本是后山人，偶作前堂客。

醉舞经阁学卷书，坐井说天阔。

新年伊始，万象更新。由于业务需求，年初开始接触 nginx。犹如一个黑洞一样，nginx 的高并发、高稳定、高扩展的特性深深的吸引了我。源码面前，了无秘密。认识上帝最好的方法就是认识解上帝创造世界。鉴于此原因，开始捧起 nginx 的源码阅读开来。

几乎每次阅读，都会拍案惊奇。惊奇于作者的设计、思维、对事物的理解。每次阅读，都会对自己说：原来还可以这样，原来球还可以这样踢？

字如其人，对于工程师而言，代码也如其人。通过阅读代码，有时候可能会复现出作者在写这段代码时在想些什么，作者的性格是什么样子的，作者要达到一种什么目的等。虽然，作者本身没有言语，但是在代码的点滴中，会读到一些代码之外的东西。代码是工程师按照他的灵魂构造的，就像上帝按照自己的样子创造了人类一样。

鉴于年后工作一直很忙，也就没有怎么持续性的读下去。只是偶尔会打开看几眼。每次看的时候，也就顺手把看代码时的感想写了下来，久而久之就形成了此文。文中不但讲了一些编程的故事，还讲了一些通过阅读 nginx 源码想到的的一些社科、人文等的东西。虽然是从源码中来，到源码中去，但是也不拘泥于源码。

2013 年，7 月　王伟

1. nginx 中的编码规范

车书文轨–始皇帝一统天下

史记：秦始皇本纪

[6] 耶和华说，看哪，他们成为一样的人民，都是一样的言语，如今既作起这事来，以后他们所要作的事就没有不成就的了。
[7] 我们下去，在那里变乱他们的口音，使他们的言语彼此不通。
[8] 于是，耶和华使他们从那里分散在全地上。他们就停工，不造那城了。

创世纪 :十一章

1883 年,11 月 18 日的正午时分，美国东部的时钟全部回拨。从此，人类开始使用人类自己的时间。

公司的力量

nginx 统一的编程标准不但让人赏心悦目。更重要的是，可以让初学之很容易的猜到模块的功能，函数的作用等等。

优秀体现于细节，正是因为这样，nginx 才能够实现代码即文档。

nginx 中的注释很少的可怜（翻翻 nginx 的源码便知），但是这并没有妨碍别人去读 nginx，更重要的是别人还能读的懂，不但读得懂，还能编写 nginx 扩展，这算的上是软件界的一个奇迹。神作呀。于此对应，我认为中国的文学界还有一个奇迹：文章没有标点。统一的力量有多强大，规范的作用有多强大现在大家看到了吧？我有时候在想，nginx 的作者在写 nginx 的时候，是不是也在构思着一个强大的帝国？一种一统的文化？一种非我不可入此门的雄心壮志？这些不得而知，但是从代码的片段中，却能让人感觉到这些的存在。

nginx 有 200 多个文件，随便打开任何一个文件，排版完全一样（注意呀，不是几乎，是完全）。不需要两个文件对比才能看出它的规范，任何一个文件的书写都是规范。天下一统，又见沙皇俄国征服天下的雄心。

nginx 中的这些规范不一定那么的完美，但是所有的文件都遵循一个规范着着实让人吃惊。

1.1. nginx 编码规范

粗略整理的一下，在 nginx 中主要的一些规范如下：

（1）使用 K&R 编码风格；
（2）每行代码不得超过 80 列（http://www.aqee.net/80-chars-per-line-is-great/）；
（3）不使用 TAB 缩进代码，而是用 4 个空格进行缩进（至于原因，你懂得）；
（4）除宏定义外，其他字符一律小写，多个单词之间用_进行连接；
（5）注释一律使用/**/，不得使用//进行行注释；
（6）中缀运算符在操作数和运算符之间必须空一格；
（7）逗号后必须空一格；
（8）文件开始空一行；
（9）较为完整的代码块之间空 2 行，如下图所示的注释和#include 语句，#include 语句和之后的函数声明；

```

/*
 * Copyright (C) Igor Sysoev
 * Copyright (C) Nginx, Inc.
 */


#include <ngx_config.h>
#include <ngx_core.h>
#include <nginx.h>


```

（10）函数声明中，如果一行显示不下，则折行后的声明须空 4 个空格；

```
static char *ngx_set_priority(ngx_conf_t *cf, ngx_com
static char *ngx_set_cpu_affinity(ngx_conf_t *cf, ngx
    void *conf);
static char *ngx_set_worker_processes(ngx_conf_t *cf,
    void *conf);


static ngx_conf_enum_t  ngx_debug_points[] = {
    { ngx_string("stop"), NGX_DEBUG_POINTS_STOP },
    { ngx_string("abort"), NGX_DEBUG_POINTS_ABORT },
    { ngx_null_string, 0 }
};
```

（11）结构体数组的{和=位于同一行；

（12）结构体数组的{、}和内容之间空一格；

（13）数组元素内容上下对齐；

（14）较大的结构体数组，数组元素开始空一行；

```
static ngx_conf_enum_t  ngx_debug_points[] = {
    { ngx_string("stop"), NGX_DEBUG_POINTS_STOP },
    { ngx_string("abort"), NGX_DEBUG_POINTS_ABORT },
    { ngx_null_string, 0 }
};


static ngx_command_t  ngx_core_commands[] = {

    { ngx_string("daemon"),
      NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_FLAG,
      ngx_conf_set_flag_slot,
      0,
      offsetof(ngx_core_conf_t, daemon),
      NULL },

    { ngx_string("master_process"),
      NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_FLAG,
      ngx_conf_set_flag_slot,
      0,
      offsetof(ngx_core_conf_t, master),
      NULL },

```

（15）注释上下对齐；

```
ngx_module_t  ngx_core_module = {
    NGX_MODULE_V1,
    &ngx_core_module_ctx,                  /* module context */
    ngx_core_commands,                     /* module directives */
    NGX_CORE_MODULE,                       /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    NULL,                                  /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    NULL,                                  /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
};
```

（16）变量名称上下对齐，仅限字母对齐；

```
struct ngx_command_s {
    ngx_str_t             name;
    ngx_uint_t            type;
    char               *(*set)(ngx_conf_t *cf, ngx_comman
    ngx_uint_t            conf;
    ngx_uint_t            offset;
    void                 *post;
};
```

（17）函数定义时，函数返回值类型单独占一行；

（18）函数定义的{单独占用一行；

```
int ngx_cdecl
main(int argc, char *const *argv)
{
    ngx_int_t         i;
    ngx_log_t        *log;
    ngx_cycle_t      *cycle, init_cycle;
    ngx_core_conf_t  *ccf;

    ngx_debug_init();

    if (ngx_strerror_init() != NGX_OK) {
        return 1;
    }

    if (ngx_get_options(argc, argv) != NGX_OK) {
        return 1;
    }
```

（19）if/while/for/switch 语句中关键字和判断条件之间、判断条件和{须空一格，关键字和{同行。

（20）else 和 {、}同行，并且 else 和{、}之间须空一格

（21）else 语句之前空一行

```
if (ngx_process == NGX_PROCESS_SINGLE) {
    ngx_single_process_cycle(cycle);

} else {
    ngx_master_process_cycle(cycle);
}
```

（22）条件表达式须折行时，关系运算符位于下一行的行首，并且与上一行条件表达式的第一个字符对齐；

（23）如果条件表达式折行，则{独占一行；

```
488         for (i = 0; i < ccf->env.nelts; i++) {
489             if (ngx_strcmp(var[i].data, "TZ") == 0
490                 || ngx_strncmp(var[i].data, "TZ=", 3) == 0)
491             {
492                 goto tz_found;
493             }
494         }
```

（24）函数调用折行时，函数参数须上下对齐；

```
641             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
642                           ngx_rename_file_n " %s to %s failed "
643                           "before executing new binary process \"%s\"",
644                           ccf->pid.data, ccf->oldpid.data, argv[0]);
645
```

1.2. 总结

1883 年 11 月 18 日的正午时分，美国东部把所有的时钟回拨，从此，美国开始使用自然时间调度火车运行，然后美国就靠着这辆货车踏上了征服全球的历程。

2． nginx 请求处理流程

2.1. 始有野心--nginx 的启动过程

nginx 的启动过程位于 src/core/nginx.c 文件中的 main()函数中，下面是一个简化版本的 main()函数。

```
int main ()
{
    ngx_get_options(argc, argv);

    /* 检查配置文件 */
    if (!ngx_test_config) {
        return 0;
    }

    /* 各种初始化操作 */
    ngx_time_init();
    ngx_regex_init();
    log = ngx_log_init(ngx_prefix);
    ngx_ssl_init();

    /* 读取命令行参数，处理命令行选项 */
    ngx_save_argv(&init_cycle, argc, argv);
    ngx_process_options(&init_cycle);

    /* 初始化操作系统：pagesize，cachline，cpu，rlimit */
    ngx_os_init();

    /* 沙场秋点兵 */
    for (i = 0; ngx_modules[i]; i++) {
        ngx_modules[i]->index = ngx_max_module++;
    }

    /* 核心模块初始化，创建各种文件和目录，创建共享内存，打开 listen 的端口，所有模块初始化 */
    ngx_init_cycle(&init_cycle);

    ngx_init_signals(cycle->log);
    ngx_daemon(cycle->log);
    ngx_create_pidifle(&ccf->pid, cycle->log);

    /* nginx 多进程模型 */
    ngx_master_process_cycle(cycle);
}
```

