淺談 Gunicorn 各個 worker type 適合的情境

Genchi Lu ·

10 min read · Jan 30, 2018

Gunicorn 在 Python 2.7 有幾種 worker type，分別是 sync、gthread、eventlet、gevent 和 tornado。

根據底層運作的原理可以將 worker 分成三種類型：

1. sync 底層實作是每個請求都由一個 process 處理。
2. gthread 則是每個請求都由一個 thread 處理。
3. eventlet、gevent、tarnado 底層則是利用非同步 IO 讓一個 process 在等待 IO 回應時繼續處理下個請求。

在接下來的文章我會以下面的 sample code 做範例，簡單描述各種類型的 worker 在 CPU bound 和 IO bound 的程式在效能上的表現。

```
from django.http import HttpResponse
import time

def ioTask(request):
    time.sleep(2)
    return HttpResponse("IO bound task finish!\n")

def cpuTask(request):
    for i in range(10000000):
        n = i*i*i
    return HttpResponse("CPU bound task finish!\n")
```

gistfile1.txt hosted with ❤ by GitHub view raw

用 process 處理請求

當 gunicorn worker type 使用 sync 時，web 啟動時會預先開好對應數量的 process 處理請求，理論上 concurrency 的上限等同於 worker 數量。如下 gunicorn 啟動時開了一個 pid 為 569 的 process 來處理請求，理論上每次只能處理一個請求：

```
$> gunicorn -w 1 -k sync HelloWorld.wsgi:application -b
192.168.55.100:80

[2018-01-29 16:35:05 +0000] [564] [INFO] Starting gunicorn 19.7.1

[2018-01-29 16:35:05 +0000] [564] [INFO] Listening at:
http://192.168.55.100:80 (564)

[2018-01-29 16:35:05 +0000] [564] [INFO] Using worker: sync
```

```
[2018-01-29 16:35:05 +0000] [569] [INFO] Booting worker with pid: 569
```

用 siege 分別對 IO bound task 和 CPU bound task 發出 2 個請求可以明顯看到第二個 request 被第一個 request 阻塞，如下：

```
$> siege -c 2 -r 1 http://192.168.55.100/ioTask -v
** SIEGE 3.0.5
** Preparing 2 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   3.01 secs:      22 bytes ==> GET  /ioTask
// 下面請求開始被阻塞
HTTP/1.1 200   5.02 secs:      22 bytes ==> GET  /ioTask
......
......
$> siege -c 2 -r 1 http://192.168.55.100/cpuTask -v
** SIEGE 3.0.5
** Preparing 2 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   1.47 secs:      23 bytes ==> GET  /cpuTask
// 下面請求開始被阻塞
HTTP/1.1 200   3.09 secs:      23 bytes ==> GET  /cpuTask
```

這種類型的好處是錯誤隔離高，一個 process 掛掉只會影響該 process 當下服務的請求，而不會影響其他請求。

壞處則為 process 資源開銷較大，開太多 worker 時對記憶體或 CPU 的影響很大，因此 concurrency 理論上限極低。

用 thread 處理請求

當 gunicorn worker type 用 gthread 時，可額外加參數 --thread 指定每個 process 能開的 thread 數量，此時 concurrency 的上限為 worker 數量乘以給個 worker 能開的 thread 數量。如下 gunicorn 啟動時開了一個 pid 為 595 的 process 來處理請求， thread 數量為 2，理論上每次只能處理二個請求：

```
$> gunicorn -w 1 -k gthread --thread=2 HelloWorld.wsgi:application -b
192.168.55.100:80

[2018-01-29 16:50:21 +0000] [590] [INFO] Starting gunicorn 19.7.1

[2018-01-29 16:50:21 +0000] [590] [INFO] Listening at:
http://192.168.55.100:80 (590)

[2018-01-29 16:50:21 +0000] [590] [INFO] Using worker: gthread

[2018-01-29 16:50:21 +0000] [595] [INFO] Booting worker with pid: 595
```

