openstack™

CLOUD SOFTWARE

redhat.

email: markmc@redhat.com

irc: markmc twitter: @markmc_

Async I/O in Python

We're here to talk about asynchronous I/O in Python
Widely accepted as a weakness of Python
People are working on fixing this in 3.4
Extremely broad, nuanced topic
I expect there's a huge range in people's level of understanding here
People often talk past each other, debating obscure points
The goal here is to learn more about the topic, not debate I/O strategies

~~Waiting~~

You're doing some I/O
It can't complete immediately (e.g. waiting for server to respond)
What do you do now?
If there's anything else useful to be done, surely do that

Async Strategies

Multiple processes
Multiple threads
Single thread, non-blocking I/O
A combination of all three might make sense
We're here to talk about frameworks for the latter

Non-blocking I/O

Usually, if an I/O operation (read/write/etc.) can't complete, you block
Blocking means that the kernel puts you too sleep
If you're sleeping, you don't have the opportunity to do other useful work
Instead, use non-blocking I/O operations
Kernel returns immediately if it can't complete the I/O
You end up with a bunch of in-progress I/O operations
You use select() to have the kernel tell you when one of them can progress

```
01
02
03    import socket
04
05    sock = socket.socket()
06    sock.connect(('localhost', 1234))
07
08
09    sock.send('foo\n' * 10 * 1024 * 1024)
10
11
12
13
14
15
16
17
18
19
20
21
```

Simple example, connect to a server on port 1234, write a bunch of stuff
This send() function won't finish until all the data is written
While you're waiting for the I/O to complete, you can't do anything else
If you're not familiar with berkley sockets ...
socket() creates a socket ... it's just a data structure
connect() initiates the TCP connection with the server
Some handshake packets will go back and forth before this completes
send() takes the data buffer, splits up and sends as TCP packets

```
01
02
03      import socket
04
05      sock = socket.socket()
06      sock.connect(('localhost', 1234))
07      sock.setblocking(0)
08
09      sock.send('foo\n' * 10 * 1024 * 1024)
10
11
12
13
14
15
16
17
18
19
20
21
```

```
01
02
03    import socket
04
05    sock = socket.socket()
06    sock.connect(('localhost', 1234))
07    sock.setblocking(0)
08
09    buf = buffer('foo\n' * 10 * 1024 * 1024)
10    while len(buf):
11        buf = buf[sock.send(buf):]
12
13
14
15
16
17
18
19
20
21
```

```
01    import errno
02    import select
03    import socket
04
05    sock = socket.socket()
06    sock.connect(('localhost', 1234))
07    sock.setblocking(0)
08
09    buf = buffer('foo\n' * 10 * 1024 * 1024)
10    while len(buf):
11        try:
12            buf = buf[sock.send(buf):]
13        except socket.error, e:
14            if e.errno != errno.EAGAIN:
15                raise e
16
17
18
19
20
21
```

```
01    import errno
02    import select
03    import socket
04
05    sock = socket.socket()
06    sock.connect(('localhost', 1234))
07    sock.setblocking(0)
08
09    buf = buffer('foo\n' * 10 * 1024 * 1024)
10    while len(buf):
11        try:
12            buf = buf[sock.send(buf):]
13        except socket.error, e:
14            if e.errno != errno.EAGAIN:
15                raise e
16
17
18
19
20
21            select.select([], [sock], [])
```

```
01    import errno
02    import select
03    import socket
04
05    sock = socket.socket()
06    sock.connect(('localhost', 1234))
07    sock.setblocking(0)
08
09    buf = buffer('foo\n' * 10 * 1024 * 1024)
10    while len(buf):
11        try:
12            buf = buf[sock.send(buf):]
13        except socket.error, e:
14            if e.errno != errno.EAGAIN:
15                raise e
16
17            i = 0
18            while i < 5000000:
19                i += 1
20
21            select.select([], [sock], [])
```