【*备注 1*】优秀体现于细节：如果再仔细看下 main()函数，会发现，nginx 的异常判断无处不在，这也为 nginx 的高稳定性打下了牢牢的基石。千里之提，溃于蚁穴。这一点尤其重要，想想 iphone/mac 的成功无不专注于细节；想想闻名于世的瑞士名表，每个零件都是手工打造。细节决定成败，优秀体现于细节。

【*备注 2*】去除无用的花哨，适用的设计原则：看完 nginx 的 main 函数，惊叹于作者的精密思维。有别于 lighttpd 服务器，nginx 在启动时会首先进行配置文件的检查，如果配置文件有错误，则停止后续所有的工作。对于 webserver 的 start 而言，这没有什么优势。但是对于 reload 而言，这种设计上的天然优势，给 nginx 的高稳定性打上了天然的基因。

试想一下：对于一个流量为 nKbps 的服务，如果在重启的过程中 webserver 因为配置文件的错误而当掉会带来多大的损失。我们宁可新的配置文件没有生效，也不想影响用户的访问。对于 lighttpd 而言，这一点根本没法保证，有可能你你修改完 lighttpd.conf 后忽然发现 lighttpd 起不来了。

但是 nginx 绝对不会出现这种情况。首先：nginx 不支持配置文件热加载；其次，如果想启用新的配置文件，需要手动重启 nginx；再者，如果重启时，配置文件检查出错，nginx 不会做任何其它操作，nginx 还是以之前的进程，之前的配置继续运行，继续提供服务。

虽然有时候不支持热加载感觉挺不爽的，但是用久了都感觉好。对于一个可以承担高并发的 webserver 而言，安全性比热加载要重要的多。正所谓：鱼和熊掌不可兼得。

nginx 的高稳定性正是在这些看起来毫不起眼的地方一点点进化出来的。

【*备注 3*】简单可依赖的设计原则，数组访问再也不用担心下标越界了：这段函数中比较有意思的就是“沙场秋点兵”那个 for 循环啦，*for (i = 0; ngx_modules[i]; i++)*。如果对于一个定义时，没有指定数组大小的数组而言，如何判断数组的大小呢？

nginx 的作者又是用了一个简单的再也不能再简单的设计思想：数组末尾增加一个内容为 NULL 的元素，然后采用遍历链表的思想。思想虽然简单，但是却广泛应用于 nginx 的各种地方：模块指令，http 变量数组……

还记得当年的 MFC 吗？虽然，现在很少有人用了，但是当年是火遍大江南北呀。还记的你给窗体控件添加事件处理函数时是怎么添加的吗？还记的那个 BEGIN_MESSAGE_MAP(CpassDlg, CDialog)和 END_MESSAGE_MAP()吗？MFC 实际上也是利用这两个宏（其实还有一个 DECLARE_MESSAGE_MAP 宏）生成一个 lpEntries 的数组，该数组中会存放所有关心的消息和对应的处理函数。那么我们看下 END_MESSAGE_MAP()宏的具体实现？

```
#define END_MESSAGE_MAP() {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0} };
```

看到了吗？也是一个空的数组元素。垃圾的构架各有各的毛病，但是优秀的构架总是那么相像。

【*备注 4*】高度模块化编程原则：整个 main()函数也就区区 200 行代码（实际上核心代码也就 30 行左右），但是却实现了 nginx 的整个启动流程。模块化编程思维由此可见一斑，由此也可看出，作者在设计&编写 nginx 时的高屋建瓴和良苦用心。真正做到了高内聚低耦合的设计原则。

2.2. 建朝立业--nginx 王国的建立

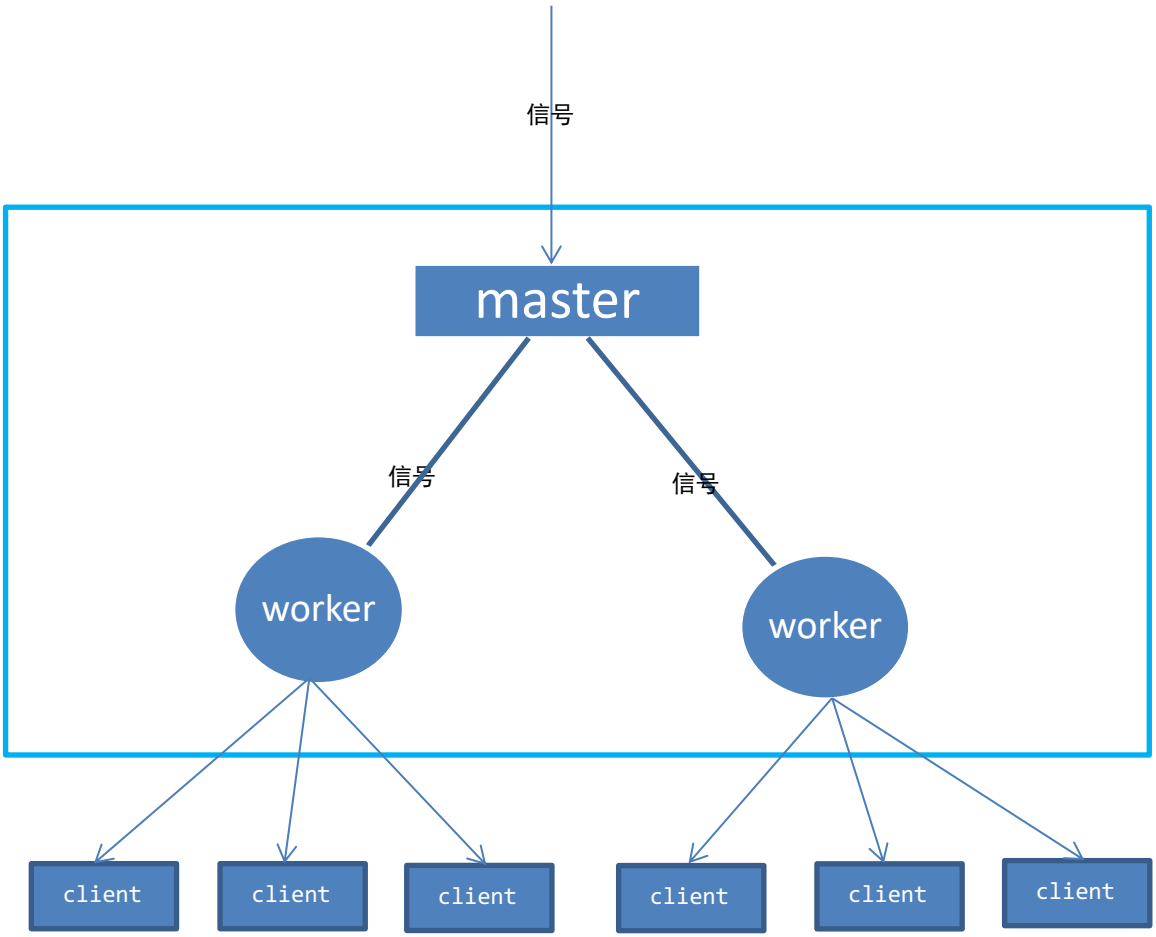

图 2-1 nginx 进程模型

nginx 多进程模型沉思

（1）对于每个 worker 进程而言，都是作为一个独立的进程接收客户端请求，因此避免了多线程模型中“锁”的开销；

（2）避免多线程中“锁”的控制，有利于开发中的程序调试以及问题跟踪；

（3）每个 worker 进程各自为政，相互之间没有干扰，一个进程退出之后，其它进程还可以正常工作，不会因为一个进程的终止而终止服务；

（4）master 进程随时监控 worker 进程的运行状态，一旦发现有小弟挂掉了，会马上 fork 一个小弟来替代之前挂掉的小弟；

【*备注 1*】想到这里，忽然让我想起了东周列国时期周天子和诸侯国的关系。如果作为一个整体，周王姬发分封诸国的社会构架无疑是最理想的管理方式。但是和 nginx 系统不同的是，master 进程具备至高权力，worker 进程对 master 进程绝对忠诚，这也正是计算机系统的优点。人类社会就完全不同了，自古以来，良禽择木而栖成了朝代更迭的一个接口。由此也可看出，绝对忠诚以及绝对的控制权对于稳定性是何等的重要，犹如宙斯对众神的控制。nginx 作者在构思 nginx 的时候，是否也借鉴了人类管理学中的放权的思想？是否也想在思考集中制弊端？集中是管理的一个不好的例子就是 apache 的 mod_php 扩展的管理。把权力放下去嘛，知人善用。

【*备注 2*】多进程的模型还让我想到了风险投资领域的名言：不要把鸡蛋放在一个篮子里。采用多进

程的模型，分散了因客户端访的多样性以及网络服务的复杂性带来的风险，进而避免了多米诺骨效应引起的服务崩溃。要知道 nginx 面对的是资源和并发的威胁，因此风险控制是必须的。

【*备注 3*】因为 worker 进程对 master 进程的绝对忠诚，才造就了 nginx 多进程模型的高稳定性。这又让我想起了春秋战国，如果各诸侯国都对周天子绝对的忠诚，世界历史会不会重写？

