Programmazione Avanzata

Concorrenza: II parte

Informazioni sul modulo concurrent.futures

- Il modulo concurrent.futures fornisce un'interfaccia per eseguire callable in modo asincrono.
- L'esecuzione asincrona può essere svolta con thread, usando ThreadPoolExecutor, o con processi separati, usando ProcessPoolExecutor. Entrambe le classi estendono la classe astratta Executor illustrata di seguito.

Informazioni sul modulo concurrent.futures

- concurrent.futures.Future è un oggetto che incapsula l'esecuzione asincrona di un callable
- oggetti Future sono creati invocando il metodo concurrent.futures.Executor.submit()
- la classe concurrent.futures.Executor non può essere usata direttamente perché è una classe astratta. Al suo posto devono essere usate le sue due seguenti sottoclassi concrete.
  - concurrent.futures.ProcessPoolExecutor realizza la concorrenza usando processi multipli.
  - concurrent.futures.ThreadPoolExecutor realizza la concorrenza con thread multipli.

Informazioni sul modulo concurrent.futures

- concurrent.futures.Executor.submit(*fn*, **args*, ***kwargs)* fa in modo che il callable fn venga eseguito come fn(*args **kwargs) e restituisce un oggetto Future che rappresenta l'esecuzione del callable.

Informazioni sul modulo concurrent.futures

- concurrent.futures.as_completed(*fs*, *timeout=None*): restituisce un iteratore su istanze di Future fornite da fs.
- Le istanze di Future vengono fornite non appena vengono completate.
  - Le istanze di future potrebbero anche essere state create da differenti istanze di executor.
- Se un future fornito da fs e` duplicato, questo viene restituito un'unica volta dall'iteratore.
- I future completati prima che as_completed() venga invocato, vengono restituiti per primi.
- L'iteratore restituito lancia concurrent.future.TimeoutError se dopo aver invocato __next__(), il risultato non è disponibile entro timeout secondi dall'invocazione di as_completed(). Timeout può essere un int o un float. Se Timeout non è specificato nella chiamata, non c'è limite di attesa.

Concorrenza ad alto livello: un esempio di mutiprocessing con uso di Futures

- Questa funzione esegue lo stesso lavoro della funzione scale() dell'implementazione precedente ma lo fa in modo completamente diverso
- La funzione comincia creando un insieme vuoto di future.
- Poi crea un oggetto ProcessPoolExecutor che dietro le scene creerà un numero di processi worker.
  - Il numero esatto per max_workers è determinato da un'euristica ma qui il numero è fissato.

```
def scale(size, smooth, source, target, concurrency):
    futures = set()
    with concurrent.futures.ProcessPoolExecutor(
            max_workers=concurrency) as executor:
        for sourceImage, targetImage in get_jobs(source, target):
            future = executor.submit(scale_one, size, smooth, sourceImage,
                    targetImage)
            futures.add(future)
        summary = wait_for(futures)
        if summary.canceled:
            executor.shutdown()
        return summary
```

Concorrenza ad alto livello: un esempio di mutiprocessing con uso di Futures

- Una volta che ha creato un oggetto ProcessPoolExecutor, scale() itera sui job restituiti da get_jobs() e crea per ciascuno di essi un future.
- Il metodo concurrent.futures.ProcessPoolExecutor.submit() accetta una funzione worker e argomenti opzionali e restituisce un oggetto Future.
- Il pool comincia a lavorare non appena ha un future su cui lavorare. Quando tutti i future sono stati creati, viene chiamata una funzione wait_for() passandole l'insieme di future. Questa funzione si bloccherà fino a quando tutti i future sono stati eseguiti o cancellati dall'utente. Se l'utente cancella, la funzione dismette il pool executor.

```
def scale(size, smooth, source, target, concurrency):
    futures = set()
    with concurrent.futures.ProcessPoolExecutor(
            max_workers=concurrency) as executor:
        for sourceImage, targetImage in get_jobs(source, target):
            future = executor.submit(scale_one, size, smooth, sourceImage,
                    targetImage)
            futures.add(future)
        summary = wait_for(futures)
        if summary.canceled:
            executor.shutdown()
        return summary
```

