Nginx源码分析：3张图看懂启动及进程工作原理

原创 2016-02-18 陈科 高可用架构

编者按：高可用架构分享及传播在架构领域具有典型意义的文章，本文由陈科在高可用架构群分享。转载请注明来自高可用架构公众号「ArchNotes」。

导读：很多工程师及架构师都希望了解及掌握高性能服务器开发，阅读优秀源代码是一种有效的方式，nginx 是业界知名的高性能 Web 服务器实现，如何有效的阅读及理解 nginx？本文用图解的方式帮助大家来更好的阅读及理解 nginx 关键环节的实现。

陈科，十年行业从业经验，曾在浙江电信、阿里巴巴、华为、五八同城任开发工程及架构师等职，目前负责河狸家后端架构和运维。博客地址：http://www.dumpcache.com/wiki/doku.php

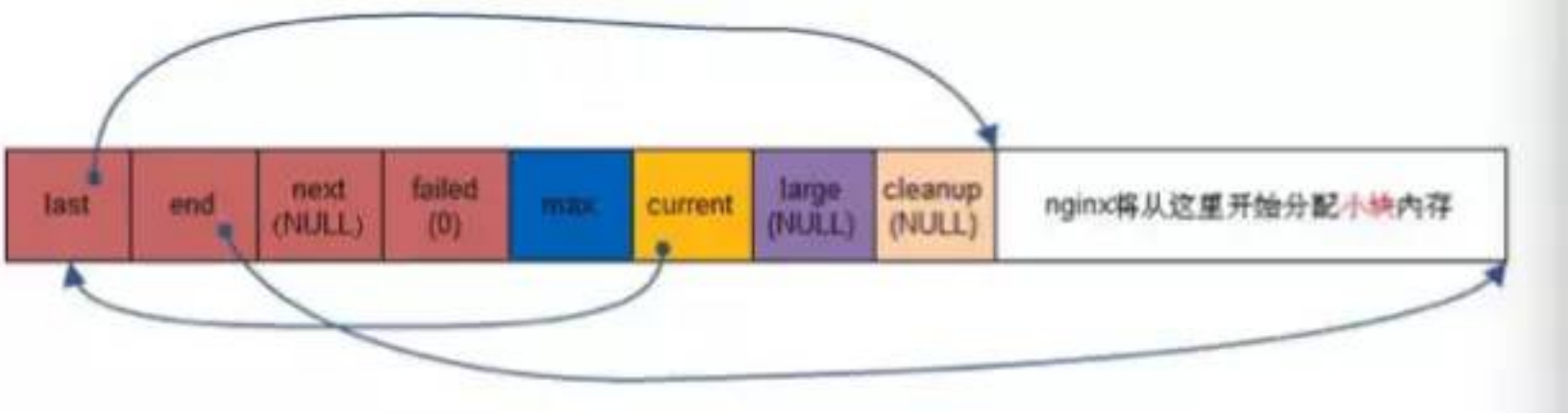
图一：nginx 启动及内存申请过程分析

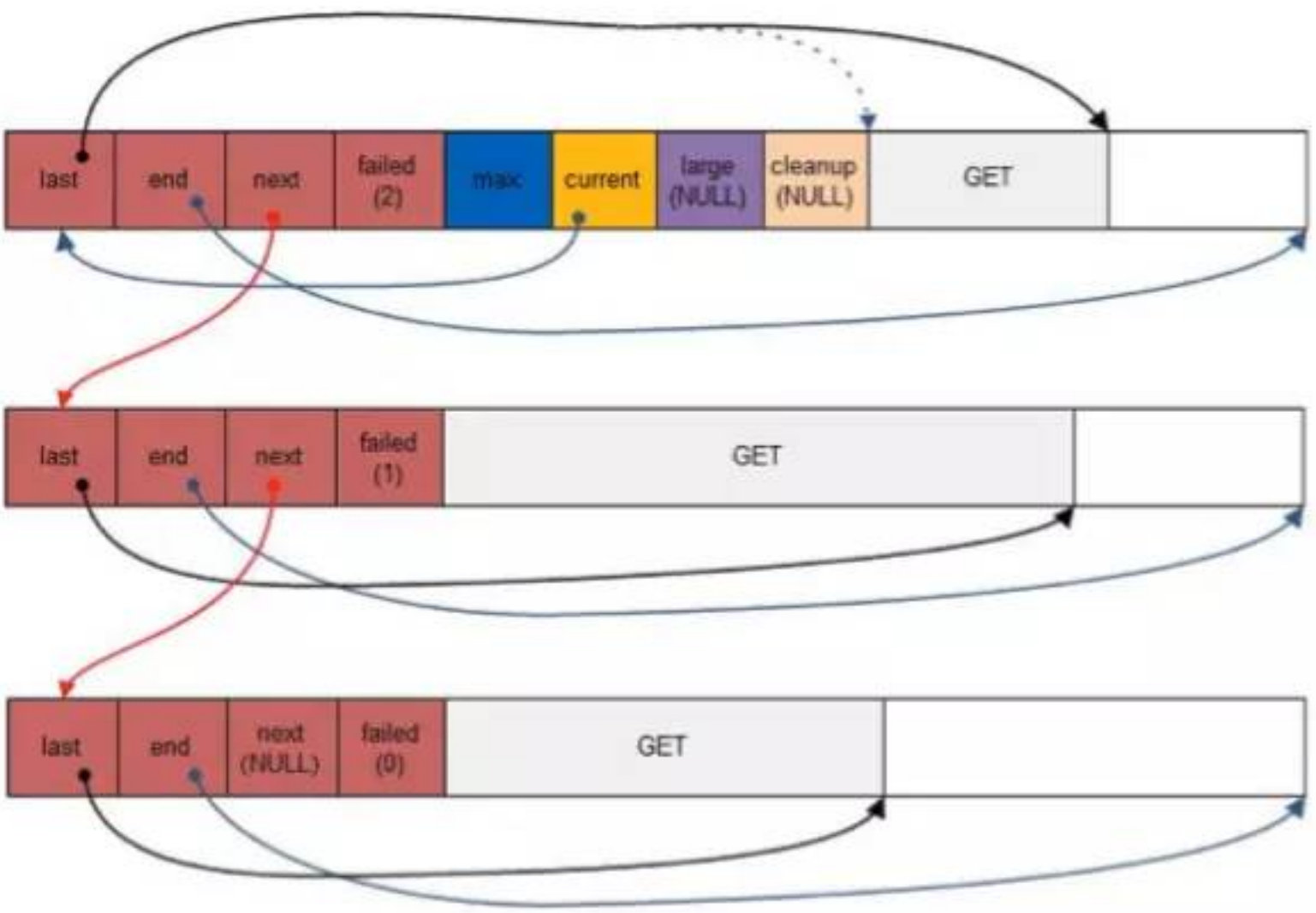
任何程序都离不开启动和配置解析。ngx 的代码离不开 ngx_cycle_s 和 ngx_pool_s 这两个核心数据结构，所以我们在启动之前先来分析下。