【*备注 4*】对于 nginx 的多进程而言，和用户连接直接打交道的永远都是 worker 进程，master 进程从来都不会抛头露面。这就是 master 进程的无上权威。master 总是居于幕后，运筹帷幄，去保证 nginx 王国的稳定性和高处理能力。每当这个王国出现灾难的时候，比如一个 worker 进程死掉了，master 进程总能通过管理手段去维护王国的稳定性。真正的高科技是什么？是管理。

如下是对进程模型的详细介绍。

在 main()函数中调用 ngx_init_cycle()函数完成配置文件解析、模块初始化、建立监听套接口等这些开奖僻壤的基础准备之后（正所谓兵马未动，粮草先行），ngx 就调用 ngx_master_process_cycle()去实现整个 nginx 王国的建立。

ngx_master_process_cycle()函数位于 src/os/unix/ngx_process_cycle.c 文件中。

```
void ngx_master_process_cycle(ngx_cycle_t *cycle){
     /* 处理信号屏蔽，防止在做帝王事业之时收到外界的干扰 */
     sigemptyset(&set);
     sigaddset(&set, SIGCHLD);
     /* …… */
     sigprocmask(SIG_BLOCK, &set, NULL);
     /* 杯酒释兵权 */
     sigemptyset(&set);
     /* 改换年号 */
     ngx_setproctitle(title);

     /* 奉天命,佣兵自重，维护王权 */
     ccf = ngx_get_conf(cycle->conf_ctx, ngx_core_module);
    ngx_start_worker_process(cycle, ccf->worker_processes, NGX_PROCESS_RESPAWN);
     ngx_start_cache_manager_processes(cycle, 0);

     /* 励精图治，创制开元盛世 */
     for ( ;; ) {
          /* 打完天下，开始做天下，无时不刻都在看奏折 */
          sigsuspend(&set);

          if (ngx_reap) {
               live = ngx_reap_children(cycle);
          }
          if (ngx_quit) {
               ngx_signal_worker_process(cycle, NGX_SHUTDOWN_SIGNAL);
               continue;
          }
     }
}
```

【*备注 1*】这个函数结束以后，一个 nginx 的帝国就已经建立起来了。函数本身很简单，总共也就 200 行左右，但是却实现了一个 nginx 的进程管理模型，不得不惊叹于 nginx 作者的模块化程度。各个函数功能明确，各司其职。这不是一个靠后期优化就能达到的水平和高度，这是一种代码的自然进化的结果。从这里，我们又可以看出，nginx 的作者在构思 nginx 时的那种严谨的思维和极具创造性和设计性的睿智。

【*备注 2*】在那个励精图治的过程中，我们会看到很多精巧的设计原则。比如示例代码中的 continue 语句，此乃神作。为什么呢？各位看官莫急，且听我娓娓道来。为什么 master 接到 SIG_QUIT 信号，向 worker 进程发送 SIG_QUIT 信号后不是直接 break 或 return，而是采用了一个 continue 呢？试想一下，宋太祖杯酒释兵权，明太祖杀功臣替子孙斩除荆棘，不看到 worker 进程死掉，皇上（master）怎么能安寝？所以嘛，master 不能直接推出，得等着监斩官发来信号（SIG_CHILD）才能安心（对应着对 ngx_reap 的处理），然后收到监斩官的信号之后，master 进程就可以安心啦，接下来就是调用 ngx_master_process_exit(cycle)。

另外，master 进程还得对 worker 进程负责一些善后工作，否则，哪些 worker 进程就成了僵尸进程啦。

【*备注 3*】如果再看一下对 TERM 信号的处理，发现 nginx 真不愧是一个高稳定性的 webserver。nginx 接到 TERM 信号后，不会立即结束掉所有工作进程，而是延迟 50ms，以便让 worker 进程能有时间处理掉已经接受到的请求。对于一个流量为 1Kbps(10Kbps)的服务而言，50ms 的时间已经是足够长了。如果 worker 进程在 1s 后还在处理工作，那 master 进程就不耐烦了，直接 kill 掉吧。有人说你想谋反，给了你 1000ms 去辩解一下你却一点重点信息都不提供，皇上都不耐放了，不杀你怎么对得起自己的江山呀。

好啦，聊到到这里的时候，nginx 已经启动起来了，如果各位看官想仔细了解 nginx 对各种信号的详细处理机制，可以去仔细阅读 *src/os/unix/process_cycle.c* 文件中 *ngx_master_process_cycle()* 函数中的 for()循环部分，总共也不过 200 行，这里就不一一介绍了。但是不管怎样，对各种信号的处理，应该都会遵循上述原理吧？你说呢？

接下来比较重要的就是 worker 进程是如何去相应客户端的请求啦，要知道 nginx 可是启动了 *N*(*N*>1)个 worker 进程呢？这 *N* 个 worker 进程时如何去争权夺位的呢？

2.3. 九子夺嫡--worker 进程相应用户请求

每个 worker 进程都想去接收用户请求（因为它们都监听一个网络套接口），就像每个皇子都想做皇帝一样。这一点，纵观中国历朝历代，都做的不好。几乎每个朝代都会出现皇子争权的事件。

还是欧洲人牛，搞出一个欧盟轮值国主席。就是嘛，大家一起玩才好。

那么在 nginx 王国中，是如何处理九子夺嫡的悲剧的呢？

nginx 是一个多进程程序，80 端口为各 worker 进程共享，每当有连接出现时，势必会产生惊群效应。当内核 accept 一个链接时，会唤醒所有等待中的进程，但实际上只有一个进程能获取连接，其它的进程都被无效唤醒。为此，nginx 提供了一把 accept 锁避免九子夺嫡的悲剧。

nginx 事件处理的入口之处为 *src/event/ngx_event.c* 文件中的 ngx_process_events_and_timers(ngx_cycle_t *cycle)函数。该函数称的上是短小惊叹，整个函数不过区区 100 行而已。

如下是一个简写版的 ngx_process_events_and_timers 函数。

```
void ngx_process_events_and_timers(ngx_cycle_t *cycle)
{
    if (ngx_use_accept_mutex) {
        if (ngx_accept_disabled > 0) {
            ngx_accept_disabled--;
        } else {
            if (ngx_trylock_accept_mutex(cycle) == NGX_ERROR) {
                return;
            }
            if (ngx_accept_mutex_held) {
                flags |= NGX_POST_EVENTS;
            } else {
                if (timer == NGX_TIMER_INFINITE
                    || timer > ngx_accept_mutex_delay)
                {
                    timer = ngx_accept_mutex_delay;
                }
            }
        }
    }
    (void) ngx_process_events(cycle, timer, flags);
    if (ngx_posted_accept_events) {
        ngx_event_process_posted(cycle, &ngx_posted_accept_events);
    }
    if (ngx_accept_mutex_held) {
        ngx_shmtx_unlock(&ngx_accept_mutex);
    }
    if (delta) {
        ngx_event_expire_timers();
    }
    ngx_event_process_posted(cycle, &ngx_posted_events);
}
```

【*备注 1*】如上的代码基本上阐述了 nginx 中是如何处理九子夺嫡的。代码只有 70 行左右，但是可谓字字玑珠，值得细细品味和沉思。为了保持代码的原汁原味，我都没敢做批注。

【*备注 2*】代码开篇就是用了 ngx_use_accept_mutex 来判断进程是否启用 accept mutex 来避免惊群效应和做负载均衡。由此可见，nginx 还是允许我们自己去指定是否需要使用 accept mutex 锁的，因为如果是单进程的话，是不需要这个锁去避免惊群效应的。从此，也更加体现 nginx 作者的“适用”思维，即通用，又高效。

【*备注 3*】接下来的 ngx_accept_disabled 判断更是绝妙精伦，堪称神作。这个值实际上是这样得来的：*ngx_accept_disabled = ngx_cycle->connection_n / 8- ngx_cycle->free_connection_n;*。

因此可以看出，如果进程的可用连接数小于总的连接数的 1/8，ngx_accept_disable > 0，否则 < 0。因此，如果这个进程运气一直很好，总是能够抢到锁去处理客户端请求，那么没多长时间 ngx_accept_disable 就大于 0 了，此时如果再让他去处理势必力不从心了。每个光鲜的背后都有一个悲苦的故事，nginx 进程也不例外，不行不能再接请求了，力不从心呀，让出去吧，让给其它兄弟吧，有肉大家一起吃嘛，我先休养生息一下。因此，当 ngx_accept_disabled > 0 时，进程就自动放弃去抢这个 accept 锁了。但是，不能白白让出天赋的神权呀，怎么着呀，让出一次，就相当于恢复了一点血，又强壮了，因此，ngx_accept_disabled--。

nginx 采用这种简单的机制就实现了强大的负载均衡机制，难道不是鬼斧神工吗？尤其是当 nginx 的并发较大时，这种策略表现的更加突出，基本上每个进程的负载都差不多。另外，这种机制还保证了不管什么情况下，nginx 总是有一部分连接是可用的，正所谓要留点绝招嘛。

关于 ngx_accept_disabled 的计算位于 *src/event/ngx_event_accept.c* 文件的 *ngx_event_accept()*函数中，详情大家可以去仔细阅读下。

