CONCURRENT.FUTURES

HISTORY, USAGE, INTERNAL, FUTURE

by laike9m

ABOUT ME

```
{
    'name': '左遥'
    'work': '中科院计算所研究生'
    'website': 'http://laike9m.com'
}
```

THE WORLD IS BECOMING ASYNCRONOUS

Java, C++, Node.js, Go,...

But what is Async?

ASYNCRONOUS PROGRAMMING

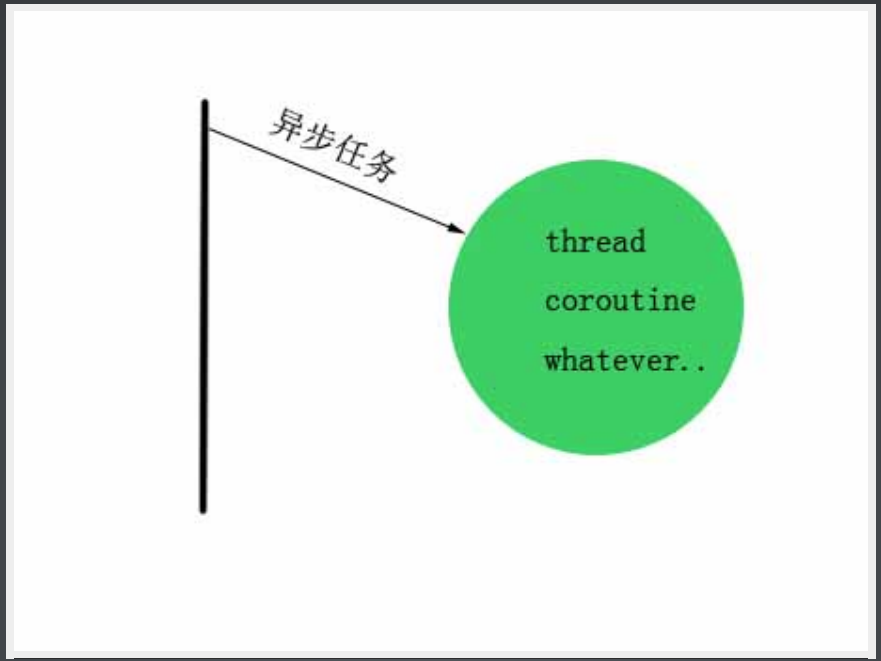
对于一个操作，我们不想等它完成（可能要等很久）
所以把它放在一边，但通过某种方法让它继续运行
我们就可以继续做别的事

HOW?

multithreading
coroutine
singlethread+eventloop

异步指的是程序的执行方式，多线程指的是底层实现

When will the task end?



METHOD ONE: CALLBACK

```
request_some_url('example.com', function(resp){
    // callback func
});
```

Javascript/Node.js use callback+eventloop

FUTURE OBJECT

WHAT IS FUTURE OBJECT?

1. Future Object encapsulates(封装) the asynchronous execution of a callable.
2. asynchronous result which might not be immediately available.