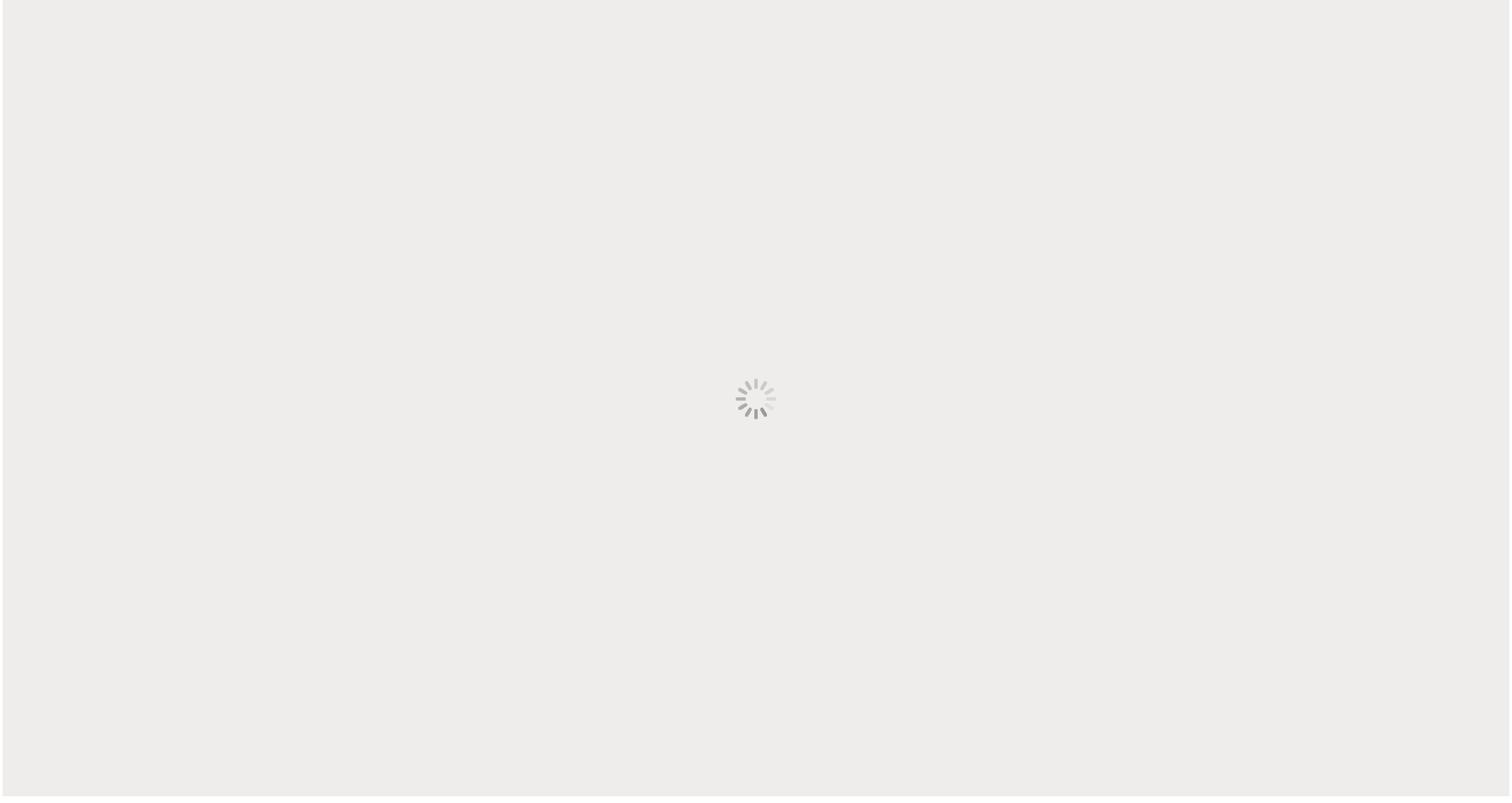
内存申请过程分为 3 步

1. 假如申请的内存小于当前块剩余的空间，则直接在当前块中分配。
2. 假如当前块空间不足，则调用 ngx_palloc_block 分配一个新块然后把新块链接到 d.next 中，然后分配数据。
3. 假如申请的大小大于当前块的最大值，则直接调用 ngx_palloc_large 分配一个大块，并且链接到 pool→large 链表中

内存分配过程图解如下


last
end
next
(NULL)
failed
(0)
max
current
large
(NULL)
cleanup
(NULL)
nginx将从这里开始分配小块内存


last
end
next
failed
(2)
max
current
large
(NULL)
cleanup
(NULL)
GET
last
end
next
failed
(1)
GET
last
end
next
(NULL)
failed
(0)
GET



（图片来自网络）

为了更好理解上面的图，可以参看文末附 2 的几个数据结构：ngx_pool_s 及 ngx_cycle_s。

知道了这两个核心数据结构之后，我们正式进入 main 函数，main 函数执行过程如下

- 调用 ngx_get_options() 解析命令参数；
- 调用 ngx_time_init() 初始化并更新时间，如全局变量ngx_cached_time；
- 调用 ngx_log_init() 初始化日志，如初始化全局变量 ngx_prefix，打开日志文件 ngx_log_file.fd；
- 清零全局变量 ngx_cycle，并为 ngx_cycle.pool 创建大小为 1024B 的内存池；

- 调用 ngx_save_argv() 保存命令行参数至全局变量 ngx_os_argv、ngx_argc、ngx_argv 中；
- 调用 ngx_process_options() 初始化 ngx_cycle 的 prefix, conf_prefix, conf_file, conf_param 等字段；
- 调用 ngx_os_init() 初始化系统相关变量，如内存页面大小 ngx_pagesize , ngx_cacheline_size , 最大连接数 ngx_max_sockets 等；
- 调用 ngx_crc32_table_init() 初始化 CRC 表 (后续的 CRC 校验通过查表进行，效率高)；
- 调用 ngx_add_inherited_sockets() 继承 sockets：
  - 解析环境变量 NGINX_VAR = "NGINX" 中的 sockets，并保存至 ngx_cycle.listening 数组；
  - 设置 ngx_inherited = 1；
  - 调用 ngx_set_inherited_sockets() 逐一对 ngx_cycle.listening 数组中的 sockets 进行设置；
- 初始化每个 module 的 index，并计算 ngx_max_module；
- 调用 ngx_init_cycle() 进行初始化；
  - 该初始化主要对 ngx_cycle 结构进行；
- 若有信号，则进入 ngx_signal_process() 处理；
- 调用 ngx_init_signals() 初始化信号；主要完成信号处理程序的注册；
- 若无继承 sockets，且设置了守护进程标识，则调用 ngx_daemon() 创建守护进程；
- 调用 ngx_create_pidfile() 创建进程记录文件；(非 NGX_PROCESS_MASTER = 1 进程，不创建该文件)
- 进入进程主循环；
  - 若为 NGX_PROCESS_SINGLE=1模式，则调用 ngx_single_process_cycle() 进入进程循环；
  - 否则为 master-worker 模式，调用 ngx_master_process_cycle() 进入进程循环；

