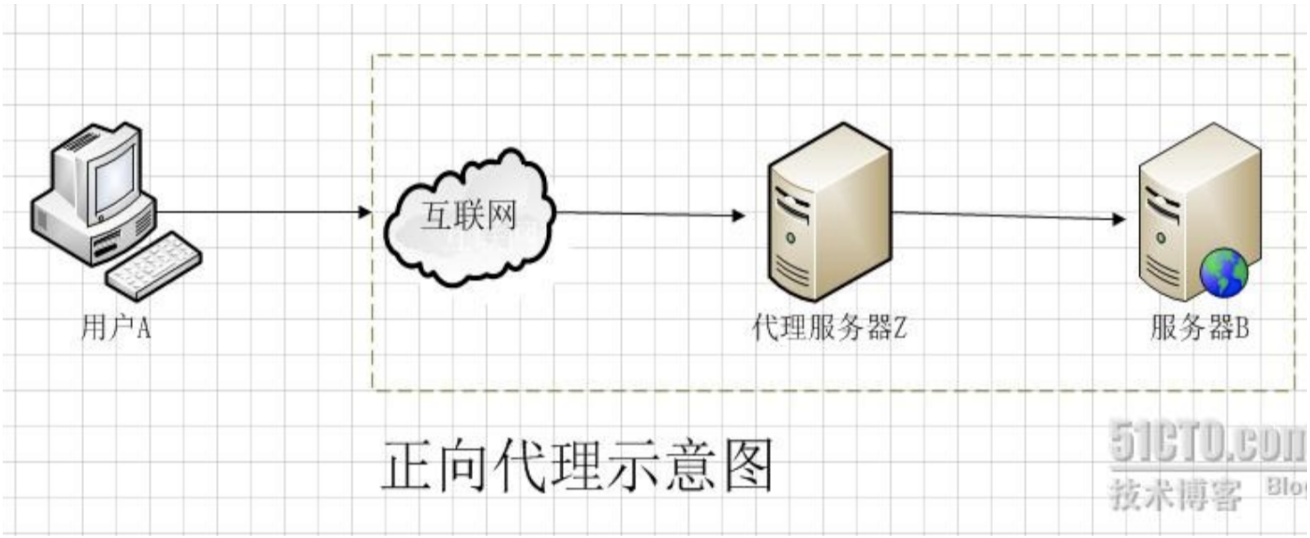
nginx+uWSGI+django+virtualenv+supervisor发布web服务器

1.nginx

1.nginx - 反向代理

1.正向代理

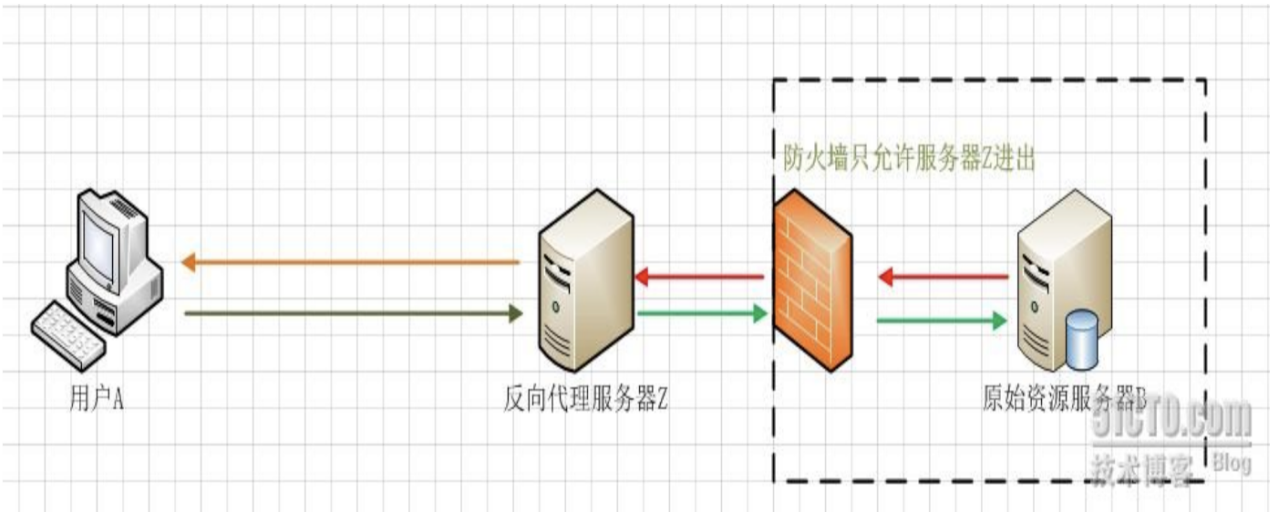
> 正向代理，也就是传说中的代理,他的工作原理就像一个跳板（VPN），简单的说：
> 我是一个用户，我访问不了某网站，但是我能访问一个代理服务器，这个代理服务器呢，他能访问那个我不能访问的网站，
> 于是我先连上代理服务器，告诉他我需要那个无法访问网站的内容，代理服务器去取回来，然后返回给我



