美妙的多进程管理

造一个类gunicorn的轮子

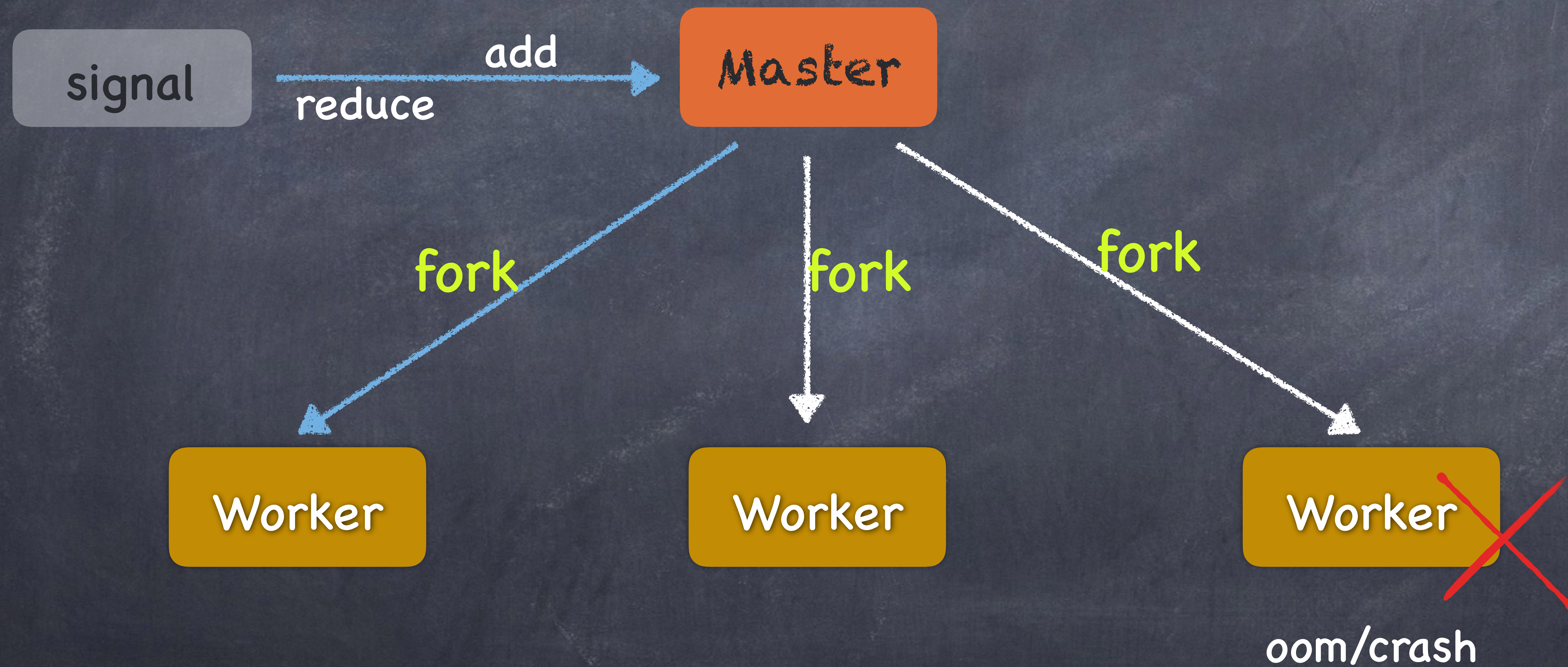
blog: xiaorui.cc

github: github.com/rfyiamcool

内容

- supervisor vs gunicorn vs uwsgi
- linux异步信号
- 孤儿进程 vs 僵尸进程
- daemon的实现
- prefork是怎么一回事
- 打造一个较完善的多进程管理轮子
- 怎么写代码

Master Worker



- elegance
  - ctrl c
  - pname
  - zombie/child
  - dup
  - ...
- daemon
- monitor
- signal
- reload
- dynamic
- ...

