说明： 文档包括了三个部分， zookeeper，I\O 模式以及 uWSGI。图片都是对应上下文内容，引用的图片都有注明，如未注明则为自己所画。例如 输出，代码，配置会用不同字体表示。

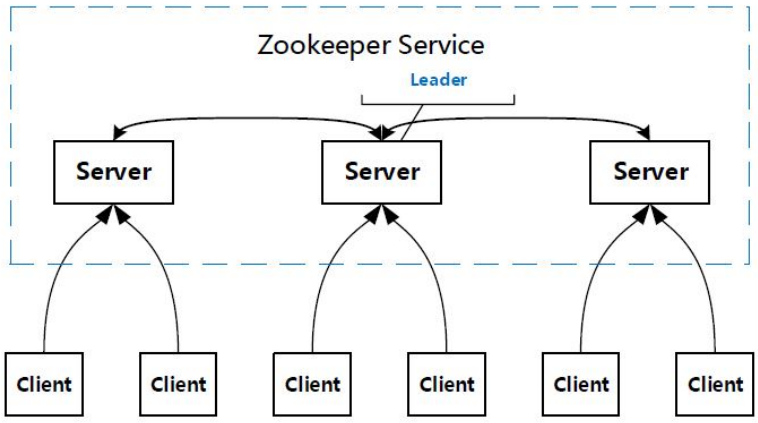
zookeeper：

- 整体把握：

  zk 是针对分布式系统的协调服务（本身就是分布式应用程序），优点是可靠，可扩展，高性能。

  遵循 C/S 模型。(这里 C 就是我们使用 zk 服务的机器，S 自然就是提供 zk 服务)。

  zk 可以提供单机服务，也可组成集群提供服务，还支持伪集群方式(一台物理机运行多个 zookeeper 实例）。客户端连接到一个单独的服务。客户端保持了一个 TCP 连接，通过这个 TCP 连接发送请求、获取响应、获取 watch 事件、和发送心跳。如果这个连接断了，会自动连接到其他不同的服务器。



zookeeper C/S 模型

  zk 的数据模型类似文件系统，由 znode 组成目录树的形式，每个节点下可以有子节点。

  节点可以是以下四种类型：

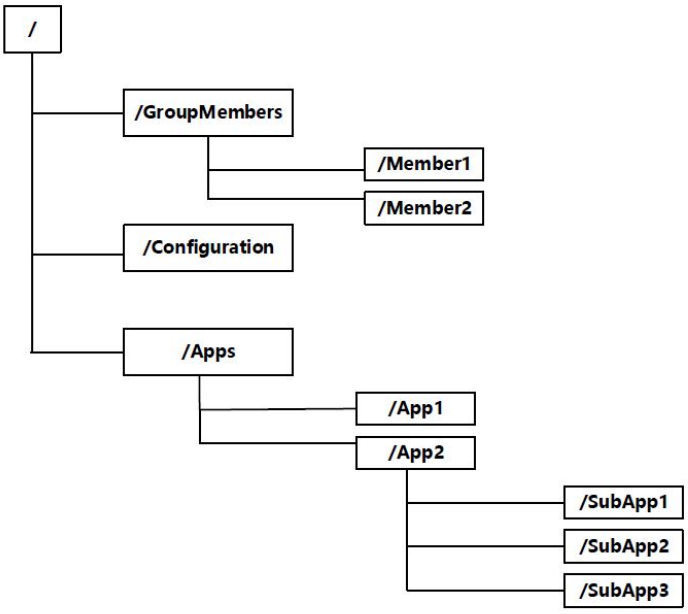
PERSISTENT：持久化目录节点，这个目录节点存储的数据不会丢失；

PERSISTENT_SEQUENTIAL：顺序自动编号的目录节点，这种目录节点会根据当前已近存在的节点数自动加 1，然后返回给客户端已经成功创建的目录节点名；

EPHEMERAL：临时目录节点，一旦创建这个节点的客户端与服务器端口也就是 session 超时，这种节点会被自动删除；

EPHEMERAL_SEQUENTIAL：临时自动编号节点。

监控节点变化时，可以监控一个节点的变化，也可以监控一个节点所有子节点的变化。zk 一些很重要的应用都是依赖这些节点的特性。



Znode 目录树

- zk 主要应用：

配置管理（Configuration Management）

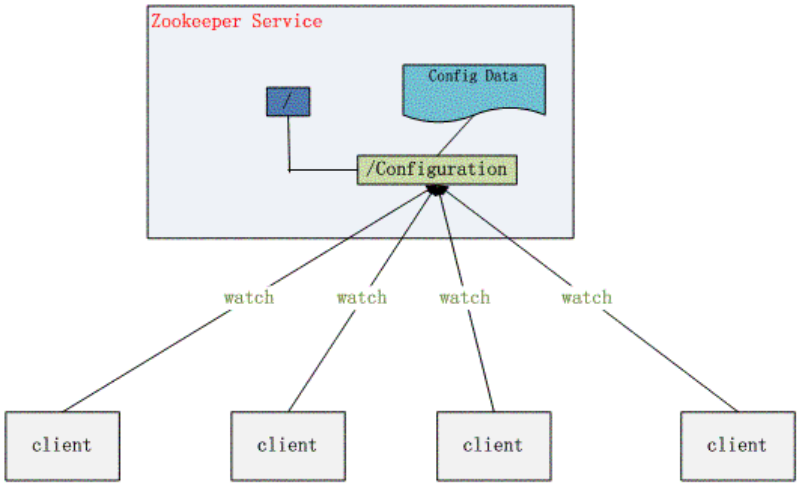
配置的管理在分布式应用环境中很常见，例如同一个应用系统需要多台 Server 运行，但是它们运行的应用系统的某些配置项是相同的，如果要修改这些相同的

配置项，那么就必须同时修改每台运行这个应用系统的 Server，这样非常麻烦而且容易出错。

像这样的配置信息完全可以交给 Zookeeper 来管理，将配置信息保存在 Zookeeper 的某个目录节点中，然后将所有需要修改的应用机器监控配置信息的状态，一旦配置信息发生变化，每台应用机器就会收到 Zookeeper 的通知，然后从 Zookeeper 获取新的配置信息应用到系统中。