2.反向代理

1、 保护和隐藏原始资源服务器

如下图2.1

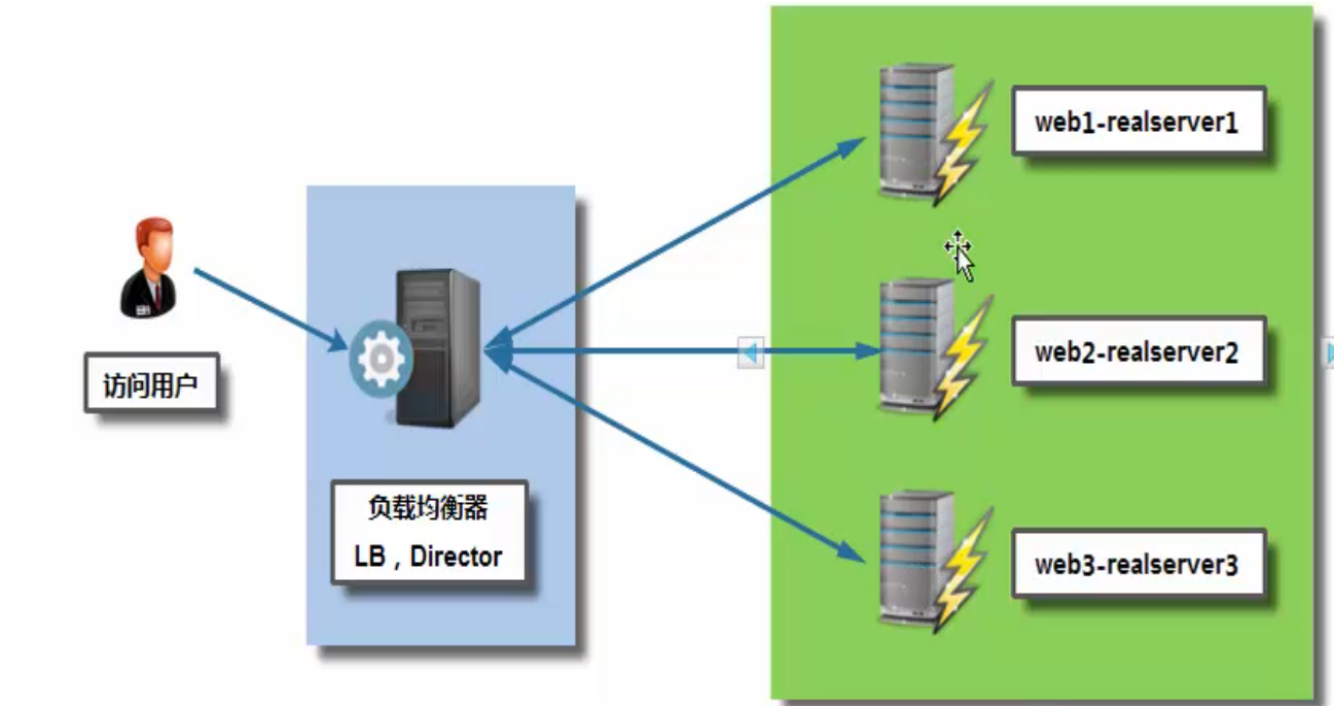



```
server{       # 一个虚拟主机
    listen 80;    # 监听的端口，访问的端口80
```

```
    server_name 192.168.11.11;  # 访问的域名192.168.11.11
    location / { # 访问的路径 /
        root html;  # 指定页面的目录，访问/会找到html目录
        index index.html # 指定网页，访问/就是访问index.html
    }
}

server{    #虚拟主机
    listen 8080;   #nginx监听端口
    server_name 192.168.11.11;   #nginx访问域名
    location / {    #location匹配url
        include uwsgi_params; #将uwsgi参数添加进nginx
        uwsgi_pass 0.0.0.0:8000; #反向代理转发请求给uwsgi
    }
}
```

