目錄

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！

我们团队在 OpenResty 社区的待做任务列表，欢迎大家提交自己的贡献！

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：

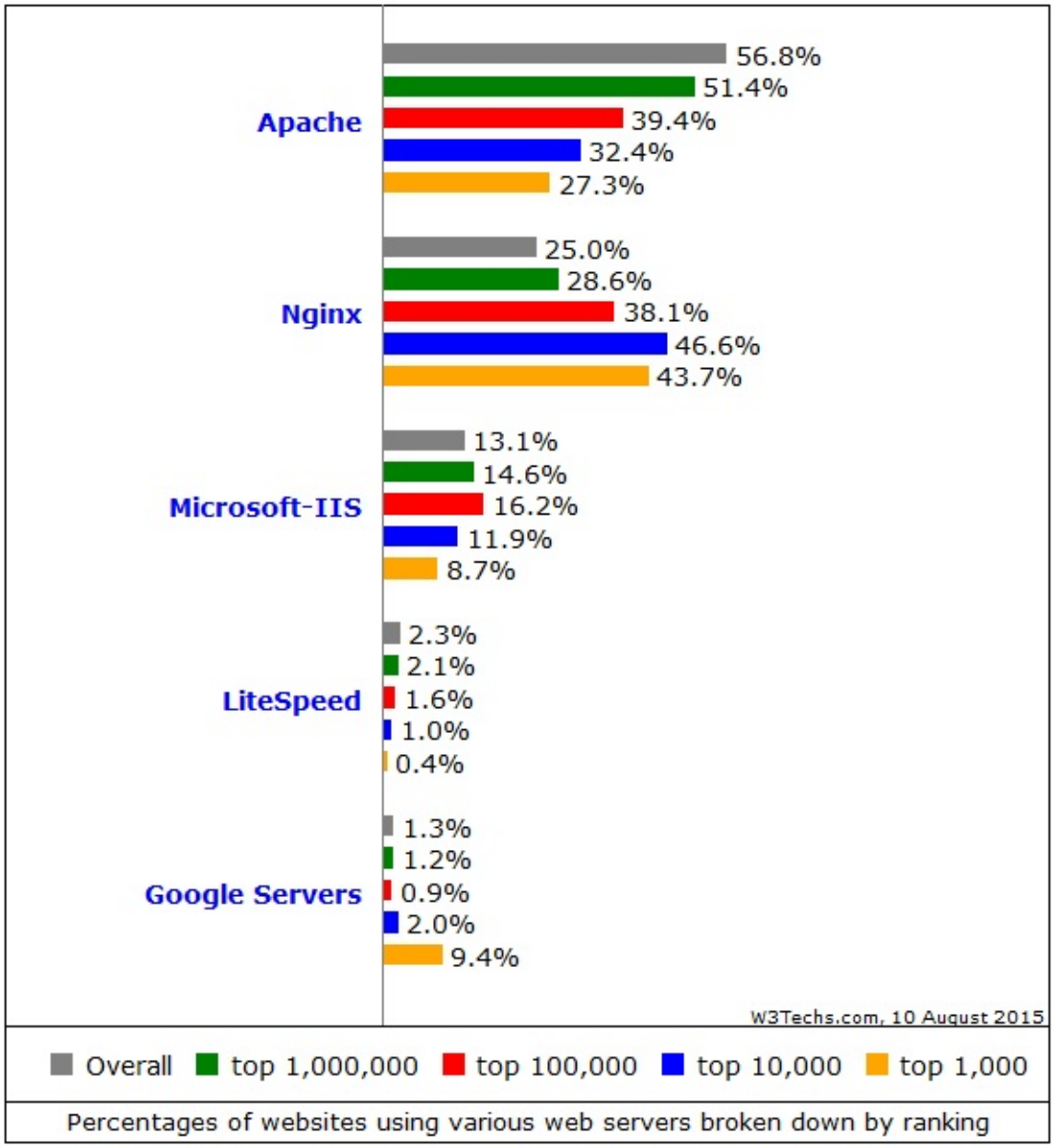
- 34782325（技术交流 ①群 已满）
- 481213820（技术交流 ②群 已满）
- 124613000（技术交流 ③群）

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 越来越收到欢迎。

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：

```
int select (int n, fd_set *readfds, fd_set *writefds,
        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

```
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。

```
struct pollfd {
    int fd; /* file descriptor */
    short events; /* requested events to watch */
    short revents; /* returned events witnessed */
};
```

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：

```
int epoll_create(int size)；
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)；
            typedef union epoll_data {
                void *ptr;
                int fd;
                __uint32_t u32;
                __uint64_t u64;
            } epoll_data_t;

            struct epoll_event {
                __uint32_t events;      /* Epoll events */
                epoll_data_t data;      /* User data variable */
            };

int epoll_wait(int epfd, struct epoll_event * events,
                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。

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 servie”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 驱动的纯 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很常见。

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 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

这一章我们简要地介绍 Lua 语言的基础知识，特别地，我们会有意将讨论放置于 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定语义;
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。

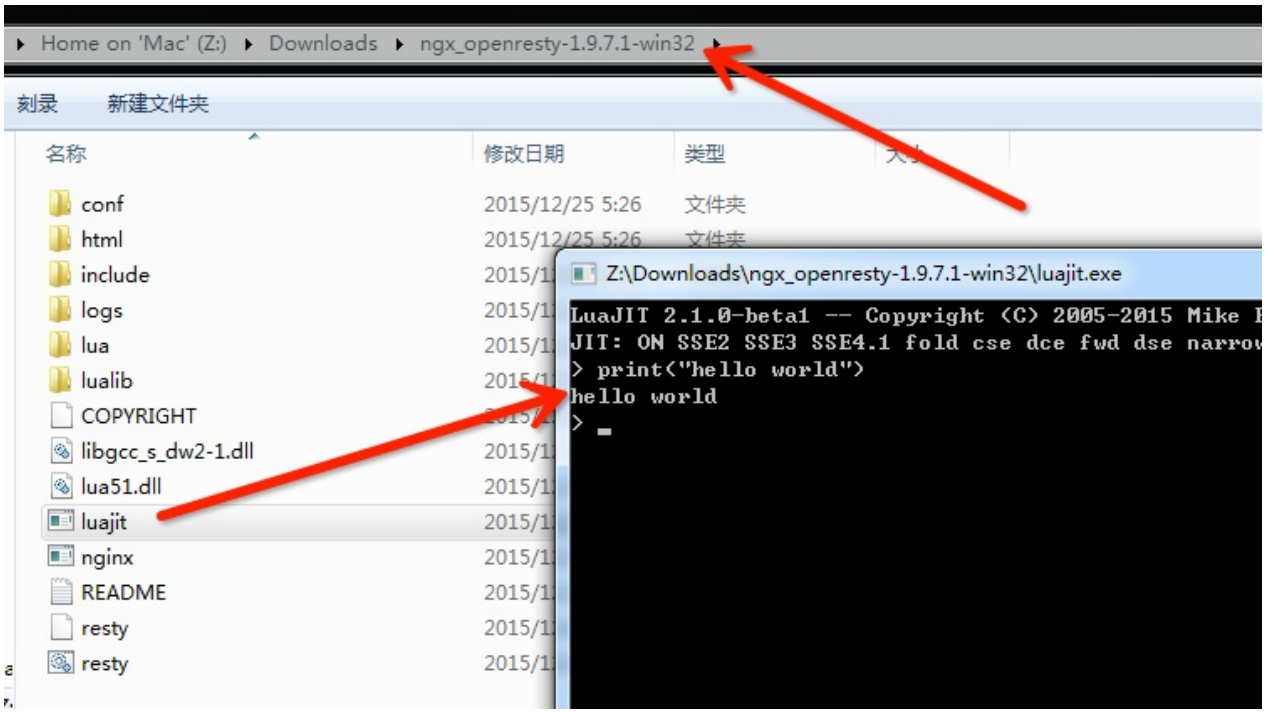
- Lua 官网链接：http://www.lua.org
- LuaJIT 官网链接：http://luajit.org

Lua 环境搭建

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。

```
# wget http://luajit.org/download/LuaJIT-2.1.0-beta1.tar.gz
# tar -xvf LuaJIT-2.1.0-beta1.tar.gz
# cd LuaJIT-2.1.0-beta1
# make
# 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

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

如果想了解其他系统安装 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。

```
# cat hello.lua
print("hello world")
# luajit hello.lua
hello world
```

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。

```
print(type("hello world")) -->output:string
print(type(print))         -->output:function
print(type(true))          -->output:boolean
print(type(360.0))         -->output:number
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。

```
local num
print(num)        -->output:nil

num = 100
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。

```
local a = true
local b = 0
local c = nil
if a then
    print("a")        -->output:a
else
    print("not a")    --这个没有执行
end

if b then
    print("b")        -->output:b
else
    print("not b")    --这个没有执行
end

if c then
    print("c")        --这个没有执行
else
    print("not c")    -->output:not c
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。

```
local order = 3.99
local score = 98.01
print(math.floor(order))   -->output:3
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

```
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。

```
local str1 = 'hello world'
local str2 = "hello lua"
local str3 = [["add\name",'hello']]
local str4 = [=[string have a [[]].]=]

print(str1)    -->output:hello world
print(str2)    -->output:hello lua
print(str3)    -->output:"add\name",'hello'
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。

```
local corp = {
    web = "www.google.com",    --索引为字符串，key = "web",
                               --            value = "www.google.com"
    telephone = "12345678",    --索引为字符串
    staff = {"Jack", "Scott", "Gary"}, --索引为字符串，值也是一个表
    100876,                --相当于 [1] = 100876，此时索引为数字
                           --      key = 1, value = 100876
    100191,                --相当于 [2] = 100191，此时索引为数字
    [10] = 360,            --直接把数字索引给出
    ["city"] = "Beijing" --索引为字符串
}

print(corp.web)               -->output:www.google.com
print(corp["telephone"])      -->output:12345678
print(corp[2])                -->output:100191
print(corp["city"])           -->output:"Beijing"
print(corp.staff[1])          -->output:Jack
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

```
local function foo()
    print("in the function")
    --dosomething()
    local x = 10
    local y = 20
    return x + y
end

local a = foo     --把函数赋给变量

print(a())

--output:
in the function
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

```
function foo()
end
```

等价于

```
foo = function ()
end
```

类似地，

```
local function foo()
end
```

等价于

```
local foo = function ()
end
```

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

```
print(1 + 2)        -->打印 3
print(5 / 10)       -->打印 0.5。 这是Lua不同于c语言的
print(5.0 / 10)     -->打印 0.5。 浮点数相除的结果是浮点数
-- print(10 / 0)    -->注意除数不能为0，计算的结果会出错
print(2 ^ 10)       -->打印 1024。 求2的10次方

local num = 1357
print(num % 2)        -->打印 1
print((num % 2) == 1) -->打印 true。 判断num是否为奇数
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

```
print(1 < 2)    -->打印 true
print(1 == 2)   -->打印 false
print(1 ~= 2)   -->打印 true
local a, b = true, false
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, userdate 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：

```
local a = { x = 1, y = 0}
local b = { x = 1, y = 0}
if a == b then
  print("a==b")
else
  print("a~=b")
end

---output:
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

```
local c = nil
local d = 0
local e = 100
print(c and d)  -->打印 nil
print(c and e)  -->打印 nil
print(d and e)  -->打印 100
print(c or d)   -->打印 0
print(c or e)   -->打印 100
print(not c)    -->打印 true
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。

```
print("Hello " .. "World")    -->打印 Hello World
print(0 .. 1)                 -->打印 01

str1 = string.format("%s-%s","hello","world")
print(str1)              -->打印 hello-world

str2 = string.format("%d-%s-%.2f",123,"world",1.21)
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：

```
local pieces = {}
for i, elem in ipairs(my_list) do
    pieces[i] = my_process(elem)
end
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：

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

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。

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

```
x = 10
if x > 0 then
    print("x is a positive number")
end
```

> 运行输出：x is a positive number

两个分支 if-else 型

```
x = 10
if x > 0 then
    print("x is a positive number")
else
    print("x is a non-positive number")
end
```

> 运行输出：x is a positive number

多个分支 if-elseif-else 型

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

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

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

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。

```
while 表达式 do
--body
end
```

> 示例代码，求 1 + 2 + 3 + 4 + 5 的结果

```
x = 1
sum = 0

while x <= 5 do
    sum = sum + x
    x = x + 1
end
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：

```
local t = {1, 3, 5, 8, 11, 18, 21}

local i
for i, v in ipairs(t) do
    if 11 == v then
        print("index[" .. i .. "] have right value[11]")
        break
    end
end
```

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：

```
x = 10
repeat
    print(x)
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 也可以在使用 break 退出。

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：

```
for var = begin, finish, step do
    --body
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，需要在外面控制，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

```
for i = 1, 5 do
  print(i)
end

-- output:
1
2
3
4
5
```

...

```
for i = 1, 10, 2 do
  print(i)
end

-- output:
1
3
5
7
9
```

> 以下是这种循环的一个典型示例：

```
for i = 10, 1, -1 do
  print(i)
end

-- output:
...
```

如果不想给循环设置上限的话，可以使用常量 math.huge：

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

```
-- 打印数组a的所有值
local a = {"a", "b", "c", "d"}
for i, v in ipairs(a) do
  print("index:", i, " value:", v)
end

-- output:
index:  1  value: a
index:  2  value: b
index:  3  value: c
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

```
-- 打印table t中所有的key
for k in pairs(t) do
    print(k)
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：

```
local days = {
  "Sunday", "Monday", "Tuesday", "Wednesday",
  "Thursday", "Friday", "Saturday"
}
```

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：

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

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：

```
local x = "Tuesday"
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：

```
local days = {
   "Monday", "Tuesday", "Wednesday", "Thursday",
   "Friday", "Saturday","Sunday"
}

local revDays = {}
for k, v in pairs(days) do
  revDays[v] = k
end

-- print value
for k,v in pairs(revDays) do
  print("k:", k, " v:", v)
end

-- output:
k:  Tuesday   v: 2
k:  Monday    v: 1
k:  Sunday    v: 7
k:  Thursday  v: 4
k:  Friday    v: 5
k:  Wednesday v: 3
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 关键字

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：

```
-- 计算最小的x,使从1到x的所有数相加和大于100
sum = 0
i = 1
while true do
    sum = sum + i
    if sum > 100 then
        break
    end
    i = i + 1
end
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：

```lua
local function add(x, y)
    return x + y
    --print("add: I will return the result " .. (x + y))
    --因为前面有个return，若不注释该语句，则会报错
end

local function is_positive(x)
    if x > 0 then
        return x .. " is positive"
    else
        return x .. " is non-positive"
    end

    --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
    --，但是不会被执行，此处不会产生输出
    print("function end!")
end

sum = add(10, 20)
print("The sum is " .. sum)  -->output:The sum is 30
answer = is_positive(-10)
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：

```lua
local function foo()
    print("before")
    do return end
    print("after")  -- 这一行语句永远不会执行到
end
```

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：

```
print("hello world!")         -- 用 print() 函数输出 hello world！
local m = math.max(1, 5)      -- 调用数学库函数 max，
                              -- 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。

函数定义

Lua 使用关键字 *function* 定义函数，语法如下：

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

```
function_name = function (arc)
  -- body
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：

```
local function function_name (arc)
  -- body
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：

```
local function max(a, b)  --定义函数 max，用来求两个数的最大值，并返回
   local temp = nil       --使用局部变量 temp，保存最大值
   if(a > b) then
      temp = a
   else
      temp = b
   end
   return temp            --返回最大值
end

local m = max(-12, 20)    --调用函数 max，找去 -12 和 20 中的最大值
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：

```
local function func()   --形参为空
    print("no parameter")
end

func()                  --函数调用，圆扩号不能省

--> output：
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

```
function foo.bar(a, b, c)
    -- body ...
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

```
foo.bar = function (a, b, c)
    print(a, b, c)
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

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：

```
local function swap(a, b) --定义函数swap,函数内部进行交换两个变量的值
   local temp = a
   a = b
   b = temp
   print(a, b)
end

local x = "hello"
local y = 20
print(x, y)
swap(x, y)     --调用swap函数
print(x, y)    --调用swap函数后，x和y的值并没有交换

-->output
hello 20
20  hello
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：

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

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
   print(a, b, c, d)
end

local x = 1
local y = 2
local z = 3

fun1(x, y, z)          -- z被函数fun1忽略掉了，参数变成 x, y
fun2(x, y, z)          -- 后面自动加上一个nil，参数变成 x, y, z, nil

-->output
1   2
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：

```
local function func( ... )                 -- 形参为 ... ,表示函数采用变长参数

   local temp = {...}                       -- 访问的时候也要使用 ...
   local ans = table.concat(temp, " ")      -- 使用 table.concat 库函数对数
                                            -- 组内容使用 " " 拼接成字符串。
   print(ans)
end

func(1, 2)         -- 传递了两个参数
func(1, 2, 3, 4)   -- 传递了四个参数

-->output
1 2

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：

```
local function change(arg) -- change 函数，改变长方形的长和宽，使其各增长一倍
  arg.width = arg.width * 2
  arg.height = arg.height * 2
  return arg
end

local rectangle = { width = 20, height = 15 }
print("before change:", "width  =", rectangle.width,
                        "height =", rectangle.height)
rectangle = change(rectangle)
print("after  change:", "width  =", rectangle.width,
                        "height =", rectangle.height)

-->output
before change: width = 20  height =  15
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：

```
function change(arg) --change函数，改变长方形的长和宽，使其各增长一倍
  arg.width = arg.width * 2  --表arg不是表rectangle的拷贝，他们是同一个表
  arg.height = arg.height * 2
end                   -- 没有return语句了

local rectangle = { width = 20, height = 15 }
print("before change:", "width = ", rectangle.width,
                        " height = ", rectangle.height)
change(rectangle)
print("after change:", "width = ", rectangle.width,
                       " height =", rectangle.height)

--> output
before change: width = 20  height = 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

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。

```
local s, e = string.find("hello world", "llo")
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

```
local function swap(a, b)    -- 定义函数 swap，实现两个变量交换值
   return b, a               -- 按相反顺序返回变量的值
end

local x = 1
local y = 20
x, y = swap(x, y)            -- 调用 swap 函数
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：

```
function init()              --init 函数 返回两个值 1 和 "lua"
  return 1, "lua"
end

x = init()
print(x)

x, y, z = init()
print(x, y, z)

--output
1
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：

```
local function init()       -- init 函数 返回两个值 1 和 "lua"
    return 1, "lua"
end

local x, y, z = init(), 2   -- init 函数的位置不在最后，此时只返回 1
print(x, y, z)              -->output  1  2  nil

local a, b, c = 2, init()   -- init 函数的位置在最后，此时返回 1 和 "lua"
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：

```
local function init()
    return 1, "lua"
end

print(init(), 2)   -->output  1  2
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

```
local function init()
    return 1, "lua"
end

print((init()), 2)   -->output  1  2
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

调用回调函数，并把一个数组参数作为回调函数的参数。

```
local args = {...} or {}
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

```
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

```
add_task(end_time, callback, params)

if os.time() >= endTime then
    callback(unpack(params, 1, table.maxn(params)))
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

```lua
local function run(x, y)
    print('run', x, y)
end

local function attack(targetId)
    print('targetId', targetId)
end

local function do_action(method, ...)
    local args = {...} or {}
    method(unpack(args, 1, table.maxn(args)))
end

do_action(run, 1, 2)          -- output: run 1 2
do_action(attack, 1111)       -- output: targetId    1111
```

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

```lua
local foo={}

local function getname()
    return "Lucy"
end

function foo.greeting()
    print("hello " .. getname())
end

return foo
```

> 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。

```lua
local fp = require("my")
fp.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 lua-releng 工具完成全局变量的检测，具体参考本章的 局部变量 一节。

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。这样数字符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

```
print(string.byte("abc", 1, 3))
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
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

-->output
97    98    99
99
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

```
print(string.char(96, 97, 98))
print(string.char())        -- 参数为空，默认是一个0，
                            -- 你可以用string.byte(string.char())测试一下
print(string.char(65, 66))

--> output
`ab

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

```
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

```
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

```
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 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

```
local find = string.find
print(find("abc cba", "ab"))
print(find("abc cba", "ab", 2)      -- 从索引为2的位置开始匹配字符串：ab
print(find("abc cba", "ba", -1))    -- 从索引为7的位置开始匹配字符串：ba
print(find("abc cba", "ba", -3))    -- 从索引为6的位置开始匹配字符串：ba
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
print(find("abc cba", "(%a+)", 1, true)) --从索引为1的位置开始匹配字符串：(%a+)

-->output
1   2
nil
nil
6   7
1   3   abc
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

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

-->output
3.1416
31 1f 37
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 索引处开始向后匹配字符串 p。

> 示例代码

```
print(string.match("hello lua", "lua"))
print(string.match("lua lua", "lua", 2))  --匹配后面那个lua
print(string.match("lua lua", "hello"))
print(string.match("today is 27/7/2015", "%d+/%d+/%d+"))

-->output
lua
lua
nil
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

```
s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do  --匹配最长连续且只含字母的字符串
    print(w)
end

-->output
hello
world
from
Lua


t = {}
s = "from=world, to=Lua"
for k, v in string.gmatch(s, "(%a+)=(%a+)") do  --匹配两个最长连续且只含字母的
    t[k] = v                                    --字符串，它们之间用等号连接
end
for k, v in pairs(t) do
print (k,v)
end

-->output
to      Lua
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

```
print(string.rep("abc", 3)) --拷贝3次"abc"

-->output  abcabcabc
```

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

```
print(string.sub("Hello Lua", 4, 7))
print(string.sub("Hello Lua", 2))
print(string.sub("Hello Lua", 2, 1))    --看到返回什么了吗
print(string.sub("Hello Lua", -3, -1))

-->output
lo L
ello Lua

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

```
print(string.gsub("Lua Lua Lua", "Lua", "hello"))
print(string.gsub("Lua Lua Lua", "Lua", "hello", 2)) --指明第四个参数

-->output
hello hello hello   3
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

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

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。

```
local color={first="red", "blue", third="green", "yellow"}
print(color["first"])                   --> output: red
print(color[1])                         --> output: blue
print(color["third"])                   --> output: green
print(color[2])                         --> output: yellow
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是 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。

```lua
local tblTest1 = { 1, a = 2, 3 }
print("Test1 " .. table.getn(tblTest1))

local tblTest2 = { 1, nil }
print("Test2 " .. table.getn(tblTest2))

local tblTest3 = { 1, nil, 2 }
print("Test3 " .. table.getn(tblTest3))

local tblTest4 = { 1, nil, 2, nil }
print("Test4 " .. table.getn(tblTest4))

local tblTest5 = { 1, nil, 2, nil, 3, nil }
print("Test5 " .. table.getn(tblTest5))

local tblTest6 = { 1, nil, 2, nil, 3, nil, 4, nil }
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：

```
# lua test.lua
Test1 2
Test2 1
Test3 3
Test4 1
Test5 3
Test6 1
# luajit test.lua
Test1 2
Test2 1
Test3 1
Test4 1
Test5 1
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

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

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

-->output
3    1    8
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

```
local a = {}
a[-1] = 10
print(table.maxn(a))
a[5] = 10
print(table.maxn(a))

-->output
0
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

```
local a = { 1, 2, 3, 4}
print(table.remove(a, 1)) --删除速索引为1的元素
print(a[1], a[2], a[3], a[4])

print(table.remove(a))   --删除最后一个元素
print(a[1], a[2], a[3], a[4])

-->output
1
2    3    4    nil
4
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

```
local function compare(x, y) --从大到小排序
   return x > y          --如果第一个参数大于第二个就返回true，否则返回false
end

local a = { 1, 7, 3, 4, 25}
table.sort(a)            --默认从小到大排序
print(a[1], a[2], a[3], a[4], a[5])
table.sort(a, compare) --使用比较函数进行排序
print(a[1], a[2], a[3], a[4], a[5])

-->output
1    3    4    7    25
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

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
year	四位数字
month	1--12
day	1--31
hour	0--23
min	0--59
sec	0--61
isdst	boolean（true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）

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

```
local day1 = { year = 2015, month = 7, day = 30 }
local t1 = os.time(day1)

local day2 = { year = 2015, month = 7, day = 31 }
local t2 = os.time(day2)
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：

```
local tab1 = os.date("*t")  --返回一个描述当前日期和时间的表
local ans1 = "{"
for k, v in pairs(tab1) do  --把tab1转换成一个字符串
    ans1 = string.format("%s %s = %s,", ans1, k, tostring(v))
end

ans1 = ans1 .. "}"
print("tab1 = ", ans1)


local tab2 = os.date("*t", 360)  --返回一个描述日期和时间数为360秒的表
local ans2 = "{"
for k, v in pairs(tab2) do      --把tab2转换成一个字符串
    ans2 = string.format("%s %s = %s,", ans2, k, tostring(v))
end

ans2 = ans2 .. "}"
print("tab2 = ", ans2)

-->output
tab1 = { hour = 17, min = 28, wday = 5, day = 30, month = 7, year = 2015, sec = 10, yd
ay = 211, isdst = false,}
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：

```
print(os.date("today is %A, in %B"))
print(os.date("now is %x %X"))

-->output
today is Thursday, in July
now is 07/30/15 17:39:22
```

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
math.rad(x)	角度x转换成弧度
math.deg(x)	弧度x转换成角度
math.max(x, ...)	返回参数中值最大的那个数，参数必须是number型
math.min(x, ...)	返回参数中值最小的那个数，参数必须是number型
math.random ([m [, n]])	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
math.randomseed (x)	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
math.abs(x)	返回x的绝对值
math.fmod(x, y)	返回 x对y取余数
math.pow(x, y)	返回x的y次方
math.sqrt(x)	返回x的算术平方根
math.exp(x)	返回自然数e的x次方
math.log(x)	返回x的自然对数
math.log10(x)	返回以10为底，x的对数
math.floor(x)	返回最大且不大于x的整数
math.ceil(x)	返回最小且不小于x的整数
math.pi	圆周率
math.sin(x)	求弧度x的正弦值
math.cos(x)	求弧度x的余弦值
math.tan(x)	求弧度x的正切值
math.asin(x)	求x的反正弦值
math.acos(x)	求x的反余弦值
math.atan(x)	求x的反正切值

> 示例代码：

```
print(math.pi)           -->output  3.1415926535898
print(math.rad(180))     -->output  3.1415926535898
print(math.deg(math.pi)) -->output  180

print(math.sin(1))       -->output  0.8414709848079
print(math.cos(math.pi)) -->output  -1
print(math.tan(math.pi / 4))  -->output  1

print(math.atan(1))      -->output  0.78539816339745
print(math.asin(0))      -->output  0

print(math.max(-1, 2, 0, 3.6, 9.1))     -->output  9.1
print(math.min(-1, 2, 0, 3.6, 9.1))     -->output  -1

print(math.fmod(10.1, 3))   -->output  1.1
print(math.sqrt(360))       -->output  18.97366596101

print(math.exp(1))          -->output  2.718281828459
print(math.log(10))         -->output  2.302585092994
print(math.log10(10))       -->output  1

print(math.floor(3.1415))  -->output  3
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，那么得得到的伪随机数序列是一样的。

> 示例代码：

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

稍等片刻，再次运行上面的代码。

```
math.randomseed (100) --把种子设置为100
print(math.random())          -->output  0.0012512588885159
print(math.random(100))       -->output  57
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：

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

稍等片刻，再次运行上面的代码。

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

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/0 操作部分相互独立，不要揉和在一起。

隐式文件描述

设置一个默认的输入或输出文件，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

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

repeat
    line = io.read()             -- 逐行读取内容，文件结束时返回nil
    if nil == line then
        break
    end
    print(line)
until (false)

io.close(file)                   -- 关闭文件

--> output
my test file
hello
lua
```

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

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

```
file = io.open("test2.txt", "r")     -- 使用 io.open() 函数，以只读模式打开文件

for line in file:lines() do          -- 使用 file:lines() 函数逐行读取文件
   print(line)
end

file:close()

-->output
my test2
hello lua
```

> 在 test2.txt 文件的最后添加一行 "hello world"

```
file = io.open("test2.txt", "a")  -- 使用 io.open() 函数，以添加模式打开文件
file:write("\nhello world")       -- 使用 file:write() 函数，在文件末尾追加内容
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nil。(默认)
number	读取指定字节数的字符。若为文件尾，则返回nil。如果number为0,则返回空字符串，若为文件尾,则返回nil

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
"set"	文件开始
"cur"	文件当前位置(默认)
"end"	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
"no"	没有缓冲，即直接输出
"full"	全缓冲，即当缓冲满后才进行输出操作(也可调用flush马上输出)
"line"	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。

元表

在 Lua 5.1 语言中，元表 *(metatable)* 的表现行为类似于 C++ 语言中的操作符重载，例如我们可以重载 "__add" 元方法 *(metamethod)*，来计算两个 Lua 数组的并集；或者重载 "__index" 方法，来定义我们自己的 Hash 函数。Lua 提供了两个十分重要的用来处理元表的方法，如下：

- setmetatable(table, metatable)：此方法用于为一个表设置元表。
- getmetatable(table)：此方法用于获取表的元表对象。

设置元表的方法很简单，如下：

```
local mytable = {}
local mymetatable = {}
setmetatable(mytable, mymetatable)
```

上面的代码可以简写成如下的一行代码：

```
local mytable = setmetatable({}, {})
```

修改表的操作符行为

通过重载 "__add" 元方法来计算集合的并集实例：

```lua
local set1 = {10, 20, 30}   -- 集合
local set2 = {20, 40, 50}   -- 集合

-- 将用于重载__add的函数，注意第一个参数是self
local union = function (self, another)
    local set = {}
    local result = {}

    -- 利用数组来确保集合的互异性
    for i, j in pairs(self) do set[j] = true end
    for i, j in pairs(another) do set[j] = true end

    -- 加入结果集合
    for i, j in pairs(set) do table.insert(result, i) end
    return result
end
setmetatable(set1, {__add = union}) -- 重载 set1 表的 __add 元方法

local set3 = set1 + set2
for _, j in pairs(set3) do
    io.write(j.." ")                -->output：30 50 20 40 10
end
```

除了加法可以被重载之外，Lua 提供的所有操作符都可以被重载：

元方法	含义
"__add"	+ 操作
"__sub"	- 操作 其行为类似于 "add" 操作
"__mul"	* 操作 其行为类似于 "add" 操作
"__div"	/ 操作 其行为类似于 "add" 操作
"__mod"	% 操作 其行为类似于 "add" 操作
"__pow"	^（幂）操作 其行为类似于 "add" 操作
"__unm"	一元 - 操作
"__concat"	..（字符串连接）操作
"__len"	# 操作
"__eq"	== 操作 函数 getcomphandler 定义了 Lua 怎样选择一个处理器来作比较操作 仅在两个对象类型相同且有对应操作相同的元方法时才起效
"__lt"	< 操作
"__le"	<= 操作

除了操作符之外，如下元方法也可以被重载，下面会依次解释使用方法：

元方法	含义
"__index"	取下标操作用于访问 table[key]
"__newindex"	赋值给指定下标 table[key] = value
"__tostring"	转换成字符串
"__call"	当 Lua 调用一个值时调用
"__mode"	用于弱表(*week table*)
"__metatable"	用于保护metatable不被访问

__index 元方法

下面的例子中，我们实现了在表中查找键不存在时转而在元表中查找该键的功能：

```
mytable = setmetatable({key1 = "value1"},    --原始表
  {__index = function(self, key)              --重载函数
    if key == "key2" then
      return "metatablevalue"
    end
  end
})

print(mytable.key1,mytable.key2)  --> output：value1 metatablevalue
```

关于 __index 元方法，有很多比较高阶的技巧，例如：__index 的元方法不需要非是一个函数，他也可以是一个表。

```
t = setmetatable({[1] = "hello"}, {__index = {[2] = "world"}})
print(t[1], t[2])   -->hello world
```

第一句代码有点绕，解释一下：先是把 {__index = {}} 作为元表，但 __index 接受一个表，而不是函数，这个表中包含 [2] = "world" 这个键值对。 所以当 t[2] 去在自身的表中找不到时，在 __index 的表中去寻找，然后找到了 [2] = "world" 这个键值对。

__index 元方法还可以实现给表中每一个值赋上默认值；和 __newindex 元方法联合监控对表的读取、修改等比较高阶的功能，待读者自己去开发吧。

__tostring 元方法

与 Java 中的 toString() 函数类似，可以实现自定义的字符串转换。

```
arr = {1, 2, 3, 4}
arr = setmetatable(arr, {__tostring = function (self)
    local result = '{'
    local sep = ''
    for _, i in pairs(self) do
        result = result ..sep .. i
        sep = ', '
    end
    result = result .. '}'
    return result
end})
print(arr)  --> {1, 2, 3, 4}
```

__call 元方法

__call 元方法的功能类似于 C++ 中的仿函数，使得普通的表也可以被调用。

```
functor = {}
function func1(self, arg)
  print ("called from", arg)
end

setmetatable(functor, {__call = func1})

functor("functor")  --> called from functor
print(functor)      --> output：0x00076fc8 （后面这串数字可能不一样）
```

__metatable 元方法

假如我们想保护我们的对象使其使用者既看不到也不能修改 metatables。我们可以对 metatable 设置了 __metatable 的值，getmetatable 将返回这个域的值，而调用 setmetatable 将会出错：

```
Object = setmetatable({}, {__metatable = "You cannot access here"})

print(getmetatable(Object)) --> You cannot access here
setmetatable(Object, {})    --> 引发编译器报错
```

Lua 面向对象编程

类

在 Lua 中，我们可以使用表和函数实现面向对象。将函数和相关的数据放置于同一个表中就形成了一个对象。

请看文件名为 `account.lua` 的源码：

```
local _M = {}

local mt = { __index = _M }

function _M.deposit (self, v)
    self.balance = self.balance + v
end

function _M.withdraw (self, v)
    if self.balance > v then
        self.balance = self.balance - v
    else
        error("insufficient funds")
    end
end

function _M.new (self, balance)
    balance = balance or 0
    return setmetatable({balance = balance}, mt)
end

return _M
```

> 引用代码示例：

```
local account = require("account")

local a = account:new()
a:deposit(100)

local b = account:new()
b:deposit(50)

print(a.balance)  --> output: 100
print(b.balance)  --> output: 50
```

上面这段代码 "setmetatable({balance = balance}, mt)"，其中 mt 代表 `{ __index = _M }`，这句话值得注意。根据我们在元表这一章学到的知识，我们明白，setmetatable 将 `_M` 作为新建表的原型，所以在自己的表内找不到 'deposit'、'withdraw' 这些方法和变量的时候，便会到 __index 所指定的 _M 类型中去寻找。

继承

继承可以用元表实现，它提供了在父类中查找存在的方法和变量的机制。在 Lua 中是不推荐使用继承方式完成构造的，这样做引入的问题可能比解决的问题要多，下面一个是字符串操作类库，给大家演示一下。

```
---------- s_base.lua
local _M = {}

local mt = { __index = _M }

function _M.upper (s)
    return string.upper(s)
end

return _M

---------- s_more.lua
local s_base = require("s_base")

local _M = {}
_M = setmetatable(_M, { __index = s_base })


function _M.lower (s)
    return string.lower(s)
end

return _M

---------- test.lua
local s_more = require("s_more")

print(s_more.upper("Hello"))   -- output: HELLO
print(s_more.lower("Hello"))   -- output: hello
```

成员私有性

在动态语言中引入成员私有性并没有太大的必要，反而会显著增加运行时的开销，毕竟这种检查无法像许多静态语言那样在编译期完成。下面的技巧把对象作为各方法的 upvalue，本身是很巧妙的，但会让子类继承变得困难，同时构造函数动态创建了函数，会导致构造函数无法被 JIT 编译。

在 Lua 中，成员的私有性，使用类似于函数闭包的形式来实现。在我们之前的银行账户的例子中，我们使用一个工厂方法来创建新的账户实例，通过工厂方法对外提供的闭包来暴露对外接口。而不想暴露在外的例如 balance 成员变量，则被很好的隐藏起来。

```
function newAccount (initialBalance)
    local self = {balance = initialBalance}
    local withdraw = function (v)
        self.balance = self.balance - v
    end
    local deposit = function (v)
        self.balance = self.balance + v
    end
    local getBalance = function () return self.balance end
    return {
        withdraw = withdraw,
        deposit = deposit,
        getBalance = getBalance
    }
end

a = newAccount(100)
a.deposit(100)
print(a.getBalance()) --> 200
print(a.balance)      --> nil
```

局部变量

Lua 的设计有一点很奇怪，在一个 block 中的变量，如果之前没有定义过，那么认为它是一个全局变量，而不是这个 block 的局部变量。这一点和别的语言不同。容易造成不小心覆盖了全局同名变量的错误。

定义

Lua 中的局部变量要用 local 关键字来显式定义，不使用 local 显式定义的变量就是全局变量：

```
g_var = 1         -- global var
local l_var = 2   -- local var
```

作用域

局部变量的生命周期是有限的，它的作用域仅限于声明它的块（block）。一个块是一个控制结构的执行体、或者是一个函数的执行体再或者是一个程序块（chunk）。我们可以通过下面这个例子来理解一下局部变量作用域的问题：

> 示例代码test.lua

```
x = 10
local i = 1          -- 程序块中的局部变量 i

while i <=x do
  local x = i * 2    -- while 循环体中的局部变量 x
  print(x)           -- output： 2, 4, 6, 8, ...
  i = i + 1
end

if i > 20 then
  local x            -- then 中的局部变量 x
  x = 20
  print(x + 2)       -- 如果i > 20 将会打印 22，此处的 x 是局部变量
else
  print(x)           -- 打印 10，这里 x 是全局变量
end

print(x)             -- 打印 10
```

使用局部变量的好处

1. 局部变量可以避免因为命名问题污染了全局环境
2. local 变量的访问比全局变量更快
3. 由于局部变量出了作用域之后生命周期结束，这样可以被垃圾回收器及时释放

常见实现如：`local print = print`

在 Lua 中，应该尽量让定义变量的语句靠近使用变量的语句，这也可以被看做是一种良好的编程风格。在 C 这样的语言中，强制程序员在一个块（或一个过程）的起始处声明所有的局部变量，所以有些程序员认为在一个块的中间使用声明语句是一种不良好地习惯。实际上，在需要时才声明变量并且赋予有意义的初值，这样可以提高代码的可读性。对于程序员而言，相比在块中的任意位置顺手声明自己需要的变量，和必须跳到块的起始处声明，大家应该能掂量哪种做法比较方便了吧？