46

Concorrenza ad alto livello: un esempio di mutiprocessing con uso di Futures

- Questa funzione svolge lo stesso compito della funzione add_jobs() dell'implementazione precedente, solo che invece di aggiungere job alla coda è una funzione generatore che restituisce job su richiesta.

```
def get_jobs(source, target):
    for name in os.listdir(source):
        yield os.path.join(source, name), os.path.join(target, name)
```


47

Concorrenza ad alto livello: un esempio di mutiprocessing con uso di Futures

- La funzione wait_for() (la figura mostra il primo segmento) viene invocata per aspettare che i future vengano completati.
- Nel for viene invocato concurrent.futures.as_completed() che si blocca fino a che non viene completato o cancellato un future e poi restituisce quel future
- Se il callable worker eseguito dal future lancia un'eccezione allora il metodo future.exception() la restituisce; altrimenti restituisce None. Se non si verifica alcuna eccezione allora viene recuperato il risultato del future e riportato il progresso all'utente.

```
def wait_for(futures):
    canceled = False
    copied = scaled = 0
    try:
        for future in concurrent.futures.as_completed(futures):
            err = future.exception()
            if err is None:
```


48

Concorrenza ad alto livello: un esempio di mutiprocessing con uso di Futures

- Se si verifica un'eccezione prevedibile (cioè proveniente dal modulo image), essa viene riportata all'utente. Ma se si verifica un'eccezione inattesa allora essa viene lanciata perché potrebbe trattarsi di un errore logico del programma. Nel caso si verifichi una cancellazione effettuata dall'utente con Ctrl+C, la funzione cancella i future una alla volta.

secondo segmento di wait_for()

```
                result = future.result()
                copied += result.copied
                scaled += result.scaled
                Qtrac.report("{} {}".format("copied" if result.copied else
                        "scaled", os.path.basename(result.name)))
            elif isinstance(err, Image.Error):
                Qtrac.report(str(err), True)
            else:
                raise err # Unanticipated
    except KeyboardInterrupt:
        Qtrac.report("canceling...")
        canceled = True
        for future in futures:
            future.cancel()
    return Summary(len(futures), copied, scaled, canceled)
```

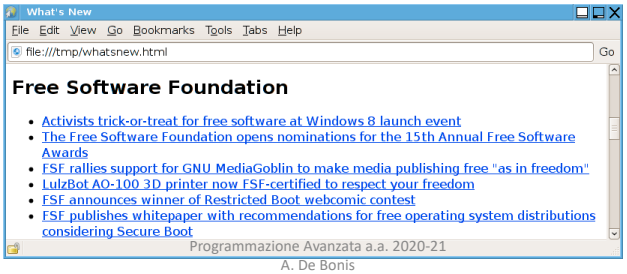
49

Programmazione Avanzata

Concorrenza I/O Bound

Un esempio

- Scaricare file o pagine web da Internet e` un esigenza molto frequente. A causa dei tempi di latenza della rete, e` di solito possibile fare molti download in modo concorrente e quindi terminare molto piu` velocemente il download.
- Il libro di Summerfield propone un codice che scarica RSS feed (piccoli documenti XML) che riportano storie relative a notizie riguardanti il mondo della tecnologia.
- I feed provengono da diversi siti web e il programma li usa per produrre una singola pagina HTML con i link a tutte le storie.