用 siege 分別對 IO bound task 和 CPU bound task 發出 4 個請求可以明顯看到第三個請求以後才會被阻塞：

```
$> siege -c 4 -r 1 http://192.168.55.100/ioTask -v
** SIEGE 3.0.5
** Preparing 4 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   2.00 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.00 secs:      22 bytes ==> GET  /ioTask
// 下面請求開始被阻塞
HTTP/1.1 200   4.01 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   4.01 secs:      22 bytes ==> GET  /ioTask
......
......
$> siege -c 4 -r 1 http://192.168.55.100/cpuTask -v
** SIEGE 3.0.5
** Preparing 4 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   3.00 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200   3.32 secs:      23 bytes ==> GET  /cpuTask
// 下面請求開始被阻塞
HTTP/1.1 200   5.20 secs:      23 bytes ==> GET  /cpuTask
```

```
HTTP/1.1 200   5.44 secs:      23 bytes ==> GET  /cpuTask
```

這種類型的 worker 好處是 concurrency 理論上限會比 process 高，壞處依然是 thread 數量，OS 中 thread 數量是有限的，過多的 thread 依然會造成系統負擔。

用非同步 IO 處理每個請求

當 gunicorn worker type 用 eventlet、gevent、tarnado 等類型時，每個請求都由同一個 process 處理，而當遇到 IO 時該 process 不會等 IO 回應，會繼續處理下個請求直到該 IO 完成，理論上 concurrency 無上限。以 gevent 為例，gunicorn 啟動時開了一個 pid 為 733 的 process 來處理請求：

```
$> gunicorn -w 1 -k gevent HelloWorld.wsgi:application -b
192.168.55.100:80

[2018-01-29 17:11:03 +0000] [728] [INFO] Starting gunicorn 19.7.1

[2018-01-29 17:11:03 +0000] [728] [INFO] Listening at:
http://192.168.55.100:80 (728)

[2018-01-29 17:11:03 +0000] [728] [INFO] Using worker: gevent

[2018-01-29 17:11:03 +0000] [733] [INFO] Booting worker with pid: 733
```

用 siege 對 IO bound task 發出 10 個請求可以明顯看到沒有任何請求被阻塞：

```
$> siege -c 10 -r 1 http://192.168.55.100/ioTask -v
** SIEGE 3.0.5
** Preparing 10 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   2.01 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.02 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.02 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.02 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.01 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.01 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.01 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.02 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.02 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.02 secs:      22 bytes ==> GET  /ioTask
```

但當面臨 CPU bound 請求時，則會退化成用 process 處理請求一樣，concurrency 上限為 worker 數量。如下用 siege 對 CPU bound task 發出 10 個請求，可以看到第二個請求以後就被阻塞：

```
$> siege -c 10 -r 1 http://192.168.55.100/cpuTask -v
```

```
** SIEGE 3.0.5
** Preparing 10 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   1.61 secs:      23 bytes ==> GET  /cpuTask
// 下面請求開始被阻塞
HTTP/1.1 200   3.20 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200   4.88 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200   6.38 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200   6.97 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200   8.60 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200  10.12 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200  11.74 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200  13.25 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200  14.72 secs:      23 bytes ==> GET  /cpuTask
```

因此使用非同步類型的 worker 好處和壞處非常明顯，對 IO bound task 的高效能，但在 CPU bound task 會不如 thread。

結論

當談到效能時，必須考慮到使用情境。 gunicorn + 非同步 IO 效能就一定比較好的說法並不一定成立。

從上面的數據三種類型的 worker 都有其相對適合的場景：

> 當需要穩定的系統時， 用 *process* 處理請求可以保證一個請求的異常導致程式 *crash* 不會影響到其他請求。
>
> 當 *web* 服務內大部分都是 *cpu* 運算時，用 *thread* 可以提供不錯的效能。
>
> 當 *web* 服務內大部分都是 *io* 時，用非同步 *io* 可以達到極高的 *concurrency* 數量。

Python Gunicorn Worker Type

Brief introduction about the types of worker in gunicorn and respective suitable scenario

Genchi Lu ·
6 min read · Jan 31, 2018

In Python 2.7, Gunciorn provides serval types of worker: sync, gthread, eventlet, gevent and tornado.

