1.介绍

python3自带的异步编程模块, 主要支持异步的网络IO操作, 异步运行子进程, 异步操作队列等功能;

2.使用示例

```python
import asyncio
import time
import sys
import logging

logging.basicConfig(level=logging.DEBUG, stream=sys.stdout)


async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)


async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))  # 3.6用ensure_future

    task2 = asyncio.create_task(
        say_after(2, 'world'))
    # time.sleep(1)

    print(f"started at {time.strftime('%X')}")

    await task1
    await task2
    print(f"finished at {time.strftime('%X')}")

asyncio.run(main(), debug=True)  # 3.7
#loop = asyncio.get_event_loop()
#loop.run_until_complete(main())
#loop.close()
```

(1)通过async关键字定义协程;

(2)通过asyncio.create_task()来定义异步任务;

(3)通过await关键字来异步等待;

(4)通过asyncio.run()来启动协程;

2.2 异步网络通信

```python
# client
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()

asyncio.run(tcp_echo_client('Hello World!'))

# server
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addr = server.sockets[0].getsockname()
    print(f'Serving on {addr}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

2.3 异步执行命令行指令

```python
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(
        cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()

    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]\n{stdout.decode()}')
    if stderr:
        print(f'[stderr]\n{stderr.decode()}')

asyncio.run(run('ls /pythonfiles'))
```

2.4 异步操作队列

asyncio.Queue(maxsize=0, *, loop=None)
和python内置的queue模块使用方法相同，区别是put()， get()等方法可异步执行；

```python
import asyncio

async def worker1(queue):
    print('aaaa')
    a = await queue.get()
    print(a)

async def worker2(queue):
    await asyncio.sleep(2)
    print('put to queue:')
    await queue.put(3)

async def main():
    queue = asyncio.Queue()
    task1 = asyncio.create_task(worker1(queue))
    task2 = asyncio.create_task(worker2(queue))
    await task1
    await task2

asyncio.run(main())
```

3.概念

3.1 awaitable

可以用在await表达式中的对象, 主要有3种: coroutines, Tasks, 和 Futures.

tasks: asyncio.create_task()返回的对象;

Future是一个特殊的低级别等待对象，它表示异步操作的最终结果。
当等待Future对象时，它意味着协程将会等到Future在其他地方解析。

3.2 task对象

> Tasks are used to run coroutines in event loops. If a coroutine awaits on a Future, the Task suspends the execution of the coroutine and waits for the completion of the Future. When the Future is done, the execution of the wrapped coroutine resumes.

> Use the high-level asyncio.create_task() function to create Tasks, or the low-level loop.create_task() or ensure_future() functions. Manual instantiation of Tasks is discouraged.

task对象用于在事件循环内运行, 管理协程;通常通过asyncio.create_task(), loop.create_task() 或者 ensure_future()来生成, 具有如下几个方法:

(1) **cancel()**

取消协程的运行, 会抛出一个CanceldeError;

(2) **canceled()/done()**

task是否取消/完成;

(3) **result()/exception()**

返回task运行异常/结果, 或者CanceledError, InvalidStateError;

(4) **add_done_callback(callback, *, context=None) / remove_done_callback(callback)**

添加/移除回调函数, task运行完成后执行;

3.3 future对象

> Future objects are used to bridge low-level callback-based code with high-level async/await code.

是连接底层的基于回调的代码和上层async/await的桥梁, 它代表一个异步操作的最终结果;

(1)result() / exception()

> Return the result of the Future.

获取future中的结果/异常;

(2)set_result(result)

> Mark the Future as done and set its result.

标记future已完成并设置future的结果;

(3)set_exception(exception)

> Mark the Future as done and set an exception.

标记future已完成并设置一个异常;

(4)done() / cancelled()

(5)add_done_callback(callback, *, context=None) \ remove_done_callback(callback)
添加或移除future完成时执行的回调函数, context参数可指定运行时的上下文;

(6)cancel()
取消future对象的等待并开始运行回调函数;

(7)get_loop()

> Return the event loop the Future object is bound to.

获取future绑定的事件循环对象;

(8)asyncio.isfuture(obj)

判断一个对象是否为future对象；

(9)asyncio.ensure_future(obj, *, loop=None)

创建task对象；

(10)asyncio.wrap_future(future, *, loop=None)

将一个concurrent.futures.Future对象封装为asyncio.Future对象；

3.4 event loop

> The event loop is the core of every asyncio application. Event loops run asynchronous tasks and callbacks, perform network IO operations, and run subprocesses.

asyncio用event loop来运行异步任务和回调函数，建立或监听网络连接，运行子程序；

(1)loop.run_until_complete(future)

> Run until the future (an instance of Future) has completed.

运行事件循环直到future执行完成；

(2)loop.run_forever()

> Run the event loop until stop() is called.

运行事件循环直到stop()执行；

(3)loop.stop()/loop.is_running()/loop.is_closed()/loop.close()

> If stop() is called while run_forever() is running, the loop will run the current batch of callbacks and then exit.

(4)loop.shutdown_asyncgens()

> Schedule all currently open asynchronous generator objects to close with an aclose() call. After calling this method, the event loop will issue a warning if a new asynchronous generator is iterated. This should be used to reliably finalize all scheduled asynchronous generators.

3.4.1 添加回调函数

(1)loop.call_soon(callback, *args, context=None)

> Schedule a callback to be called with args arguments at the next iteration of the event loop.

在事件循环的下一次迭代中执行回调；
loop.call_soon_threadsafe(callback, *args, context=None)线程安全版