在 main 函数执行过程中，有一个非常重要的函数 ngx_init_cycle，这个阶段做了什么呢？下面分析 ngx_init_cycle，初始化过程：

1. 更新 timezone 和 time
2. 创建内存池
3. 给 cycle 指针分配内存
4. 保存安装路径，配置文件，启动参数等
5. 初始化打开文件句柄
6. 初始化共享内存
7. 初始化连接队列
8. 保存 hostname
9. 调用各 NGX_CORE_MODULE 的 create_conf 方法
10. 解析配置文件
11. 调用各NGX_CORE_MODULE的init_conf方法
12. 打开新的文件句柄
13. 创建共享内存
14. 处理监听socket
15. 创建socket进行监听
16. 调用各模块的init_module

图二：master 进程工作原理及工作工程

以下过程都在ngx_master_process_cycle 函数中进行，启动过程：

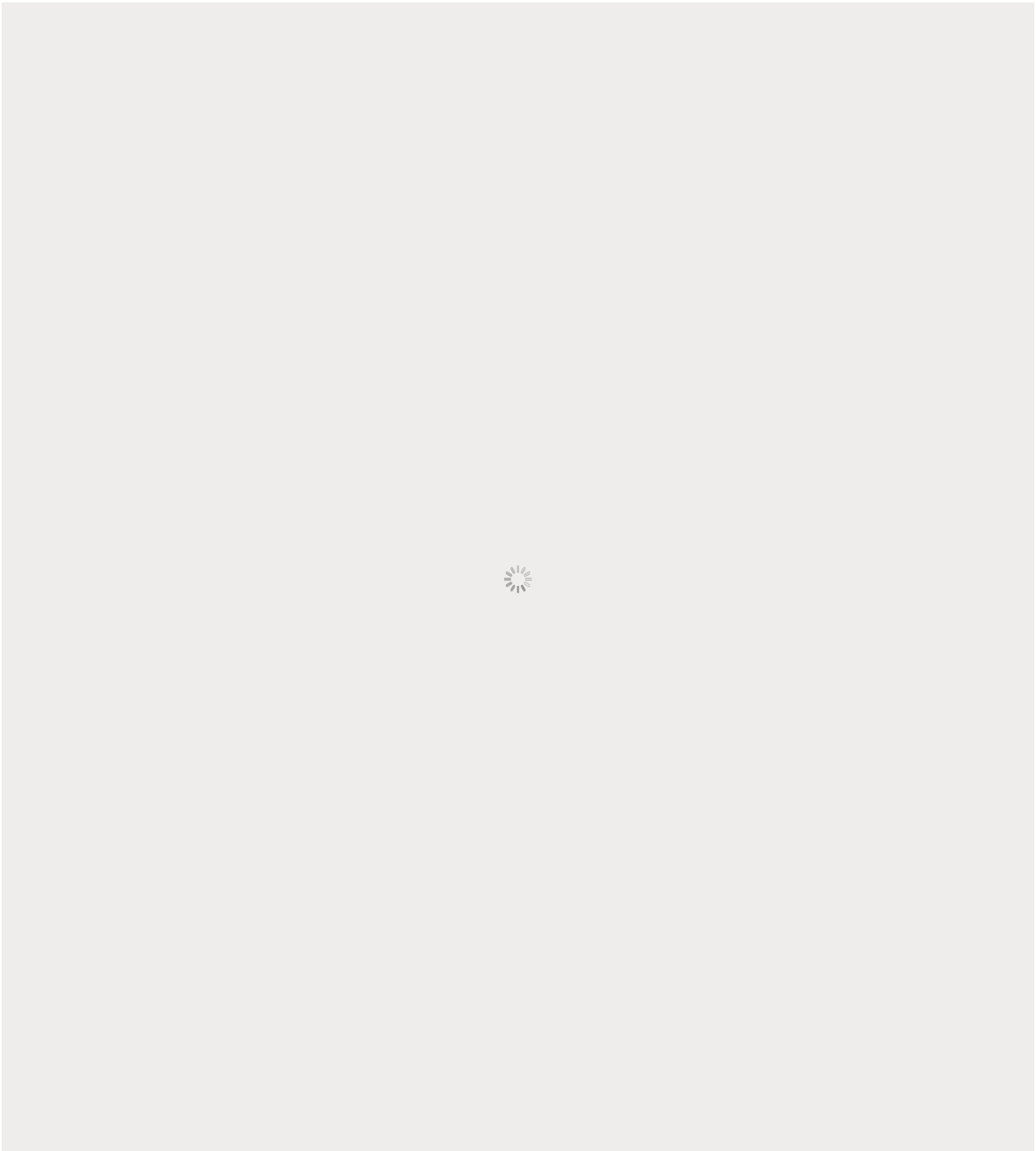
1. 暂时阻塞所有 ngx 需要处理的信号
2. 设置进程名称
3. 启动工作进程
4. 启动cache管理进程
5. 进入循环开始处理相关信号

master 进程工作过程

1. 设置 work 进程退出等待时间
2. 挂起，等待新的信号来临
3. 更新时间
4. 如果有 worker 进程因为 SIGCHLD 信号退出了，则重启 worker 进程
5. master 进程退出。如果所有 worker 进程都退出了，并且收到 SIGTERM 信号或 SIGINT 信号或 SIGQUIT 信号等，master 进程开始处理退出
6. 处理SIGTERM信号
7. 处理SIGQUIT信号，并且关闭socket
8. 处理SIGHUP信号
    a. 平滑升级，重启worker进程
    b. 不是平滑升级，需要重新读取配置

9. 处理重启 10处理SIGUSR1信号 重新打开所有文件 11处理SIGUSR2信号 热代码替换，执行新的程序 12处理SIGWINCH信号，不再处理任何请求

图三：worker 进程工作原理



启动通过执行 ngx_start_worker_processes 函数：

1. 先在 ngx_processes 数组中找坑位if (ngx_processes[s].pid == -1) {break;}
2. 进程相关结构初始化工作
    a. 创建管道 (socketpair)
    b. 设置管道为非阻塞模式
    c. 设置管道为异步模式
    d. 设置异步 I/O 的所有者
    e. 如果 exec 执行的时候本 fd 不传递给 exec 创建的进程
3. fork 创建子进程。创建成功后，子进程执行相关逻辑：proc(cycle, data)。
4. 设置 ngx_processes[s] 相关属性
5. 通知子进程新进程创建完毕 ngx_pass_open_channel(cycle, &ch);