2.nginx负载均衡



Nginx要实现负载均衡需要用到proxy_pass代理模块配置
Nginx负载均衡与Nginx代理不同地方在于
Nginx代理仅代理一台服务器，而Nginx负载均衡则是将客户端请求代理转发至一组upstream虚拟服务池
Nginx可以配置代理多台服务器，当一台服务器宕机之后，仍能保持系统可用

1.upstream配置

```
在nginx.conf > http 区域中

upstream django {
       server 10.0.0.10:8000;
       server 10.0.0.11:9000;
}
# - 在nginx.conf > http 区域 >  server区域  > location配置中

添加proxy_pass
location / {
            root   html;
            index  index.html index.htm;
            proxy_pass http://django;
}
#此时初步负载均衡已经完成，upstream默认按照轮训方式负载，每个请求按时间顺序逐一分配到后端节点。
```

2.upstream分配策略

```
# ------------------- weight 权重 ----------------------

upstream django {
       server 10.0.0.10:8000 weight=5;
       server 10.0.0.11:9000 weight=10;#这个节点访问比率是大于8000的
```

```
}
# ---------------------- ip_hash ------------------------

复制代码
每个请求按访问ip的hash结果分配，这样每个访客固定访问一个后端服务器
upstream django {
       ip_hash;
       server 10.0.0.10:8000;
       server 10.0.0.11:9000;
}
复制代码
# ----------------------backup ------------------------

在非backup机器繁忙或者宕机时，请求backup机器，因此机器默认压力最小

upstream django {
       server 10.0.0.10:8000 weight=5;
       server 10.0.0.11:9000;
       server node.oldboy.com:8080 backup;
}
```

3.nginx负载均衡调度算法

```
调度算法        概述
轮询            按时间顺序逐一分配到不同的后端服务器(默认)
weight          加权轮询,weight值越大,分配到的访问几率越高
ip_hash         每个请求按访问IP的hash结果分配,这样来自同一IP的固定访问一个后端服务器
url_hash        按照访问URL的hash结果来分配请求,是每个URL定向到同一个后端服务器
least_conn      最少链接数,那个机器链接数少就分发


#1.轮询(不做配置，默认轮询)
#2.weight权重(优先级)
#3.ip_hash配置，根据客户端ip哈希分配，不能和weight一起用
```

3.Nginx配置性能优化

1.worker_processes

定义了nginx对外提供web服务时的worker进程数。最优值取决于许多因素，包括（但不限于）CPU核的数量、存储数据的硬盘数量及负载模式。不能确定的时候，将其设置为可用的CPU内核数将是一个好的开始（设置为“auto”将尝试自动检测它）。

2.worker_rlimit_nofile

更改worker进程的最大打开文件数限制。如果没设置的话，这个值为操作系统的限制。设置后你的操作系统和Nginx可以处理比“ulimit -a”更多的文件，所以把这个值设高，这样nginx就不会有“too many open files”问题了。

3.Events模块

```
events {
      worker_connections2048;

      multi_accept on;

      use epoll;
}
```

- **worker_connections**设置可由一个worker进程同时打开的最大连接数。如果设置了上面提到的worker_rlimit_nofile，我们可以将这个值设得很高。记住，最大客户数也由系统的可用socket连接数限制（~64K），所以设置不切实际的高没什么好处。
- **multi_accept**告诉nginx收到一个新连接通知后接受尽可能多的连接。
- **use**设置用于复用客户端线程的轮询方法。如果你使用Linux 2.6+，你应该使用epoll。如果你使用*BSD，你应该使用kqueue。

HTTP 模块

```
http {

      server_tokens off;

      sendfile on;

      tcp_nopush on;

      tcp_nodelay on;

      ...

}
```