封装代码的异步执行，之后可以通过`Future`获取结果

METHOD TWO: FUTURE

C++

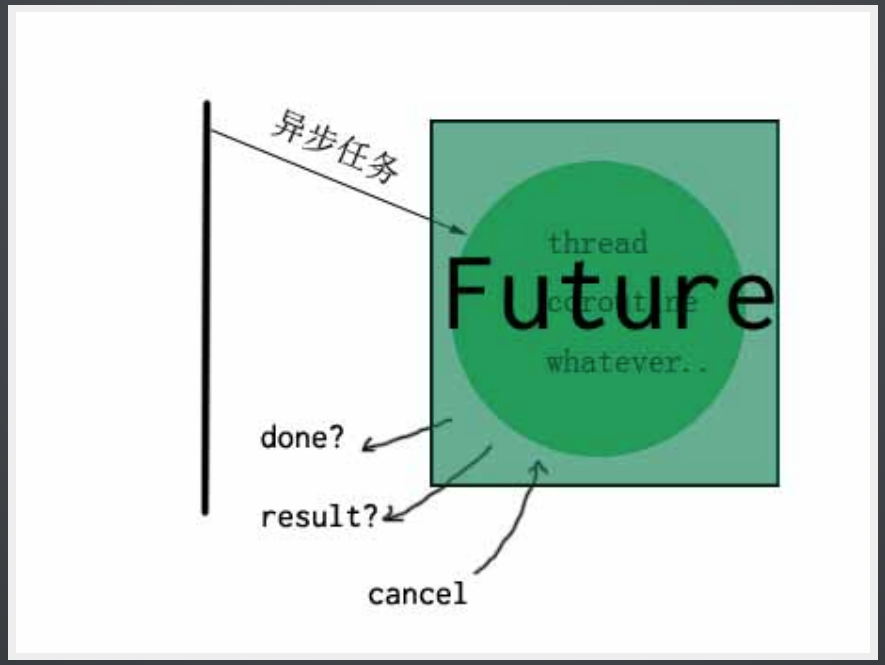
```
#include <future>
#include <iostream>
using namespace std;

void called_from_async() {
    // whatever
}

int main() {
  future<void> f(async(called_from_async));

  //other computation

  f.get();
  return 0;
}
```

Java

```
Future<String> Download(URL url){
    // Download from url
}

Future<String> f = Download(new URL("http://www.example.com"));

//other computation

String contents = f.get();
```

PYTHON'S FUTURE

```
Future.result(timeout=None)    # wait timeout or forever
Future.cancel()                # cancel task
Future.done()                  # whether the task has completed
Future.add_done_callback(fn)   # 可以添加多个回调
```



CONCURRENT.FUTURES!!

```
import time
from urllib.request import urlopen
from concurrent.futures import ThreadPoolExecutor

def web_crawl(url):
    r = urlopen(url)
    return r.code

executor = ThreadPoolExecutor(max_workers=2)

a = executor.submit(web_crawl, 'foo.com')  # a is Future Object
b = executor.submit(web_crawl, 'bar.com')  # b is Future Object

print(a.result())
print(b.result())
```

EXECUTOR

Executor是对Future的进一步封装

```
executor1 = ThreadPoolExecutor(max_workers=2)
executor2 = ProcessPoolExecutor(max_workers=2)
executor2 = ProcessPoolExecutor()
```

submit一个任务给Executor, 返回Future Object

```
future = executor.submit(fn)
```

EXECUTOR.MAP()

```
def is_odd_number(number):
    return number % 2

executor = ProcessPoolExecutor()
it = executor.map(is_odd_number, [1, 2], timeout=1)
next(it)
next(it)
```

返回一个包含函数执行结果迭代器
map不阻塞, 当执行next(it)的时候才会阻塞
等价于

```
f1 = executor.submit(is_odd_number, 1)
f2 = executor.submit(is_odd_number, 2)
f1.result(timeout=1)
f1.result(timeout=1)
```

EXECUTOR.SHUTDOWN()

等待所有任务执行完毕, 然后释放掉Executor
调用了shutdown之后, 就不能再submit新任务了

```
def wait_10_seconds(number):
    time.sleep(10)

executor1 = ThreadPoolExecutor(2)
executor1.submit(wait_10_seconds)
executor1.shutdown(wait=False)   # don't wait, execution continues

executor2 = ThreadPoolExecutor(2)
executor2.submit(wait_10_seconds)
executor2.shutdown(wait=True)    # wait for 10 seconds here
...
```

EXECUTOR.SHUTDOWN()

可以用context manager替代shutdown, 此时wait=True

```
def wait_10_seconds(number):
    time.sleep(10)

with ThreadPoolExecutor(2) as e:
    e.submit(wait_10_seconds)

...
```

No magic

```
class Executor(object):

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.shutdown(wait=True)
        return False
```

两个操作FUTURE的函数

CONCURRENT.FUTURES.WAIT()

```
r = concurrent.futures.wait(fs, timeout=None, return_when=ALL_COMPLETED)
```

