Django long running asynchronous tasks with threads/processing

Asked 7 years, 8 months ago Active 4 years, 4 months ago Viewed 17k times

29

9

Disclaimer: I do know that there are several similar questions on SO. I think I've read most if not all of them, but did not find an answer to my real question (see later). I also do know that using celery or other asynchronous queue systems is the best way to achieve long running tasks - or at least use a cron-managed script. There's also mod_wsgi doc about processes and threads but I'm not sure I got it all correct.

The question is:

what are the exact risks/issues involved with using the solutions listed down there? Is any of them viable for long running tasks (ok, even though celery is better suited)? My question is really more about understanding the internals of wsgi and python/django than finding the best overall solution. Issues with blocking threads, unsafe access to variables, zombie processing, etc.

Let's say:

1. my "long_process" is doing something really safe. even if it fails i don't care.
2. python >= 2.6
3. I'm using mod_wsgi with apache (will anything change with uwsgi or gunicorn?) in daemon mode

mod_wsgi conf:

```
WSGIDaemonProcess NAME user=www-data group=www-data threads=25
WSGIScriptAlias / /path/to/wsgi.py
WSGIProcessGroup %{ENV:VHOST}
```

I figured that these are the options available to launch separate *processes* (meant in a broad sense) to carry on a long running task while returning quickly a response to the user:

os.fork

```
import os

if os.fork()==0:
    long_process()
else:
    return HttpResponse()
```

subprocess

```
import subprocess

p = subprocess.Popen([sys.executable, '/path/to/script.py'],
                                      stdout=subprocess.PIPE,
                                      stderr=subprocess.STDOUT)
```

```
import threading

t = threading.Thread(target=long_process,
                             args=args,
                             kwargs=kwargs)
t.setDaemon(True)
t.start()
return HttpResponse()
```

NB.

> Due to the Global Interpreter Lock, in CPython only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better of use of the computational resources of multi-core machines, you are advised to use multiprocessing. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

The main thread will quickly return (the httpresponse). Will the spawned long thread block wsgi from doing something else for another request?!

multiprocessing

```
from multiprocessing import Process

p = Process(target=_bulk_action,args=(action,objs))
p.start()
return HttpResponse()
```

This should solve the thread concurrency issue, shouldn't it?

So those are the options I could think of. What would work and what not, and why?

django asynchronous mod-wsgi

edited Nov 9 '11 at 17:55

asked Nov 9 '11 at 17:23

Stefano
11.7k 11 54 76

This discussion is quite interesting to complement some of the information here specifically for django: groups.google.com/group/django-developers/browse_thread/thread/… – Stefano Nov 15 '11 at 18:43

3 Answers

os.fork

subprocess

Using `subprocess` is expected to be interactive. In other words, while you can use this to effectively spawn off a process, it's expected that at some point you'll terminate it when finished. It's possible Python might clean up for you if you leave one running, but my guess would be that this will actually result in a memory leak.

threads

Threads are defined units of logic. They start when their `run()` method is called, and terminate when the `run()` method's execution ends. This makes them well suited to creating a branch of logic that will run outside the current scope. However, as you mentioned, they are subject to the Global Interpreter Lock.

multiprocessing

This is basically threads on steroids. It has the benefits of a thread, but is not subject to the Global Interpreter Lock, and can take advantage of multi-core architectures. However, they are more complicated to work with as a result.

So, your choices really come down to threads or multiprocessing. If you can get by with a thread and it makes sense for your application, go with a thread. Otherwise, use multiprocessing.

answered Nov 9 '11 at 20:06

Chris Pratt
169k 23 256 324

1 Thanks Chris. Could you elaborate a little bit more on **Threads** -> 'subject to GDI' and **Processes** -> 'more complicated to work with as a result'. *THREADS*: If, say, 2 different web requests spawn 2 different threads that work on different resources, will they be able to run concurrently? My guess is.. not (especially if coming from a single-process mod_wsgi). But it sounds like, according to Graham, they will not block serving other requests, which is still the most important thing. *PROCESSES*: is the difficulty related to making them communicate, or is there something else? Thanks! – Stefano Nov 10 '11 at 9:39

1 The Global Interpreter Lock is only an issue with CPython implementations. That particular version of the Python interpreter enforces a hard limit of one thread at a time being able to work with bytecode. It's probably nothing you'll need to worry about, but in terms of interoperability, it's something to be aware of. – Chris Pratt Nov 10 '11 at 15:08

1 Threads are almost stupidly simple to work with. You subclass `Thread` define `__init__` and `run` and you're off to the races. Multiprocessing deals with the concept of pools of workers. There's a lot more intialization and you have to carefully manage what's going on. It's the same reason that many modern-day applications *still* don't actually utilize multiple cores. You can do amazing stuff with all that processing power, but it's like orchestrating a symphony -- you'll have to put in some work to get it just right. – Chris Pratt Nov 10 '11 at 15:12

11

I have found that using uWSGI Decorators is **quite** simpler than using Celery if you need just run some long task in background. Think Celery is best solution for serious heavy project, and it's overhead for doing something simple.

For start using uWSGI Decorators you just need to update your uWSGI config with

write code like:

```
@spoolraw
def long_task(arguments):
    try:
        doing something with arguments['myarg'])
    except Exception as e:
        ...something...
    return uwsgi.SPOOL_OK

def myView(request)
    long_task.spool({'myarg': str(someVar)})
    return render_to_response('done.html')
```

Than when you start view in uWSGI log appears:

```
[spooler] written 208 bytes to file
/here/the/path/to/dir/uwsgi_spoolfile_on_hostname_31139_2_0_1359694428_441414
```

and when task finished:

```
[spooler /here/the/path/to/dir pid: 31138] done with task
uwsgi_spoolfile_on_hostname_31139_2_0_1359694428_441414 after 78 seconds
```

There is strange(for me) restrictions:

```
    - spool can receive as argument only dictionary of strings, look like because
it's serialize in file as strings.
    - spool should be created on start up so "spooled" code it should be
contained in separate file which should be defined in uWSGI config as
<import>pyFileWithSpooledCode</import>
```

edited Mar 20 '15 at 12:09

answered Feb 1 '13 at 5:34

Oleg Neumyvakin

6,218 2 34 39

didn't know about uWSGI decorators! – Stefano Feb 1 '13 at 8:51

After running `pip install uwsgi` , I still can not `import uwsgi` in python shell. Any suggestion? @Oleg Neumyvakin – YeRuizhi Nov 8 '14 at 11:19

3

For the question:

> Will the spawned long thread block wsgi from doing something else for another request?!

the answer is no.

You still have to be careful creating background threads from a request though in case you simply create huge numbers of them and clog up the whole process. You really need a task queueing system even if you are doing stuff in process.

Using a system like Celery is still probably the best solution.

answered Nov 9 '11 at 22:14

Graham Dumpleton
49.9k 6 88 111

thanks Graham. Was hoping to get mod_wsgi creator feedback on this subject. Just to be sure: does it make any difference how I configure `WSGIDaemonProcess (processes=P) threads=T` wrt to the GDI/blocking threads? – Stefano Nov 10 '11 at 9:32