网络接口，同时会针对容器创建一对 veth 接口。其中一个 veth 接口是在主机充当网卡桥接作用，另外一个 veth 接口存在于容器的命名空间中，并且指向容器的 loopback。Docker 会自动给这个容器分配一个 IP，并且将容器内的数据通过桥接转发到外部。

## Host 模式

当网络模式设置为 `host` 时，这个容器将完全共享 `host` 的网络堆栈。`host` 所有的网络接口将完全对容器开放。容器的主机名也会存在于主机的 `hostname` 中。这时，容器所有对外暴露的端口和对其它容器的连接，将完全失效。

## Container 模式

当网络模式设置为 `Container` 时，这个容器将完全复用另外一个容器的网络堆栈。同时使用时这个容器的名称必须要符合下面的格式：`--net container:.`

比如当前有一个绑定了本地地址 `localhost` 的 `Redis` 容器。如果另外一个容器需要复用这个网络堆栈，则需要如下操作：

1. `$ sudo docker run -d --name redis example/redis --bind 127.0.0.1`
2. `$ # use the redis container's network stack to access localhost`
3. `$ sudo docker run --rm -ti --net container:redis example/redis-cli -h 127.0.0.1`

# 火焰图

- 火焰图

# 火焰图

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。

# 什么是火焰图

- [什么是火焰图](#)

## 什么是火焰图

---

火焰图和直方图、曲线图一样，是一种分析数据的方式，它可以更直观、更形象地展示数据，让人很容易发现数据中的隐藏信息。之所以叫火焰图，是因为这种图很像一簇火焰。

火焰图展现的一般是从进程（或线程）的堆栈中采集来的数据，即函数之间的调用关系。从堆栈中采集数据有很多方式，下面是几种常见的采集工具：

- Performance Event
- SystemTap
- DTrace
- OProfile
- Gprof

数据采集到了，怎么分析它呢？为此，[Brendan Gregg](#)开发了专门把采样到的堆栈轨迹（Stack Trace）转化为直观图片显示的工具——[Flame Graph](#)，这样就很容易生成火焰图了。

可见，火焰图本身其实很简单，难的是从火焰图中发现问题，并且能够解释这种现象，从而找到优化系统或者解决问题的方法。

## 什么时候使用

- [什么时候使用](#)

## 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、吞吐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。

常用的火焰图有三种：

- [lj-lua-stacks.sxx](#) 用于绘制 Lua 代码的火焰图
- [sample-bt](#) 用于绘制 C 代码的火焰图
- [sample-bt-off-cpu](#) 用于绘制 C 代码执行过程中让出 CPU 的时间（阻塞时间）的火焰图

这三种火焰图的用法相似，输出格式一致，所以接下的章节中我们只介绍最为常用的 `lj-lua-stacks.sxx`。

# 如何安装火焰图生成工具

- 如何安装火焰图生成工具
  - 安装 SystemTap
  - 火焰图绘制

## 如何安装火焰图生成工具

### 安装 SystemTap

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。

在 CentOS 上的安装方法

首先需要安装当前内核版本对应的开发包和调试包（这一步非常重要并且最为繁琐）：

```
1. # #Installlaion:
2. # rpm -ivh kernel-debuginfo-common-$(uname -r).rpm
3. # rpm -ivh kernel-debuginfo-$(uname -r).rpm
4. # rpm -ivh kernel-devel-$(uname -r).rpm
```

这些 rpm 包可以在该网址中下载：

<http://debuginfo.centos.org>

安装 systemtap:

```
1. # yum install systemtap
2. # ...
3. # 测试systemtap安装成功否:
4. # stap -v -e 'probe vfs.read {printf("read performed\n"); exit()}'
5.
6. Pass 1: parsed user script and 103 library script(s) using 201628virt/29508res/3144shr/26860data
 kb, in 10usr/190sys/219real ms.
7. 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.
8. Pass 3: translated to C into "/tmp/stapffFP7E/stap_82c0f95e47d351a956e1587c4dd4cee1_1459_src.c"
 using 296120virt/125204res/4448shr/121352data kb, in 10usr/50sys/56real ms.
9. Pass 4: compiled C into "stap_82c0f95e47d351a956e1587c4dd4cee1_1459.ko" in
 620usr/620sys/1379real ms.
10. Pass 5: starting run.
11. read performed
12. Pass 5: run completed in 20usr/30sys/354real ms.
```