返回值 r

- r.done: {所有已执行完的Future object}
- r.not_done: {所有未完成的Future object}

```python
import time
from concurrent.futures import ThreadPoolExecutor, FIRST_COMPLETED, wait
def wait_n_seconds(n):
  time.sleep(n)

def wait_and_see_result(fs):
    time.sleep(1)
    result = wait(fs, return_when=FIRST_COMPLETED)
    print("done: ", result.done)
    print("not done: %s\n" % result.not_done)

with ThreadPoolExecutor(3) as e:
  f1 = e.submit(wait_n_seconds, 1)
  f2 = e.submit(wait_n_seconds, 2)
  f3 = e.submit(wait_n_seconds, 3)
  fs = [f1, f2, f3]
  wait_and_see_result(fs)
    # done: {<Future at 0x21de080 state=finished returned NoneType>}
    # not done: {<Future at 0x2c12a20 state=running>,
    #            <Future at 0x2c1d2e8 state=running>}
  wait_and_see_result(fs)
    # done: {<Future at 0x21de080 state=finished returned NoneType>,
    #        <Future at 0x2c12a20 state=finished returned NoneType>}
    # not done: {<Future at 0x2c1d2e8 state=running>}
  wait_and_see_result(fs)
    # done:  {<Future at 0x21de080 state=finished returned NoneType>,
    #         <Future at 0x2c12a20 state=finished returned NoneType>,
    #         <Future at 0x2c1d2e8 state=finished returned NoneType>}
    # not done: set()
```

```
concurrent.futures.as_completed(fs, timeout=None)
```

返回一个iterator, 每当一个Future Object执行完成, 就yield

```
import concurrent.futures
import time
from concurrent.futures import ThreadPoolExecutor

def wait_n_seconds(n):
    time.sleep(n)
    return n

with ThreadPoolExecutor(3) as e:
    f1 = e.submit(wait_n_seconds, 1)
    f2 = e.submit(wait_n_seconds, 2)
    f3 = e.submit(wait_n_seconds, 3)
    fs = [f1, f2, f3]
    for f in concurrent.futures.as_completed(fs):
        print("done: %s, result: %d" % (f.done(), f.result()))
    # I've slept for 1 seconds
    # I've slept for 2 seconds
    # I've slept for 3 seconds
```