九子夺嫡的悲剧就这么轻而易举的被化解了，正所谓退一步海阔天空！

【*备注 4*】如果这个进程强到锁了，那接下来怎么办？要是普通人，那指定是先过一把瘾，好不容易才抢到的呢，是吧。

但是 nginx 的表现让人惊讶：

（1）它只是做了一个 NGX_POST_EVENTS 标记而已，这个标记的作用就是将产生的事件放置到一个队列中而已，该队列中存放着所有 accept 的事件。

（2）接下来，nginx 就对 accept 事件队列进行处理，处理结束后就把 accetp mutext 锁给释放掉（如果抢占了的话），一点不留恋权位。

（3）然后，nginx 再调用 *ngx_event_process_posted*()函数去处理普通的队列事件。从而通过这种稀有资源提前释放，非稀有资源推迟处理的原则实现了高并发的特性。设计模式的原则中不是也有类的实例化推迟吗（工厂模式把类的实例化推迟到子类）？书非借而不能读也，正是这个道理。这忽然让我想起之前的数据库的实例化，也让我想明白为什么在类的构造函数中实例化一个数据库连接是多么的可怕。

【*备注 5*】还有一点不知道大家注意到了没有？如果进程没有抢到锁，那么进程在下次去抢锁之前会延迟一段时间？为什么呢？因为 nginx 的 accpet mutex 是一把自旋锁，如果不延迟一段时间，那么进程就会一直对没有抢到 accept mutext 耿耿于怀，他就一直在自己的思想阴影中旋转而忘记了还有很多伟大的事业等着他去处理。

2.4. 社会化大产--nginx 的强大生产力

龙生九子，各有神通。在巧妙的避免了九子夺嫡的悲剧实现负载均衡之后，接下来就是如何去处理每一个来自客户端的请求并返回相应数据给客户端。

在这里，nginx 给大家展示出来的是一派以明确的社会分工为基础的、流水线生产方式的社会化大生产的景象。

社会分工的出现，极大的提高了生产力水平，为社会化大生产提供了基础条件。就目前而言，那种原始的自给自足的生产方式越来越难啦？1913 年，福特应用创新理念和反向思维提出了汽车的流水线生产，并且制定了流水线的不同环节的多项标准，从而将汽车的生产效率提高了 4488 倍！

从社会分工而言，在基础构架基础之上，nginx 对 http 请求的处理也分为了不同的模块：handler 模块，filter 模块，upstream 模块，负载均衡模块。各模块分工不同，各司其职，处理流水线上的不同工作。

nginx 从 ngx_posted_events 队列中取出一个请求，然后是经过什么样的过程产生用户请求的数据的呢？这时，主要涉及到的模块就是 http 模块了，大部分的代码都位于 src/http 目录下。nginx 对 HTTP Request 的处理流水线一般如下图所示：

图 2-2 nginx 对 http 的请求处理流程

【*备注 1*】http 响应流水线的各个环节的处理函数在图中已经给出了，因此，就不在去做代码摘录了。

【*备注 2*】就像福特规定了汽车流水线上的各个环节的输入、输出标准一样，nginx 作者对于图中个流水线的各个生产环节也做了明确的规定，以保证不同环节之间不会做重复的工作。并且，在代码实现环节上，也遵循流水线的逻辑思维：当前环节处理完成后，主动调用下一环节的函数已完成以下环节的操作。整个过程就是这样被串了起来。

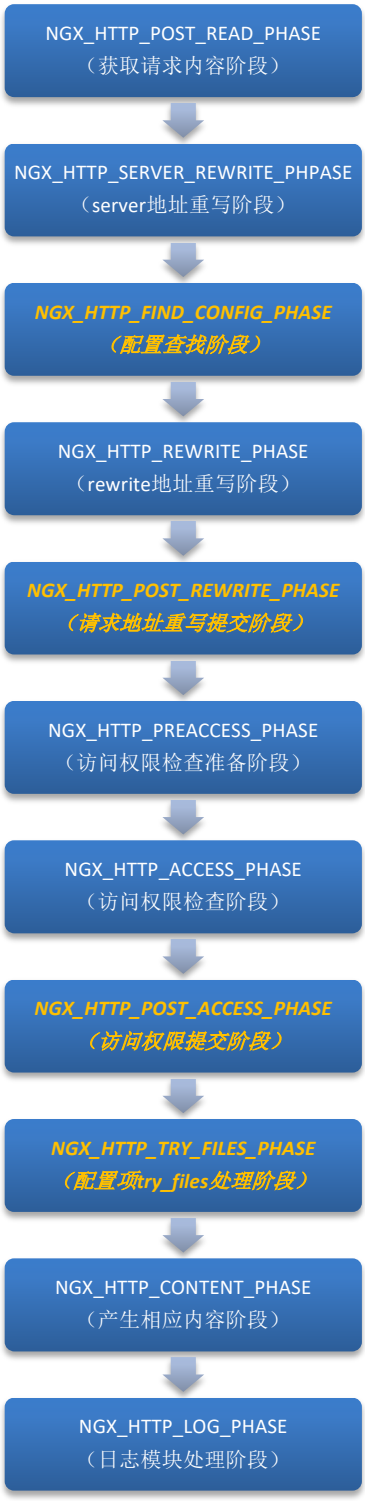

图 2-3 nginx-phase handler 的 11 个阶段

【*备注 3*】为了方便标准的制定，nginx 把一个请求划分为 11 个阶段，如图 2-3 所示（具体实现可

参见 *src/http/ngx_http.c* 文件中的 *ngx_http_init_phase_handlers()*函数，各阶段名称是定义于 *ngx_http_core_module.h* 文件中的枚举类型：ngx_http_phases）。看到这里的时候太惊叹于 nginx 作者的对复杂事物的精准划分，这是构成整个流水线生产的基础。

划分为 11 个阶段还有什么好处？

（1）模块化编程，又是模块化，精髓所在。避免把所有的代码堆积到一起，构成一个庞大的函数。正所谓分而治之嘛。模块化的本身就降低了耦合性，nginx 作者从思维上和代码上给我们展示了什么才算得上是真正的低耦合。

（2）避免出现成千上万的 if(){} else {}判断分支。if 语句在异常判断中的使用会让人感到赏心悦目，但是在业务逻辑上的使用却让人感到发蒙。如果你看到一个满满全是 if 判断的业务逻辑你会怎样想？是不是很抓狂，就算要改一点点的地方，你也得去读懂所有的 if 语句，不是吗？那 nginx 给我们呈现的流水线有什么好处呢？永远不需要关注我不关心的代码段，可维护性更强。简化了代码逻辑，没有大片大片的 if 语句啦，业务升级更方便。

（3）可裁剪性更强，并不是所有的请求都会调用相同的 handler 的；甚至并不是所有请求都会调用所有的这 11 个阶段，比如一个静态文件的请求就不需要去请求 fastcgi，一个模块权限检查的请求就不需要去请求 NGX_HTTP_CONTENT_PHASE 阶段一样。如果这一大堆东西都搞在一起，想都不敢想，太恐怖了，无异于世界灭亡。

（4）作为一个初学者，我想说的是，这种方式阅读起来更轻松。我宁可阅读 10 个每个只有 10 行的函数，也不愿意阅读 1 个 100 行的函数，就是这个道理。

（5）还有一点就是，社会分工的出现导致了社会化大生产，从而极大的提高了生产效率。

【*备注 4*】实际上 nginx 的 handler 产生数据之后，在数据发送到客户端之前还需要被依次传递个各种不同的过滤模块去过滤产生的数据。这个过程相当于生产线的最后一道润色流程。比如，我想修改下客户端显示的 webserver 名称等等，一般都是在这个过程去处理。

我一直认为 nginx 过滤模块是“*封闭开放原则*”的绝佳例子。不需要去修改 handler 模块的任何东西，就可以实现修改响应数据的目的。这也是软件设计中梦寐以求的一种境界。

2.5. 总结

写到这里，对于 nginx 从启动到处理请求的整个环节已经介绍完毕了。其中想到的一些东西也是想到哪就写到哪，未免有些不成章法，还希望大家谅解。

人们往往有一个特点：三句话不离本行。但是，如果你跳不出你的圈子，你就看不到问题的全貌。如果沉迷于细节，也就看不到全貌。正所谓横看成岭侧成峰。正因如此，本章并没有很深入的去就某一个细节，某一段代码进行分析。

3． nginx 的高效和稳定

垃圾的构架各有各的特点，但是优秀的构架总是那么的相似。对于 nginx 而言，是什么使得 nginx 可以具备高并发，高稳定性的特点呢？冰冻三尺，非一日之寒。每次想到这个问题，我想，nginx 的作者为了这一目的是不是也是呕心沥血呢？毕竟任何优秀的代码，任何优秀的构架都是自然进化的结果。

3.1. 阿克琉斯之踵--无所不在的异常判断

忒提斯为了让儿子炼成“金钟罩”，在阿克琉斯刚出生时就将其倒提着浸如冥河，遗憾的是，阿克琉斯被母亲捏住的脚后跟却不慎露在水外，留下了唯一一处“死穴”。在特洛伊战争中，阿克琉斯被帕里斯一箭射中脚踝而死。再强大的英雄也有致命的软肋。

