Asyncio Internals

Wing

October 17, 2020

Table of contents

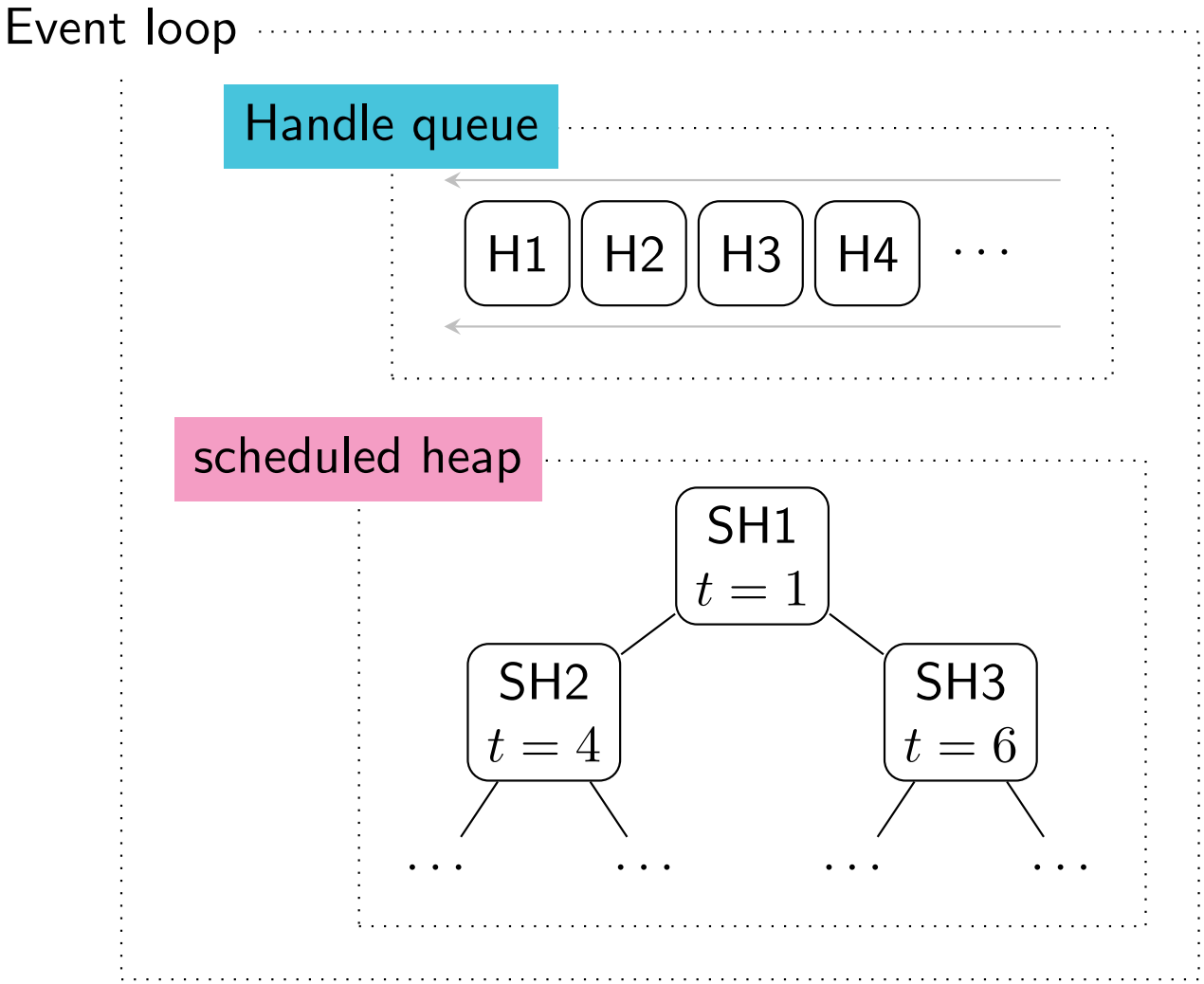
Generator internals

Secret: In cpython, generator = coroutine!

- `gen.send(?)`
- `next(gen)` ≡ `gen.send(None)`
- `gen.throw(exc)`
- `yield` ⇒ "pausing" of coroutine
- closure
- stack frame in heap