I classify them into three categories according to how they were implemented to handle a request.

1. Per request a process: while the type of worker is set to sync, gunicorn would delegate one process for each request.
2. Per request a thread: while the type of worker is set to gthread, gunicorn would delegate one thread fork from a process for each request.

3. Async IO: while the type of worker is set to evenlet, gevent or tarnado, gunicorn would handle multiple requests at one process with async IO.

Below is a snip code with two simple tasks, one would sleep 2 sec to simulate an IO-bound task, the other would just calculate multiplication to simulate a CPU-bound task. In the following article, I would briefly explain each category of the type of worker's performance while running this two task.

```
1  from django.http import HttpResponse
2  import time
3
4  def ioTask(request):
5      time.sleep(2)
6      return HttpResponse("IO bound task finish!\n")
7
8  def cpuTask(request):
9      for i in range(10000000):
10         n = i*i*i
11     return HttpResponse("CPU bound task finish!\n")
```

gistfile1.txt hosted with ❤ by GitHub view raw

Per request a process

When the type of worker is set to sync, gunicorn prepares worker processes on startup. In theory, the maximum concurrency number at the same number of the worker process. Below, when gunicorn startup, it prepares one process with PID 569 to handle incoming requests when startup. It could only handle one request at a time:

```
$> gunicorn -w 1 -k sync HelloWorld.wsgi:application -b
192.168.55.100:80

[2018-01-29 16:35:05 +0000] [564] [INFO] Starting gunicorn 19.7.1

[2018-01-29 16:35:05 +0000] [564] [INFO] Listening at:
http://192.168.55.100:80 (564)

[2018-01-29 16:35:05 +0000] [564] [INFO] Using worker: sync

[2018-01-29 16:35:05 +0000] [569] [INFO] Booting worker with pid: 569
```

Then I use siege to test each task by sending two requests simultaneously. You can observe that the second request was blocked by the first request.

```
// Test IO-bound task

$> siege -c 2 -r 1 http://192.168.55.100/ioTask -v

** SIEGE 3.0.5

** Preparing 2 concurrent users for battle.
```

```
The server is now under siege...
HTTP/1.1 200   3.01 secs:      22 bytes ==> GET  /ioTask
// Next request was blocked
HTTP/1.1 200   5.02 secs:      22 bytes ==> GET  /ioTask
......
......
// Test CPU-bound task
$> siege -c 2 -r 1 http://192.168.55.100/cpuTask -v
** SIEGE 3.0.5
** Preparing 2 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   1.47 secs:      23 bytes ==> GET  /cpuTask
// Next request was blocked
HTTP/1.1 200   3.09 secs:      23 bytes ==> GET  /cpuTask
```

The advantage is high error separation. If one process crashed it would only affect the request that process was handling, and other requests would not be affected.

The disadvantage is that a process needs more resources in OS. As the number of processes grows up, it would consume too much CPU and

memory which is not necessary. Thus it would lower concurrency when using sync.

Per request a thread

When the type of worker is set to gthread, gunicorn prepares the worker process on startup. When a request is coming, one of these processes would fork a thread to handle that request. In theory, the maximum concurrency number equals to the number of the worker process times the number of threads. Below, when gunicorn startup, it prepares one process with PID 569 when startup and the number of threads are set to 2. Theoretically, it could handle 2 requests one time theoretically:

```
$> gunicorn -w 1 -k gthread --thread=2 HelloWorld.wsgi:application -b
192.168.55.100:80

[2018-01-29 16:50:21 +0000] [590] [INFO] Starting gunicorn 19.7.1

[2018-01-29 16:50:21 +0000] [590] [INFO] Listening at:
http://192.168.55.100:80 (590)

[2018-01-29 16:50:21 +0000] [590] [INFO] Using worker: gthread

[2018-01-29 16:50:21 +0000] [595] [INFO] Booting worker with pid: 595
```