Un esempio

- La tabella mostra i tempi di varie versioni del programma

Program	Concurrency	Seconds	Speedup
`whatsnew.py`	*None*	172	*Baseline*
`whatsnew-c.py`	16 coroutines	180	0.96×
`whatsnew-q-m.py`	16 processes using a queue	45	3.82×
`whatsnew-m.py`	16 processes using a process pool	50	3.44×
`whatsnew-q.py`	16 threads using a queue	50	3.44×
`whatsnew-t.py`	16 threads using a thread pool	48	3.58×

- Poiche' la latenza della rete varia molto, la velocita` dei programmi puo` variare molto da un minimo di 2 fino ad un massimo di 10 o piu` volte, in base ai siti, la quantita` di dati scaricati e la banda della connessione.
- In considerazione di cio`, le differenze tra la versione basata su multiprocessing e quella basata su multithreading sono insignificanti.
- La cosa importante da ricordare e` che l'approccio concorrente permette di raggiungere velocita` molto piu` elevate nonostante queste varino di esecuzione in esecuzione

52

Informazioni sul pacchetto threading

- Il modulo threading costruisce interfacce a piu` alto livello per il threading al di sopra del modulo di basso livello _thread.
- La classe Thread rappresenta un'attivita` che viene eseguita in un thread separato.
- Una volta creato un oggetto thread, si da` inizio alla sua attivita` invocando il metodo start() che invoca il metodo run() del thread in un thread separato.
- Una volta iniziata l'attivita` del thread, il thread viene considerato vivo fino al momento in cui non termina il suo metodo run() (anche se a causa di un'eccezione non gestita)

53

Informazioni sul pacchetto threading

- Altri thread possono invocare il metodo join() di un thread. Cio` blocca il thread che invoca join() fino a quando non termina il thread il cui metodo join() e` stato invocato.
- Il thread ha un attributo name il cui valore puo` essere passato al costruttore e letto o modificato attraverso l'attributo name.
- I thread possono essere contrassegnati come daemon attraverso un flag. Se vi sono solo thread daemon in esecuzione, si esce dall'intero programma. Il valore iniziale del flag e` ereditato dal thread che crea il thread o puo` essere passato al costruttore.
- L'interfaccia di threading.Thread fornita e` simile a quella di multiprocessing.Process.

Un'implementazione con code e threading

- Questo esempio usa thread multipli e due code thread-safe, una per i job (URL) e l'altra per i risultati (coppie contenenti True e un frammento HTML da includere nella pagina HTML da costruire, oppure False e un messaggio di errore)
- La funzione main() comincia ricevendo dalla linea comando il massimo numero di elementi (limit) da leggere da una data URL e un livello di concorrenza (concurrency).
  - La funzione handle_commandline() pone il valore della concorrenza pari a 4 volte il numero di core (si sceglie un multiplo del numero di core, dal momento che il programma e` I/O bound.)

```
def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting...")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    jobs = queue.Queue()
    results = queue.Queue()
    create_threads(limit, jobs, results, concurrency)
    todo = add_jobs(filename, jobs)
    process(todo, jobs, results, concurrency)
```

Un'implementazione con code e threading

- Il modulo queue implementa code che possono essere utilizzate da piu` entita`.
- Sono particolarmente utili nel multithreading in quanto consentono a thread multipli di scambiarsi informazioni in modo sicuro.
- Il modulo implementa tre tipi di code: FIFO, LIFO e Coda a priorita`.
- Internamente queste code usano lock per bloccare temporaneamente thread in competizione tra loro.

Un'implementazione con code e threading

- La funzione poi riporta all'utente che sta cominciando a lavorare e mette in filename l'intero percorso del file di dati contenente le URL.
- Poi la funzione crea due code thread-safe e i thread worker.
- Una volta iniziati i thread worker (che sono bloccati perche' non c'e` alcun lavoro da svolgere ancora) vengono aggiunti i job alla coda dei job.
- Si attende quindi nella funzione process() che i job vengano completati e poi vengono forniti in output i risultati.

```
def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting...")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    jobs = queue.Queue()
    results = queue.Queue()
    create_threads(limit, jobs, results, concurrency)
    todo = add_jobs(filename, jobs)
    process(todo, jobs, results, concurrency)
```