supervisor vs gunicorn vs uwsgi

- supervisor 是基于执行文件的,可以控制多个应用进程. 但单纯的多进程管理.
- gunicorn 、uwsgi是基于application的，而且可以实现wsgi网关接口

supervisor实现

file: process.py

```
class Subprocess():
    def spawn():
        filename, argv = self.get_execv_args()
        pid = options.fork()
        if pid != 0:
            return self._spawn_as_parent(pid)
        else:
            return self._spawn_as_child(filename, argv)

    def _spawn_as_child():
        options.dup2(self.pipes['child_stdin'], 0)
        options.dup2(self.pipes['child_stdout'], 1)
        options.dup2(self.pipes['child_stdout'], 2)
        options.execve(filename, argv, env)
```

file: options.py

```
def execve(self, filename, argv, env):
    return os.execve(filename, argv, env)
#execve把新进程替换老进程,继承
```

```
[program:web]
command=python /var/www/service.py 80%(process_num)02d
process_name=%(program_name)s_%(process_num)02d
autostart=true
autorestart=true
umask=022
startsecs=0
stopwaitsecs=0
redirect_stderr=true
stdout_logfile=/tmp/codoon.log
numprocs=4
```

why repeat ?

- 基于function更加细腻
  - dynamic add/reduce
  - log reload / module reload / config reload
  - kill friendly
  - timeout ? force to kill
  - queue, 共享变量, 锁… (封装好)
  - more …

Linux Signal for diy

信号	数字	描述
sigint	2	键盘ctrl c
sigquit	3	键盘ctrl d or \
sigkill	9	暴力终止进程
sigalrm	14	定时器超时
sigterm	15	默认的kill信号
sigchld	17	子进程退出发出的信号
sigttin	21	增加一个进程
sigttou	22	减少一个进程
sigwinch	34	清理worker进程

signal

- 不可靠信号： 也称为非实时信号，不支持排队，信号可能会丢失, 比如发送多次相同的信号, 进程只能收到一次. 信号值取值区间为1~31；
- 可靠信号： 也称为实时信号，支持排队, 信号不会丢失, 发多少次, 就可以收到多少次. 信号值取值区间为32~64
- 信号不排队

signal

- sigkill无法捕获

什么时候会被sigkill

- 自己手贱，kill -9 pid
- 内存占用厉害，被oom了
- ulimit设置了cpu timeout

在python下注册sigkill error

```
In [3]: def trycache(*args):
   ...:     print args
   ...:

In [4]: signal.signal(signal.SIGKILL,trycache)
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-4-7f0697d1acc4> in <module>()
----> 1 signal.signal(signal.SIGKILL,trycache)

RuntimeError: (22, 'Invalid argument')
```

孤儿进程 vs 僵尸进程

- 孤儿进程：一个父进程退出，而它的子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。
- 僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。

怎么解决僵尸问题

- 通过signal(SIGCHLD, SIG_IGN)通知内核对子进程的结束不关心，由内核回收。
- 父进程调用wait/waitpid函数进行收尸，如果尚无子进程退出wait会导致父进程阻塞。waitpid可以通过传递WNOHANG使父进程不阻塞立即返回。
  - 通过SIGCHLD的注册函数来处理信号，如一下子很多信号发出，那么会有丢失信号的问题，因为内核发信号不排队...
- 孤儿进程的方式，通过fork setsid实现

```python
import multiprocessing
import time

def daemon():
    while 1:
        time.sleep(10)

def non_daemon():
    while 1:
        time.sleep(100)

def non_daemon_break():
    time.sleep(3)   # 就是让他主动退出

if __name__ == '__main__':
    t = []
    t.append( multiprocessing.Process(name=`daemon', target=daemon))
    t.append( multiprocessing.Process(name=`non-daemon', target=non_daemon))
    t.append( multiprocessing.Process(name=`non-daemon', target=non_daemon_break))
    for i in t:
        i.daemon = True
        i.start()
    while 1:
        time.sleep(1)
```

```
[root@iZ25wvd7l9xZ ~]#
[root@iZ25wvd7l9xZ ~]# ps aux f|grep python |grep -v grep
root     26485  0.0  0.4 138920  4820 pts/0    S+   18:48   0:00          \_ python i.py
root     26486  0.0  0.3 138920  3512 pts/0    S+   18:48   0:00              \_ python i.py
root     26487  0.0  0.3 138920  3512 pts/0    S+   18:48   0:00              \_ python i.py
root     26488  0.0  0.0      0     0 pts/0    Z+   18:48   0:00              \_ [python] <defunct>
[root@iZ25wvd7l9xZ ~]# kill 26487
[root@iZ25wvd7l9xZ ~]# ps aux f|grep python |grep -v grep
root     26485  0.0  0.4 138920  4820 pts/0    S+   18:48   0:00          \_ python i.py
root     26486  0.0  0.3 138920  3512 pts/0    S+   18:48   0:00              \_ python i.py
root     26487  0.0  0.0      0     0 pts/0    Z+   18:48   0:00              \_ [python] <defunct>
root     26488  0.0  0.0      0     0 pts/0    Z+   18:48   0:00              \_ [python] <defunct>
[root@iZ25wvd7l9xZ ~]#
```

```
import time
import os
import multiprocessing

def daemon():
    while 1:
        time.sleep(10)

def non_daemon():
    while 1:
        time.sleep(100)

def non_daemon_break():
    time.sleep(3)          # 因为是孤儿进程，子进程由init收尸了.

if __name__ == '__main__':
    t = []
    t.append(multiprocessing.Process(name='daemon', target=daemon))
    t.append(multiprocessing.Process(name='non-daemon', target=non_daemon))
    t.append(multiprocessing.Process(name='non-daemon', target=non_daemon_break))
    for i in t:
        i.daemon = True
        i.start()
    os.kill(os.getpid(), 15)
```

```
[root@iZ25wvd7l9xZ ~]# ps -ef|grep i.py|grep -v grep
root      26806     1  0 18:59 pts/2    00:00:00 python i.py
root      26807     1  0 18:59 pts/2    00:00:00 python i.py
```

