**Ngnix 架构解析**

**一 、nginx配置文件解析**

* 1. **块配置项**

块配置项由一个块配置项名和一对大括号组成如：

http {

upstream backend {

server 127.0.0.1:8080;

}

gzip on;

server {

…

location /webstatic {

gzip off;

}

}

}

块配置项之后是否如“location/webstatic{...}”那样在后面加上参数，取决于解析这个块配置项的模块，不能一概而论,块配置项可以嵌套。内层块直接继承外层块,当内外层块中的配置发生冲突时，究竟是以内层块还是外层块的配置为准，取决于解析这个配置项的模块(上列中的gzip off/on)

* 1. **配置项单位**

空间大小单位：

·K或者k千字节（KiloByte，KB）。

·M或者m兆字节（MegaByte，MB）

client\_max\_body\_size 64M;

定时间单位包括：

·ms（毫秒），s（秒），m（分钟），h（小时），d（天），w（周，包含7天），

M（月，包含30天），y（年，包含365天）。

client\_body\_timeout 2m;

* 1. **基本配置**
     1. 用于调试进程和定位问题的配置项

1. 是否以守护进程方式运行Nginx

语法： daemon on|off;

默认： daemon on;

1. 是否以master/worker方式工作

语法： master\_process on|off;

默认： master\_process on;

（3）error日志的设置

语法： error\_logpathfile level;

默认： error\_log logs/error.log error;

level是日志的输出级别，取值范围是debug、info、notice、warn、error、crit、alert、

emerg，从左至右级别依次增大。当设定为一个级别时，大于或等于该级别的日志都会被输

出到pathfile文件中，小于该级别的日志则不会输出。例如，当设定为error级别时，error、

crit、alert、emerg级别的日志都会输出

注意 :如果日志级别设定到debug，必须在configure时加入--with-debug配置项

（4）是否处理几个特殊的调试点

语法： debug\_points[stop|abort]

（5）仅对指定的客户端输出debug级别的日志

语法： debug\_connection[IP|CIDR]

（6）限制coredump核心转储文件的大小

语法： worker\_rlimit\_core size;

（7）指定coredump文件生成目录

语法： working\_directory path;

* + 1. 正常运行的配置项

（1）定义环境变量

语法： env VAR|VAR=VALUE

这个配置项可以让用户直接设置操作系统上的环境变量

（2）嵌入其他配置文件

语法： include pathfile;

（3）pid文件的路径

语法： pid path/file;

默认： pid logs/nginx.pid;

（4）Nginx worker进程运行的用户及用户组

语法： user username[groupname];

默认： user nobody nobody;

（5）指定Nginx worker进程可以打开的最大句柄描述符个数

语法： worker\_rlimit\_nofile limit;

设置一个worker进程可以打开的最大文件句柄数。

（6）限制信号队列

语法： worker\_rlimit\_sigpending limit;

设置每个用户发往Nginx的信号队列的大小。也就是说，当某个用户的信号队列满 了，这个用户再发送的信号量会被丢掉

* + 1. 优化性能的配置项

（1）Nginx worker进程个数

语法： worker\_processes number;

默认： worker\_processes 1;

每个worker进程都是单线程的进程，它们会调用各个模块以实现多种多样的功能。如果这些模块确认不会出现阻塞式的调用，那么，有多少CPU内核就应该配置多少个进程；反之，如果有可能出现阻塞式调用，那么需要配置稍多一些的worker进程

（2）绑定Nginx worker进程到指定的CPU内核

语法： worker\_cpu\_affinity cpumask[cpumask...]

注意：worker\_cpu\_affinity配置仅对Linux操作系统有效。Linux操作系统使用

sched\_setaffinity()系统调用实现这个功能

（3）SSL硬件加速

语法： ssl\_engine device；

（4）系统调用gettimeofday的执行频率

语法： timer\_resolution t;

默认情况下，每次内核的事件调用（如epoll、select、poll、kqueue等）返回时，都会执行一次gettimeofday，实现用内核的时钟来更新Nginx中的缓存时钟。在早期的Linux内核中，gettimeofday的执行代价不小，因为中间有一次内核态到用户态的内存复制。当需要降低gettimeofday的调用频率时，可以使用timer\_resolution配置。例如，“timer\_resolution

100ms；”表示至少每100ms才调用一次gettimeofday。但在目前的大多数内核中，如x86-64体系架构，gettimeofday只是一次vsyscall，仅仅对共享内存页中的数据做访问，并不是通常的系统调用，代价并不大，一般不必使用这个配置。而且，如果希望日志文件中每行打印的时间更准确，也可以使用它。

（5）Nginx worker进程优先级设置

语法： worker\_priority nice;

默认： worker\_priority 0;

优先级由静态优先级和内核根据进程执行情况所做的动态调整（目前只有±5的调整）共同决定。nice值是进程的静态优先级，它的取值范围是–20~+19，–20是最高优先级，+19是最低优先级。因此，如果用户希望Nginx占有更多的系统资源，那么可以把nice值配置得更小一些，但不建议比内核进程的nice值（通常为–5）还要小

* + 1. 事件类配置项

（1）是否打开accept锁

语法： accept\_mutex[on|off]

默认： accept\_mutext on;

accept\_mutex是Nginx的负载均衡锁，accept\_mutex这把锁可以让多个worker进程轮流地、序列化地与新的客户端建立TCP连接。当某一个worker进程建立的连接数量达到

worker\_connections配置的最大连接数的7/8时，会大大地减小该worker进程试图建立新TCP连接的机会，以此实现所有worker进程之上处理的客户端请求数尽量接近。accept锁默认是打开的，如果关闭它，那么建立TCP连接的耗时会更短，但worker进程之间的负载会非常不均衡，因此不建议关闭它。

（2）lock文件的路径

语法： lock\_file path/file;

默认： lock\_file logs/nginx.lock;

accept锁可能需要这个lock文件，如果accept锁关闭，lock\_file配置完全不生效。如果打

开了accept锁，并且由于编译程序、操作系统架构等因素导致Nginx不支持原子锁，这时才会用文件锁实现accept锁

注意：在基于i386、AMD64、Sparc64、PPC64体系架构的操作系统上，若使用

GCC、Intel C++、SunPro C++编译器来编译Nginx，则可以肯定这时的Nginx是支持原子锁

的，因为Nginx会利用CPU的特性并用汇编语言来实现它

（3）使用accept锁后到真正建立连接之间的延迟时间

语法： accept\_mutex\_delay Nms;

默认： accept\_mutex\_delay 500ms;

（4）批量建立新连接

语法： multi\_accept[on|off];

默认： multi\_accept off;

当事件模型通知有新连接时，尽可能地对本次调度中客户端发起的所有TCP请求都建立

连接。

（5）选择事件模型

语法： use[kqueue|rtsig|epoll|/dev/poll|select|poll|eventport];