- **server_tokens**并不会让nginx执行的速度更快，但它可以关闭在错误页面中的nginx版本数字，这样对于安全性是有好处的。
- **sendfile**可以让sendfile()发挥作用。sendfile()可以在磁盘和TCP socket之间互相拷贝数据(或任意两个文件描述符)。Pre-sendfile是传送数据之前在用户空间申请数据缓冲区。之后用read()将数据从文件拷贝到这个缓冲区，write()将缓冲区数据写入网络。sendfile()是立即将数据从磁盘读到OS缓存。因为这种拷贝是在内核完成的，sendfile()要比组合read()和write()以及打开关闭丢弃缓冲更加有效(更多有关于sendfile)
- **tcp_nopush**告诉nginx在一个数据包里发送所有头文件，而不一个接一个的发送。
- **tcp_nodelay**告诉nginx不要缓存数据，而是一段一段的发送--当需要及时发送数据时，就应该给应用设置这个属性，这样发送一小块数据信息时就不能立即得到返回值。
- **access_log**设置nginx是否将存储访问日志。关闭这个选项可以让读取磁盘IO操作更快(aka,YOLO)
- **error_log**告诉nginx只能记录严重的错误：
- **keepalive_timeout**给客户端分配keep-alive链接超时时间。服务器将在这个超时时间过后关闭链接。我们将它设置低些可以让ngnix持续工作的时间更长。
- **client_header_timeout 和client_body_timeout**设置请求头和请求体(各自)的超时时间。我们也可以把这个设置低些。
- **reset_timeout_connection**告诉nginx关闭不响应的客户端连接。这将会释放那个客户端所占有的内存空间。
- **send_timeout**指定客户端的响应超时时间。这个设置不会用于整个转发器，而是在两次客户端读取操作之间。如果在这段时间内，客户端没有读取任何数据，nginx就会关闭连接。

2.WSGI

1. WSGI

1.1 WSGI相关概述

```
引子：wsgi server（比如uWSGI） 要和 wsgi application（比如django ）交互，uwsgi需要将过来的请求转给 django 处理，那么uWSGI 和 django的交互和调用就需要一个统一的规范，这个规范就是WSGI WSGI（Web Server Gateway Interface）

- WSGI，全称 Web Server Gateway Interface，或者 Python Web Server Gateway Interface ，是为 Python 语言定义的 Web 服务器和 Web 应用程序或框架之间的一种简单而通用的接口。自从 WSGI 被开发出来以后，许多其它语言中也出现了类似接口。
- WSGI 的官方定义是，the Python Web Server Gateway Interface。从名字就可以看出来，这东西是一个 Gateway，也就是网关。网关的作用就是在协议之间进行转换。
- WSGI 是作为 Web 服务器与 Web 应用程序或应用框架之间的一种低级别的接口，以提升可移植 Web 应用开发的共同点。WSGI 是基于现存的 CGI 标准而设计的
```

1.2 定义一个简版的WSGI 接口

- **一个 application函数**

```
from wsgiref.simple_server import make_server

def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return [b'<h1>Hello, web!</h1>']

httpd = make_server('', 8080, application)

print('Serving HTTP on port 8080...')
# 开始监听HTTP请求:
httpd.serve_forever()
```

- **相关分析**

```
注意:

    整个application()函数本身没有涉及到任何解析HTTP的部分，也就是说，底层代码不需要我们自己编写，

    我们只负责在更高层次上考虑如何响应请求就可以了。
    application()函数必须由WSGI服务器来调用。有很多符合WSGI规范的服务器，我们可以挑选一个来用。

    Python内置了一个WSGI服务器，这个模块叫wsgiref
    application()函数就是符合WSGI标准的一个HTTP处理函数，它接收两个参数:

        //environ: 一个包含所有HTTP请求信息的dict对象;
        //start_response: 一个发送HTTP响应的函数。

在application()函数中，调用: start_response('200 OK', [('Content-Type', 'text/html')])

就发送了HTTP响应的Header，注意Header只能发送一次，也就是只能调用一次start_response()函数。
```

start_response()函数接收两个参数，一个是HTTP响应码，一个是一组list表示的HTTP Header，每个Header用一个包含两个str的tuple表示。

通常情况下，都应该把Content-Type头发送给浏览器。其他很多常用的HTTP Header也应该发送。
然后，函数的返回值b'<h1>Hello, web!</h1>'将作为HTTP响应的Body发送给浏览器。
有了WSGI，我们关心的就是如何从environ这个dict对象拿到HTTP请求信息，然后构造HTML，通过start_response()发送Header，最后返回Body。