如果出现如上输出表示安装成功。

在 *Ubuntu* 上的安装方法

对于 *Ubuntu* 上的安装, 参考 *Ubuntu* 官方维护的一个 *wiki*:

<https://wiki.ubuntu.com/Kernel/Systemtap>

一般来说, 仅需引入 *ddeb* 源, 然后 `apt-get` 就能解决了。

由于 *systemtap* 需要依赖某些内核特性, 对于 *Ubuntu Gutsy* (或更老的版本), 必须重新编译内核。

编译的步骤参见 *systemtap* 的这篇 *wiki*:

<https://sourceware.org/systemtap/wiki/SystemtapOnUbuntu>

另外, 由于 *Ubuntu 16.04* 官方库里的 *systemtap* 版本过旧 (version 2.9), 从 `apt-get` 安装的 *systemtap* 有些情况下并不能正确地运行。

这时候需要从 *systemtap* 源码中编译出可用的 *systemtap*。

编译的过程参考 *systemtap* 的这篇文档:

[https://sourceware.org/git/?p=systemtap.git;a=blob\\_plain;f=README;hb=HEAD](https://sourceware.org/git/?p=systemtap.git;a=blob_plain;f=README;hb=HEAD)

大体上就这几步:

```
1. # 下载依赖.....
2. sudo apt install elfutils
3. sudo apt-get build-dep systemtap
4.
5. # 下载最新的版本
6. wget/git ...
7.
8. # 构建, 并祈祷能一次成功
9. ./configure
10. make all
11. [sudo] make install
```

## 火焰图绘制

首先, 需要下载 *stapxx* 工具包: [Github地址](#)。

该工具包中包含用 *perl* 写的, 会生成 *stap* 探测代码并运行的脚本。如果是要抓 *Lua* 级别的情况, 请使用其中的 *lj-lua-stacks.sxx*。

由于 *lj-lua-stacks.sxx* 输出的是文件绝对路径和行号, 要想匹配具体的 *Lua* 代码, 需要用 *fix-lua-bt* 进行转换。

```
1. # 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)
2. 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
```



```
3. hippo 15010 14857 0 Jul01 ? 00:00:12 nginx: worker process
4. # ./samples/lj-lua-stacks.sxx --arg time=5 --skip-badvars -x 15010 > tmp.bt (-x 是要抓的进程的
 pid, 探测结果输出到 tmp.bt)
5. # ./fix-lua-bt tmp.bt > flame.bt (处理 lj-lua-stacks.sxx 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

```
1. # stackcollapse-stap.pl flame.bt > flame.cbt
2. # 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。