再优秀的开发人员，也会犯下错误。nginx 的作者深知这一道理，因此，在 nginx 的源码中，随处可以看到对异常逻辑的判断。比如：内存申请的判断，函数调用的判断，nginx 不但不信任系统的函数，甚至连自己写的函数都不信任。正是这种不信任机制才保证了 nginx 的高稳定性。从之前的 nginx 的启动过程对配置文件的检查，以及到代码层面中各种异常的判断，无不透漏着 nginx 作者对安全性的良苦用心。

一般而言，在 nginx 的模块开发中，如果增加了扩展导致 nginx 出 core，那么大部分人首先意识到的肯定是自己写的模块出现了内存泄露了。这种对原生 nginx 的信任正是基于 nginx 那些无处不在的异常判断。

可以看一下 src/http/modules/ngx_http_static_module.c 这个模块。这个模块应该称得上是最简单、最古老、最正宗的，任何一个 webserver 都必须实现的一个 handler 模块了。正如这个模块的文件名所述，该模块的作用就是读取静态文件，并把文件的内容作为响应结果返回给客户端。对于模块开发而言，估计第一个要看的模块，要学习的模块也是这个模块。

ngx_http_static_module 模块就两个函数，ngx_http_static_init()用于把该模块挂载在 NGX_HTTP_CONTENT_PHASE 阶段，ngx_http_static_hanadler()函数为真正的产生数据的函数。

```
274   static ngx_int_t
275   ngx_http_static_init(ngx_conf_t *cf)
276   {
277       ngx_http_handler_pt        *h;
278       ngx_http_core_main_conf_t  *cmcf;
279
280       cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
281
282       h = ngx_array_push(&cmcf->phases[NGX_HTTP_CONTENT_PHASE].handlers);
283       if (h == NULL) {
284           return NGX_ERROR;
285       }
286
287       *h = ngx_http_static_handler;
288
289       return NGX_OK;
290   }
291
```

这个函数对于内存申请之后，使用申请的内存之前有个重要的操作就是判断内存申请是否成功。在使用某个对象之前，先判断该对象是否是我们需要操作的类型已经改对象是否可以被操作时如此的重要，如果没有这个操作，轻者导致程序逻辑错误，重者程序直接出 core。

想想那些年，被 core 折磨的死去活来的样子，想想 php 那满屏的警告日志：`PHP Warning:  Invalid argument supplied for foreach()`，想想对系统函数的过分信任……

有时候，我们总说调优，提高并发。实际上对于 php 而言，少打一些警告信息都会有利用并发量的提高。php 中没打一条警告日志都会进行一次系统调用呀，系统调用可都是写费时费力的操作。并且，在打

日志的过程中，还要开启互斥锁保证文件同一时刻只有一个进程在写。如果一个脚本中，不做任何的异常判断，warning 日志打的漫天遍野，并发量能上去吗？

漫天遍野的 warning 日志还有另外一个问题，就是会淹没了哪些有价值的 php 日志，典型的经济学上的柠檬效应。

ngx_http_static_handler()函数就不再做详细介绍了，大家有兴趣可以去看一下，函数不是很长，200 多行的样子。再看的时候也可以想象，如果这个模块让我们自己实现，我们会写成什么样子？是不是会嫌麻烦而去掉了应有的异常判断？

成功就是简单的事情重复做，别人的代码之所以优秀就是因为别人做了我们懒得去做的事情。

3.2. 赫尔墨斯的飞鞋--nginx 的高效算法

在古希腊众神中，赫尔墨斯在奥林匹斯山单人宙斯和众神的使者和传译，他是宙斯最忠实的信使，为宙斯传送消息，并完成宙斯交给他的各种任务。赫尔墨斯行走敏捷，精力充沛。他经常被描绘成一个脚着带翼飞鞋的年轻人。看来，高效是需要法宝的。

nginx 又是如何对抗高并发和有限的资源的呢？nginx 中的哪些技巧保证了 nginx 能够在高压力下还能表现出良好的性能呢？

（1）字符串的实现

```
16  typedef struct {
17      size_t      len;
18      u_char     *data;
19  } ngx_str_t;
20
```

nginx 使用如上所定义的字符串有什么优点呢？

首先，通过长度来表示字符串的长度，避免调用 strlen()函数计算字符串的长度；

其次，可以重复引用一段字符串内存，data 可以指向任意的地址，长度表示结束，避免字符串的拷贝操作。

【*备注 1*】对于普通的应用而言，这个小技巧可能被认为是无所谓。但是 nginx 面对的是每秒几万上百万的并发量，面对这种压力，任何一个小的改进都能极大的优化性能。就像在数据挖掘中的一个经典问题：任何优秀的算法在海量数据面前都会失效。时刻谨记：在高并发面前，任何一个微笑的问题都会被无限的放大。由此也能看出作者的良苦用心，费这么大劲无非是避免几个 cpu 周期而已。

（2）字符串复制操作的实现

对于字符串拷贝操作的函数大家肯定都自己实现过，那么看到 nginx 中的实现后我们又能想到什么呢？高效的法宝正是在这些看起来毫不起眼的地方实现的。由此也看出 nginx 作者为了 nignx 的高效是做了多少努力。

```
#if ( __INTEL_COMPILER >= 800 )

/*
 * the simple inline cycle copies the variable length strings up to 16
 * bytes faster than icc8 autodetecting _intel_fast_memcpy()
 */

static ngx_inline u_char *
ngx_copy(u_char *dst, u_char *src, size_t len)
{
    if (len < 17) {

        while (len) {
            *dst++ = *src++;
            len--;
        }

        return dst;

    } else {
        return ngx_cpymem(dst, src, len);
    }
}

#else

#define ngx_copy                  ngx_cpymem

#endif
```

(3)nginx 在解析请求行时的请求方法的判断

在看到 *ngx_http_parse.c* 文件中的 *ngx_http_parse_request_line()* 函数时，又一次惊呆了。竟然把字符串的比较转换为整数的比较？那真是玩出水平了。小优化，大效益。特意摘录了这段代码，如下：

```
                case 3:
                    if (ngx_str3_cmp(m, 'G', 'E', 'T', ' ')) {
                        r->method = NGX_HTTP_GET;
                        break;
                    }

                    if (ngx_str3_cmp(m, 'P', 'U', 'T', ' ')) {
                        r->method = NGX_HTTP_PUT;
                        break;
                    }

                    break;
```

```
#define ngx_str3_cmp(m, c0, c1, c2, c3)                                      \
    *(uint32_t *) m == ((c3 << 24) | (c2 << 16) | (c1 << 8) | c0)
```

(4)nginx 对不同资源的使用

相信大家还记得 nginx 在九子夺嫡中介绍的 ngx_accept_mutex 的使用，作为稀有资源，nginx 的做法是在必须使用之时才申请，一旦不再使用就立即释放，让别的进程去使用。

与之对应的是 nginx 对内存的使用，对于每一个连接，nginx 在连接建立之时都会为之分配内存池，之后建立在连接之上的所有对内存的操作都位于这块内存池上，等请求结束之后，统一释放内存池。因为，在目前而言，内存已经不算事稀缺资源，这样做更有利于提高效率。

想到这里，忽然想起了之前的数据库连接不够用的情况，其实不就是因为在 php 的构造函数中初始化了一个数据库连接导致的吗？对于稀缺资源的过早占用，推迟释放有时候不正是导致并发量上不去的原因

吗？

3.3. 设计模式在 nginx 中的应用

（1）老子一气化三清–nginx 中的三大模块

关于这一点，就不再多讲了，网络上讲的太多了。任何一本介绍 nginx 模块开发的资料首先讲的就是 nginx 的模块分为 handler，filter，upstream 三类。每种模块各司其职，nginx 会在合适的时候去选择一个合适的模块去响应用户请求。

（2）责任链模式的应用–nginx 中得过滤模块

nginx 利用责任链的设计模式去设计过滤模块，巧妙的避免了代码中大量的 if else 的逻辑判断，保证代码的开闭设计原则。这一点我们可以自省一下，想想我们写的代码中的大量的 if 业务逻辑语句，太恐怖了。

4. nginx 中自旋锁的实现

```
void
ngx_spinlock(ngx_atomic_t *lock, ngx_atomic_int_t value, ngx_uint_t spin)
{
    ngx_uint_t  i, n;
    for ( ;; ) {
        if (*lock == 0 && ngx_atomic_cmp_set(lock, 0, value)) {
            return;
        }
        if (ngx_ncpu > 1) {
            for (n = 1; n < spin; n <<= 1) {
                for (i = 0; i < n; i++) {
                    ngx_cpu_pause();
                }
                if (*lock == 0 && ngx_atomic_cmp_set(lock, 0, value)) {
                    return;
                }
            }
        }
        ngx_sched_yield();
    }
}
```

【*备注 1*】ngx_atomic_t 的定义位于 os\unix\ngx_atomic.h，如下所示：

```
typedef long                        ngx_atomic_int_t;
typedef unsigned long               ngx_atomic_uint_t;
typedef volatile ngx_atomic_uint_t  ngx_atomic_t;
```

由此可见，lock 参数实际上就是一个 unsigned long 的整型数据，value 参数实际上是一个 long 型的变量，而 spin 则是一个 usigned int 类型的变量.