默认： Nginx会自动使用最适合的事件模型。

（6）每个worker的最大连接数

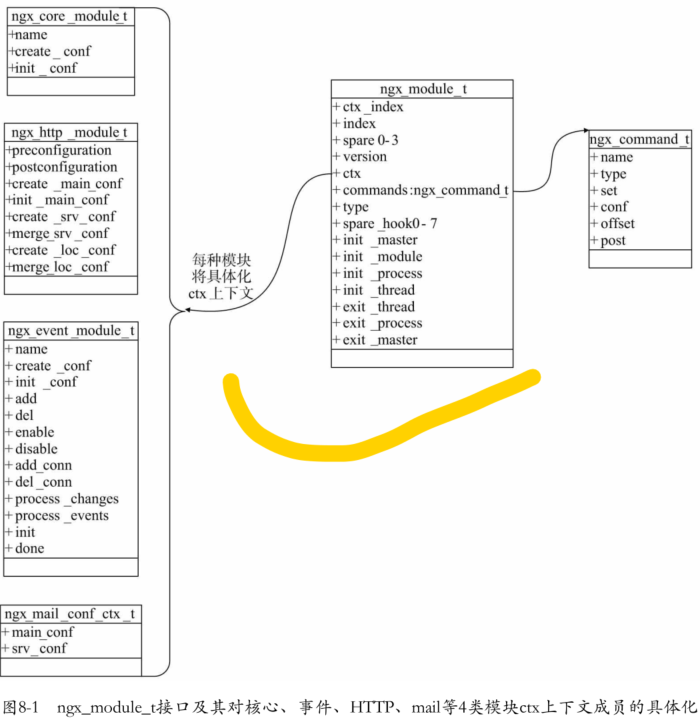
语法： worker\_connections number;

二 nginx 基础架构

2.1.1 Nginx的架构设计

1. 高度抽象的模块接口

所有的模块都遵循着同样的ngx\_module\_t接口设计规范



1. 模块接口非常简单，具有很高的灵活性

模块的基本接口ngx\_module\_t足够简单，只涉及模块的初始化、退出以及对配置项的处理，这同时也带来了足够的灵活性，使得Nginx比较简单地实现了动态可修改性

nginx\_module\_t 结构体如下

struct ngx\_module\_s {

ngx\_uint\_t ctx\_index;

ngx\_uint\_t index;

ngx\_uint\_t spare0;

ngx\_uint\_t spare1;

ngx\_uint\_t spare2;

ngx\_uint\_t spare3;

ngx\_uint\_t version;

void \*ctx;

ngx\_command\_t \*commands;

ngx\_uint\_t type;

ngx\_int\_t (\*init\_master)(ngx\_log\_t \*log);

ngx\_int\_t (\*init\_module)(ngx\_cycle\_t \*cycle);

ngx\_int\_t (\*init\_process)(ngx\_cycle\_t \*cycle);

ngx\_int\_t (\*init\_thread)(ngx\_cycle\_t \*cycle);

void (\*exit\_thread)(ngx\_cycle\_t \*cycle);

void (\*exit\_process)(ngx\_cycle\_t \*cycle);

void (\*exit\_master)(ngx\_cycle\_t \*cycle);

uintptr\_t spare\_hook0;

uintptr\_t spare\_hook1;

uintptr\_t spare\_hook2;

uintptr\_t spare\_hook3;

uintptr\_t spare\_hook4;

uintptr\_t spare\_hook5;

uintptr\_t spare\_hook6;

uintptr\_t spare\_hook7;

};

1. 配置模块的设计

配置类型模块是唯一一种只有1个模块的模块类型。配置模块的类型叫做NGX\_CONF\_MODULE，它仅有的模块叫做ngx\_conf\_module，这是Nginx最底层的模块，它指导着所有模块以配置项为核心来提供功能。因此，它是其他所有模块的基础

1. 核心模块接口的简单化

Nginx还定义了一种基础类型的模块：核心模块，它的模块类型叫做

NGX\_CORE\_MODULE。目前官方的核心类型模块中共有6个具体模块，分别是

ngx\_core\_module、ngx\_errlog\_module、ngx\_events\_module、ngx\_openssl\_module、ngx\_http\_module、ngx\_mail\_module模块

核心模块的接口非常简单，它将ctx上下文进一步实例化为ngx\_core\_module\_t结构体

typedef struct {

ngx\_str\_t name;

void \*(\*create\_conf)(ngx\_cycle\_t \*cycle);

#创建存储配置项的数据结构，在读取nginx.conf配置文件时，会根据模块#中的ngx\_command\_t把解析出的配置项存放在这个数据结构中

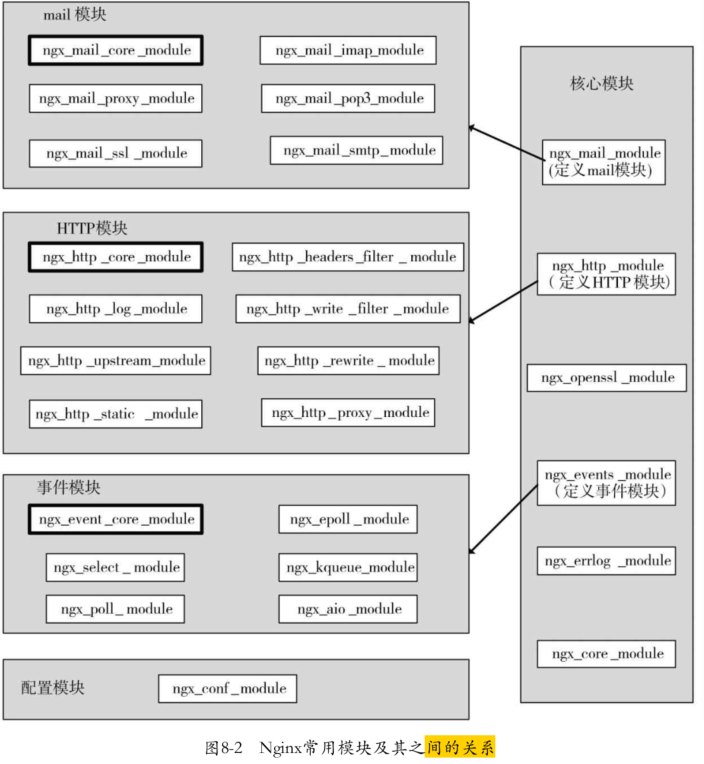
char \*(\*init\_conf)(ngx\_cycle\_t \*cycle, void \*conf);

#在解析完配置文件后，使用解析出的配置项初始化核心模块

} ngx\_core\_module\_t;

1. 多层次、多类别的模块设计

官方Nginx共有五大类型的模块：核心模块、配置模块、事件模块、HTTP模块、mail模块。虽然它们都具备相同的ngx\_module\_t接口，但在请求处理流程中的层次并不相同。就如同上面介绍过的核心模块一样，事件模块、HTTP模块、mail模块都会再次具体化ngx\_module\_t接口



