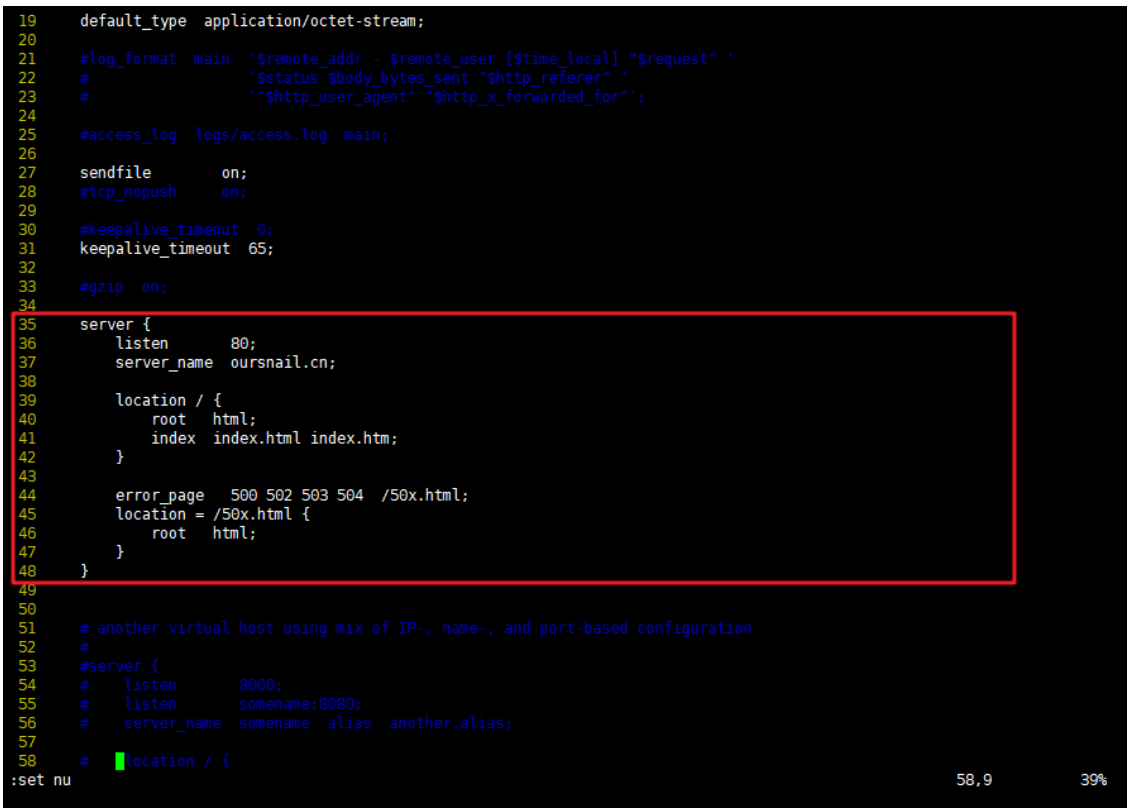
前面介绍了Nginx的基本安装，也看到了最简单的欢迎页面，不禁好奇它的欢迎页面是如何展示出来的？nginx是如何处理来自我的请求的，它内部机制是什么？为什么说nginx性能好能抗住较高的并发？我们常用的tomcat跟nginx为什么是两种使用场景？带着这些问题我们出发，不过本系列坚持的是基础路线，不会太过深入。