【*备注 2*】在 nginx1.0.1（包括 1.0.1 版本）之前的版本，如果 cpu 支持原子操作，则共享锁的实现还能明显的看到是 ngx_spinlock 的宏定义，如下所示。

```
#define ngx_shmtx_lock(mtx)   ngx_spinlock((mtx)->lock, ngx_pid, 1024)
#define ngx_shmtx_unlock(mtx) (void) ngx_atomic_cmp_set((mtx)->lock, ngx_pid, 0)
```

但是之后的版本，为了代码风格上的统一，没有进行显示宏定义操作，而是统一到函数定义中去，但是实际上还是 ngx_spinlock 函数的重写而已。

【*备注 3*】ngx_atomic_cmp_set()也是一个比较有趣的函数。其函数实现如下所示。

可以看到，如果 lock 和 old（一般为 0）相等，则说明没有获取锁，此时可把 lock 设置为获得锁的进程号并且返回 1。否则，如果 lock 不为 0，则该函数直接返回 0.

```
277  static ngx_inline ngx_atomic_uint_t
278  ngx_atomic_cmp_set(ngx_atomic_t *lock, ngx_atomic_uint_t old,
279      ngx_atomic_uint_t set)
280  {
281      if (*lock == old) {
282          *lock = set;
283          return 1;
284      }
285
286      return 0;
287  }
288
```

【*备注 4*】下面在看下 ngx_ncpu 这个变量，这个变量实际上很简单，就是简单的调用 *ngx_ncpu = sysconf(_SC_NPROCESSORS_ONLN)*来获取系统的 cpu 核数。

【*备注 5*】接下来就是 ngx_cpu_pause()啦。

该函数也是和平台相关的，大部分平台都是这届宏定义为一个空操作，比如：

#define ngx_cpu_pause()

但是也有的会使用 gcc 的内联汇编进行定义

#define ngx_cpu_pause() __asm__ ("pause")

Intel 对 pause 指令的解释如下：

PAUSE 指令提升了自旋等待循环（spin-wait loop）的性能。当执行一个循环等待时，Intel P4 或 Intel Xeon 处理器会因为检测到一个可能的内存顺序违规（memory order violation）而在退出循环时使性能大幅下降。PAUSE 指令给处理器提了个醒：这段代码序列是个循环等待。处理器利用这个提示可以避免在大多数情况下的内存顺序违规，这将大幅提升性能。因为这个原因，所以推荐在循环等待中使用 PAUSE 指令。

PAUSE 的另一个功能就是降低 Intel P4 在执行循环等待时的耗电量。Intel P4 处理器在循环等待时会执行得非常快，这将导致处理器消耗大量的电力，而在循环中插入一个 PAUSE 指令会大幅降低处理器的电力消耗。

PAUSE 指令虽然是在 Intel P4 处理器开始出现的，但是它可以向后与所有的 IA32 处理器兼容。在早期的 IA32 CPU 中，PAUSE 就像 NOP 指令。Intel P4 和 Intel Xeon 处理器将 PAUSE 实现成一个预定义的延迟(pre-defined delay)。这种延迟是有限的，而且一些处理器可以为 0。PAUSE 指令不改变处理器的架构状态（也就是说，它实际上只是执行了一个延迟——并不做任何其他事情——的操作）。

这个指令的操作在 64 位和非 64 位模式下是一致的。

【*备注 5*】接下最后一个不见就是 ngx_sched_yield()了。函数实现如下：

```
70  #if (NGX_HAVE_SCHED_YIELD)
71  #define ngx_sched_yield()  sched_yield()
72  #else
73  #define ngx_sched_yield()  usleep(1)
74  #endif
75
```

其实可以粗暴的认为，该函数的意思就是让进程休眠 1us。对于 cpu 这种高速运转的怪物而言，休眠 1us 哪是相当奢侈啦。

sched_yield()主要功能：

进程自动让出 cpu 而让另一个级别等于或高于当前进程的进程先运行。调用该函数之后，当前进程将移动到可执行进程队列的末尾，并且新的进程将获取 cpu。如果在进程队列中只有当前进程，那么这个函数将会立刻返回并且继续执行当前线程的程序。

在有了如上的解释时候，我们再来看 nginx 自旋锁的原理：

（1）在用户态尝试竞争一个共享资源时，如果竞争不到，则不断尝试竞争。但是此时应该尽量避免使用内核提供的 mutex 等变量机制，因为在用户态涉及到内核，就意味着降低效率。

（2）要想在用户态实现竞争一个共享资源，必须借助 cpu 提供的原子操作指令。如果是 SMP 多 cpu，还需要 lock 指令锁总线。

（3）为了避免在长时间竞争却一直得不到资源导致的不断自旋浪费 cpu，在每两次尝试之间需要间隔一段时间。并且随着尝试次数的增加，间隔时间也增加。

间隔期间可以让 cpu 稍加休息。注意，此时的休息绝不是让出 cpu，这依赖于 cpu 提供 pausse 指令。如果 cpu 没有提供 pause 指令也没关系，简单的执行空操作就可以了。有时候在 3D 编程中就是使用空操作去实现动画的。关于 cpu 的 pause 指令可以参见【备注 4】。

（4）在等待相当长时间还是得不到锁之后，只好让出 cpu。但让出的时间必须短，否则就不叫自旋锁了。如何既能让出 cpu，又可以很快的回来？这就用到了【备注 5】提供的 ngx_sched_yield()啦。

4.1. 总结

关于这一章的总结，暂时还没想好怎么写。呵呵。先到这里吧，有时间了再补充。

5． Nginx 定时器的实现

5.1. 为什么要使用 RBTree 作为定时器

src/event/ngx_event_timer.h

```
31
32   extern ngx_thread_volatile ngx_rbtree_t  ngx_event_timer_rbtree;
33
```

由上述代码可以看出，在 nginx 中，使用红黑树实现的事件定时器。那么为什么使用红黑树作为定时器就可以是精准的定时器呢？

实际上，这和定时器的特性有关系。每次时间循环，都需要找出超时时间最小的事件，然后判断其是否超时。如果发现有超时的时间还得从定时器中删除，并且处理该事件。如果有新的时间的话，还需要把该时间的超时时间插入到超时队列中。

从上述描述可以看出，影响定时器的精准度的因素有：节点插入时间、节点查询时间、节点删除时间、节点排序时间。综合如上的因素，那么利用红黑树去实现定时器是一个不错的选择。

实际上还可以使用 AVL 树作为计时器，但是 AVL 是一颗严格的平衡二叉树，虽然时间复杂度和红黑树一致，但是统计时间比红黑树要高。

当然也可以使用 B 树，或者是 B 的衍生数据结构是实现定时器。操作系统的文件索引用的是 B 树，貌似好多数据库底层也是用 B 树实现索引。但是，如果使用 B 树的话，未免有些大材小用。关于 B 树的更多详细介绍可以参考《算法导论》，在此就不一一介绍了。

5.2. 如何使用定时器

在 nginx 中，以超时时间戳作为红黑树的 key。在 event/ngx_event_timer.h 文件中的 ngx_event_add_timer()函数中可以看到具体的操作。对其摘录如下：

```
static ngx_inline void
ngx_event_add_timer(ngx_event_t *ev, ngx_msec_t timer)
{
    ngx_msec_t  key;
    key = ngx_current_msec + timer;
    ev->timer.key    = key;
    ngx_rbtree_insert(&ngx_event_timer_rbtree, &ev->timer);
    ev->timer-set    = 1;
}
```

就是这么简单，nginx 就是这样把一个事件的超时时间戳作为 key 插入事件定时器的。

接下来，相信大家都有一个疑问：rbtree 的 key 是超时时间戳，利用 ngx_rbtree_min()可以取到时间戳最小的 rbtree_node，但是 nginx 是如何判断该 rbtree_node 究竟是那个事件呢？

事情的重点就是位于上述代码的 ngx_rbtree_insert 的第二个参数。

我们可以翻一下该函数的定义：

```
void ngx_rbtree_insert(ngx_thread_volatile ngx_rbtree_t *tree,
    ngx_rbtree_node_t *node);
```

然后，在看 ngx_event_add_timer()是如何插入一个 rbtree_node 的？相信看到这里大家就明白了吧。该函数通过&ev->timer 来获取 event 时间中的 ngx_rbtree_node_t 类型的地址。这样做的好处就是可以通过 ngx_rbtree_node_t 类型的变量的地址获取到 ngx_event_t 类型的变量的地址。这种方

式在 nginx 中用的比较多，包括内存池中的地址计算啦，配置文件的内存地址计算啦都用到了这种方式。

为了验证这种猜测，继续往下看 ngx_event_expire_timers()函数。

```
void
ngx_event_expire_timers(void)
{
 for ( ;; ) {
     root = ngx_event_timer_rbtree.root;
     node = ngx_rbtree_min(root, sentinel);
     if ((ngx_msec_int_t)(node->key - ngx-current_msce) < 0) {
         /* 通过红黑树获取超时的事件 */
        ev = (ngx_event_t *) ((char *) node - offsetof(ngx_event_t, timer));
         /* 为什么不用 node 节点呢？读者可以想一下为什么？ */
         ngx_rbtree_delete(&ngx_event_timer_rbtree, &ev->timer);

         /* 修改事件的标志位 */
         //……
         ev->handler(ev);
     }
 }
}
```