2. uWSGI

1.1 uWSGI的相关概述

uWSGI是一个Web服务器，它实现了WSGI协议、uwsgi、http等协议。Nginx中HttpUwsgiModule的作用是与uWSGI服务器进行交换。

- WSGI是一种通信协议。
- uwsgi同WSGI一样是一种通信协议。
- 而uWSGI是实现了uwsgi和WSGI两种协议的Web服务器。

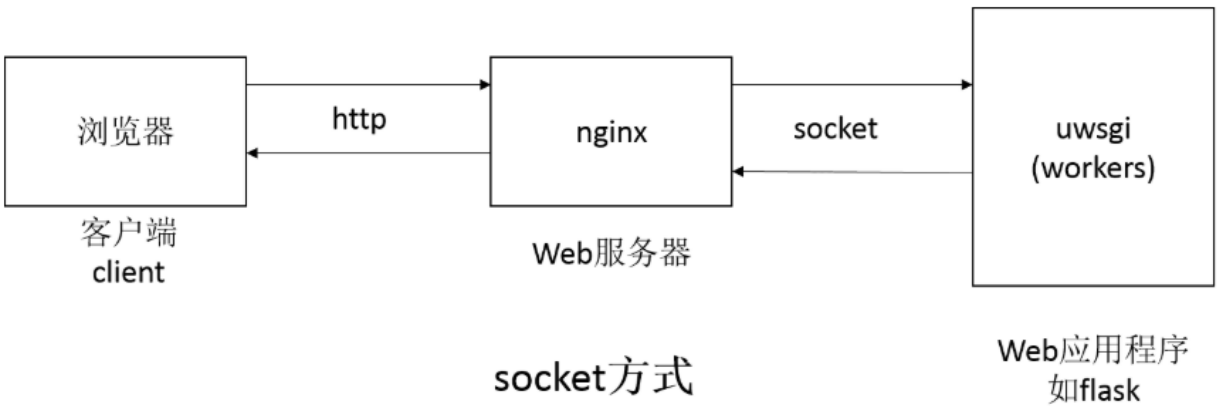
可以看作是一个应用程序，帮助我们实现WSGI协议，Http协议，这样我们可以不再关注网络通信的底层实现，将精力更多放在处理HTTP请求数据，返回HTML。利用uWIGS可以是我们的web应用得到更强的并发能力，uWIGS也可以返回静态文件(css,js,img…)，但是很笨拙，一般静态文件都交由Nginx进行传输，所以配置中一般不配置static-map,如果直接由uWIGS接受HTTP请求则需要设置http:xxxx，如果只需要与反向代理服务器进行交互则只需要接受socket，uWIGS与Nginx交互相当于两个进程间交互，一般使用的是.sock文件或者指定端口接受socket。指定端口时再使用浏览器访问相应端口，uWIGS会提示skip，跳过该HTTP请求

1.2 uWSGI配置的的理解**

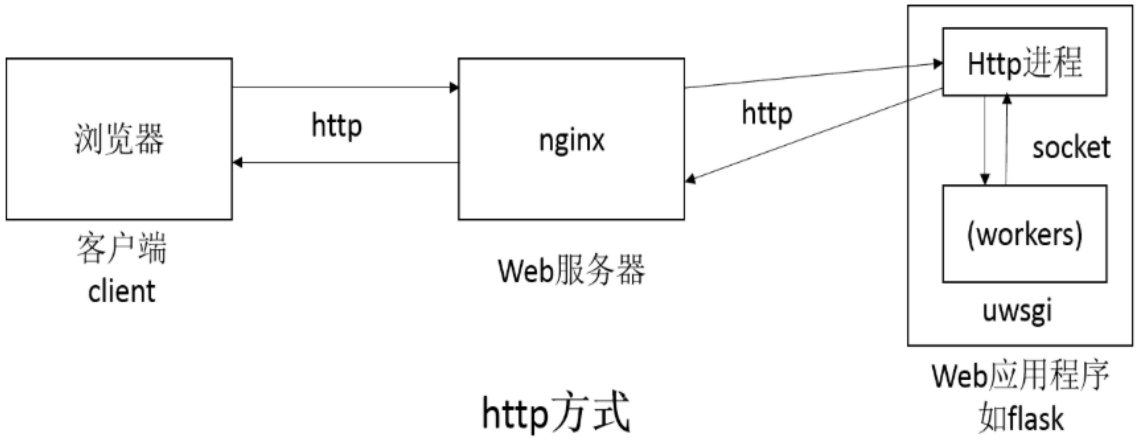
(1) 通信接口: http/http-socket/socket

socket 方式: socket = 0.0.0.0:8000

- **现在大部分web服务器（如nginx）支持uwsgi，这是这三种方式最高效的一种形式，socket通信速度会比http快，**
- 注: 指定socket协议，运行django，只能与nginx结合时



