Tekst pobrany ze strony
http://aosabook.org/en/500L/a-web-crawler-with-asyncio-coroutines.html
Autorzy: A. Jesse Jiryu Davis, Guido van Rossum.

1 asyncio

1.1 Introduction

Classical computer science emphasizes efficient algorithms that complete computations as quickly as possible. But many networked programs spend their time not computing, but holding open many connections that are slow, or have infrequent events. These programs present a very different challenge: to wait for a huge number of network events efficiently. A contemporary approach to this problem is asynchronous I/O, or ąsync".

This chapter presents a simple web crawler. The crawleris an archetypal async application because it waits for many responses, but does little computation. The more pages it can fetch at once, the sooner it completes. If it devotes a thread to each in-flight request, then as the number of concurrent requests rises it will run out of memory or other thread-related resource before it runs out of sockets. It avoids the need for threads by using asynchronous I/O.

We present the example in three stages. First, we show an async event loop and sketch a crawlerthat uses the event loop with callbacks: it is very efficient, but extending it to more complex problems would lead to unmanageable spaghetti code. Second, therefore, we show that Python coroutines are both efficient and extensible. We implement simple coroutines in Python using generator functions. In the third stage, we use the full-featured coroutines from Python's standard ąsynciołibrary1, and coordinate them using an async queue.

1.2 The Task

A web crawlerfinds and downloads all pages on a website, perhaps to archive or index them. Beginning with a root URL, it fetches each page, parses it for links to unseen pages, and adds these to a queue. It stops when it fetches a page with no unseen links and the queue is empty.

We can hasten this process by downloading many pages concurrently. As the crawlerfinds new links, it launches simultaneous fetch operations for the new pages on separate sockets. It parses responses as they arrive, adding new links to the queue. There may come some point of diminishing returns where too much concurrency degrades performance, so we cap the number of concurrent requests, and leave the remaining links in the queue until some in-flight requests complete.

1.3 The Traditional Approach

How do we make the crawlerconcurrent? Traditionally we would create a thread pool. Each thread would be in charge of downloading one page at a time over a socket. For example, to download a page from xkcd.com:

```
def fetch(url):
    sock = socket.socket()
    sock.connect(('xkcd.com', 80))
    request = 'GET {} HTTP/1.0\r\nHost: xkcd.com\r\n\r\n'.format(url)
    sock.send(request.encode('ascii'))
    response = b''
    chunk = sock.recv(4096)
    while chunk:
```

```
            response += chunk
            chunk = sock.recv(4096)

        # Page is now downloaded.
        links = parse_links(response)
        q.add(links)
```

By default, socket operations are blocking: when the thread calls a method like `connect` or `recv`, it pauses until the operation completes. Consequently to download many pages at once, we need many threads. A sophisticated application amortizes the cost of thread-creation by keeping idle threads in a thread pool, then checking them out to reuse them for subsequent tasks; it does the same with sockets in a connection pool.

And yet, threads are expensive, and operating systems enforce a variety of hard caps on the number of threads a process, user, or machine may have. On Jesse's system, a Python thread costs around 50k of memory, and starting tens of thousands of threads causes failures. If we scale up to tens of thousands of simultaneous operations on concurrent sockets, we run out of threads before we run out of sockets. Per-thread overhead or system limits on threads are the bottleneck.

In his influential article "The C10K problem", Dan Kegel outlines the limitations of multi-threading for I/O concurrency. He begins,

> It's time for web servers to handle ten thousand clients simultaneously, don't you think? After all, the web is a big place now. Kegel coined the term C10K"in 1999. Ten thousand connections sounds dainty now, but the problem has changed only in size, not in kind. Back then, using a thread per connection for C10K was impractical. Now the cap is orders of magnitude higher. Indeed, our toy web crawlerwould work just fine with threads. Yet for very large scale applications, with hundreds of thousands of connections, the cap remains: there is a limit beyond which most systems can still create sockets, but have run out of threads. How can we overcome this?

1.4 Async

Asynchronous I/O frameworks do concurrent operations on a single thread using non-blocking sockets. In our async crawler, we set the socket non-blocking before we begin to connect to the server:

```
sock = socket.socket()
sock.setblocking(False)
try:
    sock.connect(('xkcd.com', 80))
except BlockingIOError:
    pass
```

Irritatingly, a non-blocking socket throws an exception from `connect`, even when it is working normally. This exception replicates the irritating behavior of the underlying C function, which sets `errno` to `EINPROGRESS` to tell you it has begun.

Now our crawlerneeds a way to know when the connection is established, so it can send the HTTP request. We could simply keep trying in a tight loop:

```
request = 'GET {} HTTP/1.0\r\nHost: xkcd.com\r\n\r\n'.format(url)
encoded = request.encode('ascii')
```

```
while True:
    try:
        sock.send(encoded)
        break  # Done.
    except OSError as e:
        pass

print('sent')
```

This method not only wastes electricity, but it cannot efficiently await events on multiple sockets. In ancient times, BSD Unix's solution to this problem was `select`, a C function that waits for an event to occur on a non-blocking socket or a small array of them. Nowadays the demand for Internet applications with huge numbers of connections has led to replacements like `poll`, then `kqueue` on BSD and `epoll` on Linux. These APIs are similar to `select`, but perform well with very large numbers of connections.

Indeks