“尽量使用局部变量”是一种良好的编程风格。然而，初学者在使用 Lua 时，很容易忘记加上 local 来定义局部变量，这时变量就会自动变成全局变量，很可能导致程序出现意想不到的问题。那么我们怎么检测哪些变量是全局变量呢？我们如何防止全局变量导致的影响呢？下面给出一段代码，利用元表的方式来自动检查全局变量，并打印必要的调试信息：

检查模块的函数使用全局变量

> 把下面代码保存在 foo.lua 文件中。

```
local _M = { _VERSION = '0.01' }

function _M.add(a, b)      --两个number型变量相加
    return a + b
end

function _M.update_A()     --更新变量值
    A = 365
end

return _M
```

> 把下面代码保存在 use_foo.lua 文件中。该文件和 foo.lua 在相同目录。

```
A = 360      --定义全局变量
local foo = require("foo")

local b = foo.add(A, A)
print("b = ", b)

foo.update_A()
print("A = ", A)
```

输出结果:

```
#  luajit use_foo.lua
b =   720
A =   365
```

无论是做基础模块或是上层应用，肯定都不愿意存在这类灰色情况存在，因为他对我们系统的存在，带来很多不确定性，生产中我们是要尽力避免这种情况的出现。

Lua 上下文中应当严格避免使用自己定义的全局变量。可以使用一个 lua-releng 工具来扫描 Lua 代码，定位使用 Lua 全局变量的地方。lua-releng 的相关链接：https://github.com/openresty/lua-nginx-module#lua-variable-scope

如果使用 macOS 或者 Linux，可以使用下面命令安装 `lua-releng`：

```
curl -L https://github.com/openresty/openresty-devel-utils/raw/master/lua-releng > /us
r/local/bin/lua-releng
chmod +x /usr/local/bin/lua-releng
```

Windows 用户把 lua-releng 文件所在的目录的绝对路径添加进 PATH 环境变量。然后进入你自己的 Lua 文件所在的工作目录，得到如下结果：

```
#  lua-releng
foo.lua: 0.01 (0.01)
Checking use of Lua global variables in file foo.lua...
  op no.  line  instruction args  ; code
  2 [8] SETGLOBAL 0 -1  ; A
Checking line length exceeding 80...
WARNING: No "_VERSION" or "version" field found in `use_foo.lua`.
Checking use of Lua global variables in file use_foo.lua...
  op no.  line  instruction args  ; code
  2 [1] SETGLOBAL 0 -1  ; A
  7 [4] GETGLOBAL 2 -1  ; A
  8 [4] GETGLOBAL 3 -1  ; A
  18  [8] GETGLOBAL 4 -1  ; A
Checking line length exceeding 80...
```

结果显示： 在 foo.lua 文件中，第 8 行设置了一个全局变量 A ； 在 use_foo.lua 文件中，没有版本信息，并且第 1 行设置了一个全局变量 A ，第 4、8 行使用了全局变量 A 。

判断数组大小

table.getn(t) 等价于 #t 但计算的是数组元素，不包括 hash 键值。而且数组是以第一个 nil 元素来判断数组结束。`#` 只计算 array 的元素个数，它实际上调用了对象的 metatable 的 `__len` 函数。对于有 `__len` 方法的函数返回函数返回值，不然就返回数组成员数目。

Lua 中，数组的实现方式其实类似于 C++ 中的 map，对于数组中所有的值，都是以键值对的形式来存储（无论是显式还是隐式），*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。尤其需要注意的一点是：Lua 数组中允许 nil 值的存在，但是数组默认结束标志却是 nil。这类比于 C 语言中的字符串，字符串中允许 '\0' 存在，但当读到 '\0' 时，就认为字符串已经结束了。

初始化是例外，在 Lua 相关源码中，初始化数组时首先判断数组的长度，若长度大于 0 ，并且最后一个值不为 nil，返回包括 nil 的长度；若最后一个值为 nil，则返回截至第一个非 nil 值的长度。

注意：一定不要使用 `#` 操作符或 `table.getn` 来计算包含 nil 的数组长度，这是一个未定义的操作，不一定报错，但不能保证结果如你所想。如果你要删除一个数组中的元素，请使用 remove 函数，而不是用 nil 赋值。

```
-- test.lua
local tblTest1 = { 1, a = 2, 3 }
print("Test1 " .. #(tblTest1))

local tblTest2 = { 1, nil }
print("Test2 " .. #(tblTest2))

local tblTest3 = { 1, nil, 2 }
print("Test3 " .. #(tblTest3))

local tblTest4 = { 1, nil, 2, nil }
print("Test4 " .. #(tblTest4))

local tblTest5 = { 1, nil, 2, nil, 3, nil }
print("Test5 " .. #(tblTest5))

local tblTest6 = { 1, nil, 2, nil, 3, nil, 4, nil }
print("Test6 " .. #(tblTest6))
```

我们分别使用 Lua 和 LuaJIT 来执行一下：

```
➜ luajit test.lua
Test1 2
Test2 1
Test3 1
Test4 1
Test5 1
Test6 1

➜ lua test.lua
Test1 2
Test2 1
Test3 3
Test4 1
Test5 3
Test6 1
```

这一段的输出结果，就是这么 匪夷所思。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。

非空判断

大家在使用 Lua 的时候，一定会遇到不少和 nil 有关的坑吧。有时候不小心引用了一个没有赋值的变量，这时它的值默认为 nil。如果对一个 nil 进行索引的话，会导致异常。

如下：

```
local person = {name = "Bob", sex = "M"}

-- do something
person = nil
-- do something

print(person.name)
```

上面这个例子把 nil 的错误用法显而易见地展示出来，执行后，会提示下面的错误：

```
stdin:1:attempt to index global 'person' (a nil value)
stack traceback:
   stdin:1: in main chunk
   [C]: ?
```

然而，在实际的工程代码中，我们很难这么轻易地发现我们引用了 nil 变量。因此，在很多情况下我们在访问一些 table 型变量时，需要先判断该变量是否为 nil，例如将上面的代码改成：

```
local person = {name = "Bob", sex = "M"}

-- do something
person = nil
-- do something
if person ~= nil and person.name ~= nil then
  print(person.name)
else
  -- do something
end
```

对于简单类型的变量，我们可以用 *if (var == nil) then* 这样的简单句子来判断。但是对于 table 型的 Lua 对象，就不能这么简单判断它是否为空了。一个 table 型变量的值可能是 `{}`，这时它不等于 nil。我们来看下面这段代码：

```lua
local next = next
local a = {}
local b = {name = "Bob", sex = "Male"}
local c = {"Male", "Female"}
local d = nil

print(#a)
print(#b)
print(#c)
--print(#d)    -- error

if a == nil then
    print("a == nil")
end

if b == nil then
    print("b == nil")
end

if c == nil then
    print("c == nil")
end

if d== nil then
    print("d == nil")
end

if next(a) == nil then
    print("next(a) == nil")
end

if next(b) == nil then
    print("next(b) == nil")
end

if next(c) == nil then
    print("next(c) == nil")
end
```

返回的结果如下：

```
0
0
2
d == nil
next(a) == nil
```

因此，我们要判断一个 table 是否为 `{}`，不能采用 `#table == 0` 的方式来判断。可以用下面这样的方法来判断：

```
function isTableEmpty(t)
    return t == nil or next(t) == nil
end
```

注意：`next` 指令是不能被 LuaJIT 的 JIT 编译优化，并且 LuaJIT 貌似没有明确计划支持这个指令优化，在不是必须的情况下，尽量少用。

正则表达式

在 *OpenResty* 中，同时存在两套正则表达式规范：*Lua* 语言的规范和 `ngx.re.*` 的规范，即使您对 *Lua* 语言中的规范非常熟悉，我们仍不建议使用 *Lua* 中的正则表达式。一是因为 *Lua* 中正则表达式的性能并不如 `ngx.re.*` 中的正则表达式优秀；二是 *Lua* 中的正则表达式并不符合 *POSIX* 规范，而 `ngx.re.*` 中实现的是标准的 *POSIX* 规范，后者明显更具备通用性。

Lua 中的正则表达式与 Nginx 中的正则表达式相比，有 5% - 15% 的性能损失，而且 Lua 将表达式编译成 Pattern 之后，并不会将 Pattern 缓存，而是每此使用都重新编译一遍，潜在地降低了性能。`ngx.re.*` 中的正则表达式可以通过参数缓存编译过后的 Pattern，不会有类似的性能损失。

`ngx.re.*` 中的 `o` 选项，指明该参数，被编译的 Pattern 将会在工作进程中缓存，并且被当前工作进程的每次请求所共享。Pattern 缓存的上限值通过 `lua_regex_cache_max_entries` 来修改。

`ngx.re.*` 中的 `j` 选项，指明该参数，如果使用的 PCRE 库支持 JIT，OpenResty 会在编译 Pattern 时启用 JIT。启用 JIT 后正则匹配会有明显的性能提升。较新的平台，自带的 PCRE 库均支持 JIT。如果系统自带的 PCRE 库不支持 JIT，出于性能考虑，最好自己编译一份 libpcre.so，然后在编译 OpenResty 时链接过去。要想验证当前 PCRE 库是否支持 JIT，可以这么做

1. 编译 OpenResty 时在 `./configure` 中指定 `--with-debug` 选项
2. 在 `error_log` 指令中指定日志级别为 `debug`
3. 运行正则匹配代码，查看日志中是否有 `pcre JIT compiling result: 1`

即使运行在不支持 JIT 的 OpenResty 上，加上 `j` 选项也不会带来坏的影响。在 OpenResty 官方的 Lua 库中，正则匹配至少都会带上 `jo` 这两个选项。

```
location /test {
    content_by_lua_block {
        local regex = [[\d+]]

        -- 参数 "j" 启用 JIT 编译，参数 "o" 是开启缓存必须的
        local m = ngx.re.match("hello, 1234", regex, "jo")
        if m then
            ngx.say(m[0])
        else
            ngx.say("not matched!")
        end
    }
}
```

测试结果如下：

```
➜  ~ curl 127.0.0.1/test
1234
```

另外还可以试试引入 `lua-resty-core` 中的正则表达式 API。这么做需要在代码里加入 `require 'resty.core.regex'` 。 `lua-resty-core` 版本的 `ngx.re.*` ，是通过 FFI 而非 Lua/C API 来跟 OpenResty C 代码交互的。某些情况下，会带来明显的性能提升。

Lua 正则简单汇总

Lua 中正则表达式语法上最大的区别，*Lua* 使用 *'%'* 来进行转义，而其他语言的正则表达式使用 '\' 符号来进行转义。其次，*Lua* 中并不使用 *'?'* 来表示非贪婪匹配，而是定义了不同的字符来表示是否是贪婪匹配。定义如下：

符号	匹配次数	匹配模式
+	匹配前一字符 1 次或多次	非贪婪
*	匹配前一字符 0 次或多次	贪婪
-	匹配前一字符 0 次或多次	非贪婪
?	匹配前一字符 0 次或1次	仅用于此，不用于标识是否贪婪

符号	匹配模式
.	任意字符
%a	字母
%c	控制字符
%d	数字
%l	小写字母
%p	标点字符
%s	空白符
%u	大写字母
%w	字母和数字
%x	十六进制数字
%z	代表 0 的字符

- *string.find* 的基本应用是在目标串内搜索匹配指定的模式的串。函数如果找到匹配的串，就返回它的开始索引和结束索引，否则返回 *nil*。*find* 函数第三个参数是可选的：标示目标串中搜索的起始位置，例如当我们想实现一个迭代器时，可以传进上一次调用时的结

束索引，如果返回了一个 *nil* 值的话，说明查找结束了.

```
local s = "hello world"
local i, j = string.find(s, "hello")
print(i, j) --> 1 5
```

- *string.gmatch* 我们也可以使用返回迭代器的方式。

```
local s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do
    print(w)
end

-- output :
--    hello
--    world
--    from
--    Lua
```

- *string.gsub* 用来查找匹配模式的串，并将使用替换串其替换掉，但并不修改原字符串，而是返回一个修改后的字符串的副本，函数有目标串，模式串，替换串三个参数，使用范例如下：

```
local a = "Lua is cute"
local b = string.gsub(a, "cute", "great")
print(a) --> Lua is cute
print(b) --> Lua is great
```

- 还有一点值得注意的是，'%b' 用来匹配对称的字符，而不是一般正则表达式中的单词的开始、结束。'%b' 用来匹配对称的字符，而且采用贪婪匹配。常写为 '%bxy'，x 和 y 是任意两个不同的字符；x 作为 匹配的开始，y 作为匹配的结束。比如，'%b()' 匹配以 '(' 开始，以 ')' 结束的字符串：

```
print(string.gsub("a (enclosed (in) parentheses) line", "%b()", ""))

-- output: a  line 1
```

不用标准库

虚变量

当一个方法返回多个值时，有些返回值有时候用不到，要是声明很多变量来一一接收，显然不太合适（不是不能）。Lua 提供了一个虚变量(dummy variable)，以单个下划线（“_”）来命名，用它来丢弃不需要的数值，仅仅起到占位的作用。

> 看一段示例代码：

```
-- string.find (s,p) 从string 变量s的开头向后匹配 string
-- p，若匹配不成功，返回nil，若匹配成功，返回第一次匹配成功
-- 的起止下标。

local start, finish = string.find("hello", "he") --start 值为起始下标，finish
                                                  --值为结束下标
print ( start, finish )                           --输出 1   2

local start = string.find("hello", "he")       -- start值为起始下标
print ( start )                                -- 输出 1


local _,finish = string.find("hello", "he")    --采用虚变量（即下划线），接收起
                                               --始下标值，然后丢弃，finish接收
                                               --结束下标值
print ( finish )                               --输出 2
```

代码倒数第二行，定义了一个用 local 修饰的 `虚变量` （即 单个下划线）。使用这个虚变量接收 string.find() 第一个返回值，静默丢掉，这样就直接得到第二个返回值了。

虚变量不仅仅可以被用在返回值，还可以用在迭代等。

> 在for循环中的使用：

```
-- test.lua 文件
local t = {1, 3, 5}

print("all  data:")
for i,v in ipairs(t) do
    print(i,v)
end

print("")
print("part data:")
for _,v in ipairs(t) do
    print(v)
end
```

执行结果：

```
# luajit test.lua
all  data:
1   1
2   3
3   5

part data:
1
3
5
```

抵制使用 module() 定义模块

旧式的模块定义方式是通过 `module("filename"[,package.seeall])*` 来显式声明一个包，现在官方不推荐再使用这种方式。这种方式将会返回一个由 `filename` 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。

`module("filename", package.seeall)` 这种写法是不提倡的，官方给出了两点原因：

1. `package.seeall` 这种方式破坏了模块的高内聚，原本引入 "filename" 模块只想调用它的 *foobar()* 函数，但是它却可以读写全局属性，例如 `"filename.os"`。
2. `module` 函数压栈操作引发的副作用，污染了全局环境变量。例如 `module("filename")` 会创建一个 `filename` 的 `table`，并将这个 `table` 注入全局环境变量中，这样使得没有引用它的文件也能调用 `filename` 模块的方法。

比较推荐的模块定义方法是：

```
-- square.lua 长方形模块
local _M = {}           -- 局部的变量
_M._VERSION = '1.0'     -- 模块版本

local mt = { __index = _M }

function _M.new(self, width, height)
    return setmetatable({ width=width, height=height }, mt)
end

function _M.get_square(self)
    return self.width * self.height
end

function _M.get_circumference(self)
    return (self.width + self.height) * 2
end

return _M
```

> 引用示例代码：

```
local square = require "square"

local s1 = square:new(1, 2)
print(s1:get_square())          --output: 2
print(s1:get_circumference())   --output: 6
```

另一个跟 Lua 的 module 模块相关需要注意的点是，当 lua_code_cache on 开启时，require 加载的模块是会被缓存下来的，这样我们的模块就会以最高效的方式运行，直到被显式地调用如下语句（这里有点像模块卸载）：

```
package.loaded["square"] = nil
```

我们可以利用这个特性代码来做一些高阶玩法，比如代码热更新等。

调用代码前先定义函数

Lua 里面的函数必须放在调用的代码之前，下面的代码是一个常见的错误：

```
-- test.lua 文件
local i = 100
i = add_one(i)

function add_one(i)
    return i + 1
end
```

我们将得到如下错误：

```
# luajit test.lua
luajit: test.lua:2: attempt to call global 'add_one' (a nil value)
stack traceback:
    test.lua:2: in main chunk
    [C]: at 0x0100002150
```

为什么放在调用后面就找不到呢？原因是 Lua 里的 function 定义本质上是变量赋值，即

```
function foo() ... end
```

等价于

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

因此在函数定义之前使用函数相当于在变量赋值之前使用变量，Lua 世界对于没有赋值的变量，默认都是 nil，所以这里也就产生了一个 nil 的错误。

一般地，由于全局变量是每个请求的生命期，因此以此种方式定义的函数的生命期也是每个请求的。为了避免每个请求创建和销毁 Lua closure 的开销，建议将函数的定义都放置在自己的 Lua module 中，例如：

```
-- my_module.lua
local _M = {_VERSION = "0.1"}

function _M.foo()
    -- your code
    print("i'm foo")
end

return _M
```

然后，再在 `content_by_lua_file` 指向的 `.lua` 文件中调用它：

```
local my_module = require "my_module"
my_module.foo()
```

因为 Lua module 只会在第一次请求时加载一次（除非显式禁用了 `lua_code_cache` 配置指令），后续请求便可直接复用。

点号与冒号操作符的区别

> 看下面示例代码：

```
local str = "abcde"
print("case 1:", str:sub(1, 2))
print("case 2:", str.sub(str, 1, 2))
```

> 执行结果:

```
case 1: ab
case 2: ab
```

冒号操作会带入一个 `self` 参数，用来代表 `自己` 。而点号操作，只是 `内容` 的展开。

在函数定义时，使用冒号将默认接收一个 `self` 参数，而使用点号则需要显式传入 `self` 参数。

示例代码：

```
obj = { x = 20 }

function obj:fun1()
    print(self.x)
end
```

等价于

```
obj = { x = 20 }

function obj.fun1(self)
    print(self.x)
end
```

参见 官方文档 中的以下片段:

```
The colon syntax is used for defining methods, that is, functions that
have an implicit extra parameter self. Thus, the statement

     function t.a.b.c:f (params) body end

is syntactic sugar for

     t.a.b.c.f = function (self, params) body end
```

冒号的操作，只有当变量是类对象时才需要。有关如何使用 Lua 构造类，大家可参考相关章节。

module 是邪恶的

Lua 是所有脚本语言中最快、最简洁的，我们爱她的快、她的简洁，但是我们也不得不忍受因为这些快、简洁最后带来的一些弊端，我们来挨个数数 module 有多少“邪恶”的吧。

由于 `lua_code_cache off` 情况下，缓存的代码会伴随请求完结而释放。module 的最大好处缓存这时候是无法发挥的，所以本章的内容都是基于 `lua_code_cache on` 的情况下。

先看看下面代码：

```
local ngx_socket_tcp = ngx.socket.tcp           -- ①

local _M = { _VERSION = '0.06' }                -- ②
local mt = { __index = _M }                     -- ③

function _M.new(self)
    local sock, err = ngx_socket_tcp()          -- ④
    if not sock then
        return nil, err
    end
    return setmetatable({ sock = sock }, mt)    -- ⑤
end

function _M.set_timeout(self, timeout)
    local sock = self.sock
    if not sock then
        return nil, "not initialized"
    end

    return sock:settimeout(timeout)
end

-- ... 其他功能代码，这里简略

return _M
```

① 对于比较底层的模块，内部使用到的非本地函数，都需要 local 本地化，这样做的好处：

- 避免命名冲突：防止外部是 `require(...)` 的方法调用造成全局变量污染
- 访问局部变量的速度比全局变量更快、更快、更快（重要事情说三遍）

② 每个基础模块最好有自己 `_VERSION` 标识，方便后期利用 `_VERSION` 完成热代码部署等高级特性，也便于使用者对版本有整体意识。

③ 其实 `_M` 和 `mt` 对于不同的请求实例（require 方法得到的对象）是相同的，因为 module 会被缓存到全局环境中。所以在这个位置千万不要放单请求内个性信息，例如 ngx.ctx 等变量。

④ 这里需要实现的是给每个实例绑定不同的 tcp 对象，后面 setmetatable 确保了每个实例拥有自己的 socket 对象，所以必须放在 new 函数中。如果放在 ③ 的下面，那么这时候所有的不同实例内部将绑定了同一个 socket 对象。

```
?local mt = { __index = _M }                    -- ③
?local sock = ngx_socket_tcp()                   -- ④ 错误的
?
?function _M.new(self)
?    return setmetatable({ sock = sock }, mt)    -- ⑤
?end
```

⑤ Lua 的 module 有两种类型：支持面向对象痕迹可以保留私有属性；静态方法提供者，没有任何私有属性。真正起到区别作用的就是 setmetatable 函数，是否有自己的个性元表，最终导致两种不同的形态。

笔者写这章的时候，想起一个场景，我觉得两者之间重叠度很大。`不幸的婚姻有千万种，可幸福的婚姻只有一种`。糟糕的 module 有千万个错误，可好的 module 都一个样。我们真没必要尝试了解所有错误格式的不好，但是正确的格式就摆在那里，不懂就照搬，搬多了就有感觉了。起点的不同，可以让我们从一开始有正确的认知形态，少走弯路，多一些时间学习有价值的东西。

也许你要问，哪里有正确的 module 所有格式？先从 OpenResty 默认绑定的各种 lua-resty-* 代码开始熟悉吧，她就是我说的正确格式（注意：这里我用了一个女字旁的 她，看的出来我有多爱她了）。

FFI

`FFI` 库，是 `LuaJIT` 中最重要的一个扩展库。它允许从纯 `Lua` 代码调用外部 C 函数，使用 C 数据结构。有了它，就不用再像 `Lua` 标准 `math` 库一样，编写 `Lua` 扩展库。把开发者从开发 `Lua` 扩展 C 库（语言/功能绑定库）的繁重工作中释放出来。学习完本小节对开发纯 `ffi` 的库是有帮助的，像 lru-resty-lrucache 中的 `pureffi.lua`，这个纯 `ffi` 库非常高效地完成了 lru 缓存策略。

简单解释一下 Lua 扩展 C 库，对于那些能够被 Lua 调用的 C 函数来说，它的接口必须遵循 `Lua` 要求的形式，就是 `typedef int (*lua_CFunction)(lua_State* L)`，这个函数包含的参数是 `lua_State` 类型的指针 L。可以通过这个指针进一步获取通过 `Lua` 代码传入的参数。这个函数的返回值类型是一个整型，表示返回值的数量。需要注意的是，用 C 编写的函数无法把返回值返回给 `Lua` 代码，而是通过虚拟栈来传递 `Lua` 和 C 之间的调用参数和返回值。不仅在编程上开发效率变低，而且性能上比不上 `FFI` 库调用 C 函数。

`FFI` 库最大限度的省去了使用 C 手工编写繁重的 `Lua/C` 绑定的需要。不需要学习一门独立/额外的绑定语言——它解析普通 C 声明。这样可以从 C 头文件或参考手册中，直接剪切，粘贴。它的任务就是绑定很大的库，但不需要捣鼓脆弱的绑定生成器。

`FFI` 紧紧的整合进了 `LuaJIT`（几乎不可能作为一个独立的模块）。`JIT` 编译器在 C 数据结构上所产生的代码，等同于一个 C 编译器应该生产的代码。在 `JIT` 编译过的代码中，调用 C 函数，可以被内连处理，不同于基于 `Lua/C API` 函数调用。

ffi 库 词汇

noun	*Explanation*
cdecl	A definition of an abstract C type(actually, is a lua string)
ctype	C type object
cdata	C data object
ct	C type format, is a template object, may be cdecl, cdata, ctype
cb	callback object
VLA	An array of variable length
VLS	A structure of variable length

ffi.* API

功能：*Lua ffi* 库的 *API*，与 *LuaJIT* 不可分割。

毫无疑问，在 `lua` 文件中使用 `ffi` 库的时候，必须要有下面的一行。

```
local ffi = require "ffi"
```

ffi.cdef

语法：*ffi.cdef(def)*

功能：声明 *C* 函数或者 *C* 的数据结构，数据结构可以是结构体、枚举或者是联合体，函数可以是 *C* 标准函数，或者第三方库函数，也可以是自定义的函数，注意这里只是函数的声明，并不是函数的定义。声明的函数应该要和原来的函数保持一致。

```
ffi.cdef[[
typedef struct foo { int a, b; } foo_t;  /* Declare a struct and typedef.   */
int printf(const char *fmt, ...);        /* Declare a typical printf function. */
]]
```

注意：所有使用的库函数都要对其进行声明，这和我们写 *C* 语言时候引入 *.h* 头文件是一样的。

顺带一提的是，并不是所有的 C 标准函数都能满足我们的需求，那么如何使用 第三方库函数 或 自定义的函数 呢，这会稍微麻烦一点，不用担心，你可以很快学会。:) 首先创建一个 `myffi.c`，其内容是：

```
int add(int x, int y)
{
  return x + y;
}
```

接下来在 Linux 下生成动态链接库：

```
gcc -g -o libmyffi.so -fpic -shared myffi.c
```

为了方便我们测试，我们在 `LD_LIBRARY_PATH` 这个环境变量中加入了刚刚库所在的路径，因为编译器在查找动态库所在的路径的时候其中一个环节就是在 `LD_LIBRARY_PATH` 这个环境变量中的所有路径进行查找。命令如下所示。

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:your_lib_path
```

在 Lua 代码中要增加如下的行：

```
ffi.load(name [,global])
```

`ffi.load` 会通过给定的 `name` 加载动态库，返回一个绑定到这个库符号的新的 C 库命名空间，在 `POSIX` 系统中，如果 `global` 被设置为 `ture`，这个库符号被加载到一个全局命名空间。另外这个 `name` 可以是一个动态库的路径，那么会根据路径来查找，否则的话会在默认的搜索路径中去找动态库。在 `POSIX` 系统中，如果在 `name` 这个字段中没有写上点符号 `.`，那么 `.so` 将会被自动添加进去，例如 `ffi.load("z")` 会在默认的共享库搜寻路径中去查找 `libz.so`，在 `windows` 系统，如果没有包含点号，那么 `.dll` 会被自动加上。

下面看一个完整例子：

```
local ffi = require "ffi"
local myffi = ffi.load('myffi')

ffi.cdef[[
int add(int x, int y);   /* don't forget to declare */
]]

local res = myffi.add(1, 2)
print(res)  -- output: 3   Note: please use luajit to run this script.
```

除此之外，还能使用 `ffi.C` (调用 `ffi.cdef` 中声明的系统函数) 来直接调用 `add` 函数，记得要在 `ffi.load` 的时候加上参数 `true`，例如 `ffi.load('myffi', true)`。

完整的代码如下所示：

```
local ffi = require "ffi"
ffi.load('myffi',true)

ffi.cdef[[
int add(int x, int y);   /* don't forget to declare */
]]

local res = ffi.C.add(1, 2)
print(res)  -- output: 3   Note: please use luajit to run this script.
```

ffi.typeof

语法： *ctype = ffi.typeof(ct)*

功能： 创建一个 *ctype* 对象，会解析一个抽象的 *C* 类型定义。

```
local uintptr_t = ffi.typeof("uintptr_t")
local c_str_t = ffi.typeof("const char*")
local int_t = ffi.typeof("int")
local int_array_t = ffi.typeof("int[?]")
```

ffi.new

语法： *cdata = ffi.new(ct [,nelem] [,init...])*

功能： 开辟空间，第一个参数为 *ctype* 对象，*ctype* 对象最好通过 *ctype = ffi.typeof(ct)* 构建。

顺便一提，可能很多人会有疑问，到底 `ffi.new` 和 `ffi.C.malloc` 有什么区别呢？

如果使用 `ffi.new` 分配的 `cdata` 对象指向的内存块是由垃圾回收器 `LuaJIT GC` 自动管理的，所以不需要用户去释放内存。

如果使用 `ffi.C.malloc` 分配的空间便不再使用 `LuaJIT` 自己的分配器了，所以不是由 `LuaJIT GC` 来管理的，但是，要注意的是 `ffi.C.malloc` 返回的指针本身所对应的 `cdata` 对象还是由 `LuaJIT GC` 来管理的，也就是这个指针的 `cdata` 对象指向的是用 `ffi.C.malloc` 分配的内存空间。这个时候，你应该通过 `ffi.gc()` 函数在这个 C 指针的 `cdata` 对象上面注册自己的析构函数，这个析构函数里面你可以再调用 `ffi.C.free`，这样的话当 C 指针所对应的 `cdata` 对象被 `Luajit GC` 管理器垃圾回收时候，也会自动调用你注册的那个析构函数来执行 C 级别的内存释放。

请尽可能使用最新版本的 `Luajit`，`x86_64` 上由 `LuaJIT GC` 管理的内存已经由 `1G->2G`，虽然管理的内存变大了，但是如果要使用很大的内存，还是用 `ffi.C.malloc` 来分配会比较好，避免耗尽了 `LuaJIT GC` 管理内存的上限，不过还是建议不要一下子分配很大的内存。

```
local int_array_t = ffi.typeof("int[?]")
local bucket_v = ffi.new(int_array_t, bucket_sz)

local queue_arr_type = ffi.typeof("lrucache_pureffi_queue_t[?]")
local q = ffi.new(queue_arr_type, size + 1)
```

ffi.fill

语法： *ffi.fill(dst, len [,c])*

功能： 填充数据，此函数和 *memset(dst, c, len)* 类似，注意参数的顺序。

```
ffi.fill(self.bucket_v, ffi_sizeof(int_t, bucket_sz), 0)
ffi.fill(q, ffi_sizeof(queue_type, size + 1), 0)
```

ffi.cast

语法： *cdata = ffi.cast(ct, init)*

功能： 创建一个 *scalar cdata* 对象。

```
local c_str_t = ffi.typeof("const char*")
local c_str = ffi.cast(c_str_t, str)       -- 转换为指针地址

local uintptr_t = ffi.typeof("uintptr_t")
tonumber(ffi.cast(uintptr_t, c_str))       -- 转换为数字
```

cdata 对象的垃圾回收

所有由显式的 `ffi.new(), ffi.cast() etc.` 或者隐式的 `accessors` 所创建的 `cdata` 对象都是能被垃圾回收的，当他们被使用的时候，你需要确保有在 `Lua stack` ， `upvalue` ，或者 `Lua table` 上保留有对 `cdata` 对象的有效引用，一旦最后一个 `cdata` 对象的有效引用失效了，那么垃圾回收器将自动释放内存（在下一个 `GC` 周期结束时候）。另外如果你要分配一个 `cdata` 数组给一个指针的话，你必须保持这个持有这个数据的 `cdata` 对象活跃，下面给出一个官方的示例：

```
ffi.cdef[[
typedef struct { int *a; } foo_t;
]]

local s = ffi.new("foo_t", ffi.new("int[10]")) -- WRONG!

local a = ffi.new("int[10]") -- OK
local s = ffi.new("foo_t", a)
-- Now do something with 's', but keep 'a' alive until you're done.
```

相信看完上面的 `API` 你已经很累了，再坚持一下吧！休息几分钟后，让我们来看看下面对官方文档中的示例做剖析，希望能再加深你对 `ffi` 的理解。

调用 C 函数

真的很用容易去调用一个外部 C 库函数，示例代码：

```
local ffi = require("ffi")
ffi.cdef[[
int printf(const char *fmt, ...);
]]
ffi.C.printf("Hello %s!", "world")
```

以上操作步骤，如下：

1. 加载 `FFI` 库。
2. 为函数增加一个函数声明。这个包含在 `中括号` 对之间的部分，是标准 C 语法。
3. 调用命名的 C 函数——非常简单。

事实上，背后的实现远非如此简单：③ 使用标准 C 库的命名空间 `ffi.C` 。通过符号名 `printf` 索引这个命名空间，自动绑定标准 C 库。索引结果是一个特殊类型的对象，当被调用时，执行 `printf` 函数。传递给这个函数的参数，从 `Lua` 对象自动转换为相应的 C 类型。

再来一个源自官方的示例代码：

```
local ffi = require("ffi")
ffi.cdef[[
unsigned long compressBound(unsigned long sourceLen);
int compress2(uint8_t *dest, unsigned long *destLen,
        const uint8_t *source, unsigned long sourceLen, int level);
int uncompress(uint8_t *dest, unsigned long *destLen,
         const uint8_t *source, unsigned long sourceLen);
]]
local zlib = ffi.load(ffi.os == "Windows" and "zlib1" or "z")

local function compress(txt)
  local n = zlib.compressBound(#txt)
  local buf = ffi.new("uint8_t[?]", n)
  local buflen = ffi.new("unsigned long[1]", n)
  local res = zlib.compress2(buf, buflen, txt, #txt, 9)
  assert(res == 0)
  return ffi.string(buf, buflen[0])
end

local function uncompress(comp, n)
  local buf = ffi.new("uint8_t[?]", n)
  local buflen = ffi.new("unsigned long[1]", n)
  local res = zlib.uncompress(buf, buflen, comp, #comp)
  assert(res == 0)
  return ffi.string(buf, buflen[0])
end

-- Simple test code.
local txt = string.rep("abcd", 1000)
print("Uncompressed size: ", #txt)
local c = compress(txt)
print("Compressed size: ", #c)
local txt2 = uncompress(c, #txt)
assert(txt2 == txt)
```

解释一下这段代码。我们首先使用 `ffi.cdef` 声明了一些被 zlib 库提供的 C 函数。然后加载 zlib 共享库，在 Windows 系统上，则需要我们手动从网上下载 zlib1.dll 文件，而在 POSIX 系统上 libz 库一般都会被预安装。因为 `ffi.load` 函数会自动填补前缀和后缀，所以我们简单地使用 z 这个字母就可以加载了。我们检查 `ffi.os` ，以确保我们传递给 `ffi.load` 函数正确的名字。

一开始，压缩缓冲区的最大值被传递给 `compressBound` 函数，下一行代码分配了一个要压缩字符串长度的字节缓冲区。`[?]` 意味着他是一个变长数组。它的实际长度由 `ffi.new` 函数的第二个参数指定。

我们仔细审视一下 `compress2` 函数的声明就会发现，目标长度是用指针传递的！这是因为我们要传递进去缓冲区的最大值，并且得到缓冲区实际被使用的大小。

在 C 语言中，我们可以传递变量地址。但因为在 Lua 中并没有地址相关的操作符，所以我们使用只有一个元素的数组来代替。我们先用最大缓冲区大小初始化这唯一一个元素，接下来就是很直观地调用 `zlib.compress2` 函数了。使用 `ffi.string` 函数得到一个存储着压缩数据的 Lua 字符串，这个函数需要一个指向数据起始区的指针和实际长度。实际长度将会在 `buflen` 这个数组中返回。因为压缩数据并不包括原始字符串的长度，所以我们要显式地传递进去。

使用 C 数据结构

`cdata` 类型用来将任意 C 数据保存在 `Lua` 变量中。这个类型相当于一块原生的内存，除了赋值和相同性判断，Lua 没有为之预定义任何操作。然而，通过使用 `metatable`（元表），程序员可以为 `cdata` 自定义一组操作。`cdata` 不能在 `Lua` 中创建出来，也不能在 `Lua` 中修改。这样的操作只能通过 `C API`。这一点保证了宿主程序完全掌管其中的数据。

我们将 C 语言类型与 `metamethod`（元方法）关联起来，这个操作只用做一次。`ffi.metatype` 会返回一个该类型的构造函数。原始 C 类型也可以被用来创建数组，元方法会被自动地应用到每个元素。

尤其需要指出的是，`metatable` 与 C 类型的关联是永久的，而且不允许被修改，`__index` 元方法也是。

> 下面是一个使用 C 数据结构的实例

```
local ffi = require("ffi")
ffi.cdef[[
typedef struct { double x, y; } point_t;
]]

local point
local mt = {
  __add = function(a, b) return point(a.x+b.x, a.y+b.y) end,
  __len = function(a) return math.sqrt(a.x*a.x + a.y*a.y) end,
  __index = {
    area = function(a) return a.x*a.x + a.y*a.y end,
  },
}
point = ffi.metatype("point_t", mt)

local a = point(3, 4)
print(a.x, a.y)  --> 3  4
print(#a)        --> 5
print(a:area())  --> 25
local b = a + point(0.5, 8)
print(#b)        --> 12.5
```

> 附表：Lua 与 C 语言语法对应关系

Idiom	*C code*	*Lua code*
Pointer dereference	x = *p	x = p[0]
int *p	*p = y	p[0] = y
Pointer indexing	x = p[i]	x = p[i]
int i, *p	p[i+1] = y	p[i+1] = y
Array indexing	x = a[i]	x = a[i]
int i, a[]	a[i+1] = y	a[i+1] = y
struct/union dereference	x = s.field	x = s.field
struct foo s	s.field = y	s.field = y
struct/union pointer deref	x = sp->field	x = sp.field
struct foo *sp	sp->field = y	s.field = y
int i, *p	y = p - i	y = p - i
Pointer dereference	x = p1 - p2	x = p1 - p2
Array element pointer	x = &a[i]	x = a + i

内存问题

todo：介绍 FFI 就必须从必要的 C 基础，包括内存管理的细节，说起，同时也须介绍包括 Valgrind 在内的内存问题调试工具的细节（by agentzh），后面重点补充。

什么是 JIT？

自从 OpenResty 1.5.8.1 版本之后，默认捆绑的 Lua 解释器就被替换成了 LuaJIT，而不再是标准 Lua。单从名字上，我们就可以直接看到这个新的解释器多了一个 `JIT`，接下来我们就一起来聊聊 `JIT`。

先看一下 LuaJIT 官方的解释：LuaJIT is a Just-In-Time Compilerfor the Lua programming language。

LuaJIT 的运行时环境包括一个用手写汇编实现的 Lua 解释器和一个可以直接生成机器代码的 JIT 编译器。

Lua 代码在被执行之前总是会先被 lfn 成 LuaJIT 自己定义的字节码（Byte Code）。关于 LuaJIT 字节码的文档，可以参见：http://wiki.luajit.org/Bytecode-2.0（这个文档描述的是 LuaJIT 2.0 的字节码，不过 2.1 里面的变化并不算太大）。

一开始的时候，Lua 字节码总是被 LuaJIT 的解释器解释执行。LuaJIT 的解释器会在执行字节码时同时记录一些运行时的统计信息，比如每个 Lua 函数调用入口的实际运行次数，还有每个 Lua 循环的实际执行次数。当这些次数超过某个预设的阈值时，便认为对应的 Lua 函数入口或者对应的 Lua 循环足够的“热”，这时便会触发 JIT 编译器开始工作。

JIT 编译器会从热函数的入口或者热循环的某个位置开始尝试编译对应的 Lua 代码路径。编译的过程是把 LuaJIT 字节码先转换成 LuaJIT 自己定义的中间码（IR），然后再生成针对目标体系结构的机器码（比如 x86_64 指令组成的机器码）。

如果当前 Lua 代码路径上的所有的操作都可以被 JIT 编译器顺利编译，则这条编译过的代码路径便被称为一个“trace”，在物理上对应一个 `trace` 类型的 GC 对象（即参与 Lua GC 的对象）。

你可以通过 `ngx-lj-gc-objs` 工具看到指定的 Nginx worker 进程里所有 `trace` 对象的一些基本的统计信息，见 https://github.com/openresty/stapxx#ngx-lj-gc-objs

比如下面这一行 `ngx-lj-gc-objs` 工具的输出

