concurrent.futures — Manage Pools of Concurrent Tasks

Purpose: Easily manage tasks running concurrently and in parallel.

The `concurrent.futures` modules provides interfaces for running tasks using pools of thread or process workers. The APIs are the same, so applications can switch between threads and processes with minimal changes.

The module provides two types of classes for interacting with the pools. *Executors* are used for managing pools of workers, and *futures* are used for managing results computed by the workers. To use a pool of workers, an application creates an instance of the appropriate executor class and then submits tasks for it to run. When each task is started, a `Future` instance is returned. When the result of the task is needed, an application can use the `Future` to block until the result is available. Various APIs are provided to make it convenient to wait for tasks to complete, so that the `Future` objects do not need to be managed directly.

Using map() with a Basic Thread Pool

The `ThreadPoolExecutor` manages a set of worker threads, passing tasks to them as they become available for more work. This example uses `map()` to concurrently produce a set of results from an input iterable. The task uses `time.sleep()` to pause a different amount of time to demonstrate that, regardless of the order of execution of concurrent tasks, `map()` always returns the values in order based on the inputs.

```python
# futures_thread_pool_map.py

from concurrent import futures
import threading
import time


def task(n):
    print('{}: sleeping {}'.format(
        threading.current_thread().name,
        n)
    )
    time.sleep(n / 10)
    print('{}: done with {}'.format(
        threading.current_thread().name,
        n)
    )
    return n / 10


ex = futures.ThreadPoolExecutor(max_workers=2)
print('main: starting')
results = ex.map(task, range(5, 0, -1))
print('main: unprocessed results {}'.format(results))
print('main: waiting for real results')
real_results = list(results)
print('main: results: {}'.format(real_results))
```

The return value from `map()` is actually a special type of iterator that knows to wait for each response as the main program iterates over it.

```
$ python3 futures_thread_pool_map.py

main: starting
ThreadPoolExecutor-0_0: sleeping 5
ThreadPoolExecutor-0_1: sleeping 4
main: unprocessed results <generator object
Executor.map.<locals>.result_iterator at 0x103e12780>
main: waiting for real results
ThreadPoolExecutor-0_1: done with 4
```

```
ThreadPoolExecutor-0_1: sleeping 3
ThreadPoolExecutor-0_0: done with 5
ThreadPoolExecutor-0_0: sleeping 2
ThreadPoolExecutor-0_0: done with 2
ThreadPoolExecutor-0_0: sleeping 1
ThreadPoolExecutor-0_1: done with 3
ThreadPoolExecutor-0_0: done with 1
main: results: [0.5, 0.4, 0.3, 0.2, 0.1]
```

Scheduling Individual Tasks

In addition to using `map()`, it is possible to schedule an individual task with an executor using `submit()`, and use the `Future` instance returned to wait for that task's results.

```python
# futures_thread_pool_submit.py

from concurrent import futures
import threading
import time


def task(n):
    print('{}: sleeping {}'.format(
        threading.current_thread().name,
        n)
    )
    time.sleep(n / 10)
    print('{}: done with {}'.format(
        threading.current_thread().name,
        n)
    )
    return n / 10


ex = futures.ThreadPoolExecutor(max_workers=2)
print('main: starting')
f = ex.submit(task, 5)
print('main: future: {}'.format(f))
print('main: waiting for results')
result = f.result()
print('main: result: {}'.format(result))
print('main: future after result: {}'.format(f))
```

The status of the future changes after the tasks is completed and the result is made available.

```
$ python3 futures_thread_pool_submit.py

main: starting
ThreadPoolExecutor-0_0: sleeping 5
main: future: <Future at 0x1034e1ef0 state=running>
main: waiting for results
ThreadPoolExecutor-0_0: done with 5
main: result: 0.5
main: future after result: <Future at 0x1034e1ef0 state=finished
 returned float>
```

Waiting for Tasks in Any Order

Invoking the `result()` method of a `Future` blocks until the task completes (either by returning a value or raising an exception), or is canceled. The results of multiple tasks can be accessed in the order the tasks were scheduled using `map()`. If it does not matter what order the results should be processed, use `as_completed()` to process them as each task finishes.

```python
# futures_as_completed.py

from concurrent import futures
import random
import time
```

```
def task(n):
    time.sleep(random.random())
    return (n, n / 10)


ex = futures.ThreadPoolExecutor(max_workers=5)
print('main: starting')

wait_for = [
    ex.submit(task, i)
    for i in range(5, 0, -1)
]

for f in futures.as_completed(wait_for):
    print('main: result: {}'.format(f.result()))
```

Because the pool has as many workers as tasks, all of the tasks can be started. They finish in a random order so the values generated by `as_completed()` are different each time the example runs.

```
$ python3 futures_as_completed.py

main: starting
main: result: (1, 0.1)
main: result: (5, 0.5)
main: result: (3, 0.3)
main: result: (2, 0.2)
main: result: (4, 0.4)
```

Future Callbacks

To take some action when a task completed, without explicitly waiting for the result, use `add_done_callback()` to specify a new function to call when the `Future` is done. The callback should be a callable taking a single argument, the `Future` instance.

```
# futures_future_callback.py

from concurrent import futures
import time


def task(n):
    print('{}: sleeping'.format(n))
    time.sleep(0.5)
    print('{}: done'.format(n))
    return n / 10


def done(fn):
    if fn.cancelled():
        print('{}: canceled'.format(fn.arg))
    elif fn.done():
        error = fn.exception()
        if error:
            print('{}: error returned: {}'.format(
                fn.arg, error))
        else:
            result = fn.result()
            print('{}: value returned: {}'.format(
                fn.arg, result))


if __name__ == '__main__':
    ex = futures.ThreadPoolExecutor(max_workers=2)
    print('main: starting')
    f = ex.submit(task, 5)
    f.arg = 5
    f.add_done_callback(done)
    result = f.result()
```

