# 其他模块 (40%)

Nginx的模块种类挺多的，除了HTTP模块，还有一些核心模块和mail系列模块。核心模块主要是做一些基础功能，比如Nginx的启动初始化，event处理机制，错误日志的初始化，ssl的初始化，正则处理初始化。

mail模块可以对imap，pop3，smtp等协议进行反向代理，这些模块本身不对邮件内容进行处理。

## core模块 (40%)

### Nginx的启动模块 (40%)

启动模块从启动Nginx进程开始，做了一系列的初始化工作，源代码位于src/core/nginx.c，从main函数开始:

1. 时间、正则、错误日志、ssl等初始化
2. 读入命令行参数
3. OS相关初始化
4. 读入并解析配置
5. 核心模块初始化
6. 创建各种暂时文件和目录
7. 创建共享内存
8. 打开listen的端口
9. 所有模块初始化
10. 启动worker进程

## event模块 (40%)

### event的类型和功能 (40%)

Nginx是以event（事件）处理模型为基础的模块。它为了支持跨平台，抽象出了event模块。它支持的event处理类型有：AIO（异步IO），/dev/poll（Solaris 和Unix特有），epoll（Linux特有），eventport（Solaris 10特有），kqueue（BSD特有），poll，rtsig（实时信号），select等。

event模块的主要功能就是，监听accept后建立的连接，对读写事件进行添加删除。事件处理模型和Nginx的非阻塞IO模型结合在一起使用。当IO可读可写的时候，相应的读写事件就会被唤醒，此时就会去处理事件的回调函数。

特别对于Linux，Nginx大部分event采用epoll EPOLLET（边沿触发）的方法来触发事件，只有listen端口的读事件是EPOLLLT（水平触发）。对于边沿触发，如果出现了可读事件，必须及时处理，否则可能会出现读事件不再触发，连接饿死的情况。

typedef struct {  
 /\* 添加删除事件 \*/  
 ngx\_int\_t (\*add)(ngx\_event\_t \*ev, ngx\_int\_t event, ngx\_uint\_t flags);  
 ngx\_int\_t (\*del)(ngx\_event\_t \*ev, ngx\_int\_t event, ngx\_uint\_t flags);  
  
 ngx\_int\_t (\*enable)(ngx\_event\_t \*ev, ngx\_int\_t event, ngx\_uint\_t flags);  
 ngx\_int\_t (\*disable)(ngx\_event\_t \*ev, ngx\_int\_t event, ngx\_uint\_t flags);  
  
 /\* 添加删除连接，会同时监听读写事件 \*/  
 ngx\_int\_t (\*add\_conn)(ngx\_connection\_t \*c);  
 ngx\_int\_t (\*del\_conn)(ngx\_connection\_t \*c, ngx\_uint\_t flags);  
  
 ngx\_int\_t (\*process\_changes)(ngx\_cycle\_t \*cycle, ngx\_uint\_t nowait);  
 /\* 处理事件的函数 \*/  
 ngx\_int\_t (\*process\_events)(ngx\_cycle\_t \*cycle, ngx\_msec\_t timer,  
 ngx\_uint\_t flags);  
  
 ngx\_int\_t (\*init)(ngx\_cycle\_t \*cycle, ngx\_msec\_t timer);  
 void (\*done)(ngx\_cycle\_t \*cycle);  
} ngx\_event\_actions\_t;

上述是event处理抽象出来的关键结构体，可以看到，每个event处理模型，都需要实现部分功能。最关键的是add和del功能，就是最基本的添加和删除事件的函数。

### accept锁 (40%)

Nginx是多进程程序，80端口是各进程所共享的，多进程同时listen 80端口，势必会产生竞争，也产生了所谓的"惊群"效应。当内核accept一个连接时，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，其他的进程都是被无效唤醒的。所以Nginx采用了自有的一套accept加锁机制，避免多个进程同时调用accept。Nginx多进程的锁在底层默认是通过CPU自旋锁来实现。如果操作系统不支持自旋锁，就采用文件锁。

Nginx事件处理的入口函数是ngx\_process\_events\_and\_timers()，下面是部分代码，可以看到其加锁的过程：

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;  
 }  
 }  
 }  
}

在ngx\_trylock\_accept\_mutex()函数里面，如果拿到了锁，Nginx会把listen的端口读事件加入event处理，该进程在有新连接进来时就可以进行accept了。注意accept操作是一个普通的读事件。下面的代码说明了这点：

(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);  
}

ngx\_process\_events()函数是所有事件处理的入口，它会遍历所有的事件。抢到了accept锁的进程跟一般进程稍微不同的是，它被加上了NGX\_POST\_EVENTS标志，也就是说在ngx\_process\_events() 函数里面只接受而不处理事件，并加入post\_events的队列里面。直到ngx\_accept\_mutex锁去掉以后才去处理具体的事件。为什么这样？因为ngx\_accept\_mutex是全局锁，这样做可以尽量减少该进程抢到锁以后，从accept开始到结束的时间，以便其他进程继续接收新的连接，提高吞吐量。

ngx\_posted\_accept\_events和ngx\_posted\_events就分别是accept延迟事件队列和普通延迟事件队列。可以看到ngx\_posted\_accept\_events还是放到ngx\_accept\_mutex锁里面处理的。该队列里面处理的都是accept事件，它会一口气把内核backlog里等待的连接都accept进来，注册到读写事件里。

而ngx\_posted\_events是普通的延迟事件队列。一般情况下，什么样的事件会放到这个普通延迟队列里面呢？我的理解是，那些CPU耗时比较多的都可以放进去。因为Nginx事件处理都是根据触发顺序在一个大循环里依次处理的，因为Nginx一个进程同时只能处理一个事件，所以有些耗时多的事件会把后面所有事件的处理都耽搁了。

除了加锁，Nginx也对各进程的请求处理的均衡性作了优化，也就是说，如果在负载高的时候，进程抢到的锁过多，会导致这个进程被禁止接受请求一段时间。

比如，在ngx\_event\_accept函数中，有类似代码：

ngx\_accept\_disabled = ngx\_cycle->connection\_n / 8  
 - ngx\_cycle->free\_connection\_n;

ngx\_cycle->connection\_n是进程可以分配的连接总数，ngx\_cycle->free\_connection\_n是空闲的进程数。上述等式说明了，当前进程的空闲进程数小于1/8的话，就会被禁止accept一段时间。

### 定时器 (40%)

Nginx在需要用到超时的时候，都会用到定时器机制。比如，建立连接以后的那些读写超时。Nginx使用红黑树来构造定期器，红黑树是一种有序的二叉平衡树，其查找插入和删除的复杂度都为O(logn)，所以是一种比较理想的二叉树。

定时器的机制就是，二叉树的值是其超时时间，每次查找二叉树的最小值，如果最小值已经过期，就删除该节点，然后继续查找，直到所有超时节点都被删除。

## mail模块

### mail模块的实现

### mail模块的功能