Python3.4: AsyncIO

The next generation asynchronous I/O framework

Python3.4: AsyncIO

JuneHyeon Bae

PyCon Korea 2014

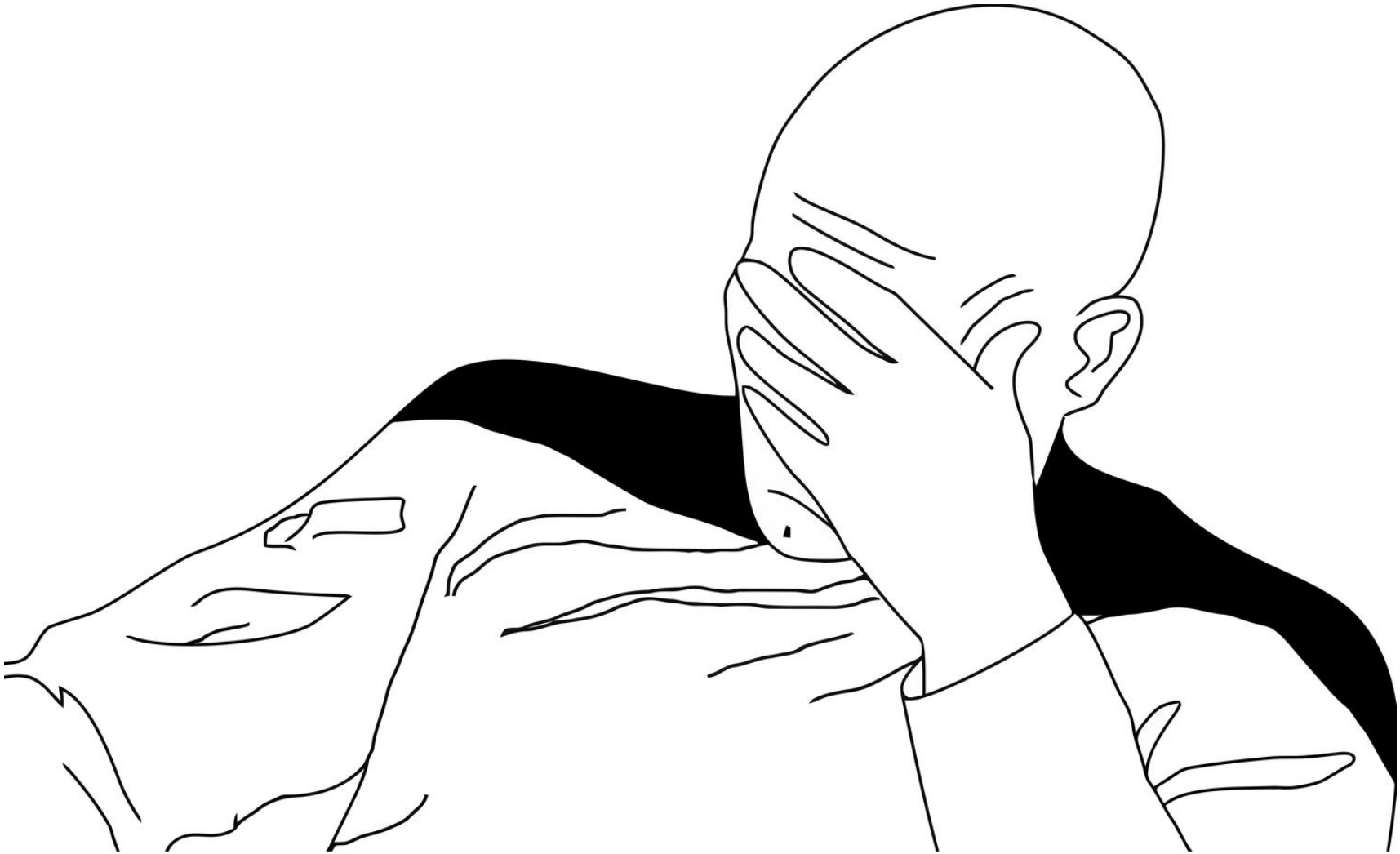
발표자 소개

- 닉네임: devunt
- 한국디지털미디어고등학교 HD 11기
- 위키미디어 재단 미디어위키 개발/번역팀
- PyPI warp-proxy 메인테이너