Eventlet

Eventlet is one of several async I/O frameworks for Python
It's magic, to its detriment IMHO
Co-operative co-routine scheduling
Eventlet coroutines are called green threads (from Java)
green threads != threads
Many greenthreads in one OS thread
They are very cheap
We call them coroutines because they cooperatively yield to one another
greenlet allows you to save a stack and switch to another stack

```
01
02    from eventlet.green import socket
03
04
05
06
07
08
09
10
11
12
13    sock = socket.socket()
14    sock.connect(('localhost', 1234))
15    sock.send('foo\n' * 10 * 1024 * 1024)
16
17
18
19
20
21
```

```
01    import eventlet
02    from eventlet.green import socket
03
04    def busy_loop():
05        pass
06
07
08
09
10
11    eventlet.spawn(busy_loop)
12
13    sock = socket.socket()
14    sock.connect(('localhost', 1234))
15    sock.send('foo\n' * 10 * 1024 * 1024)
16
17
18
19
20
21
```

```
01    import eventlet
02    from eventlet.green import socket
03
04    def busy_loop():
05        while True:
06            i = 0
07            while i < 5000000:
08                i += 1
09
10
11    eventlet.spawn(busy_loop)
12
13    sock = socket.socket()
14    sock.connect(('localhost', 1234))
15    sock.send('foo\n' * 10 * 1024 * 1024)
16
17
18
19
20
21
```

```
01    import eventlet
02    from eventlet.green import socket
03
04    def busy_loop():
05        while True:
06            i = 0
07            while i < 5000000:
08                i += 1
09            eventlet.sleep()
10
11    eventlet.spawn(busy_loop)
12
13    sock = socket.socket()
14    sock.connect(('localhost', 1234))
15    sock.send('foo\n' * 10 * 1024 * 1024)
16
17
18
19
20
21
```

Twisted

Eventlet has a scheduler, twisted has an event loop
Event loop is a loop (i.e. while loop) which checks for I/O events
Twisted invokes callbacks for events, rather than scheduling coroutines
Eventlet is confusing because magic
Twisted is confusing because abstractions

```
01
02    from twisted.internet import reactor
03
04
05
06
07
08
09
10
11
12
13
14
15
16
17
18
19
20
21    reactor.run()
```

```
01    from twisted.internet import protocol
02    from twisted.internet import reactor
03
04
05
06
07
08
09
10
11
12    reactor.connectTCP('localhost', 1234,
13                       protocol.ClientFactory())
14
15
16
17
18
19
20
21    reactor.run()
```

```
01    from twisted.internet import protocol
02    from twisted.internet import reactor
03
04    class Test(protocol.Protocol):
05        pass
06
07
08    class TestFactory(protocol.ClientFactory):
09        def buildProtocol(self, addr):
10            return Test()
11
12    reactor.connectTCP('localhost', 1234, TestFactory())
13
14
15
16
17
18
19
20
21    reactor.run()
```

```
01    from twisted.internet import protocol
02    from twisted.internet import reactor
03
04    class Test(protocol.Protocol):
05        def connectionMade(self):
06            self.transport.write('foo\n' * 2 * 1024 * 1024)
07
08    class TestFactory(protocol.ClientFactory):
09        def buildProtocol(self, addr):
10            return Test()
11
12    reactor.connectTCP('localhost', 1234, TestFactory())
13
14
15
16
17
18
19
20
21    reactor.run()
```

```
01    from twisted.internet import protocol
02    from twisted.internet import reactor
03
04    class Test(protocol.Protocol):
05        def connectionMade(self):
06            self.transport.write('foo\n' * 2 * 1024 * 1024)
07
08    class TestFactory(protocol.ClientFactory):
09        def buildProtocol(self, addr):
10            return Test()
11
12    reactor.connectTCP('localhost', 1234, TestFactory())
13
14    def busy_loop():
15        i = 0
16        while i < 5000000:
17            i += 1
18
19    reactor.callLater(0, busy_loop)
20
21    reactor.run()
```

```
01    from twisted.internet import protocol
02    from twisted.internet import reactor
03
04    class Test(protocol.Protocol):
05        def connectionMade(self):
06            self.transport.write('foo\n' * 2 * 1024 * 1024)
07
08    class TestFactory(protocol.ClientFactory):
09        def buildProtocol(self, addr):
10            return Test()
11
12    reactor.connectTCP('localhost', 1234, TestFactory())
13
14    def busy_loop():
15        i = 0
16        while i < 5000000:
17            i += 1
18        reactor.callLater(0, busy_loop)
19    reactor.callLater(0, busy_loop)
20
21    reactor.run()
```