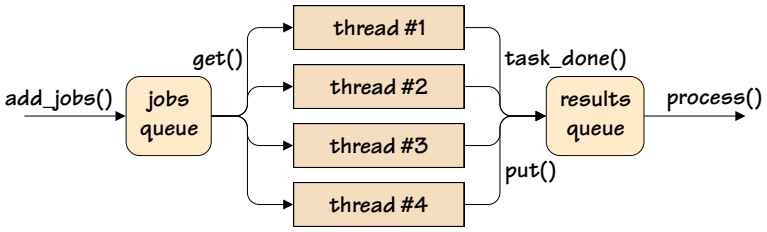
Un'implementazione con code e threading

- Questa funzione crea un numero di thread worker pari al valore specificato da concurrency e da` a ciascuno di questi thread una funzione worker da eseguire e gli argomenti con cui la funzione deve essere invocata.
- Ciascun thread viene trasformato in thread daemon in modo che venga terminato al termine del programma.
- Alla fine viene invocato start sul thread che si blocchera` in attesa di un job. In questa attesa sono solo i thread worker ad essere bloccati non il thread principale.

```
def create_threads(limit, jobs, results, concurrency):
    for _ in range(concurrency):
        thread = threading.Thread(target=worker, args=(limit, jobs,
                results))
        thread.daemon = True
        thread.start()
```


Un'implementazione con code e threading

- Questa e` la struttura del programma concorrente.



Un'implementazione con code e threading

- La funzione Feed.iter() restituisce ciascun feed come una coppia (*title*, *url*) che viene aggiunta alla coda jobs. Alla fine viene restituito il numero di job.
- In questo caso la funzione avrebbe potuto restituire lo stesso valore invocando *jobs*.qsize() piuttosto che computare direttamente il numero di job. Se pero` add_jobs() fosse stato eseguito nel suo proprio thread allora il valore restituito da qsize() non sarebbe stato attendibile dal momento che i job sarebbero stati prelevati nello stesso momento in cui venivano aggiunti.

```
def add_jobs(filename, jobs):
    for todo, feed in enumerate(Feed.iter(filename), start=1):
        jobs.put(feed)
    return todo
```


60

Un'implementazione con code e threading

- La funzione worker esegue un loop infinito. Il loop infinito termina sicuramente al termine del programma dal momento che il thread e` un daemon.
- La funzione si blocca in attesa di prendere un job dalla coda dei job e non appena prende un job usa la funzione Feed.read() (del modulo Feed.py) per leggere il file identificato dalla URL.
- Se la read fallisce, il flag ok e` False e viene stampato il risultato che e` un messaggio di errore. Altrimenti, sempre che il programma ottenga un risultato (una lista di stringhe HTML), viene stampato il primo elemento (privato dei tag HTML) e aggiunto il risultato alla coda dei risultati.
- Il blocco try ... finally garantisce che jobs.task_done() venga invocato ogni volta che che viene invocato queue.Queue.get() call.

La funzione Feed.read() legge una data URL (feed) e tenta di farne il parsing. Se il parsing ha successo, la funzione restituisce True insieme ad una lista di frammenti HTML. Altrimenti, restituisce False insieme a None o a un messaggio di errore.

```
def worker(limit, jobs, results):
    while True:
        try:
            feed = jobs.get()
            ok, result = Feed.read(feed, limit)
            if not ok:
                Qtrac.report(result, True)
            elif result is not None:
                Qtrac.report("read {}".format(result[0][4:-6]))
                results.put(result)
        finally:
            jobs.task_done()
```


61

Un'implementazione con code e threading

- Questa funzione viene invocata una volta che i thread sono stati creati e i job aggiunti alla coda. Essa invoca queue.Queue.join() che si blocca fino a quando la coda non si svuota, cioe` fino a che non vengono eseguiti tutti i job o l'utente non cancella l'esecuzione.
- Se l'utente non cancella l'esecuzione, viene invocata la funzione output() per scrivere nel file HTML le liste di link e poi viene stampato un report con la funzione Qtrac.report().
- Alla fine la funzione open() del modulo webbrowser viene invocata sul file HTML per aprirlo nel browser di default.