配置管理结构图

（引用 https://www.ibm.com/developerworks/cn/opensource/os-cn-zookeeper/）



集群管理（Group Membership）

Zookeeper 能够很容易的实现集群管理的功能，如有多台 Server 组成一个服务集群，那么必须要一个“总管”知道当前集群中每台机器的服务状态，一旦有机器不能提供服务，集群中其它集群必须知道，从而做出调整重新分配服务策略。同样当增加集群的服务能力时，就会增加一台或多台 Server，同样也必须让“总管”知道。

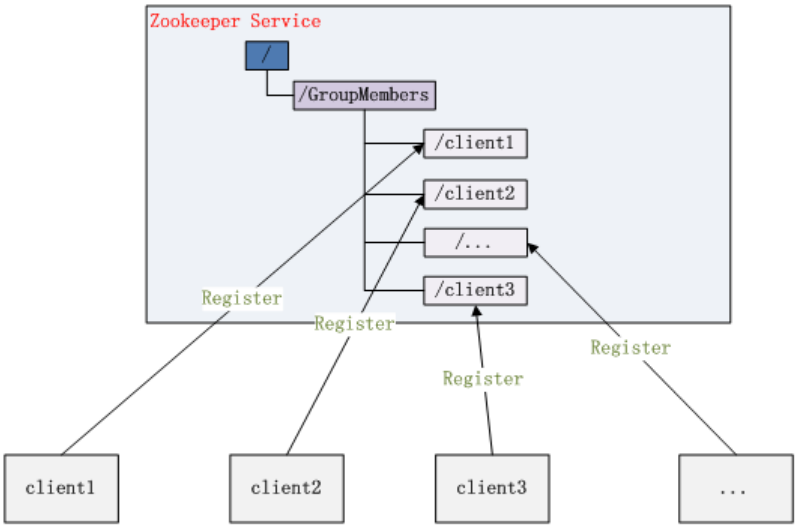
Zookeeper 不仅能够帮你维护当前的集群中机器的服务状态，而且能够帮你选出一个“总管”，让这个总管来管理集群，这就是 Zookeeper 的另一个功能 Leader Election。

它们的实现方式都是在 Zookeeper 上创建一个 EPHEMERAL 类型的目录节点，然后每个 Server 在它们创建目录节点的父目录节点上调用 getChildren(String path, boolean watch) 方法并设置 watch 为 true，由于是 EPHEMERAL 目录节点，当创建它的 Server 死去，这个目录节点也随之被删除，所以 Children 将会变化，这时 getChildren 上的 Watch 将会被调用，所以其它 Server 就知道已经有某台 Server 死去了。新增 Server 也是同样的原理。

Zookeeper 如何实现 Leader Election，也就是选出一个 Master Server。和前面的一样每台 Server 创建一个 EPHEMERAL 目录节点，不同的是它还是一个 SEQUENTIAL 目录节点，所以它是个 EPHEMERAL_SEQUENTIAL 目录节点。之所以它是 EPHEMERAL_SEQUENTIAL 目录节点，是因为我们可以给每台 Server 编号，我们可以选择当前是最小编号的 Server 为 Master，假如这个最小编号的 Server 死去，由于是 EPHEMERAL 节点，死去的 Server 对应的节点也被删除，所以当前的节点列表中又出现一个最小编号的节点，我们就选择这个节点为当前 Master。这样就实现了动态选择 Master，避免了传统意义上单 Master 容易出现单点故障的问题

集群管理结构图

(引用 https://www.ibm.com/developerworks/cn/opensource/os-cn-zookeeper/)



共享锁（Locks）

共享锁在同一个进程中很容易实现，但是在跨进程或者在不同 Server 之间就不好实现了。Zookeeper 却很容易实现这个功能，实现方式也是需要获得锁的 Server 创建一个 EPHEMERAL_SEQUENTIAL 目录节点，然后调用 getChildren 方法获取当前的目录节点列表中最小的目录节点是不是就是自己创建的目录节点，如果正是自己创建的，那么它就获得了这个锁，如果不是那么它就调用 exists(String path, boolean watch) 方法并监控 Zookeeper 上目录节点列表的变化，一直到自己创建的节点是列表中最小编号的目录节点，从而获得锁，释放锁很简单，只要删除前面它自己所创建的目录节点就行了。

队列管理

Zookeeper 可以处理两种类型的队列：

1. 当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达，这种是同步队列。
2. 队列按照 FIFO 方式进行入队和出队操作，例如实现生产者和消费者模型。

创建一个父目录 /synchronizing，每个成员都监控标志（Set Watch）位目录 /synchronizing/start 是否存在，然后每个成员都加入这个队列，加入队列的方

式就是创建 /synchronizing/member_i 的临时目录节点，然后每个成员获取 /synchronizing 目录的所有目录节点，也就是 member_i。判断 i 的值是否已经是成员的个数，如果小于成员个数等待 /synchronizing/start 的出现，如果已经相等就创建 /synchronizing/start。

FIFO 队列用 Zookeeper 实现思路如下：

在特定的目录下创建 SEQUENTIAL 类型的子目录 /queue_i ,这样就能保证所有成员加入队列时都是有编号的，出队列时通过 getChildren( ) 方法可以返回当前所有的队列中的元素，然后消费其中最小的一个，这样就能保证 FIFO。

- **实践部分**

mac 下用 brew install zookeeper

在集合体中，可以包含一个节点，但它不是一个高可用和可靠的系统。如果在集合体中有两个节点，那么这两个节点都必须已经启动并让服务正常运行，因为两个节点中的一个并不是严格意义上的多数。如果在集合体中有三个节点，即使其中一个停机了，您仍然可以获得正常运行的服务（三个中的两个是严格意义上的多数）。出于这个原因，ZooKeeper 的集合体中通常包含奇数数量的节点 ,因为就容错而言 ,与三个节点相比，四个节点并不占优势，因为只要有两个节点停机，ZooKeeper 服务就会停止。在有五个节点的集群上，需要三个节点停机才会导致 ZooKeeper 服务停止运作。