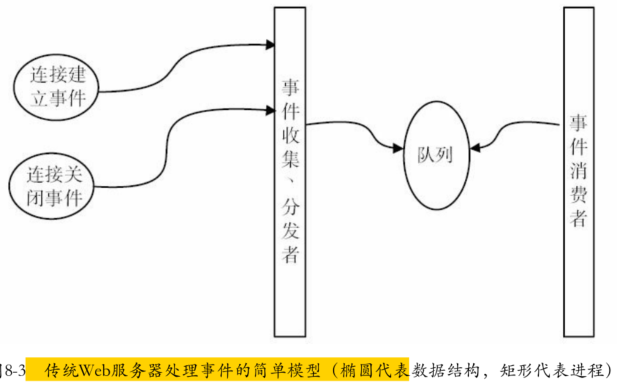
2.2.2 事件驱动架构

所谓事件驱动架构，简单来说，就是由一些事件发生源来产生事件，由一个或者多个事

件收集器来收集、分发事件，然后许多事件处理器会注册自己感兴趣的事件，同时会“消

费”这些事件

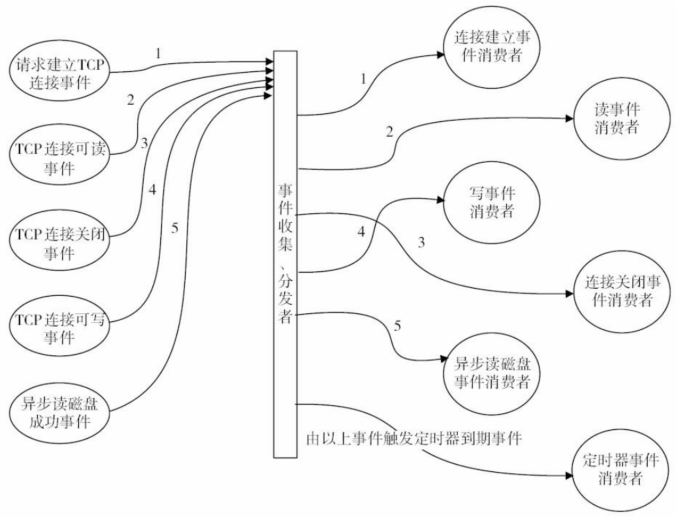
Nginx采用完全的事件驱动架构来处理业务，传统的web服务器往往把进程或者线程作为事件消费者，当一个请求产生的事件被该进程处理时，直到这个请求处理结束时进程资源都将被这一个请求所占用如下图所示



Nginx则不然，它不会使用进程或线程来作为事件消费者，所谓的事件消费者只能是某

个模块（在这里没有进程的概念）。只有事件收集、分发器才有资格占用进程资源，它们会

在分发某个事件时调用事件消费模块使用当前占用的进程资源，如下图所示



2.2.3 请求的多阶段异步处理

这里所讲的多阶段异步处理请求与事件驱动架构是密切相关的，换句话说，请求的多阶

段异步处理只能基于事件驱动架构实现。什么意思呢？就是把一个请求的处理过程按照事件

的触发方式划分为多个阶段，每个阶段都可以由事件收集、分发器来触发。

阻塞代码段上按照下面4种方式来划分请求阶段

1. 将阻塞进程的方法按照相关的触发事件分解为两个阶段。

大部分情况下，一个阻塞进程的方法调用时可以划分为两个阶段：阻塞方法改为非阻塞方法调用，这个调用非阻塞方法并将进程归还给事件分发器的阶段就是第一阶段；增加新的处理阶段（第二阶段）用于处理非阻塞方法最终返回的结果，这里的结果返回事件就是第二阶段的触发事件

例如：在使用send调用发送数据时，我们需要把send调用分解为两个阶段：发送且不等待结果阶段、send结果返回阶段。因此，可以使用非阻塞socket句柄，这样调用send发送数据后，进程是不会进入休眠的，这就是发送且不等待结果阶段；再把socket句柄加入到事件收集器中就可以等待相应的事件触发下一个阶段，send发送的数据被对方收到后这个事件就会触发send

结果返回阶段。这个send调用就是请求的划分阶段点

1. 将阻塞方法调用按照时间分解为多个阶段的方法调用

系统中的事件收集、分发者并非可以处理任何事件。如果按照前一种方式试图划分某个方法时，那么可能会发现找出的触发事件不能够被事件收集、分发器所处理，这时只能按照执行时间来拆分这个方法了

例如：当我们读取10MB的文件时我们可以这样来分解读取文件调用：把10MB均分成1000份，每次只读取10KB。这样，读取10KB的时间就是可控的，意味着这个事件接收器占用进程的时间不会太久，整个系统可以及时地处理其他请求。读取0-10kb阶段完成后可以使用网络事件触发或者设置简单定时器继续调用下一阶段。

（3）在“无所事事”且必须等待系统的响应，从而导致进程空转时，使用定时器划分阶

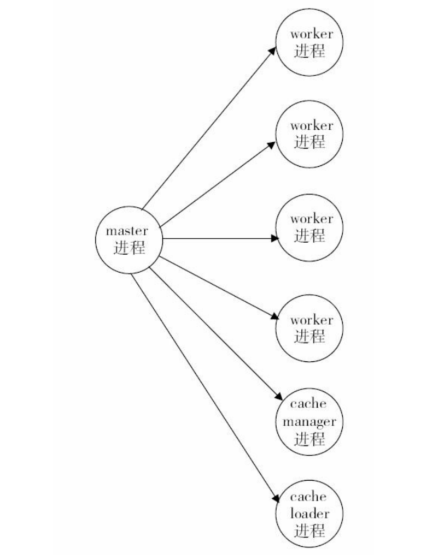
段

因为进程需要持续检查标志位来判断是否需要继续执行时，而导致进程阻塞时，应该使用定时器来代替循环检查

1. 如果阻塞方法完全无法继续划分，则必须使用独立的进程执行这个阻塞方法

2.2.4 管理进程，多工作进程设计

Nginx采用一个master管理进程、多个worker工作进程的设计方式



这种设计带来以下优点：（1）充分利用多核的并发处理能力（2）多个worker工作进程间通过进程间通信来实现负载均衡（3）管理进程会负责监控工作进程的状态，并负责管理其行为

2.2.5 平台无关的代码实现

使用C语言实现Nginx时，尽量减少使用与操作系统平台相关的代码，在与操作系统相关的系统调用上则分别针对各个操作系统都有独立的实现，这最终造就了Nginx的可移植性

2.2.6 内存池的设计

为了避免出现内存碎片、减少向操作系统申请内存的次数、降低各个模块的开发复杂

度，Nginx设计了简单的内存池。这个内存池没有很复杂的功能：通常它不负责回收内存池

中已经分配出的内存。这种内存池最大的优点在于：把多次向系统申请内存的操作整合成一

次，这大大减少了CPU资源的消耗，同时减少了内存碎片.