http 方式: http = 0.0.0.0:8000



http方式

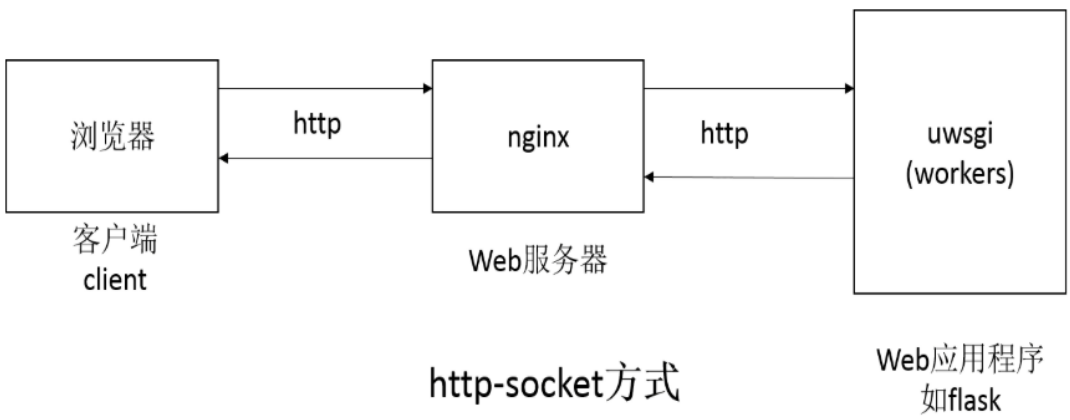
http方式

上面两个图都是http方式，使用http启动uwsgi，系统会额外启动一个http进程，从级别上来说，它和nginx是同一级别的，所以客户端和uwsgi通信，完全可以绕过nginx，不需要额外进行一个转发（如第二张图一样），但很显然，这是并不是一个很明智但选择，这样会失去了nginx很多优秀的功能。

因此: 如果你没用nginx，只想自己启动一个http界面，用这个

--- 另外 还有一方式,但是在uWSGI的配置文件中并未展示出: 具体使用待定---

http -socket 方式 http-socket=127.0.0.1:8000



http-socket方式

http-socket方式，这个适用于web服务器不支持uwsgi时。

(2) processes/workers.

表示开启多进程，根据你的应有开启合适的进程数，在一些参考资料上，可能会看到processes = 2 * cpucores或者workers = 2 * cpucores，如果应有比较简单，这样设置一般可以。如果想更合理，官方提供了uwsgitop去获得一个较为合理的值。

(3)threads & enable-threads

python中存在GIL，实际上不存在真正意义上的多线程，但是否需要，这个就根据各自但需求设定了。结合processes:

- processes=2
- threads=4

表示2个进程，每个进程中有4个线程。

- 由于GIL的存在，uwsgi索性默认不支持多线程，不对GIL进行初始化。但如果希望程序中的线程发挥作用，需要加入enable-threads=True;
- 但如果已经在多线程模式（使用 threads 选项）下，那么uWSGI将会自动启用线程支持。

(4)uid & gid & chmod-socket

uwsgi不建议使用root权限去启动uwsgi实例。可以通过root用户去运行uwsgi文件，当通过uid和gid去修改用户（移除root权限）。并且，如果你使用的是socket的通信方式，最好加上chmod-socket字段，在linux下，socket的启动方式，套接字类似文件，你必须保证有权限去读取它。

- chmod-socket=664
- uid=1000
- gid=1000

(5)master

意味着启动一个master主进程来管理其他进程，建议启动这个进程，在管理的时候比较方便；如果kill这个master进程，相当于关闭所有的uwsgi进程

3. uwsgi

概述: uwsgi是服务器和服务端应用程序的通信协议，规定了怎么把请求转发给应用程序和返回

关于 uwsgi的理解

本文根据代码阅读以及参照多种文档，描述了uwsgi的启动多进程+多线程工作原因，以及thunder_lock参数的作用：

- uwsgi是用c语言写的一个webserver，可以启动**多个进程，进程里面可以启动多个线程来服务。**进程分为主进程和worker进程，worker里面可以有多个线程。
- **main函数**，启动这个就是主进程了
- **uwsgi_setup函数**