本机 3 个 zk 实例 伪集群配置：

在 usr/local/var/run/zookeeper/ 下新建文件夹 zk1, zk2, zk3

在 /usr/local/etc/zookeeper 下 新建 zk1.cfg, zk2.cfg, zk3.cfg

- 修改项：

dataDir=/usr/local/var/run/zookeeper/zk1

the port at which the clients will connect

clientPort=2181

server.1=localhost:2888:3888

server.2=localhost:2889:3889

server.3=localhost:2890:3890

在文件夹 /usr/local/var/run/zookeeper/zk1 下 echo "1" >

/usr/local/var/run/zookeeper/zk1/myid

zk2 下 echo "2" > myid　　zk3 下 echo "3" > myid;　　myid 里面的值与 配置

文件里的 server. 编号 一致

依次启动：zkServer start /usr/local/etc/zk1.cfg

zkServer start /usr/local/etc/zk2.cfg

zkServer start /usr/local/etc/zk3.cfg

检查状态：zkServer status /usr/local/etc/zk1.cfg

输出 ：

ZooKeeper JMX enabled by default

Using config: /usr/local/etc/zookeeper/zk1.cfg

Mode: follower

连接 zk 服务器： zkCli -server 127.0.0.1:2181

输出：

Connecting to 127.0.0.1:2181

Welcome to ZooKeeper!

JLine support is enabled

WATCHER::

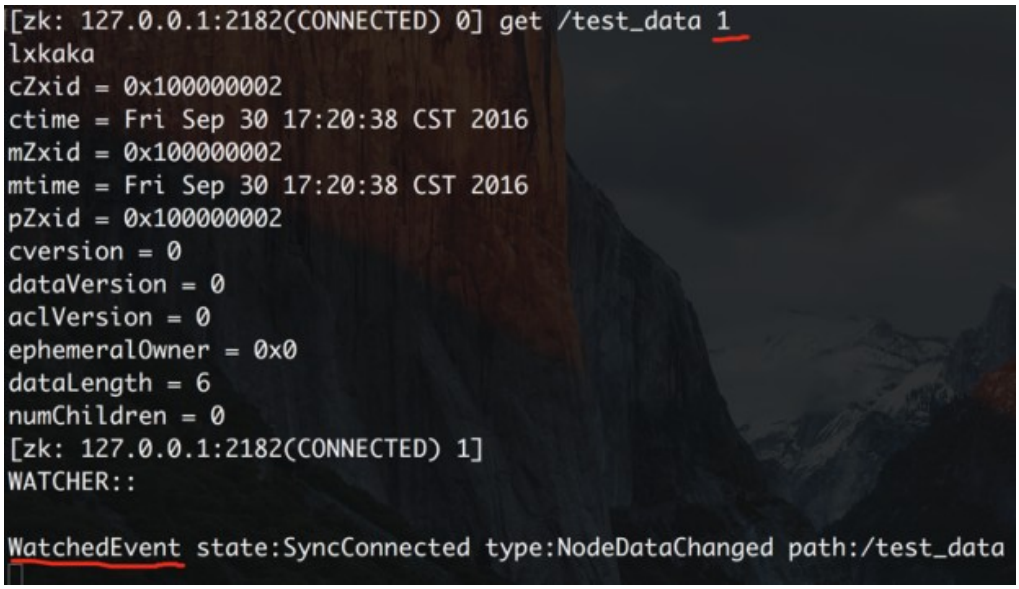
WatchedEvent state:SyncConnected type:None path:null

创建一个新 znode 和相关联的数据:　create /test_data　test1234

用 get 获取数据： get /test_data； 用 set 修改： set/test_data abcd1234

连接到其他 zk 服务器，可获取到同样的数据： get /test_data 1

这里在最后提供了一个可选参数 1。此参数为 /tes_data 上的数据设置了一个一次性的触发器（名称为 *watch*）。如果另一个客户端在 /test_data 上修改数据，该客户端将会获得一个异步通知。该通知只发送一次，除非 watch 被重新设置，否则不会因数据发生改变而再次发送通知。



利用 python 实现的 zookeeper 客户端 kazoo 与 zk server 交换，简单示例

- 创建临时自动编号节点

```python
# -*- coding: utf-8 -*-
import logging
from kazoo.client import KazooClient
from time import sleep

logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)


def watch_func(child):
    print 'action triggered:', child

hosts_list = ['127.0.0.1:2181', '127.0.0.1:2182', '127.0.0.1:2183']
zk = KazooClient(hosts=hosts_list[0])
zk.start()
print zk.state
# 创建一个目录节点
zk.create('/seq_test')
#创建一个临时自动编号子目录节点
zk.create('/seq_test/test_i', ephemeral=True, sequence=True)
print zk.get('/seq_test')
# 获取子节点信息
child = zk.get_children('/seq_test')
print child
```

- 定义一个 watch event， watch 一个节点的变化

```python
def watch_func(child):
    print 'action triggered:', child

zk.create('/election')
child = zk.get_children('/election', watch=watch_func)
zk.create('/election/child')
zk.delete('/election/child')
#print zk.state
while True:
    sleep(2)
```

```
INFO:Connecting to 127.0.0.1:2182
DEBUG:Sending request(xid=None): Connect(protocol_version=0, last_zxid_seen=0, time
INFO:Zookeeper connection established, state: CONNECTED
DEBUG:Sending request(xid=1): GetChildren(path='/election', watcher=<function watch
DEBUG:Received response(xid=1): [u'child']
DEBUG:Sending request(xid=2): Delete(path='/election/child', version=-1)
CONNECTED
action triggered: WatchedEvent(type='CHILD', state='CONNECTED', path=u'/election')
DEBUG:Received EVENT: Watch(type=4, state=3, path=u'/election')
DEBUG:Received response(xid=2): True
```

I\O 模式， select，poll,epoll 总结：

1. 异步同步，阻塞非阻塞梳理：

- 总结：

  同步：发起一个调用，得到结果才返回。

  异步：调用发起后，调用直接返回；调用方主动询问被调用方获取结果，或被调用方通过回调函数返回。

  阻塞：调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。

  非阻塞：调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

  同步才有阻塞和非阻塞之分

  (通俗的理解)