一、Nginx显示默认首页的过程解析

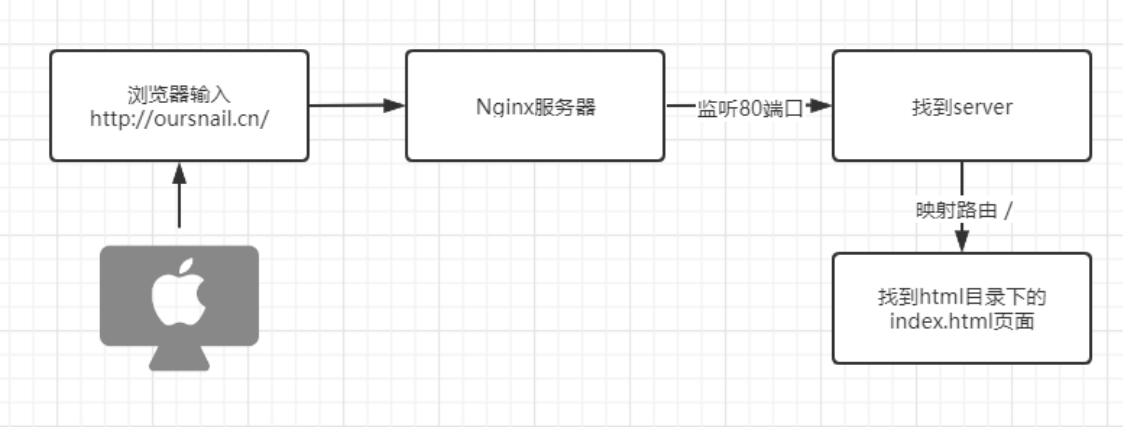
前面文章中已经说明了三个文件夹的基本作用，提到`cong/nginx.conf`是最核心的配置文件，而里面最最核心的配置是下面这块（已将多余的注释删除）：




```
    server {
        listen       80; # 表示Nginx监听的默认端口，这里可以进行修改
        server_name  oursnail.cn; # 域名或IP
        # 上面匹配上了，则来到下面，默认配置的是/，即访问的是nginx根路径
        location / {
            root   html; # 默认是找到与conf与文件夹同级目录下的html目录
            index  index.html index.htm; # 默认展示html目录下的index页面
        }
        # nginx异常显示的目录
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
    }
```

可以看到，这里配置的是监听的端口和域名信息，并且可以根据访问的路径进行资源的映射。那么如果我将路径映射到我们的前端工程的目录下，是不是就有可能通过nginx展示出页面了呢？答案当然是了！

显示首页的流程也比较清晰了：



二、Nginx的进程模型

Nginx启动的时候，会默认有两个进程：

- master进程：主进程
- worker进程：工作进程

我们来看下进程：

```
[root@VM-0-13-centos sbin]# ps -ef | grep nginx
root     12124     1  0 16:07 ?        00:00:00 nginx: master
process ./nginx
nobody   19984 12124  0 16:48 ?        00:00:00 nginx: worker
process
root     25051  3126  0 17:15 pts/0    00:00:00 grep --color=auto
nginx
```

可以看到，有 `master` 和 `worker` 两个进程。 `worker` 进程的数量是可以配置的，在 `nginx.conf` 一开头的地方就可以进行配置：

```
#user  nobody;
worker_processes  1;
```

如果我这里配置成2，重启nginx：

```
#user  nobody;
worker_processes  2;
```

再来查看进程就会显示如下：

```
[root@VM-0-13-centos sbin]# ps -ef | grep nginx
root     12124     1  0 16:07 ?        00:00:00 nginx: master
process ./nginx
nobody   25516 12124  0 17:17 ?        00:00:00 nginx: worker
process
nobody   25517 12124  0 17:17 ?        00:00:00 nginx: worker
process
root     25534  3126  0 17:17 pts/0    00:00:00 grep --color=auto
nginx
```

一般情况下，这个值设置的跟机器的CPU核心数一样即可。我们先来学习下如何计算的。其实很简单：

> 总核数 = 物理CPU个数 X 每颗物理CPU的核数
> 总逻辑CPU数 = 物理CPU个数 X 每颗物理CPU的核数 X 超线程数

物理CPU是什么？

> 实实在在插在主机上看得见摸得着那块CPU硬件：
> 
>

如何查看CPU的个数呢?

```
cat /proc/cpuinfo| grep "physical id"| sort| uniq| wc -l
```

CPU核数:

> 一块物理CPU上能处理数据的芯片组数量。也就是说一个物理CPU上可能会有多个核心，日常中说的双核，四核就是指的CPU核心。

查看每个物理CPU中core的个数(即核数):

```
cat /proc/cpuinfo| grep "cpu cores"| uniq
```

超线程:

> 一个CPU核就是一个物理线程，不过可以通过超线程技术，使得单个核心用起来像两个核一样，以充分发挥CPU的性能。

逻辑CPU:

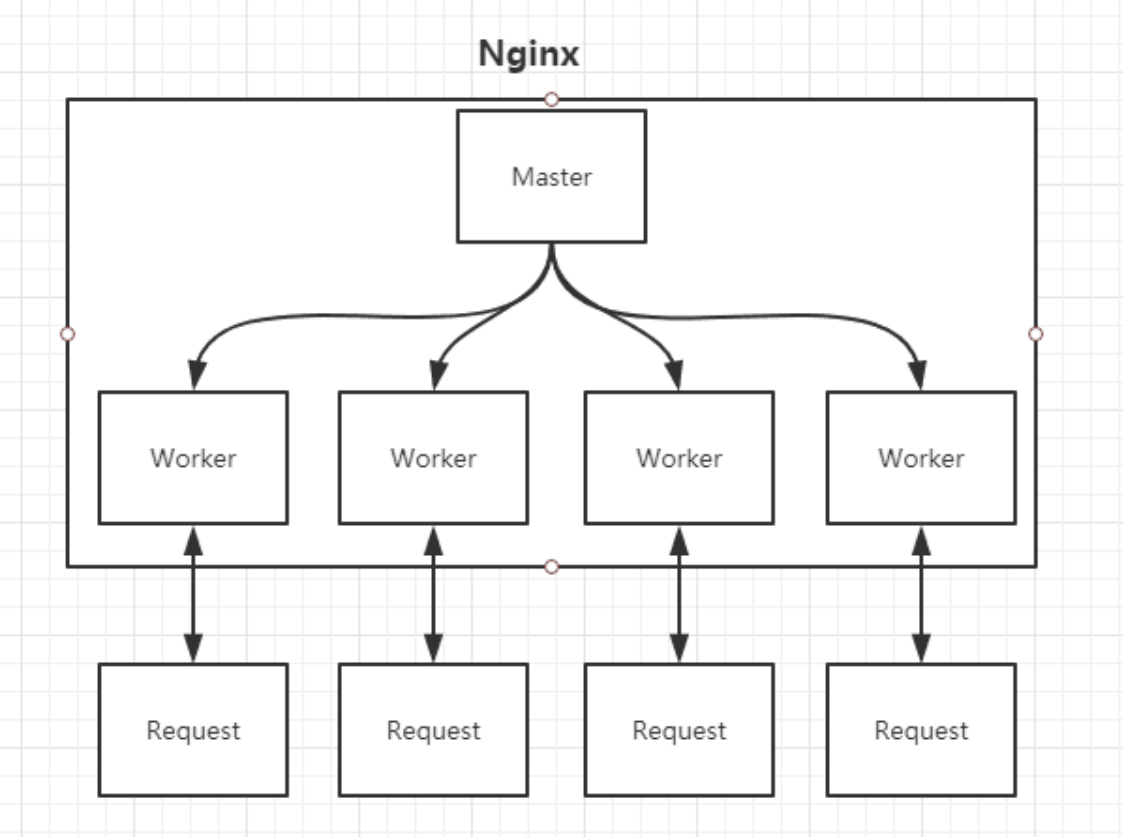
> 通常来说，总的逻辑CPU数对应总的CPU核数，但借助超线程技术，一个核用起来像两个核，这时逻辑CPU数就是核心数的两倍了。

查看逻辑CPU的个数:

```
cat /proc/cpuinfo| grep "processor"| wc -l
```

`worker` 进程的数量设置为 `auto` 后可以根据实际的核心数设置工作进程，从而合理利用CPU资源，因为当配置过少时就会浪费CPU的性能，有些核心可能就帮不上忙；如果配置过多超出了核心数量，那么一个核心就要处理多个进程，进程相互竞争cpu资源，谁也不让谁，为了公平起见，多个进程都会获得执行的机会，继而就会有进程切换的问题，我们要知道进程的切换是比较费劲的。

那么 `master` 和 `worker` 进程之间到底是什么关系呢?



可以看出，`master`进程是大管家，会把请求分发给各个`worker`进程去处理。
`master`进程的功能很关键：

- 接收来自外界的信号（所谓的信号就是指令，比如./nginx -stop这样的指令等等）。
- 向各worker进程发送信号。
- 监控worker进程的运行状态，当worker进程退出后(异常情况下)，会自动重新启动新的worker进程。

而`worker`进程就是接收来自`master`的任务去处理的伙计们：

- Nginx采用worker进程来处理请求，一个worker进程只有一个主线程，那么有多少个worker子进程就能处理多少个并发。
- 一个完整的请求读取请求、解析请求、处理请求，产生数据后，再返回给客户端，最后断开连接。
- 一个完整的请求完全由一个worker进程处理。