2.2.7使用统一管道过滤器模式的HTTP过滤模块

有一类HTTP模块被命名为HTTP过滤模块，其中每一个过滤模块都有输入端和输出端，

这些输入端和输出端都具有统一的接口。这些过滤模块将按照configure执行时决定的顺序组成一个流水线式的加工HTTP响应的中心，每一个过滤模块都是完全独立的，它处理着输入端接收到的数据，并由输出端传递给下一个过滤模块。每一个过滤模块都必须可以增量地处理数据，也就是说能够正确处理完整数据流的一部分

**2.3 Nginx框架中的核心结构体ngx\_cycle\_t**

无论是master管理进程、worker工作进程还是cache manager（loader）进程，每一个进程都毫无例外地拥有唯一一个ngx\_cycle\_t结构体。服务在初始化时就以ngx\_cycle\_t对象为中心来提供服务，在正常运行时仍然会以ngx\_cycle\_t对象为中心

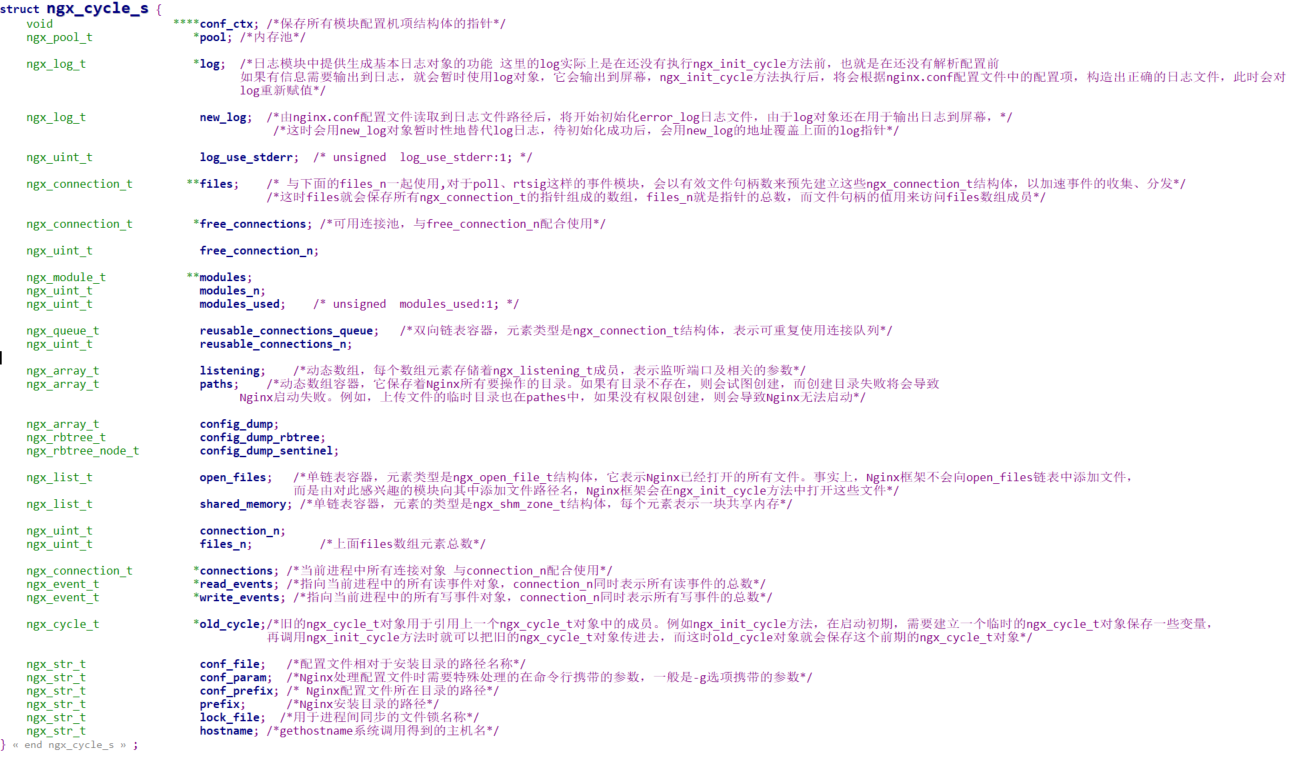
2.3.1 ngx\_listening\_t结构体

ngx\_cycle\_t对象中有一个动态数组成员叫做listening，它的每个数组元素都是ngx\_listening\_t结构体，而每个ngx\_listening\_t结构体又代表着Nginx服务器监听的一个端口, ngx\_listening\_t的定义如下：



2.3.2 ngx\_cycle\_t结构体

Nginx框架是围绕着ngx\_cycle\_t结构体来控制进程运行的 其定义如下：



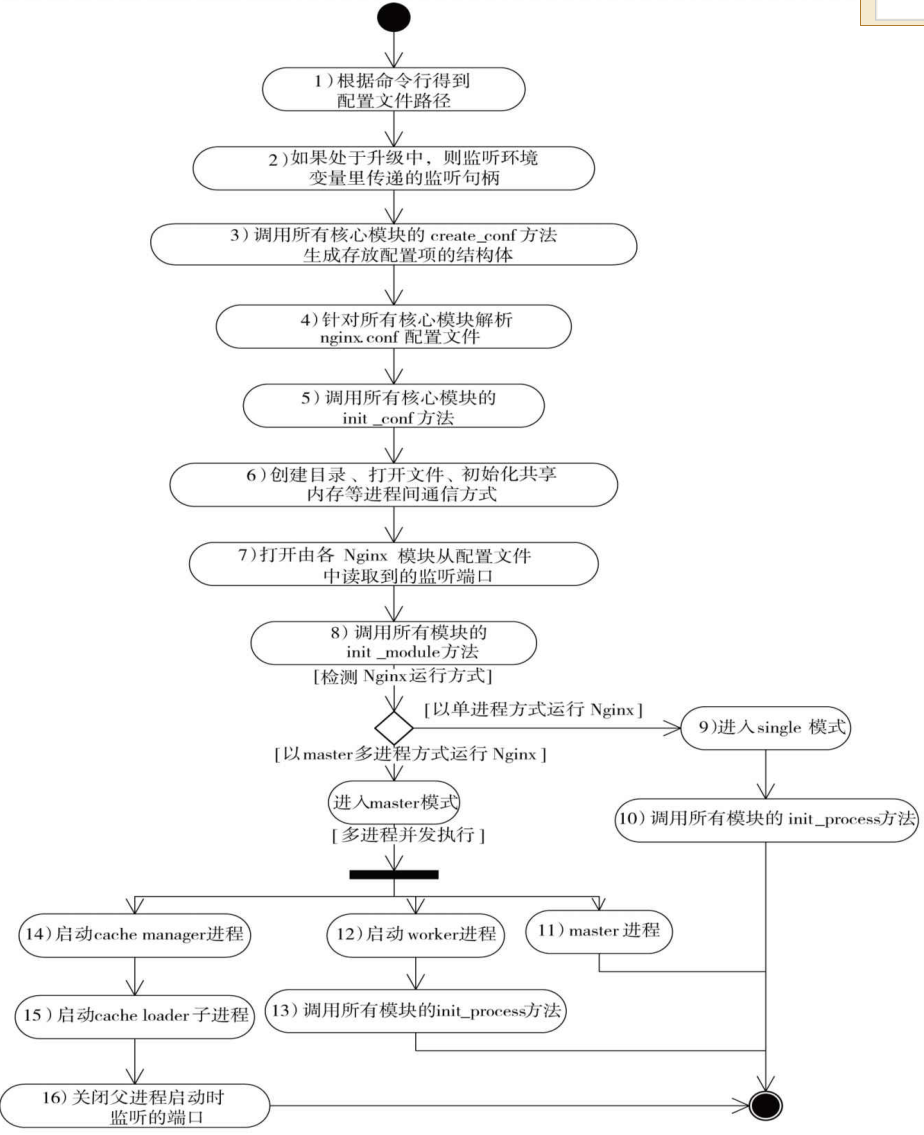
在构造ngx\_cycle\_t结构体成员的ngx\_init\_cycle方法中，上面所列出的pool内存池成员、