  阻塞与非阻塞关乎如何对待事情产生的结果。

  阻塞：不等到想要的结果我就不走了。

2. 首先明确进程的状态

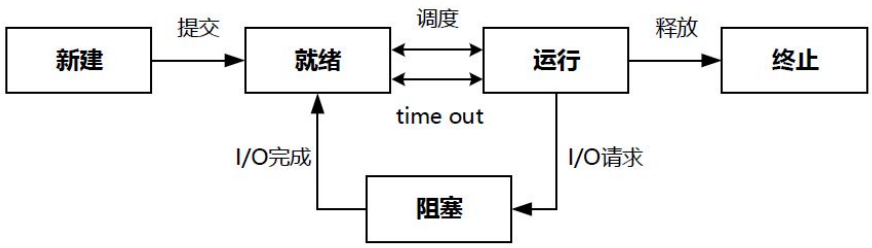
- 进程状态转换

  就绪状态 -> 运行状态：处于就绪状态的进程被调度后，获得 CPU 资源（分派 CPU 时间片），于是进程由就绪状态转换为运行状态。

  运行状态 -> 就绪状态 :处于运行状态的进程在时间片用完后 ,不得不让出 CPU，从而进程由运行状态转换为就绪状态。此外，在可剥夺的操作系统中，当有更高优先级的进程就 、 绪时，调度程度将正执行的进程转换为就绪状态，让更高优先级的进程执行。

运行状态 -> 阻塞状态：当进程请求某一资源（如外设）的使用和分配或等待某一事件的发生（如 I/O 操作的完成）时，它就从运行状态转换为阻塞状态。进程以系统调用的形式请求操作系统提供服务，这是一种特殊的、由运行用户态程序调用操作系统内核过程的形式。

阻塞状态 -> 就绪状态：当进程等待的事件到来时，如 I/O 操作结束或中断结束时，中断处理程序必须把相应进程的状态由阻塞状态转换为就绪状态。



进程状态转换图

3. 从操作系统层面执行应用程序理解 IO 模型

阻塞 I/O 模型：

简介：进程会一直阻塞，直到数据拷贝完成

应用程序调用一个 IO 函数，导致应用程序阻塞，等待数据准备好。 如果数据没有准备好，一直等待....数据准备好了，从内核拷贝到用户空间，IO 函数返回成功指示。

我们 第一次接触到的网络编程都是从 listen()、send()、recv()等接口开始的。使用这些接口可以很方便的构建服务器 /客户机的模型。

阻塞 I/O 模型图：在调用 recv()/recvfrom（ ）函数时，发生在内核中等待数据和复制数据的过程。

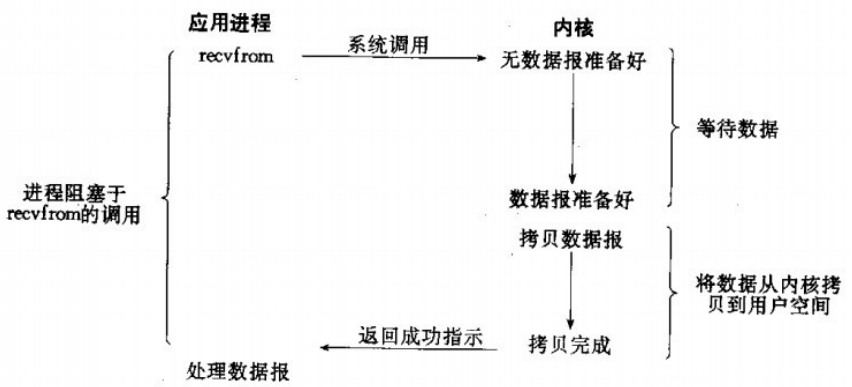


图 6.1 阻塞 I/O 模型

（引用 http://www.cnblogs.com/nufangrensheng/p/3588690.html）

当调用 recv()函数时，系统首先查是否有准备好的数据。如果数据没有准备好，那么系统就处于等待状态。当数据准备好后，将数据从系统缓冲区复制到用户空间，然后该函数返回。在套接应用程序中，当调用 recv()函数时，未必用户空间就已经存在数据，那么此时 recv()函数就会处于等待状态。

阻塞模式给网络编程带来了一个很大的问题，如在调用 send()的同时，线程将被阻塞，在此期间，线程将无法执行任何运算或响应任何的网络请求。这给多客户机、多业务逻辑的网络编程带来了挑战。这时，我们可能会选择多线程的方式来解决这个问题。

应对多客户机的网络应用，最简单的解决方式是在服务器端使用多线程（或多进程）。多线程（或多进程）的目的是让每个连接都拥有独立的线程（或进程），这样任何一个连接的阻塞都不会影响其他的连接。

具体使用多进程还是多线程，并没有一个特定的模式。传统意义上，进程的开销要远远大于线程，所以，如果需要同时为较多的客户机提供服务，则不推荐使用多进程；如果单个服务执行体需要消耗较多的 CPU 资源，譬如需要进行大规模或长时间的数据运算或文件访问，则进程较为安全。

非阻塞 IO 模型 ：

简介：非阻塞 IO 通过进程反复调用 IO 函数（多次系统调用，并马上返回）；在数据拷贝的过程中，进程是阻塞的；

我们把一个 SOCKET 接口设置为非阻塞就是告诉内核，当所请求的 I/O 操作无法完成时，不要将进程睡眠，而是返回一个错误。这样我们的 I/O 操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用 CPU 的时间。

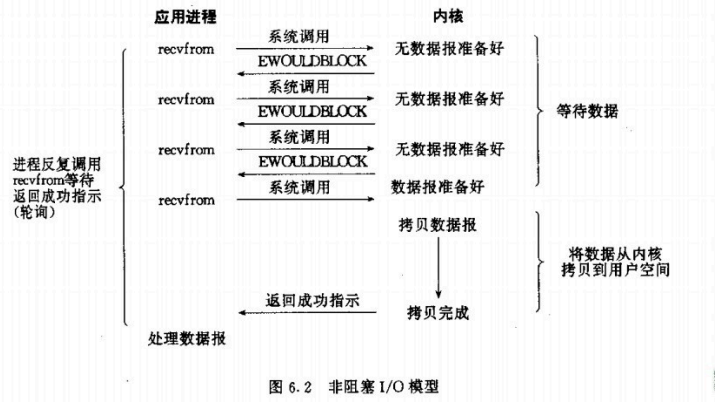


图 6.2 非阻塞 I/O 模型

（引用 http://www.cnblogs.com/nufangrensheng/p/3588690.html）

IO 复用模型：

IO multiplexing 就是我们说的 select，poll，epoll，有些地方也称这种 IO 方式为 event driven IO。select/epoll 的好处就在于单个 process 就可以同时处理多