- https://img.tl/
- 파이썬 뉴비

D-75

2015학년도 대학수학능력시험



동시성을 구현하기

multithreading

여러 작업을 동시에 실행

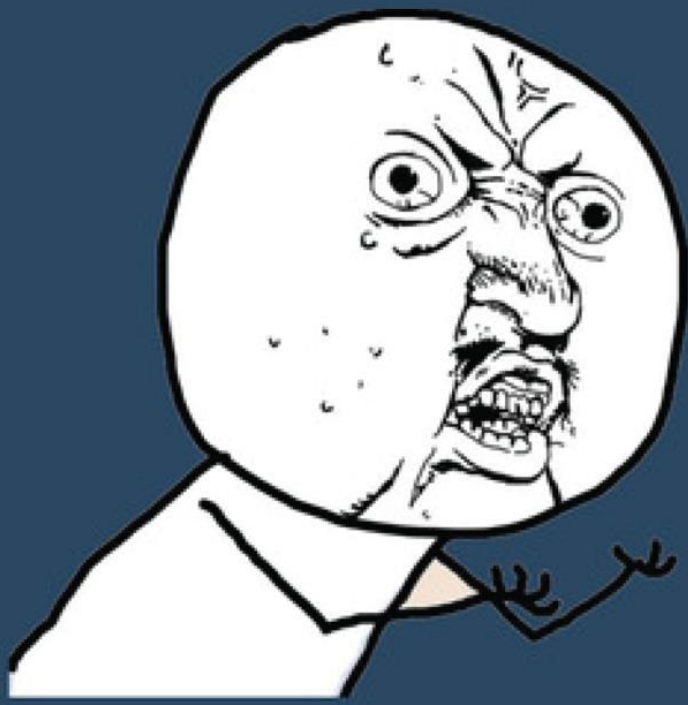
event-driven programming

콜백 함수를 깨워 실행

단일 스레드

```
1
2  var async = require("async");
3
4  User.find(userId, function(err, user){
5    if (err) return errorHandler(err);
6    User.all({where: {id: {$in: user.friends}}}, function(err, friends) {
7      if (err) return errorHandler(err);
8      async.each(friends, function(friend, done){
9        friend.posts = [];
10       Post.all({where: {userId: {$in: friend.id}}}, function(err, posts) {
11         if (err) return errorHandler(err);
12         async.each(posts, function(post, donePosts){
13           friend.push(post);
14           Comments.all({where: post.id}, function(err, comments) {
15             if (err) donePosts(err);
16             post.comments = comments;
17             donePosts();
18           });
19         }, function(err) {
20           if (err) return errorHandler(err);
21           done();
22         });
23       });
24     }, function(err) {
25       if (err) return errorHandler(err);
26       render(user, friends);
27     });
28   }
29 });
30
```

HEY LOOK!



CALLBACK EVERYWHERE!