Tulip

Guido's "Async I/O Support Rebooted"
Support for both coroutine and callback models
It's event loop is called an event loop

```
01
02
03
04    import tulip
05
06
07
08
09
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27    event_loop = tulip.get_event_loop()
28
39
30    event_loop.run_forever()
```

```
01
02    import socket
03
04    import tulip
05
06    sock = socket.socket()
07    sock.connect(('localhost', 1234))
08    sock.setblocking(0)
09
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27    event_loop = tulip.get_event_loop()
28
39
30    event_loop.run_forever()
```

```
01    import errno
02    import socket
03
04    import tulip
05
06    sock = socket.socket()
07    sock.connect(('localhost', 1234))
08    sock.setblocking(0)
09
10    buf = memoryview(str.encode('foo\n' * 2 * 1024 * 1024))
11    def do_write():
12       global buf
13       while True:
14           try:
15               buf = buf[sock.send(buf):]
16           except socket.error as e:
17               if e.errno != errno.EAGAIN:
18                   raise e
19               return
20
21
22
23
24
25
26
27    event_loop = tulip.get_event_loop()
28    event_loop.add_writer(sock, do_write)
39
30    event_loop.run_forever()
```

```
01    import errno
02    import socket
03
04    import tulip
05
06    sock = socket.socket()
07    sock.connect(('localhost', 1234))
08    sock.setblocking(0)
09
10    buf = memoryview(str.encode('foo\n' * 2 * 1024 * 1024))
11    def do_write():
12       global buf
13       while True:
14           try:
15               buf = buf[sock.send(buf):]
16           except socket.error as e:
17               if e.errno != errno.EAGAIN:
18                   raise e
19               return
20
21    def busy_loop():
22        i = 0
23        while i < 5000000:
24            i += 1
25        event_loop.call_soon(busy_loop)
26
27    event_loop = tulip.get_event_loop()
28    event_loop.add_writer(sock, do_write)
39    event_loop.call_soon(busy_loop)
30    event_loop.run_forever()
```

```
01    import tulip
02
03
04
05
06
07
08
09
10
11
12
13
14
15
16
17
18
19
20
21    event_loop = tulip.get_event_loop()
22
23    event_loop.run_forever()
```

```
01    import tulip
02
03    class Protocol(tulip.Protocol):
04
05        buf = b'foo\n' * 10 * 1024 * 1024
06
07        def connection_made(self, transport):
08
09            transport.write(self.buf)
10            transport.close()
11
12
13
14
15
16
17
18
19
20
21    event_loop = tulip.get_event_loop()
22
23    event_loop.run_forever()
```

```
01    import tulip
02
03    class Protocol(tulip.Protocol):
04
05        buf = b'foo\n' * 10 * 1024 * 1024
06
07        def connection_made(self, transport):
08
09            transport.write(self.buf)
10            transport.close()
11
12
13
14
15
16
17
18
19
20
21    event_loop = tulip.get_event_loop()
22    tulip.Task(event_loop.create_connection(Protocol, 'localhost', 1234))
23    event_loop.run_forever()
```

```
01    import tulip
02
03    class Protocol(tulip.Protocol):
04
05        buf = b'foo\n' * 10 * 1024 * 1024
06
07        def connection_made(self, transport):
08
09            transport.write(self.buf)
10            transport.close()
11
12        def connection_lost(self, exc):
13            event_loop.stop()
14
15
16
17
18
19
20
21    event_loop = tulip.get_event_loop()
22    tulip.Task(event_loop.create_connection(Protocol, 'localhost', 1234))
23    event_loop.run_forever()
```

```
01    import tulip
02
03    class Protocol(tulip.Protocol):
04
05        buf = b'foo\n' * 10 * 1024 * 1024
06
07        def connection_made(self, transport):
08            event_loop.call_soon(busy_loop)
09            transport.write(self.buf)
10            transport.close()
11
12        def connection_lost(self, exc):
13            event_loop.stop()
14
15    def busy_loop():
16        i = 0
17        while i < 5000000:
18            i += 1
19        event_loop.call_soon(busy_loop)
20
21    event_loop = tulip.get_event_loop()
22    tulip.Task(event_loop.create_connection(Protocol, 'localhost', 1234))
23    event_loop.run_forever()
```