```python
import asyncio

def hello_world(loop):
    """A callback to print 'Hello World' and stop the event loop"""
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

(2)loop.call_later(delay, callback, *args, context=None)
loop.call_at(when, callback, *args, context=None)

设置回调运行的时间;

```python
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.get_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

3.4.2 创建future，task对象

(1)loop.create_future() / loop.create_task(coro)
创建future/task对象；

3.4.3 创建网络连接

(1)loop.create_connection(protocol_factory, host=None, port=None, *, ssl=None, family=0, proto=0, flags=0, sock=None, local_addr=None, server_hostname=None, ssl_handshake_timeout=None)

以给定参数建立一个tcp连接，返回（transport, protocol)的元组；

(2)loop.create_datagram_endpoint(protocol_factory, local_addr=None, remote_addr=None, *, family=0, proto=0, flags=0, reuse_address=None, reuse_port=None, allow_broadcast=None, sock=None)

返回（transport, protocol)的元组,用于udp通信；

(3)loop.create_unix_connection(protocol_factory, path=None, *, ssl=None, sock=None, server_hostname=None, ssl_handshake_timeout=None)

创建基于Unix文件的连接，返回（transport, protocol)的元组；

3.4.4 创建network server

(1)loop.create_server(protocol_factory, host=None, port=None, *, family=socket.AF_UNSPEC, flags=socket.AI_PASSIVE, sock=None, backlog=100, ssl=None, reuse_address=None, reuse_port=None, ssl_handshake_timeout=None, start_serving=True)

创建tcp server监听对应的端口, 返回一个server对象;

(2)loop.create_unix_server(protocol_factory, path=None, *, sock=None, backlog=100, ssl=None, ssl_handshake_timeout=None, start_serving=True))

按对应的文件创建一个unix server, 返回一个server对象;

(3)loop.connect_accepted_socket(protocol_factory, sock, *, ssl=None, ssl_handshake_timeout=None)

将一个已经连接的socket对象封装为asyncio可处理的transport, protocol对象;

3.4.5 发送文件

(1)loop.sendfile(transport, file, offset=0, count=None, *, fallback=True)

方法内部使用os.sendfile()发送文件;

3.4.6 监听文件描述符

(1)loop.add_reader(fd, callback, *args) / loop.remove_reader(fd)

> Start monitoring the fd file descriptor for read availability and invoke callback with the specified arguments once fd is available for reading.)

监听或取消对文件描述符, 如果文件描述符可读,则执行callback;

(2)loop.add_writer(fd, callback, *args) / loop.remove_writer(fd)

> Start monitoring the fd file descriptor for write availability and invoke callback with the specified arguments once fd is available for writing.)

监听或取消对文件描述符, 如果文件描述符可写入,则执行callback;

```
import asyncio
from socket import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()

loop = asyncio.get_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())

    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)