hostname主机名、日志文件new\_log和log、存储所有路径的pathes数组、共享内存、监听端口等都会在该方法中初始化

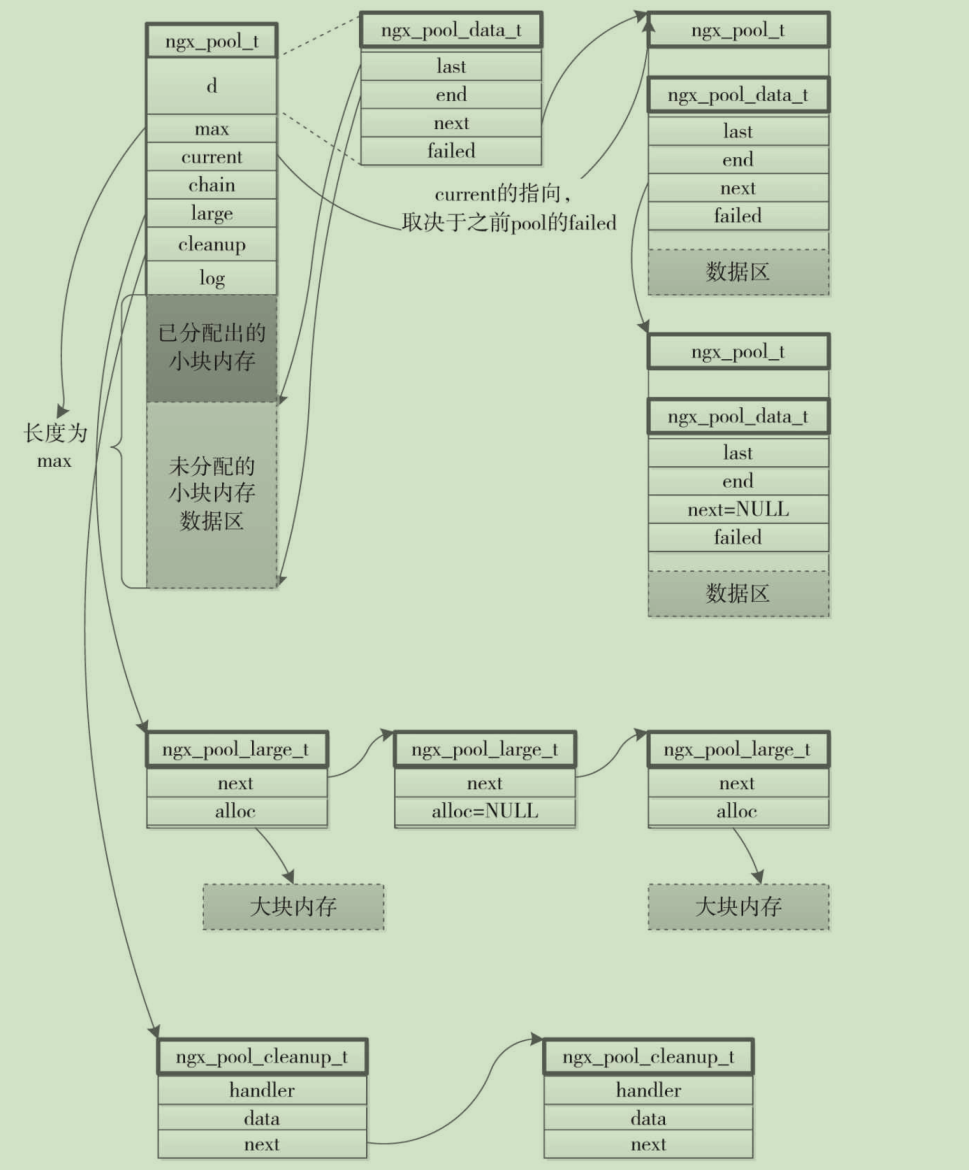
2.3.3 ngx\_cycle\_t 支持的方法

详见源码中 ngx\_process\_cycle.c 文件

2.3.4 nginx 启动流程



**2.3.5 nginx 内存池设计**



结构体设计详见：readcode中的ngx\_palloc.h 和ngx\_palloc.c 文件

**2.4 事件模块**

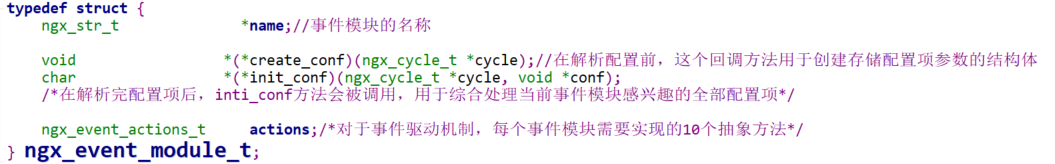
**2.4.1 事件处理框架概述**

Nginx定义了一系列（目前为9个）运行在不同操作系统、不同内核版本上的事件

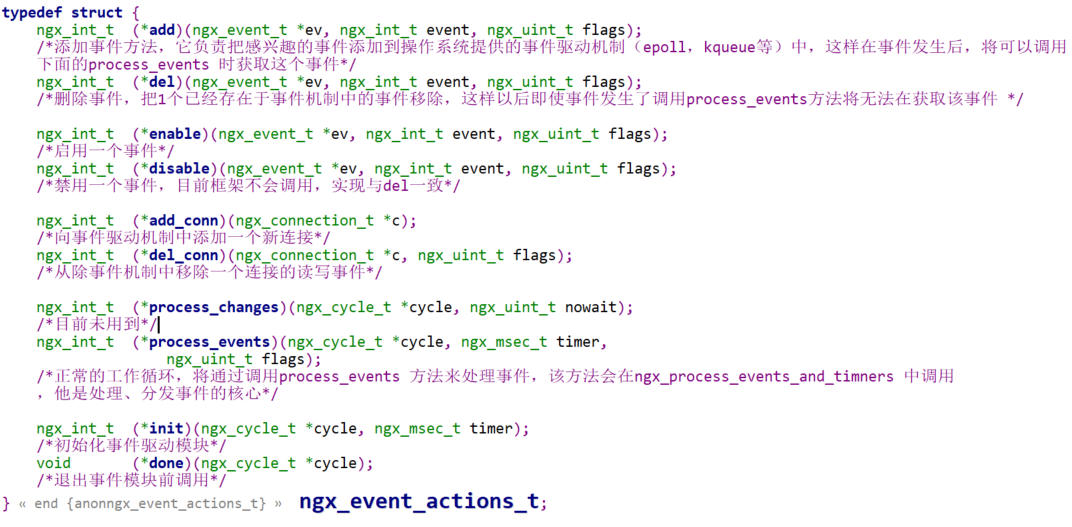
驱动模块，包括：ngx\_epoll\_module、ngx\_kqueue\_module、ngx\_poll\_module、ngx\_select\_module、ngx\_devpoll\_module、ngx\_eventport\_module、ngx\_aio\_module、

ngx\_rtsig\_module和基于Windows的ngx\_select\_module模块。

事件模块的通用接口是ngx\_event\_module\_t 结构体

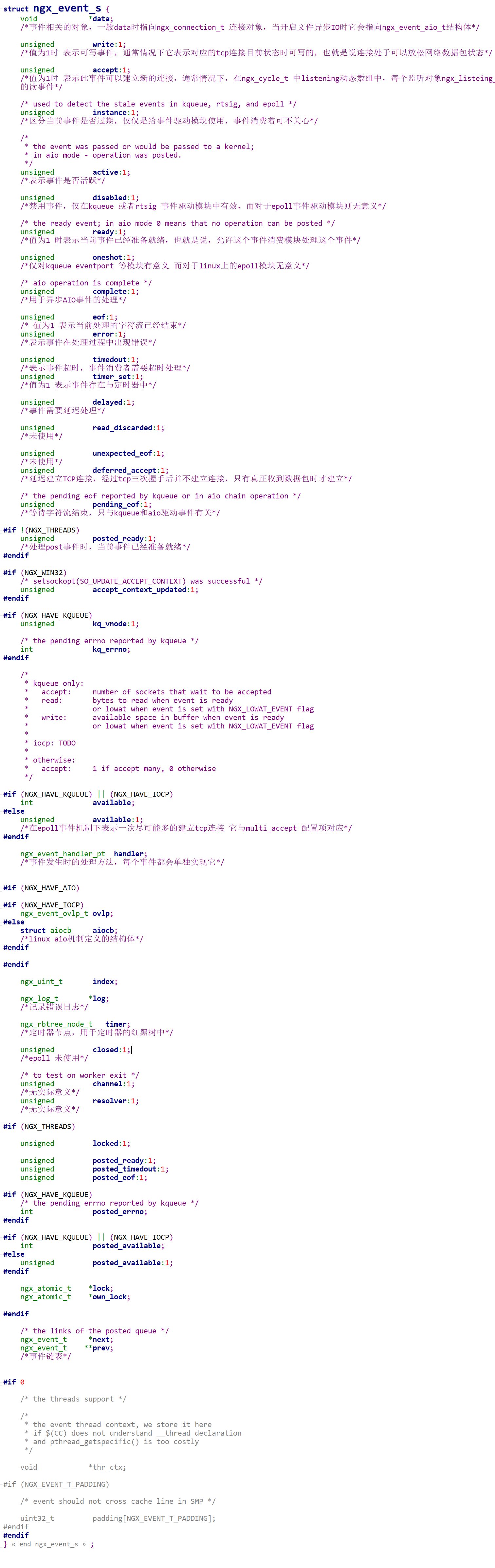


actions 成员定义事件驱动模块的10个核心方法



**2.4.2 nginx 事件的定义**

Nginx 中，每一个事件都有ngx\_event\_t结构体来表示



操作事件的方法：

事件是不要创建的，nginx 再启动时已经再ngx\_cycle\_t 的read\_events 成员中预先分配了所有的读事件，并在write\_events 分配好了所有的写事件

ngx\_int\_t ngx\_handle\_read\_event(ngx\_event\_t \*rev, ngx\_uint\_t flags)

/\* 将读事件添加到事件驱动模块中

rev:要操作的事件

flags:事件驱动方式，epoll驱动机制为列 flags取值为0时 或者为NGX\_CLOSE\_EVENT，nginx 主要工作在ET模式下 一般忽略flags参数