见证奇迹的时刻到了，看到了吗？nginx 中就是这样去获取超时事件的。

可见 nginx 对超时事件的处理流程：

STEP1：从定时器中取最小时间戳；

STEP2：判断是否超时；

STEP3：若超时，则把该事件的定时器从红黑树中删除；

STEP4：修改改事件的超时字段；

STEP5：调用改时间的超时处理函数去处理该事件。

6． 总结

关于这一章的总结，暂时还没想好怎么写。呵呵。先到这里吧，有时间了再补充。

7． 惊群现象

7.1. 什么是惊群现象

在 epoll 模型的多进程编程中，大家经常会讨论“惊群”，那么什么是“惊群”呢？

以 nginx 为例，在 nginx 进入 ngx_master_process_cycle()之前会调用 ngx_init_cycle()函数去初始化本次启动的所有事情，在 ngx_init_cycle()函数中会根据配置文件中监听的端口号创建 linux 的 socket 端口监听指定的端口。然后再 ngx_master_proces_cycle()中会 fork()多个 worker 进程，根据 nginx 的运行顺序，则 fork()的多个 worker 子进程也会同时监听指定的端口。

因为，有 N 个进程在同时监听 S 端口。

> Normally, the server process is put to sleep in the call to accept, waiting for a client connection to arrive and be accepted. A TCP connection uses what is called a three-way handshake to establish a connection. When this handshake completes, accept returns, and the return value from the function is a new descriptor (connfd) that is called the connected descriptor. This new descriptor is used for communication with the new client. A new descriptor is returned by accept for each client that connects to our server. —— ***UNIX Network Programming Volume 1.***

因此，当有新的客户请求到来时，由于 N 个进程监听 S 端口，因此操作系统会唤醒所有的 N 个 work 进程，并且这 N 个进程都会去执行 accept()函数，但是只有一个进程的 accept()会执行成功，并且返回 fd，其它进程的 accept()都将失败，并且返回 EAGIN 错误码。因此其它的 N-1 个进程都是无效唤醒，与其说是唤醒，倒不如说是惊醒的（Thundering Herd）。

关于 epoll 和 accetp 的关系可以参见：http://baike.baidu.com/view/1385104.htm。

7.2. 惊群对系统的影响

那么如果是单核 cpu，我们看一下情形是什么样的？

对于多核 cpu 呢，情形又是如下：

由此，可见，不管是单核还是多核 cpu，惊群的发生无疑会占用 cpu 周期，并且浪费系统资源，从而影响系统整体的效率。

此处，牵强附会的举一个不算恰当的例子吧。

于是乎，我们得到了一个惊群的大概的流程图。如下所示。

7.3. 如何避免惊群

从惊群的流程图我们可以猜想，在哪个阶段去避免这种惊群的发生呢？

nginx 中如何避免发生“惊群”的呢？

此处，也牵强附会的举个不算恰当的例子吧。

8． nginx 中的 core 模块的模块上下文

8.1. core 模块

```
static ngx_core_module_t  ngx_core_module_ctx = {
    ngx_string("core"),
    ngx_core_module_create_conf,
    ngx_core_module_init_conf
};
```

8.2. log 模块

```
static ngx_core_module_t  ngx_errlog_module_ctx = {
    ngx_string("errlog"),
    NULL,
    NULL
};
```

8.3. events 模块

```
static ngx_core_module_t  ngx_events_module_ctx = {
    ngx_string("events"),
    NULL,
    ngx_event_init_conf
};
```

8.4. regex 模块

```
static ngx_core_module_t  ngx_regex_module_ctx = {
    ngx_string("regex"),
    ngx_regex_create_conf,
    ngx_regex_init_conf
};
```

8.5. HTTP 模块

```
static ngx_core_module_t  ngx_http_module_ctx = {
    ngx_string("http"),
    NULL,
    NULL
};
```

8.6. openssl 模块

```
static ngx_core_module_t  ngx_openssl_module_ctx = {
    ngx_string("openssl"),
    ngx_openssl_create_conf,
    NULL
};
```

8.7. mail 模块

```
static ngx_core_module_t  ngx_mail_module_ctx = {
    ngx_string("mail"),
    NULL,
    NULL
};
```

8.8. 谷歌工具模块

```
static ngx_core_module_t  ngx_google_perftools_module_ctx = {
    ngx_string("google_perftools"),
    ngx_google_perftools_create_conf,
    NULL
};
```

9． nginx 中比较特别的模块

9.1. ngx_conf_module 模块

```
ngx_module_t  ngx_conf_module = {
    NGX_MODULE_V1,
    NULL,                                  /* module context */
    ngx_conf_commands,                      /* module directives */
    NGX_CONF_MODULE,                        /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    NULL,                                  /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    ngx_conf_flush_files,                   /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
};
```

9.2. ngx_regex_module 模块

```
ngx_module_t  ngx_regex_module = {
    NGX_MODULE_V1,
    &ngx_regex_module_ctx,               /* module context */
    ngx_regex_commands,                  /* module directives */
    NGX_CORE_MODULE,                     /* module type */
    NULL,                                /* init master */
    ngx_regex_module_init,               /* init module */
    NULL,                                /* init process */
    NULL,                                /* init thread */
    NULL,                                /* exit thread */
    NULL,                                /* exit process */
    NULL,                                /* exit master */
    NGX_MODULE_V1_PADDING
};
```

9.3. event_core_module 模块

```
ngx_module_t  ngx_event_core_module = {
    NGX_MODULE_V1,
    &ngx_event_core_module_ctx,          /* module context */
    ngx_event_core_commands,             /* module directives */
    NGX_EVENT_MODULE,                    /* module type */
    NULL,                                /* init master */
    ngx_event_module_init,               /* init module */
    ngx_event_process_init,              /* init process */
    NULL,                                /* init thread */
    NULL,                                /* exit thread */
    NULL,                                /* exit process */
    NULL,                                /* exit master */
    NGX_MODULE_V1_PADDING
};
```

9.4. openssl 模块

```
ngx_module_t  ngx_openssl_module = {
    NGX_MODULE_V1,
    &ngx_openssl_module_ctx,              /* module context */
    ngx_openssl_commands,                 /* module directives */
    NGX_CORE_MODULE,                      /* module type */
    NULL,                                 /* init master */
    NULL,                                 /* init module */
    NULL,                                 /* init process */
    NULL,                                 /* init thread */
    NULL,                                 /* exit thread */
    NULL,                                 /* exit process */
    ngx_openssl_exit,                     /* exit master */
    NGX_MODULE_V1_PADDING
};
```

9.5. userid 过滤模块

```
ngx_module_t  ngx_http_userid_filter_module = {
    NGX_MODULE_V1,
    &ngx_http_userid_filter_module_ctx,    /* module context */
    ngx_http_userid_commands,              /* module directives */
    NGX_HTTP_MODULE,                       /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    ngx_http_userid_init_worker,           /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    NULL,                                  /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
};
```

9.6. xslt 过滤模块

```
ngx_module_t  ngx_http_xslt_filter_module = {
    NGX_MODULE_V1,
    &ngx_http_xslt_filter_module_ctx,      /* module context */
    ngx_http_xslt_filter_commands,         /* module directives */
    NGX_HTTP_MODULE,                       /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    NULL,                                  /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    ngx_http_xslt_filter_exit,             /* exit process */
    ngx_http_xslt_filter_exit,             /* exit master */
    NGX_MODULE_V1_PADDING
};
```

9.7. perl 模块

```
ngx_module_t  ngx_http_perl_module = {
    NGX_MODULE_V1,
    &ngx_http_perl_module_ctx,             /* module context */
    ngx_http_perl_commands,                /* module directives */
    NGX_HTTP_MODULE,                       /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    ngx_http_perl_init_worker,             /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    NULL,                                  /* exit process */
    ngx_http_perl_exit,                    /* exit master */
    NGX_MODULE_V1_PADDING
};
```

9.8. spdy 模块

```
ngx_module_t  ngx_http_spdy_module = {
    NGX_MODULE_V1,
    &ngx_http_spdy_module_ctx,            /* module context */
    ngx_http_spdy_commands,               /* module directives */
    NGX_HTTP_MODULE,                      /* module type */
    NULL,                                 /* init master */
    ngx_http_spdy_module_init,            /* init module */
    NULL,                                 /* init process */
    NULL,                                 /* init thread */
    NULL,                                 /* exit thread */
    NULL,                                 /* exit process */
    NULL,                                 /* exit master */
    NGX_MODULE_V1_PADDING
};
```

9.9. 谷歌工具模块

```
ngx_module_t  ngx_google_perftools_module = {
    NGX_MODULE_V1,
    &ngx_google_perftools_module_ctx,     /* module context */
    ngx_google_perftools_commands,        /* module directives */
    NGX_CORE_MODULE,                      /* module type */
    NULL,                                 /* init master */
    NULL,                                 /* init module */
    ngx_google_perftools_worker,          /* init process */
    NULL,                                 /* init thread */
    NULL,                                 /* exit thread */
    NULL,                                 /* exit process */
    NULL,                                 /* exit master */
    NGX_MODULE_V1_PADDING
};
```

10. nginx 中 http 模块的运行机制

11． 使用 gdb 调试 nginx

调试 nginx 能够更方便的了解 nginx 的运行状态，从而达到熟悉 nginx 的目的。工欲善其事，必先利其器。君子性非异也，善假于物也。在 linux 下，可以选择使用 gdb 器调试 nginx。下面我们简单的介绍下如何使用 gdb 调试 nginx。