    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # Run the event loop
    loop.run_forever()
finally:
    # We are done. Close sockets and the event loop.
    rsock.close()
    wsock.close()
    loop.close()
```

3.4.7 异步socket

(1)loop.sock_recv(sock, nbytes)

> Receive up to nbytes from sock. Asynchronous version of socket.recv().

(2)loop.sock_sendall(sock, data)

> Send data to the sock socket. Asynchronous version of socket.sendall().

(3)loop.sock_connect(sock, address)

> Connect sock to a remote socket at address.
> Asynchronous version of socket.connect().

(4)loop.sock_accept(sock)

> Accept a connection. Modeled after the blocking socket.accept() method.

(5)loop.sock_sendfile(sock, file, offset=0, count=None, *, fallback=True)

> Send a file using high-performance os.sendfile if possible. Return the total number of bytes sent.
> Asynchronous version of socket.sendfile().)

3.4.8 在线程或进程池中执行代码

(1)loop.run_in_executor(executor, func, *args)

> Arrange for func to be called in the specified executor.
> The executor argument should be an concurrent.futures.Executor instance. The default executor is used if executor is None.

```
import asyncio
import concurrent.futures

def blocking_io():
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    ## Options:
    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(
        None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

asyncio.run(main())
```

3.4.9 执行命令行指令

(1)loop.subprocess_exec(protocol_factory, *args, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, **kwargs)

> Create a subprocess from one or more string arguments specified by args.
> This is similar to the standard library subprocess.Popen class called with shell=False; Returns a pair of (transport, protocol)

(2) loop.subprocess_shell(protocol_factory, cmd, *, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, **kwargs)

> Create a subprocess from cmd; This is similar to the standard library subprocess.Popen class called with shell=True.

3.5 Handle/TimeHandle对象

由loop.call_soon(), loop.call_soon_threadsafe()返回, 可用于取消callback（3.7新增）:

(1) cancel()

> Cancel the callback. If the callback has already been canceled or executed, this method has no effect.

(2) cancelled()

> Return True if the callback was cancelled.

3.6 server对象

由loop.create_server(), loop.create_unix_server()等方法返回，可对创建的server进行一些控制；
(1)start_serving()

> Start accepting connections.
> This method is idempotent, so it can be called when the server is already being serving

开始接受连接， 主要在loop.create_server()和asyncio.start_server()的start_serving参数设为false时使用；

(2)serve_forever()

> Start accepting connections until the coroutine is cancelled. Cancellation of serve_forever task causes the server to be closed.

开始接受连接，直到协程被取消；

```
async def client_connected(reader, writer):
    # Communicate with the client with
    # reader/writer streams.  For example:
    await reader.readline()