Daemon 守候进程

- fork子进程，然后父进程退出，这已经构成基本的daemon！但子进程还在父进程的会话里面.
- 子进程调用setsid，使子进程成为新的会话组长. 但新的会话组长可申请控制终端.
- 再次fork一个子孙进程，干掉子进程，保留孙子进程.
- 切换工作目录，关闭stdin\stdout\stderr的句柄，umask

如何摆脱终端?

- 进程 —> 进程组 —> 会话 = 登录终端
  - 摆脱当前终端, new session
- 关闭终端会触发SIGHUP
  - 屏蔽SIGHUP
- NOHUP = signal(SIGHUP, SIG_IGN)
- NOHUP sleep 100 > dehub.log 2 > &1 == ignore sighup + os.dup(1,2)

daemon代码示例

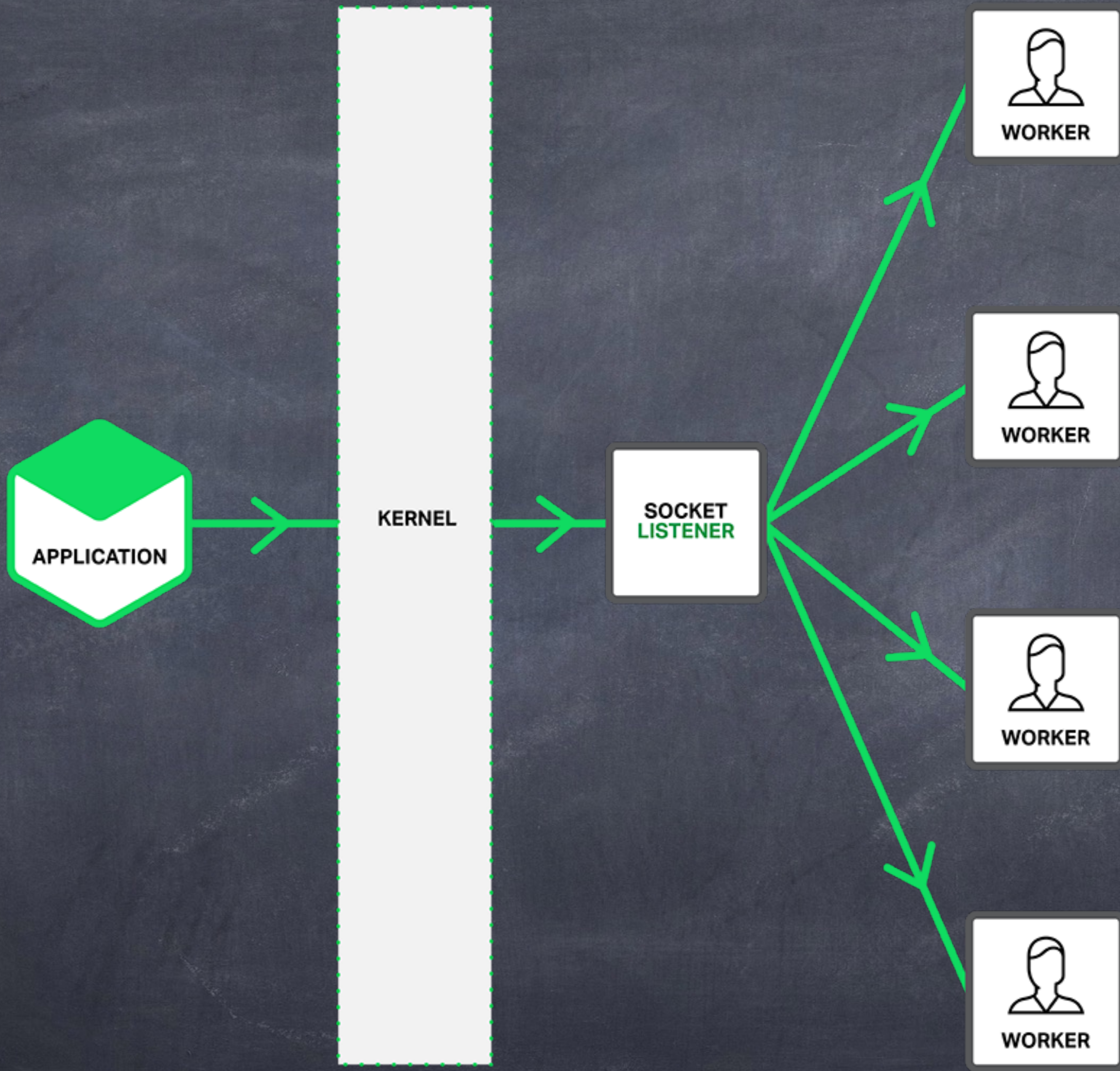
```
import os
import time

def daemonize():
    pid=os.fork() # fork1
    if pid<0: # error
        print "fork1 error"
        return -1
    elif pid>0: # parent.
        exit(0)
    os.chdir("/")
    os.setsid()
    pid=os.fork() # fork 2
    if pid<0:
        print "fork2 error"
        return -1
    elif pid>0:
        exit(0)
    os.umask(022)
    os.close(0)
    os.close(1)
    os.close(2)
    fd=os.open('/dev/null', 2)
    os.dup(fd)
    os.dup(fd)

if __name__ == "__main__":
    daemonize()
    time.sleep(30)
```