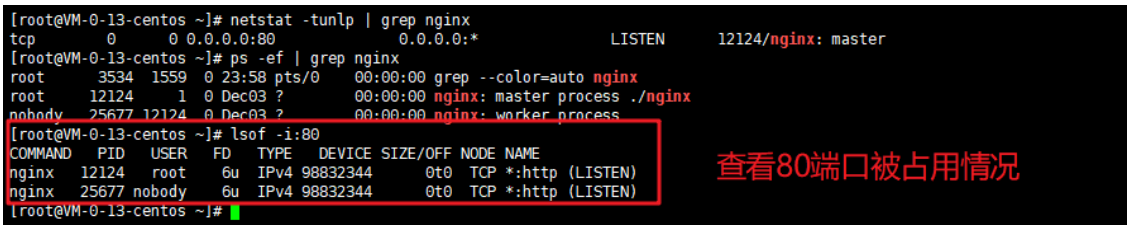
多进程之间资源互相隔离，不会互相产生影响，这是多进程的好处。因此Nginx的进程模型就是：`mater`主进程+N个`worker`进程，`worker`进程接收信号、管理`worker`等，而`worker`进程处理请求并返回结果给客户端。

此外，配置文件中最上面可以看到一个配置，比较简单，我们来注释下它的含义。

```
events {
    # 默认使用epoll
    use epoll;
    # 每个worker允许连接的客户端最大连接数
    worker_connections  1024;
}
```

可以看到，每个`worker`进程，默认可以最大同时维持1024个客户端的连接，我们可以根据实际情况增大一点。说到这里，上下翻动配置文件看看，实际上最重要的可配置的部分都已经说完了。

我们再来看下端口占用情况：



我用`netstat -tunlp`命令则只能查询出一个master监听的进程，这个命令是专门用于显示 tcp，udp 的端口和进程等相关情况。而没有显示worker进程，这个符合我们的认知，因为标识一个进程的地址就是端口号，如果两个进程的端口号一样，不就乱套了吗？不过如果用`losf`命令可以看到80端口是被两个进程占用的。对此我就产生了疑惑，好像哪里不对劲。。。

开启多线程很简单，通过`new Thread()`即可创建线程，但是如何多进程呢？我们知道，多次启动同一个进程会报“Address already in use!"的错误的。这是由于该端口号已经被监听了：比如我们的tomcat，如果之前的进程没有被关闭而被重复启动，就会出现这样的错误，提示8080端口被占用。

结论是：**通过 fork 创建子进程的方式可以实现，其他情况下不行。**

因为我们监听端口发现重复了，那么一个策略是：只要在绑定端口号（bind函数）之后，监听端口号之前（listen函数），用fork（）函数生成子进程，这样子进程就可以克隆父进程，达到监听同一个端口的目的。找了一个人写的代码：

```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>

#define oops(m)     {perror(m); exit(1);}
int main(){
    int sock_id;
    struct  sockaddr_in saddr;
    sock_id = socket(PF_INET, SOCK_STREAM, 0);
    saddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    saddr.sin_port        = htons(9988);
    saddr.sin_family      = AF_INET;
    int ret = bind(sock_id, (struct sockaddr *) &saddr, sizeof(saddr));//绑定IP地址和端口
    if(ret == -1) oops("bind error");//如果返回-1，则绑定失败，一般为"Address alreay in use"
    int i;
    for(i = 0; i < 6; i++){//连续创建六个子进程
        int pid = fork();
        if(pid == 0) break;
    }
    listen(sock_id, 1);
    while(1){
        int sock = accept(sock_id, NULL, 0);
        char    buf[128];
        int readnum;
        readnum = read(sock, buf, 127);
        buf[readnum] = '\0';
        printf("pid=%d, mesg: %s\n", getpid(), buf);
        fflush(stdout);
        close(sock);
    }
    return 1;
}
```

每个worker进程都是从master进程fork过来，在master进程里面，先建立好需要listen的socket之后，然后再fork出多个worker进程，这样每个worker进程都可以去accept这个socket(**当然不是同一个socket，只是每个进程的这个socket会监控在同一个ip地址与端口，这个在网络协议里面是允许的**)。一般来说，当一个连接进来后，所有在accept在这个socket上面的进程，都会收到通知，而只有一个进程可以accept这个连接，其它的则accept失败，这是所谓的惊群现象。下面详细来说说这个现象和nginx如何解决的。

三、Nginx的进程抢占机制

假设配置文件配置 `worker` 进程数量为3，此时来一个请求到 `Nginx` ，首先是 `master` 进程处理，然后要给一个 `worker` 进程去处理。不过谁来处理呢?