接下来是 ngx_worker_process_cycle worker 进程逻辑

1. ngx_worker_process_init
    a. 初始化环境变量
    b. 设置进程优先级
    c. 设置文件句柄数量限制
    d. 设置 core_file 文件
    e. 用户组设置
    f. cpu 亲和度设置
    g. 设定工作目录
    h. 设置随机种子数
    i. 初始化监听状态
    j. 调用各模块的init_process方法进行初始化
    k. 关闭别人的fd[1],保留别人的fd[1]用于互相通信。自己的fd[1]接收master进程的消息。
    l. 监听channel读事件

2. 进程模式
    a. 处理管道信号。这个过程由 ngx_channel_handler 完成，这部分具体实现在管道事件中讲解。
3. 线程模式
    a. ngx_worker_thread_cycle 是一个线程的循环：死循环中除了处理退出信号。主要进行ngx_event_thread_process_posted工作,这块具体内容在后面讲事件模型的时候再展开。

4. 处理相关信号

master 和 worker 通信原理为：

Nginx 事件机制介绍

先看几个主要方法

- ngx_add_channel_event 主要是把事件注册到事件池中，并且添加事件 handler，具体要结合后面的事件机制来展开。
- ngx_write_channel 主要是将数据写入到 pipe 中：

  n = sendmsg(s, &msg, 0);

  Top of Form

  Bottom of Form
- ngx_read_channel 从 pipe 中读取数据：n = recvmsg(s, &msg, 0);

接下来分析事件模块工作流程

ngx_event模块结构

ngx_events_module 的数据结构如下：

```
ngx_module_t ngx_events_module = {
    NGX_MODULE_V1,
    &ngx_events_module_ctx, /* module context */
    ngx_events_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
};
```

ngx_event 模块初始化

```
static ngx_command_t ngx_events_commands[] = {
    {
        ngx_string("events") ,
        NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS ,
         ngx_events_block, 0, 0, NULL
     },
    ngx_null_command
};
```

通过 ngx_events_commands 数组可以知道，event 模块初始化函数为 ngx_events_block，该函数工作内容如下：

1. 创建模块 context 结构
2. 调用所有 NGX_EVENT_MODULE 模块的 create_conf
3. 解析 event 配置
4. 调用所有 NGX_EVENT_MODULE 模块的 init_conf

ngx_core_event模块初始化

ngx_core_event_module 是在 ngx_cycle_init 的时候初始化的：

```
for (i = 0; ngx_modules[i]; i++) {
    if (ngx_modules[i]->init_module) {
        if (ngx_modules[i]->init_module(cycle) != NGX_OK) { /* fatal */
            exit(1);
        }
    }
}
```

我们先来看下 ngx_core_event_module 的结构：

```
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
};
```

ngx_event_module_init 实现了初始化过程，该过程分以下几个步骤：

1. 连接数校验
2. 初始化互斥锁

事件进程初始化

在工作线程初始化的时候，将会调用 ngx_event_process_init:

```
for (i = 0; ngx_modules[i]; i++) {
    if (ngx_modules[i]->init_process)  {
        if (ngx_modules[i]->init_process(cycle) == NGX_ERROR) { /*fatal */
            exit(2);
        }
    }
}
```

ngx_event_process_init 该过程分以下几步：

1. 设置 ngx_accept_mutex_held
2. 初始化定时器
3. 初始化真正的事件引擎（linux 中为 epoll）
4. 初始化连接池
5. 添加 accept 事件

ngx_process_events_and_timers 事件处理开始工作

工作流程如下：

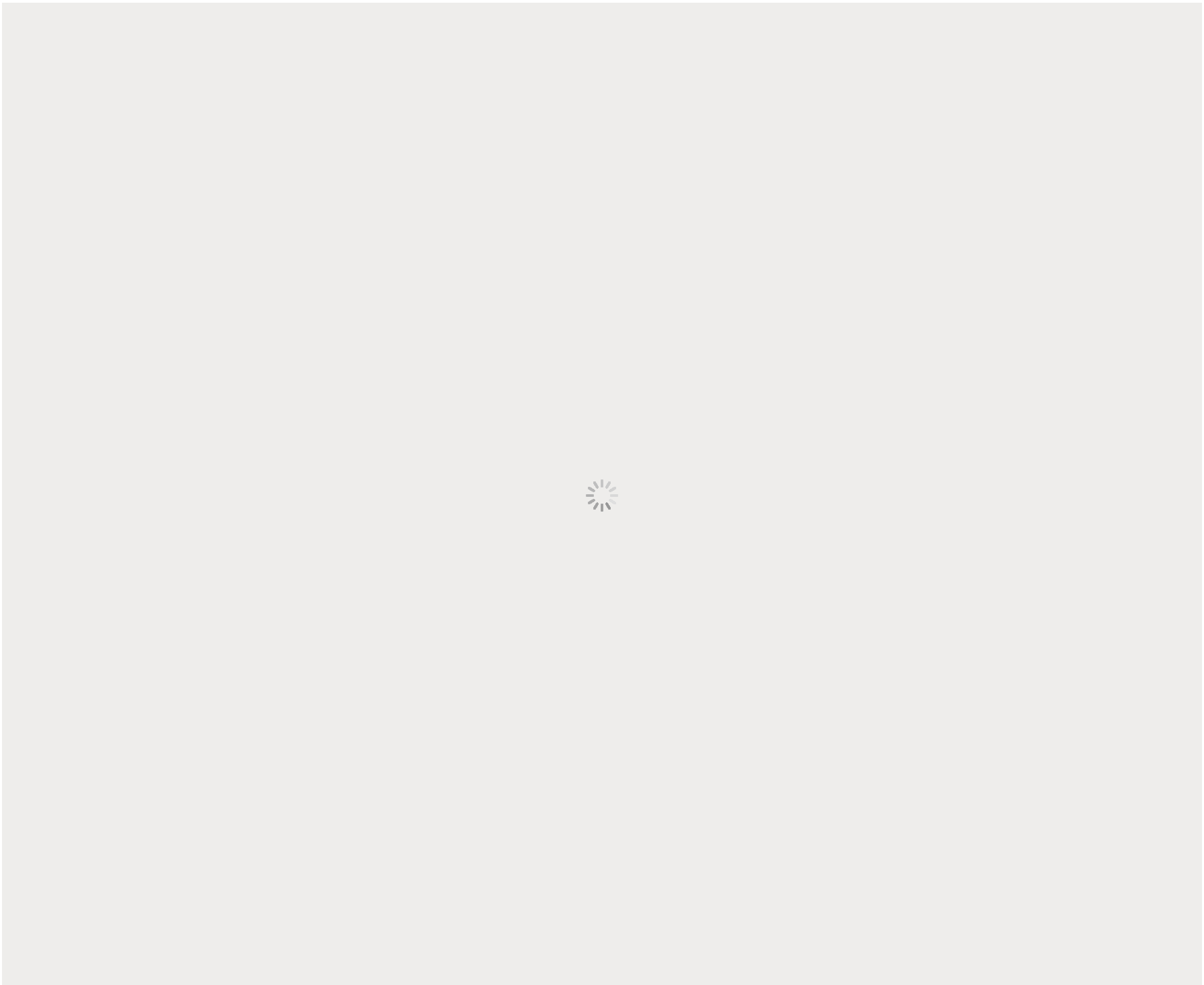
1. ngx_trylock_accept_mutex 当获取到标志位后才进行 accept 事件注册。