async def main(host, port):
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))
```

(3) wait_closed()

> Wait until the close() method completes.

异步等待server关闭；

3.7 transport和protocol

> At the highest level, the transport is concerned with how bytes are transmitted, while the protocol determines which bytes to transmit (and to some extent when).

> A different way of saying the same thing: a transport is an abstraction for a socket (or similar I/O endpoint) while a protocol is an abstraction for an application, from the transport’s point of view.

> the protocol calls transport methods to send data, while the transport calls protocol methods to pass it data that has been received.

transport决定数据如何传输(socket)， protocol决定传输什么数据以及何时传递(application)；transport和protocol成对使用，protocol调用transport的方法来发送数据，transport调用protocol的方法来将接受到的数据传递给protocol;

3.7.1 transport

Transport, DatagramTransport, SubprocessTransport

(1)close() / is_closing() / abort()

close()关闭transport，然后调用protocol.connection_lost()；如果有数据正在传输，等待传输完成;
abort()立即关闭transport， 正在传输的数据会丢失；

(2)set_protocol(protocol) / get_protocol()
设置或获取protocol；

(3)pause_reading() / resume_reading()

暂停接收数据/恢复接受数据， 用于控制数据接收； 数据接收完成会调用 protocol.data_received()

(4)can_write_eof() / write_eof()

是否有can_write_eof()方法/在发送完所有数据后关闭transport的输入流；

(5)set_write_buffer_limits(high=None, low=None) / get_write_buffer_limits()

> Set the high and low watermarks for write flow control.

> These two values (measured in number of bytes) control when the protocol's protocol.pause_writing() and protocol.resume_writing() methods are called. If specified, the low watermark must be less than or equal to the high watermark. Neither high nor low can be negative.

> pause_writing() is called when the buffer size becomes greater than or equal to the high value. If writing has been paused, resume_writing() is called when the buffer size becomes less than or equal to the low value.

设置写入数据的缓冲区的上下限；当缓冲数据超过high值会调用protocol的pause_writing(), 当缓冲数据大小低于low值将会调用protocol的resume_writing();

(6)write(data) / writelines(list_of_data)

向transport异步写入数据;

(7)DatagramTransport.sendto(data, addr=None)

发送数据报格式的数据;

(8)SubprocessTransport.get_pipe_transport(fd)

> Return the transport for the communication pipe corresponding to the integer file descriptor fd:

- 0: readable streaming transport of the standard input (stdin), or None if the subprocess was not created with stdin=PIPE
- 1: writable streaming transport of the standard output (stdout), or None if the subprocess was not created with stdout=PIPE
- 2: writable streaming transport of the standard error (stderr), or None if the subprocess was not created with stderr=PIPE

3.7.2 protocol

protocol用于提供发送的数据,处理接收到的数据, 主要以回调的形式调用;
(1)connection_made(transport)

> Called when a connection is made.

当连接建立时调用;
(2)connection_lost(exc)

> Called when the connection is lost or closed.

当连接丢失或关闭时调用,exec是异常对象或None;

(3)pause_writing()

> Called when the transport’s buffer goes over the high watermark.

(4)resume_writing()

> Called when the transport’s buffer drains below the low watermark.

(5)Protocol.data_received(data)

> Called when some data is received. data is a non-empty bytes object containing the incoming data.

当transport接收到数据时调用;

(6)Protocol.eof_received()

> Called when the other end signals it won’t send any more data (for example by calling transport.write_eof(), if the other end also uses asyncio).

当收到另一端停止发送数据的信号时调用, 停止接收数据;

(7)DatagramProtocol.datagram_received(data, addr)

当transport接收到数据时调用;

(8)SubprocessProtocol.pipe_data_received(fd, data)

> Called when the child process writes data into its stdout or stderr pipe.
> fd is the integer file descriptor of the pipe.

当子进程将数据写入stdout或stderr时调用;

(9)SubprocessProtocol.pipe_connection_lost(fd, exc)

> Called when one of the pipes communicating with the child process is closed.
> fd is the integer file descriptor that was closed.

当和子进程通信的管道关闭时调用;

(10)SubprocessProtocol.process_exited()

> Called when the child process has exited.

当子进程退出时调用;

3.8 StreamReader和StreamWriter

3.8.1 StreamReader

(1)read(n=-1) / readexactly(n)

读取n字节数据， -1代表读取完, readexactly表示如果不到n字节会抛出错误；

(2)readline()

> Read one line, where “line” is a sequence of bytes ending with \n.

(3)readuntil(separator=b'\n')

> Read data from the stream until separator is found.

3.8.2 StreamWriter

(1)write(data) / writelines(data)
发送数据，和drain()一起连用, 进行缓冲数据流的控制；

```
writer.write(data)
await writer.drain()
```

(2)drain()

> Wait until it is appropriate to resume writing to the stream

等待直到缓冲区可以继续写入

(3)close()

> Close the stream.

关闭连接，和wait_closed()连用；

3.9 Process 对象

和subprocess.Popen对象相似；

(1)communicate(input=None)

> Interact with process:

- send data to stdin (if input is not None);
- read data from stdout and stderr, until EOF is reached;

- wait for process to terminate.

返回数据： (stdout_data, stderr_data).

4. api

(1)asyncio.run(coro, *, debug=False)

> This function runs the passed coroutine, taking care of managing the asyncio event loop and finalizing asynchronous generators.

(2)asyncio.create_task(coro)

> Wrap the coro coroutine into a Task and schedule its execution. Return the Task object.

将协程封装成一个task对象

(3)asyncio.sleep(delay, result=None, *, loop=None)

异步sleep

(4)asyncio.gather(*aws, loop=None, return_exceptions=False)

> Run awaitable objects in the aws sequence concurrently.

异步执行多个task, 返回一个执行结果的列表; return_exceptions决定是否返回抛出的异常;

```python
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({i})...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")

async def main():
    # Schedule three calls *concurrently*:
    await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )

asyncio.run(main())

# Expected output:
#
#     Task A: Compute factorial(2)...
#     Task B: Compute factorial(2)...
#     Task C: Compute factorial(2)...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3)...
#     Task C: Compute factorial(3)...
#     Task B: factorial(3) = 6
#     Task C: Compute factorial(4)...
#     Task C: factorial(4) = 24
```

(5)asyncio.shield(aw, *, loop=None)

> Protect an awaitable object from being cancelled.