```
102 trace objects: max=928, avg=337, min=160, sum=34468 (in bytes)
```

则表明当前进程内的 LuaJIT VM 里一共有 102 个 trace 类型的 GC 对象，其中最小的 trace 占用 160 个字节，最大的占用 928 个字节，平均大小是 337 字节，而所有 trace 的总大小是 34468 个字节。

LuaJIT 的 JIT 编译器的实现目前还不完整，有一些基本原语它还无法编译，比如 pairs() 函数、unpack() 函数、string.match() 函数、基于 lua_CFunction 实现的 Lua C 模块、FNEW 字节码，等等。所以当 JIT 编译器在当前代码路径上遇到了它不支持的操作，便会立即终止当前的 trace 编译过程（这被称为 trace abort），而重新退回到解释器模式。

JIT 编译器不支持的原语被称为 NYI（Not Yet Implemented）原语。比较完整的 NYI 列表在这篇文档里面：

```
http://wiki.luajit.org/NYI
```

所谓“让更多的 Lua 代码被 JIT 编译”，其实就是帮助更多的 Lua 代码路径能为 JIT 编译器所接受。这一般通过两种途径来实现：

1. 调整对应的 Lua 代码，避免使用 NYI 原语。
2. 增强 JIT 编译器，让越来越多的 NYI 原语能够被编译。

对于第 2 种方式，春哥一直在推动公司（CloudFlare）赞助 Mike Pall 的开发工作。不过有些原语因为本身的代价过高，而永远不会被编译，比如基于经典的 lua_CFunction 方式实现的 Lua C 模块（所以需要尽量通过 LuaJIT 的 FFI 来调用 C）。

而对于第 1 种方法，我们如何才能知道具体是哪一行 Lua 代码上的哪一个 NYI 原语终止了 trace 编译呢？答案很简单。就是使用 LuaJIT 安装自带的 jit.v 和 jit.dump 这两个 Lua 模块。这两个 Lua 模块会打印出 JIT 编译器工作的细节过程。

在 Nginx 的上下文中，我们可以在 nginx.conf 文件中的 http {} 配置块中添加下面这一段：

```
init_by_lua_block {
    local verbose = false
    if verbose then
        local dump = require "jit.dump"
        dump.on(nil, "/tmp/jit.log")
    else
        local v = require "jit.v"
        v.on("/tmp/jit.log")
    end

    require "resty.core"
}
```

那一行 require "resty.core" 倒并不是必需的，放在那里的主要目的是为了尽量避免使用 ngx_lua 模块自己的基于 lua_CFunction 的 Lua API，减少 NYI 原语。

在上面这段 Lua 代码中，当 verbose 变量为 false 时（默认就为 false 哈），我们使用 jit.v 模块打印出比较简略的流水信息到 /tmp/jit.log 文件中；而当 verbose 变量为 true 时，我们则使用 jit.dump 模块打印所有的细节信息，包括每个 trace 内部的字节码、IR 码和最终生成的机

器指令。

这里我们主要以 jit.v 模块为例。在启动 Nginx 之后，应当使用 ab 和 weighttp 这样的工具对相应的服务接口进行预热，以触发 LuaJIT 的 JIT 编译器开始工作（还记得刚才我们说的“热函数”和“热循环”吗？）。预热过程一般不用太久，跑个二三百个请求足矣。当然，压更多的请求也没关系。完事后，我们就可以检查 /tmp/jit.log 文件里面的输出了。

jit.v 模块的输出里如果有类似下面这种带编号的 TRACE 行，则指示成功编译了的 trace 对象，例如

[TRACE 6 shdict.lua:126 return]

这个 trace 对象编号为 6，对应的 Lua 代码路径是从 shdict.lua 文件的第 126 行开始的。

下面这样的也是成功编译了的 trace:

```
[TRACE  16 (15/1) waf-core.lua:419 -> 15]
```

这个 trace 编号为 16，是从 waf-core.lua 文件的第 419 行开始的，同时它和编号为 15 的 trace 联接了起来。

而下面这个例子则是被中断的 trace:

```
[TRACE --- waf-core.lua:455 -- NYI: FastFunc pairs at waf-core.lua:458]
```

上面这一行是说，这个 trace 是从 waf-core.lua 文件的第 455 行开始编译的，但当编译到 waf-core.lua 文件的第 458 行时，遇到了一个 NYI 原语编译不了，即 pairs() 这个内建函数，于是当前的 trace 编译过程被迫终止了。

类似的例子还有下面这些：

```
[TRACE --- exit.lua:27 -- NYI: FastFunc coroutine.yield at waf-core.lua:439]
[TRACE --- waf.lua:321 -- NYI: bytecode 51 at raven.lua:107]
```

上面第二行是因为操作码 51 的 LuaJIT 字节码也是 NYI 原语，编译不了。

那么我们如何知道 51 字节码究竟是啥呢？我们可以用 nginx-devel-utils 项目中的 ljbc.lua 脚本来取得 51 号字节码的名字：

```
$ /usr/local/openresty/luajit/bin/luajit-2.1.0-alpha ljbc.lua 51
opcode 51:
FNEW
```

我们看到原来是用来（动态）创建 Lua 函数的 FNEW 字节码。ljbc.lua 脚本的位置是

```
https://github.com/agentzh/nginx-devel-utils/blob/master/ljbc.lua
```

非常简单的一个脚本，就几行 Lua 代码。

这里需要提醒的是，不同版本的 LuaJIT 的字节码可能是不相同的，所以一定要使用和你的 Nginx 链接的同一个 LuaJIT 来运行这个 ljbc.lua 工具，否则有可能会得到错误的结果。

我们实际做个对比实验，看看 JIT 带来的好处：

```
➜ cat test.lua
local s = [[aaaaaabbbbbbbccccccccccddddddddddddeeeeeeeeeeee
fffffffffffffffggggggggggggggaaaaaaaaaaabbbbbbbbbbbbb
ccccccccccllllll]]

for i=1,10000 do
    for j=1,10000 do
        string.find(s, "ll", 1, true)
    end
end

➜ time luajit test.lua
5.19s user
0.03s system
96% cpu
5.392 total

➜  time lua test.lua
9.20s user
0.02s system
99% cpu
9.270 total
```

本例子可以看到效率相差大约 9.2/5.19 ≈ 1.77 倍，换句话说标准 Lua 需要 177% 的时间才能完成同样的工作。估计大家觉得这个还不过瘾，再看下面示例代码：

> 文件 test.lua：

```
local loop_count = tonumber(arg[1])
local fun_pair = "ipairs" == arg[2] and ipairs or pairs

local t = {}
for i=1,100 do
    t[i] = i
end

for i=1,loop_count do
    for j=1,1000 do
        for k,v in fun_pair(t) do
            --
        end
    end
end
```

执行参数	执行结果
time lua test.lua 1000 ipairs	3.96s user 0.02s system 98% cpu 4.039 total
time lua test.lua 1000 pairs	3.97s user 0.01s system 99% cpu 3.992 total
time luajit test.lua 1000 ipairs	0.10s user 0.00s system 95% cpu 0.113 total
time luajit test.lua 10000 ipairs	0.98s user 0.00s system 99% cpu 0.991 total
time luajit test.lua 1000 pairs	1.54s user 0.01s system 99% cpu 1.559 total

从这个执行结果中，大致可以总结出下面几个观点：

- 在标准 Lua 解释器中，使用 ipairs 或 pairs 没有区别；
- 对于 pairs 方式，LuaJIT 的性能大约是标准 Lua 的 4 倍；
- 对于 ipairs 方式，LuaJIT 的性能大约是标准 Lua 的 40 倍。

可以被 JIT 编译的元操作

下面给大家列一下截止到目前已经可以被 JIT 编译的元操作。 其他还有 IO、Bit、FFI、Coroutine、OS、Package、Debug、JIT 等分类，使用频率相对较低，这里就不罗列了，可以参考官网：http://wiki.luajit.org/NYI。

基础库的支持情况

函数	编译?	备注
assert	yes	
collectgarbage	no	
dofile	never	
error	never	
getfenv	2.1 partial	只有 getfenv(0) 能编译
getmetatable	yes	
ipairs	yes	
load	never	
loadfile	never	
loadstring	never	
next	no	
pairs	no	
pcall	yes	
print	no	
rawequal	yes	
rawget	yes	
rawlen (5.2)	yes	
rawset	yes	
select	partial	第一个参数是静态变量的时候可以编译
setfenv	no	
setmetatable	yes	
tonumber	partial	不能编译非10进制，非预期的异常输入
tostring	partial	只能编译：字符串、数字、布尔、nil 以及支持 __tostring 元方法的类型
type	yes	
unpack	no	
xpcall	yes	

字符串库

函数	编译?	备注
string.byte	yes	
string.char	2.1	
string.dump	never	
string.find	2.1 partial	只有字符串样式查找（没有样式）
string.format	2.1 partial	不支持 %p 或 非字符串参数的 %s
string.gmatch	no	
string.gsub	no	
string.len	yes	
string.lower	2.1	
string.match	no	
string.rep	2.1	
string.reverse	2.1	
string.sub	yes	
string.upper	2.1	

表

函数	编译?	备注
table.concat	2.1	
table.foreach	no	2.1: 内部编译，但还没有外放
table.foreachi	2.1	
table.getn	yes	
table.insert	partial	只有 push 操作
table.maxn	no	
table.pack (5.2)	no	
table.remove	2.1	部分，只有 pop 操作
table.sort	no	
table.unpack (5.2)	no	

math 库

函数	编译?	备注
math.abs	yes	
math.acos	yes	
math.asin	yes	
math.atan	yes	
math.atan2	yes	
math.ceil	yes	
math.cos	yes	
math.cosh	yes	
math.deg	yes	
math.exp	yes	
math.floor	yes	
math.fmod	no	
math.frexp	no	
math.ldexp	yes	
math.log	yes	
math.log10	yes	
math.max	yes	
math.min	yes	
math.modf	yes	
math.pow	yes	
math.rad	yes	
math.random	yes	
math.randomseed	no	
math.sin	yes	
math.sinh	yes	
math.sqrt	yes	
math.tan	yes	
math.tanh	yes	

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

为什选择 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：

```
ubuntu: /opt/nginx-1.7.7/conf$ tree |grep -v default
.
├── fastcgi.conf
├── fastcgi_params
├── koi-utf
├── koi-win
├── mime.types
├── nginx.conf
├── scgi_params
├── uwsgi_params
└── 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：

```
worker_process      # 表示工作进程的数量，一般设置为cpu的核数

worker_connections  # 表示每个工作进程的最大连接数

server{}            # 块定义了虚拟主机

    listen          # 监听端口

    server_name     # 监听域名

    location {}     # 是用来为匹配的 URI 进行配置，URI 即语法中的“/uri/”

    location /{}    # 匹配任何查询，因为所有请求都以 / 开头

        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
                    # /opt/nginx-1.7.7/html/

        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
                    # 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：

```
ubuntu:/opt/nginx-1.7.7/html$ ls
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

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：

```
location = / {
   echo "规则A";
}
location = /login {
   echo "规则B";
}
location ^~ /static/ {
   echo "规则C";
}
location ^~ /static/files {
    echo "规则X";
}
location ~ \.(gif|jpg|png|js|css)$ {
   echo "规则D";
}
location ~* \.png$ {
   echo "规则E";
}
location /img {
    echo "规则Y";
}
location / {
   echo "规则F";
}
```

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：

```
# 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。
# 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
# 第一个必选规则
location = / {
    proxy_pass http://tomcat:8080/index
}

# 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
# 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
location ^~ /static/ {
    root /webroot/static/;
}
location ~* \.(gif|jpg|jpeg|png|css|js|ico)$ {
    root /webroot/res/;
}

# 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
# 非静态文件请求就默认是动态请求，自己根据实际把握
# 毕竟目前的一些框架的流行，带.php、.jsp后缀的情况很少了
location / {
    proxy_pass http://tomcat:8080/
}
```

rewrite 语法

- last – 基本上都用这个 Flag
- break – 中止 Rewirte，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：

```
-f 和 !-f 用来判断是否存在文件
-d 和 !-d 用来判断是否存在目录
-e 和 !-e 用来判断是否存在文件或目录
-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

```
例：http://localhost:88/test1/test2/test.php?k=v
$host：localhost
$server_port：88
$request_uri：/test1/test2/test.php?k=v
$document_uri：/test1/test2/test.php
$document_root：D:\nginx/html
$request_filename：D:\nginx/html/test1/test2/test.php
```

redirect 语法

```
server {
    listen 80;
    server_name start.igrow.cn;
    index index.html index.php;
    root html;
    if ($http_host !~ "^star\.igrow\.cn$") {
        rewrite ^(.*) http://star.igrow.cn$1 redirect;
    }
}
```

防盗链

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

根据文件类型设置过期时间

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

禁止访问某个目录

```
location ~* \.(txt|doc)${
    root /data/www/wwwroot/linuxtone/test;
    deny all;
}
```

一些可用的全局变量，可以参考获取 Nginx 内置绑定变量章节。

if 是邪恶的

当在 location 区块中使用 if 指令的时候会有一些问题，在某些情况下它并不按照你的预期运行而是做一些完全不同的事情。而在另一些情况下他甚至会出现段错误。一般来说避免使用 if 指令是个好主意。

在 location 区块里 if 指令下唯一 100% 安全的指令应该只有：

> return …; rewrite … last;

除此以外的指令都可能导致不可预期的行为，包括诡异的发出段错误信号 (SIGSEGV)。

要着重注意的是 if 的行为不是反复无常的，给出两个条件完全一致的请求，Nginx 并不会出现一个正常工作而一个请求失败的随机情况，在明晰的测试和理解下 if 是完全可用的。尽管如此，在这里还是建议使用其他指令。

总有一些情况你无法避免去使用 if 指令，比如你需要测试一个变量，而它没有相应的配置指令。

```
if ($request_method = POST) {
    return 405;
}
if ($args ~ post=140){
    rewrite ^ http://example.com/ permanent;
}
```

如何替换掉 if

使用 try_files 如果他适合你的需求。在其他的情况下使用 `return …` 或者 `rewrite … last`。还有一些情况可能要把 if 移动到 server 区块下(只有当其他的 rewrite 模块指令也允许放在的地方才是安全的)。

如下可以安全地改变用于处理请求的 location。

```
location / {
    error_page 418 = @other;
    recursive_error_pages on;

    if ($something) {
        return 418;
    }

    # some configuration
    # ...
}

location @other {
    # some other configuration
    # ...
}
```

在某些情况下使用嵌入脚本模块(嵌入 perl 或者其他一些第三方模块)处理这些脚本更佳。

以下是一些例子用来解释为什么 if 是邪恶的。非专业人士, 请勿模仿!

```
# 这里收集了一些出人意料的坑爹配置来展示 location 中的 if 指令是万恶的

# 只有第二个 header 才会在响应中展示
# 这不是 Bug, 只是他的处理流程如此

location /only-one-if {
    set $true 1;

    if ($true) {
        add_header X-First 1;
    }

    if ($true) {
        add_header X-Second 2;
    }

    return 204;
}

# 因为 if, 请求会在未改变 uri 的情况下下发送到后台的 '/'

location /proxy-pass-uri {
    proxy_pass http://127.0.0.1:8080/;

    set $true 1;

    if ($true) {
        # nothing
    }
}
```

```
# 因为if, try_files 失效

location /if-try-files {
    try_files  /file  @fallback;

    set $true 1;

    if ($true) {
        # nothing
    }
}

# nginx 将会发出段错误信号(SIGSEGV)

location /crash {

    set $true 1;

    if ($true) {
        # fastcgi_pass here
        fastcgi_pass  127.0.0.1:9000;
    }

    if ($true) {
        # no handler here
    }
}

# alias with captures isn't correcly inherited into implicit nested location created b
y if
# alias with captures 不能正确的继承到由if创建的隐式嵌入的location

location ~* ^/if-and-alias/(?<file>.*) {
    alias /tmp/$file;

    set $true 1;

    if ($true) {
        # nothing
    }
}
```

为什么会这样且到现在都没修复这些问题?

if 指令是 rewrite 模块中的一部分, 是实时生效的指令。另一方面来说, Nginx 配置大体上是陈述式的。在某些时候用户出于特殊是需求的尝试, 会在 if 里写入一些非 rewrite 指令, 这直接导致了我们现处的情况。大多数情况下他可以工作, 但是...看看上面。 看起来唯一正确的修复方式是完全禁用 if 中的非 rewrite 指令。但是这将破坏很多现有可用的配置, 所以还没有修复。

如果你还是想知道该如何使用 if

如果你看完了上面所有内容还是想使用 if ，请确认你确实理解了该怎么用它。一些比较基本的用法可以在这里找到。

做适当的测试

我已经警告过你了!

> 文章选自：http://xwsoul.com/posts/761 TODO:这个文章后面需要自己翻译，可能有版权问题：https://www.nginx.com/resources/wiki/start/topics/depth/ifisevil/

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：

```
server {
        listen       80;
        server_name www.test.com;
        charset utf-8;
        root   /data/www.test.com;
        index  index.html index.htm;
       }
```

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。

```
http {
    # 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，
    # 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。
    open_file_cache max=204800 inactive=20s;

    # open_file_cache 指令中的inactive 参数时间内文件的最少使用次数，
    # 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
    # 文件在inactive 时间内一次没被使用，它将被移除。
    open_file_cache_min_uses 1;

    # 这个是指多长时间检查一次缓存的有效信息
    open_file_cache_valid 30s;

    # 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
    # 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 ，
    # 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。
    gzip on;
    gzip_min_length 1k;
    gzip_buffers 4 16k;
    gzip_http_version 1.0;
    gzip_comp_level 2;
    gzip_types text/plain application/x-javascript text/css application/xml;

    server {
            listen       80;
            server_name www.test.com;
            charset utf-8;
            root   /data/www.test.com;
            index  index.html index.htm;
           }
}
```

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）和服务器上的负载平衡和反向代理。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平衡器、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。

```
proxy_cache_path/path/to/cache levels=1:2 keys_zone=my_cache:10m max_size=10g inactive
=60m
use_temp_path=off;

server {
    ...
    location / {
        proxy_cache my_cache;
        proxy_pass http://my_upstream;
    }

}
```

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。当缓存达到这个上线，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。我们强烈建议你将参数设置为 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：

```
location / {
    ...
    proxy_cache_use_stale error timeout http_500 http_502 http_503 http_504;
}
```

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：

```
proxy_cache_path /path/to/cache levels=1:2 keys_zone=my_cache:10m max_size=10g inactiv
e=60m
use_temp_path=off;
server {
    ...
    location / {
        proxy_cache my_cache;
        proxy_cache_revalidate on;
        proxy_cache_min_uses 3;
        proxy_cache_use_stale error timeout updating http_500 http_502 http_503 http_5
04;
        proxy_cache_lock on;
        proxy_pass http://my_upstream;
    }
}
```

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不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：

```
proxy_cache_path /path/to/hdd1 levels=1:2 keys_zone=my_cache_hdd1:10m max_size=10g

inactive=60m use_temp_path=off;
proxy_cache_path /path/to/hdd2 levels=1:2 keys_zone=my_cache_hdd2:10m max_size=10g ina
ctive=60m use_temp_path=off;
split_clients $request_uri $my_cache {
    50% "my_cache_hdd1";
    50% "my_cache_hdd2";
}

server {
    ...
    location / {
        proxy_cache $my_cache;
        proxy_pass http://my_upstream;
    }
}
```

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

```
log_format combined '$remote_addr - $remote_user  [$time_local]  '
                    ' "$request"  $status  $body_bytes_sent  '
                    ' "$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

```
log_format myformat '$remote_addr  $status  $time_local';
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
$remote_addr与$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，从左至右，日志详细程度逐级递减，即 debug 最详细，crit 最少，默认为 crit。

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

```
error_log /dev/null;

http {
    # ...
}
```

> Windows 系统把存储位置设置为空设备

```
error_log nul;

http {
    # ...
}
```

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。

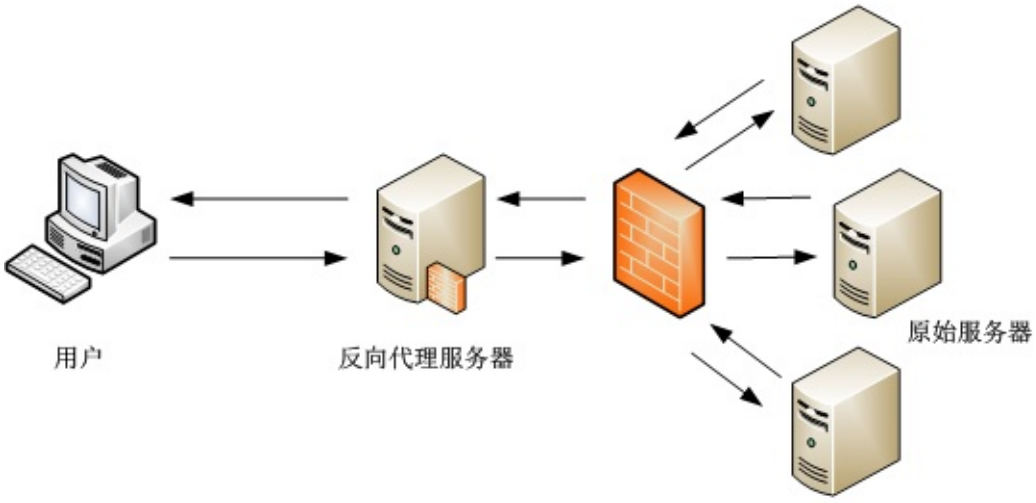
反向代理

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 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 获取 readme 页面一样。这里所提到的 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：

```
worker_processes 1;

pid logs/nginx.pid;
error_log logs/error.log warn;

events {
    worker_connections 3000;
}

http {
    include mime.types;
    server_tokens off;

    ## 下面配置反向代理的参数
    server {
        listen    80;

        ## 1. 用户访问 http://ip:port，则反向代理到 https://github.com
        location / {
            proxy_pass  https://github.com;
            proxy_redirect     off;
            proxy_set_header   Host             $host;
            proxy_set_header   X-Real-IP        $remote_addr;
            proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
        }

        ## 2.用户访问 http://ip:port/README.md，则反向代理到
        ##   https://github.com/.../README.md
        location /README.md {
            proxy_set_header  X-Real-IP  $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_pass https://github.com/moonbingbing/openresty-best-practices/blob/m
aster/README.md;
        }
    }
}
```

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本书的 URL 匹配章节。

(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（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：

```
upstream test.net{
    ip_hash;
    server 192.168.10.13:80;
    server 192.168.10.14:80  down;
    server 192.168.10.15:8009  max_fails=3  fail_timeout=20s;
    server 192.168.10.16:8080;
}
server {
    location / {
        proxy_pass  http://test.net;
    }
}
```

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)。

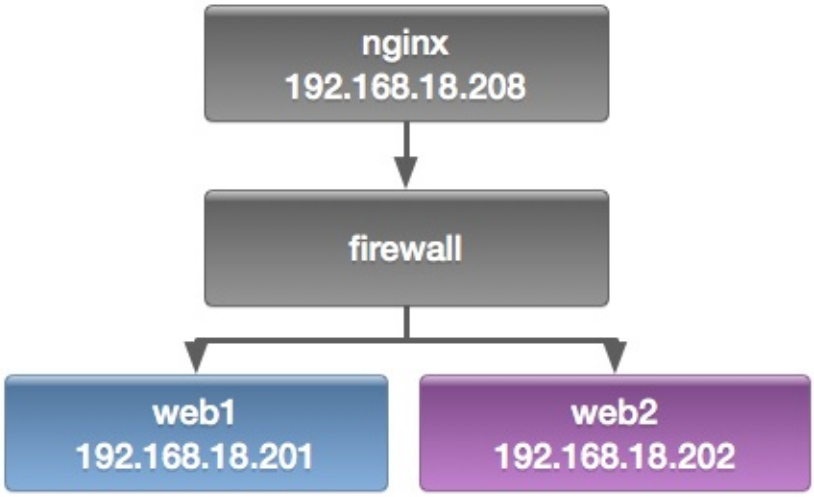
upstream 支持的状态参数

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

```
upstream webservers {
    server 192.168.18.201 weight=1;
    server 192.168.18.202 weight=1;
}
server {
    listen       80;
    server_name  localhost;
    #charset koi8-r;
    #access_log  logs/host.access.log  main;
    location / {
        proxy_pass      http://webservers;
        proxy_set_header  X-Real-IP  $remote_addr;
    }
}
```

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

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

```
# curl http://192.168.18.208
web1.test.com
# curl http://192.168.18.208
web2.test.com
# curl http://192.168.18.208
web1.test.com
# curl http://192.168.18.208
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:

```
# tail /var/log/nginx/access_log
192.168.18.138 - - [04/Sep/2013:09:41:58 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:41:58 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:41:59 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:41:59 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:42:00 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:42:00 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:42:00 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:44:21 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:44:22 +0800] "GET / HTTP/1.0" 200 23 "-"
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。

```
# LogFormat "%{X-Real-IP}i %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\""
combined
# tail /var/log/nginx/access_log
192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:50:28 +0800] "GET / HTTP/1.0" 200 23 "-"
192.168.18.138 - - [04/Sep/2013:09:50:29 +0800] "GET / HTTP/1.0" 200 23 "-"
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：

```
upstream webservers {
    server 192.168.18.201 weight=1 max_fails=2 fail_timeout=2;
    server 192.168.18.202 weight=1 max_fails=2 fail_timeout=2;
}
```

重新加载一下配置文件:

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

先停止 Web1，进行测试：

```
# service nginx stop
停止 nginx：                                                  [确定]
```

```
# curl http://192.168.18.208
web2.test.com
# curl http://192.168.18.208
web2.test.com
# curl http://192.168.18.208
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。

```
# service nginx start
正在启动 nginx：                                           [确定]
```

```
# curl http://192.168.18.208
web1.test.com
# curl http://192.168.18.208
web2.test.com
# curl http://192.168.18.208
web1.test.com
# curl http://192.168.18.208
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：

```
server {
    listen 8080;
    server_name localhost;
    root /data/www/errorpage;
    index index.html;
}
```

> index.html 文件内容：

```
# cat index.html
<h1>Sorry......</h1>
```

> 负载均衡配置：

```
upstream webservers {
    server 192.168.18.201 weight=1 max_fails=2 fail_timeout=2;
    server 192.168.18.202 weight=1 max_fails=2 fail_timeout=2;
    server 127.0.0.1:8080 backup;
}
```

重新加载配置文件：

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

关闭 Web 服务器并进行测试：

```
# service nginx stop
停止 nginx：                                               [确定]
```

进行测试：

```
# curl http://192.168.18.208
<h1>Sorry......</h1>
# curl http://192.168.18.208
<h1>Sorry......</h1>
# curl http://192.168.18.208
<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。

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

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，所以了不能配置 backup 服务器。

重新加载一下服务器：

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

测试一下：

```
# curl http://192.168.18.208
web2.test.com
# curl http://192.168.18.208
web2.test.com
# curl http://192.168.18.208
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。

Nginx 陷阱和常见错误

翻译自：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：

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

把 root 放在 location 区块内

糟糕的配置：

```
server {
    server_name www.example.com;
    location / {
        root /var/www/Nginx -default/;
        # [...]
      }
    location /foo {
        root /var/www/Nginx -default/;
        # [...]
    }
    location /bar {
        root /var/www/Nginx -default/;
        # [...]
    }
}
```

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：

```
server {
    server_name www.example.com;
    root /var/www/Nginx -default/;
    location / {
        # [...]
    }
    location /foo {
        # [...]
    }
    location /bar {
        # [...]
    }
}
```

重复的 index 指令

糟糕的配置：

```
http {
    index index.php index.htm index.html;
    server {
        server_name www.example.com;
        location / {
            index index.php index.htm index.html;
            # [...]
        }
    }
    server {
        server_name example.com;
        location / {
            index index.php index.htm index.html;
            # [...]
        }
        location /foo {
            index index.php;
            # [...]
        }
    }
}
```

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：

```
http {
    index index.php index.htm index.html;
    server {
        server_name www.example.com;
        location / {
            # [...]
        }
    }
    server {
        server_name example.com;
        location / {
            # [...]
        }
        location /foo {
            # [...]
        }
    }
}
```

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看看邪恶的 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：

```
server {
    server_name example.com *.example.com;
        if ($host ~* ^www\.(.+)) {
            set $raw_domain $1;
            rewrite ^/(.*)$ $raw_domain/$1 permanent;
        }
        # [...]
    }
}
```

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：

```
 server {
    server_name www.example.com;
    return 301 $scheme://example.com$request_uri;
}
server {
    server_name example.com;
    # [...]
}
```

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：

```
server {
    root /var/www/example.com;
    location / {
        if (!-f $request_filename) {
            break;
        }
    }
}
```

推荐的配置：

```
server {
    root /var/www/example.com;
    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

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。

在上面配置的例子里面，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的消除 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：

```
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：

```
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：

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

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`。这会让 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

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

- 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

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

- 使用 *try_files* 指令过滤出文件不存在的情况

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

- 使用嵌套的 location 过滤出文件不存在的情况

```
location ~* \.php$ {
    location ~ \..*/.*\.php$ {return 404;}
    fastcgi_pass backend;
    # [...]
}
```

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：

```
fastcgi_param  SCRIPT_FILENAME    $document_root$fastcgi_script_name;
```

糟糕的配置：

```
fastcgi_param  SCRIPT_FILENAME    /var/www/yoursite.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：

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

好点儿的配置：

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

更好的配置：

```
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：

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

推荐的配置：

```
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：

```
server {
    server_name _;
    root /var/www/site;
    location / {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_pass unix:/tmp/phpcgi.socket;
    }
}
```

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：

```
server {
    server_name _;
    root /var/www/site;
    location / {
        try_files $uri $uri/ @proxy;
    }
    location @proxy {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_pass unix:/tmp/phpcgi.socket;
    }
}
```

另外一个推荐的配置：

```
server {
    server_name _;
    root /var/www/site;
    location / {
        try_files $uri $uri/ /index.php;
    }
    location ~ \.php$ {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_pass unix:/tmp/phpcgi.socket;
    }
}
```

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擎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：

```
  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)

```
server {
    root /;

    location / {
        try_files /web/$uri $uri @php;
    }

    location @php {
        [...]
    }
}
```

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：

```
upstream {
    server http://someserver;
}

server {
    listen myhostname:80;
    # [...]
}
```

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：

```
upstream {
    server http://10.48.41.12;
}

server {
    listen 127.0.0.16:80;
    # [...]
}
```

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：

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

环境搭建

实践的前提是搭建环境，本节的几个小节将介绍在几种常见操作平台上 OpenResty 的安装。

为了降低用户安装门槛，对于不同系统安装，部分章节存在比较大的重复内容。读者只需要选择自己需要的平台并尝试安装即可。除了 windows 版本是以二进制发行，其他平台由于系统自身的兼容性，推荐的都是源码编译方式。

在 OpenResty 的规划路线中，准备发行独立的 `opm` 安装工具（截止到目前，目前还没完成，名称可能依然会变），帮会我们完成从环境到周边库的下载、更新。

Windows 平台安装

1、下载 Windows 版的 OpenResty 压缩包，这里我下载的是 openresty_for_windows_1.7.10.2001_64bit ，你也可以选择 32bit 的版本。如果你对源码感兴趣，下面是源码地址 https://github.com/LomoX-Offical/nginx-openresty-windows。

2、解压到要安装的目录，这里我选择D盘根目录，你可以根据自己的喜好选择位置。

3、进入到 openresty_for_windows_1.7.10.2001_64bit 目录，双击执行 nginx.exe 或者使用命令 start nginx 启动 nginx，如果没有错误现在 nginx 已经开始运行了。

4、验证 nginx 是否成功启动：

- 使用 `tasklist /fi "imagename eq nginx.exe"` 命令查看 nginx 进程，其中一个是 master 进程，另一个是 worker 进程，如下图：

- 在浏览器的地址栏输入 localhost，加载 nginx 的欢迎页面。成功加载说明 nginx 正在运行。如下图：

另外当 nginx 成功启动后，master 进程的 pid 存放在 `logs\nginx.pid` 文件中。

PS：OpenResty 当前也发布了 windows 版本，两个版本编译方式还是有区别的，这里更推荐这个版本。

CentOS 平台安装

从包管理安装

OpenResty 现在提供了 CentOS 上的 官方包。 你只需运行下面的命令：

```
sudo yum-config-manager --add-repo https://openresty.org/yum/cn/centos/OpenResty.repo
sudo yum install openresty
```

如果一切顺利，OpenResty 应该已经安装好了。 接下来，我们就可以进入到后面的章节 HelloWorld 学习。

如果你想了解更多 OpenResty 上的细节，且不介意弄脏双手；抑或有自定义 OpenResty 安装的需求，可以往下看从源码安装的方式。

源码包准备

我们首先要在官网下载 OpenResty 的源码包。官网上会提供很多的版本，各个版本有什么不同也会有说明，我们可以按需选择下载。 笔者选择下载的源码包为 ngx_openresty-1.9.7.1.tar.gz（请大家跟进使用最新版本，这里只是个例子）。

依赖库安装

将这些相关的库 `perl 5.6.1+,libreadline, libpcre, libssl` 安装在系统中。 按照以下步骤：

1. 输入以下命令 `yum install readline-devel pcre-devel openssl-devel perl` ，一次性安装需要的库。
2. 相关库安装成功。安装成功后会有 “Complete！” 字样。

OpenResty 安装

1. 在命令行中切换到源码包所在目录。
2. 解压源码包， `tar xzvf ngx_openresty-1.9.7.1.tar.gz` 。若你下载的源码包版本不一样，将相应的版本号改为你所下载的即可。
3. 切换工作目录到 `cd ngx_openresty-1.9.7.1` 。
4. 了解默认激活的组件。OpenResty 官网有组件列表,我们可以参考，列表中大部分组件默认激活，也有部分默认不激活。 默认不激活的组件，我们可以在编译的时激活，后面步骤详说明。
5. 配置安装目录及需要激活的组件。使用选项 --prefix=install_path，指定安装目录（默认为/usr/local/openresty）。

使用选项 --with-Components 激活组件，--without 则是禁止组件。 你可以根据自己实际需要选择 with 或 without。如下命令，OpenResty 将配置安装在 /opt/openresty 目录下（注意使用 root 用户）,并激活 `luajit` 、 `http_iconv_module` 并禁止 `http_redis2_module` 组件。

```
# ./configure --prefix=/opt/openresty\
            --with-luajit\
            --without-http_redis2_module \
            --with-http_iconv_module
```

6. 在上一步中，最后没有什么 error 的提示就是最好的。若有错误，最后会显示 具体原因 可以看源码包目录下的 `build/nginx-VERSION/objs/autoconf.err` 文件查看。若没有错误，则会出现如下信息：

```
Type the following commands to build and install:
    gmake
    gmake install
```

7. 编译：根据上一步命令提示，输入 `gmake` 。
8. 安装：输入 `gmake install` 。
9. 上面的步骤顺利完成之后，安装已经完成。可以在你指定的安装目录下看到一些相关目录及文件。

设置环境变量

为了后面启动 OpenResty 的命令简单一些，不用在 OpenResty 的安装目录下进行启动，我们设置环境变量来简化操作。 将 nginx 目录添加到 PATH 中。打开文件 /etc/profile， 在文件末尾加入 `export PATH=$PATH:/opt/openresty/nginx/sbin` ，若你的安装目录不一样，则做相应修改。 注意：这一步操作需要重新加载环境变量才会生效，可通过命令 `source /etc/profile` 或者重启服务器等方式实现。

接下来，我们就可以进入到后面的章节 HelloWorld 学习。

Ubuntu 平台安装

源码包准备

我们首先要在官网下载 OpenResty 的源码包。官网上会提供很多的版本，各个版本有什么不同也会有说明，我们可以按需选择下载。 笔者选择下载的源码包为 ngx_openresty-1.9.7.1.tar.gz。

相关依赖包的安装

首先你要安装 OpenResty 需要的多个库 请先配置好你的apt源，配置源的过程在这就不阐述了，然后执行以下命令安装OpenResty编译或运行时所需要的软件包。

```
# apt-get install libreadline-dev libncurses5-dev libpcre3-dev \
    libssl-dev perl make build-essential
```

如果你只是想测试一下OpenResty，并不想实际使用，那么你也可以不必去配置源和安装这些依赖库，请直接往下看。

OpenResty 安装

1. 在命令行中切换到源码包所在目录。
2. 解压源码包，`tar xzvf ngx_openresty-1.9.7.1.tar.gz`。若你下载的源码包版本不一样，将相应的版本号改为你所下载的即可。
3. 切换工作目录到 `cd ngx_openresty-1.9.7.1`。
4. 了解默认激活的组件。OpenResty 官网有组件列表,我们可以参考，列表中大部分组件默认激活，也有部分默认不激活。 默认不激活的组件，我们可以在编译的时激活，后面步骤详说明。
5. 配置安装目录及需要激活的组件。使用选项 --prefix=install_path，指定安装目录（默认为/usr/local/openresty）。

   使用选项 --with-Components 激活组件，--without 则是禁止组件。 你可以根据自己实际需要选择 with 或 without。如下命令，OpenResty 将配置安装在 /opt/openresty 目录下（注意使用 root 用户）,并激活 `luajit` 、 `http_iconv_module` 并禁止 `http_redis2_module` 组件。

```
# ./configure --prefix=/opt/openresty\
            --with-luajit\
            --without-http_redis2_module \
            --with-http_iconv_module
```

6. 在上一步中，最后没有什么 error 的提示就是最好的。若有错误，最后会显示 具体原因 可以看源码包目录下的 `build/nginx-VERSION/objs/autoconf.err` 文件查看。若没有错误，则会出现如下信息：