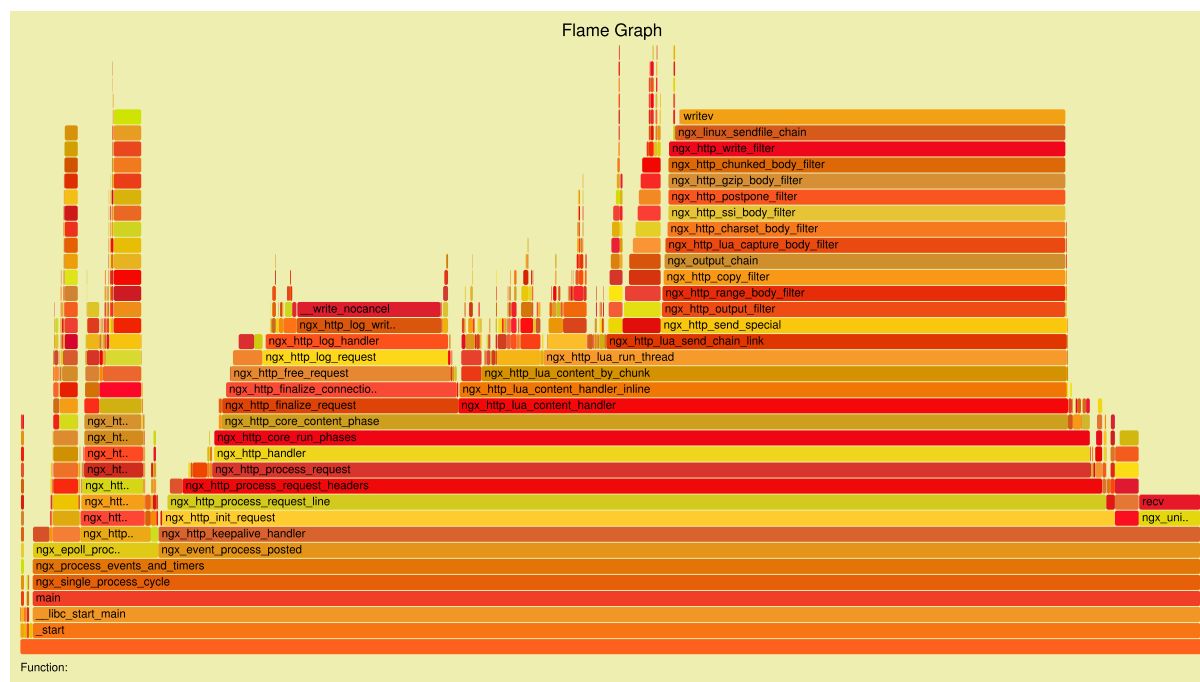
ps: 如果在执行 lj-lua-stacks.sxx 的时间周期内 (上面的命令是 5 秒), 抓取的 worker 没有任何业务在跑, 那么生成的火焰图便没有业务内容。为了让生成的火焰图更有代表性, 我们通常都会在抓取的同时进行压测。

# 如何定位问题

- [如何定位问题](#)

## 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 函数以正确的方式处理导致的，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
```

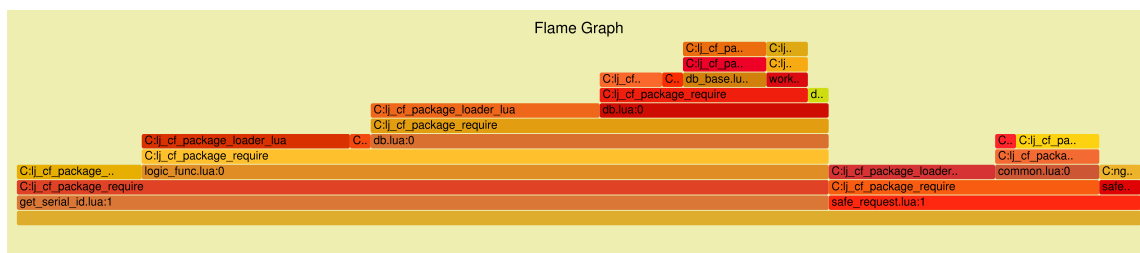
- 使用 `lj-lua-stacks.sxx` 抓取栈信息，并用 `fix-lua-bt` 工具处理：

```
1. # making the ./stap++ tool visible in PATH:
2. $ export PATH=$PWD:$PATH
3. # assuming the nginx worker process pid is 6949:
4. $./samples/lj-lua-stacks.sxx --arg time=5 --skip-badvars -x 6949 > tmp.bt
5. Start tracing 6949 (/opt/nginx/sbin/nginx)
6. Please wait for 5 seconds
7. $./fix-lua-bt tmp.bt > a.bt
```

- 使用 `stackcollapse-stap.pl` 和 `flamegraph.pl`：

```
./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”。