个网络连接的 IO。它的基本原理就是 select，poll，epoll 这个 function 会不断的轮询所负责的所有 socket，当某个 socket 有数据到达了，就通知用户进程。

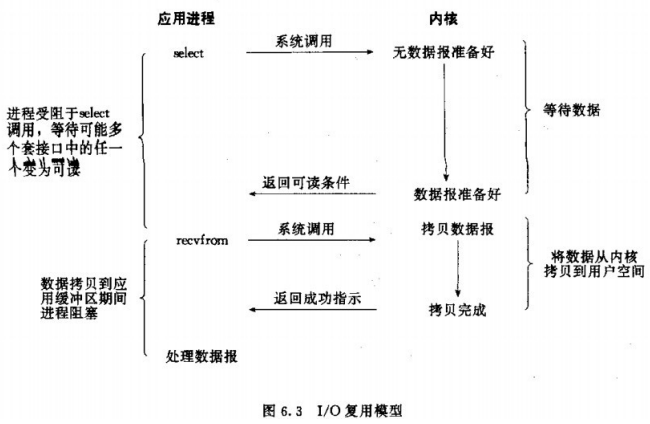


图 6.3 I/O 复用模型

（引用 http://www.cnblogs.com/nufangrensheng/p/3588690.html）

当用户进程调用了 select，那么整个进程会被 block，而同时，kernel 会“监视”所有 select 负责的 socket，当任何一个 socket 中的数据准备好了，select 就会返回。这个时候用户进程再调用 read 操作，将数据从 kernel 拷贝到用户进程。所以，I/O 多路复用的特点是通过一种机制一个进程能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。

异步 IO 模型

用户进程发起 read 操作之后，立刻就可以开始去做其它的事。而另一方面，从 kernel 的角度，当它受到一个 asynchronous read 之后，首先它会立刻返回，所以不会对用户进程产生任何 block。然后，kernel 会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel 会给用户进程发送一个 signal，告诉它 read 操作完成了。

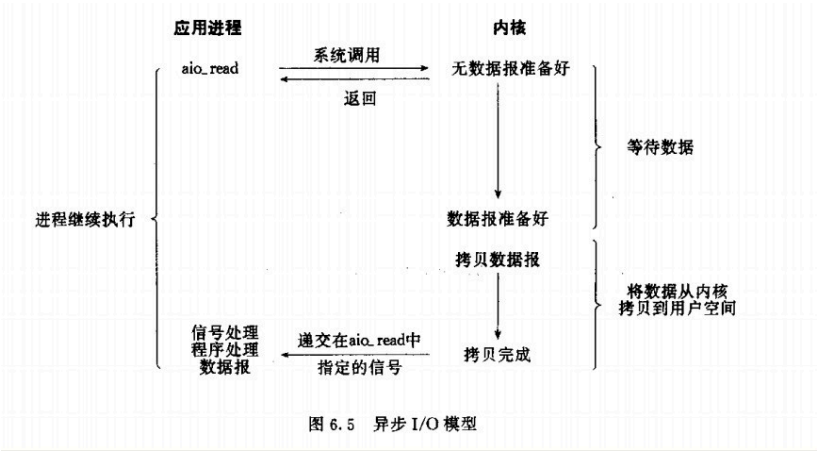


图 6.5 异步 I/O 模型

（引用 http://www.cnblogs.com/nufangrensheng/p/3588690.html）

4. I\O 复用中的 select，poll, epoll

select

```
int select (int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
timeval *timeout);
```

select 函数监视的文件描述符分 3 类，分别是 writefds、readfds、和 exceptfds。调用后 select 函数会阻塞，直到有描述副就绪（有数据 可读、可写、或者有 except），或者超时(timeout 指定等待时间 ,如果立即返回设为 null 即可）,函数返回。当 select 函数返回后，可以 通过遍历 fdset，来找到就绪的描述符

poll

```
int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

不同与 select 使用三个位图来表示三个 fdset 的方式，poll 使用一个 pollfd 的指针实现。pollfd 并没有最大数量限制（但是数量过大后性能也是会下降）。 和 select 函数一样，poll 返回后，需要轮询 pollfd 来获取就绪的描述符。

epoll

epoll 是通过事件的就绪通知方式，调用 epoll_create 创建实例，调用 epoll_ctl 添加或删除监控的文件描述符，调用 epoll_wait 阻塞住，直到有就绪的文件描述符，通过 epoll_event 参数返回就绪状态的文件描述符和事件。

epoll 操作过程需要三个接口，分别如下：

```
int epoll_create(int size)；//创建一个 epoll 的句柄，size 用来告诉内核这个监听的数目一共有多大
```

生成一个 epoll 专用的文件描述符，其实是申请一个内核空间，用来存放想关注的 socket fd 上是否发生以及发生了什么事件。

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)；
```