```
Type the following commands to build and install:
    gmake
    gmake install
```

7. 编译：根据上一步命令提示，输入 `gmake` 。
8. 安装：输入 `gmake install` 。
9. 上面的步骤顺利完成之后，安装已经完成。可以在你指定的安装目录下看到一些相关目录及文件。

设置环境变量

为了后面启动 OpenResty 的命令简单一些，不用在 OpenResty 的安装目录下进行启动，我们设置环境变量来简化操作。 将 nginx 目录添加到 PATH 中。打开文件 /etc/profile， 在文件末尾加入 `export PATH=$PATH:/opt/openresty/nginx/sbin` ，若你的安装目录不一样，则做相应修改。 注意：这一步操作需要重新加载环境变量才会生效，可通过命令 `source /etc/profile` 或者重启服务器等方式实现。

接下来，我们就可以进入到后面的章节 HelloWorld 学习。

Mac OS X 平台安装

从包管理安装

通过 Homebrew，OpenResty 提供了 OSX 上的 官方包。 你只需运行下面的命令：

```
brew tap homebrew/nginx
brew install homebrew/nginx/openresty
```

如果一切顺利，OpenResty 应该已经安装好了。 接下来，我们就可以进入到后面的章节 HelloWorld 学习。

如果你想了解更多 OpenResty 上的细节，且不介意弄脏双手；抑或有自定义 OpenResty 安装的需求，可以往下看从源码安装的方式。

源码包准备

我们首先要在官网下载 `OpenResty` 的源码包。官网上会提供很多的版本，各个版本有什么不同也会有说明，我们可以按需选择下载。笔者选择下载的源码包 ngx_openresty-1.9.7.1.tar.gz。

相关库的安装

将这些相关库安装到系统中，推荐如 Homebrew 这类包管理方式完成包管理：

```
$ brew update
$ brew install pcre openssl
```

OpenResty 安装

1. 在命令行中切换到源码包所在目录。
2. 输入命令 `tar xzvf ngx_openresty-1.9.7.1.tar.gz` ，按回车键解压源码包。若你下载的源码包版本不一样， 将相应的版本号改为你所下载的即可，或者直接拷贝源码包的名字到命令中。 此时当前目录下会出现一个 `ngx_openresty-1.9.7.1` 文件夹。
3. 在命令行中切换工作目录到 `ngx_openresty-1.9.7.1` 。输入命令 `cd ngx_openresty-1.9.7.1` 。
4. 配置安装目录及需要激活的组件。使用选项 --prefix=install_path ，指定其安装目录（默认为/usr/local/openresty）。 使用选项 --with-Components 激活组件， --without 则是禁止组件，你可以根据自己实际需要选择 with 及 without 。 输入如下命令，OpenResty 将

配置安装在 /opt/openresty 目录下（注意使用root用户），激活 LuaJIT、HTTP_iconv_module 并禁止 http_redis2_module 组件。

```
./configure --prefix=/opt/openresty\
            --with-cc-opt="-I/usr/local/include"\
            --with-luajit\
            --without-http_redis2_module \
            --with-ld-opt="-L/usr/local/lib"
```

5. 在上一步中，最后没有什么error的提示就是最好的。若有错误，最后会显示error字样，具体原因可以看源码包目录下的build/nginx-VERSION/objs/autoconf.err文件查看。 若没有错误，则会出现如下信息，提示下一步操作：

```
Type the following commands to build and install:
make
sudo make install
```

6. 编译。根据上一步命令提示，输入 `make` 。
7. 安装。输入 `sudo make install` ，这里可能需要输入你的管理员密码。
8. 上面的步骤顺利完成之后，安装已经完成。可以在你指定的安装目录下看到一些相关目录及文件。

设置环境变量

为了后面启动 `OpenResty` 的命令简单一些，不用在 `OpenResty` 的安装目录下进行启动，我们通过设置环境变量来简化操作。 将 `OpenResty` 目录下的 nginx/sbin 目录添加到 PATH 中。

接下来，我们就可以进入到后面的章节 Hello World 学习。

HelloWorld

`HelloWorld` 是我们亘古不变的第一个入门程序。但是 `OpenResty` 不是一门编程语言，跟其他编程语言的 `HelloWorld` 不一样，让我们看看都有哪些不一样吧。

创建工作目录

OpenResty 安装之后就有配置文件及相关的目录的，为了工作目录与安装目录互不干扰，并顺便学下简单的配置文件编写，我们另外创建一个 OpenResty 的工作目录来练习，并且另写一个配置文件。我选择在当前用户目录下创建 openresty-test 目录，并在该目录下创建 logs 和 conf 子目录分别用于存放日志和配置文件。

```
$ mkdir ~/openresty-test ~/openresty-test/logs/ ~/openresty-test/conf/
$
$ tree ~/openresty-test
/Users/yuansheng/openresty-test
├── conf
└── logs

2 directories, 0 files
```

创建配置文件

在 conf 目录下创建一个文本文件作为配置文件，命名为 nginx.conf，文件内容如下:

```
worker_processes  1;        #nginx worker 数量
error_log logs/error.log;   #指定错误日志文件路径
events {
    worker_connections 1024;
}

http {
    server {
        #监听端口，若你的6699端口已经被占用，则需要修改
        listen 6699;
        location / {
            default_type text/html;

            content_by_lua_block {
                ngx.say("HelloWorld")
            }
        }
    }
}
```

提示：如果你安装的是 openresty 1.9.3.1 及以下版本，请使用 `content_by_lua` 命令代替示例中的 `content_by_lua_block`。可使用 `nginx -V` 命令查看版本号。

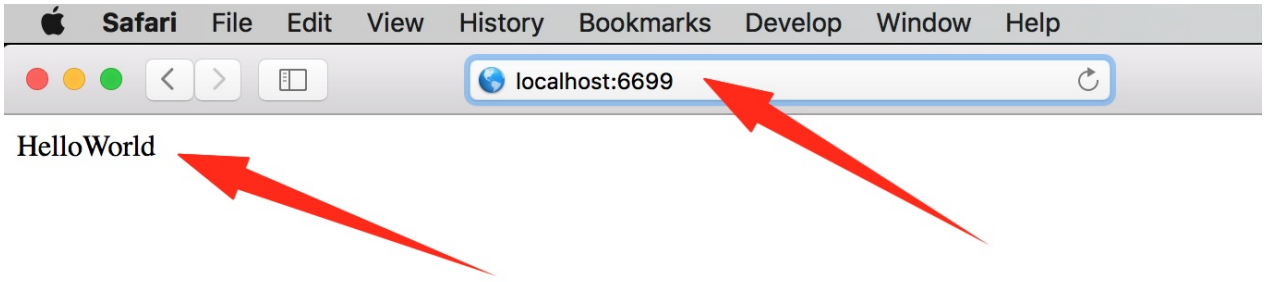
万事俱备只欠东风

我们启动 Nginx 即可，输入命令形式为：`nginx -p ~/openresty-test`，如果没有提示错误。如果提示 nginx 不存在，则需要在环境变量中加入安装路径，可以根据你的操作平台，参考前面的安装章节（一般需要重启生效）。

启动成功后，我们可以查看 nginx 进程是否存在，并通过访问 HTTP 页面查看应答内容。操作提示如下：

```
➜  ~ nginx -p ~/openresty-test
➜  ~ ps -ef | grep nginx
  501 88620     1   0 10:58AM ?? 0:00.00 nginx: master process nginx -p
                                    /Users/yuansheng/openresty-test
  501 88622 88620   0 10:58AM ?? 0:00.00 nginx: worker process
➜  ~ curl http://localhost:6699 -i
HTTP/1.1 200 OK
Server: openresty/1.9.7.3
Date: Sun, 20 Mar 2016 03:01:35 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive

HelloWorld
```

在浏览器中完成同样的访问：



与其他 location 配合

nginx 世界的 location 是异常强大的，毕竟 nginx 的主要应用场景是在负载均衡、API server，在不同 server、location 之间跳转更是家常便饭。利用不同 location 的功能组合，我们可以完成内部调用、流水线方式跳转、外部重定向等几大不同方式，下面将给大家介绍几个主要应用，就当抛砖引玉。

内部调用

例如对数据库、内部公共函数的统一接口，可以把它们放到统一的 location 中。通常情况下，为了保护这些内部接口，都会把这些接口设置为 internal 。这么做的最主要好处就是可以让这个内部接口相对独立，不受外界干扰。

示例代码：

```
location = /sum {
    # 只允许内部调用
    internal;

    # 这里做了一个求和运算只是一个例子，可以在这里完成一些数据库、
    # 缓存服务器的操作，达到基础模块和业务逻辑分离目的
    content_by_lua_block {
        local args = ngx.req.get_uri_args()
        ngx.say(tonumber(args.a) + tonumber(args.b))
    }
}

location = /app/test {
    content_by_lua_block {
        local res = ngx.location.capture(
                        "/sum", {args={a=3, b=8}}
                        )
        ngx.say("status:", res.status, " response:", res.body)
    }
}
```

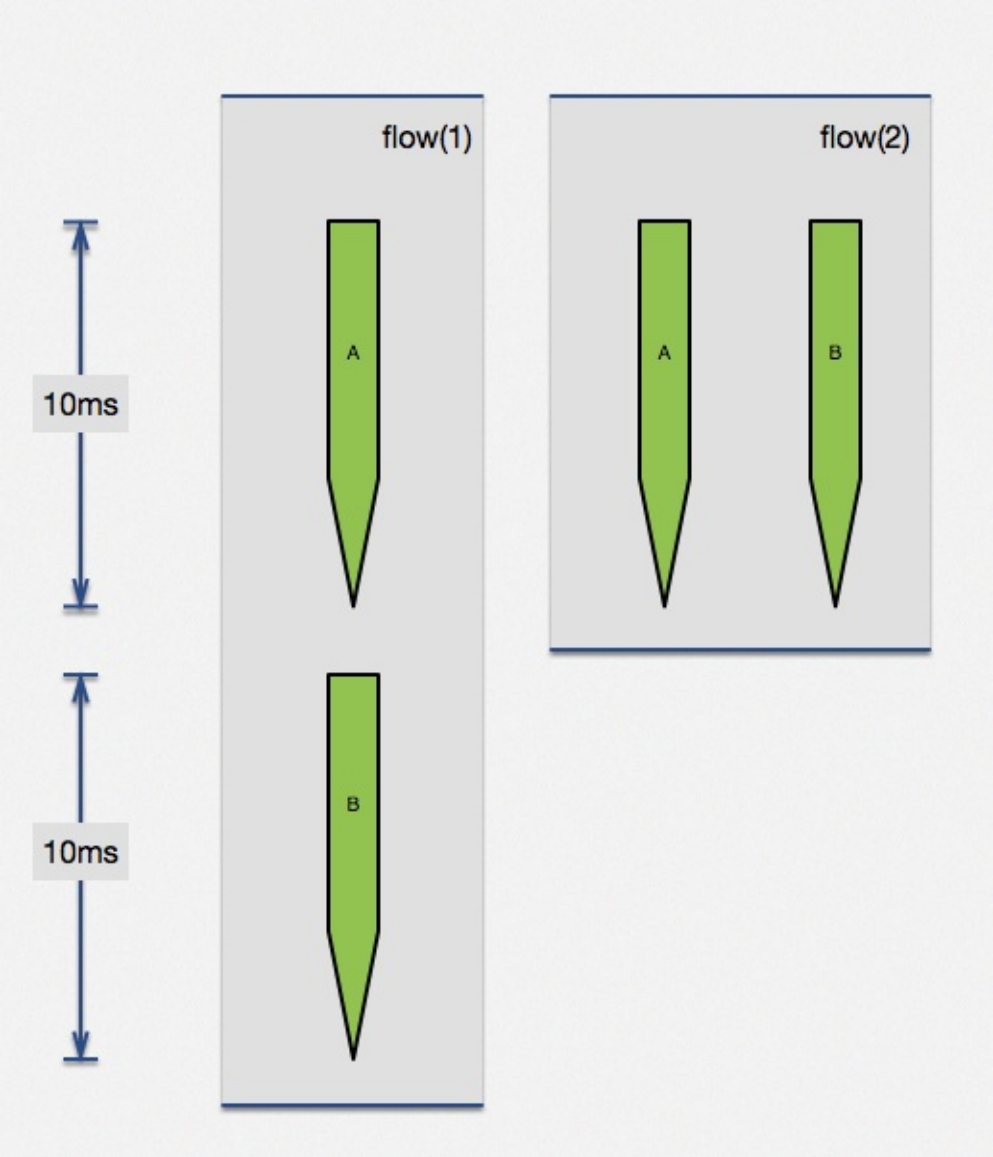
紧接着，稍微扩充一下，并行请求的效果，示例如下：

```
location = /sum {
    internal;
    content_by_lua_block {
        ngx.sleep(0.1)
        local args = ngx.req.get_uri_args()
        ngx.print(tonumber(args.a) + tonumber(args.b))
    }
}

location = /subduction {
    internal;
    content_by_lua_block {
        ngx.sleep(0.1)
        local args = ngx.req.get_uri_args()
        ngx.print(tonumber(args.a) - tonumber(args.b))
    }
}

location = /app/test_parallels {
    content_by_lua_block {
        local start_time = ngx.now()
        local res1, res2 = ngx.location.capture_multi( {
                        {"/sum", {args={a=3, b=8}}},
                        {"/subduction", {args={a=3, b=8}}}
                    })
        ngx.say("status:", res1.status, " response:", res1.body)
        ngx.say("status:", res2.status, " response:", res2.body)
        ngx.say("time used:", ngx.now() - start_time)
    }
}

location = /app/test_queue {
    content_by_lua_block {
        local start_time = ngx.now()
        local res1 = ngx.location.capture_multi( {
                        {"/sum", {args={a=3, b=8}}}
                    })
        local res2 = ngx.location.capture_multi( {
                        {"/subduction", {args={a=3, b=8}}}
                    })
        ngx.say("status:", res1.status, " response:", res1.body)
        ngx.say("status:", res2.status, " response:", res2.body)
        ngx.say("time used:", ngx.now() - start_time)
    }
}
```

测试结果：

```
➜  ~ curl 127.0.0.1/app/test_parallels
status:200 response:11
status:200 response:-5
time used:0.10099983215332
➜  ~ curl 127.0.0.1/app/test_queue
status:200 response:11
status:200 response:-5
time used:0.20199990272522
```

利用 `ngx.location.capture_multi` 函数，直接完成了两个子请求并行执行。当两个请求没有相互依赖，这种方法可以极大提高查询效率。两个无依赖请求，各自是 100ms，顺序执行需要 200ms，但通过并行执行可以在 100ms 完成两个请求。实际生产中查询时间可能没这么规整，但思想大同小异，这个特性是很有用的。



该方法，可以被广泛应用于广告系统（1：N模型，一个请求，后端从N家供应商中获取条件最优广告）、高并发前端页面展示（并行无依赖界面、降级开关等）。

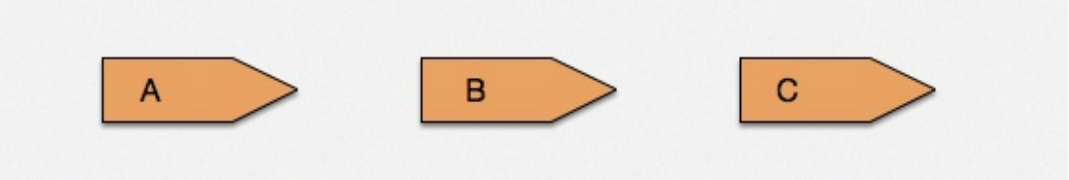
流水线方式跳转

现在的网络请求，已经变得越来越拥挤。各种不同 API 、下载请求混杂在一起，就要求不同厂商对下载的动态调整有各种不同的定制策略，而这些策略在一天的不同时间段，规则可能还不一样。这时候我们还可以效仿工厂的流水线模式，逐层过滤、处理。

示例代码：

```
location ~ ^/static/([-_a-zA-Z0-9/]+).jpg {
    set $image_name $1;
    content_by_lua_block {
        ngx.exec("/download_internal/images/"
                .. ngx.var.image_name .. ".jpg");
    };
}

location /download_internal {
    internal;
    # 这里还可以有其他统一的 download 下载设置，例如限速等
    alias ../download;
}
```

注意，ngx.exec 方法与 ngx.redirect 是完全不同的，前者是个纯粹的内部跳转并且没有引入任何额外 HTTP 信号。 这里的两个 location 更像是流水线上工人之间的协作关系。第一环节的工人对完成自己处理部分后，直接交给第二环节处理人（实际上可以有更多环节），它们之间的数据流是定向的。



外部重定向

不知道大家什么时候开始注意的，百度的首页已经不再是 HTTP 协议，它已经全面修改到了 HTTPS 协议上。但是对于大家的输入习惯，估计还是在地址栏里面输入 `baidu.com` ，回车后发现它会自动跳转到 `https://www.baidu.com` ，这时候就需要的外部重定向了。

```
location = /foo {
    content_by_lua_block {
        ngx.say([[I am foo]])
    }
}

location = / {
    rewrite_by_lua_block {
        return ngx.redirect('/foo');
    }
}
```

执行测试，结果如下：

```
➜  ~  curl 127.0.0.1 -i
HTTP/1.1 302 Moved Temporarily
Server: openresty/1.9.3.2rc3
Date: Sun, 22 Nov 2015 11:04:03 GMT
Content-Type: text/html
Content-Length: 169
Connection: keep-alive
Location: /foo

<html>
<head><title>302 Found</title></head>
<body bgcolor="white">
<center><h1>302 Found</h1></center>
<hr><center>openresty/1.9.3.2rc3</center>
</body>
</html>

➜  ~  curl 127.0.0.1/foo -i
HTTP/1.1 200 OK
Server: openresty/1.9.3.2rc3
Date: Sun, 22 Nov 2015 10:43:51 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive

I am foo
```

当我们使用浏览器访问页面 `http://127.0.0.1` 就可以发现浏览器会自动跳转到 `http://127.0.0.1/foo` 。

与之前两个应用实例不同的，外部重定向是可以跨域名的。例如从 A 网站跳转到 B 网站是绝对允许的。在 CDN 场景的大量下载应用中，一般分为调度、存储两个重要环节。调度就是通过根据请求方 IP 、下载文件等信息寻找最近、最快节点，应答跳转给请求方完成下载。

获取 uri 参数

上一章节，主要介绍了一下如何使用不同 location 进行协作，对 location 进行糅合，往往都是要需要参数的二次调整。如何正确获取传递参数、设置参数，就是你的必修课了。本章目的是给出在 OpenResty 的世界中，我们如何正确获取、设置 uri 参数。

获取请求 uri 参数

首先看一下官方 API 文档，获取一个 uri 有两个方法：`ngx.req.get_uri_args` 、`ngx.req.get_post_args`，二者主要的区别是参数来源有区别。

参考下面例子：

```
server {
   listen    80;
   server_name  localhost;

   location /print_param {
       content_by_lua_block {
           local arg = ngx.req.get_uri_args()
           for k,v in pairs(arg) do
               ngx.say("[GET ] key:", k, " v:", v)
           end

           ngx.req.read_body() -- 解析 body 参数之前一定要先读取 body
           local arg = ngx.req.get_post_args()
           for k,v in pairs(arg) do
               ngx.say("[POST] key:", k, " v:", v)
           end
       }
   }
}
```

输出结果：

```
➜  ~  curl '127.0.0.1/print_param?a=1&b=2%26' -d 'c=3&d=4%26'
[GET ] key:b v:2&
[GET ] key:a v:1
[POST] key:d v:4&
[POST] key:c v:3
```

从这个例子中，我们可以很明显看到两个函数 `ngx.req.get_uri_args` 、`ngx.req.get_post_args` 获取数据来源是有明显区别的，前者来自 uri 请求参数，而后者来自 post 请求内容。

传递请求 uri 参数

当我们可以获取到请求参数，自然是需要这些参数来完成业务控制目的。大家都知道，URI 内容传递过程中是需要调用 ngx.encode_args 进行规则转义。

参看下面例子：

```
    location /test {
        content_by_lua_block {
            local res = ngx.location.capture(
                    '/print_param',
                    {
                       method = ngx.HTTP_POST,
                       args = ngx.encode_args({a = 1, b = '2&'}),
                       body = ngx.encode_args({c = 3, d = '4&'})
                    }
                )
            ngx.say(res.body)
        }
    }
```

输出结果：

```
➜  ~  curl '127.0.0.1/test'
[GET]  key:b v:2&
[GET]  key:a v:1
[POST] key:d v:4&
[POST] key:c v:3
```

与我们预期是一样的。

如果这里不调用 `ngx.encode_args` ，可能就会比较丑了，看下面例子：

```
local res = ngx.location.capture('/print_param',
         {
            method = ngx.HTTP_POST,
            args = 'a=1&b=2%26',  -- 注意这里的 %26 ,代表的是 & 字符
            body = 'c=3&d=4%26'
        }
     )
ngx.say(res.body)
```

PS：对于 ngx.location.capture 这里有个小技巧，args 参数可以接受字符串或Lua 表的，这样我们的代码就更加简洁直观。

```
local res = ngx.location.capture('/print_param',
         {
            method = ngx.HTTP_POST,
            args = {a = 1, b = '2&'},
            body = 'c=3&d=4%26'
        }
     )
ngx.say(res.body)
```

获取请求 body

在 Nginx 的典型应用场景中，几乎都是只读取 HTTP 头即可，例如负载均衡、正反向代理等场景。但是对于 API Server 或者 Web Application ，对 body 可以说就比较敏感了。由于 OpenResty 基于 Nginx ，所以天然的对请求 body 的读取细节与其他成熟 Web 框架有些不同。

最简单的 “Hello ****”

我们先来构造最简单的一个请求，POST 一个名字给服务端，服务端应答一个 “Hello ****”。

```
http {
    server {
        listen    80;

        location /test {
            content_by_lua_block {
                local data = ngx.req.get_body_data()
                ngx.say("hello ", data)
            }
        }
    }
}
```

测试结果：

```
➜  ~  curl 127.0.0.1/test -d jack
hello nil
```

大家可以看到 data 部分获取为空，如果你熟悉其他 web 开发框架，估计立刻就觉得 OpenResty 弱爆了。查阅一下官方 wiki 我们很快知道，原来我们还需要添加指令 lua_need_request_body 。究其原因，主要是 Nginx 诞生之初主要是为了解决负载均衡情况，而这种情况，是不需要读取 body 就可以决定负载策略的，所以这个点对于 API Server 和 Web Application 开发的同学有点怪。

参看下面例子：

```
http {
    server {
        listen    80;

        # 默认读取 body
        lua_need_request_body on;

        location /test {
            content_by_lua_block {
                local data = ngx.req.get_body_data()
                ngx.say("hello ", data)
            }
        }
    }
}
```

再次测试，符合我们预期：

```
➜  ~  curl 127.0.0.1/test -d jack
hello jack
```

如果你只是某个接口需要读取 body（并非全局行为），那么这时候也可以显示调用 ngx.req.read_body() 接口，参看下面示例：

```
http {
    server {
        listen    80;

        location /test {
            content_by_lua_block {
                ngx.req.read_body()
                local data = ngx.req.get_body_data()
                ngx.say("hello ", data)
            }
        }
    }
}
```

body 偶尔读取不到？

ngx.req.get_body_data() 读请求体，会偶尔出现读取不到直接返回 nil 的情况。

如果请求体尚未被读取，请先调用 ngx.req.read_body (或打开 lua_need_request_body 选项强制本模块读取请求体，此方法不推荐）。

如果请求体已经被存入临时文件，请使用 ngx.req.get_body_file 函数代替。

如需要强制在内存中保存请求体，请设置 client_body_buffer_size 和 client_max_body_size 为同样大小。

参考下面代码：

```
http {
    server {
        listen    80;

        # 强制请求 body 到临时文件中（仅仅为了演示）
        client_body_in_file_only on;

        location /test {
            content_by_lua_block {
                function getFile(file_name)
                    local f = assert(io.open(file_name, 'r'))
                    local string = f:read("*all")
                    f:close()
                    return string
                end

                ngx.req.read_body()
                local data = ngx.req.get_body_data()
                if nil == data then
                    local file_name = ngx.req.get_body_file()
                    ngx.say(">> temp file: ", file_name)
                    if file_name then
                        data = getFile(file_name)
                    end
                end

                ngx.say("hello ", data)
            }
        }
    }
}
```

测试结果：

```
➜  ~  curl 127.0.0.1/test -d jack
>> temp file: /Users/rain/Downloads/nginx/client_body_temp/0000000018
hello jack
```

由于 Nginx 是为了解决负载均衡场景诞生的，所以它默认是不读取 body 的行为，会对 API Server 和 Web Application 场景造成一些影响。根据需要正确读取、丢弃 body 对 OpenResty 开发是至关重要的。

输出响应体

HTTP响应报文分为三个部分：

1. 响应行
2. 响应头
3. 响应体

对于 HTTP 响应体的输出，在 OpenResty 中调用 `ngx.say` 或 `ngx.print` 即可。经过查看官方 wiki ，这两者都是输出响应体，区别是 `ngx.say` 会对输出响应体多输出一个 `\n` 。如果你用的是浏览器完成的功能调试，使用这两着是没有区别的。但是如果使用各种终端工具，这时候使用 `ngx.say` 明显就更方便了。

`ngx.say` 与 `ngx.print` 均为异步输出

首先需要明确一下的，是这两个函数都是异步输出的，也就是说当调用 `ngx.say` 后并不会立刻输出响应体。参考下面的例子：

```
    server {
        listen    80;

        location /test {
            content_by_lua_block {
                ngx.say("hello")
                ngx.sleep(3)
                ngx.say("the world")
            }
        }

        location /test2 {
            content_by_lua_block {
                ngx.say("hello")
                ngx.flush() -- 显式的向客户端刷新响应输出
                ngx.sleep(3)
                ngx.say("the world")
            }
        }
    }
```

测试接口可以观察到，`/test` 响应内容实在触发请求 3s 后一起接收到响应体，而 `/test2` 则是先收到一个 `hello` 停顿 3s 后又接收到后面的 `the world` 。

再看下面的例子：

```
    server {
        listen    80;
        lua_code_cache off;

        location /test {
            content_by_lua_block {
                ngx.say(string.rep("hello", 1000))
                ngx.sleep(3)
                ngx.say("the world")
            }
        }
    }
```

执行测试，可以发现首先收到了所有的 "hello" ，停顿大约 3 秒后，接着又收到了 "the world" 。

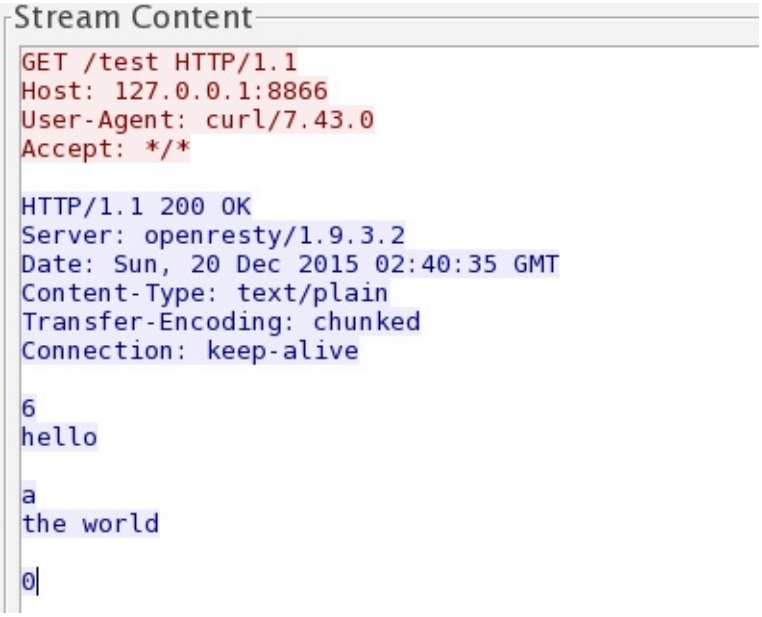
通过两个例子对比，可以知道，因为是异步输出，两个响应体的输出时机是 不一样 的。

如何优雅处理响应体过大的输出

如果响应体比较小，这时候相对就比较随意。但是如果响应体过大（例如超过 2G），是不能直接调用 API 完成响应体输出的。响应体过大，分两种情况：

1. 输出内容本身体积很大，例如超过 2G 的文件下载
2. 输出内容本身是由各种碎片拼凑的，碎片数量庞大，例如应答数据是某地区所有人的姓名

第①个情况，要利用 HTTP 1.1 特性 CHUNKED 编码来完成，一起来看看 CHUNKED 编码格式样例：



可以利用 CHUNKED 格式，把一个大的响应体拆分成多个小的应答体，分批、有节制的响应给请求方。

参考下面的例子：

```
location /test {
    content_by_lua_block {
        -- ngx.var.limit_rate = 1024*1024
        local file, err = io.open(ngx.config.prefix() .. "data.db","r")
        if not file then
            ngx.log(ngx.ERR, "open file error:", err)
            ngx.exit(ngx.HTTP_SERVICE_UNAVAILABLE)
        end

        local data
        while true do
            data = file:read(1024)
            if nil == data then
                break
            end
            ngx.print(data)
            ngx.flush(true)
        end
        file:close()
    }
}
```

按块读取本地文件内容（每次 1KB），并以流式方式进行响应。笔者本地文件 `data.db` 大小是 4G ， Nginx 服务可以稳定运行，并维持内存占用在 几MB 范畴。

注：其实 nginx 自带的静态文件解析能力已经非常好了。这里只是一个例子，实际中过大响应体都是后端服务生成的，为了演示环境相对封闭，所以这里选择本地文件。

第②个情况，其实就是要利用 `ngx.print` 的特性了，它的输入参数可以是单个或多个字符串参数，也可以是 table 对象。

参考官方示例代码：

```
local table = {
     "hello, ",
     {"world: ", true, " or ", false,
         {": ", nil}}
 }
 ngx.print(table)
```

将输出：

```
 hello, world: true or false: nil
```

也就是说当有非常多碎片数据时，没有必要一定连接成字符串后再进行输出。完全可以直接存放在 table 中，用数组的方式把这些碎片数据统一起来，直接调用 `ngx.print(table)` 即可。这种方式效率更高，并且更容易被优化。

日志输出

你如何测试和调试你的代码呢？Lua 的两个主力作者是这样回复的：

Luiz Henrique de Figueiredo：我主要是一块一块的构建，分块测试。我很少使用调试器。即使用调试器，也只是调试 C 代码。我从不用调试器调试 Lua 代码。对于 Lua 来说，在适当的位置放几条打印语句通常就可以胜任了。

Roberto Ierusalimschy：我差不多也是这样。当我使用调试器时，通常只是用来查找代码在哪里崩溃了。对于 C 代码，有个像 Valgrind 或者 Purify 这样的工具是必要的。

摘自《编程之魂 -- 采访 Lua 发明人的一篇文章》。

由此可见掌握日志输出是多么重要，下至入门同学，上至 Lua 作者，使用日志输出来确定问题，是很必要的基本手段。

标准日志输出

OpenResty 的标准日志输出原句为 `ngx.log(log_level, ...)` ，几乎可以在任何 ngx_lua 阶段进行日志的输出。

请看下面的示例：

```
#user  nobody;
worker_processes  1;

error_log  logs/error.log error;    # 日志级别
#pid        logs/nginx.pid;

events {
    worker_connections  1024;
}

http {
    server {
        listen    80;
        location / {
            content_by_lua_block {
                local num = 55
                local str = "string"
                local obj
                ngx.log(ngx.ERR, "num:", num)
                ngx.log(ngx.INFO, " string:", str)
                print([[i am print]])
                ngx.log(ngx.ERR, " object:", obj)
            }
        }
    }
}
```

访问网页，生成日志（logs/error.log 文件）结果如下：

```
2016/01/22 16:43:34 [error] 61610#0: *10 [lua] content_by_lua(nginx.conf:26):5:
 num:55, client: 127.0.0.1, server: , request: "GET /hello HTTP/1.1",
 host: "127.0.0.1"
2016/01/22 16:43:34 [error] 61610#0: *10 [lua] content_by_lua(nginx.conf:26):7:
 object:nil, client: 127.0.0.1, server: , request: "GET /hello HTTP/1.1",
 host: "127.0.0.1"
```

大家可以在单行日志中获取很多有用的信息，例如：时间、日志级别、请求ID、错误代码位置、内容、客户端 IP 、请求参数等等，这些信息都是环境信息，可以用来辅助完成更多其他操作。当然我们也可以根据自己需要定义日志格式，具体可以参考 nginx 的 log_format 章节。

细心的读者发现了，中间的两行日志哪里去了？这里不卖关子，其实是日志输出级别的原因。上面的例子，日志输出级别使用的 error，只有等于或大于这个级别的日志才会输出。这里还有一个知识点就是 OpenResty 里面的 `print` 语句是 INFO 级别。

有关 Nginx 的日志级别，请看下表：

```
ngx.STDERR      -- 标准输出
ngx.EMERG       -- 紧急报错
ngx.ALERT       -- 报警
ngx.CRIT        -- 严重，系统故障，触发运维告警系统
ngx.ERR         -- 错误，业务不可恢复性错误
ngx.WARN        -- 告警，业务中可忽略错误
ngx.NOTICE      -- 提醒，业务比较重要信息
ngx.INFO        -- 信息，业务琐碎日志信息，包含不同情况判断等
ngx.DEBUG       -- 调试
```

他们是一些常量，越往上等级越高。读者朋友可以尝试把 error log 日志级别修改为 info，然后重新执行一下测试用例，就可以看到全部日志输出结果了。

对于应用开发，一般使用 ngx.INFO 到 ngx.CRIT 就够了。生产中错误日志开启到 error 级别就够了。如何正确使用这些级别呢？可能不同的人、不同的公司可能有不同见解。

网络日志输出

如果你的日志需要归集，并且对时效性要求比较高那么这里要推荐的库可能就让你很喜欢了。 lua-resty-logger-socket ，可以说很好的解决了上面提及的几个特性。

lua-resty-logger-socket 的目标是替代 Nginx 标准的 ngx_http_log_module 以非阻塞 IO 方式推送 access log 到远程服务器上。对远程服务器的要求是支持 syslog-ng 的日志服务。

引用官方示例：

```
lua_package_path "/path/to/lua-resty-logger-socket/lib/?.lua;;";

    server {
        location / {
            log_by_lua_block {
                local logger = require "resty.logger.socket"
                if not logger.initted() then
                    local ok, err = logger.init{
                        host = 'xxx',
                        port = 1234,
                        flush_limit = 1234,
                        drop_limit = 5678,
                    }
                    if not ok then
                        ngx.log(ngx.ERR, "failed to initialize the logger: ",
                                err)
                        return
                    end
                end

                -- construct the custom access log message in
                -- the Lua variable "msg"

                local bytes, err = logger.log(msg)
                if err then
                    ngx.log(ngx.ERR, "failed to log message: ", err)
                    return
                end
            }
        }
    }
```

例举几个好处：

- 基于 cosocket 非阻塞 IO 实现
- 日志累计到一定量，集体提交，增加网络传输利用率
- 短时间的网络抖动，自动容错
- 日志累计到一定量，如果没有传输完毕，直接丢弃
- 日志传输过程完全不落地，没有任何磁盘 IO 消耗

简单API Server框架

实现一个最最简单的数学计算：加、减、乘、除，给大家演示如何搭建简单的 API Server。

按照前面几章的写法，先来看看加法、减法示例代码：

```
worker_processes  1;        #nginx worker 数量
error_log logs/error.log;   #指定错误日志文件路径
events {
    worker_connections 1024;
}
http {
    server {
        listen 80;

        # 加法
        location /addition {
           content_by_lua_block {
                local args = ngx.req.get_uri_args()
                ngx.say(args.a + args.b)
            }
        }

        # 减法
        location /subtraction {
            content_by_lua_block {
                local args = ngx.req.get_uri_args()
                ngx.say(args.a - args.b)
            }
        }

        # 乘法
        location /multiplication {
            content_by_lua_block {
                local args = ngx.req.get_uri_args()
                ngx.say(args.a * args.b)
            }
        }

        # 除法
        location /division {
            content_by_lua_block {
                local args = ngx.req.get_uri_args()
                ngx.say(args.a / args.b)
            }
        }
    }
}
```

代码写多了一眼就可以看出来，这么简单的加减乘除，居然写了这么长，而且还要对每个 API 都写一个 location ，作为有追求的人士，怎能容忍这种代码风格？

- 首先是需要把这些 location 合并；
- 其次是这些接口的实现放到独立文件中，保持 nginx 配置文件的简洁；

基于这两点要求，可以改成下面的版本，看上去有那么几分模样的样子：

> nginx.conf 内容：

```
worker_processes  1;        #nginx worker 数量
error_log logs/error.log;   #指定错误日志文件路径
events {
    worker_connections 1024;
}

http {
    # 设置默认 lua 搜索路径，添加 lua 路径
    # 此处写相对路径时，对启动 nginx 的路径有要求，必须在 nginx 目录下启动，require 找不到
    # comm.param 绝对路径当然也没问题，但是不可移植，因此应使用变量 $prefix 或
    # ${prefix}，OR 会替换为 nginx 的 prefix path。

    # lua_package_path 'lua/?.lua;/blah/?.lua;;';
    lua_package_path '$prefix/lua/?.lua;/blah/?.lua;;';

    # 这里设置为 off，是为了避免每次修改之后都要重新 reload 的麻烦。
    # 在生产环境上务必确保 lua_code_cache 设置成 on。
    lua_code_cache off;

    server {
        listen 80;

        # 在代码路径中使用nginx变量
        # 注意： nginx var 的变量一定要谨慎，否则将会带来非常大的风险
        location ~ ^/api/([-_a-zA-Z0-9/]+) {
            # 准入阶段完成参数验证
            access_by_lua_file  lua/access_check.lua;

            #内容生成阶段
            content_by_lua_file lua/$1.lua;
        }
    }
}
```

> 其他文件内容：

```
---------- {$prefix}/lua/addition.lua
local args = ngx.req.get_uri_args()
ngx.say(args.a + args.b)

---------- {$prefix}/lua/subtraction.lua
local args = ngx.req.get_uri_args()
ngx.say(args.a - args.b)

---------- {$prefix}/lua/multiplication.lua
local args = ngx.req.get_uri_args()
ngx.say(args.a * args.b)

