# 启动流程

启动流程范围是从main函数开始执行到事务进程创建完成，开始处理网络事件、定时事件等。其中最关键的部分应该是ngx\_cycle\_t结构的创建。启动流程有两种方式，一种是第一次启动。另一种是nginx执行relaod操作，这种情况新的nginx进程需要从旧nginx进程中继承部分数据。

下面先介绍一下启动流程中设计到关键数据结构和关键函数，然后再说明一下两种启动方式的差异。

## 核心数据结构

nginx启动流程中涉及到很多数据结构，此处仅挑选核心的几个数据结构进行详细说明。

### ngx\_cycle\_s

struct ngx\_cycle\_s {

void \*\*\*\*conf\_ctx;

ngx\_pool\_t \*pool;

ngx\_module\_t \*\*modules;

ngx\_array\_t listening;

ngx\_connection\_t \*connections;

…

};

**conf\_ctx**：指针数组，数组的成员是指向另一个指针数组的指针。所以是4个\*。

**pool**：指向一个16KB大小的内存池，ngx\_cycle\_s结构也是从这个内存池中申请的。同时也负责给其他成员提供内存申请。

**modules**：

**connections**：

## 关键函数

### main函数

按函数流程来主要是做了几步操作：

1. 获取nginx启动时，通过命令行方式传入的各种参数。
2. 时间、正则表达式、日志的初始化。
3. 旧进程socket的继承，模块预初始化。
4. 调用ngx\_init\_cycle，创建ngx\_cycle\_t数据结构。这是启动过程中一个核心的函数。
5. 判断ngx\_signal变量，如果有值，则调用ngx\_signal\_process并返回。ngx\_signal变量是在ngx\_get\_options赋值的。如果启动nginx的时候设置了-s的参数，那么ngx\_signal就有值了，比如-s reload/stop/quit等，main函数在这个位置就返回了。
6. 初始化各种信号。
7. 根据ngx\_process变量判断是单进程模式ngx\_single\_process\_cycle还是父子进程模式ngx\_master\_process\_cycle。

### ngx\_preinit\_modules

出参：返回NGX\_OK表示成功。但只有这一个返回值，没有其他返回值。所以出参的设定感觉没有必要。也许是为了后续扩展性预留的。

各个模块预初始化。ngx\_module\_names实在编译的时候生成的，保存在objs/ngx\_module.c

ngx\_modules是管理所有模块的数组，这个预初始化函数就是根据编译生成的模块名称给每个模块进行编号，以及统计所有模块的数量ngx\_modules\_n，ngx\_max\_module。ngx\_max\_module后续会用于ngx\_cycle\_t中conf\_ctx所指向内存的分配。

### ngx\_cycle\_modules

入参cycle：在ngx\_init\_cycle创建的ngx\_cycle\_t数据结构。

出参：NGX\_OK表示成功，没有其他返回值。

为cycle->modules和cycle->modules\_n成员赋值，然后将ngx\_modules中的数据拷贝到cycle->moduels当中。

### ngx\_init\_modules

入参cycle：在ngx\_init\_cycle创建的ngx\_cycle\_t数据结构。

出参：NGX\_OK表示成功，NGX\_ERROR表示失败。

遍历cycle->modules数组，调用每个module定义的init\_module方法。如果init\_module没有执行成功，则返回NGX\_ERROR。

### ngx\_init\_cycle

入参ngx\_cycle\_t \*old\_cycle：从变量名理解就是旧nginx进程的ngx\_cycle\_t数据。实际是main函数中定义的一个局部变量init\_cycle。在传入ngx\_init\_cycle之前存储了本次启动所设置的参数、选项等信息。

出参ngx\_cycle\_t \*：函数正常处理完成返回的是初始化完成的ngx\_cycle\_t数据结构。如果函数异常则返回NULL。

函数流程如下：

1. 函数入口出关于时间相关的处理在《Nginx时间管理》文档中说明。
2. 创建一块16KB大小的内存池，再从该内存池中分配ngx\_cycle\_t数据结构。该内存池也作为ngx\_cycle\_t中其他成员内存分配的来源。
3. 创建cycle->listening记录监听端口的数组。如果是旧进程reload操作，那么数组的大小保持和旧进程一致。如果是新启动的进程，那么初始大小为10。
4. 创建cycle->conf\_ctx，看这个变量的定义就是一个保存指针数组的指针数组。

cycle->conf\_ctx = ngx\_pcalloc(pool, ngx\_max\_module \* sizeof(void \*));

第一层就是一个指针数组，数组成员是void \*类型的指针，个数为ngx\_max\_module。数组每个成员的赋值则是在多个不同的代码段。一部分是在接下来的for循环中：

if (module->create\_conf) {

rv = module->create\_conf(cycle);

…

cycle->conf\_ctx[cycle->module[i]->index] = rv;

}

先是通过ngx\_cycle\_modules初始化了cycle->modules成员，然后遍历所有类型为NGX\_CORE\_MODULE的module，如果module定义了create\_conf方法，则调用创建module的ctx数据结构。然后将ctx根据其module的编号，存到数组对应的位置。

但是并非所有的module都定义了create\_conf函数。比如ngx\_http\_module就没有定义create\_conf方法，该模块的ctx结构则是在配置解析调用ngx\_http\_block函数中创建的。

而有些module之所以要定义create\_conf方法，是因为该模块ctx中部分数据结构要在配置解析之前就完成初始化操作。比如ngx\_thread\_pool\_module。

还有一点需要注意的是，并非所有的conf\_ctx成员指针都是指向一个指针数组的。每个conf\_ctx成员所指向的对象是由各个模块自行决定的。比如ngx\_http\_module是指向ngx\_http\_conf\_ctx\_t，而ngx\_event\_module则是指向一个指针数组。

1. 调用ngx\_conf\_param和ngx\_conf\_parse进行配置解析，配置解析相关处理在《Nginx配置解析》文档中说明。
2. 对所有NGX\_CORE\_MOUDLE模块调用init\_conf方法。
3. 共享内存即其他内容的初始化（待完善）。
4. 调用ngx\_open\_listening\_sockets根据配置开启所有需要监听的端口。
5. 调用ngx\_init\_modules对所有模块进行初始化。
6. 关闭所有旧进程使用但新进程没有使用的监听端口。
7. 创建ngx\_cleaner\_event定时任务，每30秒执行一次。
8. 返回创建的ngx\_cycle\_t数据结构。

### ngx\_single\_process\_cycle

### ngx\_master\_process\_cycle