```python
res = await shield(something())
```

避免任务被取消;
用于需要取消协程的情况,如果主协程被取消,那么shield可以避免内部的任务被取消;

(6)asyncio.wait_for(aw, timeout, *, loop=None)

> Wait for the aw awaitable to complete with a timeout.

异步执行的超时控制, 当超时发生将抛出异常;

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except asyncio.TimeoutError:
        print('timeout!')

asyncio.run(main())

# Expected output:
#
#     timeout!
```

(7)asyncio.wait(aws, *, loop=None, timeout=None, return_when=ALL_COMPLETED)

> Run awaitable objects in the aws set concurrently and block until the condition specified by return_when.
>
> Returns two sets of Tasks/Futures: (done, pending).

运行多个task并且在满足给定条件或达到time out时返回已完成和未完成task的集合;

return_when参数:

Constant	Description
FIRST_COMPLETED	The function will return when any future finishes or is cancelled.
FIRST_EXCEPTION	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to ALL_COMPLETED.
ALL_COMPLETED	The function will return when all futures finish or are cancelled.

```
async def foo():
    return 42

async def main():
    task = asyncio.create_task(foo())
    done, pending = await asyncio.wait({task})

    if task in done:
        # do something

asyncio.run(main(), debug=True)
```

(8)asyncio.as_completed(aws, *, loop=None, timeout=None)

> Run awaitable objects in the aws set concurrently. Return an iterator of Future objects. Each Future object returned represents the earliest result from the set of the remaining awaitables.

并发执行多个任务,并返回future对象的迭代器;

(9)asyncio.run_coroutine_threadsafe(coro, loop)

> Submit a coroutine to the given event loop. Thread-safe.
>
> Return a concurrent.futures.Future to wait for the result from another OS thread.

在不同线程中把协程放入给定的事件循环;

(10)asyncio.current_task(loop=None)

> Return the currently running Task instance, or None if no task is running.

返回当前正在运行的task对象;

(11)asyncio.all_tasks(loop=None)

> Return a set of not yet finished Task objects run by the loop.

返回事件循环中所有的task对象;

(12)asyncio.iscoroutine(obj)

判断一个对象是否是协程对象

(13)asyncio.iscoroutinefunction(func)

判断一个对象是否是协程函数(async def 或者 @asyncio.coroutine)

(14)asyncio.open_connection(host=None, port=None, *, loop=None, limit=None, ssl=None, family=0, proto=0, flags=0, sock=None, local_addr=None, server_hostname=None, ssl_handshake_timeout=None)

> Establish a network connection and return a pair of (reader, writer) objects.

利用给定的参数创建tcp连接，返回reader和writer对象；

(15)asyncio.start_server(client_connected_cb, host=None, port=None, *, loop=None, limit=None, family=socket.AF_UNSPEC, flags=socket.AI_PASSIVE, sock=None, backlog=100, ssl=None, reuse_address=None, reuse_port=None, ssl_handshake_timeout=None, start_serving=True)¶

> Start a socket server.

利用给定参数开启一个tcp server， 当有client连接成功时调用client_connected_cb， 并传入参数reader和writer；

(16)asyncio.create_subprocess_exec(program, *args, stdin=None, stdout=None, stderr=None, loop=None, limit=None, **kwds)

> Create a subprocess.

创建子进程运行program指定的程序；

(17)asyncio.create_subprocess_shell(cmd, stdin=None, stdout=None, stderr=None, loop=None, limit=None, **kwds)

> Run the cmd shell command.

运行shell命令；

5 Generator-based Coroutines

```python
import asyncio
import time


@asyncio.coroutine
def say_after(delay, what, task=0):
    yield from asyncio.sleep(delay)
    print(what)

@asyncio.coroutine
def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world', task1))
    print(f"started at {time.strftime('%X')}")
    yield from task1
    yield from task2
    print(f"finished at {time.strftime('%X')}")
    return 123

#a = asyncio.run(main())
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```