---------- {$prefix}/lua/division.lua
local args = ngx.req.get_uri_args()
ngx.say(args.a / args.b)
```

既然对外提供的是 API Server，作为一个服务端程序员，怎么可以容忍输入参数不检查呢？万一对方送过来的不是数字或者为空，这些都要过滤掉嘛。参数检查过滤的方法是统一，在这几个 API 中如何共享这个方法呢？这时候就需要 Lua 模块来完成了。

- 使用统一的公共模块，完成参数验证；
- 验证入口最好也统一，不要分散在不同地方；

> nginx.conf 内容：

```
worker_processes  1;        #nginx worker 数量
error_log logs/error.log;   #指定错误日志文件路径
events {
    worker_connections 1024;
}
http {
    server {
        listen 80;

        # 在代码路径中使用nginx变量
        # 注意： nginx var 的变量一定要谨慎，否则将会带来非常大的风险
        location ~ ^/api/([-_a-zA-Z0-9/]+) {
            access_by_lua_file  lua/access_check.lua;
            content_by_lua_file lua/$1.lua;
        }
    }
}
```

> 新增文件内容：

```
--========== {$prefix}/lua/comm/param.lua
local _M = {}

-- 对输入参数逐个进行校验，只要有一个不是数字类型，则返回 false
function _M.is_number(...)
    local arg = {...}

    local num
    for _,v in ipairs(arg) do
        num = tonumber(v)
        if nil == num then
            return false
        end
    end

    return true
end

return _M

--========== {$prefix}/lua/access_check.lua
local param= require("comm.param")
local args = ngx.req.get_uri_args()

if not args.a or not args.b or not param.is_number(args.a, args.b) then
    ngx.exit(ngx.HTTP_BAD_REQUEST)
    return
end
```

看看curl测试结果吧：

```
$  nginx  curl '127.0.0.1:80/api/addition?a=1'
<html>
<head><title>400 Bad Request</title></head>
<body bgcolor="white">
<center><h1>400 Bad Request</h1></center>
<hr><center>openresty/1.9.3.1</center>
</body>
</html>
$  nginx  curl '127.0.0.1:80/api/addition?a=1&b=3'
4
```

基本是按照预期执行的。参数不全、错误时，会提示400错误。正常处理，可以返回预期结果。

来整体看一下目前的目录关系：

```
.
├── conf
│   ├── nginx.conf
├── logs
│   ├── error.log
│   └── nginx.pid
├── lua
│   ├── access_check.lua
│   ├── addition.lua
│   ├── subtraction.lua
│   ├── multiplication.lua
│   ├── division.lua
│   └── comm
│       └── param.lua
└── sbin
    └── nginx
```

怎么样，有点 magic 的味道不？其实你的接口越是规范，有固定规律可寻，那么 OpenResty 就总是很容易能找到适合你的位置。当然这里你也可以把 `access_check.lua` 内容分别复制到加、减、乘、除实现的四个 Lua 文件中，肯定也是能用的。这里只是为了给大家提供更多的玩法，需要的时候可以有更多的选择。

本章目的是搭建一个简单API Server，记住这绝对不是终极版本。这里面还有很多需要进一步去考虑的地方，但是作为最基本的框架已经有了。

使用 **Nginx** 内置绑定变量

`Nginx` 作为一个成熟、久经考验的负载均衡软件，与其提供丰富、完整的内置变量是分不开的，它极大增加了对 `Nginx` 网络行为的控制细度。这些变量大部分都是在请求进入时解析的，并把他们缓存到请求 `cycle` 中，方便下一次获取使用。首先来看看 `Nginx` 对都开放了那些 `API` 。

参看下表：

名称	说明
$arg_name	请求中的name参数
$args	请求中的参数
$binary_remote_addr	远程地址的二进制表示
$body_bytes_sent	已发送的消息体字节数
$content_length	HTTP请求信息里的"Content-Length"
$content_type	请求信息里的"Content-Type"
$document_root	针对当前请求的根路径设置值
$document_uri	与$uri相同; 比如 /test2/test.php
$host	请求信息中的"Host"，如果请求中没有Host行，则等于设置的服务器名
$hostname	机器名使用 gethostname系统调用的值
$http_cookie	cookie 信息
$http_referer	引用地址
$http_user_agent	客户端代理信息
$http_via	最后一个访问服务器的Ip地址。
$http_x_forwarded_for	相当于网络访问路径
$is_args	如果请求行带有参数，返回“?”，否则返回空字符串
$limit_rate	对连接速率的限制
$nginx_version	当前运行的nginx版本号
$pid	worker进程的PID
$query_string	与$args相同
$realpath_root	按root指令或alias指令算出的当前请求的绝对路径。其中的符号链接都会解析成真是文件路径

$remote_addr	客户端IP地址
$remote_port	客户端端口号
$remote_user	客户端用户名，认证用
$request	用户请求
$request_body	这个变量（0.7.58+）包含请求的主要信息。在使用proxy_pass或fastcgi_pass指令的location中比较有意义
$request_body_file	客户端请求主体信息的临时文件名
$request_completion	如果请求成功，设为"OK"；如果请求未完成或者不是一系列请求中最后一部分则设为空
$request_filename	当前请求的文件路径名，比如/opt/nginx/www/test.php
$request_method	请求的方法，比如"GET"、"POST"等
$request_uri	请求的URI，带参数
$scheme	所用的协议，比如http或者是https
$server_addr	服务器地址，如果没有用listen指明服务器地址，使用这个变量将发起一次系统调用以取得地址(造成资源浪费)
$server_name	请求到达的服务器名
$server_port	请求到达的服务器端口号
$server_protocol	请求的协议版本，"HTTP/1.0"或"HTTP/1.1"
$uri	请求的URI，可能和最初的值有不同，比如经过重定向之类的

其实这还不是全部，`Nginx` 在不停迭代更新是一个原因，还有一个是有些变量太冷门，借助它们，会有很多玩法。

首先，在 `OpenResty` 中如何引用这些变量呢？参考 ngx.var.VARIABLE 小节。

利用这些内置变量，来做一个简单的数学求和运算例子：

```
    server {
        listen       80;
        server_name  localhost;

        location /sum {
            #处理业务
           content_by_lua_block {
                local a = tonumber(ngx.var.arg_a) or 0
                local b = tonumber(ngx.var.arg_b) or 0
                ngx.say("sum: ", a + b )
            }
        }
    }
```

验证一下：

```
➜  ~  curl 'http://127.0.0.1/sum?a=11&b=12'
sum: 23
```

也许你笑了，这个 `API` 太简单没有实际意义。我们做个简易防火墙，看看如何开始玩耍。

参看下面示例代码：

```
    server {
        listen    80;
        server_name  localhost;

        location /sum {
            # 使用access阶段完成准入阶段处理
            access_by_lua_block {
                local black_ips = {["127.0.0.1"]=true}

                local ip = ngx.var.remote_addr
                if true == black_ips[ip] then
                    ngx.exit(ngx.HTTP_FORBIDDEN)
                end
            };

            #处理业务
           content_by_lua_block {
                local a = tonumber(ngx.var.arg_a) or 0
                local b = tonumber(ngx.var.arg_b) or 0
                ngx.say("sum:", a + b )
            }
        }
    }
```

运行测试：

```
➜  ~  curl '192.168.1.104/sum?a=11&b=12'
sum:23
➜  ~
➜  ~
➜  ~  curl '127.0.0.1/sum?a=11&b=12'
<html>
<head><title>403 Forbidden</title></head>
<body bgcolor="white">
<center><h1>403 Forbidden</h1></center>
<hr><center>openresty/1.9.3.1</center>
</body>
</html>
```

通过测试结果看到，提取了终端的 `IP` 地址后进行限制。扩充一下，就可以支持 IP 地址段，如果再与系统 `iptables` 进行配合，那么就足以达到软防火墙的目的。

目前为止，所有的例子都是对 `Nginx` 内置变量的获取，是否可以对其进行设置呢？其实大多数内容都是不允许写入的，例如刚刚的终端 `IP` 地址，在请求中是不允许对其进行更新的。对于可写的变量中的 `limit_rate` ，值得一提，它能完成传输速率限制，并且它的影响是单个请求级别。

参看下面示例：

```
        location /download {
            access_by_lua_block {
                ngx.var.limit_rate = 1000
            };
        }
```

下载测试：

```
➜  ~  wget '127.0.0.1/download/1.cab'
--2015-09-13 13:59:51--  http://127.0.0.1/download/1.cab
Connecting to 127.0.0.1... connected.
HTTP request sent, awaiting response... 200 OK
Length: 135802 (133K) [application/octet-stream]
Saving to: '1.cab'

1.cab               6%[===>             ]   8.00K  1.01KB/s   eta 1m 53s
```

子查询

Nginx 子请求是一种非常强有力的方式，它可以发起非阻塞的内部请求访问目标 location。目标 location 可以是配置文件中其他文件目录，或 任何 其他 nginx C 模块，包括 `ngx_proxy` 、 `ngx_fastcgi` 、 `ngx_memc` 、 `ngx_postgres` 、 `ngx_drizzle` ，甚至 ngx_lua 自身等等 。

需要注意的是，子请求只是模拟 HTTP 接口的形式， 没有 额外的 HTTP/TCP 流量，也 没有 IPC (进程间通信) 调用。所有工作在内部高效地在 C 语言级别完成。

子请求与 HTTP 301/302 重定向指令 (通过 ngx.redirect) 完全不同，也与内部重定向 ((通过 ngx.exec) 完全不同。

在发起子请求前，用户程序应总是读取完整的 HTTP 请求体 (通过调用 ngx.req.read_body 或设置 lua_need_request_body 指令为 on).

该 API 方法（ngx.location.capture_multi 也一样）总是缓冲整个请求体到内存中。因此，当需要处理一个大的子请求响应，用户程序应使用 cosockets 进行流式处理，

下面是一个简单例子：

```
res = ngx.location.capture(uri)
```

返回一个包含四个元素的 Lua 表 (`res.status` , `res.header` , `res.body` , 和 `res.truncated`)。

`res.status` (状态) 保存子请求的响应状态码。

`res.header` (头) 用一个标准 Lua 表储子请求响应的所有头信息。如果是“多值”响应头，这些值将使用 Lua (数组) 表顺序存储。例如，如果子请求响应头包含下面的行：

```
Set-Cookie: a=3
Set-Cookie: foo=bar
Set-Cookie: baz=blah
```

则 `res.header["Set-Cookie"]` 将存储 Lua 表 `{"a=3", "foo=bar", "baz=blah"}` 。

`res.body` (体) 保存子请求的响应体数据，它可能被截断。用户需要检测 `res.truncated` (截断) 布尔值标记来判断 `res.body` 是否包含截断的数据。这种数据截断的原因只可能是因为子请求发生了不可恢复的错误，例如远端在发送响应体时过早中断了连接，或子请求在接收远端响应体时超时。

URI 请求串可以与 URI 本身连在一起，例如，

```
res = ngx.location.capture('/foo/bar?a=3&b=4')
```

因为 Nginx 内核限制，子请求不允许类似 `@foo` 命名 location。请使用标准 location，并设置 `internal` 指令，仅服务内部请求。

例如，发送一个 POST 子请求，可以这样做：

```
res = ngx.location.capture(
    '/foo/bar',
    { method = ngx.HTTP_POST, body = 'hello, world' }
)
```

除了 POST 的其他 HTTP 请求方法请参考 HTTP method constants。 `method` 选项默认值是 `ngx.HTTP_GET` 。

`args` 选项可以设置附加的 URI 参数，例如：

```
ngx.location.capture('/foo?a=1',
    { args = { b = 3, c = ':' } }
)
```

等同于

```
ngx.location.capture('/foo?a=1&b=3&c=%3a')
```

也就是说，这个方法将根据 URI 规则转义参数键和值，并将它们拼接在一起组成一个完整的请求串。 `args` 选项要求的 Lua 表的格式与 ngx.encode_args 方法中使用的完全相同。

`args` 选项也可以直接包含 (转义过的) 请求串：

```
ngx.location.capture('/foo?a=1',
    { args = 'b=3&c=%3a' } }
)
```

这个例子与上个例子的功能相同。

请注意，通过 ngx.location.capture 创建的子请求默认继承当前请求的所有请求头信息，这有可能导致子请求响应中不可预测的副作用。例如，当使用标准的 `ngx_proxy` 模块服务子请求时，如果主请求头中包含 "Accept-Encoding: gzip"，可能导致子请求返回 Lua 代码无法正确处理的 gzip 压缩过的结果。通过设置 proxy_pass_request_headers 为 `off` ，在子请求 location 中忽略原始请求头。

注：ngx.location.capture 和 ngx.location.capture_multi 指令无法抓取包含以下指令的 location： add_before_body, add_after_body, auth_request, echo_location, echo_location_async, echo_subrequest, 或 echo_subrequest_async 。

```
location /foo {
    content_by_lua_block {
        res = ngx.location.capture("/bar")
    }
}
location /bar {
    echo_location /blah;
}
location /blah {
    echo "Success!";
}
```

```
$ curl -i http://example.com/foo
```

他们将不会按照预期工作。

不同阶段共享变量

在 OpenResty 的体系中，可以通过共享内存的方式完成不同工作进程的数据共享，可以通过 Lua 模块方式完成单个进程内不同请求的数据共享。如何完成单个请求内不同阶段的数据共享呢？最典型的例子，估计就是在 log 阶段记录一些请求的特殊变量。

ngx.ctx 表就是为了解决这类问题而设计的。参考下面例子：

```
location /test {
     rewrite_by_lua_block {
         ngx.ctx.foo = 76
     }
     access_by_lua_block {
         ngx.ctx.foo = ngx.ctx.foo + 3
     }
     content_by_lua_block {
         ngx.say(ngx.ctx.foo)
     }
 }
```

首先 ngx.ctx 是一个表，所以我们可以对他添加、修改。它用来存储基于请求的 Lua 环境数据，其生存周期与当前请求相同 (类似 Nginx 变量)。它有一个最重要的特性：单个请求内的 rewrite (重写)，access (访问)，和 content (内容) 等各处理阶段是保持一致的。

额外注意，每个请求，包括子请求，都有一份自己的 ngx.ctx 表。例如：

```
 location /sub {
     content_by_lua_block {
         ngx.say("sub pre: ", ngx.ctx.blah)
         ngx.ctx.blah = 32
         ngx.say("sub post: ", ngx.ctx.blah)
     }
 }

 location /main {
     content_by_lua_block {
         ngx.ctx.blah = 73
         ngx.say("main pre: ", ngx.ctx.blah)
         local res = ngx.location.capture("/sub")
         ngx.print(res.body)
         ngx.say("main post: ", ngx.ctx.blah)
     }
 }
```

访问 GET /main 输出

```
main pre: 73
sub pre: nil
sub post: 32
main post: 73
```

任意数据值，包括 Lua 闭包与嵌套表，都可以被插入这个“魔法”表，也允许注册自定义元方法。

也可以将 ngx.ctx 覆盖为一个新 Lua 表，例如，

```
ngx.ctx = { foo = 32, bar = 54 }
```

`ngx.ctx` 表查询需要相对昂贵的元方法调用，这比通过用户自己的函数参数直接传递基于请求的数据要慢得多。所以不要为了节约用户函数参数而滥用此 API，因为它可能对性能有明显影响。

由于 ngx.ctx 保存的是指定请求资源，所以这个变量是不能直接共享给其他请求使用的。

防止 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。具体来说，它是利用现有应用程序，将（恶意）的 SQL 命令注入到后台数据库引擎执行的能力，它可以通过在 Web 表单中输入（恶意）SQL 语句得到一个存在安全漏洞的网站上的数据库，而不是按照设计者意图去执行 SQL 语句。比如先前的很多影视网站泄露 VIP 会员密码大多就是通过 Web 表单递交查询字符暴出的，这类表单特别容易受到 SQL 注入式攻击。

SQL 注入例子

下面给了一个完整的可复现的 SQL 注入例子，实际上注入的 SQL 语句写法有很多，下例是比较简单的。

```
location /test {
    content_by_lua_block {
        local mysql = require "resty.mysql"
        local db, err = mysql:new()
        if not db then
            ngx.say("failed to instantiate mysql: ", err)
            return
        end

        db:set_timeout(1000) -- 1 sec

        local ok, err, errno, sqlstate = db:connect{
            host = "127.0.0.1",
            port = 3306,
            database = "ngx_test",
            user = "ngx_test",
            password = "ngx_test",
            max_packet_size = 1024 * 1024 }

        if not ok then
            ngx.say("failed to connect: ", err, ": ", errno, " ", sqlstate)
            return
        end

        ngx.say("connected to mysql.")

        local res, err, errno, sqlstate =
            db:query("drop table if exists cats")
        if not res then
            ngx.say("bad result: ", err, ": ", errno, ": ", sqlstate, ".")
            return
        end
```

```
        res, err, errno, sqlstate =
            db:query("create table cats "
                     .. "(id serial primary key, "
                     .. "name varchar(5))")
        if not res then
            ngx.say("bad result: ", err, ": ", errno, ": ", sqlstate, ".")
            return
        end

        ngx.say("table cats created.")

        res, err, errno, sqlstate =
            db:query("insert into cats (name) "
                     .. "values (\'Bob\'),(\'\'),(null)")
        if not res then
            ngx.say("bad result: ", err, ": ", errno, ": ", sqlstate, ".")
            return
        end

        ngx.say(res.affected_rows, " rows inserted into table cats ",
                "(last insert id: ", res.insert_id, ")")

        -- 这里有 SQL 注入（后面的 drop 操作）
        local req_id = [[1'; drop table cats;--]]
        res, err, errno, sqlstate =
            db:query(string.format([[select * from cats where id = '%s']], req_id))
        if not res then
            ngx.say("bad result: ", err, ": ", errno, ": ", sqlstate, ".")
            return
        end

        local cjson = require "cjson"
        ngx.say("result: ", cjson.encode(res))

        -- 再次查询，table 被删
        res, err, errno, sqlstate =
            db:query([[select * from cats where id = 1]])
        if not res then
            ngx.say("bad result: ", err, ": ", errno, ": ", sqlstate, ".")
            return
        end

        db:set_keepalive(10000, 100)
    }
}
```

其他变种，大家可以自行爬行搜索引擎了解。

OpenResty 中如何规避

其实大家可以大概网络爬行一下看看如何解决 SQL 注入，可以发现实现方法很多，比如替换各种关键字等。在 OpenResty 中，其实就简单很多了，只需要对输入参数进行一层过滤即可。

对于 MySQL ，可以调用 `ndk.set_var.set_quote_sql_str` ，进行一次过滤即可。

```
-- for MySQL
local req_id = [[1'; drop table cats;--]]
res, err, errno, sqlstate =
    db:query(string.format([[select * from cats where id = %s]],
    ndk.set_var.set_quote_sql_str(req_id)))
if not res then
    ngx.say("bad result: ", err, ": ", errno, ": ", sqlstate, ".")
    return
end
```

如果恰巧你使用的是 PostgreSQL ，调用 `ndk.set_var.set_quote_pgsql_str` 过滤输入变量。读者这时候可以再次把这段代码放到刚刚的示例代码中，如果您可以得到下面的错误，恭喜您，以正确的姿势防止 SQL 注入。

```
bad result: You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near
'1\'; drop table cats;--''' at line 1: 1064: 42000.
```

如何发起新 HTTP 请求

OpenResty 最主要的应用场景之一是 API Server，有别于传统 Nginx 的代理转发应用场景，API Server 中心内部有各种复杂的交易流程和判断逻辑，学会高效的与其他 HTTP Server 调用是必备基础。本文将介绍 OpenResty 中两个最常见 HTTP 接口调用方法。

我们先来模拟一个接口场景，一个公共服务专门用来对外提供加了“盐” md5 计算，业务系统调用这个公共服务完成业务逻辑，用来判断请求本身是否合法。

利用 proxy_pass

参考下面示例，利用 proxy_pass 完成 HTTP 接口访问的成熟配置+调用方法。

```
http {
    upstream md5_server{
        server 127.0.0.1:81;        # ①
        keepalive 20;               # ②
    }

    server {
        listen    80;

        location /test {
            content_by_lua_block {
                ngx.req.read_body()
                local args, err = ngx.req.get_uri_args()

                -- ③
                local res = ngx.location.capture('/spe_md5',
                    {
                        method = ngx.HTTP_POST,
                        body = args.data
                    }
                )

                if 200 ~= res.status then
                    ngx.exit(res.status)
                end

                if args.key == res.body then
                    ngx.say("valid request")
                else
                    ngx.say("invalid request")
                end
            }
        }
```

```
        location /spe_md5 {
            proxy_pass http://md5_server;   -- ④
            #For HTTP, the proxy_http_version directive should be set to “1.1” and the
 “Connection”
            #header field should be cleared.（from:http://nginx.org/en/docs/http/ngx_ht
tp_upstream_module.html#keepalive)
            proxy_http_version 1.1;
            proxy_set_header Connection "";
        }
    }

    server {
        listen    81;           -- ⑤

        location /spe_md5 {
            content_by_lua_block {
                ngx.req.read_body()
                local data = ngx.req.get_body_data()
                ngx.print(ngx.md5(data .. "*&^%$#$^&kjtrKUYG"))
            }
        }
    }
}
```

重点说明：① 上游访问地址清单(可以按需配置不同的权重规则)；② 上游访问长连接，是否开启长连接，对整体性能影响比较大（大家可以实测一下）；③ 接口访问通过 `ngx.location.capture` 的子查询方式发起；④ 由于 `ngx.location.capture` 方式只能是 nginx 自身的子查询，需要借助 proxy_pass 发出 HTTP 连接信号；⑤ 公共 API 输出服务；

这里大家可以看到，借用 nginx 周边成熟组件力量，为了发起一个 HTTP 请求，我们需要绕好几个弯子，甚至还有可能踩到坑（upstream 中长连接的细节处理），显然没有足够优雅，所以我们继续看下一章节。

利用 cosocket

立马开始我们的新篇章，给大家展示优雅的解决方式。

```
http {
    server {
        listen    80;

        location /test {
            content_by_lua_block {
                ngx.req.read_body()
                local args, err = ngx.req.get_uri_args()

                local http = require "resty.http"   -- ①
                local httpc = http.new()
                local res, err = httpc:request_uri( -- ②
                    "http://127.0.0.1:81/spe_md5",
                        {
                        method = "POST",
                        body = args.data,
                      }
                )

                if 200 ~= res.status then
                    ngx.exit(res.status)
                end

                if args.key == res.body then
                    ngx.say("valid request")
                else
                    ngx.say("invalid request")
                end
            }
        }
    }

    server {
        listen    81;

        location /spe_md5 {
            content_by_lua_block {
                ngx.req.read_body()
                local data = ngx.req.get_body_data()
                ngx.print(ngx.md5(data .. "*&^%$#$^&kjtrKUYG"))
            }
        }
    }
}
```

重点解释：① 引用 `resty.http` 库资源，它来自 github https://github.com/pintsized/lua-resty-http。② 参考 `resty-http` 官方 wiki 说明，我们可以知道 request_uri 函数完成了连接池、HTTP 请求等一系列动作。

题外话，为什么这么简单的方法我们还要求助外部开源组件呢？其实我也觉得这个功能太基础了，真的应该集成到 OpenResty 官方包里面，只不过目前官方默认包里还没有。

如果你的内部请求比较少，使用 `ngx.location.capture` + `proxy_pass` 的方式还没什么问题。但如果你的请求数量比较多，或者需要频繁的修改上游地址，那么 `resty.http` 就更适合你。

另外 `ngx.thread.*` 与 `resty.http` 相互结合也是很不错的玩法，推荐大家有时间研究一下。

访问有授权验证的 Redis

对于有授权验证的 Redis，正确的认证方法，请参考下面例子：

```
server {
    location /test {
        content_by_lua_block {
            local redis = require "resty.redis"
            local red = redis:new()

            red:set_timeout(1000) -- 1 sec

            local ok, err = red:connect("127.0.0.1", 6379)
            if not ok then
                ngx.say("failed to connect: ", err)
                return
            end

            -- 请注意这里 auth 的调用过程
            local count
            count, err = red:get_reused_times()
            if 0 == count then
                ok, err = red:auth("password")
                if not ok then
                    ngx.say("failed to auth: ", err)
                    return
                end
            elseif err then
                ngx.say("failed to get reused times: ", err)
                return
            end

            ok, err = red:set("dog", "an animal")
            if not ok then
                ngx.say("failed to set dog: ", err)
                return
            end

            ngx.say("set result: ", ok)

            -- 连接池大小是100个，并且设置最大的空闲时间是 10 秒
            local ok, err = red:set_keepalive(10000, 100)
            if not ok then
                ngx.say("failed to set keepalive: ", err)
                return
            end
        }
    }
}
```

这里解释一下 `tcpsock:getreusedtimes()` 方法，如果当前连接不是从内建连接池中获取的，该方法总是返回 0 ，也就是说，该连接还没有被使用过。如果连接来自连接池，那么返回值永远都是非零。所以这个方法可以用来确认当前连接是否来自池子。

对于 Redis 授权，实际上只需要建立连接后，首次认证一下，后面只需直接使用即可。换句话说，从连接池中获取的连接都是经过授权认证的，只有新创建的连接才需要进行授权认证。所以大家就看到了 `count, err = red:get_reused_times()` 这段代码，并有了下面 `if 0 == count then` 的判断逻辑。

select + set_keepalive 组合操作引起的数据读写错误

在高并发编程中，必须要使用连接池技术，通过减少建连、拆连次数来提高通讯速度。

错误示例代码：

```
local redis = require "resty.redis"
local red = redis:new()

red:set_timeout(1000) -- 1 sec

-- or connect to a unix domain socket file listened
-- by a redis server:
--     local ok, err = red:connect("unix:/path/to/redis.sock")

local ok, err = red:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("failed to connect: ", err)
    return
end

ok, err = red:select(1)
if not ok then
    ngx.say("failed to select db: ", err)
    return
end

ngx.say("select result: ", ok)

ok, err = red:set("dog", "an animal")
if not ok then
    ngx.say("failed to set dog: ", err)
    return
end

ngx.say("set result: ", ok)

-- put it into the connection pool of size 100,
-- with 10 seconds max idle time
local ok, err = red:set_keepalive(10000, 100)
if not ok then
    ngx.say("failed to set keepalive: ", err)
    return
end
```

如果单独执行这个用例，没有任何问题，用例是成功的。但是这段“没问题”的代码，却导致了诡异的现象。

大部分 Redis 请求的代码应该是类似这样的：

```
local redis = require "resty.redis"
local red = redis:new()

red:set_timeout(1000) -- 1 sec

-- or connect to a unix domain socket file listened
-- by a redis server:
--     local ok, err = red:connect("unix:/path/to/redis.sock")

local ok, err = red:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("failed to connect: ", err)
    return
end

ok, err = red:set("cat", "an animal too")
if not ok then
    ngx.say("failed to set cat: ", err)
    return
end

ngx.say("set result: ", ok)

-- put it into the connection pool of size 100,
-- with 10 seconds max idle time
local ok, err = red:set_keepalive(10000, 100)
if not ok then
    ngx.say("failed to set keepalive: ", err)
    return
end
```

这时候第二个示例代码在生产运行中，会出现 cat 偶会被写入到数据库 1 上，且几率大约 1% 左右。出错的原因在于错误示例代码使用了 `select(1)` 操作，并且使用了长连接，并潜伏在连接池中。当下一个请求刚好从连接池中把他选出来，又没有重置 `select(0)` 操作，后面所有的数据操作就都会默认触发在数据库 1 上了。

怎么解决这个问题？

1. 谁制造问题，谁把问题遗留尾巴擦干净；
2. 处理业务前，先把 `前辈` 的尾巴擦干净；

这里明显是第一个好，对吧。

Redis 接口的二次封装

> 先看一下官方的调用示例代码：

```
local redis = require "resty.redis"
local red = redis:new()

red:set_timeout(1000) -- 1 sec

local ok, err = red:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("failed to connect: ", err)
    return
end

ok, err = red:set("dog", "an animal")
if not ok then
    ngx.say("failed to set dog: ", err)
    return
end

ngx.say("set result: ", ok)

-- put it into the connection pool of size 100,
-- with 10 seconds max idle time
local ok, err = red:set_keepalive(10000, 100)
if not ok then
    ngx.say("failed to set keepalive: ", err)
    return
end
```

这是一个标准的 Redis 接口调用，如果你的代码中 Redis 被调用频率不高，那么这段代码不会有任何问题。但如果你的项目重度依赖 Redis，工程中有大量的代码在重复创建连接-->数据操作-->关闭连接（或放到连接池）这个完整的链路调用完毕，甚至还要考虑不同的 return 情况做不同处理，就很快发现代码中有大量的重复。

`Lua` 是不支持面向对象的。很多人用尽各种招术利用元表来模拟。可是，`Lua` 的发明者似乎不想看到这样的情形，因为他们把取长度的 `__len` 方法以及析构函数 `__gc` 留给了 C API，纯 Lua 只能望洋兴叹。

> 我们期望的代码应该是这样的：

```
local red = redis:new()
local ok, err = red:set("dog", "an animal")
if not ok then
    ngx.say("failed to set dog: ", err)
    return
end

ngx.say("set result: ", ok)

local res, err = red:get("dog")
if not res then
    ngx.say("failed to get dog: ", err)
    return
end

if res == ngx.null then
    ngx.say("dog not found.")
    return
end

ngx.say("dog: ", res)
```

期望它自身具备以下几个特征：

- new、connect 函数合体，使用时只负责申请，尽量少关心什么时候具体连接、释放；
- 默认 Redis 数据库连接地址，但是允许自定义；
- 每次 Redis 使用完毕，自动释放 Redis 连接到连接池供其他请求复用；
- 要支持 Redis 的重要优化手段 pipeline；

> 不卖关子，只要干货，我们最后是这样干的，可以这里看到 gist代码

```
-- file name: resty/redis_iresty.lua
local redis_c = require "resty.redis"


local ok, new_tab = pcall(require, "table.new")
if not ok or type(new_tab) ~= "function" then
    new_tab = function (narr, nrec) return {} end
end


local _M = new_tab(0, 155)
_M._VERSION = '0.01'


local commands = {
    "append",            "auth",              "bgrewriteaof",
    "bgsave",            "bitcount",          "bitop",
    "blpop",             "brpop",
    "brpoplpush",        "client",            "config",
```

```
    "dbsize",
    "debug",            "decr",             "decrby",
    "del",              "discard",          "dump",
    "echo",
    "eval",             "exec",             "exists",
    "expire",           "expireat",         "flushall",
    "flushdb",          "get",              "getbit",
    "getrange",         "getset",           "hdel",
    "hexists",          "hget",             "hgetall",
    "hincrby",          "hincrbyfloat",     "hkeys",
    "hlen",
    "hmget",             "hmset",      "hscan",
    "hset",
    "hsetnx",           "hvals",            "incr",
    "incrby",           "incrbyfloat",      "info",
    "keys",
    "lastsave",         "lindex",           "linsert",
    "llen",             "lpop",             "lpush",
    "lpushx",           "lrange",           "lrem",
    "lset",             "ltrim",            "mget",
    "migrate",
    "monitor",          "move",             "mset",
    "msetnx",           "multi",            "object",
    "persist",          "pexpire",          "pexpireat",
    "ping",             "psetex",           "psubscribe",
    "pttl",
    "publish",      --[[ "punsubscribe", ]]   "pubsub",
    "quit",
    "randomkey",        "rename",           "renamenx",
    "restore",
    "rpop",             "rpoplpush",        "rpush",
    "rpushx",           "sadd",             "save",
    "scan",             "scard",            "script",
    "sdiff",            "sdiffstore",
    "select",           "set",              "setbit",
    "setex",            "setnx",            "setrange",
    "shutdown",         "sinter",           "sinterstore",
    "sismember",        "slaveof",          "slowlog",
    "smembers",         "smove",            "sort",
    "spop",             "srandmember",      "srem",
    "sscan",
    "strlen",       --[[ "subscribe",  ]]     "sunion",
    "sunionstore",      "sync",             "time",
    "ttl",
    "type",         --[[ "unsubscribe", ]]    "unwatch",
    "watch",            "zadd",             "zcard",
    "zcount",           "zincrby",          "zinterstore",
    "zrange",           "zrangebyscore",    "zrank",
    "zrem",             "zremrangebyrank",  "zremrangebyscore",
    "zrevrange",        "zrevrangebyscore", "zrevrank",
    "zscan",
    "zscore",           "zunionstore",      "evalsha"
}
```

```
local mt = { __index = _M }


local function is_redis_null( res )
    if type(res) == "table" then
        for k,v in pairs(res) do
            if v ~= ngx.null then
                return false
            end
        end
        return true
    elseif res == ngx.null then
        return true
    elseif res == nil then
        return true
    end

    return false
end


-- change connect address as you need
function _M.connect_mod( self, redis )
    redis:set_timeout(self.timeout)
    return redis:connect("127.0.0.1", 6379)
end


function _M.set_keepalive_mod( redis )
    -- put it into the connection pool of size 100, with 60 seconds max idle time
    return redis:set_keepalive(60000, 1000)
end


function _M.init_pipeline( self )
    self._reqs = {}
end


function _M.commit_pipeline( self )
    local reqs = self._reqs

    if nil == reqs or 0 == #reqs then
        return {}, "no pipeline"
    else
        self._reqs = nil
    end

    local redis, err = redis_c:new()
    if not redis then
        return nil, err
```

```
    end

    local ok, err = self:connect_mod(redis)
    if not ok then
        return {}, err
    end

    redis:init_pipeline()
    for _, vals in ipairs(reqs) do
        local fun = redis[vals[1]]
        table.remove(vals , 1)

        fun(redis, unpack(vals))
    end

    local results, err = redis:commit_pipeline()
    if not results or err then
        return {}, err
    end

    if is_redis_null(results) then
        results = {}
        ngx.log(ngx.WARN, "is null")
    end
    -- table.remove (results , 1)

    self.set_keepalive_mod(redis)

    for i,value in ipairs(results) do
        if is_redis_null(value) then
            results[i] = nil
        end
    end

    return results, err
end


function _M.subscribe( self, channel )
    local redis, err = redis_c:new()
    if not redis then
        return nil, err
    end

    local ok, err = self:connect_mod(redis)
    if not ok or err then
        return nil, err
    end

    local res, err = redis:subscribe(channel)
    if not res then
        return nil, err
    end
```

```
    res, err = redis:read_reply()
    if not res then
        return nil, err
    end

    redis:unsubscribe(channel)
    self.set_keepalive_mod(redis)

    return res, err
end


local function do_command(self, cmd, ... )
    if self._reqs then
        table.insert(self._reqs, {cmd, ...})
        return
    end

    local redis, err = redis_c:new()
    if not redis then
        return nil, err
    end

    local ok, err = self:connect_mod(redis)
    if not ok or err then
        return nil, err
    end

    local fun = redis[cmd]
    local result, err = fun(redis, ...)
    if not result or err then
        -- ngx.log(ngx.ERR, "pipeline result:", result, " err:", err)
        return nil, err
    end

    if is_redis_null(result) then
        result = nil
    end

    self.set_keepalive_mod(redis)

    return result, err
end


for i = 1, #commands do
    local cmd = commands[i]
    _M[cmd] =
            function (self, ...)
                return do_command(self, cmd, ...)
            end
end
```

```
function _M.new(self, opts)
    opts = opts or {}
    local timeout = (opts.timeout and opts.timeout * 1000) or 1000
    local db_index= opts.db_index or 0

    return setmetatable({
            timeout = timeout,
            db_index = db_index,
            _reqs = nil }, mt)
end


return _M
```

> 调用示例代码：

```
local redis = require "resty.redis_iresty"
local red = redis:new()

local ok, err = red:set("dog", "an animal")
if not ok then
    ngx.say("failed to set dog: ", err)
    return
end

ngx.say("set result: ", ok)
```

在最终的示例代码中看到，所有的连接创建、销毁连接、连接池部分，都被完美隐藏了，我们只需要业务就可以了。妈妈再也不用担心我把 Redis 搞垮了。

Todo list： 目前 `resty.redis` 并没有对 Redis 3.0 的集群 API 做支持，既然统一了 Redis 的入口、出口，那么对这个 `redis_iresty` 版本做适当调整完善，就可以支持 Redis 3.0 的集群协议。由于我们目前还没引入 Redis 集群，这里也希望有使用的同学贡献自己的补丁或文章。

Redis 接口的二次封装（发布订阅）

其实这一小节完全可以放到上一个小节，只是这里用了完全不同的玩法，所以我还是决定单拿出来分享一下这个小细节。

上一小节有关订阅部分的代码，请看：

```
function _M.subscribe( self, channel )
    local redis, err = redis_c:new()
    if not redis then
        return nil, err
    end

    local ok, err = self:connect_mod(redis)
    if not ok or err then
        return nil, err
    end

    local res, err = redis:subscribe(channel)
    if not res then
        return nil, err
    end

    res, err = redis:read_reply()
    if not res then
        return nil, err
    end

    redis:unsubscribe(channel)
    self.set_keepalive_mod(redis)

    return res, err
end
```

其实这里的实现是有问题的，各位看官，你能发现这段代码的问题么？给个提示，在高并发订阅场景下，极有可能存在漏掉部分订阅信息。原因在与每次订阅到内容后，都会把 Redis 对象进行释放，处理完订阅信息后再次去连接 Redis，在这个时间差里面，很可能有消息已经漏掉了。

正确的代码应该是这样的：