La funzione output() crea un file whatsnew.html e lo popola con i titoli dei feed e con i loro link. Queste informazioni sono presenti nei result all'interno della coda results. Ogni result contiene una lista di frammenti HTML (un titolo seguito da uno o piu` link).
Al termine output() restituisce il numero di result (numero di jobs terminati con successo) e il nome del file HTML creato.

```
def process(todo, jobs, results, concurrency):
    canceled = False
    try:
        jobs.join() # Wait for all the work to be done
    except KeyboardInterrupt:
        Qtrac.report("canceling...")
        canceled = True
    if canceled:
        done = results.qsize()
    else:
        done, filename = output(results)
    Qtrac.report("read {}/{} feeds using {} threads{}".format(done, todo,
            concurrency, " [canceled]" if canceled else ""))
    print()
    if not canceled:
        webbrowser.open(filename)
```


Un'implementazione che usa Futures e threading

- La funzione main crea un insieme di future inizialmente vuoto e poi crea un esecutore di un pool di thread che lavora allo stesso modo di un esecutore di un pool di processi.
- Per ogni feed, viene creato un nuovo future invocando il metodo concurrent.futures.ThreadPoolExecutor.submit() che eseguira` la funzione Feed.read() sulla URL del feed e restituira` al piu` un numero di link pari a limit.

```
def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting...")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    futures = set()
    with concurrent.futures.ThreadPoolExecutor(
            max_workers=concurrency) as executor:
        for feed in Feed.iter(filename):
            future = executor.submit(Feed.read, feed, limit)
            futures.add(future)
        done, filename, canceled = process(futures)
        if canceled:
            executor.shutdown()
    Qtrac.report("read {}/{} feeds using {} threads{}".format(done,
            len(futures), concurrency, " [canceled]" if canceled else ""))
    print()
    if not canceled:
        webbrowser.open(filename)
```

Un'implementazione che usa Futures e threading

- Ciascun future creato viene aggiunto al pool futures con add().
- Una volta che i future sono stati creati, viene invocata la funzione process() che aspettera` fino a quando non vengono terminati tutti i future o fino a quando l'utente non cancella l'esecuzione.
- Alla fine viene stampato un sunto e se l'utente non ha cancellato l'esecuzione, la pagina HTML generata viene aperta nel browser dell'utente.

```
def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting...")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    futures = set()
    with concurrent.futures.ThreadPoolExecutor(
            max_workers=concurrency) as executor:
        for feed in Feed.iter(filename):
            future = executor.submit(Feed.read, feed, limit)
            futures.add(future)
        done, filename, canceled = process(futures)
        if canceled:
            executor.shutdown()
    Qtrac.report("read {}/{} feeds using {} threads{}".format(done,
            len(futures), concurrency, " [canceled]" if canceled else ""))
    print()
    if not canceled:
        webbrowser.open(filename)
```


Un'implementazione che usa Futures e threading

- Questa funzione scrive l'inizio del file HTML e poi invoca la funzione wait_for() per aspettare che il lavoro venga fatto.
- Se l'utente non cancella l'esecuzione, la funzione itera sui risultati (le coppie gia` descritte) e per quelli che contengono una lista (che consiste di titoli, ciascuno seguito da uno o piu` link) gli elementi della lista vengono scritti nel file HTML.
- Se l'utente cancella l'esecuzione, la funzione calcola semplicemente quanti feed sono stati letti con successo.
- In ogni caso, la funzione restituisce il numero di feed letti, il nome del file e True o False a seconda che l'utente abbia cancellato o meno l'esecuzione.

```
def process(futures):
    canceled = False
    done = 0
    filename = os.path.join(tempfile.gettempdir(), "whatsnew.html")
    with open(filename, "wt", encoding="utf-8") as file:
        file.write("<!doctype html>\n")
        file.write("<html><head><title>What's New</title></head>\n")
        file.write("<body><h1>What's New</h1>\n")
        canceled, results = wait_for(futures)
        if not canceled:
            for result in (result for ok, result in results if ok and
                    result is not None):
                done += 1
                for item in result:
                    file.write(item)
        else:
            done = sum(1 for ok, result in results if ok and result is not
                       None)
        file.write("</body></html>\n")
    return done, filename, canceled
```