coroutine

cooperative routines

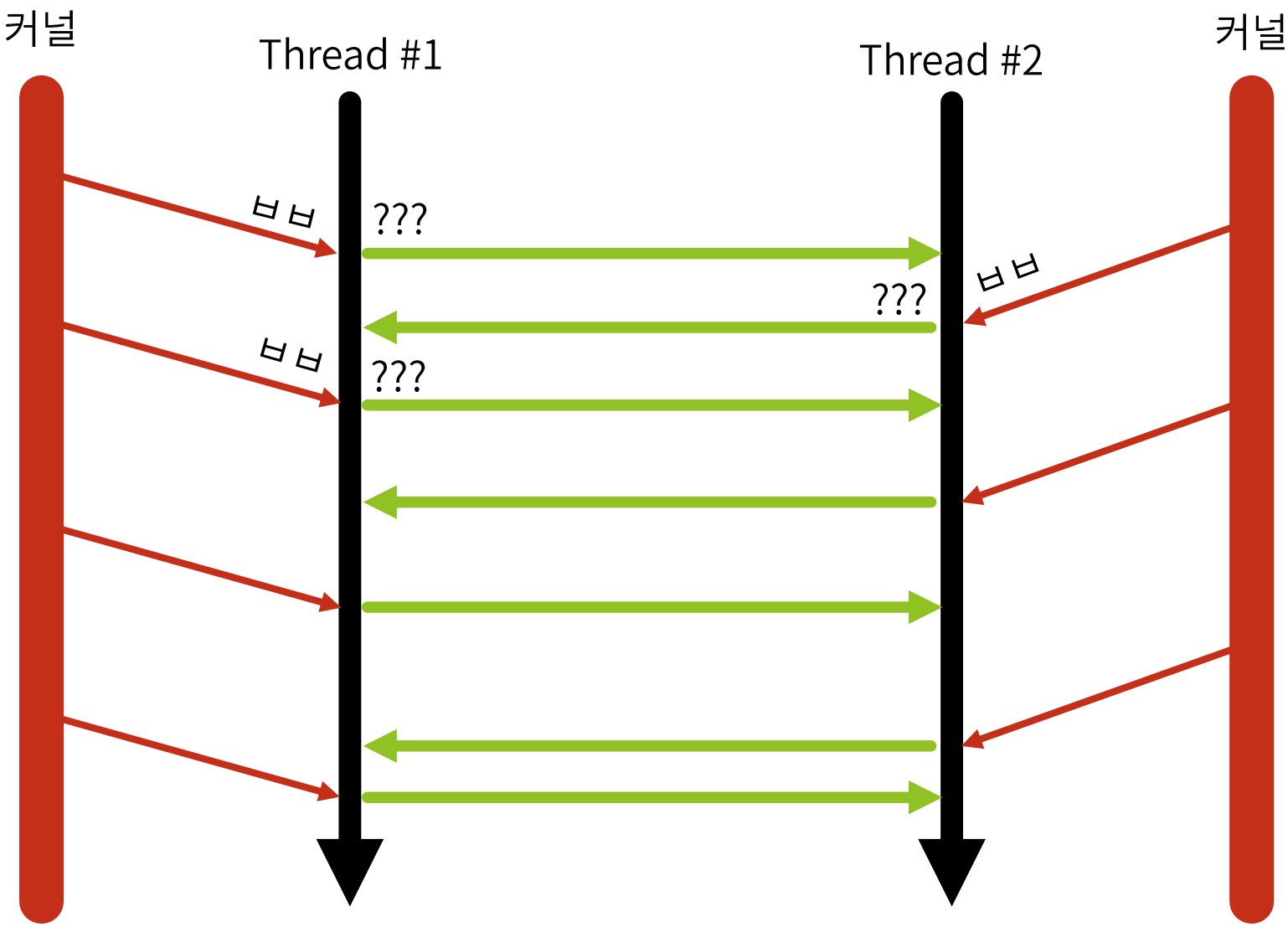
단일 스레드

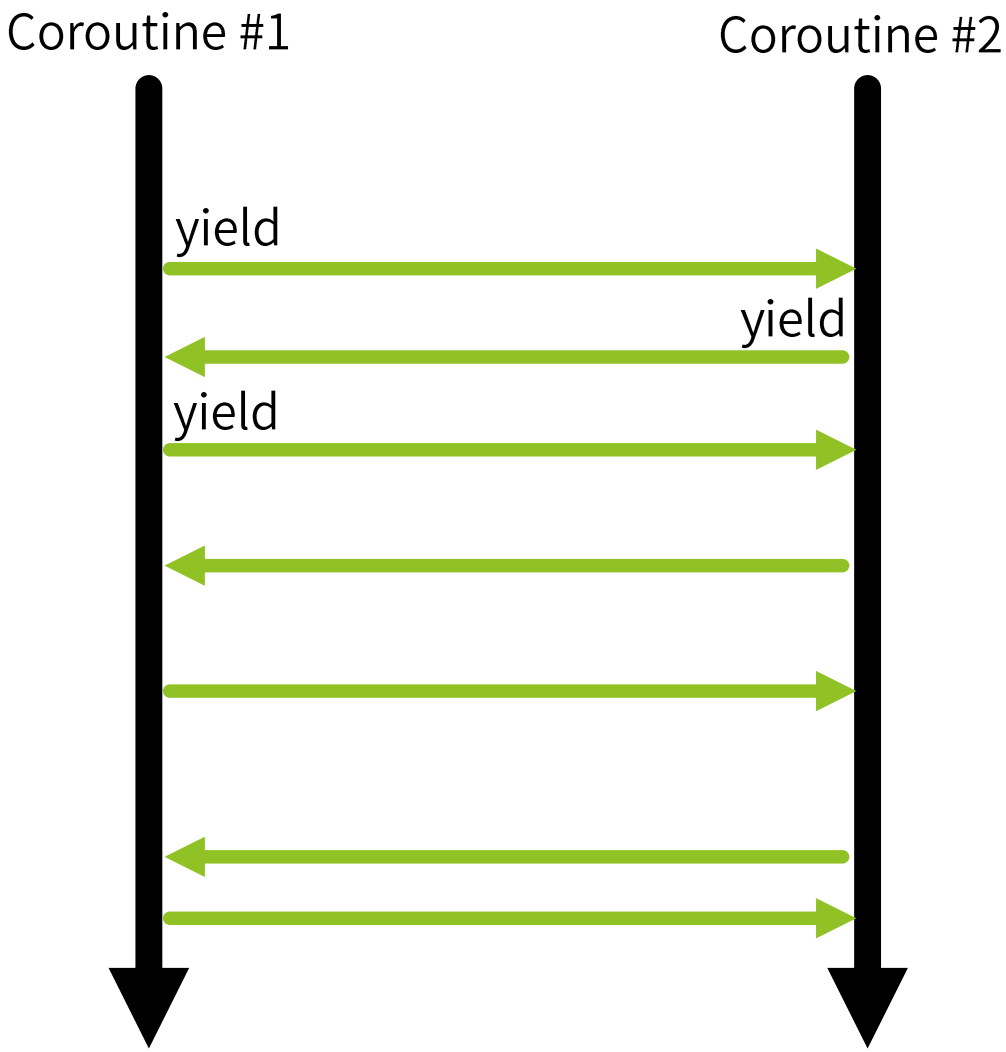
깨어나는 것이 아닌 잠드는 것

Event-driven programming과는 반대

사용자가 직접 스케줄링

스케줄링을 커널이 하는 것이 아닌 개발자가 직접 명시


커널
Thread #1
Thread #2
커널
ㅂㅂ
???
???
ㅂㅂ
ㅂㅂ
???


Coroutine #1
Coroutine #2
yield
yield
yield

코루틴에는 무엇이 필요할까?

coroutine scheduler

yield syntax

coroutine methods

이 모든 것을 한번에!

AsyncIO

2012년 12월

[PEP-3156]
Asynchronous IO Support Rebooted:
the “asyncio” Module

tulip project

reference implementation

2012.12 – 2013.10

1154개의 커밋

15명의 기여자

2년간의 개발 기간

2013년 10월

[Python Issue #19262]

Add asyncio (tulip, PEP 3156) to stdlib

New library modules:

- `asyncio`: *New provisional API for asynchronous IO* (**PEP 3156**).