Using siege to test each task by sending 4 requests simultaneously. You can observe that requests began to be blocked when the third request coming in.

```
// Test IO-bound task
$> siege -c 4 -r 1 http://192.168.55.100/ioTask -v
** SIEGE 3.0.5
** Preparing 4 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   2.00 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.00 secs:      22 bytes ==> GET  /ioTask
// Next request was blocked
HTTP/1.1 200   4.01 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   4.01 secs:      22 bytes ==> GET  /ioTask
......
......
// Test CPU-bound task
$> siege -c 4 -r 1 http://192.168.55.100/cpuTask -v
** SIEGE 3.0.5
** Preparing 4 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   3.00 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200   3.32 secs:      23 bytes ==> GET  /cpuTask
```

```
// Next request was blocked
HTTP/1.1 200   5.20 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200   5.44 secs:      23 bytes ==> GET  /cpuTask
```

The advantage is that it would have higher concurrency than request per process. But the number of thread is still limit in OS, it still would affect system if there were too many threads.

Using async IO to handle each request

When the type of worker is set to eventlet, gevent, or tarnado, multiple requests would be handled by one process which was using async IO. That process would not wait for IO and continue to handle other requests until that IO is completed. In theory, it has unlimited concurrency. I use gevent as an example: when gunicorn startup, it prepares one process with PID 733 to handle incoming requests.

```
$> gunicorn -w 1 -k gevent HelloWorld.wsgi:application -b
192.168.55.100:80
[2018-01-29 17:11:03 +0000] [728] [INFO] Starting gunicorn 19.7.1
[2018-01-29 17:11:03 +0000] [728] [INFO] Listening at:
http://192.168.55.100:80 (728)
[2018-01-29 17:11:03 +0000] [728] [INFO] Using worker: gevent
```

```
[2018-01-29 17:11:03 +0000] [733] [INFO] Booting worker with pid: 733
```

Using siege to send 10 concurrency request to IO-bound task simultaneously, there are no requests were blocked:

```
// Test IO-bound task
$> siege -c 10 -r 1 http://192.168.55.100/ioTask -v
** SIEGE 3.0.5
** Preparing 10 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   2.01 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.02 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.02 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.02 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.01 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.01 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.01 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.02 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.02 secs:      22 bytes ==> GET  /ioTask
HTTP/1.1 200   2.02 secs:      22 bytes ==> GET  /ioTask
```

But when sending 10 concurrency request to CPU-bound task simultaneously, it would behave just like sync. Except for the first request, other requests were blocked by the previous request.

```
// Test CPU-bound task
$> siege -c 10 -r 1 http://192.168.55.100/cpuTask -v
** SIEGE 3.0.5
** Preparing 10 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   1.61 secs:      23 bytes ==> GET  /cpuTask
// Next request was blocked
HTTP/1.1 200   3.20 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200   4.88 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200   6.38 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200   6.97 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200   8.60 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200  10.12 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200  11.74 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200  13.25 secs:      23 bytes ==> GET  /cpuTask
HTTP/1.1 200  14.72 secs:      23 bytes ==> GET  /cpuTask
```

It very clear that while using async IO to handle IO-bound requests is very efficient, but it would have lower concurrency when handling CPU-bound requests.

Conclusion

I think that we must define scenario first before choosing the type of worker:

> *If you want a stable system and hope that an exception in one request would not affect other requests, sync is what you need.*
>
> *If the tasks in your app are almost IO-bound, then async IO is good for you.*
>
> *If most of all tasks in your app are CPU-bound, then you should consider gthread first.*

Python Gunicorn Type Of Worker