```
function _M.subscribe( self, channel )
    local redis, err = redis_c:new()
    if not redis then
        return nil, err
    end

    local ok, err = self:connect_mod(redis)
    if not ok or err then
        return nil, err
    end

    local res, err = redis:subscribe(channel)
    if not res then
        return nil, err
    end

    local function do_read_func ( do_read )
        if do_read == nil or do_read == true then
            res, err = redis:read_reply()
            if not res then
                return nil, err
            end
            return res
        end

        redis:unsubscribe(channel)
        self.set_keepalive_mod(redis)
        return
    end

    return do_read_func
end
```

调用示例代码：

```
local red     = redis:new({timeout=1000})
local func  = red:subscribe( "channel" )
if not func then
  return nil
end

while true do
    local res, err = func()
    if err then
        func(false)
    end
    ... ...
end

return cbfunc
```

pipeline 压缩请求数量

通常情况下，我们每个操作 Redis 的命令都以一个 TCP 请求发送给 Redis，这样的做法简单直观。然而，当我们有连续多个命令需要发送给 Redis 时，如果每个命令都以一个数据包发送给 Redis，将会降低服务端的并发能力。

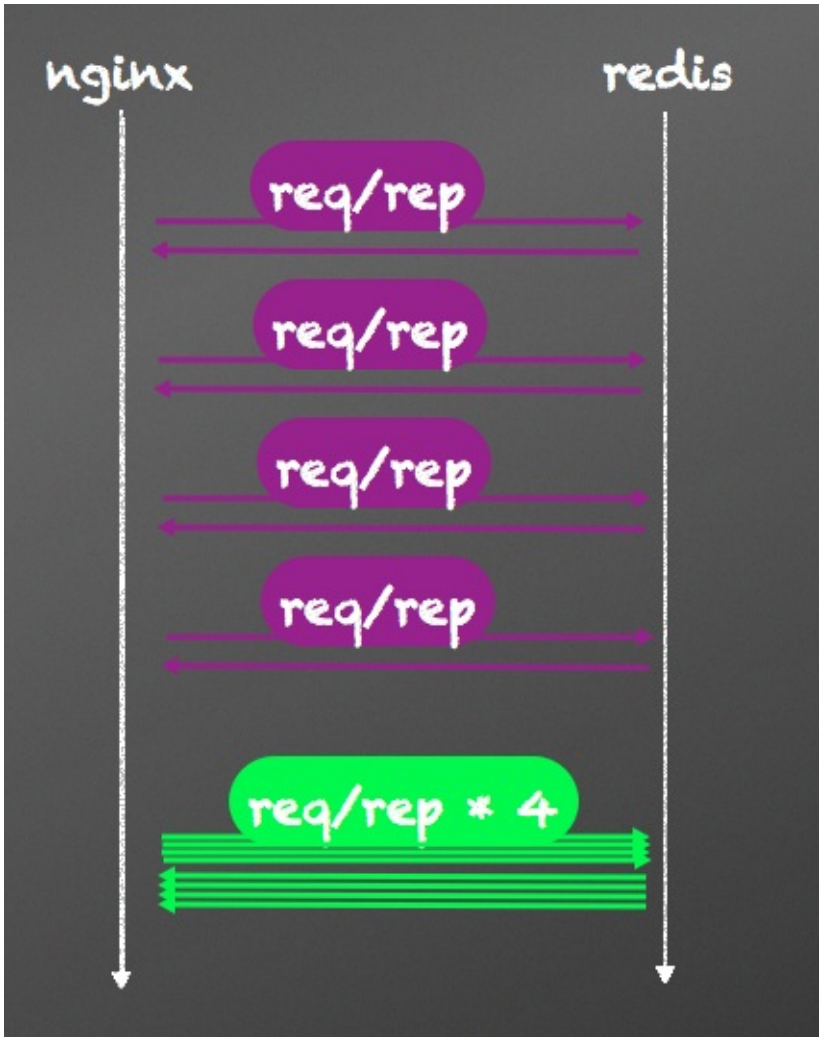
为什么呢？大家知道每发送一个 TCP 报文，会存在网络延时及操作系统的处理延时。大部分情况下，网络延时要远大于 CPU 的处理延时。如果一个简单的命令就以一个 TCP 报文发出，网络延时将成为系统性能瓶颈，使得服务端的并发数量上不去。

首先检查你的代码，是否明确完整使用了 Redis 的长连接机制。作为一个服务端程序员，要对长连接的使用有一定了解，在条件允许的情况下，一定要开启长连接。验证方式也比较简单，直接用 tcpdump 或 wireshark 抓包分析一下网络数据即可。

> set_keepalive的参数：按照业务正常运转的并发数量设置，不建议使用峰值情况设置。

如果我们确定开启了长连接，发现这时候 Redis 的 CPU 的占用率还是不高，在这种情况下，就要从 Redis 的使用方法上进行优化。

如果我们可以把所有单次请求，压缩到一起，如下图：



很庆幸 Redis 早就为我们准备好了这道菜，就等着我们吃了，这道菜就叫 `pipeline`。pipeline 机制将多个命令汇聚到一个请求中，可以有效减少请求数量，减少网络延时。下面是对比使用 pipeline 的一个例子：

```
# you do not need the following line if you are using
    # the ngx_openresty bundle:
    lua_package_path "/path/to/lua-resty-redis/lib/?.lua;;";

    server {
        location /withoutpipeline {
           content_by_lua_block {
                local redis = require "resty.redis"
                local red = redis:new()

                red:set_timeout(1000) -- 1 sec

                -- or connect to a unix domain socket file listened
                -- by a redis server:
                --     local ok, err = red:connect("unix:/path/to/redis.sock")

                local ok, err = red:connect("127.0.0.1", 6379)
                if not ok then
                    ngx.say("failed to connect: ", err)
```

```
                return
            end

            local ok, err = red:set("cat", "Marry")
            ngx.say("set result: ", ok)
            local res, err = red:get("cat")
            ngx.say("cat: ", res)

            ok, err = red:set("horse", "Bob")
            ngx.say("set result: ", ok)
            res, err = red:get("horse")
            ngx.say("horse: ", res)


            -- put it into the connection pool of size 100,
            -- with 10 seconds max idle time
            local ok, err = red:set_keepalive(10000, 100)
            if not ok then
                ngx.say("failed to set keepalive: ", err)
                return
            end
        }
    }

    location /withpipeline {
        content_by_lua_block {
            local redis = require "resty.redis"
            local red = redis:new()

            red:set_timeout(1000) -- 1 sec

            -- or connect to a unix domain socket file listened
            -- by a redis server:
            --     local ok, err = red:connect("unix:/path/to/redis.sock")

            local ok, err = red:connect("127.0.0.1", 6379)
            if not ok then
                ngx.say("failed to connect: ", err)
                return
            end

            red:init_pipeline()
            red:set("cat", "Marry")
            red:set("horse", "Bob")
            red:get("cat")
            red:get("horse")
            local results, err = red:commit_pipeline()
            if not results then
                ngx.say("failed to commit the pipelined requests: ", err)
                return
            end

            for i, res in ipairs(results) do
```

```
                    if type(res) == "table" then
                        if not res[1] then
                            ngx.say("failed to run command ", i, ": ", res[2])
                        else
                            -- process the table value
                        end
                    else
                        -- process the scalar value
                    end
                end

                -- put it into the connection pool of size 100,
                -- with 10 seconds max idle time
                local ok, err = red:set_keepalive(10000, 100)
                if not ok then
                    ngx.say("failed to set keepalive: ", err)
                    return
                end
            }
        }
    }
```

在我们实际应用场景中，正确使用 pipeline 对性能的提升十分明显。我们曾经某个后台应用，逐个处理大约 100 万条记录需要几十分钟，经过 pileline 压缩请求数量后，最后时间缩小到 20 秒左右。做之前能预计提升性能，但是没想到提升如此巨大。

script 压缩复杂请求

从pipeline章节，我们知道对于多个简单的 Redis 命令可以汇聚到一个请求中，提升服务端的并发能力。然而，在有些场景下，我们每次命令的输入需要引用上个命令的输出，甚至可能还要对第一个命令的输出做一些加工，再把加工结果当成第二个命令的输入。pipeline 难以处理这样的场景。庆幸的是，我们可以用 Redis 里的 script 来压缩这些复杂命令。

script 的核心思想是在 Redis 命令里嵌入 Lua 脚本，来实现一些复杂操作。Redis 中和脚本相关的命令有：

```
- EVAL
- EVALSHA
- SCRIPT EXISTS
- SCRIPT FLUSH
- SCRIPT KILL
- SCRIPT LOAD
```

官网上给出了这些命令的基本语法，感兴趣的同学可以到这里查阅。其中 EVAL 的基本语法如下：

> EVAL script numkeys key [key ...] arg [arg ...]

EVAL 的第一个参数 *script* 是一段 Lua 脚本程序。这段 Lua 脚本不需要（也不应该）定义函数。它运行在 Redis 服务器中。 EVAL 的第二个参数 *numkeys* 是参数的个数，后面的参数 *key*（从第三个参数），表示在脚本中所用到的那些 Redis 键(key)，这些键名参数可以在 Lua 中通过全局变量 KEYS 数组，用 1 为基址的形式访问(KEYS[1]，KEYS[2]，以此类推)。 在命令的最后，那些不是键名参数的附加参数 *arg [arg ...]*，可以在 Lua 中通过全局变量 ARGV 数组访问，访问的形式和 KEYS 变量类似(ARGV[1]、ARGV[2]，诸如此类)。下面是执行 eval 命令的简单例子：

```
eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
1) "key1"
2) "key2"
3) "first"
4) "second"
```

OpenResty 中已经对 Redis 的所有原语操作进行了封装。下面我们以 EVAL 为例，来看一下 OpenResty 中如何利用 script 来压缩请求：

```
# you do not need the following line if you are using
    # the ngx_openresty bundle:
    lua_package_path "/path/to/lua-resty-redis/lib/?.lua;;";
```

```
server {
    location /usescript {
        content_by_lua_block {
        local redis = require "resty.redis"
        local red = redis:new()

        red:set_timeout(1000) -- 1 sec

        -- or connect to a unix domain socket file listened
        -- by a redis server:
        --     local ok, err = red:connect("unix:/path/to/redis.sock")

        local ok, err = red:connect("127.0.0.1", 6379)
        if not ok then
            ngx.say("failed to connect: ", err)
            return
        end

        --- use scripts in eval cmd
        local id = 1
        local res, err = red:eval([[
            -- 注意在 Redis 执行脚本的时候，从 KEYS/ARGV 取出来的值类型为 string
            local info = redis.call('get', KEYS[1])
            info = cjson.decode(info)
            local g_id = info.gid

            local g_info = redis.call('get', g_id)
            return g_info
            ]], 1, id)

        if not res then
           ngx.say("failed to get the group info: ", err)
           return
        end

        ngx.say(res)

        -- put it into the connection pool of size 100,
        -- with 10 seconds max idle time
        local ok, err = red:set_keepalive(10000, 100)
        if not ok then
            ngx.say("failed to set keepalive: ", err)
            return
        end

        -- or just close the connection right away:
        -- local ok, err = red:close()
        -- if not ok then
        --     ngx.say("failed to close: ", err)
        --     return
        -- end
        }
```

```
        }
    }
```

从上面的例子可以看到，我们要根据一个对象的 id 来查询该 id 所属 gourp 的信息时，我们的第一个命令是从 Redis 中读取 id 为 1（id 的值可以通过参数的方式传递到 script 中）的对象的信息（由于这些信息一般 json 格式存在 Redis 中，因此我们要做一个解码操作，将 info 转换成 Lua 对象）。然后提取信息中的 groupid 字段，以 groupid 作为 key 查询 groupinfo。这样我们就可以把两个 get 放到一个 TCP 请求中，做到减少 TCP 请求数量，减少网络延时的效果啦。

动态生成的 lua-resty-redis 模块方法

刚接触 lua-resty-redis 的文档的时候，你可能会惊讶于上面列出的方法之少。 Redis 有好几十个命令，而Method一节列出的方法却寥寥无几。 事实上，如果仔细阅读了文档，你会在 Method 一节的开头读到这么一段话：

> All of the Redis commands have their own methods with the same name except all in lower case. You can find the complete list of Redis commands here: http://redis.io/commands ... In addition to all those redis command methods, the following methods are also provided:

看来不是 lua-resty-redis 支持的方法少，而是大部分方法都不需要单独列出来。

动态语言，动态方法

其实，lua-resty-redis 并没有显式定义这一类跟 Redis 命令同名的方法。

熟悉 Redis 的人对 `redis.call` 应该不会感到陌生，这是 Redis Lua 脚本中调用 Redis 命令的唯一方法。 无论是什么 Redis 命令，你都可以通过它调用。 lua-resty-redis 内部就有一个类似于这样的方法，它负责把请求参数发给 Redis，然后处理来自 Redis 的响应。

出于易用性，lua-resty-redis 用 `$command(arg1, arg2)` 的形式封装了 `call($command, arg1, arg2)`。每次调用时可以少打四个字符呢。 由于动态语言支持动态生成方法，lua-resty-redis 并不用给每个命令补上一个对应的方法，它只需要：

```
-- 仅为示例，不是真正的实现
local cmds = {
    'get', 'set', ...
}

for i = 1, #cmds do
    local cmd = cmds[i]
    _M[cmd] = function(...)
        call(cmd, ...)
    end
end
```

现在，要想支持新的 Redis 命令，往 cmds 里加多一个字符串就好了。这就叫良好的拓展性。 当然，有些命令，比如 `subscribe`，需要额外的特殊处理。

动态方法，惰性生成

从 OpenResty 1.11.2 版本开始，lua-resty-redis 模块使用了一个巧妙的技巧，推迟到实际需要时才动态生成模块方法。 依靠惰性生成方法，要想支持新的 Redis 命令，大多数情况下 lua-resty-redis 连一个字符串都不用加。无需拓展，才是真正的“良好的拓展性”。

前面说到，动态语言支持动态生成方法，这不仅意味着可以动态地生成方法，也意味着可以在运行时 按需 生成方法。 跟其他动态语言一样，Lua 提供了一个方法，允许程序员在找不到对应方法时调用特定的处理逻辑，那就是 `__index` 。

`__index` 在前面的元表一章中已经介绍过了。跟“给表中的键附上默认值”类似，我们也可以给模块中的方法名附上默认实现。 所需的只是如下的代码：

```
setmetatable(_M, {__index = function(self, cmd)
    local method =
        function (self, ...)
            return call(self, cmd, ...)
        end

    -- cache the lazily generated method in our
    -- module table
    _M[cmd] = method
    return method
end})
```

现在我们可以不用准备一份超长的命令列表，也无需为用不到的命令付生成方法的开销，同时给未来的命令也留好了位置。 一切魔法均隐藏于代码之中，may the source be with you!

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

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）：

```
-- http://www.kyne.com.au/~mark/software/lua-cjson.php
-- version: 2.1 devel

local json = require("cjson")
local str  = [[ {"key:"value"} ]]

local t    = json.decode(str)
ngx.say(" --> ", type(t))


-- ... do the other things
ngx.say("all fine")
```

代码执行错误日志如下：

```
2015/06/27 00:01:42 [error] 2714#0: *25 lua entry thread aborted: runtime error: ...or
k/git/github.com/lua-resty-memcached-server/t/test.lua:8: Expected colon but found inv
alid token at character 9
stack traceback:
coroutine 0:
    [C]: in function 'decode'
    ...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: l
ocalhost, 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：

```
local json = require("cjson")

local function _json_decode(str)
  return json.decode(str)
end

function json_decode( str )
    local ok, t = pcall(_json_decode, str)
    if not ok then
      return nil
    end

    return t
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 或者或 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`。

```
local json = require("cjson.safe")
local str  = [[ {"key:"value"} ]]

local t    = json.decode(str)
if t then
    ngx.say(" --> ", type(t))
end
```

稀疏数组

请看示例代码（注意 data 的数组下标）：

```
-- http://www.kyne.com.au/~mark/software/lua-cjson.php
-- version: 2.1 devel

local json = require("cjson")

local data = {1, 2}
data[1000] = 99

-- ... do the other things
ngx.say(json.encode(data))
```

运行日志报错结果：

```
2015/06/27 00:23:13 [error] 2714#0: *40 lua entry thread aborted: runtime error: ...or
k/git/github.com/lua-resty-memcached-server/t/test.lua:13: Cannot serialise table: exc
essively sparse array
stack traceback:
coroutine 0:
    [C]: in function 'encode'
    ...ork/git/github.com/lua-resty-memcached-server/t/test.lua:13: in function <...or
k/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是怎样完成的。

编码为 array 还是 object

首先大家请看这段源码：

```
-- http://www.kyne.com.au/~mark/software/lua-cjson.php
-- version: 2.1 devel

local json = require("cjson")
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做容错。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。

```
-- 内容节选lua-cjson-2.1.0.2/tests/agentzh.t
=== TEST 1: empty tables as objects
--- lua
local cjson = require "cjson"
print(cjson.encode({}))
print(cjson.encode({dogs = {}}))
--- out
{}
{"dogs":{}}


=== TEST 2: empty tables as arrays
--- lua
local cjson = require "cjson"
cjson.encode_empty_table_as_object(false)
print(cjson.encode({}))
print(cjson.encode({dogs = {}}))
--- out
[]
{"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：

```
local json = require("cjson")
--稀疏数组会被处理成object
json.encode_sparse_array(true)

local function _json_encode(data)
    return json.encode(data)
end

function json_encode( data, empty_table_as_object )
    --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
    --cjson对于空的table，就会被处理为object，也就是{}
    --处理方法：对于cjson，使用encode_empty_table_as_object这个方法。
    json.encode_empty_table_as_object(empty_table_as_object or false) -- 空的table默认为
array
    local ok, json_value = pcall(_json_encode, data)
    if not ok then
        return nil
    end
    return json_value
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：

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

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。

PostgresNginxModule 模块的调用方式

ngx_postgres 模块使用方法

```
location /postgres {
    internal;

    default_type text/html;
    set_by_lua_block $query_sql {return ngx.unescape_uri(ngx.var.arg_sql)}

    postgres_pass   pg_server;
    rds_json          on;
    rds_json_buffer_size 16k;
    postgres_query  $query_sql;
    postgres_connect_timeout 1s;
    postgres_result_timeout 2s;
}
```

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：

```
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：

```
upstream pg_server {
    postgres_server  192.168.1.2:5432 dbname=pg_database
            user=postgres password=postgres;
    postgres_keepalive max=800 mode=single overflow=reject;
}
```

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）：

```
local json = require "cjson"

function test()
    local res = ngx.location.capture('/postgres',
        { args = {sql = "SELECT * FROM test" } }
    )

    local status = res.status
    local body = json.decode(res.body)

    if status == 200 then
        status = true
    else
        status = false
    end
    return status, body
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。

```
# you do not need the following line if you are using
# the ngx_openresty bundle:
lua_package_path "/path/to/lua-resty-mysql/lib/?.lua;;";

server {
    location /test {
       content_by_lua_block {
            local mysql = require "resty.mysql"
            local db, err = mysql:new()
            if not db then
                ngx.say("failed to instantiate mysql: ", err)
                return
            end

            db:set_timeout(1000) -- 1 sec

            local ok, err, errno, sqlstate = db:connect{
                host = "127.0.0.1",
                port = 3306,
                database = "ngx_test",
                user = "ngx_test",
                password = "ngx_test",
                max_packet_size = 1024 * 1024 }

            if not ok then
                ngx.say("failed to connect: ", err, ": ", errno, " ", sqlstate)
                return
            end
```

```
            ngx.say("connected to mysql.")

            -- run a select query, expected about 10 rows in
            -- the result set:
            res, err, errno, sqlstate =
                db:query("select * from cats order by id asc", 10)
            if not res then
                ngx.say("bad result: ", err, ": ", errno, ": ", sqlstate, ".")
                return
            end

            local cjson = require "cjson"
            ngx.say("result: ", cjson.encode(res))

            -- put it into the connection pool of size 100,
            -- with 10 seconds max idle timeout
            local ok, err = db:set_keepalive(10000, 100)
            if not ok then
                ngx.say("failed to set keepalive: ", err)
                return
            end
        }
    }
}
```

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

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 `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。

```
    local json = require "cjson"

    function db_exec(sql_str)
        local res = ngx.location.capture('/postgres',
            { args = {sql = sql_str } }
        )

        local status = res.status
        local body = json.decode(res.body)

        if status == 200 then
            status = true
        else
            status = false
        end
        return status, body
    end

    -- 转账操作，对ID=100的用户加10，同时对ID=200的用户减10。
?    local status
?    status = db_exec("BEGIN")
?    if status then
?        db_exec("ROLLBACK")
?    end
?
?    status = db_exec("UPDATE ACCOUNT SET MONEY=MONEY+10  WHERE ID = 100")
?    if status then
?        db_exec("ROLLBACK")
?    end
?
?    status = db_exec("UPDATE ACCOUNT SET MONEY=MONEY-10  WHERE ID = 200")
?    if status then
?        db_exec("ROLLBACK")
?    end
?
?    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，是不会有任何问题的。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：

```
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：

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

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：

```
location /postgres {
    internal;

    default_type text/html;
    set_by_lua_block $query_sql {return ngx.unescape_uri(ngx.var.arg_sql)}

    postgres_pass   pg_server;
    rds_json          on;
    rds_json_buffer_size 16k;
    postgres_query  $query_sql;
    postgres_connect_timeout 1s;
    postgres_result_timeout 2s;
}
```

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。

健康监测

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，

```
    local sql_normal = [[select id, name from user where name=']] .. ngx.var.arg_name
.. [[' and password=']] .. ngx.var.arg_password .. [[' limit 1;]]

    local res = ngx.location.capture('/postgres',
        { args = {sql = sql } }
    )

    local body = json.decode(res.body)

    if (table.getn(res) > 0) {
        return res[1];
    }

    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：

```
#    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：

```
    local sql_normal = [[select id, name from user where name='person' and password='1
2345' 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：

```
    name="' or ''='"
    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。

```
    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：

```
    ndk.set_var.set_quote_sql_str()
    ndk.set_var.set_quote_pgsql_str()
    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：

```
    local name = ngx.quote_sql_str(ngx.var.arg_name)
    local password = ngx.quote_sql_str(ngx.var.arg_password)
    local sql_normal = [[select id, name from user where name=]] .. name .. [[ and pas
sword=]] .. password .. [[ limit 1;]]

    local res = ngx.location.capture('/postgres',
        { args = {sql = sql } }
    )

    local body = json.decode(res.body)

    if (table.getn(res) > 0) {
        return res[1];
    }

    return nil;
```

注意上述代码有两个变化：

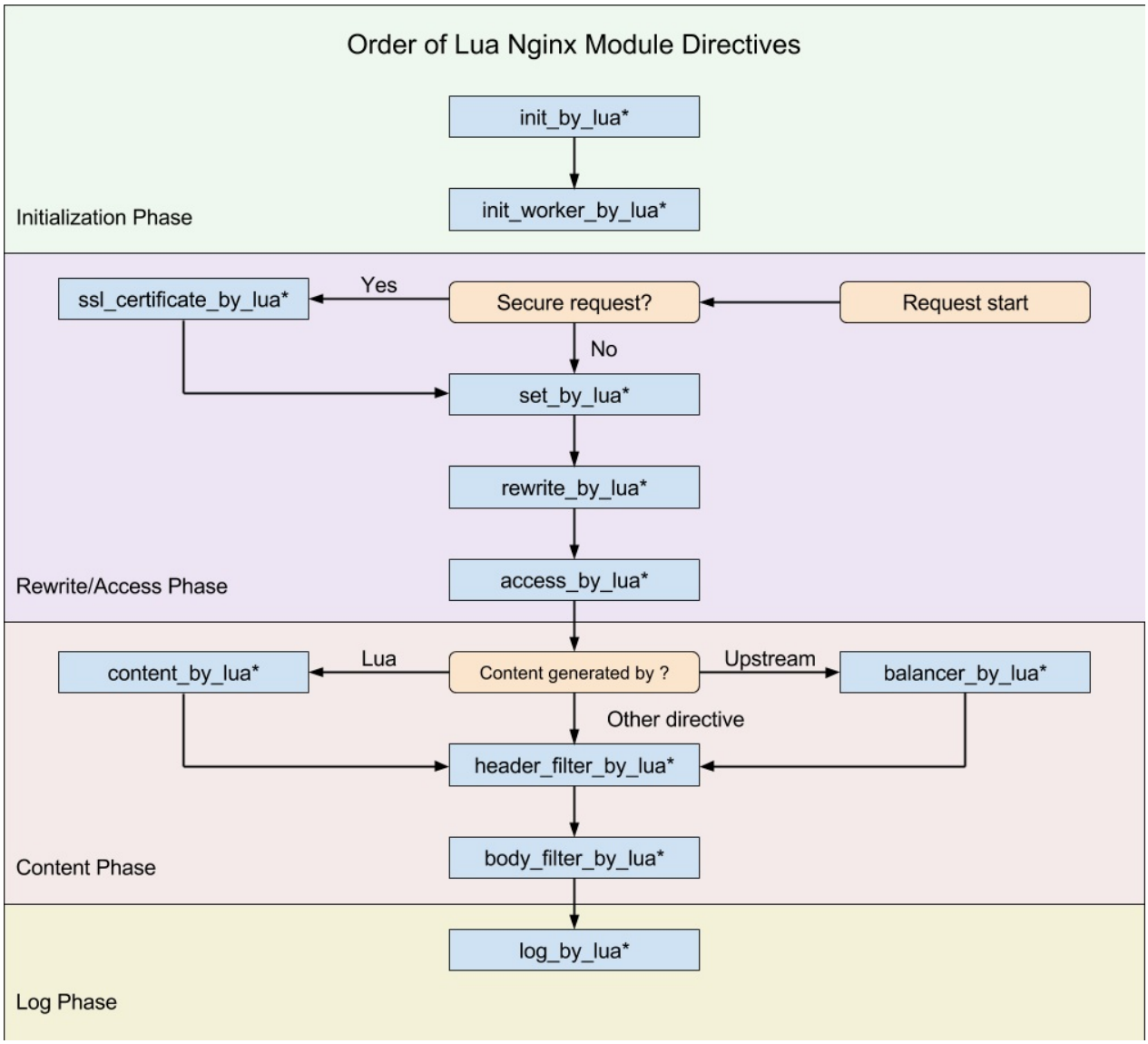
```
* 用 ngx.quote_sql_str 把 ngx.var.arg_name 和 ngx.var.arg_password 包了一层，把返回值作为 s
ql 语句拼凑起来。
* 原本在 sql 语句中添加的单引号去掉了，因为 ngx.quote_sql_str 的返回值正确的带上引号了。
```

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

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：

```
location /mixed {
    set_by_lua_block $a {
        ngx.log(ngx.ERR, "set_by_lua*")
    }
    rewrite_by_lua_block {
        ngx.log(ngx.ERR, "rewrite_by_lua*")
    }
    access_by_lua_block {
        ngx.log(ngx.ERR, "access_by_lua*")
    }
    content_by_lua_block {
        ngx.log(ngx.ERR, "content_by_lua*")
    }
    header_filter_by_lua_block {
        ngx.log(ngx.ERR, "header_filter_by_lua*")
    }
    body_filter_by_lua_block {
        ngx.log(ngx.ERR, "body_filter_by_lua*")
    }
    log_by_lua_block {
        ngx.log(ngx.ERR, "log_by_lua*")
    }
}
```

执行结果日志(截取了一下)：

```
set_by_lua*
rewrite_by_lua*
access_by_lua*
content_by_lua*
header_filter_by_lua*
body_filter_by_lua*
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：

```
# 明文协议版本
location /mixed {
    content_by_lua_file ...;       # 请求处理
}

# 加密协议版本
location /mixed {
    access_by_lua_file ...;        # 请求加密解码
    content_by_lua_file ...;       # 请求处理，不需要关心通信协议
    body_filter_by_lua_file ...;   # 应答加密编码
}
```

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

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，如果我们有日志输出的情况，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）：

```
-- lua_package_path "/path/to/lua-resty-logger-socket/lib/?.lua;;";
--
--    server {
--        location / {
--            content_by_lua_file lua/log.lua;
--        }
--    }

-- lua/log.lua
local logger = require "resty.logger.socket"
if not logger.initted() then
    local ok, err = logger.init{
        host = 'xxx',
        port = 1234,
        flush_limit = 1,    --日志长度大于flush_limit的时候会将msg信息推送一次
        drop_limit = 99999,
    }
    if not ok then
        ngx.log(ngx.ERR, "failed to initialize the logger: ",err)
        return
    end
end

local msg = string.format(.....)
local bytes, err = logger.log(msg)
if err then
    ngx.log(ngx.ERR, "failed to log message: ", err)
    return
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

那么我们来看lua-resty-logger-socket这个库的 log 函数是如何实现的呢，代码如下：

```
function _M.log(msg)
   ...

    if (debug) then
        ngx.update_time()
        ngx_log(DEBUG, ngx.now(), ":log message length: " .. #msg)
    end

    local msg_len = #msg

    if (is_exiting()) then
        exiting = true
        _write_buffer(msg)
        _flush_buffer()
        if (debug) then
            ngx_log(DEBUG, "Nginx worker is exiting")
        end
        bytes = 0
    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
        _write_buffer(msg)
        bytes = msg_len
    elseif (msg_len + buffer_size <= drop_limit) then
        _write_buffer(msg)
        _flush_buffer()
        bytes = msg_len
    else
        _flush_buffer()
        if (debug) then
            ngx_log(DEBUG, "logger buffer is full, this log message will be "
                    .. "dropped")
        end
        bytes = 0
        --- this log message doesn't fit in buffer, drop it

        ...
```

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：

```
lua_package_path "/path/to/lua-resty-logger-socket/lib/?.lua;;";

    server {
        location / {
            content_by_lua_file lua/content.lua;
            log_by_lua_file lua/log.lua;
        }
    }
```

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：

```
    location /test {
        rewrite_by_lua_block {
            ngx.say("foo = ", ngx.ctx.foo)
            ngx.ctx.foo = 76
        }
        access_by_lua_block {
            ngx.ctx.foo = ngx.ctx.foo + 3
        }
        content_by_lua_block {
            ngx.say(ngx.ctx.foo)
        }
    }
```

更多有关 ngx.ctx 信息，请看这里。

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。

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

- 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 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：

```
local user_script = [[
    local a = 0
    local rand = math.random
    for i = 1, 200 do
        a = a + rand(i)
    end
    ngx.say("hi")
]]

local function handle_timeout(typ)
    return error("user script too hot")
end

local function handle_error(err)
    return string.format("%s: %s", err or "", debug.traceback())
end

-- disable JIT in the user script to ensure debug hooks always work:
user_script = [[jit.off(true, true) ]] .. user_script

local f, err = loadstring(user_script, "=user script")
if not f then
    ngx.say("ERROR: failed to load user script: ", err)
    return
end

-- only enable math.*, and ngx.say in our sandbox:
local env = {
    math = math,
    ngx = { say = ngx.say },
    jit = { off = jit.off },
}
setfenv(f, env)

local instruction_limit = 1000
debug.sethook(handle_timeout, "", instruction_limit)
local ok, err = xpcall(f, handle_error)
if not ok then
    ngx.say("failed to run user script: ", err)
end
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

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：

```
function get_from_cache(key)
    local cache_ngx = ngx.shared.my_cache
    local value = cache_ngx:get(key)
    return value
end

function set_to_cache(key, value, exptime)
    if not exptime then
        exptime = 0
    end

    local cache_ngx = ngx.shared.my_cache
    local succ, err, forcible = cache_ngx:set(key, value, exptime)
    return succ
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：

```
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：

```
local _M = {}

-- alternatively: local lrucache = require "resty.lrucache.pureffi"
local lrucache = require "resty.lrucache"

-- we need to initialize the cache on the Lua module level so that
-- it can be shared by all the requests served by each nginx worker process:
local c = lrucache.new(200)  -- allow up to 200 items in the cache
if not c then
    return error("failed to create the cache: " .. (err or "unknown"))
end

function _M.go()
    c:set("dog", 32)
    c:set("cat", 56)
    ngx.say("dog: ", c:get("dog"))
    ngx.say("cat: ", c:get("cat"))

    c:set("dog", { age = 10 }, 0.1)  -- expire in 0.1 sec
    c:delete("dog")
end

return _M
```

可以看出来，这个 cache 是 worker 级别的，不会在 Nginx wokers 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。lrucache 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

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。

```
# you do not need the following line if you are using
# the ngx_openresty bundle:
lua_package_path "/path/to/lua-resty-redis/lib/?.lua;;";

server {
    location /non_block {
        content_by_lua_block {
            ngx.sleep(0.1)
        }
    }
}
```

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

```
location /sleep_1 {
    default_type 'text/plain';
    content_by_lua_block {
        ngx.sleep(0.01)
        ngx.say("ok")
    }
}

location /sleep_2 {
    default_type 'text/plain';
    content_by_lua_block {
        function sleep(n)
            os.execute("sleep " .. n)
        end
        sleep(0.01)
        ngx.say("ok")
    }
}
```

ab 测试一下

```
➜  nginx git:(master) ab -c 10 -n 20  http://127.0.0.1/sleep_1
...
Requests per second:    860.33 [#/sec] (mean)
...
➜  nginx git:(master) ab -c 10 -n 20  http://127.0.0.1/sleep_2
...
Requests per second:    56.87 [#/sec] (mean)
...
```

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)，而后者却会触发进程切换，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

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()。 这个函数是在后台用 Nginx 轻线程（light thread），在指定的延时后，调用指定的函数。 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。

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)，原因是这些请求都需要绑定某个请求，但是对于 `ngx.timer.at` 自身的运行，是与当前任何请求都没关系的。

比较典型的用法，如下示例：

```
local delay = 5
local handler
handler = function (premature)
    -- do some routine job in Lua just like a cron job
    if premature then
        return
    end
    local ok, err = ngx.timer.at(delay, handler)
    if not ok then
        ngx.log(ngx.ERR, "failed to create the timer: ", err)
        return
    end
end

local ok, err = ngx.timer.at(delay, handler)
if not ok then
    ngx.log(ngx.ERR, "failed to create the timer: ", err)
    return
end
```

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）

```
location / {
    deny  192.168.1.1;
    allow 192.168.1.0/24;
    allow 10.1.1.0/16;
    allow 2001:0db8::/32;
    deny  all;
}
```

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，2001:0db8::/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）

```
Example:

geo $country {
    default        ZZ;
    proxy          192.168.100.0/24;

    127.0.0.0/24   US;
    127.0.0.1/32   RU;
    10.1.0.0/16    RU;
    192.168.1.0/24 UK;
}

if ($country == ZZ){
    return 403;
}
```

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：

```
init_by_lua_block {
  local iputils = require("resty.iputils")
  iputils.enable_lrucache()
  local whitelist_ips = {
      "127.0.0.1",
      "10.10.10.0/24",
      "192.168.0.0/16",
  }

  -- WARNING: Global variable, recommend this is cached at the module level
  -- https://github.com/openresty/lua-nginx-module#data-sharing-within-an-nginx-worker
  whitelist = iputils.parse_cidrs(whitelist_ips)
}

access_by_lua_block {
    local iputils = require("resty.iputils")
    if not iputils.ip_in_cidrs(ngx.var.remote_addr, whitelist) then
      return ngx.exit(ngx.HTTP_FORBIDDEN)
    end
}
```

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

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：

```
local response, user_stat = logic_func.get_response(request)
ngx.say(response)
ngx.eof()

if user_stat then
   local ret = db_redis.update_user_data(user_stat)
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

```
local response, user_stat = logic_func.get_response(request)
```

运行了 0.1 秒，而

```
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。

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。

```
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 lrucache，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：

```
local response, err = redis_op.finish_client_task(client_mid, task_id)
if response then
    put_job(client_mid, result)
    ngx.log(ngx.WARN, "put job:", common.json_encode({channel="task_status", mid=clien
t_mid, data=result}))
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。

调用其他 C 函数动态库

Linux 下的动态库一般都以 .so 结束命名，而 Windows 下一般都以 .dll 结束命名。Lua 作为一种嵌入式语言，和 C 具有非常好的亲缘性，这也是 Lua 赖以生存、发展的根本，所以 Nginx + Lua=OpenResty，魔法就这么神奇的发生了。

NgxLuaModule 里面尽管提供了十分丰富的 API，但他一定不可能满足我们的形形色色的需求。我们总是要和各种组件、算法等形形色色的第三方库进行协作。那么如何在 Lua 中加载动态加载第三方库，就显得非常有用。

扯一些额外话题，Lua 解释器目前有两个最主流分支。

- Lua 官方发布的标准版Lua
- Google 开发维护的LuaJIT

LuaJIT 中加入了 Just In Time 等编译技术，使得 Lua 的解释、执行效率有非常大的提升。除此以外，还提供了FFI。

> 什么是FFI？

```
The FFI library allows calling external C functions and using C data
structures from pure Lua code.
```

通过 FFI 的方式加载其他 C 接口动态库，这样我们就可以有很多有意思的玩法。

当我们碰到 CPU 密集运算部分，我们可以把他用 C 的方式实现一个效率最高的版本，对外导出 API，打包成动态库，通过 FFI 来完成 API 调用。这样我们就可以兼顾程序灵活、执行高效，大大弥补了 LuaJIT 自身的不足。

> 使用FFI判断操作系统

```lua
local ffi = require("ffi")
if ffi.os == "Windows" then
    print("windows")
elseif ffi.os == "OSX" then
    print("MAC OS X")
else
    print(ffi.os)
end
```

> 调用zlib压缩库

```
local ffi = require("ffi")
ffi.cdef[[
unsigned long compressBound(unsigned long sourceLen);
int compress2(uint8_t *dest, unsigned long *destLen,
              const uint8_t *source, unsigned long sourceLen, int level);
int uncompress(uint8_t *dest, unsigned long *destLen,
               const uint8_t *source, unsigned long sourceLen);
]]
local zlib = ffi.load(ffi.os == "Windows" and "zlib1" or "z")

local function compress(txt)
  local n = zlib.compressBound(#txt)
  local buf = ffi.new("uint8_t[?]", n)
  local buflen = ffi.new("unsigned long[1]", n)
  local res = zlib.compress2(buf, buflen, txt, #txt, 9)
  assert(res == 0)
  return ffi.string(buf, buflen[0])
end

local function uncompress(comp, n)
  local buf = ffi.new("uint8_t[?]", n)
  local buflen = ffi.new("unsigned long[1]", n)
  local res = zlib.uncompress(buf, buflen, comp, #comp)
  assert(res == 0)
  return ffi.string(buf, buflen[0])
end

-- Simple test code.
local txt = string.rep("abcd", 1000)
print("Uncompressed size: ", #txt)
local c = compress(txt)
print("Compressed size: ", #c)
local txt2 = uncompress(c, #txt)
assert(txt2 == txt)
```

> 自定义定义C类型的方法

```
local ffi = require("ffi")
ffi.cdef[[
typedef struct { double x, y; } point_t;
]]

local point
local mt = {
  __add = function(a, b) return point(a.x+b.x, a.y+b.y) end,
  __len = function(a) return math.sqrt(a.x*a.x + a.y*a.y) end,
  __index = {
    area = function(a) return a.x*a.x + a.y*a.y end,
  },
}
point = ffi.metatype("point_t", mt)

local a = point(3, 4)
print(a.x, a.y)  --> 3  4
print(#a)        --> 5
print(a:area())  --> 25
local b = a + point(0.5, 8)
print(#b)        --> 12.5
```