这个时候实际上有两种处理机制。

假设你养了一百只小鸡，现在你有一粒粮食，那么有两种喂食方法：

- 你把这粒粮食直接扔到小鸡中间，一百只小鸡一起上来抢，最终只有一只小鸡能得手，其它九十九只小鸡只能铩羽而归。

- 你主动抓一只小鸡过来，把这粒粮食塞到它嘴里，其它九十九只小鸡对此浑然不知，该睡觉睡觉。

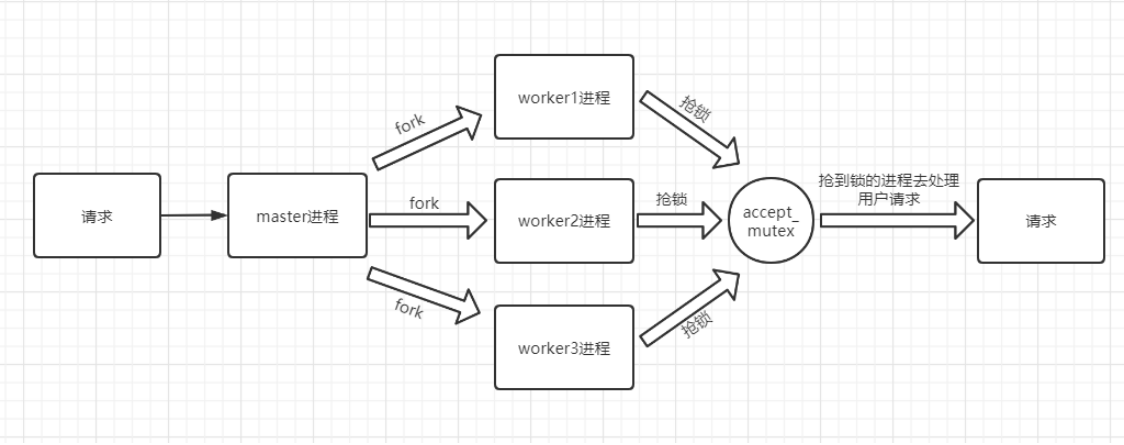
你就是 `master` ，小鸡就是 `worker` 们，一粒粮食就是一个请求。

到底用哪个机制呢？ `Nginx` 中通过 `accept_mutex` 的配置来决定。我们来看下它的含义。

当一个新连接到达时，如果激活了 `accept_mutex` ，那么多个 `Worker` 将以串行方式来处理，其中有一个 `Worker` 会被唤醒，其他的 `Worker` 继续保持休眠状态；

抢占的流程为：

- master进程先建好需要listen的socket后，然后再fork出多个woker进程，这样每个work进程都可以去accept这个socket
- 当一个client连接到来时，所有accept的work进程都会受到通知，但只有一个进程可以accept成功，其它的则会accept失败，Nginx提供了一把共享的互斥锁accept_mutex来保证同一时刻只有一个work进程在accept连接，从而解决惊群问题
- 当一个worker进程accept这个连接后，就开始读取请求，解析请求，处理请求，产生数据后，再返回给客户端，最后才断开连接，这样一个完成的请求就结束了



如果没有激活 `accept_mutex` ，那么所有的 `Worker` 都会被唤醒，不过只有一个 `Worker` 能获取新连接，其它的 `Worker` 会重新进入休眠状态，这就是惊群问题。

> 所谓惊群问题，就是指的像Nginx这种多进程的服务器，在fork后同时监听同一个端口时，如果有一个外部连接进来，会导致所有休眠的子进程被唤醒，而最终只有一个子进程能够成功处理accept事件，其他进程都会重新进入休眠中。

Nginx默认关闭了 `accept_mutex` 。从只有一粒粮食的场景看出来，激活 `accept_mutex` 相对更好一些。为什么会默认关闭了呢？

> As of nginx mainline version 1.11.3 (released 2016-07-26), accept_mutex now defaults to off.