控制某个 epoll 文件描述符上的事件：注册、修改、删除。其中参数 epfd 是 epoll_create() 创建 epoll 专用的文件描述符。

```
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

等待 I/O 事件的发生；返回发生事件数。参数说明：

epfd: 由 epoll_create() 生成的 epoll 专用的文件描述符；

epoll_event: 用于回传代处理事件的数组；

maxevents: 每次能处理的事件数；

timeout: 等待 I/O 事件发生的超时值；

区别总结：

（1）select，poll 实现需要自己不断轮询所有 fd 集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而 epoll 其实也需要调用 epoll_wait 不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪 fd 放入就绪链

表中，并唤醒在 epoll_wait 中进入睡眠的进程。虽然都要睡眠和交替，但是 select 和 poll 在“醒着”的时候要遍历整个 fd 集合，而 epoll 在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的 CPU 时间。这就是回调机制带来的性能提升。

（2）select，poll 每次调用都要把 fd 集合从用户态往内核态拷贝一次，epoll 通过 mmap 把内核空间和用户空间映射到同一块内存，省去了拷贝的操作。

tornado 框架：

- 使用单线程的方式，避免线程切换的性能开销，同时避免在使用一些函数接口时出现线程不安全的情况
- 支持异步非阻塞网络 IO 模型，避免主进程阻塞等待。

tornado 的 IOLoop 模块 是异步机制的核心，它包含了一系列已经打开的文件描述符和每个描述符的处理器（handlers）。这些 handlers 就是对 select， poll，epoll 等的封装。（所以本质上说是 IO 复用）

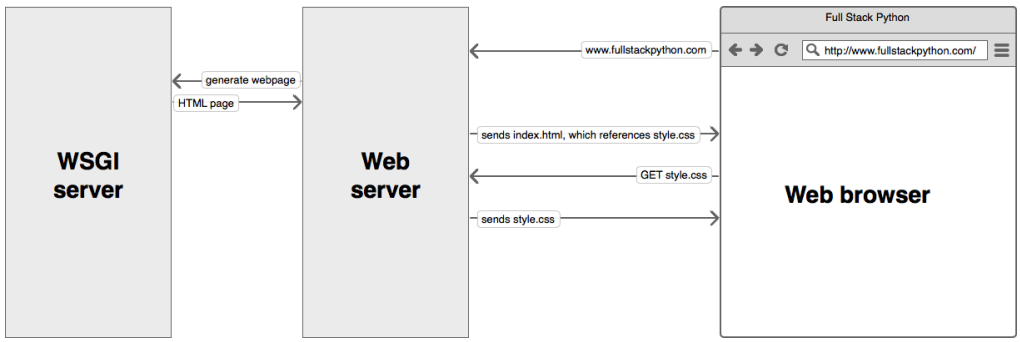
Django 就没有用异步，通过使用多线程的 WSGI server（比如 uWSGI）来实现并发，这也是 WSGI 普遍的做法。

uWSGI— 一个 WSGI server：

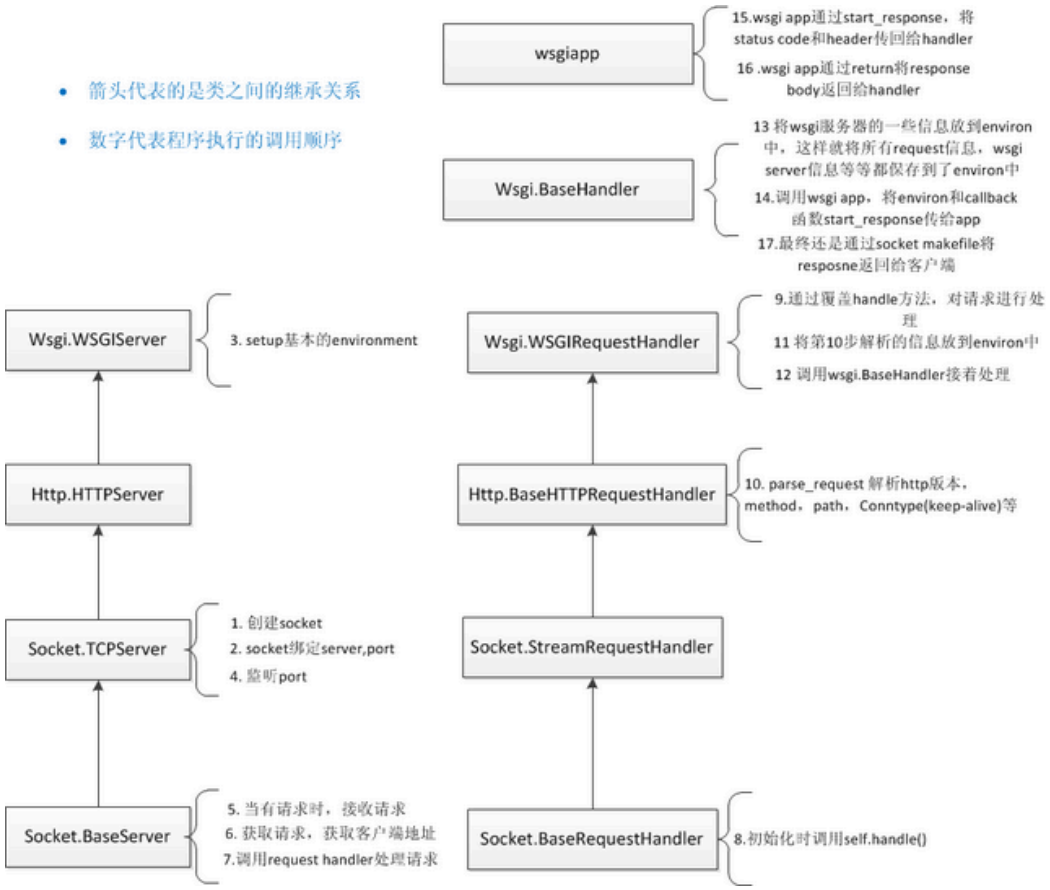
- 相关背景知识，从 WSGI 说起：

  Web 应用框架的选择将限制可用的 Web 服务器的选择，反之亦然。那时的 Python 应用程序通常是为 CGI，FastCGI，mod_python 中的一个而设计，甚至是为特定 Web 服务器的自定义的 API 接口而设计的。WSGI 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。WSGI 区分为两个部分：一为“服务器”或“网关”，另一为“应用程序”或“应用框架”。在处理一个 WSGI 请求时，服务器会为应用程序提供环境信息及一个回调函数(Callback Function)。当应用程序完成处理请求后，通过前述的回调函数，将结果回传给服务器。

  （引用 https://www.fullstackpython.com/wsgi-servers.html）



  wsgi server 可以理解为一个符合 wsgi 规范的 web server，接收 request 请求，封装一系列环境变量，按照 wsgi 规范调用注册的 wsgi app，最后将 response 返回给客户端。下图总结了 WSGI 的调用关系。