\*/

ngx\_int\_t ngx\_handle\_write\_event(ngx\_event\_t \*wev, size\_t lowat)

/\*

将写事件添加到事件驱动模型中

wev:操作的事件

lowat:只有当连接对应的套接字缓冲区必须有lowat 大小的可用空间时 才处理这个可写事件

\*/

**2.4.3 nginx 连接定义**

ngx\_connection\_t 表示客户端主动发起的TCP连接，即为nginx 服务器的被动连接

ngx\_peer\_connection\_t 表示nginx主动发起的连接，这两种连接不可以随意创建必须从连接池中获取



/\*连接结构体ngx\_connection\_s定义的rev send rev\_chain send\_chain 原型方法\*/

typedef ssize\_t (\*ngx\_recv\_pt)(ngx\_connection\_t \*c, u\_char \*buf, size\_t size);

typedef ssize\_t (\*ngx\_recv\_chain\_pt)(ngx\_connection\_t \*c, ngx\_chain\_t \*in);

typedef ssize\_t (\*ngx\_send\_pt)(ngx\_connection\_t \*c, u\_char \*buf, size\_t size);

typedef ngx\_chain\_t \*(\*ngx\_send\_chain\_pt)(ngx\_connection\_t \*c, ngx\_chain\_t \*in,

off\_t limit);

**2.4.4 nginx 主动连接**

作为Web服务器，Nginx也需要向其他服务器主动发起连接

主动连接结构体定义：



typedef ngx\_int\_t (\*ngx\_event\_get\_peer\_pt)(ngx\_peer\_connection\_t \*pc,

void \*data);

/\*当使用长连接与上游服务器通信时，可通过该方法由连接池中获取一个新连接\*/

typedef void (\*ngx\_event\_free\_peer\_pt)(ngx\_peer\_connection\_t \*pc, void \*data,