Python 3.3 ≤

단, Python 2.x에서도 trollius를 이용해
일부 문법 변경을 거쳐 사용 가능

비동기 작업을 위한 표준 스택 제공

통일된 표준 비동기 I/O 스택을 언어 차원에서 기본 제공

여러 서드파티 라이브러리들이 이벤트 루프 공유 가능

AsyncIO versus Gevent

AsyncIO: 새로운 표준을 정의

Gevent: 기존의 표준에 gevent를 맞춤

AsyncIO: yield 시점이 명시적

Gevent: 명시적인 yield 없이 monkey-patch에 의해
암시적으로 yield

AsyncIO: 자체 스케줄러와 루프가 존재

Gevent: 스케줄러와 이벤트 루프로 greenlet과 libev를 사용

상황에 따라 적절히 사용

그럼 코드를 짜 보자!

Rewriting WARP

https://github.com/devunt/warp

Part 1: Imports

```
from gevent.monkey import patch_all; patch_all()
from threading import Thread
from Queue import Queue
from socket import (AF_INET, IPPROTO_TCP, SO_REUSEADDR,
                    SOCK_STREAM, SOL_SOCKET, TCP_NODELAY,
                    socket, error)
from time import sleep
```

```
from gevent.monkey import patch_all; patch_all()
from threading import Thread
from Queue import Queue
from socket import (AF_INET, IPPROTO_TCP, SO_REUSEADDR,
                    SOCK_STREAM, SOL_SOCKET, TCP_NODELAY,
                    socket, error)
from time import sleep
```

```
import asyncio
```

```
import asyncio
```

Part 2: Server Initializing

```
class Server(object):
    def __init__(self, hostname, port, count):
        self.hostname = hostname; self.port = port; self.count = count
        self.q = Queue()

    def start(self):
        for i in range(0, self.count):
            th = WorkerThread(self.q)
            th.daemon = True
            th.start()

        self.sc = socket(AF_INET, SOCK_STREAM)
        self.sc.bind((self.hostname, self.port))
        self.sc.listen(10)

        while True:
            self.q.put(self.sc.accept())
```

```
class Server(object):
    def __init__(self, hostname, port, count):
        self.hostname = hostname; self.port = port; self.count = count
        self.q = Queue()

    def start(self):
        for i in range(0, self.count):
            th = WorkerThread(self.q)
            th.daemon = True
            th.start()

        self.sc = socket(AF_INET, SOCK_STREAM)
        self.sc.bind((self.hostname, self.port))
        self.sc.listen(10)

        while True:
            self.q.put(self.sc.accept())
```

```
@asyncio.coroutine
def start_warp_server(host, port):
    yield from asyncio.start_server(
        accept_client, host=host, port=port)
```

```
@asyncio.coroutine
def start_warp_server(host, port):
    yield from asyncio.start_server(
        accept_client, host=host, port=port)
```

Part 3: Socket Accepting

```
class WorkerThread(Thread):
    def __init__(self, q):
        self.q = q
        Thread.__init__(self)

    def run(self):
        while True:
            conn, addr = self.q.get(block=True)
```

```
class WorkerThread(Thread):
    def __init__(self, q):
        self.q = q
        Thread.__init__(self)

    def run(self):
        while True:
            conn, addr = self.q.get(block=True)
```

```
def accept_client(client_reader, client_writer):
    task = asyncio.Task(
        process_warp(client_reader, client_writer))
    clients[task] = (client_reader, client_writer)

    def client_done(task):
        del clients[task]
        client_writer.close()

    task.add_done_callback(client_done)
```

```
def accept_client(client_reader, client_writer):
    task = asyncio.Task(
        process_warp(client_reader, client_writer))
    clients[task] = (client_reader, client_writer)

    def client_done(task):
        del clients[task]
        client_writer.close()

    task.add_done_callback(client_done)
```