2. ngx_process_events 处理事件
3. 释放 accept_mutex 锁
4. 处理定时器事件
5. ngx_event_process_posted 处理 posted 队列的事件

ngx 定时器实现
ngx 的定时器利用了红黑树的实现

ngx 惊群处理
accept_mutex 解决了惊群问题，虽然linux的新内核已经解决了这个问题，但是ngx 是为了兼容。

整体原理图：



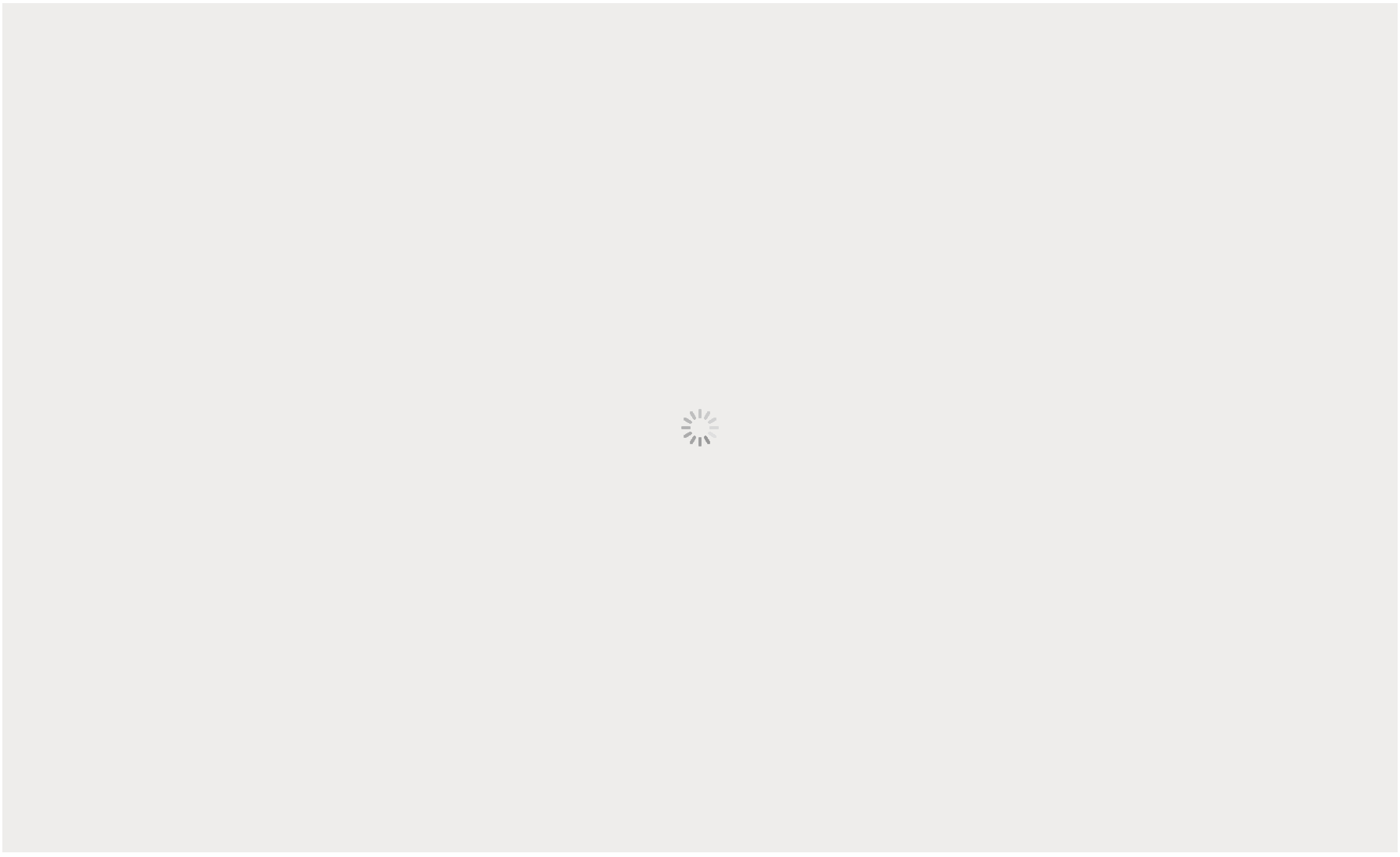
Nginx 配置解析

再补充一下配置解析，**Nginx 配置解析最大的亮点是用一个三级指针和 ctx 关联了起来，然后每个模块关注各自的配置专注解析和初始化就行了。**

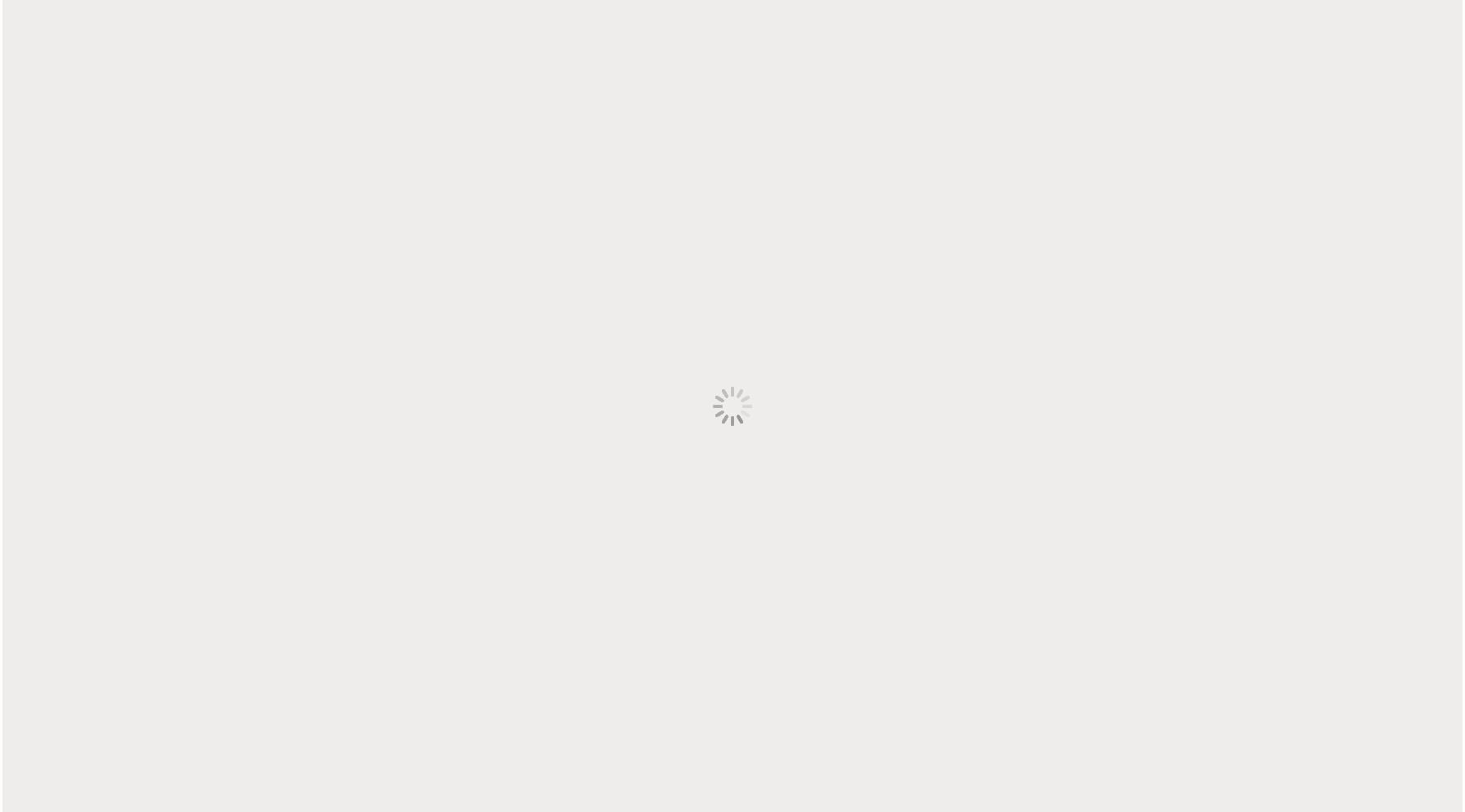
配置文件解析

ngx 在 main 函数执行的时候会调用 ngx_init_cycle，在这个过程中，会进行初始化的几个步骤：

- create_conf 针对 core_module 类型的模块，将会调用 create_conf 方法：



并且把根据模块号存入了 cycle→conf_ctx 中。这个过程主要是进行配置数据结构的初始化。以epoll模块为例：