（引用 http://network.chinabyte.com/45/13151045.shtml）

a. 服务器创建 socket，监听端口，等待客户端连接。

b. 当有请求来时，服务器解析客户端信息放到环境变量 environ 中，并调用绑定的 handler 来处理请求。

c. handler 解析这个 http 请求，将请求信息例如 method，path 等放到 environ 中。

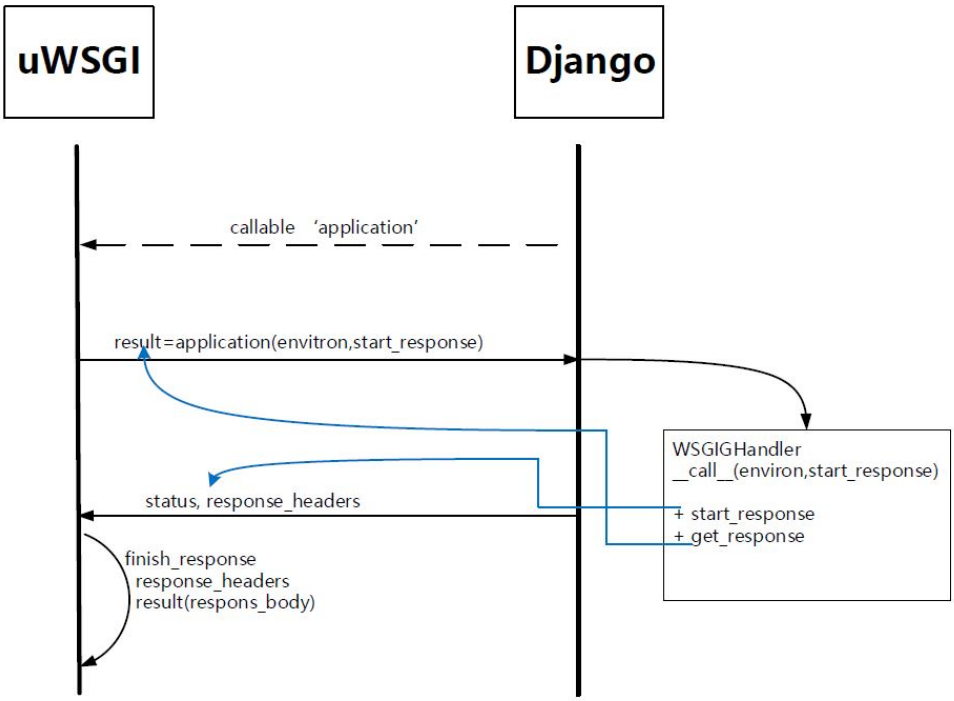
d. wsgi handler 再将一些服务器端信息也放到 environ 中 ,最后服务器信息，客户端信息，本次请求信息全部都保存到了环境变量 environ 中。

e. wsgi handler 调用注册的 wsgi app，并将 environ 和回调函数（start_response)传给 wsgi app

f. wsgi app 将 reponse header/status/body 回传给 wsgi handler

g. 最终 handler 还是通过 socket 将 response 信息塞回给客户端。

下图具体说明了 uWSGI 和 framework 比如 Django 的交互流程。



uWSGI 与 web framework 交互

CGI：外部应用程序（CGI 程序）与 Web 服务器之间的接口标准，是在 CGI 程序和 Web 服务器之间传递信息的规程。简单点说 CGI 就是规定 Web server 要传哪些数据、以什么样的格式传递给 CGI 程序。CGI 方式在遇到连接请求（用户请求）先要创建 cgi 的子进程，激活一个 CGI 进程，然后处理请求，处理完后结束这个子进程。这就是 fork-and-execute 模式。所以用 cgi 方式的服务器有多少连接请求就会有多少 cgi 子进程，子进程反复加载是 cgi 性能低下的主要原因。当用户请求数量非常多时，会大量挤占系统的资源如内存，CPU 时间等，造成效能低下。

FASTCGI: FastCGI 是从 CGI 发展改进而来的。Fastcgi 会先启一个 master，解析配置文件，初始化执行环境，然后再启动多个 worker。当请求过来时，master

会传递给一个 worker ,然后立即可以接受下一个请求。这样就避免了重复的劳动，效率自然是高。而且当 worker 不够用时，master 可以根据配置预先启动几个 worker 等着。

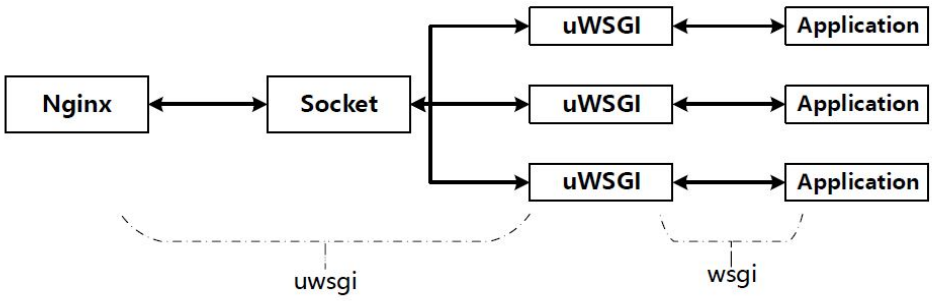
- uWSGI

  uWSGI 是一个 Web 服务器 ,它实现了 WSGI 协议、uwsgi、http 等协议。uWSGI，既不用 wsgi 协议也不用 FastCGI 协议，而是自创了一个 uwsgi 的协议，uwsgi 协议是一个 uWSGI 服务器自有的协议，

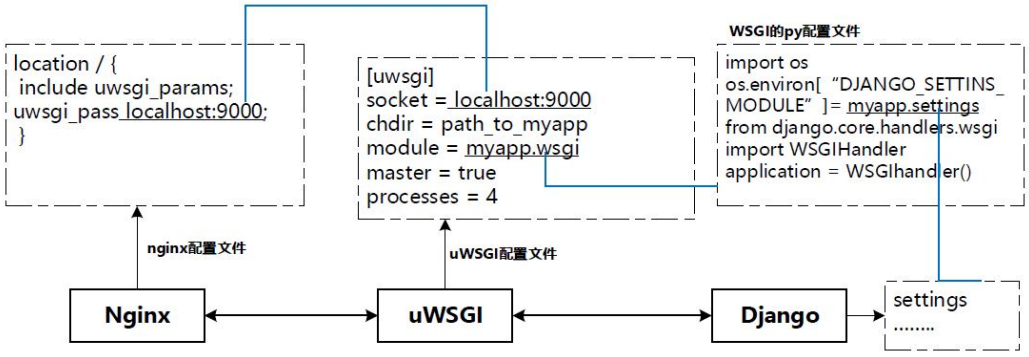
  它是一个二进制协议,能够携带任意类型的信息，每一个 uwsgi packet 前 4byte 为传输信息类型描述，它与 WSGI 相比是两样东西。

  uWSGI 保留了 fastcgi 的优点，实现进程控制，预先设置好启动多个 worker 处理请求。

  用下面这两张图展示 uWSGI 在系统中的位置，作用和相关的配置调用关系。



相关配置以及流程



- Nginx 和 uWSGI 的重要配置：

  nginx 通过 ngx_http_uwsgi_module 模块把请求传递个 uWSGI 服务器。

  示例配置

