9-Android & Concurrency

Concurrency

La concorrenza è la capacità di eseguire contemporaneamente diverse parti di un programma o diversi programmi, può migliorare notevolmente la velocità di esecuzione di un programma se determinate attività possono essere svolte in modo asincrono o in parallelo.

Gli utenti di computer danno per scontato che i loro sistemi possano fare più di una cosa contemporaneamente, assumendo di poter continuare a lavorare su un elaboratore di testi mentre altre applicazioni scaricano file, gestiscono la coda di stampa e riproducono audio in streaming.
Anche un'applicazione singola spesso deve svolgere più di una funzione contemporaneamente, ad esempio, un'applicazione di streaming audio deve leggere contemporaneamente i dati audio digitali dalla rete, decomprimerli, gestire la riproduzione e aggiornare la visualizzazione.

Anche un elaboratore di testi deve essere sempre pronto a rispondere a eventi di tastiera e mouse, indipendentemente da quanto sia occupato a riformattare il testo o aggiornare la visualizzazione.

> Nota
> Il software in grado di fare tali cose è noto come software concorrente

Quasi tutti i computer di oggi hanno più CPU o più core all'interno di una CPU e la capacità di sfruttare questi multi-core può essere la chiave per un'applicazione ad alto volume di successo.

Approaches to concurrency

Gli approcci alla concorrenza sono:

- Comunicazione a memoria condivisa
  - I componenti concorrenti comunicano lavorando concorrentemente sugli stessi contenuti di un'area condivisa di memoria
  - Di solito richiede l'utilizzo di una qualche forma di locking (ad esempio, mutex, semafori o monitor) per coordinare tra i thread che lavorano sulle stesse sezioni critiche
- Comunicazione per scambio di messaggi
  - I componenti concorrenti comunicano scambiandosi messaggi, lo scambio può essere effettuato in modo asincrono o utilizzando uno stile di rendezvous in cui il mittente si blocca fino a quando il messaggio non viene ricevuto
  - La concorrenza basata sul passaggio di messaggi tende ad essere molto più facile da comprendere rispetto alla concorrenza basata su memoria condivisa ed è generalmente considerata una forma più robusta di programmazione concorrente

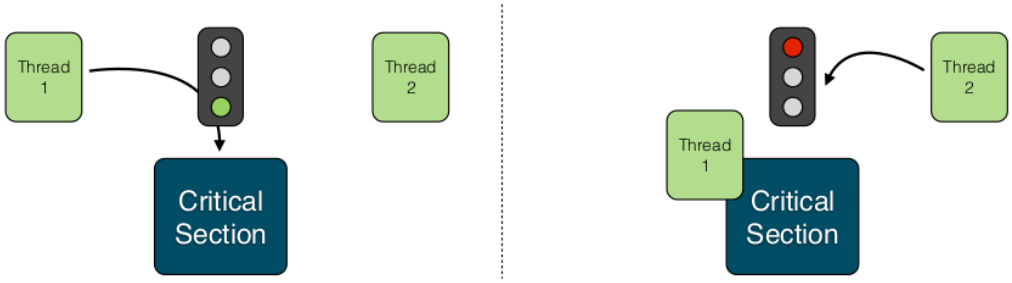
> Nota
> Lo scambio di messaggi asincrono può essere affidabile o non affidabile (talvolta chiamato "invia e prega")

Critical section

Una sezione critica è una porzione di codice che accede a risorse (strutture dati o dispositivi) condivise, che non devono essere acceduto contemporaneamente da più thread.

> Nota
> Di solito si conclude in un tempo fisso, e un thread, un'attività o un processo deve attendere un tempo fisso per accedervi

Un meccanismo di sincronizzazione è richiesto all'ingresso e all'uscita delle sezioni critiche per assicurare l'utilizzo esclusivo.