- ngx_conf_parse 解析配置文件

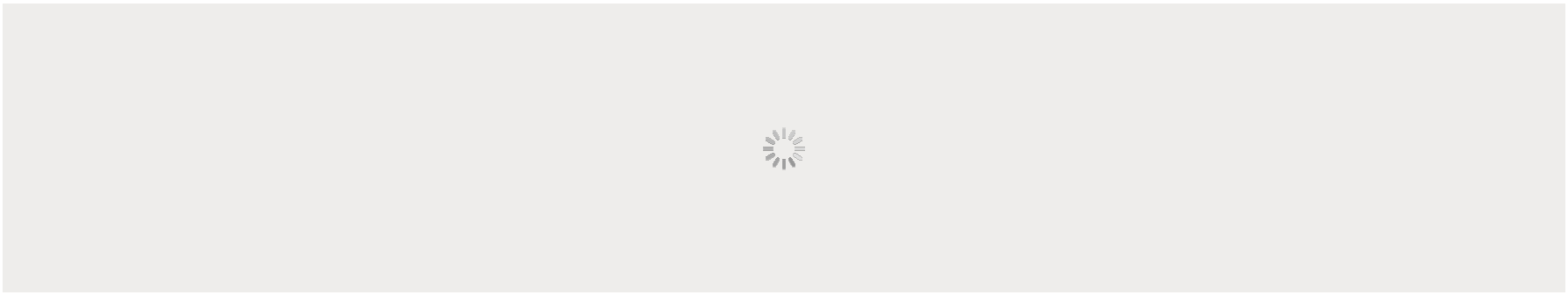
这个函数一共有以下几个过程：

- ngx_conf_read_token 这个过程主要进行配置配置的解析工作，解析完成的一个配置结构为：

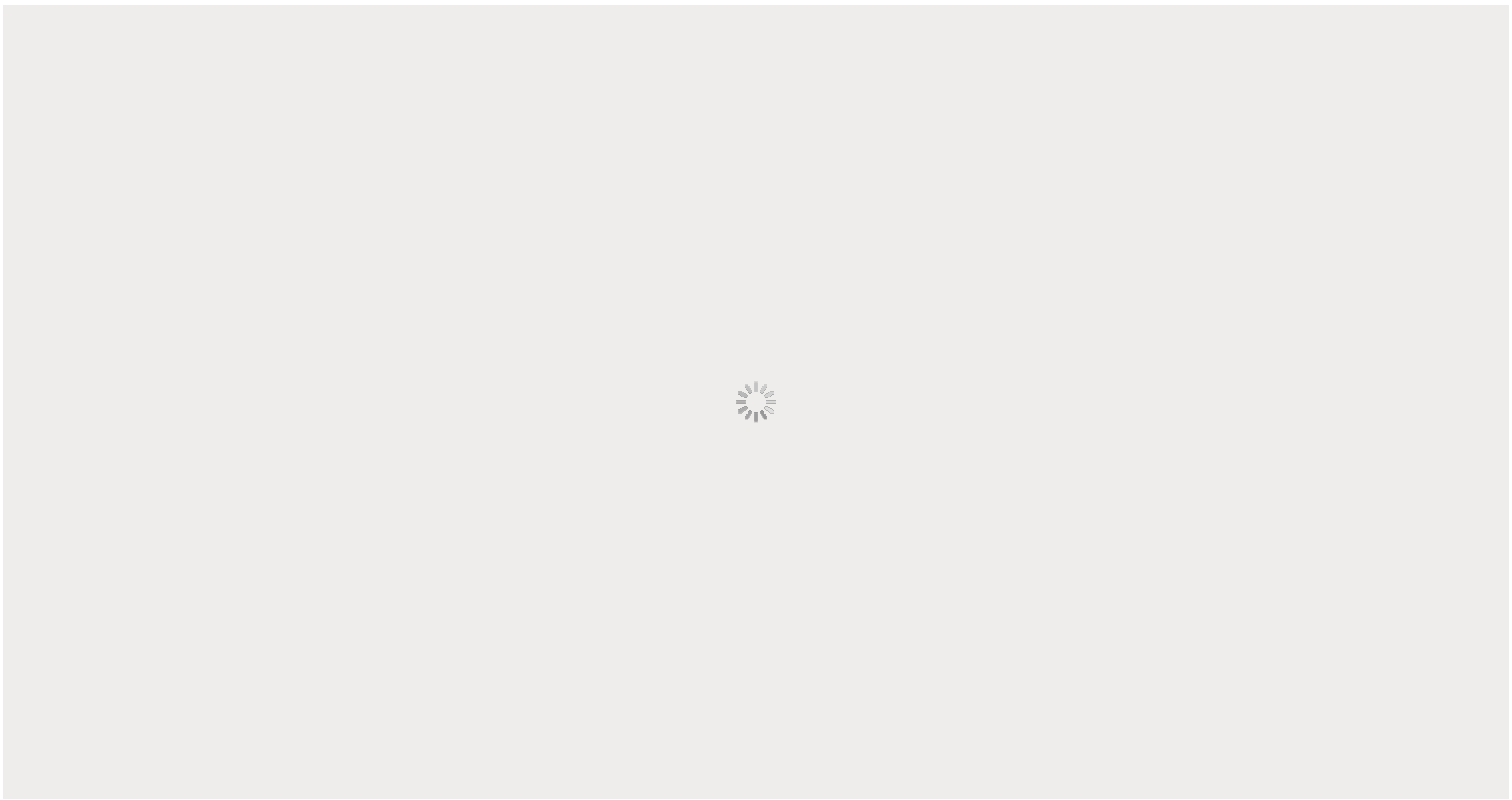
```
struct ngx_conf_s {
        char    *name;
        ngx_array_t     *args;
        ngx_cycle_t    *cycle;
        ngx_pool_t    *pool;
        ngx_pool_t    *temp_pool;
        ngx_conf_file_t     *conf_file;
        ngx_log_t    *log;
        void     *ctx;
        ngx_uint_t    module_type;
        ngx_uint_t    cmd_type;
         ngx_conf_handler_pt   handler;
        char    *handler_conf;
    };
```