> Lua和LuaJIT对比

可以这么说，LuaJIT 应该是全面胜出，无论是功能、效率都是标准 Lua 不能比的。目前最新版 OpenResty 默认也都使用 LuaJIT。

世界为我所用，总是有惊喜等着你，如果哪天你发现自己站在了顶峰，那我们就静下心来改善一下顶峰，把他推到更高吧。

请求中断后的处理

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

3.0 避免使用table.insert()

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。
使用table.insert

```
local a = {}
local table_insert = table.insert
for i = 1,100 do
   table_insert( a, i )
end
```

使用循环的计数

```
local a = {}
for i = 1,100 do
   a[i] = i
end
```

使用table的size

```
local a = {}
for i = 1,100 do
   a[#a+1] = i
end
```

使用计数器

```
local a = {}
local index = 1
for i = 1,100 do
   a[index] = i
   index = index+1
end
```

4.0 减少使用 unpack()函数
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。

使用unpack()

```
local a = { 100, 200, 300, 400 }
for i = 1,100 do
   print( unpack(a) )
end
```

代替方法

```
local a = { 100, 200, 300, 400 }
for i = 1,100 do
   print( a[1],a[2],a[3],a[4] )
end
```

针对这篇文章内容写了一些测试代码：

```
local start = os.clock()
```

```
local function sum( ... )
    local args = {...}
    local a = 0
    for k,v in pairs(args) do
        a = a + v
    end
    return a
end

local function test_unit(  )
    -- t1: 0.340182 s
    -- local a = {}
    -- for i = 1,1000 do
    --    table.insert( a, i )
    -- end

    -- t2: 0.332668 s
    -- local a = {}
    -- for i = 1,1000 do
    --    a[#a+1] = i
    -- end

    -- t3: 0.054166 s
    -- local a = {}
    -- local index = 1
    -- for i = 1,1000 do
    --    a[index] = i
    --    index = index+1
    -- end

    -- p1: 0.708012 s
    -- local a = 0
    -- for i=1,1000 do
    --     local t = { 1, 2, 3, 4 }
    --     for i,v in ipairs( t ) do
    --        a = a + v
    --     end
    -- end

    -- p2: 0.660426 s
    -- local a = 0
    -- for i=1,1000 do
    --     local t = { 1, 2, 3, 4 }
    --     for i = 1,#t do
    --        a = a + t[i]
    --     end
    -- end

    -- u1: 2.121722 s
    -- local a = { 100, 200, 300, 400 }
    -- local b = 1
    -- for i = 1,1000 do
    --    b = sum(unpack(a))
```

```
    -- end

    -- u2: 1.701365 s
    -- local a = { 100, 200, 300, 400 }
    -- local b = 1
    -- for i = 1,1000 do
    --    b = sum(a[1], a[2], a[3], a[4])
    -- end

    return b
end

for i=1,10 do
    for j=1,1000 do
        test_unit()
    end
end

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。

todo：优化测试用例，让他更直观，自己先备注一下。

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：

```
-- index.lua
local uri_args = ngx.req.get_uri_args()
local mo = require('mo')
mo.args = uri_args
```

```
-- mo.lua

local showJs = function(callback, data)
    local cjson = require('cjson')
    ngx.say(callback .. '(' .. cjson.encode(data) .. ')')
end
local self.jsonp = self.args.jsonp
local keyList = string.split(self.args.key_list, ',')
for i=1, #keyList do
    -- do something
    ngx.say(self.args.kind)
end
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：

```
    location /test {
        rewrite_by_lua_block {
            ngx.ctx.foo = 76
        }
        access_by_lua_block {
            ngx.ctx.foo = ngx.ctx.foo + 3
        }
        content_by_lua_block {
            ngx.say(ngx.ctx.foo)
        }
    }
```

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

```
location = /index {
    content_by_lua_file conf/lua/web/index.lua;
}
```

index.lua

```
local var = require "var"


if var.calc() == 1 then
    ngx.say("ok")
else
    ngx.status = ngx.HTTP_INTERNAL_SERVER_ERROR
    ngx.say("error")
end
```

var.lua

```
local count = 1

local _M = {}

local function add()
    count = count + 1
end

local function sub()
    count = count - 1
end

function _M.calc()
    add()
    -- 模拟协程调度
    ngx.sleep(ngx.time()%0.003)
    sub()
    return count
end

return _M
```

分别用单个客户端和两个客户端请求之:

```
~/test/openresty wrk --timeout 10s -t 1 -c 1 -d 1s http://localhost:6699/index
Running 1s test @ http://localhost:6699
  1 threads and 1 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.22ms  291.09us   3.48ms   77.30%
    Req/Sec   822.64    175.27     1.01k    54.55%
  900 requests in 1.10s, 153.76KB read
~/test/openresty wrk --timeout 10s -t 2 -c 2 -d 1s http://localhost:6699/index
Running 1s test @ http://localhost:6699
  2 threads and 2 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.18ms  387.03us   7.92ms   85.98%
    Req/Sec     0.86k   168.12     1.02k    60.00%
  1709 requests in 1.00s, 310.29KB read
  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:

```
coroutine A | coroutine B | count
add         |             | 2
sleep       |             | 2
            | add         | 3
            | sleep       | 3
sub         |             | 2

(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。

```
init_worker_by_lua_block {
    local delay = 5
    local handler
    handler = function()
        counter = counter or 0
        counter = counter + 1
        ngx.log(ngx.ERR, counter)
        local ok, err = ngx.timer.at(delay, handler)
        if not ok then
            ngx.log(ngx.ERR, "failed to create the timer: ", err)
            return
        end
    end
    local ok, err = ngx.timer.at(delay, handler)
    if not ok then
        ngx.log(ngx.ERR, "failed to create the timer: ", err)
        return
    end
}
```

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。

（当然也可以选择在 `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

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` 指令的设值。 一如既往，通过 OpenResty，我们可以直接在 Lua 代码中动态设置它。

```
access_by_lua_block {
    -- 设定当前请求的响应上限是 每秒 300K 字节
    ngx.var.limit_rate = "300K"
}
```

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（表现为池中出现蓄水），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：

```
# nginx.conf
lua_code_cache on;
# 注意 limit_conn_store 的大小需要足够放置限流所需的键值。
# 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
。
# 100M 可以放 1.6M 个键值对
lua_shared_dict limit_conn_store 100M;
server {
    listen 8080;
    location / {
        access_by_lua_file src/access.lua;
        content_by_lua_file src/content.lua;
        log_by_lua_file src/log.lua;
    }
}
```

```lua
-- utils/limit_conn.lua
local limit_conn = require "resty.limit.conn"

-- 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
local limit, limit_err = limit_conn.new("limit_conn_store", 10, 2, 0.05)
if not limit then
    error("failed to instantiate a resty.limit.conn object: ", limit_err)
end

local _M = {}

function _M.incoming()
    local key = ngx.var.binary_remote_addr
    local delay, err = limit:incoming(key, true)
    if not delay then
        if err == "rejected" then
            return ngx.exit(503)
        end
        ngx.log(ngx.ERR, "failed to limit req: ", err)
        return ngx.exit(500)
    end

    if limit:is_committed() then
        local ctx = ngx.ctx
        ctx.limit_conn_key = key
        ctx.limit_conn_delay = delay
    end

    if delay >= 0.001 then
        ngx.log(ngx.WARN, "delaying conn, excess ", delay,
                "s per binary_remote_addr by limit_conn_store")
        ngx.sleep(delay)
    end
end

function _M.leaving()
    local ctx = ngx.ctx
    local key = ctx.limit_conn_key
    if key then
        local latency = tonumber(ngx.var.request_time) - ctx.limit_conn_delay
        local conn, err = limit:leaving(key, latency)
        if not conn then
            ngx.log(ngx.ERR,
            "failed to record the connection leaving ",
            "request: ", err)
        end
    end
end

return _M
```

```
-- src/access.lua
local limit_conn = require "utils.limit_conn"


-- 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。
if ngx.req.is_internal() then
    return
end
limit_conn.incoming()
```

```
-- src/log.lua
local limit_conn = require "utils.limit_conn"


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。

`ngx.shared.DICT` 非队列性质

==========

执行阶段和主要函数请参考 `HttpLuaModule#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:

```
-- [ngx.thread.spawn](https://github.com/openresty/lua-nginx-module#ngxthreadspawn) #1
 存储线程 理解为生产者

    ....
    local cache_str = string.format([[%s&%s&%s&%s&%s&%s]], net, name, ip,
                    mac, ngx.var.remote_addr, method, md5)
    local ok, err = ngx_nf_data:safe_set(mac, cache_str, 60*60)  --这些是缓存数据
    if not ok then
        ngx.log(ngx.ERR, "stored nf report data error: "..err)
    end
    ....


-- [ngx.thread.spawn](https://github.com/openresty/lua-nginx-module#ngxthreadspawn) #2
 取线程 理解为消费者

    while not ngx.worker.exiting() do
        local keys = ngx_share:get_keys(50)  -- 一秒处理50个数据

        for index, key in pairs(keys) do
            str = ((nil ~= str) and str..[[#]]..ngx_share:get(key)) or ngx_share:get(k
ey)
            ngx_share:delete(key)  --干掉这个key
        end
        .... --一些消费过程，看官不要在意
        ngx.sleep(1)
    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

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。

```
-- 错误的代码
local function send()
    for i = 1, count do
        local ssdb_db, err = ssdb:new()
        local ok, err = ssdb_db:connect(SSDB_HOST, SSDB_PORT)
        if not ok then
            ngx.log(ngx.ERR, "create new ssdb failed!")
        else
               local key,err = ssdb_db:qpop(something)
             if not key then
                   ngx.log(ngx.ERR, "ssdb qpop err:", err)
               else
                   local data, err = ssdb_db:get(key[1])
                   -- other operations
               end
        end
    end
    ssdb_db:set_keepalive(SSDB_KEEP_TIMEOUT, SSDB_KEEP_COUNT)
end
-- 调用
while true do
    local ths = {}
    for i=1,THREADS do
        ths[i] = ngx.thread.spawn(send)        ----创建线程
    end
    for i = 1, #ths do
        ngx.thread.wait(ths[i])                ----等待线程执行
    end
    ngx.sleep(0.020)
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。

```
local function send()
    for i = 1, count do
        local ssdb_db, err = ssdb:new()
        local ok, err = ssdb_db:connect(SSDB_HOST, SSDB_PORT)
        if not ok then
            ngx.log(ngx.ERR, "create new ssdb failed!")
            return
        else
               local key,err = ssdb_db:qpop(something)
             if not key then
                   ngx.log(ngx.ERR, "ssdb qpop err:", err)
               else
                   local data, err = ssdb_db:get(key[1])
                   -- other operations
               end
               ssdb_db:set_keepalive(SSDB_KEEP_TIMEOUT, SSDB_KEEP_COUNT)
        end
    end
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

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` (pintsized/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：

```
server {

    listen       8080 default_server;
    server_name  _;

    resolver 8.8.8.8;

    location /baidu {
        content_by_lua_block {
            local http = require "resty.http"
            local httpc = http.new()
            local res, err = httpc:request_uri("http://www.baidu.com")
            if res.status == ngx.HTTP_OK then
                ngx.say(res.body)
            else
                ngx.exit(res.status)
            end
        }
    }
}
```

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）。

body 在 location 中的传递

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

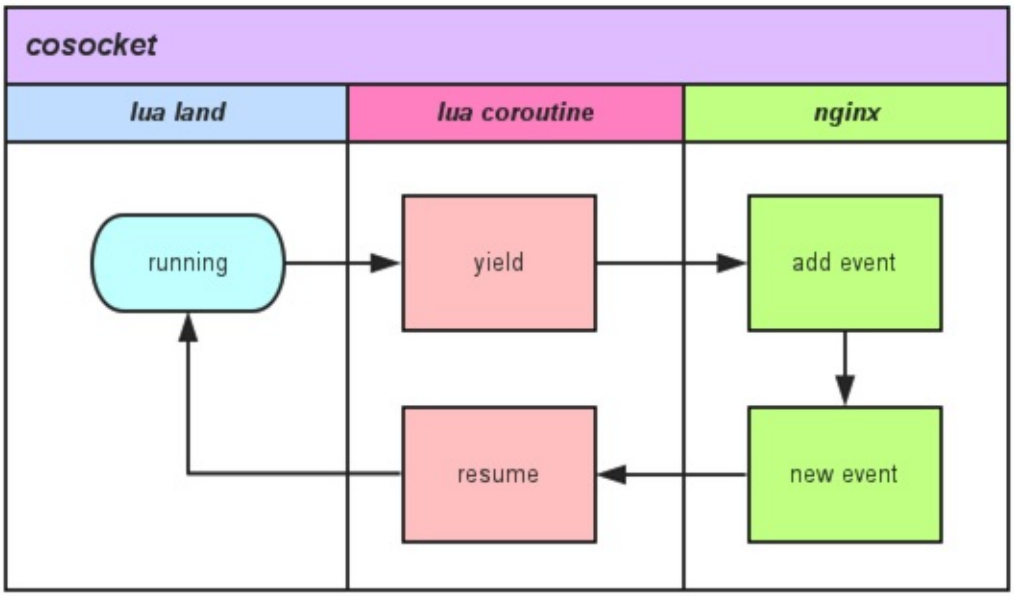
笔者认为，cosocket 是 OpenResty 世界中技术、实用价值最高部分。让我们可以用非常低廉的成本，优雅的姿势，比传统 socket 编程效率高好几倍的方式进行网络编程。无论资源占用、执行效率、并发能力都非常出色。

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。

以此为蓝板，封装实现 connect、read、recieve 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。

```
location /test {
    resolver 114.114.114.114;

    content_by_lua_block {
        local sock = ngx.socket.tcp()
        local ok, err = sock:connect("www.baidu.com", 80)
        if not ok then
            ngx.say("failed to connect to baidu: ", err)
            return
        end

        local req_data = "GET / HTTP/1.1\r\nHost: www.baidu.com\r\n\r\n"
        local bytes, err = sock:send(req_data)
        if err then
            ngx.say("failed to send to baidu: ", err)
            return
        end

        local data, err, partial = sock:receive()
        if err then
            ngx.say("failed to recieve to baidu: ", err)
            return
        end

        sock:close()
        ngx.say("successfully talk to baidu! response first line: ", data)
    }
}
```

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()。

```
语法: seq_id = ngx.worker.id()

返回当前 Nginx 工作进程的一个顺序数字（从 0 开始）。

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。

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。

```
init_worker_by_lua_block {
     local delay = 3  -- in seconds
     local new_timer = ngx.timer.at
     local log = ngx.log
     local ERR = ngx.ERR
     local check

     check = function(premature)
         if not premature then
             -- do the health check or other routine work
             local ok, err = new_timer(delay, check)
             if not ok then
                 log(ERR, "failed to create timer: ", err)
                 return
             end
         end
     end

     if 0 == ngx.worker.id() then
         local ok, err = new_timer(delay, check)
         if not ok then
             log(ERR, "failed to create timer: ", err)
             return
         end
     end
}
```

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

```
local client = mysql:new()
client:connect({
    host = "rdsmxxxxxx.mysql.rds.xxxx.com",
    port = 3306,
    database = "test",
    user = "test",
    password = "123456"
})
```

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

```
Syntax:    resolver address ... [valid=time] [ipv6=on|off];
Default:     —
Context:    http, server, location
```

一个简单的例子

```
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。

```
local pcall = pcall
local io_open = io.open
local ngx_re_gmatch = ngx.re.gmatch

local ok, new_tab = pcall(require, "table.new")

if not ok then
    new_tab = function (narr, nrec) return {} end
end

local _dns_servers = new_tab(5, 0)

local _read_file_data = function(path)
    local f, err = io_open(path, 'r')

    if not f or err then
        return nil, err
    end

    local data = f:read('*all')
    f:close()
    return data, nil
end

local _read_dns_servers_from_resolv_file = function()
    local text = _read_file_data('/etc/resolv.conf')

    local captures, it, err
    it, err = ngx_re_gmatch(text, [[^nameserver\s+(\d+?\.\d+?\.\d+?\.\d+$)]], "jomi")

    for captures, err in it do
        if not err then
            _dns_servers[#_dns_servers + 1] = captures[1]
        end
    end
end

_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

```
local require = require
local ngx_re_find = ngx.re.find
local lrucache = require "resty.lrucache"
local resolver = require "resty.dns.resolver"
local cache_storage = lrucache.new(200)

local _is_addr = function(hostname)
    return ngx_re_find(hostname, [[\d+?\.\d+?\.\d+?\.\d+$]], "jo")
end

local _get_addr = function(hostname)
    if _is_addr(hostname) then
        return hostname, hostname
    end

    local addr = cache_storage:get(hostname)

    if addr then
        return addr, hostname
    end

    local r, err = resolver:new({
        nameservers = _dns_servers,
        retrans = 5,  -- 5 retransmissions on receive timeout
        timeout = 2000,  -- 2 sec
    })

    if not r then
        return nil, hostname
    end

    local answers, err = r:query(hostname, {qtype = r.TYPE_A})

    if not answers or answers.errcode then
        return nil, hostname
    end

    for i, ans in ipairs(answers) do
        if ans.address then
            cache_storage:set(hostname, ans.address, 300)
            return ans.address, hostname
        end
    end

    return nil, hostname
end

ngx.say(_get_addr("www.baidu.com"))
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

```
local client = mysql:new()
client:connect({
    host = _get_addr(conf.mysql_hostname),
    port = 3306,
    database = "test",
    user = "test",
    password = "123456"
})
```

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 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：

```
lua_package_path "/path/to/lua-resty-dns/lib/?.lua;;";

 server {
     location = /dns {
         content_by_lua_block {
             local resolver = require "resty.dns.resolver"
             local r, err = resolver:new{
                 nameservers = {"8.8.8.8", {"8.8.4.4", 53} },
                 retrans = 5,  -- 5 retransmissions on receive timeout
                 timeout = 2000,  -- 2 sec
             }

             if not r then
                 ngx.say("failed to instantiate the resolver: ", err)
                 return
             end

             local answers, err = r:query("www.google.com")
             if not answers then
                 ngx.say("failed to query the DNS server: ", err)
                 return
             end

             if answers.errcode then
                 ngx.say("server returned error code: ", answers.errcode,
                         ": ", answers.errstr)
             end

             for i, ans in ipairs(answers) do
                 ngx.say(ans.name, " ", ans.address or ans.cname,
                         " type:", ans.type, " class:", ans.class,
                         " ttl:", ans.ttl)
             end
         }
     }
 }
```

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。

```
local resolver = require "resty.dns.resolver"
local http     = require "resty.http"

function get_domain_ip_by_dns( domain )
  -- 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）
  local dns = "8.8.8.8"

  local r, err = resolver:new{
      nameservers = {dns, {dns, 53} },
      retrans = 5,  -- 5 retransmissions on receive timeout
      timeout = 2000,  -- 2 sec
  }

  if not r then
      return nil, "failed to instantiate the resolver: " .. err
  end

  local answers, err = r:query(domain)
  if not answers then
      return nil, "failed to query the DNS server: " .. err
  end

  if answers.errcode then
      return nil, "server returned error code: " .. answers.errcode .. ": " .. answers
.errstr
  end

  for i, ans in ipairs(answers) do
```

```
    if ans.address then
      return ans.address
    end
  end

  return nil, "not founded"
end

function http_request_with_dns( url, param )
    -- get domain
    local domain = ngx.re.match(url, [[//([\S]+?)/]])
    domain = (domain and 1 == #domain and domain[1]) or nil
    if not domain then
        ngx.log(ngx.ERR, "get the domain fail from url:", url)
        return {status=ngx.HTTP_BAD_REQUEST}
    end

    -- add param
    if not param.headers then
        param.headers = {}
    end
    param.headers.Host = domain

    -- get domain's ip
    local domain_ip, err = get_domain_ip_by_dns(domain)
    if not domain_ip then
        ngx.log(ngx.ERR, "get the domain[", domain ,"] ip by dns failed:", err)
        return {status=ngx.HTTP_SERVICE_UNAVAILABLE}
    end

    -- http request
    local httpc = http.new()
    local temp_url = ngx.re.gsub(url, "//"..domain.."/", string.format("//%s/", domain
_ip))

    local res, err = httpc:request_uri(temp_url, param)
    if err then
        return {status=ngx.HTTP_SERVICE_UNAVAILABLE}
    end

    -- httpc:request_uri 内部已经调用了keepalive，默认支持长连接
    -- httpc:set_keepalive(1000, 100)
    return res
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。

lock

缓存失效风暴

看下这个段伪代码：

```
local value = get_from_cache(key)
if not value then
    value = query_db(sql)
    set_to_cache(value， timeout = 100)
end
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

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

```
    #you do not need the following line if you are using
    #the ngx_openresty bundle:

    lua_package_path "/path/to/lua-resty-redis/lib/?.lua;;";

    server {
        location /test {
            content_by_lua_file test_case_lua/unit/test_example.lua;
        }
    }
```

> test_case_lua/unit/test_example.lua:

```
local iresty_test    = require "resty.iresty_test"
local tb = iresty_test.new({unit_name="example"})

function tb:init(  )
    self:log("init complete")
end

function tb:test_00001(  )
    error("invalid input")
end

function tb:atest_00002()
    self:log("never be called")
end

function tb:test_00003(  )
   self:log("ok")
end

-- units test
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：

```
TIME   Name            Log
0.000  [example] unit test start
0.000  [example] init complete
0.000    \_[test_00001] fail ...de/nginx/main_server/test_case_lua/unit/test_example.l
ua:9: invalid input
0.000    \_[test_00003] ↓ ok
0.000    \_[test_00003] PASS
0.000  [example] unit test complete
```

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：

```
luarocks install luacov
```

当然，你也可以通过 GitHub 上的源码 编译来安装。这个方式你可以修改 LuaCov 的一些默认配置。

比如 LuaCov 的分析文件是按照 100 条一批来写入的，如果你的代码量不大，可能就会不准确。你可以修改 /src/luacov/defaults.Lua 里面的 savestepsize，改为 2 ，来适应你的应用场景。

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。

```
init_by_lua_block {
    require 'luacov.tick'
    jit.off()
}
```

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，请使用最新的来版本来测试。

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：

```
luacov
```

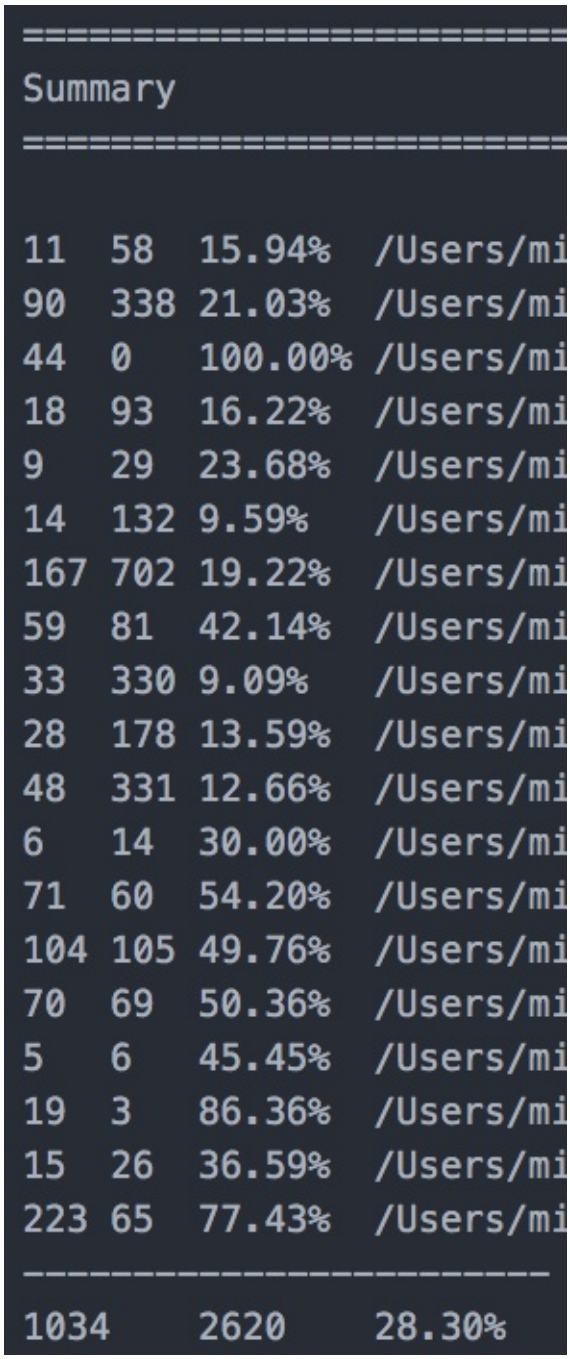
就会生成 `luacov.report.out` 这个可读性比较好的覆盖率报告。需要注意的是，luacov 这个命令后面不用加任何的参数，这个在官方文档里面有说明，只是比较隐晦。

我们看下 `luacov.report.out` 里面的一个片段：

```
1    function get_config(mid, args)
13       local configs = {}
13       local res, err = red:hmget("client_".. mid, "tpl_id", "gid")
13       if err then
****0      return nil, err
         end
       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：



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（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:

```
location ~* /api/([\w_]+?)\.json {
    content_by_lua_file lua/$1.lua;
}

location ~* /unit_test/([\w_]+?)\.json {
    lua_check_client_abort on;
    content_by_lua_file test_case_lua/unit/$1.lua;
}
```

> API测试代码：

```
-- unit test for /api/get_client_tasks.json
local tb = require "resty.iresty_test"
local json   = require("cjson")
local test = tb.new({unit_name="get_client_tasks"})

function tb:init(  )
    self.mid = string.rep('0',32)
end

function tb:test_0000()
    -- 正常请求
    local res = ngx.location.capture(
        '/api/get_client_tasks.json?mid='..self.mid,
        { method = ngx.HTTP_POST, body=[[{"type":[1600,1700]}]] }
    )
```

```
    if 200 ~= res.status then
        error("failed code:" .. res.status)
    end
end

function tb:test_0001()
    -- 缺少body
    local res = ngx.location.capture(
        '/api/get_client_tasks.json?mid='..self.mid,
        { method = ngx.HTTP_POST }
    )

    if 400 ~= res.status then
        error("failed code:" .. res.status)
    end
end

function tb:test_0002()
    -- 错误的json内容
    local res = ngx.location.capture(
        '/api/get_client_tasks.json?mid='..self.mid,
        { method = ngx.HTTP_POST, body=[[{"type":"[1600,1700]}]] }
    )

    if 400 ~= res.status then
        error("failed code:" .. res.status)
    end
end

function tb:test_0003()
    -- 错误的json格式
    local res = ngx.location.capture(
        '/api/get_client_tasks.json?mid='..self.mid,
        { method = ngx.HTTP_POST, body=[[{"type":"[1600,1700]"}]] }
    )

    if 400 ~= res.status then
        error("failed code:" .. res.status)
    end
end

test:run()
```

Nginx output:

```
0.000  [get_client_tasks] unit test start
0.001    \_[test_0000] PASS
0.001    \_[test_0001] PASS
0.001    \_[test_0002] PASS
0.001    \_[test_0003] PASS
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力测试

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。

```
#ab -n10 -c2 http://haosou.com/

-- output:
...
Complete requests:      10
Failed requests:        0
Non-2xx responses:      10
Total transferred:      3620 bytes
HTML transferred:       1780 bytes
Requests per second:    22.00 [#/sec] (mean)
Time per request:       90.923 [ms] (mean)
Time per request:       45.461 [ms] (mean, across all concurrent requests)
Transfer rate:          7.78 [Kbytes/sec] received
...
```

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

```
-- example dynamic request script which demonstrates changing
-- the request path and a header for each request
-------------------------------------------------------------
-- NOTE: each wrk thread has an independent Lua scripting
-- context and thus there will be one counter per thread

counter = 0

request = function()
   path = "/" .. counter
   wrk.headers["X-Counter"] = counter
   counter = counter + 1
   return wrk.format(nil, path)
end
```

> shell执行

```
# ./wrk -c10 -d1 -s scripts/counter.lua http://baidu.com
Running 1s test @ http://baidu.com
  2 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    20.44ms    3.74ms  34.87ms   77.48%
    Req/Sec   226.05     42.13   270.00     70.00%
  453 requests in 1.01s, 200.17KB read
  Socket errors: connect 0, read 9, write 0, timeout 0
Requests/sec:    449.85
Transfer/sec:    198.78KB
```

> WireShark抓包印证一下

```
GET /228 HTTP/1.1
Host: baidu.com
X-Counter: 228

...(应答包 省略)

GET /232 HTTP/1.1
Host: baidu.com
X-Counter: 232

...(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：

```
wrk contains code from a number of open source projects including the 'ae'
 event loop from redis, the nginx/joyent/node.js 'http-parser', and Mike
 Pall's LuaJIT.
```

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

我们做的还不够好，先占个坑。

欢迎贡献章节。

Web 服务

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：

```
server {
    listen       80;
    server_name  localhost;

    location /app/set {
        content_by_lua_block {
             ngx.say('set data')
        }
    }

    location /app/get {
        content_by_lua_block {
            ngx.say('get data')
        }
    }

    location /app/del {
        content_by_lua_block {
            ngx.say('del data')
        }
    }
}
```

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，

设计良好的网络应用表现为一系列的网页，这些网页可以看作的虚拟的状态机，用户选择这些链接
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

- 客户端和服务器结构
- 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。

```
server {
    listen       80;
    server_name  localhost;

    location /app/task01 {
        content_by_lua_block {
            ngx.say(ngx.req.get_method() .. ' task01')
        }
    }
    location /app/task02 {
        content_by_lua_block {
            ngx.say(ngx.req.get_method() .. ' task02')
        }
    }
    location /app/task03 {
        content_by_lua_block {
            ngx.say(ngx.req.get_method() .. ' task03')
        }
    }
}
```

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。

```
# curl -X GET http://127.0.0.1/app/task01
# curl -X PUT http://127.0.0.1/app/task01
# 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。

```
# use nginx var in code path
# WARNING: contents in nginx var must be carefully filtered,
# otherwise there'll be great security risk!
location ~ ^/app/([-_a-zA-Z0-9/]+) {
    set $path $1;
    content_by_lua_file /path/to/lua/app/root/$path.lua;
}
```

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

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。

```
require "resty.core.regex"


-- 纯 lua 版本，优点是兼容性好，可以适用任何 lua 语言环境
function check_hex_lua( str )
    if "string" ~= type(str) then
        return false
    end

    for i = 1, #str do
        local ord = str:byte(i)
        if not (
            (48 <= ord and ord <= 57) or
            (65 <= ord and ord <= 70) or
            (97 <= ord and ord <= 102)
        ) then
            return false
        end
    end
    return true
end
```

```
-- 使用 ngx.re.* 完成，没有使用任何调优参数
function check_hex_default( str )
    if "string" ~= type(str) then
        return false
    end

    return ngx.re.find(str, "[^0-9a-fA-F]") == nil
end

-- 使用 ngx.re.* 完成，使用调优参数 "jo"
function check_hex_jo( str )
    if "string" ~= type(str) then
        return false
    end

    return ngx.re.find(str, "[^0-9a-fA-F]", "jo") == nil
end


-- 下面就是测试用例部分代码
function do_test( name, fun )
    ngx.update_time()
    local start = ngx.now()

    local t = "0123456789012345678901234567890abcdefABCDEF"
    assert(fun(t))
    for i=1,10000*300 do
        fun(t)
    end

    ngx.update_time()
    print(name, "\ttimes:", ngx.now() - start)
end

do_test("check_hex_lua", check_hex_lua)
do_test("check_hex_default", check_hex_default)
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：

```
➜  resty test.lua
check_hex_lua    times:1.0390000343323
check_hex_default    times:5.1929998397827
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：

```
{
    "username":"myname",
    "age":8,
    "tel":88888888,
    "mobile_no":13888888888,
    "email":"***@**.com",
    "love_things":["football", "music"]
}
```

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：

```
{
    "username":"",
    "age":0,
    "tel":0,
    "mobile_no":0,
    "email":"",
    "love_things":[]
}
```

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：

```
function check_args_template(args, template)
    if type(args) ~= type(template) then
      return false
    elseif "table" ~= type(args) then
      return true
    end

    for k,v in pairs(template) do
      if type(v) ~= type(args[k]) then
        return false
      elseif "table" == type(v) then
        if not check_args_template(args[k], v) then
          return false
        end
      end
    end

    return true
end

local args = {name="myname", tel=888888, age=18,
    mobile_no=13888888888, love_things = {"football", "music"}}

print("valid   check: ", check_args_template(args, {name="", tel=0, love_things={}}))
print("unvalid check: ", check_args_template(args, {name="", tel=0, love_things={}, em
ail=""}))
```

运行一下上面的代码，结果如下：

```
➜  resty test.lua
valid   check: true
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

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

```
location ~ ^/api/([-_a-zA-Z0-9/]+).json {
    content_by_lua_file /path/to/lua/api/$1.lua;
}
```

> 1.0明文协议引用示例：

```
# curl http://ip:port/api/hearbeat.json?key=value -d '...'
```

> 2.0密文协议引用示例：

```
# curl http://ip:port/api/hearbeat.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。

```
location ~ ^/api/([-_a-zA-Z0-9/]+).json {
    access_by_lua_file  /path/to/lua/api/protocal_decode.lua;
    content_by_lua_file /path/to/lua/api/$1.lua;
    body_filter_by_lua_file  /path/to/lua/api/protocal_encode.lua;
}
```

> 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，而来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

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

```
local ffi = require "ffi"
local ffi_new = ffi.new
local ffi_str = ffi.string
local C = ffi.C
local setmetatable = setmetatable
local error = error
local tonumber = tonumber


local _M = { _VERSION = '0.09' }


ffi.cdef[[
typedef unsigned char u_char;

u_char * ngx_hex_dump(u_char *dst, const u_char *src, size_t len);

intptr_t ngx_atoi(const unsigned char *line, size_t n);
]]

local str_type = ffi.typeof("uint8_t[?]")


function _M.to_hex(s)
    local len = #s * 2
    local buf = ffi_new(str_type, len)
    C.ngx_hex_dump(buf, s, #s)
    return ffi_str(buf, len)
end


function _M.atoi(s)
    return tonumber(C.ngx_atoi(s, #s))
end


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，那么就很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很不爽的，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。

```
local
function test()
    -- do something
end

function test2()
    -- do something
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

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问题。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。

```
server {
    location /test {
        content_by_lua_block {
            local redis = require "resty.redis"
            local red = redis:new()

            local ok, err = red:connect("127.0.0.1", 6379)
            if not ok then
                ngx.say("failed to connect: ", err)
                return
            end

            -- red:set_keepalive(10000, 100)       -- 坑①

            ok, err = red:set("dog", "an animal")
            if not ok then
                -- red:set_keepalive(10000, 100)   -- 坑②
                return
            end

            -- 坑③
            red:set_keepalive(10000, 100)
        }
    }
}
```

- 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

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？

告诉你肯定是有的，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。

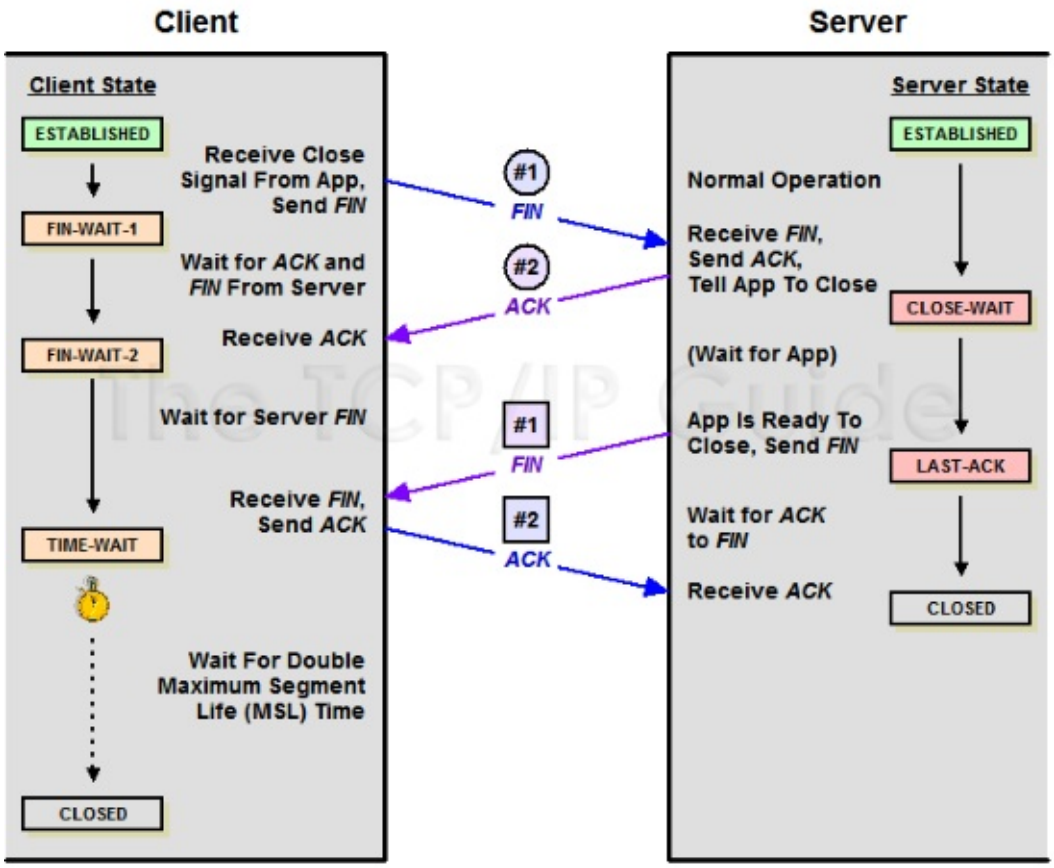
C10M 这个动作比较大，而且还需要特定的硬件型号支持（主要是网卡，网络处理嘛），所以目前这个项目进展还比较缓慢。不过对于有追求的人，可能就要两眼放光了。

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

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猿，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：

```
GET /hello HTTP/1.0
User-Agent: curl/7.37.1
Host: 127.0.0.1
Accept: */*
Accept-Encoding: deflate, gzip
```

应答包：

```
HTTP/1.1 200 OK
Date: Wed, 08 Jul 2015 02:53:54 GMT
Content-Type: text/plain
Connection: close

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：

```
POST /api/heartbeat.json HTTP/1.1

Content-Type: application/x-www-form-urlencoded
Cache-Control: no-cache
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT)
Accept-Encoding: gzip, deflate
Accept: */*
Connection: Keep-Alive
Content-Length: 0
```