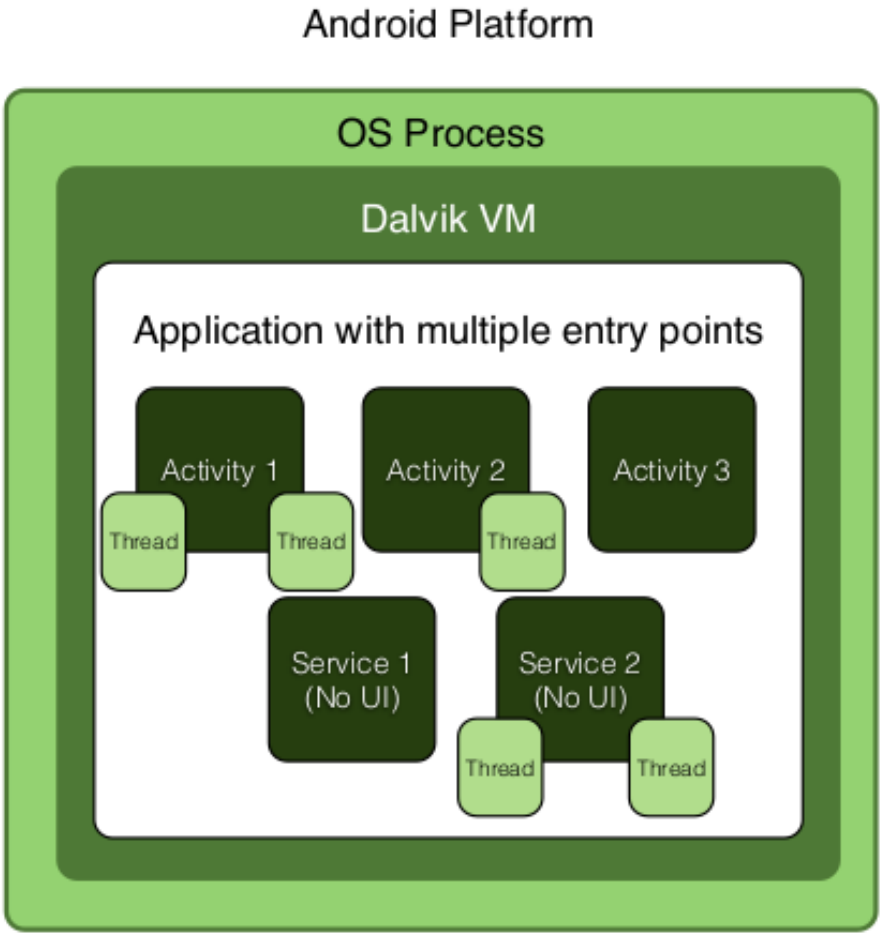
Process & Threads

Processo

Un processo viene eseguito indipendentemente e isolato da altri processi, non può accedere direttamente ai dati condivisi in altri processi. Le risorse del processo vengono allocate attraverso il sistema operativo, ad esempio memoria e tempo della CPU.

Threads

I thread sono chiamati processi leggeri che hanno il proprio stack di chiamate ma possono accedere a dati condivisi. Ogni thread ha la propria cache di memoria, se legge dati condivisi li memorizza nella sua propria cache di memoria.



Concurrency issues

Quando due o più thread hanno accesso allo stesso set di variabili, è possibile che i thread modifichino tali variabili in modo tale da produrre corruzione dei dati e rompere la logica dell'applicazione.
Ci possono essere 2 problemi basilari:

- Visibilità, questo tipo di problema occorre se un thread legge dati condivisi che successivamente vengono modificati da un altro thread e il thread che aveva letto i dati in precedenza non vede la modifica
- Accesso, questo tipo di problema occorre quando più thread cercano di accedere e modificare gli stessi dati condivisi contemporaneamente

Questi problemi possono portare a:

- Fallimento di vitalità, il programma non reagisce più a causa di problemi nell'accesso concorrente ai dati, ad esempio deadlock
- Fallimento di sicurezza, il programma crea dati incorretti

Concurrency & Java

Java supporta i thread come parte del linguaggio, dalla versione 1.5 fornisce anche un miglior supporto per la concorrenza con il pacchetto `java.util.concurrent`.
Inoltre fornisce anche i "locks" per proteggere alcune parti del codice in modo che possano essere eseguite da diversi thread contemporaneamente.
Il modo più semplice per bloccare un determinato metodo o classe Java è utilizzare la parola chiave `synchronized` in una dichiarazione di metodo o classe il quale garantisce che:

- Solo un singolo thread possa eseguire un blocco di codice contemporaneamente
- Ogni thread che entra in un blocco di codice sincronizzato veda gli effetti di tutte le modifiche precedenti che sono state protette dallo stesso blocco di sincronizzazione

> Nota
> Una semplice regola per evitare violazioni di sicurezza dei thread in Java è: "Quando due thread diversi accedono alla stessa variabile di stato modificabile (mutable state), tutti gli accessi a tale stato devono essere eseguiti mantenendo un singolo blocco di sincronizzazione"

`synchronized` keyword

La sincronizzazione è necessaria per l'accesso reciproco esclusivo di blocchi di codice e per una comunicazione affidabile tra i thread.

La parola chiave `sincronized` può essere utilizzata in tre contesti:

- Per creare un blocco (in questo caso, la parola chiave prende come argomento un riferimento a un oggetto da utilizzare come semaforo. I tipi primitivi non possono essere utilizzati come semaforo, ma qualsiasi oggetto può esserlo)
- Su un metodo dinamico
- Su un metodo statico

Synchronized

```
public class SynchronizationExample {
        public synchronized void firstSynchMethod() {
                // ...
        }
        public synchronized void secondSynchMethod() {
                synchronized(this){
                        // ...
                }
        }
        private Object objLock = new Object();
        public synchronized void thirdSynchMethod() {
                synchronized(objLock) {
                        // ...
                }
        }
}
```

`synchronized`, `wait` & `notify`

`sincronized` è molto conveniente ma bisogna usarlo con cautela, per esempio una classe complicata che ha diversi metodi molto utilizzati e sincronizzarli potrebbe voler dire prepararli per una competizione per il lock (lock contention).