- ngx_conf_handler 进行配置的处理
- cmd→set，以 ngx_http 模块为例

rv = ngx_conf_parse(cf, NULL) ; 在初始化完 http 的上下文之后，继续进行内部的解析逻辑。这样就会调用到 ngx_conf_handler 的下面部分逻辑：



- init_conf阶段



core 模块将会按照配置项的值在这个阶段进行初始化。ngx 的配置架构如下：

整体架构

serv_conf 结构

loc_conf 结构

附1：Nginx 主要数据结构

我们可以参考 ngx_connection_s 结构体，在 ngx_connection_s 中保存了链表的指针：ngx_queue_t queue

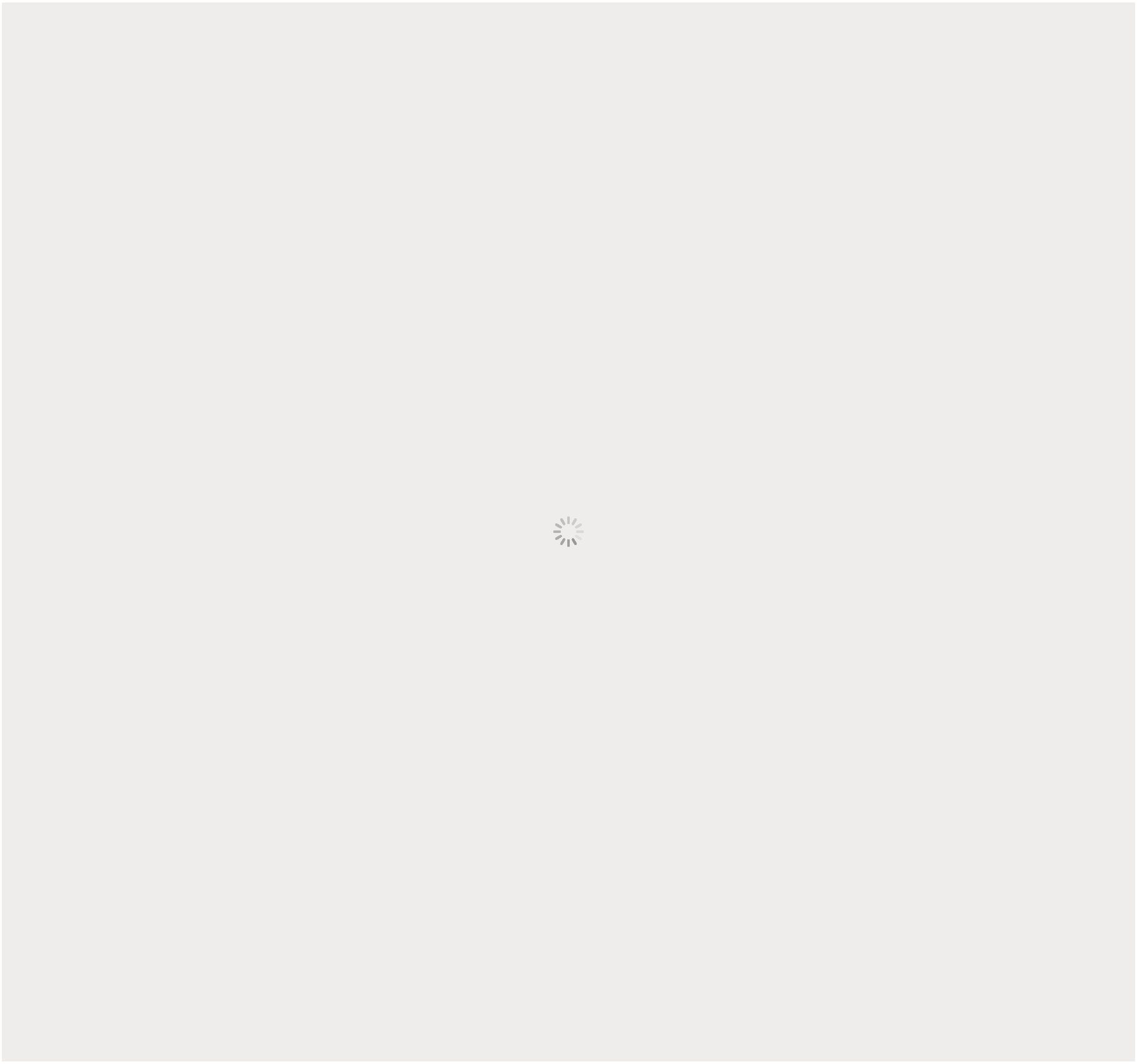
6 . ngx_hash_t

ngx 的 hash 表没有链表，如果找不到则往右继续查找空闲的 bucket。总的初始化 ngx_hash_init 流程即为：

1. 预估需要的桶数量

2. 搜索需要的桶数量
3. 分配桶内存
4. 初始化每一个 ngx_hash_elt_t

ngx 对内存非常扣，假设了 hash 表不会占用太多的数据和空间，所以采用了这样的方式。



附2：内存分配的数据结构

ngx_pool_s 是 ngx 的内存池，每个工作线程都会持有一个，我们来看它的结构：

```
struct ngx_pool_s {
    ngx_pool_data_t d ; // 数据块
```

```
    size_t max ; // 小块内存的最大值
    ngx_pool_t *current ; // 指向当前内存池
    ngx_chain_t *chain ;
    ngx_pool_large_t *large; // 分配大块内存用，即超过max的内存请求
    ngx_pool_cleanup_t *cleanup ; // 挂载一些内存池释放的时候，同时释放的资源
    ngx_log_t *log;
} ;
```

ngx_pool_data_t 数据结构：

```
typedef struct {
    u_char *last ; // 当前数据块分配结束位置
    u_char *end ; // 数据块结束位置
    ngx_pool_t *next ; // 链接到下一个内存池
    ngx_uint_t failed ; // 统计该内存池不能满足分配请求的次数
} ngx_pool_data_t ;
```