Un'implementazione che usa Futures e threading

- Questa funzione itera sui future, bloccandosi fino a quando uno di essi non termina o e` cancellato.
- Una volta ricevuto un future la funzione riporta un errore o un successo e in entrambi i casi appende il Booleano e il risultato (una lista di stringhe o una stringa errore) ad una lista di risultati.

```
def wait_for(futures):
    canceled = False
    results = []
    try:
        for future in concurrent.futures.as_completed(futures):
            err = future.exception()
            if err is None:
                ok, result = future.result()
                if not ok:
                    Qtrac.report(result, True)
                elif result is not None:
                    Qtrac.report("read {}".format(result[0][4:-6]))
                results.append((ok, result))
            else:
                raise err # Unanticipated
    except KeyboardInterrupt:
        Qtrac.report("canceling...")
        canceled = True
        for future in futures:
            future.cancel()
    return canceled, results
```


Oggetti condivisi

È possibile creare oggetti condivisi usando memoria condivisa che puo` essere ereditata dai processi figlio.

multiprocessing.Value(*typecode_or_type*, **args*, *lock=True*)

- Restituisce un oggetto ctypes allocato dalla memoria condivisa.
- Si puo` accedere all'oggetto mediante l'attributo value di un Value.
- *typecode_or_type* determina il tipo dell'oggetto restituito: puo` essere un tipo ctypes o un carattere (typecode) del genere usato dal modulo array (https://docs.python.org/3/library/array.html)
  - ctypes è una libreria di funzioni che fornisce tipi di dati compatibili con C
    - esempi: c_int, c_long.
- **args* è passato al costruttore per il tipo.
- Se lock e` True (valore di default) allora viene creato un nuovo lock ricorsivo (di tipo RLock) per sincronizzare l'accesso al valore. Se lock è False l'oggetto restituito non sara` automaticamente protetto da un lock e quindi non sara` necesssariamente process-safe.
  - i lock di tipo RLock possono essere acquisiti piu` volte da uno stesso thread e possono essere usati negli statement with

Oggetti condivisi

N.B.

- Operazioni quali += che coinvolgono un'operazione di lettura ed una di scrittura non sono atomiche. Se quindi vogliamo per esempio incrementare un valore condiviso non è sufficiente scrivere counter.value += 1
- Assumendo che il lock associato sia ricorsivo (lo è per default), esso supporta il protocollo del context manager e possiamo scrivere

```
with counter.get_lock():
        counter.value += 1
```


Oggetti condivisi

- multiprocessing.Array(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)
  - Restituisce un array di ctypes allocato dalla memoria condivisa
  - *typecode_or_type* determina il tipo dell'oggetto restituito: puo` essere un tipo ctypes o un carattere (typecode) del genere usato dal modulo array (https://docs.python.org/3/library/array.html)
  - Se *size_or_initializer* è un intero allora esso determina la lunghezza dell'array e l'array inizialmente conterra` solo zeri. In alternativa *size_or_initializer* è una sequenza usata per inizializzare l'array e la lunghezza dell'array è fissata dalla lunghezza della sequenza.
  - *Se lock* è True (il valore default) allora viene creato un nuovo oggetto lock per sincronizzare l'accesso all'array. .Se lock è False l'oggetto restituito non sara` automaticamente protetto da un lock e quindi non sara` necesssariamente process-safe.