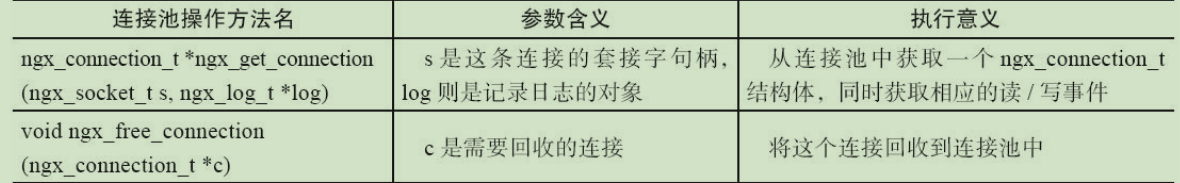
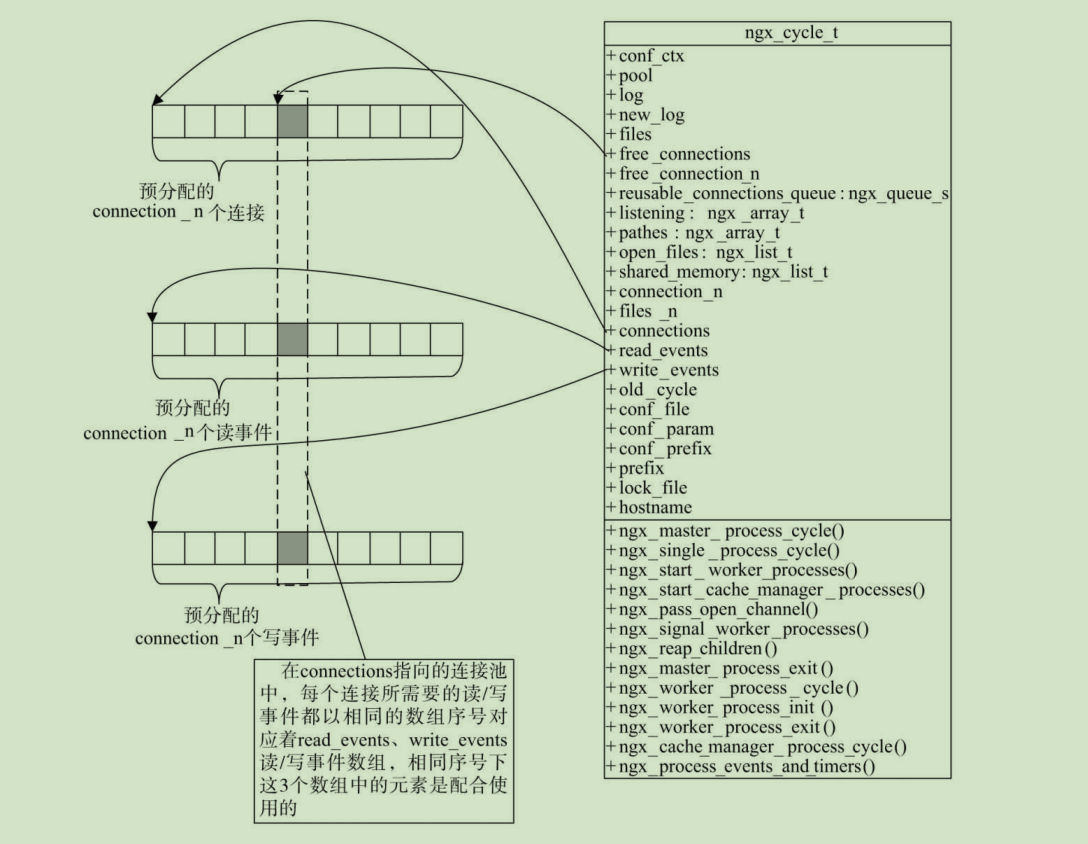
ngx\_uint\_t state);

/\*当使用长连接与上游服务器通信时，通过该方法将使用完毕的连接\*/

**2.4.5 ngx\_connection\_t 连接池**

Nginx在接受客户端的连接时，所使用的ngx\_connection\_t结构体都是在启动阶段就预分配好的，使用时从连接池中获取。在ngx\_cycle\_t中的connections和free\_connections这两个成员构成了一个连接池，Nginx认为每一个连接一定至少需要一个读事件和一个写事件，有多少连接就分配多少个读写事件。

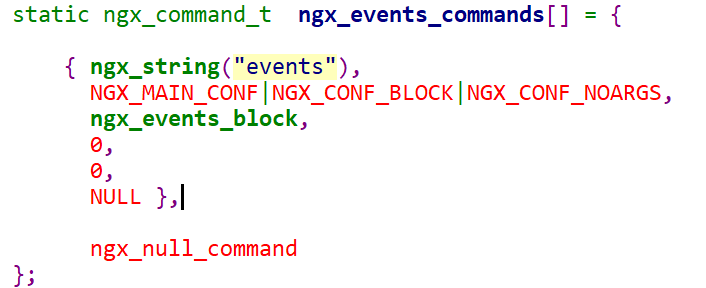
ngx\_connection\_t 连接池示意图：



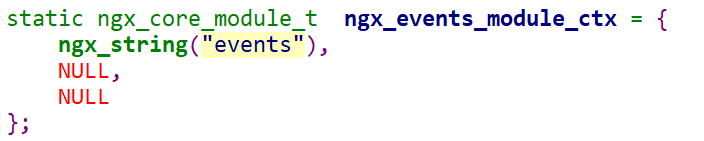
**2.3.6 ngx\_event\_module 核心模块**

ngx\_events\_module模块是一个核心模块，它定义了一类新模块：事件模块，每个事件模块都需要实现的ngx\_event\_module\_t接口，还需要管理这些事件模块生成的配置项结构体，并解析事件类配置项，当然，在解析配置项时会调用其在ngx\_command\_t数组中定义的回调方法。

定义一个Nginx模块就是在实现ngx\_modult\_t结构体。这里需要先定义好ngx\_command\_t（决定这个模块如何处理自己感兴趣的配置项）数组，因为任何模块都是以配置项来定制功能的。ngx\_events\_commands数组决定了ngx\_events\_module模块是如何定制其功能的。