Part 4: Socket Receiving

```
while True:
    data = conn.recv(1024)
    cont += data
    if data.find('\r\n\r\n') != -1:
        break
```

```
while True:
    data = conn.recv(1024)
    cont += data
    if data.find('\r\n\r\n') != -1:
        break
```

```
while True:
    line = yield from client_reader.readline()
    if line == b'\r\n':
        break
    header += line.decode('utf-8')
```

```
while True:
    line = yield from client_reader.readline()
    if line == b'\r\n':
        break
    header += line.decode('utf-8')
```

```
while (len(ct) < cl):
    ct += conn.recv(1024)
```

```
while (len(ct) < cl):
    ct += conn.recv(1024)
```

```
while (len(payload) < cl):
    payload += yield from client_reader.read(1024)
```

```
while (len(payload) < cl):
    payload += yield from client_reader.read(1024)
```

Part 5: Socket Sending

```
req_sc = socket(AF_INET, SOCK_STREAM)
req_sc.setsockopt(IPPROTO_TCP, TCP_NODELAY, 1)
req_sc.connect((host, port))
req_sc.send('%s\r\n' % new_head)
sleep(0.2)
```

```
req_sc = socket(AF_INET, SOCK_STREAM)
req_sc.setsockopt(IPPROTO_TCP, TCP_NODELAY, 1)
req_sc.connect((host, port))
req_sc.send('%s\r\n' % new_head)
sleep(0.2)
```

```
req_reader, req_writer =
    yield from asyncio.open_connection(
        host, port, flags=TCP_NODELAY)
req_writer.write(('%s\r\n' % new_head).encode('utf-8'))
yield from asyncio.sleep(0.2)
```

```
req_reader, req_writer =
    yield from asyncio.open_connection(
        host, port, flags=TCP_NODELAY)
req_writer.write(('%s\r\n' % new_head).encode('utf-8'))
yield from asyncio.sleep(0.2)
```

Part 6: Relaying Streams

```
@asyncio.coroutine
def relay_stream(reader, writer):
    while True:
        data = yield from reader.read(1024)
        if len(data) == 0:
            break
        writer.write(data)
tasks = [
    asyncio.Task(relay_stream(client_reader, req_writer)),
    asyncio.Task(relay_stream(req_reader, client_writer)),
]
yield from asyncio.wait(tasks)
```

```
@asyncio.coroutine
def relay_stream(reader, writer):
    while True:
        data = yield from reader.read(1024)
        if len(data) == 0:
            break
        writer.write(data)
tasks = [
    asyncio.Task(relay_stream(client_reader, req_writer)),
    asyncio.Task(relay_stream(req_reader, client_writer)),
]
yield from asyncio.wait(tasks)
```

Part 7: Main Loops

```
server = Server(options.host, options.port, options.count)
server.start()
```

```
loop = asyncio.get_event_loop()
asyncio.async(start_warp_server(args.host, args.port))
loop.run_forever()
```

```
loop = asyncio.get_event_loop()
asyncio.async(start_warp_server(args.host, args.port))
loop.run_forever()
```