让我们修改一下问题的场景，我不再只有一粒粮食，而是一盆粮食，怎么办？大家可以设想一下几十只小鸡排队等着喂食时那种翘首以盼的情景，是不是十分低效？此时更好的方法是把这盆粮食直接撒到小鸡中间，让它们自己去抢，虽然这可能会造成一定程度的混乱，但是整体的效率无疑大大增强了。

我们可以理解为当 `accept_mutex` 关闭后，依靠 `accept` 这个跨进程的互斥锁，来保证只有一个进程具备监听accept事件的能力。从而解决惊群问题。

> 这是一个在event模块初始化时就分配好的锁，放在一块进程间共享的内存中，以保证所有进程都能访问这一个实例，其加锁解锁是借由linux的原子变量来做CAS，如果加锁失败则立即返回，是一种非阻塞的锁。

nginx利用这种抢占机制，大大提升了nginx的处理能力，不过除了抢占机制，更应该归功于它利用了epoll模型，这个模型是linux上最契合的事件处理模型。

四、Nginx如何实现高并发

主要依靠的是多路复用模型，也有地方叫事件驱动模型，目前linux主流的是用epoll来实现的。我们来简单认识下。

由于I/O 操作（比如要读磁盘文件）都比较慢，这会导致某一文件的 I/O 阻塞导致整个进程无法对其它客户提供服务，而 I/O 多路复用就是为了解决这个问题而出现的。

如何理解多路复用模型呢？假设你是一个老师，让30个学生解答一道题目，然后检查学生做的是否正确，你有下面几个选择：

> 第一种选择：按顺序逐个检查，先检查A，然后是B，之后是C、D。。。这中间如果有一个学生卡主，全班都会被耽误。这种模式就好比，你用循环挨个处理socket，根本不具有并发能力。

> 第二种选择：你创建30个分身，每个分身检查一个学生的答案是否正确。 这种类似于为每一个用户创建一个进程或者线程处理连接。

> 第三种选择，你站在讲台上等，谁解答完谁举手。这时C、D举手，表示他们解答问题完毕，你下去依次检查C、D的答案，然后继续回到讲台上等。此时E、A又举手，然后去处理E和A。。。

第三种就是IO多路复用模型，Linux下的select、poll和epoll就是干这个的。I/O多路复用的本质是通过一种机制（系统内核缓冲I/O数据），让单个进程可以监视多个文件描述符，一旦某个描述符就绪（一般是读就绪或写就绪），能够通知程序进行相应的读写操作。

多路复用模式下，一个进程可以同时处理成千上万个请求，而不需要阻塞在某一个请求上。很适合IO密集型的任务。下面说说什么是IO密集型。说到IO密集型就不得不提对面的计算密集型了。

计算密集型：

> 计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数。

IO密集型：

> 第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web应用。

nginx使用这个多路复用模型的原因也是跟它的业务使用场景相关：

- Nginx更主要是作为反向代理，而非Web服务器使用。其网络模式是事件驱动（select、poll、epoll）。
- 事件驱动服务器，最适合做的就是这种IO密集型工作，如反向代理，它在客户端与WEB服务器之间起一个数据中转作用，纯粹是IO操作，自身并不涉及到复杂计算。
- Nginx处理静态文件效果也很好，那是因为静态文件本身也是磁盘IO操作，处理过程一样。

而tomcat这种应用服务器就不适合用多路复用模型，原因其实很简单，即使tomcat跟nginx一样支持了几万的并发，但是大多数的连接都要阻塞很久，比如操作数据库，再加上如果CPU计算再多一点，整个服务器的CPU、内存都会瞬间飙到峰值，甚至导致服务器的崩溃。因此我们说tomcat不适合使用这种事件驱动类型的模式。

也因此，nginx和tomcat需要相辅相成，前者作为反向代理服务器和静态资源服务器，后者作为处理业务逻辑的服务器，谁也不能替换谁。

五、并发量和QPS

顺便理一下并发量和QPS的区别吧，我认为简单合理的理解是：

- 并发量是客户端的事情，有多少并发量取决于有多少客户端并发访问
- qps是服务端的指标，服务端每秒处理请求数

当并发量较小的时候，服务器有足够的能力来处理请求，这是qps=并发量。当并发量增多，qps也随之增长，当并发量达到一个临界值，服务端处理能力达到极限。由于服务端需要处理额外的事情，比如线程切换等等，qps略微下降，当并发量再继续上升，由于请求堆积严重，可能导致服务端崩溃，qps急剧下降。一般情况下问并发量就是问的这个临界值。

另外服务端还有一个指标，响应时间。并发量是输入，qps和响应时间是输出。

也可以看到，线程数不是越多越好，线程的切换会对性能产生较大的影响。

也可以看出来，ngxin的并发量和tomcat的并发量根本没有可比性，因为通过上面的学习我们知道它们处理的维度不一样。