Generators as Coroutines

```
01    import os
02
03    def contents_of_files(dir):
04        ret = []
05        for f in os.listdir(dir):
06            path = os.path.join(dir, f)
07            ret.append((path, file(path).read()))
08        return ret
09
10    for path, contents in contents_of_files(dir):
11        if search_for in contents:
12            print("found", search_for, "in", path)
13            break
```

```
01    import os
02
03    def contents_of_files(dir):
04
05        for f in os.listdir(dir):
06            path = os.path.join(dir, f)
07            yield (path, file(path).read())
08
09
10    for path, contents in contents_of_files(dir):
11        if search_for in contents:
12            print("found", search_for, "in", path)
13            break
```

```
01    def gen():
02        i = 0
03        while i < 2:
04            print(i)
05            yield
06            i += 1
07
08    i = gen()
09    print("yo!")
10    next(i)
11    print("hello!")
12    next(i)
13    print("bye!")
14    try:
15        next(i)
16    except StopIteration:
17        print("stopped")
```

```
01    Task = collections.namedtuple('Task', ['generator', 'wfd', 'idle'])
02
03    running = True
04
05    def quit():
06        global running
07        running = False
08
09    while running:
10        finished = []
11        for n, t in enumerate(tasks):
12            try:
13                next(t.generator)
14            except StopIteration:
15                finished.append(n)
16        map(tasks.pop, finished)
17
18        wfds = [t.wfd for t in tasks if t.wfd]
19        timeout = 0 if [t for t in tasks if t.idle] else None
20
21        select.select([], wfds, [], timeout)
```

```
01    def busy_loop():
02         while True:
03             i = 0
04             while i < 5000000:
05                 i += 1
06             yield
07
08    tasks.append(Task(busy_loop(), wfd=None, idle=True))
09
10
11
12
13
14
15
16
17
18
19
20
21
```

```
01    sock = socket.socket()
02    sock.connect(('localhost', 1234))
03    sock.setblocking(0)
04
05    def write(data):
06        buf = memoryview(data)
07        while len(buf):
08            try:
09                buf = buf[sock.send(buf):]
10            except socket.error as e:
11                if e.errno != errno.EAGAIN:
12                    raise e
13                yield
14
15    def write_stuff():
16        yield from write(b'foon' * 2 * 1024 * 1024)
17        yield from write(b'barn' * 2 * 1024 * 1024)
18        quit()
19
20    tasks.append(Task(write_stuff(), wfd=sock, idle=False))
21
```

References

- My blog - http://blogs.gnome.org/markmc/2013/06/04/async-io-and-python
- Eventlet - http://eventlet.net
- Twisted - http://twistedmatrix.com
- Tulip - http://www.python.org/dev/peps/pep-3156
- yield from - http://www.python.org/dev/peps/pep-0380