IMPLEMENTATION

- Future
- ~~ThreadPoolExecutor~~
- ProcessPoolExecutor

时间所限, `ThreadPoolExecutor`比较简单就不讲了

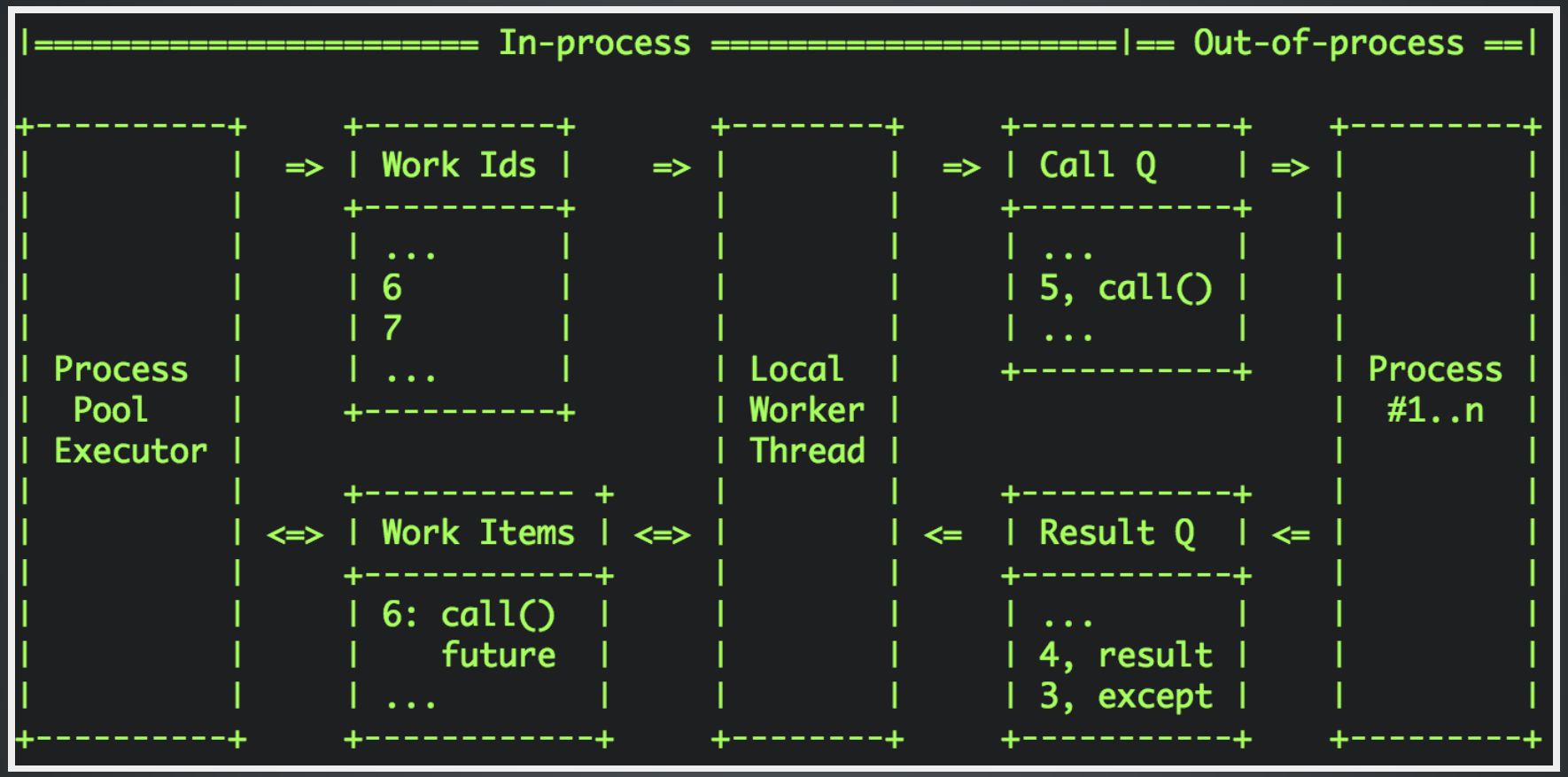
FUTURE.RESULT() 原理

KEY: CONDITION

```
class Future(object):
    self._condition = threading.Condition()  # Condition = Lock + Event
    self._state = PENDING  # Future Object 的状态
    self._result = None  # 执行结果

    def set_result(self, result):  # 执行完毕, 调用 set_result
        with self._condition:
            self._result = result
            self._state = FINISHED
            # ...
            self._condition.notify_all()

    def result(self, timeout=None):
        with self._condition:
            # ...
            self._condition.wait(timeout)
            if self._state in [CANCELLED, CANCELLED_AND_NOTIFIED]:
                raise CancelledError()
            elif self._state == FINISHED:
                return self.__get_result()  # 返回执行结果
            else:
                raise TimeoutError()  # 没有FINISHED,但又没有cancel,超时咯
```

PROCESSPOOLEXECUTOR

ProcessPoolExecutor 数据流



Future 只存在于左半边, worker Thread/Process只负责执行

call_queue->Processing->result_queue
这些代码发生在worker进程中

```
def _process_worker(call_queue, result_queue):
    while True:
        call_item = call_queue.get(block=True)  # block here

        if call_item is None:  # None is call Q's sentinel
            result_queue.put(os.getpid())
            return

        try:
            r = call_item.fn(*call_item.args, **call_item.kwargs)
        except BaseException as e:
            result_queue.put(_ResultItem(call_item.work_id, exception=e))
        else:
            result_queue.put(_ResultItem(call_item.work_id, result=r))
```

WHAT CONCURRENT.FUTURES CAN'T DO

只能返回一次

```
from subprocess import Popen, PIPE
proc = Popen(['a-program'], stdout=PIPE)
print(proc.stdout.read())
# other operations...
print(proc.stdout.read())
```