    - 主进程里面针对选项参数做一些处理，执行环境设置，执行一些hook，语言环境初始化（python），如果没有设置延迟加载app，则app在主进程加载；如果设置了延迟加载，则在每一个worker进程中都会加载一次。
    - 还执行了插件的初始化（当前我只关心http、python：http是gateway类型的插件，这种插件是向外暴露http服务的，python的requests类型的插件，用来服务请求的）、**tcp socket的绑定与监听（这个是指http与work之间的通信，且它的端口是自动产生的）。**
    - 最后uwsgi主进程fork了指定worker进程用来接收(accept)请求。虽然在setup中就fork了子进程，但是现在还没有开始accept。
- **wsgi_run函数** 就是真正的开始执行了，这个函数分为俩个部分，
    - **主进程执行部分**是一个无限循环，他可以执行特定的hook以及接收信号等，总之是用来**管理worker进程以及一些定时或者事件触发任务。**
    - **worker部分**：注册型号处理函数，执行一些hook，循环接收(accept)请求。在启动woker时可以根据--threads参数指定要产生的线程个数，否则只在当前进程启动一个线程。这些线程循环接收请求并处理。
- 在worker中接收请求的函数**wsgi_req_accept**有一个锁：**thunder_lock**，这个锁用来串行化accept，防止"惊群"现象：
- 现在这样的情况：主进程绑定并监听socket，然后调用fork，在各个子进程进行accept。无论任何时候，只要有一个连接尝试连接，所有的子进程都将被唤醒，但只有一个会连接成功，其他的会得到一个EAGAIN的错误，这将导致巨大的CPU资源浪费，如果在进程中使用线程，这个问题被再度放大。一个解决方法是串行化accept，在accept前防止一个锁。

3..supervisor是什么？如何使用?

```
#后台管理进程任务的  -->你本来自己手动敲的命令,现在交给supervisor去管理
使用:
1.安装 easy_install supervisor
2.生成配置文件 echo_supervisord_conf > /etc/supervisor.conf
3.写入自定义的配置
[program:crm] ; 项目名称
command=/root/Envs/knight/bin/uwsgi –ini /opt/knight/uwsgi.ini ;启动项目的命令
stopasgroup=true ;默认为false,进程被杀死时，是否向这个进程组发送stop信号，包括子进程
killasgroup=true ;默认为false，向进程组发送kill信号，包括子进程
4.启动supervisor服务
supervisord -c /etc/supervisor.conf
5.启动所有项目
supervisorctl -c /etc/supervisor.conf start all
```

2.3 项目流程

其实网上很多教程，都是关于uwsgi+nginx部署django的，StackOverflow也有一些解决常见错误的方法，但是部署还是容易出问题，新手难解决。 归根到底是自己不了解整个项目的流程。教程都只教方法，但为什么这样部署，这样部署有什么好处，每个组件都起什么作用却只字不提。致使只要部署稍微有那么一点不同，就无可是从了。 所以说，项目流程和每个组件的用途才是此次部署最重要的部分。

首先客户端请求服务资源，
nginx作为直接对外的服务接口，接收到客户端发送过来的http请求，会解包、分析，
如果是静态文件请求就根据nginx配置的静态文件目录，返回请求的资源，
如果是动态的请求，nginx就通过配置文件，将请求传递给uWSGI；uWSGI 将接收到的包进行处理，并转发给wsgi，
wsgi根据请求调用django工程的某个文件或函数，处理完后django将返回值交给wsgi，
wsgi将返回值进行打包，转发给uWSGI，
uWSGI接收后转发给nginx，nginx最终将返回值返回给客户端(如浏览器)。
*注:不同的组件之间传递信息涉及到数据格式和协议的转换

作用:

1. 第一级的nginx并不是必须的，uwsgi完全可以完成整个的和浏览器交互的流程；
2. 在nginx上加上安全性或其他的限制，可以达到保护程序的作用；
3. uWSGI本身是内网接口，开启多个work和processes可能也不够用，而nginx可以代理多台uWSGI完成uWSGI的负载均衡；
4. django在debug=False下对静态文件的处理能力不是很好，而用nginx来处理更加高效。