然后我们结合 ngx_palloc 方法来看一下内存池的分配原理：

```
void * ngx_palloc (ngx_pool_t *pool, size_t size) {
    u_char *m; ngx_pool_t *p ;
    if (size <= pool->max) {
        p = pool->current ;
        do {
            m = ngx_align_ptr(p->d.last, NGX_ALIGNMENT) ;
            if ((size_t) (p->d.end - m) >= size) {
                p->d.last = m + size ;
                return m ;
            }
            p = p->d.next ;
        } while (p) ;
        return ngx_palloc_block(pool, size) ;
    }
    return ngx_palloc_large(pool, size) ;
}
```

ngx_cycle_s 每个工作进程都会维护一个：

```
struct ngx_cycle_s {
```

```
    void    ****conf_ctx ;  // 配置上下文数组(含所有模块)
    ngx_pool_t    *pool ;  // 内存池
    ngx_log_t    *log ; // 日志
    ngx_log_t    new_log ;
    ngx_connection_t    **files ; // 连接文件
    ngx_connection_t    *free_connections ;  // 空闲连接
    ngx_uint_t   free_connection_n ; // 空闲连接个数
    ngx_queue_t    reusable_connections_queue ;  // 再利用连接队列
    ngx_array_t   listening ; // 监听数组
    ngx_array_t   pathes ; // 路径数组
    ngx_list_t    open_files ; // 打开文件链表
    ngx_list_t    shared_memory ; // 共享内存链表
    ngx_uint_t    connection_n ;  // 连接个数
    ngx_uint_t    iles_n ; // 打开文件个数
    ngx_connection_t    *connections ; // 连接
    ngx_event_t   *read_events ; // 读事件
    ngx_event_t   *write_events ; // 写事件
    ngx_cycle_t    *old_cycle;     //old cycle指针
    ngx_str_t    conf_file;     //配置文件
    ngx_str_t    conf_param;    //配置参数
    ngx_str_t    conf_prefix;   //配置前缀
    ngx_str_t    prefix;        //前缀
    ngx_str_t    lock_file;     //锁文件
    ngx_str_t    hostname;      //主机名
};
```

附3：Nginx 内存管理 & 内存对齐

内存的申请最终调用的是 malloc 函数，ngx_calloc 则在调用 ngx_alloc 后，使用 memset 来填0。假如自己开发NGX模块，不要直接使用 ngx_malloc/ngx_calloc，可以使用 ngx_palloc 否则还需要自己管理内存的释放。在 ngx_http_create_request 的时候会创建 request 级别的 pool：

```
pool = ngx_create_pool(cscf->request_pool_size, c->log) ;
if (pool == NULL) {
    return NULL;
}
```

```
r = ngx_pcalloc(pool, sizeof(ngx_http_request_t));
if (r == NULL) {
        ngx_destroy_pool(pool) ;
        return NULL ;
}

r->pool = pool ;
```

在 ngx_http_free_request 释放 request 的时候会调用 ngx_destroy_pool (pool) 释放连接。内存对齐，首先在创建 pool 的时候对齐：p = ngx_memalign(NGX_POOL_ALIGNMENT, size, log) 。ngx_memalign（返回基于一个指定 alignment 的大小为 size 的内存空间，且其地址为 alignment 的整数倍，alignment 为 2 的幂。）最终通过：posix_memalign 或 memalign 来申请。

> 数 据的对齐 (alignment) 是指数据的地址和由硬件条件决定的内存块大小之间的关系。一个变量的地址是它大小的倍数的时候，这就叫做自然对齐 (naturally aligned)。例如，对于一个 32bit 的变量，如果它的地址是 4 的倍数，-- 就是说，如果地址的低两位是 0，那么这就是自然对齐了。所以，如果一个类型的大小是 2n 个字节，那么它的地址中，至少低 n 位是 0。对齐的规则是由硬件引起 的。一些体系的计算机在数据对齐这方面有着很严格的要求。在一些系统上，一个不对齐的数据的载入可能会引起进程的陷入。在另外一些系统，对不对齐的数据的 访问是安全的，但却会引起性能的下降。在编写可移植的代码的时候，对齐的问题是必须避免的，所有的类型都该自然对齐。
>
> 预对齐内存的分配在大多数情况下，编译器和 C 库透明地帮你处理对齐问题。POSIX 标明了通过 malloc( ), calloc( ), 和 realloc( ) 返回的地址对于任何的C类型来说都是对齐的。在 Linux 中，这些函数返回的地址在 32 位系统是以 8 字节为边界对齐，在 64 位系统是以 16 字节为边界对齐 的。有时候，对于更大的边界，例如页面，程序员需要动态的对齐。虽然动机是多种多样的，但最常见的是直接块 I/O 的缓存的对齐或者其它的软件对硬件的交 互，因此，POSIX 1003.1d 提供一个叫做 posix_memalign( ) 的函数。
>
> 调用 posix_memalign( ) 成功时会返回 size 字节的动态内存，并且这块内存的地址是 alignment 的倍数。参数 alignment 必须是 2 的幂，还是 void 指针的大小的倍数。返回的内存块的地址放在了 memptr 里面，函数返回值是 0.
>
> 指针对齐：#define ngx_align_ptr(p, a) (u_char *) (((uintptr_t) (p) + ((uintptr_t) a - 1)) & ~ ((uintptr_t) a - 1))
>
> 例如：计算宏 ngx_align (1, 64) = 64，只要输入d < 64，则结果总是 64，如果输入 d = 65，则结果为 128，以此类推。

> 进行内存池管理的时候，对于小于64字节的内存，给分配64字节，使之总是cpu二级缓存读写行的大小倍数，从而有利cpu二级缓存取速度和效率。

由于公众号文章篇幅关系，以上就是陈科分享的 nginx 源码分析前半部分，关注本公众号可收到后半部分内容。

本文策划邓启明，编辑王杰，审校 Tim Yang，如需第一时间获取高可用架构分享文章，请关注以下公众号。转载请注明来自高可用架构 「ArchNotes」微信公众号及包含以下二维码。