有些时候可能想持续获取输出

NO WAY

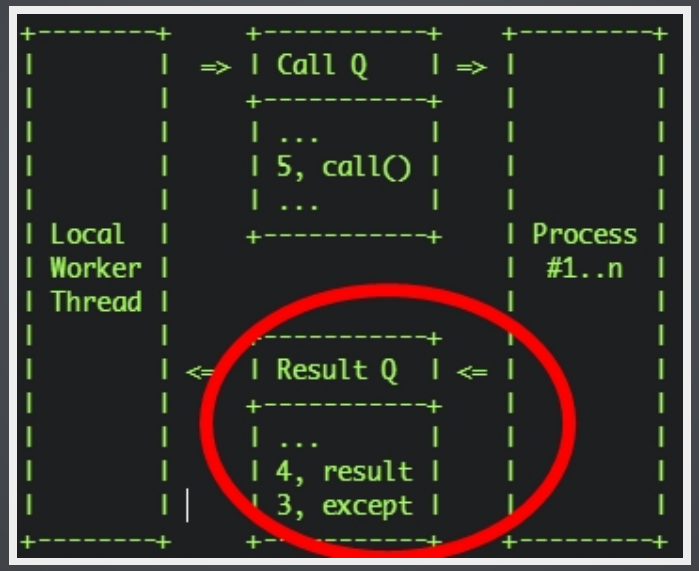
```
class _CallItem(object):
    def __init__(self, work_id, fn, args, kwargs):
        self.work_id = work_id
        self.fn = fn
        self.args = args
        self.kwargs = kwargs

# execute function
r = call_item.fn(*call_item.args, **call_item.kwargs)
```

WHAT ABOUT THIS?

```
def make_call():
    process = subprocess.Popen(['a-program'], stdout=PIPE)
    return process

executor = ProcessPoolExecutor()
future = executor.submit(make_call)
returned_process = future.result()

print(returned_process.stdout.read())

# other operations...

print(returned_process.stdout.read())
```

IT'S A QUEUE

(ノ ゜ □゜)ノ ⌒┻━┻



Queue.put()只支持pickable

```
>>> from multiprocessing import SimpleQueue
>>> q = SimpleQueue()
>>> q.put((1 for i in range(10)))
```

_pickle.PicklingError: Can't pickle : attribute lookup generator on builtins failed

OUTPUT, OUTPUT ONCE

```
def make_call():
    process = Popen(['a-program'], stdout=PIPE)
    return process.communicate()[0]  # stdout

executor = ThreadPoolExecutor()
future = executor.submit(make_call)
print(future.result())
```

WHY USE CONCURRENT.FUTURES?

PEP3148

parallelizing simple operations requires a lot of work i.e. explicitly launching processes/threads, constructing a work/results queue, and waiting for completion or some other termination condition

简化多进程/线程操作, 易于使用

WHY USE CONCURRENT.FUTURES?

It is also difficult to design an application with a global process/thread limit when each component invents its own parallel execution strategy.

统一多进程/多线程控制接口，便于代码维护

```
e = ThreadPoolExecutor(4)
f = e.submit(task)  # 如果不使用cf, 进程<->线程 互换需要动其它代码
# control: wait, as_completed, ...
```

WHY USE CONCURRENT.FUTURES?

> *We hope to either add, or move existing, concurrency-related libraries to this in the future. A prime example is the multiprocessing.Pool work, as well as other "addons" included in that module, which work across thread and process boundaries.*

Brian Quinla said:

> *The plan was to move concurrency-related librraries to the concurrent package.*

未来可能把`threading, multiprocessing`移入`concurrent`

实际上目前`concurrent.futures`已经能够替代进程池

WHY USE CONCURRENT.FUTURES?

Future 是一个通用概念

concurrent.futures最早在Python中引入它

```
@asyncio.coroutine
def ex_multi_async(to_fetch):
    futures, results = [], [] for url in to_fetch:
        futures.append(extract_async(url))

    for future in asyncio.as_completed(futures):
        results.append((yield from future)) return results
```

code snippet Python3.4 asyncio

SUMMARY

官方实现的线程池、进程池
简单易用的接口，未来可能更通用
提高代码的可维护性
异步模型在 Python 中最初的实践，引入`Future`

`concurrent.futures` is **NOT** a medicine of GIL!!

QUESTION