应答包：

```
HTTP/1.1 200 OK
Date: Mon, 06 Jul 2015 09:35:34 GMT
Content-Type: text/plain
Transfer-Encoding: chunked
Connection: close
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，而我们也没法控制终端的 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。

```
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，感谢大牛的分享。

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不一致性。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：

```
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。

```
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

```
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。
--dns=[]         : Set custom dns servers for the container
--net="bridge"   : Set the Network mode for the container
                         'bridge': creates a new network stack for the container on t
he docker bridge
                         'none': no networking for this container
                         'container:<name|id>': reuses another container network stac
k
                         'host': use the host network stack inside the container
--add-host=""    : Add a line to /etc/hosts (host:IP)
--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 12:34:56:78:9a: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:.

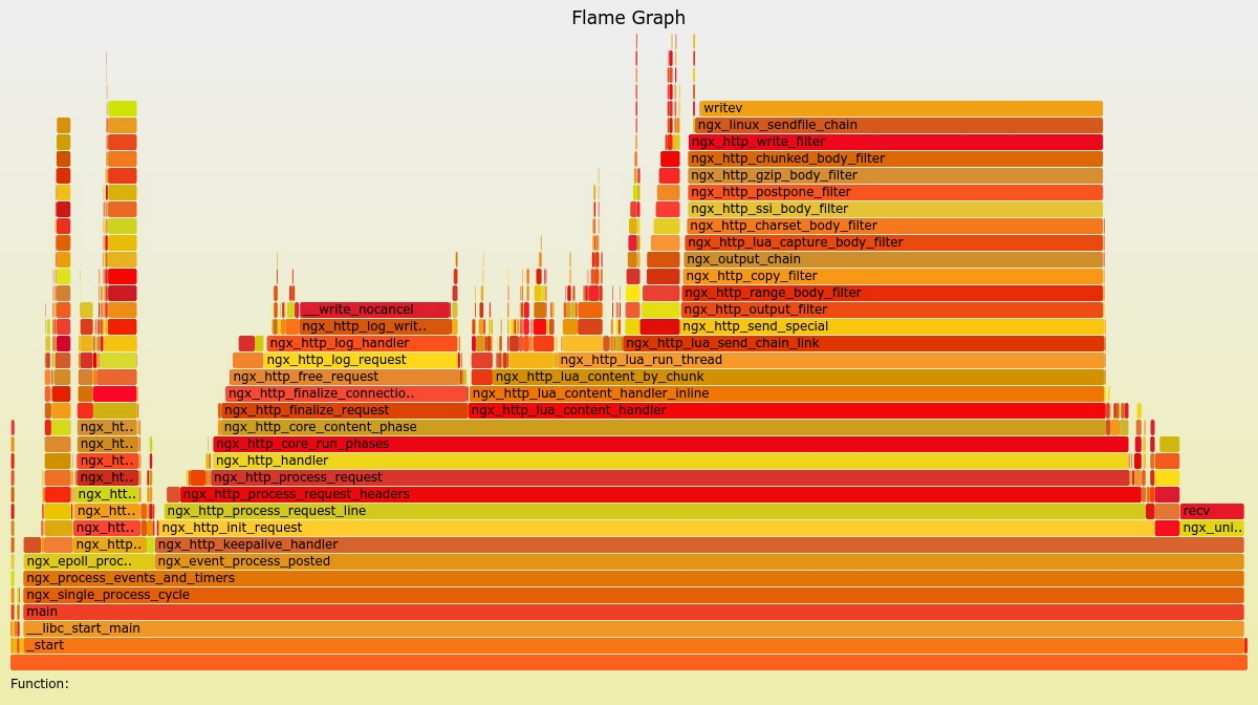
比如当前有一个绑定了本地地址 localhost 的 Redis 容器。如果另外一个容器需要复用这个网络堆栈，则需要如下操作：

```
$ sudo docker run -d --name redis example/redis --bind 127.0.0.1
$ # use the redis container's network stack to access localhost
$ sudo docker run --rm -ti --net container:redis example/redis-cli -h 127.0.0.1
```

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、吐吞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。

OpenResty 官方提供的由 perl 实现的栈抓取的程序是一个傻瓜化的 stap 脚本，如果有需要可以自行使用 stap 进行栈的抓取并生成火焰图，各位看官可以自行尝试。

显示的是什么

如何安装火焰图生成工具

安装 SystemTap

> 环境 CentOS 6.5 2.6.32-504.23.4.el6.x86_64

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。

首先需要安装内核开发包和调试包（这一步非常重要并且最为繁琐）：

```
# #Installaion:
# rpm -ivh kernel-debuginfo-($version).rpm
# rpm -ivh kernel-debuginfo-common-($version).rpm
# rpm -ivh kernel-devel-($version).rpm
```

其中$version 使用 linux 命令 uname -r 查看，需要保证内核版本和上述开发包版本一致才能使用 systemtap。(下载)

安装 systemtap：

```
# yum install systemtap
# ...
# 测试systemtap安装成功否：
# stap -v -e 'probe vfs.read {printf("read performed\n"); exit()}'

Pass 1: parsed user script and 103 library script(s) using 201628virt/29508res/3144shr
/26860data kb, in 10usr/190sys/219real ms.
Pass 2: analyzed script: 1 probe(s), 1 function(s), 3 embed(s), 0 global(s) using 2961
20virt/124876res/4120shr/121352data kb, in 660usr/1020sys/1889real ms.
Pass 3: translated to C into "/tmp/stapffFP7E/stap_82c0f95e47d351a956e1587c4dd4cee1_14
59_src.c" using 296120virt/125204res/4448shr/121352data kb, in 10usr/50sys/56real ms.
Pass 4: compiled C into "stap_82c0f95e47d351a956e1587c4dd4cee1_1459.ko" in 620usr/620s
ys/1379real ms.
Pass 5: starting run.
read performed
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 14.04 Desktop 上的安装方法

打开 Systemtap Ubuntu 系统安装官方 wiki 地址，获取 systemtap 安装包：

```
sudo apt-get install systemtap
```

其次我们还需要内核支持（具有 CONFIG_DEBUG_FS, CONFIG_DEBUG_KERNEL, CONFIG_DEBUG_INFO 和 CONFIG_KPROBES 标识的内核，不需要重新编译内核）。对于 `Ubuntu Gutsy` (或更老的版本)，必须重新编译内核。

生成 ddeb repository 配置：

```
# cat > /etc/apt/sources.list.d/ddebs.list << EOF
deb http://ddebs.ubuntu.com/ precise main restricted universe multiverse
EOF

etc.

# apt-key adv --keyserver keyserver.ubuntu.com --recv-keys ECDCAD72428D7C01
# apt-get update
```

针对 Ubuntu 14.04 版本（其他版本，只要不太老，相差不大），我们按照下面顺序尝试重新编译内核：

```
# uname -r
3.13.0-34-generic
# dpkg --list | grep linux | grep 3.13.0-34-generic
ii  linux-headers-3.13.0-34-generic                      3.13.0-34.60
amd64        Linux kernel headers for version 3.13.0 on 64 bit x86 SMP
ii  linux-image-3.13.0-34-generic                        3.13.0-34.60
amd64        Linux kernel image for version 3.13.0 on 64 bit x86 SMP
ii  linux-image-extra-3.13.0-34-generic                  3.13.0-34.60
amd64        Linux kernel extra modules for version 3.13.0 on 64 bit x86 SMP
# apt-get install linux-image-3.13.0-34-generic
```

上面的输出比较乱，大家要跟紧一条主线，`3.13.0-34-generic` 也就是 `uname -r` 的输出结果（如果您的系统和这个不一样，请自行更改），结合刚刚给出的 systemtap 官方 wiki 我们可以知道，正确的安装包地址应当是 `linux-image-**` 开头。这样我们，就可以很容易找到 `linux-image-3.13.0-34-generic` 是我们需要的。

火焰图绘制

首先，需要下载 ngx 工具包：Github地址，该工具包即是用 perl 生成 stap 探测脚本并运行的脚本，如果是要抓 Lua 级别的情况，请使用工具 ngx-sample-lua-bt

```
# 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）
hippo    14857     1  0 Jul01 ?        00:00:00 nginx: master process /opt/openresty/n
ginx/sbin/nginx -p /home/hippo/skylar_server_code/nginx/main_server/ -c conf/nginx.con
f
hippo    15010 14857  0 Jul01 ?        00:00:12 nginx: worker process
# ./ngx-sample-lua-bt -p 15010 --luajit20 -t 5 > tmp.bt （-p 是要抓的进程的pid --luajit20
|--luajit51 是LuaJIT的版本 -t是探测的时间，单位是秒， 探测结果输出到tmp.bt）
# ./fix-lua-bt tmp.bt > flame.bt  (处理ngx-sample-lua-bt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

```
# stackcollapse-stap.pl flame.bt > flame.cbt
# flamegraph.pl flame.cbt > flame.svg
```

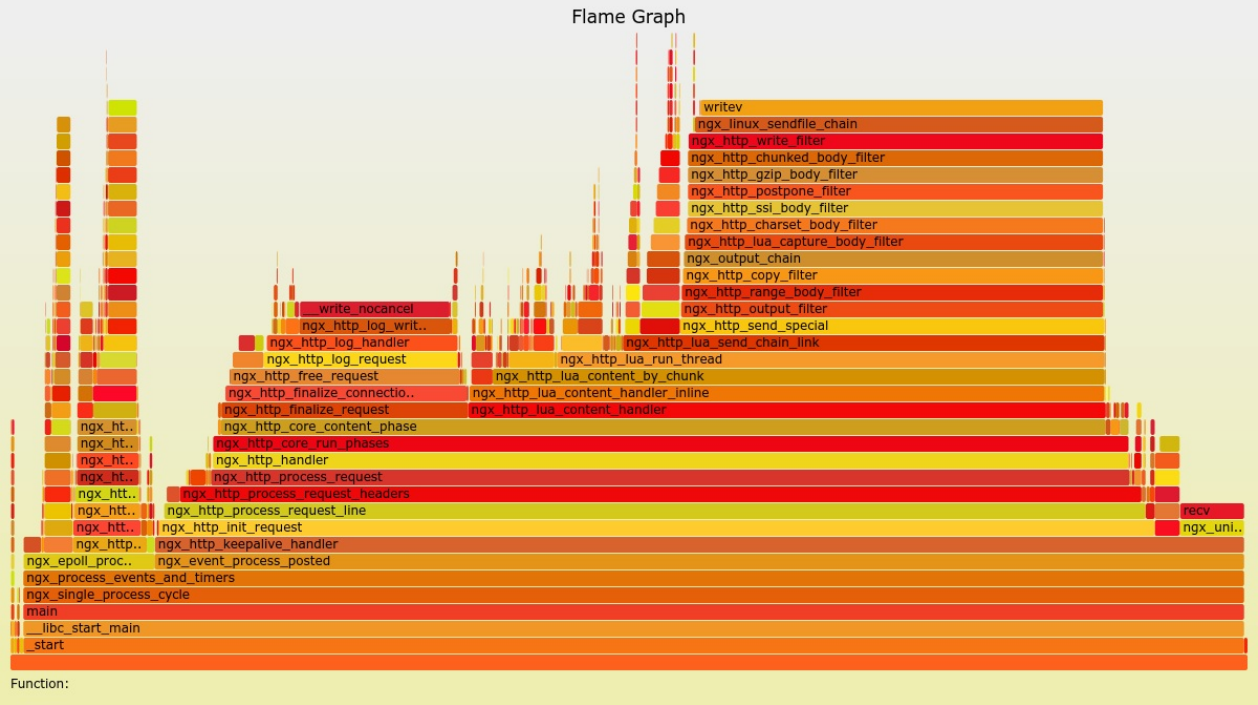
如果一切正常，那么会生成 flame.svg，这便是火焰图，用浏览器打开即可。

问题回顾

在整个安装部署过程中，遇到的最大问题便是内核开发包和调试信息包的安装，找不到和内核版本对应的，好不容易找到了又不能下载，@！￥#@……%@#，于是升级了内核，在后面的过程便没遇到什么问题。 ps：如果在执行 ngx-sample-lua-bt 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，不要惊讶哦～

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 函数以正确地方式处理导致的，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
  ```

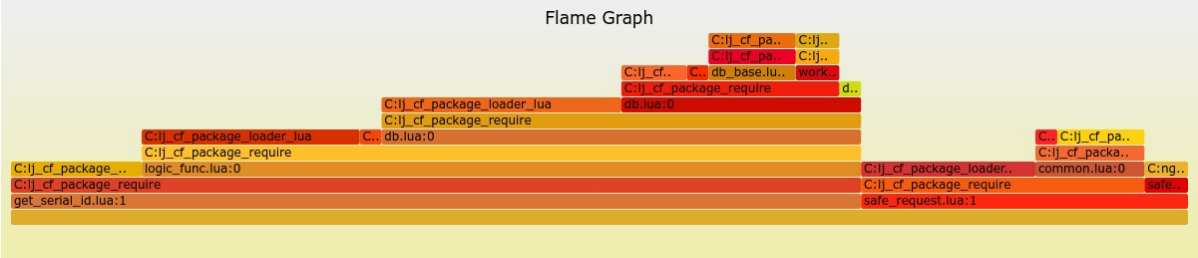
- 使用 lj-lua-stacks.sxx抓取栈信息，并用 fix-lua-bt 工具处理：

```
# making the ./stap++ tool visible in PATH:
$ export PATH=$PWD:$PATH
# assuming the nginx worker process pid is 6949:
$ ./samples/lj-lua-stacks.sxx --arg time=5 --skip-badvars -x 6949 > tmp.bt
Start tracing 6949 (/opt/nginx/sbin/nginx)
Please wait for 5 seconds
$ ./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”。

Vanilla 介绍

Vanilla 是新浪移动事业部系统架构组的同学基于 OpenResty 自主研发的一个 Web 开发框架，目前服务于新浪移动后端多条产品线及核心接口池。

当前版本

V0.1.0-rc4.0

项目地址

GitHub：https://github.com/idevz/vanilla

GitOSC：http://git.oschina.net/idevz/vanilla

kong 介绍

Kong 是在客户端和微服务间提供转发功能的 API 网关，基于 OpenResty 研发，并通过插件来扩展自身功能，官方网站上已提供二十多个插件模块，涉及到认证、安全、日志等功能。

当前版本

V0.9.0

项目地址

GitHub: https://github.com/Mashape/kong

官方网站

Homepage: https://getkong.org/

如何对 Nginx Lua module 添加新 api

本文真正的目的，绝对不是告诉大家如何在 Nginx Lua module 添加新 api 这么点东西。而是以此为例，告诉大家 Nginx 模块开发环境搭建、码字编译、编写测试用例、代码提交、申请代码合并等。给大家顺路普及一下 git 的使用。

目前有个应用场景，需要获取当前 Nginx worker 数量的需要，所以添加一个新的接口 ngx.config.workers()。由于这个功能实现简单，非常适合大家当做例子。废话不多说，let's fly now！

> 获取openresty默认安装包（辅助搭建基础环境）：

```
$ wget http://openresty.org/download/ngx_openresty-1.7.10.1.tar.gz
$ tar -xvf ngx_openresty-1.7.10.1.tar.gz
$ cd ngx_openresty-1.7.10.1
```

> 从 GitHub 上 fork 代码

- 进入lua-nginx-module，点击右侧的 Fork 按钮
- Fork 完毕后，进入自己的项目，点击 Clone in Desktop 把项目 clone 到本地

> 预编译，本步骤参考这里：

```
$ ./configure
$ make
```

`注意这里不需要make install`

> 修改自己的源码文件

```
# ngx_lua-0.9.15/src/ngx_http_lua_config.c
```

> 编译变化文件

```
$ rm ./nginx-1.7.10/objs/addon/src/ngx_http_lua_config.o
$ make
```

搭建测试模块

> 安装perl cpan 点击查看

```
$ cpan
cpan[2]> install Test::Nginx::Socket::Lua
```

> 书写测试单元

```
$ cat 131-config-workers.t
# vim:set ft= ts=4 sw=4 et fdm=marker:
use lib 'lib';
use Test::Nginx::Socket::Lua;

#worker_connections(1014);
#master_on();
#workers(2);
#log_level('warn');

repeat_each(2);
#repeat_each(1);

plan tests => repeat_each() * (blocks() * 3);

#no_diff();
#no_long_string();
run_tests();

__DATA__

=== TEST 1: content_by_lua
--- config
    location /lua {
        content_by_lua_block {
            ngx.say("workers: ", ngx.config.workers())
        }
    }
--- request
GET /lua
--- response_body_like chop
^workers: 1$
--- no_error_log
[error]
```

```
$ cat 132-config-workers_5.t
# vim:set ft= ts=4 sw=4 et fdm=marker:
use lib 'lib';
use Test::Nginx::Socket::Lua;

#worker_connections(1014);
#master_on();
workers(5);
#log_level('warn');

repeat_each(2);
#repeat_each(1);

plan tests => repeat_each() * (blocks() * 3);

#no_diff();
#no_long_string();
run_tests();

__DATA__

=== TEST 1: content_by_lua
--- config
    location /lua {
        content_by_lua_block {
            ngx.say("workers: ", ngx.config.workers())
        }
    }
--- request
GET /lua
--- response_body_like chop
^workers: 5$
--- no_error_log
[error]
```

单元测试

```
$ export PATH=/path/to/your/nginx/sbin:$PATH    #设置nginx查找路径
$ cd ngx_lua-0.9.15                      # 进入你修改的模块
$ prove t/131-config-workers.t           # 测试指定脚本
t/131-config-workers.t .. ok
All tests successful.
Files=1, Tests=6,  1 wallclock secs ( 0.04 usr  0.00 sys +  0.18 cusr  0.05 csys =  0.
27 CPU)
Result: PASS
$
$ prove t/132-config-workers_5.t         # 测试指定脚本
t/132-config-workers_5.t .. ok
All tests successful.
Files=1, Tests=6,  0 wallclock secs ( 0.03 usr  0.00 sys +  0.17 cusr  0.04 csys =  0.
24 CPU)
Result: PASS
```

提交代码，推动我们的修改被官方合并

- 首先把代码 commit 到 GitHub
- commit 成功后，以次点击 GitHub 右上角的 Pull request -> New pull request
- 这时候 GitHub 会弹出一个自己与官方版本对比结果的页面，里面包含有我们所有的修改，确定我们的修改都被包含其中，点击 Create pull request 按钮
- 输入标题、内容（you'd better write in english），点击 Create pull request 按钮
- 提交完成，就可以等待官方作者是否会被采纳了（代码 + 测试用例，必不可少）

来看看我们的成果吧：

pull request : 点击查看 commit detail: 点击查看

Test::Nginx 能指定现成的 nginx.conf，而不是自动生成一个吗

> Question：

如题

> Answer:

或许你可以用 nginx 的 include 指令来加载核心的 nginx 配置，比如

```
=== TEST 1: App
--- config
 include ../../../conf/app-core.conf;
--- request
GET /t
--- response_body
hello world
--- no_error_log
[error]
```

假设你的项目目录结构是这样的：

```
t/
└── a.t
conf/
└── app-core.conf
```

nginx.conf 的 boiler-plate 由 Test::Nginx 来自动生成是必要的，否则我们无法实现这里列举的 Test::Nginx 的各种高级测试模式，具体可以看这里。

access 日志字符编码问题

> Question：

```
    浏览器请求 json 数据格式
    http://127.0.0.1:8866/?a={%22b%22:%22%E4%B8%AD%22,%22a%22:%22zh_cn%22}
```

```
openresty access 日志为：
    {\x22b\x22:\x22aaa\x22}
```

配置文件

```
location /a.gif {
       internal;
       log_escape_non_ascii off;
       set_unescape_uri $u_jsona $arg_a;
       log_subrequest on;

       access_log /source/428/web/access.log accessjson;
}
```

自动转换的为 json中的 "" 请问有解决方法吗？

> Answer：

两种做法：

1. 避免使用标准的 access log 模块，而使用 lua-resty-logger-socket 这样的库。
2. 使用标准的 access log 模块，但在日志接收端对 \xx 进行转码。

share_dict 中的过期时间有时候过期有时候不过期？

> Question：

如题

> Answer：

是的，nginx 核心里的 timer 过期时间使用的是绝对时间戳。建议在改系统时间时使用 ntp 这样可以逐步校正时间的工具，以避免系统时钟直接往后跳或者太快向前蹦。

Lua 变量的传递和内存的使用

> Question：

Lua 初学者有一个关于内存使用的问题， 有的 language 在传递变量到 function 的时候是传递这块 memory 的 reference，有 的 language 是 copy 一份 memory 的 reference，有的 language 是 copy 了一份 memory，我想知道 lua 在传递变量的时候是如何实现的

> Answer：

Lua 里面的变量都是值的引用（或者说是值的别名），而并不是值的容器。所以 Lua 里的赋值和参数传递全部都是引用传递。

ngx.log 可不可以选择几个不同的 log path

> Question：

如题

> Answer：

你需要使用不同的 location {} 或者 server {} 进行隔离。如果不能这么做的话，你只能自己用 Lua 实现一个错误日志模块了，这倒也挺简单，只是需要注意复用文件名柄并及时刷缓冲（最好不要带缓冲）就好。

http://nginx.org/en/docs/ngx_core_module.html#error_log

不同的 location 使用不同的 error log path 即可。也就是上面说的 locatin 和server 隔离。

如何在后台开启轻量级线程完成定时任务？

> Question:

现在有一个场景：需要定时（30s）从redis里面拉取数据灌入 lua cache 共享内存。我现在是用 ngx.timer 这个 API 来实现的，请问这会不会有问题？因为对 ngx.timer 这个 API 不是很了解看了文档说是：在后台开启了一个轻量级线程来执行，与原来的请求脱钩。场景需求是拉取灌入的操作不能阻塞 worker 。

1. 请问这样能够满足需求吗？会不会有阻塞问题？
2. ngx.timer 我现在是放在 content_by_lua_file 中的，需要访问接口才能启动，有没有可以开启 worker 的时候就启动 timer 的办法（我试了放在 init_by_lua_file 里面好像不生效。。。）
3. 这个 timer 是每个 worker 里面都会有还是只有一个 worker 里面会有呢？（看文档貌似是只有一个 worker 里面会有，这也是我的需求）

> Answer:

1. 请问这样能够满足需求吗？会不会有阻塞问题？

   只要使用的都是 OpenResty 的 API 和库，是不存在阻塞问题的。只要大体确认一下当前 nginx 进程负载压力不要太高，能够确定获取到工作时间片即可。

2. 有没有可以开启 worker 的时候就启动 timer 的办法 。。。

   其实这里你已经点出关键字了，借助 init_worker_by_lua 即可。

3. 这个 timer 是每个 worker 里面都会有还是只有一个 worker 。。。

   通过 init_worker_by_lua 启动的 ngx.timer 是对每个 worker 的。如果这里需要控制 timer 的存在数量，可以借助 ngx.worker.id 完成 ngx.timer 数量控制，比如只启动一个或多个，并让他们确定绑定在哪个 worker 上。

如何使用 os.getenv 获取系统环境变量

> Question： 如题

> Answer：

如果你想在 Lua 中通过标准 Lua API os.getenv 来访问系统环境变量，例如 `foo` ，那么你需要在你的 nginx.conf 中，通过 env 指令，把这个环境变量列出来。 例如：

```
env foo;
```

lua-resty-mongol3 用户名、密码验证部分占用时间过长

> Question

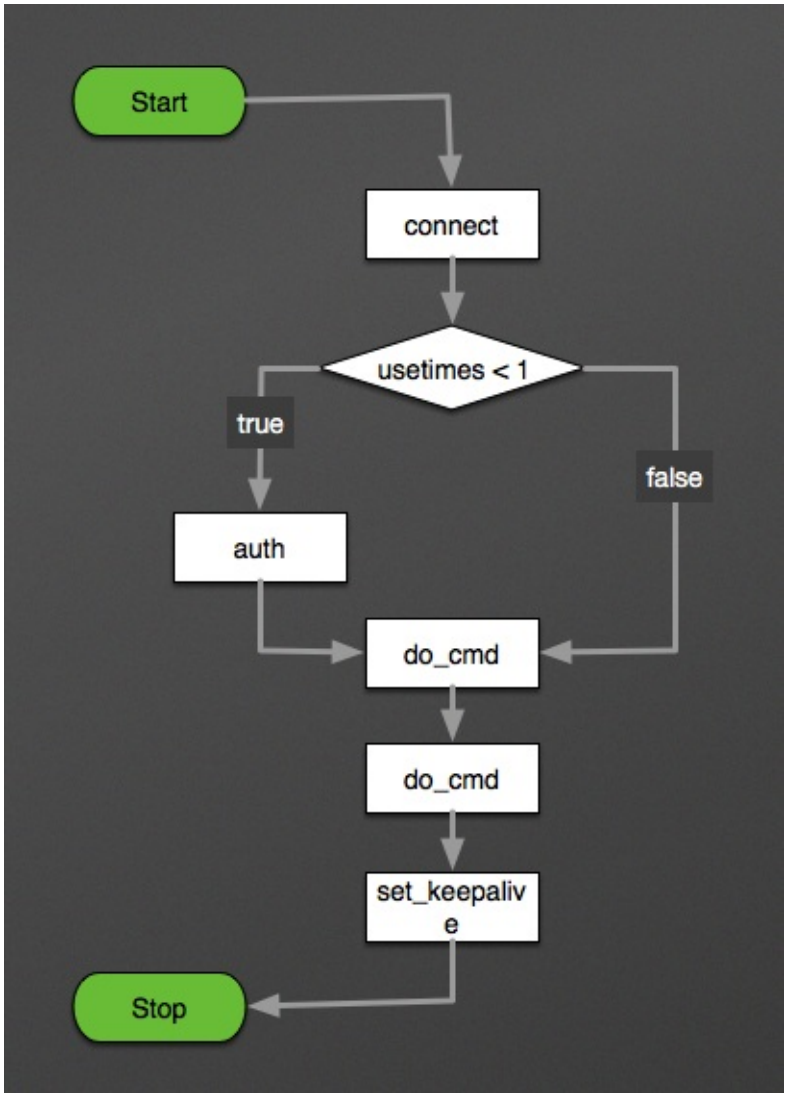
这个库的使用流程大致如下：

1. 创建一个 mongo 对象（local conn =mongo:new()）
2. 创建一个连接 mongodb（conn:connect("192.168.1.254",27017)）
3. 选择数据库（local db = conn:new_db_handle("openresty")）
4. 然后 auth 验证（db:auth_scram_sha1("username","password")）
5. 接收数据执行 insert 操作
6. 放入连接池（conn:set_keepalive(10000, 100)）

主要问题是用户登录部分占用将近 1/3 的时间，这个合理么？如何优化？

> Answer

为了回答上面的问题，我们先粗略整理一下数据库连接池的通常做法，看下面流程图：



主要区别：如何减少不必要的用户验证过程，合理高效的复用已有连接。其实对于已经验证过的连接，直接使用即可。

比较推荐的改进方法，参考 lua-resty-mysql 的实现，对不同 ip、port、db、user、password 绑定不同的连接池名字，让不同连接目的连接归类存放，并在数据库层直接完成用户验证动作。

OpenResty 中不能被 jit 编译的地方有日志提示吗？

> Question：如题
>
> Answer：

是可以借用 jit.v 或者 jit.dump 这两个 lua 模块可以输出 NYI 等日志。它们都是 luajit 自带的标准模块,在针对 jit 编译器做 lua 代码优化时，这两个模块之一是必须的。具体用法可以参考 lua-resty-core 项目的测试集。

jit.dump 输出的信息最详尽，从 trace 的 bc 到 ir 再到 mcode，jit.v 则比较简略。书鑫老师正在做一个更好的 luajit IR dumper，jit.dump 目前输出的 IR 列表不够直观，不够友好。书鑫老师的 IR 输出看着像高级语言伪码。

引用一下 lua-resty-core/t/md5.t 的第一个测试用例，为了突出重点，这里做了一下节选：

```
our $HttpConfig = <<_EOC_;
    lua_package_path "$pwd/lib/?.lua;../lua-resty-lrucache/lib/?.lua;;";
    init_by_lua_block {
        local v = require "jit.v"
        v.on("$Test::Nginx::Util::ErrLogFile")
        require "resty.core"
    }
_EOC_

__DATA__

=== TEST 1: set md5 hello
--- http_config eval: $::HttpConfig
--- config
    location = /md5 {
        content_by_lua_block {
            local s
            for i = 1, 100 do
                s = ngx.md5("hello")
            end
            ngx.say(s)
        }
    }
--- request
GET /md5
--- response_body
5d41402abc4b2a76b9719d911017c592
--- error_log eval
qr/\[TRACE   1 content_by_lua\(nginx\.conf:\d+\):3 loop\]/
--- no_error_log
[error]
```

解释一下，对是否执行了 LuaJIT 优化编译，最后是通过 `error_log eval` 这个小节匹配确定的。这是预期的一条被 JIT 优化编译的日志输出结果。

而对于没有被 JIT 编译优化，是有下面类似日志输出的，会出现 NYI（Not Yet Implemented） 关键字，比如下面：

```
[TRACE --- db_base.lua:247 -- NYI: bytecode 51 at db_base.lua:252]
```

有兴趣的同学，可以自己玩下 jit.dump ，对汇编比较了解的同学，有惊喜哦。

一个 openresty 内存“泄漏”问题

> Question：

大家好，这里向大家咨询一个在 openresty-1.9.15.1 下的泄漏问题，复现手段非常简单：

1. 直接在 openresty 下 ./configure 的时候，开启 mp4 模块，即 --with-http_mp4_module, 然后再配置文件配置一个location，开启 mp4 指令，例如：

   ```
   location /live/hls/ {
       mp4;
       root /home/test;
   }
   ```

   保证有一个测试 mp4 文件是可以访问的。

2. 使用ab测试该location,例如：

   ab -c 1000 -n 1000 "http://192.168.182.128:8081/live/hls/chuntianli.mp4?start=1&end=10"

> Answer：

这里的原因应该是启用 ngx_lua 模块时，会强制 glibc 的 malloc 使用 mmap 来分配内存，而不是 sbrk 这样的系统调用在 data segment 里面分配，目的是为了把低 2GB 的内存地址空间都预留给 LuaJIT 的 GC 来使用。glibc 对于 mmap 分配的块可能会更加激进地缓存 free 掉的内存页。所以你看到的所谓的“不释放”其实并不是应用程序不释放，而只是 glibc 这样的基础系统库缓存了起来。

为进一步确认这一点，你可以持续无限制地保持压力，然后观察 nginx worker 进程的 RSS 是否会无限增长。

在书鑫老师的建议下，春哥在 lua-nginx-module 仓库的 malloc-trim 分支里实现了 lua_malloc_trim N 这个新的 nginx 配置指令。该指令会在 nginx 每处理 N 个请求之后，自动调用 libc 的 malloc_trim() 函数（如果当前 libc 有这个函数的话）。默认 N 是 1000. 将 N 置为 0 时即禁掉自动 trim 的行为。这个新功能应该会显著降低流量高峰以后 nginx worker 进程的（RSS）内存占用。

更多细节请参见：

https://github.com/openresty/lua-nginx-module/tree/malloc-trim

https://github.com/openresty/lua-nginx-module/commit/f0b45946d

https://github.com/openresty/lua-nginx-module/issues/872#issuecomment-250988928

相同的，如果 nginx 因为某些原因内存占用在不停增长但是却无法正常减少，这时候建议把你的 openresty 里面的 lua-nginx-module 升级到 github 仓库里 master 分支上的最新版本。然后在生产 nginx.conf 配置用于测试：

```
lua_malloc_trim 1;
```

生产上不建议使用 1 这么小的数字，但测试时为了效果明显，用 1 比较清楚。因为使用 glibc 缓存了很多由应用释放掉的内存块，而没有把这些内存及时归还给操作系统。

书鑫老师先前弄了一个独立的最小化的 C 程序来演示 glibc 内存分配器行为的一些细微方面，主要是和 OpenResty/LuaJIT 相关的细节。我又自己做了一些修改，使它的自动化程度更高一些。大家可以自己改一改，玩一玩：http://agentzh.org/misc/test-malloc-trim.c。当然了，只能在 Linux + glibc 上面跑哈。

注 参考下面链接进行整理：

https://groups.google.com/forum/#!searchin/openresty/lua_malloc_trim%7Csort:relevance/openresty/JA1mZwPgKtU/GZVsvmjzBQAJ

用 do-end 整理你的代码

do-end 代码块主要是解决变量作用域问题。例如，下面这个示例代码将输出什么呢？

```
local x = 10
if x > 0 then
    local x = 17
    print(x)    -- output: 17
end
print(x)        -- output: 10
```

这里出现的本地变量，Lua 使用标准词法作用域，所以这里 lua 的变量可以按照思维习惯输出。这么做有下面几个原因：

- 变量作用域是静态的。你可以通过查看源代码，就能知晓什么变量和函数对应这代码中的每个标识。
- 变量范围有限，这有助于可读性，并避免一些错误。
- 避免命名冲突。
- 访问局部变量的速度比全局变量更快。

```
local i = 8

do
  local a = i
  x1 = a + 1
end         -- scope of `a` ends here
print(x1)   -- output: 9
print(a)    -- output: nil
```

没有 do-end 块，输出的结果就不一样了。

```
local i = 8

local a = i
x1 = a + 1
print(x1)   -- output: 9
print(a)    -- output: 8
```

这里做一些有趣的尝试，对于隐藏变量，看下面三个示例：

```lua
-- 本地变量外部函数
local func
do
  local a = 0
  func = function(inc)
    a = a + inc
    return a
  end
end

print(func(1))  -- output: 1
print(func(2))  -- output: 3
print(func(3))  -- output: 6
```

```lua
-- 全局函数
do
  local a = 0
  function func(inc)
    a = a + inc
    return a
  end
end

print(func(1))  -- output: 1
print(func(2))  -- output: 3
print(func(3))  -- output: 6
```

```lua
-- 方法
local tbl = {}
do
  local a = 0
  function tbl.func(self, inc)
    a = a + inc
    return a
  end
end

print(tbl:func(1))  -- output: 1
print(tbl:func(2))  -- output: 3
print(tbl:func(3))  -- output: 6
```

那么哪一种更好呢？如果只从 Lua 语言角度看，貌似已经有点难做出选择。但是考虑不同开发语言之间的通用，这三个风格都不足够通用。 所以这里笔者更推荐下面的方式实现上面的逻辑。

```
-- 类方式
local tbl = { a = 0 }

function tbl.func(self, inc)
    self.a = self.a + inc
    return self.a
end

print(tbl:func(1))  -- output: 1
print(tbl:func(2))  -- output: 3
print(tbl:func(3))  -- output: 6
```

lua 中如何 continue

要说 lua 语言中最让大家有些不适应的，应该有两个：

1. 数组下表是从 1 开始的
2. 循环中没有 continue 关键字

第一条，估计只能大家默默接受，没有翻身余地。

循环中没有 continue ，现在终于可以翻身变相实现又不失效率，毕竟每次为了 continue 让我们手工写很多 if、else 还是很恼火的。

LuaJIT 开始跟进了 Lua 5.2 语言和库的一些特性。详细信息大家可以到这里参考：http://luajit.org/extensions.html#lua52 。

其中第一个就大大的写明，开始支持 goto 和 ::labels:: 机制。我们赶紧来写我们的第一个 continue 例子：

```
for i=1, 3 do
    if i <= 2 then
        print(i, "yes continue")
        goto continue
    end

    print(i, " no continue")

    ::continue::
    print([[i'm end]])
end
```

输出结果：

```
$ luajit test.lua
1   yes continue
i'm end
2   yes continue
i'm end
3    no continue
i'm end
```

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。

调用 FFI 出现 "table overflow"

> Question：

1. 首先安装 uuid 模块（ubuntu/debian请安装：sudo apt-get install uuid uuid-dev 两个模块）
2. FFI 调用，代码示例：

```
local ffi = require 'ffi'

ffi.cdef[[
typedef unsigned char uuid_t[16];
void uuid_generate(uuid_t out);
void uuid_unparse(const uuid_t uu, char *out);
]]

local uuid = ffi.load('libuuid')

if uuid then
    local uuid_t   = ffi.new("uuid_t")
    local uuid_out = ffi.new("char[64]")
    uuid.uuid_generate(uuid_t)
    uuid.uuid_unparse(uuid_t, uuid_out)
    result = ffi.string(uuid_out)
    print(result)
end
```

当使用这个的时候刚开始不会出现错误，但是运行一段时间后就会出现 table overflow， 第二天早上回来看到的，请问遇到过这样的情况么?

> Answer：

貌似这里每请求都调用 ffi.cdef？建议把 ngx_ffi.lua 实现为一个真正的 Lua 模块，这样就可以享受只加载一次的好处：

http://wiki.nginx.org/HttpLuaModule#Data_Sharing_within_an_Nginx_Worker

你遇见的 table overflow 的错误很容易通过下面这个独立的 Lua 脚本复现：

```
-- test.lua
local ffi = require "ffi"
while true do
    ffi.cdef[[
    typedef struct { void* ev_ptr; void* char_ptr; } bif_result;
    ]]
end
```

命令行上的执行结果如下：

```
$ /usr/local/openresty/luajit/bin/luajit-2.0.0-beta10 test.lua
/usr/local/openresty/luajit/bin/luajit-2.0.0-beta10: table overflow
stack traceback:
        [C]: in function 'cdef'
        a.lua:6: in main chunk
        [C]: ?
```

所以只要把 ffi.cdef 移出循环就不会溢出了。（而在你的场景中，则是把 ffi.cdef 调用移进 Lua 模块加载代码。）

可以参考一下 lua-resty-string 库中的 resty.md5 模块的实现：

https://github.com/agentzh/lua-resty-string/blob/master/lib/resty/md5.lua

注 根据下面链接完成整理

https://groups.google.com/forum/#!topic/openresty/zGxwOqUN4fc

如何定位 openresty 崩溃 bug

遇到 coredump 的问题，发现无论是在 google group 或 QQ 交流群里，由于大家不会提问（主要是信息不全)，最终找到问题的过程是磕磕绊绊。这里总结一下如何定位崩溃、异常 bug，一般可以按照下面顺序来自检（适用于linux、mac 平台）：

1. 更新的最新版本复现，以确保不是已经修复过的 bug
2. 移除不必要的第三方模块，以排除第三方的问题
3. 在 `./configure` 的时候指定 `--with-cc-opt="-g -O0"`
4. 准备完整的复现配置和步骤
5. 用 gdb 提供奔溃点的 backtrace

这里的步骤，完全可以复制到其他程序或场景下。以后再遇到这类问题，你就可以踏实按部就班的查找原因了。