11.1. gdb 的基本命令

attach，run，continue，bt，frame，break，next，step，print，list，delete，info 等命令不再做详细介绍，具体使用可以是用 help attach 命令进行查看，或者直接百度一下。实际上，平时用到的也不多，边用边学，边学边用就好了。

11.2. 编译 nginx

NOTE：在编译 nginx 的时候，修改 auto/cc/conf 文件，增加-g 编译选项，从而增加 gdb 的调试信息。如下图所示：

```
ngx_include_opt="-I "
ngx_compile_opt="-c -g"
ngx_objout="-o "
ngx_binout="-o "
ngx_objext="o"
ngx_binext=
```

编译 nginx，生成可执行文件。

11.3. 利用 gdb 调试 nginx

（STEP 1）为了调试方便，我们把 nginx 的子进程数设置为 1 个，这样，nginx 在启动的时候就会生成 1 个 master 进程和一个 worker 进程，因此，我们的所有请求都会有这 1 个 worker 进程去处理。

利用/usr/sbin/lsof –i:8990 查看，发现 nginx 启动了 2 个进程。

```
COMMAND   PID    USER   FD   TYPE     DEVICE SIZE NODE NAME
nginx   13780 wangwei    8u  IPv4 5259310586       TCP *:8990 (LISTEN)
nginx   13781 wangwei    8u  IPv4 5259310586       TCP *:8990 (LISTEN)
```

cat logs/nginx.pid 发现主进程号为 13780，因此 nginx 的 worker 进程为 13781。

此处可以用 ps aux | grep nginx 来查看 nginx 的启动情况，由于我的机器上开启了 6 个 nginx，因此直接使用 lsof 命令去查看监听端口的进程信息。

（STEP 2）gcore 13781 生成 core 文件，如下图所示：

```
drwx------  2 wangwei wangwei    4096 Jun 30 12:02 client_body_temp
drwxrwxr-x  3 wangwei wangwei    4096 Sep 10 20:14 conf
-rw-rw-r--  1 wangwei wangwei 5429200 Sep 10 21:16 core.13781
drwx------  2 wangwei wangwei    4096 Jun 30 12:02 fastcgi_temp
drwxr-xr-x  2 wangwei wangwei    4096 Jun 30 13:08 html
-rwxr--r--  1 wangwei wangwei     811 Jun 30 17:37 load_nginx.sh
drwxrwxr-x  2 wangwei wangwei    4096 Sep 10 20:42 logs
drwx------  2 wangwei wangwei    4096 Jun 30 12:02 proxy_temp
drwxrwxr-x  2 wangwei wangwei    4096 Sep 10 20:02 sbin
drwx------  2 wangwei wangwei    4096 Jun 30 12:02 scgi_temp
-rw-rw-r--  1 wangwei wangwei     320 Aug 16 12:51 test.pl
drwx------  2 wangwei wangwei    4096 Jun 30 12:02 uwsgi_temp
```

（STEP 3）停止 nginx：./sbin/nginx –s stop

（STEP 4）gdb ./sbin/nginx core.13781 进入 gdb 调试模式。

```
(gdb) list
194     static u_char      *ngx_conf_params;
195     static char        *ngx_signal;
196
197
198     static char **ngx_os_environ;
199
200
201     int ngx_cdecl
202     main(int argc, char *const *argv)
203     {
(gdb)
```

（STEP 5）在需要设置断点的地方设置断点，break 的使用请查看相关资料。此处，以调试我写的扩展模块 ngx_http_mheader_filter_module.c 为例子进行调试。首先查看该模块的相关信息，如下图：

```
(gdb) list ngx_http_mheader_filter_module.c:185
180     ngx_http_mheader_filter(ngx_http_request_t *r)
181     {
182         ngx_http_mheader_loc_conf_t *conf = NULL;
183         conf    = ngx_http_get_module_loc_conf(r, ngx_http_mheader_filter_module);
184
185
186         if (conf == NULL) {
187             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
188                           "get the module configure failed. \n");
189             return ngx_http_next_header_filter(r);
(gdb)
190         }
191
192         u_char *server_name = get_server_name_from_conf(conf->str_server);
193
194         ngx_log_debug(NGX_LOG_WARN, r->connection->log, 0,
195                       "headers_out.server : %s", server_name);
196
197         if (r->headers_out.server == NULL) {
198             r->headers_out.server = ngx_list_push(&r->headers_out.headers);
199             if (r->headers_out.server ==NULL) {
```

为了在进入进入 ngx_http_mheader_filter 函数之后，查看该函数的执行情况，使用如下命令在该

模块的 182 行设置一个断点： break ngx_http_mheader_filter_module:182。

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004689b8 in ngx_http_mheader_filter
                                                   at /home/wangwei/unix_progr
```

（STEP 6）使用 run 命令在 gdb 中运行 nginx

```
(gdb) run
Starting program: /home/wangwei/nginx/sbin/nginx
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/tls/libthread_db.so.1".
[Inferior 1 (process 10249) exited normally]
(gdb)
```

此时利用 ps aux | grep nginx 命令应该可以发现，nginx 已经启动了。我的 nginx 的子进程为 10254。

（STEP 7）利用 attach 命令把 nginx 的 worker 进程 attach 到 gdb 中：

```
(gdb) info proc
process 10254
cmdline = 'nginx: worker process'
cwd = '/home/wangwei/nginx'
exe = '/home/wangwei/nginx/sbin/nginx'
```

（STEP 8）前端访问该 nginx，然后输入 continue，请求会在断点出暂停：

```
(gdb) continue
Continuing.

Breakpoint 1, ngx_http_mheader_filter (r=0x5cb440) at /home/wangwei/unix_programming/ngi
183         conf    = ngx_http_get_module_loc_conf(r, ngx_http_mheader_filter_module);
```

然后就可以利用 print 命令去查看此处的任意变量了。如下图所示。

```
(gdb) print *r->header_in
$2 = {
  pos = 0x5ca872 "",
  last = 0x5ca872 "",
  file_pos = 0,
  file_last = 0,
  start = 0x5ca3e0 "GET /a.php HTTP/1.1\r\nHost",
  end = 0x5cb3e0 "",
  tag = 0x0,
  file = 0x0,
  shadow = 0x0,
  temporary = 1,
  memory = 0,
  mmap = 0,
  recycled = 0,
  in_file = 0,
  flush = 0,
  sync = 0,
  last_buf = 0,
  last_in_chain = 0,
  last_shadow = 0,
  temp_file = 0,
  num = 0
}
```

如果想调试程序的每一步执行情况可以使用 next 或者 step 去单步执行，再次不做介绍了，感兴趣的可以去尝试。

调试完毕后，为了让请求完成，可以再次输入 continue 完成请求的处理。

还可以给进程发送各种信号，如 ctrl+c 发送 SIGINT 信号从而终端改进程。

（STEP 9）退出 gdb。

12． NGINX 中的 rewrite 模块相关内容

12.1. 相关结构体

1 ngx_http_script_regex_code_t;

```
typedef struct {
    ngx_http_script_code_pt     code;
    ngx_http_regex_t           *regex;
    ngx_array_t                *lengths;
    uintptr_t                   size;
    /* 改规则对应的状态 */
    uintptr_t                   status;
    uintptr_t                   next;

    uintptr_t                   test:1;
    uintptr_t                   negative_test:1;
    uintptr_t                   uri:1;
    uintptr_t                   args:1;

    /* add the r->args to the new arguments */
    /* 如果 replacement 以? 结尾，则置位该标记位 */
    uintptr_t                   add_args:1;
    /* 如果 replacement 已 http 或 https 或$scheme 开头或第四个参数为 redriect|perment，
       则置位改标记位 */
    uintptr_t                   redirect:1;
    /* 如果第四个参数为 break，则置位改标记位 */
    uintptr_t                   break_cycle:1;
    /* the rewrite regex */
    ngx_str_t                   name;
} ngx_http_script_regex_code_t;
```

2 ngx_http_script_compile_t

```
typedef struct {
    /* 读取到的 rewrite 配置文件 */
    ngx_conf_t                *cf;
    /* rewrite 中的 replacement 配置 */
    ngx_str_t                 *source;

    ngx_array_t              **flushes;
    ngx_array_t              **lengths;
    /* rewrite 的 ngx_http_script_code_pt 函数 */
    ngx_array_t              **values;
    /* replacement 中 nginx 变量的个数 */
    ngx_uint_t                 variables;
    ngx_uint_t                 ncaptures;
    /* replacement 中的$i 中的 i 在 captures_mask 中的第 i 位置 1*/
    ngx_uint_t                 captures_mask;
    ngx_uint_t                 size;
    /* 该配置的 ngx_http_script_regex_code_t 结构 */
    void                      *main;

    unsigned                   compile_args:1;
    unsigned                   complete_lengths:1;
    unsigned                   complete_values:1;
    unsigned                   zero:1;
    unsigned                   conf_prefix:1;
    unsigned                   root_prefix:1;
    /*  replacement 中重复使用$i */
    unsigned                   dup_capture:1;
    unsigned                   args:1;
} ngx_http_script_compile_t
```