The callback is invoked regardless of the reason the Future is considered “done,” so it is necessary to check the status of the

The callback is invoked regardless of the reason the Future is considered "done," so it is necessary to check the status of the object passed in to the callback before using it in any way.

```
$ python3 futures_future_callback.py

main: starting
5: sleeping
5: done
5: value returned: 0.5
```

Canceling Tasks

A Future can be canceled, if it has been submitted but not started, by calling its `cancel()` method.

```python
# futures_future_callback_cancel.py

from concurrent import futures
import time


def task(n):
    print('{}: sleeping'.format(n))
    time.sleep(0.5)
    print('{}: done'.format(n))
    return n / 10


def done(fn):
    if fn.cancelled():
        print('{}: canceled'.format(fn.arg))
    elif fn.done():
        print('{}: not canceled'.format(fn.arg))


if __name__ == '__main__':
    ex = futures.ThreadPoolExecutor(max_workers=2)
    print('main: starting')
    tasks = []

    for i in range(10, 0, -1):
        print('main: submitting {}'.format(i))
        f = ex.submit(task, i)
        f.arg = i
        f.add_done_callback(done)
        tasks.append((i, f))

    for i, t in reversed(tasks):
        if not t.cancel():
            print('main: did not cancel {}'.format(i))

    ex.shutdown()
```

`cancel()` returns a Boolean indicating whether or not the task was able to be canceled.

```
$ python3 futures_future_callback_cancel.py

main: starting
main: submitting 10
10: sleeping
main: submitting 9
9: sleeping
main: submitting 8
main: submitting 7
main: submitting 6
main: submitting 5
main: submitting 4
main: submitting 3
main: submitting 2
main: submitting 1
1: canceled
```

```
2: canceled
3: canceled
4: canceled
5: canceled
6: canceled
7: canceled
8: canceled
main: did not cancel 9
main: did not cancel 10
10: done
10: not canceled
9: done
9: not canceled
```

Exceptions in Tasks

If a task raises an unhandled exception, it is saved to the `Future` for the task and made available through the `result()` or `exception()` methods.

```python
# futures_future_exception.py

from concurrent import futures


def task(n):
    print('{}: starting'.format(n))
    raise ValueError('the value {} is no good'.format(n))


ex = futures.ThreadPoolExecutor(max_workers=2)
print('main: starting')
f = ex.submit(task, 5)

error = f.exception()
print('main: error: {}'.format(error))

try:
    result = f.result()
except ValueError as e:
    print('main: saw error "{}" when accessing result'.format(e))
```

If `result()` is called after an unhandled exception is raised within a task function, the same exception is re-raised in the current context.

```
$ python3 futures_future_exception.py

main: starting
5: starting
main: error: the value 5 is no good
main: saw error "the value 5 is no good" when accessing result
```

Context Manager

Executors work as context managers, running tasks concurrently and waiting for them all to complete. When the context manager exits, the `shutdown()` method of the executor is called.

```python
# futures_context_manager.py

from concurrent import futures


def task(n):
    print(n)


with futures.ThreadPoolExecutor(max_workers=2) as ex:
    print('main: starting')
    ex.submit(task, 1)
    ex.submit(task, 2)
    ex.submit(task, 3)
    ex.submit(task, 4)

print('main: done')
```

This mode of using the executor is useful when the thread or process resources should be cleaned up when execution leaves the current scope.

```
$ python3 futures_context_manager.py

main: starting
1
2
3
4
main: done
```

Process Pools

The `ProcessPoolExecutor` works in the same way as `ThreadPoolExecutor`, but uses processes instead of threads. This allows CPU-intensive operations to use a separate CPU and not be blocked by the CPython interpreter’s global interpreter lock.

```
# futures_process_pool_map.py
```

```
from concurrent import futures
import os


def task(n):
    return (n, os.getpid())


ex = futures.ProcessPoolExecutor(max_workers=2)
results = ex.map(task, range(5, 0, -1))
for n, pid in results:
    print('ran task {} in process {}'.format(n, pid))
```

As with the thread pool, individual worker processes are reused for multiple tasks.

```
$ python3 futures_process_pool_map.py

ran task 5 in process 40854
ran task 4 in process 40854
ran task 3 in process 40854
ran task 2 in process 40854
ran task 1 in process 40854
```

If something happens to one of the worker processes to cause it to exit unexpectedly, the `ProcessPoolExecutor` is considered “broken” and will no longer schedule tasks.

```
# futures_process_pool_broken.py
```

```
from concurrent import futures
import os
import signal


with futures.ProcessPoolExecutor(max_workers=2) as ex:
    print('getting the pid for one worker')
    f1 = ex.submit(os.getpid)
    pid1 = f1.result()

    print('killing process {}'.format(pid1))
    os.kill(pid1, signal.SIGHUP)

    print('submitting another task')
    f2 = ex.submit(os.getpid)
    try:
        pid2 = f2.result()
    except futures.process.BrokenProcessPool as e:
        print('could not start new tasks: {}'.format(e))
```

The BrokenProcessPool exception is actually thrown when the results are processed, rather than when the new task is submitted.

```
$ python3 futures_process_pool_broken.py

getting the pid for one worker
killing process 40858
submitting another task
could not start new tasks: A process in the process pool was
terminated abruptly while the future was running or pending.
```

See also

- Standard library documentation for concurrent.futures
- **PEP 3148** – The proposal for creating the `concurrent.futures` feature set.
- Combining Coroutines with Threads and Processes
- `threading`
- `multiprocessing`

Debugging with asyncio

Networking

This page was last updated 2018-03-18.

Navigation

Debugging with asyncio
Networking