```
location / {
        include uwsgi_params;
        uwsgi_pass localhost:9000;
 }
```

  常用配置参数：

  - `uwsgi_pass [protocol://]address;`

    设置 uwsgi 服务器的协议和地址，协议可是 uwsgi 或 suwsgi（uwsgi over ssl）； 地址可以是 ip 地址，域名，和可选的端口：

    `uwsgi_pass localhost:9000;`

    `uwsgi_pass uwsgi://localhost:9000;`

    `uwsgi_pass suwsgi://[2001:db8::1]:9090;`

    也可是 unix socket：

    `uwsgi_pass unix:/tmp/uwsgi.socket;`

- `uwsgi_read_timeout time;`

  Default:`uwsgi_read_timeout 60s;`

  定义从 uwsgi 服务器读取响应的超时时间，如果在超时时间内 uwsgi 服务器没有传输任何东西，连接会被断开。

- `uwsgi_send_timeout time;`

  Default:`uwsgi_send_timeout 60s;`

  定义向 uwsgi 服务器传输请求的超时时间，如果在超时时间内 uwsgi 服务器没有收到任何东西，连接会被断开。

uwsgi_params 定义了传递到 uWSGI 服务器的参数，示例：

```
uwsgi_param  QUERY_STRING       $query_string;
uwsgi_param  REQUEST_METHOD     $request_method;
uwsgi_param  CONTENT_TYPE       $content_type;
uwsgi_param  CONTENT_LENGTH     $content_length;
uwsgi_param  REQUEST_URI        $request_uri;
uwsgi_param  PATH_INFO          $document_uri;
uwsgi_param  DOCUMENT_ROOT      $document_root;
uwsgi_param  SERVER_PROTOCOL    $server_protocol;
uwsgi_param  REMOTE_ADDR        $remote_addr;
uwsgi_param  REMOTE_PORT        $remote_port;
uwsgi_param  SERVER_PORT        $server_port;
uwsgi_param  SERVER_NAME        $server_name;
```

- uWSGI 常用配置项

  socket or uwsgi-socket

指定 uwsgi 的客户端将要连接的 socket 的路径（使用 UNIX socket 的情况）或者地址（使用网络地址的情况）。最多可以同时指定 8 个 socket 选项。当使用命令行变量时，可以使用“-s”这个缩写。

```
--socket /tmp/uwsgi.sock
```

以上配置将会绑定到 /tmp/uwsgi.sock 指定的 UNIX socket

protocol

设置默认的通信协议（uwsgi，http，fastcgi）

```
--protocol <protocol>
```

processes or workers

为预先派生模式设置工作进程的数量。这个设置是 app 能实现简单并且安全的并发能力的基础。设置的工作进程越多，就能越快的处理请求。每一个工作进程都等同于一个系统进程，它消耗内存，所以需要小心设置工作进程的数量。如果设置的数量太多，就有可能是系统崩溃。

```
--processes 8
```

以上配置会产生 8 个工作进程

harakiri

这个选项会设置 harakiri 超时时间。如果一个请求花费的时间超过了这个 harakiri 超时时间，那么这个请求都会被丢弃，并且当前处理这个请求的工作进程会被回收再利用（即重启）。

```
--harakiri 60
```

这个设置会使 uwsgi 丢弃所有需要 60 秒才能处理完成的请求。

harakiri-verbose

当一个请求被 harakiri 杀掉以后，你将在 uWSGI 日志中得到一条消息。激活这个选项会打印出额外的信息（例如，在 linux 中会打印出当前的 syscall）。

```
--harakiri-verbose
```

以上配置会开启 harakiri 的额外信息。

daemonize

使进程在后台运行，并将日志打到指定的日志文件或者 udp 服务器

```
--daemonize /var/log/uwsgi.log
```

这个指令会让 uWSGI 在后台运行并将日志打到 /var/log/uwsgi.log 文件中。

buffer-size

设置用于 uwsgi 包解析的内部缓存区大小。默认是 4k。

如果接受一个拥有很多请求头的大请求，可以增加这个值到 64k。

```
--buffer-size 32768
```

这个命令会允许 uWSGI 服务器接收最大为 32k 的 uwsgi 包，再大的包就会被拒绝。

auto-procname

这个选项将自动给 uWSGI 的进程设置一些有意义的名字，例如“uWSGI master”，“uWSGI worker 1”，“uWSGI worker 2”。

procname-prefix

这个选项为进程名指定前缀。

```
--procname-prefix <value>
```

procname-prefix-spaced

用这个选项给进程名指定前缀时，前缀和进程名之间有空格分隔。

```
--procname-prefix-spaced <value>
```

procname-append

这个选项为进程名增加指定的后缀。

```
--procname-append <value>
```

master

启动主进程。

max-requests

为每个工作进程设置请求数的上限。当一个工作进程处理的请求数达到这个值，那么该工作进程就会被回收重用（重启）。可以使用这个选项来默默地对抗内存泄漏（尽管这类情况使用 reload-on-as 和 reload-on-rss 选项更有用）。

```
[uwsgi]
max-requests = 1000
```

上述配置设置工作进程没处理 1000 个请求就会被回收重用。

socket-timeout

为所有的 socket 操作设置内部超时时间（默认 4 秒）。

```
--socket-timeout 10
```

这个配置会结束那些处于不活动状态超过 10 秒的连接。

module [python plugin required]

加载指定的 python WSGI 模块（模块路径必须在 PYTHONPATH 里）

最后我用下面这张图展示了我们的基础服务的基本架构，图中也明确显示了 nginx 和 uWSGI 的关系位置。