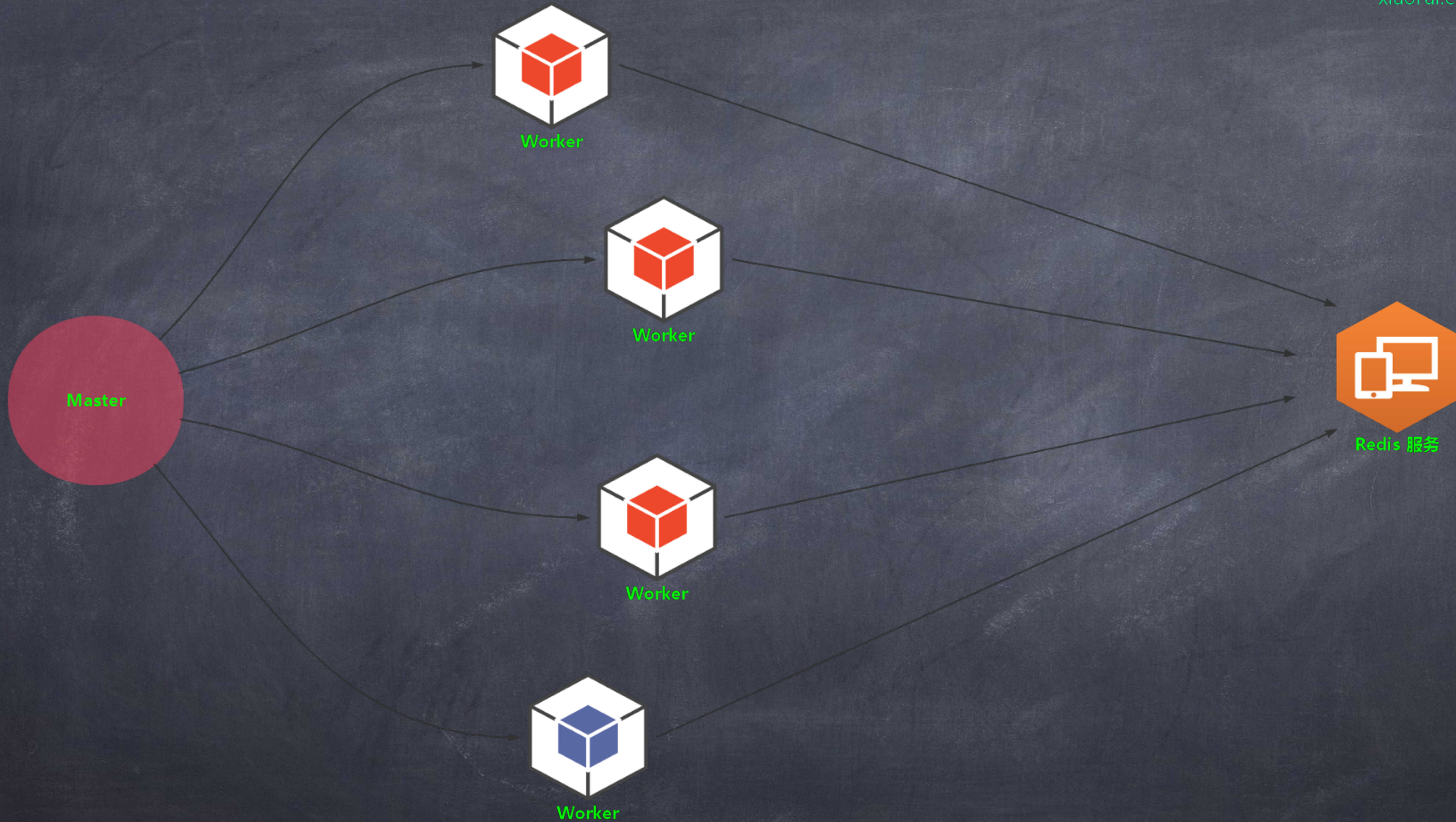
```
[root@iZ25wvd7l9xZ ~]# ps aux|grep a.py|grep -v grep
root      2883  0.0  0.2 117176  2340 ?        S    00:07   0:00 python a.py
[root@iZ25wvd7l9xZ ~]# lsof -p 2883
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
python  2883 root  cwd    DIR  202,1     4096       2 /
python  2883 root  rtd    DIR  202,1     4096       2 /
python  2883 root  txt    REG  202,1     9032 1055201 /usr/bin/python
python  2883 root  mem    REG  202,1  1672544 1053226 /usr/lib64/libpython2.6.so.1.0
python  2883 root  mem    REG  202,1   157032  262158 /lib64/ld-2.12.so
python  2883 root  mem    REG  202,1  1926760  262162 /lib64/libc-2.12.so
python  2883 root  mem    REG  202,1    22536  262166 /lib64/libdl-2.12.so
python  2883 root  mem    REG  202,1   599392  262273 /lib64/libm-2.12.so
python  2883 root  mem    REG  202,1    17520  262197 /lib64/libutil-2.12.so