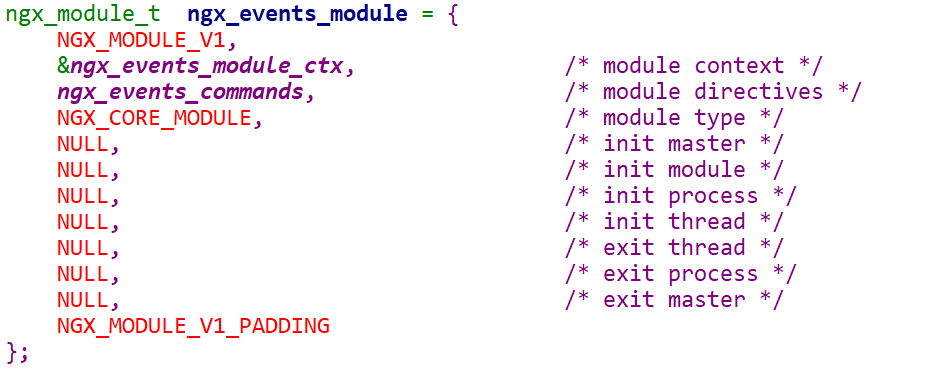


作为核心模块，ngx\_events\_module还需要实现核心模块的共同接口ngx\_core\_module\_t



ngx\_core\_module\_t接口中定义的create\_conf方法和init\_conf方法都没有实现这是因为ngx\_events\_module模块并不会解析配置项的参数，只是在出现events配置项后会调用各事件模块去解析events{...}块内的配置项。

ngx\_events\_module模块的定义：



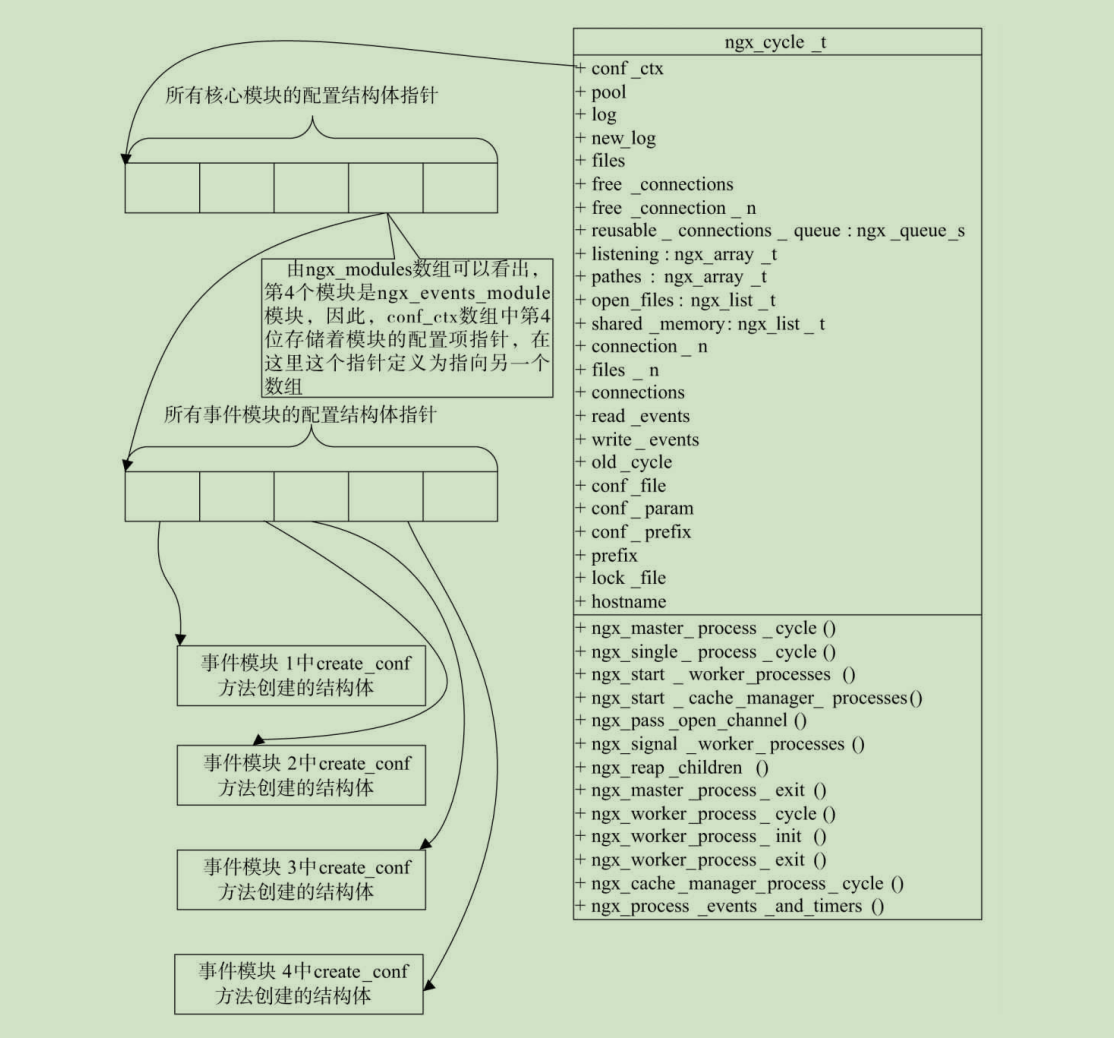
**2.3.7 如何管理所有事件模块的配置项**

每一个事件模块都必须实现ngx\_event\_module\_t接口，这个接口中允许每个事件模块建立自己的配置项结构体，用于存储感兴趣的配置项在nginx.conf中对应的参数。ngx\_event\_module\_t中的create\_conf方法就是用于创建这个结构体的方法，事件模块只需要在这个方法中分配内存即可

ngx\_cycle\_t核心结构体中的conf\_ctx，每个模块通过调用通用接口中也有create\_conf方法将配置信息解析到conf\_ctx 中

事件模块可以通过以下宏接口获取create\_conf 分配的结构体指针





2.3.8 **管理事件模块**

Nginx各模块在ngx\_modules数组中的顺序是很重要的，依靠index成员，每一个模块才可以把自己的位置与其他模块的位置进行比较，并以此决定行为。但是，Nginx同时又允许再次定义子类型，如事件类型、HTTP类型、mail类型，那同一类型的模块间又如何区分顺序呢（依靠index当然可以区分顺序，但index是针对所有模块的，这样效率太差）？这就得依靠ctx\_index成员了。ctx\_index表明了模块在相同类型模块中的顺序。

Ngx\_event\_module 核心模块加载事件模块：（ngx\_event\_module 的ngx\_events\_block方法）



4步中的：针对所有事件类型的模块解析配置项。这时，每个事件模块定义的ngx\_command\_t决定了配置项的解析方法，如果在nginx.conf中发现相应的配置项，就会回调各事件模块定义的方法。

**2.3.9 ngx\_event\_core\_module 事件模块**

ngx\_event\_core\_module模块是一个事件类型的模块，它在所有事件模块中的顺序是第一位（configure执行时必须把它放在其他事件模块之前）。这就保证了它会先于其他事件模块执行。ngx\_event\_core\_module模块它会创建连接池（包括读/写事件）同时会决定究竟使用哪些事件驱动机制，以及初始化将要使用的事件模块。

ngx\_event\_core\_module模块感兴趣配置项：