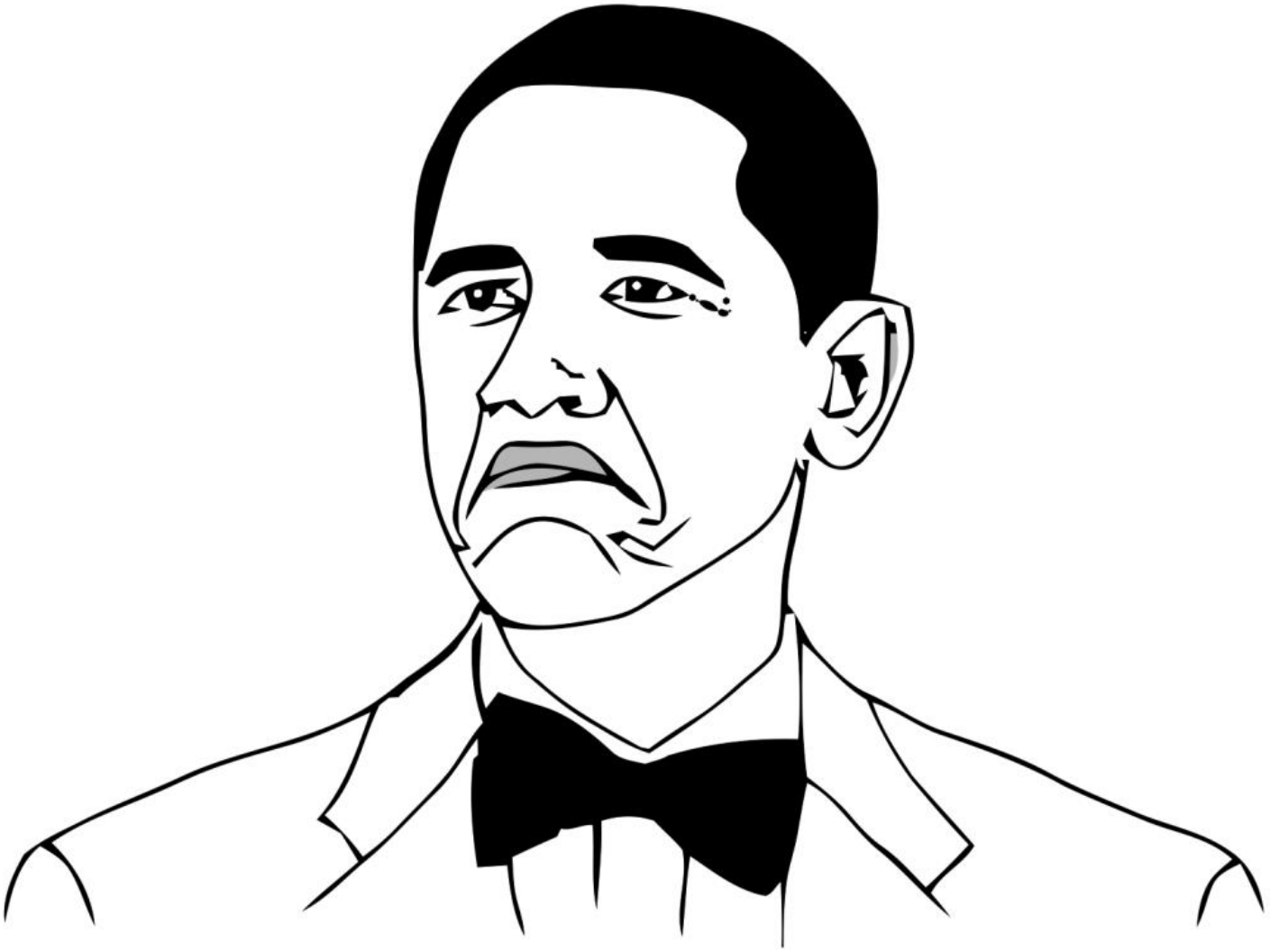
성능 비교

AsyncIO	1202 req/sec
Gevent monkey-patch	997 req/sec
Worker Threads	478 req/sec

AsyncIO	1202 req/sec
Gevent monkey-patch	997 req/sec
Worker Threads	478 req/sec

약 250%의 성능 향상

Gevent 대비 120%



NOT BAD

이 외에도 많은 기능들

- Future
- Delayed calls
- File watcher
- UNIX signals
- Event, Lock, Condition
- Semaphore
- Queue

최종 정리

이해하기 쉽고 직관적인 코드

빠르고 간결하게 작성

다양한 기본 API 제공

250% 이상의 효율

그러나

Python 3.3 ≦

외부 라이브러리 호환성

하지만 이러한 단점에도 불구하고

충분히 사용할 가치가 있다!

```
import asyncio
```

```
import asyncio
```

감사합니다

tony@starks.industries
devunt@starks.industries