python  2883 root  mem    REG  202,1   145896  271526 /lib64/libpthread-2.12.so
python  2883 root  mem    REG  202,1    20328 1068675 /usr/lib64/python2.6/lib-dynload/timemodule.so
python  2883 root  mem    REG  202,1 99158576 1052128 /usr/lib/locale/locale-archive
python  2883 root   0u    CHR    1,3      0t0    3866 /dev/null
python  2883 root   1u    CHR    1,3      0t0    3866 /dev/null
python  2883 root   2u    CHR    1,3      0t0    3866 /dev/null
[root@iZ25wvd7l9xZ ~]# ps -ef |grep 2883 |grep -v grep
root      2883     1  0 00:07 ?        00:00:00 python a.py
[root@iZ25wvd7l9xZ ~]#
```

那么问题来了，如何造一个健全的后端服务！

- 用配置文件控制
- uid权限
- 规范进程名
- daemon守候进程
- 调整进程，Add、Reduce
- 调优配置：最大处理任务,是否线程、协程
- 捕获各种信号，解决僵尸进程
- fcntl lock/check sock.pid
- 传参获取服务状态，重启服务，开启，停止

a program


APPLICATION
KERNEL
SOCKET
LISTENER
WORKER
WORKER
WORKER
WORKER


xiaorui.cc
Worker
Worker
Master
Worker
Worker
Redis 服务

"end..."

– xiaorui.cc