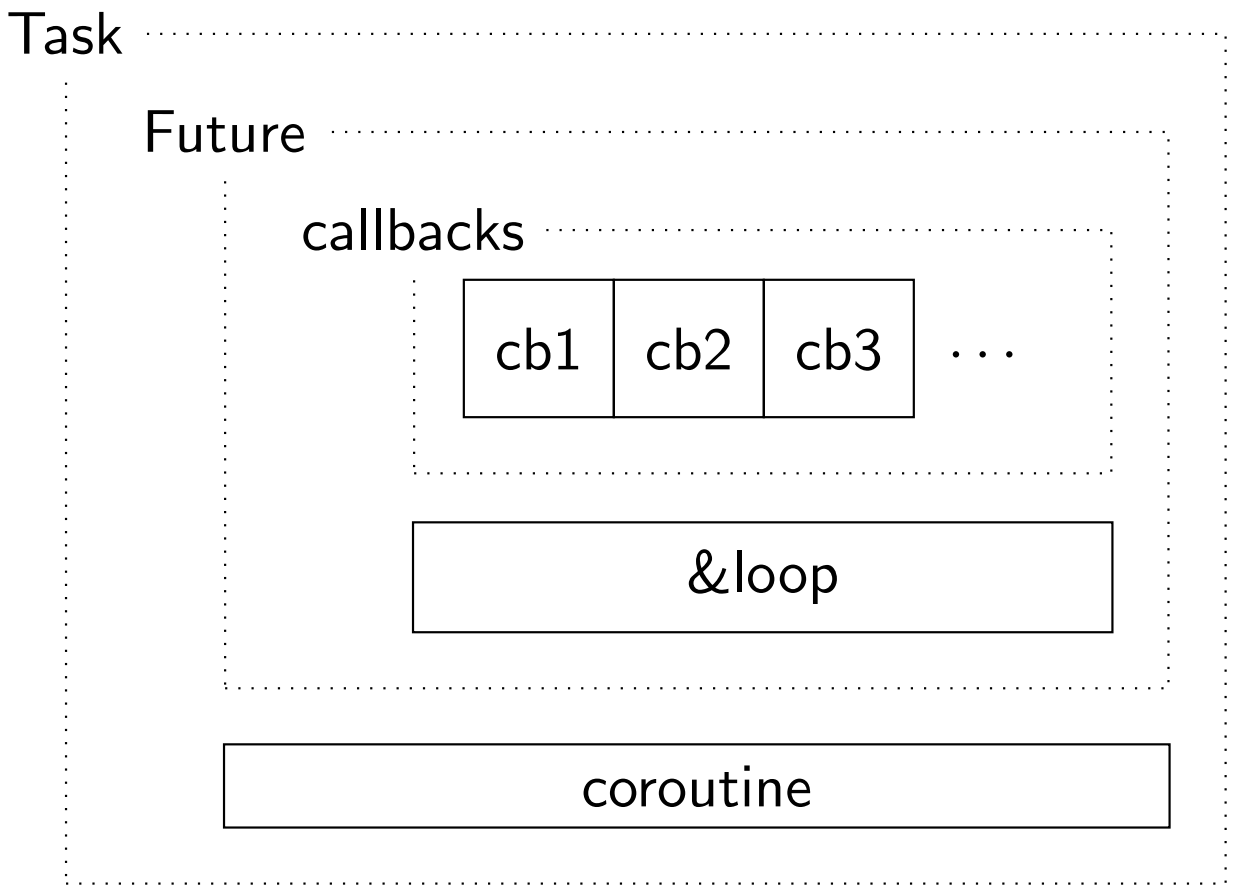
demo `gen_send.py`

Event loop



`Handle` **wraps** `Task.__step`

Task



asyncio.run(main)

```
def run(main):
    loop = new_event_loop()
    return loop.run_until_complete(main)

class Loop:
    def run_until_complete(coro_or_fut):
        task = ensure_future(coro_or_fut)
        task.add_done_callback(<<stop loop>>)
        loop.run_forever()
        return task.result()

def ensure_future(coro_or_fut):
    if isinstance(coro_or_fut, Future)
        return coro_or_fut
    else:
        # coro
        return Task(coro_or_fut, loop=self)
```

Task

```
class Task(Future):
    def __init__(self, coro, loop, ...):
        super().__init__(loop)
        ...
        self._coro = coro
        self._loop.call_soon(self.__step, ...)
```

Future

```
class Future:
    def __init__(self, loop):
        self._loop = loop

    def __iter__(self):
        if not self.done():
            yield self  # future is blocking
        return self.result()

    def add_done_callback(self, fn, ...):
        if self.done():
            self._loop.call_soon(fn, self)
        else:
            self._callbacks.append((fn,))
    ...
```

Future

```
class Future:
    ...

    def set_result(self, result):  # similar for set_exception
        self._result = result
        self._state = _FINISHED  # done
        self.__schedule_callbacks()

    def __schedule_callbacks(self):
        callbacks = self._callbacks[:]
        self._callbacks[:] = []  # clear callbacks
        for callback in callbacks:
            self._loop.call_soon(callback, self)

    def result(self):
        if self._exception is not None:
            raise self._exception
        return self._result
```

`__schedule_callbacks()` effectively moves all callbacks to Handle queue

loop.call_soon(callback)

```
class Loop:
    def call_soon(self, callback):
        handle = Handle(callback)

        Handle queue.append(handle)

        return handle
```

loop.run_forever()

```
class Loop:
    def run_forever(self):
        while True:
            self._run_once()
            if self._stopping:
                break

    def _run_once(self):
        ...
```

loop._run_once()

def `_run_once()`:

$$\texttt{timeout} = \begin{cases} 0, & \text{if Handle queue is not empty} \\ \text{minimal timeout}, & \text{if scheduled heap is not empty} \\ \text{None}, & \text{otherwise} \end{cases}$$

```
// block if timeout is None
ev_list = self._selector.select(timeout)
self._process_events(ev_list)
```

Handle queue += handles from scheduled heap which the time is up

`handles` = pop all from Handle queue

for `handle: handles` **do**
  `handle._run()` // runs `task.__step`

Task.__step(exc)

```
def __step(exc):
    coro = self._coro
    try:
        result = coro.send(None)
    except StopIteration as exc:
        self.set_result(exc.value)
    except BaseException as exc:
        self.set_exception(exc)
    else:
        if result <<is a blocking future>>:
            result.add_done_callback(self.__wakeup)
    elif result is None:  # bare yield used
        self._loop.call_soon(self.__step)

def __wakeup(self, future):
    # check if cancelled or errored
    ...
    self.__step()
```

demo

- `run_once_demo.py`
- `timeout_zero_future.py`

References


Talk by Saúl Ibarra Corretgé in PyGrunn 2014
https://www.youtube.com/watch?v=HppNu0-ANYw