Se diversi thread esterni stanno cercando di accedere contemporaneamente a pezzi di dati non correlati, è meglio proteggere quei pezzi di dati con blocchi separati.
La classe `java.lang.Object` definisce i metodi seguenti come parte del protocollo di lock che fa parte di ogni oggetto:

- `wait()`, porta l'attuale thread di esecuzione ad attendere fino a quando un altro thread invoca il metodo notify() o il metodo notifyAll() per questo oggetto, o fino a quando non è trascorso un periodo di tempo specificato
- `notify()`, sveglia un singolo thread in attesa sul monitor di questo oggetto, se ci sono thread in attesa su questo oggetto, uno di essi viene scelto per essere risvegliato, la scelta è arbitraria e avviene a discrezione dell'implementazione
- `notifyAll()`, sveglia tutti i thread che sono in attesa sul monitor di questo oggetto

> Nota
> Un thread attende sul monitor di un oggetto chiamando uno dei metodi di attesa (`wait`)

Concurrency & User Interface

Un uso attento della concorrenza è particolarmente importante per un'applicazione con interfaccia utente (User Interface - UI), un programma UI ben scritto utilizza la concorrenza per creare un'interfaccia utente che non si "blocca" mai e che consente all'applicazione di essere sempre pronta a rispondere all'interazione dell'utente, indipendentemente da ciò che sta facendo.

> Nota
> Tradizionalmente, l'interfaccia utente viene eseguita su un singolo thread dedicato a gestire sia l'input (mouse, touchscreen, tastiera, ecc...) che i dispositivi di output (display, ecc...) ed esegue le richieste di ciascuno, sequenzialmente, di solito nell'ordine in cui sono state ricevute

Gli utenti richiedono un'applicazione reattiva, per questo motivo le operazioni che richiedono molto tempo (come networking, salvataggio e caricamento dati) non dovrebbero bloccare il thread principale dell'UI.

> Nota
> Quando si utilizzano timeout, grandi quantità di dati o elaborazioni aggiuntive (come il parsing o l'elaborazione dei dati), è opportuno spostare queste operazioni intensive in termini di tempo fuori dal thread principale dell'UI.

Concurrency in Android

La piattaforma Android supporta quattro diverse modalità di elaborazione attiva in background/attività in background:

- Threads, supporta l'uso della classe `Threads` per eseguire elaborazioni asincrone, fornisce anche il pacchetto `java.util.concurrent` per eseguire operazioni in background, ad esempio utilizzando le classi `ThreadPools` e `Executor`. Solo l'interfaccia utente è autorizzata ad aggiornare l'interfaccia utente, se è necessario aggiornarla da un altro Thread, è necessario sincronizzarsi con il Thread dell'interfaccia utente o è possibile utilizzare le classi `android.os.Handler` o `AsyncTask`
- `Handler`, la classe `Handler` può aggiornare l'interfaccia utente, fornisce metodi per ricevere istanze della classe `Message` o `Runnable`
- `AsyncTask`, è una classe speciale per lo sviluppo di Android che incapsula l'elaborazione in background e facilita la comunicazione e l'aggiornamento dell'interfaccia utente
- `Service`, consente a un'applicazione di implementare operazioni in background di lunga durata, controlla quando viene eseguito il suo servizio esplicitandone l'avvio e la terminazione

AsyncTask is deprecated

Per molto tempo `AsyncTask` è stata una delle soluzioni più ampiamente utilizzate per scrivere codice concorrente in Android, ora però è ufficialmente deprecata.

> Nota
> Un interessante articolo su questo cambiamento può essere trovato al seguente link: asynctask-deprecated (https://www.techyourchance.com/asynctask-deprecated/)

Threads - Define task behavior

`Thread` e `Runnable` sono classi di base di Java per definire il comportamento di un task, attraverso un `Runnable` si può definire il tuo comportamento e poi allegarlo a un thread ed eseguirlo.
Il metodo `run()` contiene il codice che desideri eseguire il codice del `Runnable` non verrà eseguito sul thread dell'interfaccia utente (UI thread), quindi non è possibile modificare direttamente oggetti UI come oggetti View.

> Nota
> La comunicazione con l'interfaccia utente sarà gestita tramite soluzioni dedicate (`Handlers`)

```
public class MyTaskRunnable implements Runnable {
        // ..
        @Override public void run() {
                // Moves the current Thread into the background
                android.os.Process.setThreadPriority(android.os.Process.THREAD PRIORITY
BACKGROUND);
                /*
                Code you want to run on the thread goes here
                */
                doSomething();
```

```
        }
}
```

Threads - Running the thread

```
// Crea un nuova istanza della task
MyTaskRunnable myTask = new MyTaskRunnable();
// Crea un nuovo thread attaccato al Runnable definito
Thread thread = new Thread(myTask);
// Avvia il thread
thread.start();
```

Threads - Managing multiple threads

Per consentire l'esecuzione di più attività contemporaneamente, è necessario fornire una raccolta gestita di thread, una buona soluzione è `ThreadPoolExecutor` per gestire più thread contemporaneamente, consentendo di eseguire un'attività dalla coda quando un thread nel pool diventa disponibile.

> Nota
> Per eseguire una nuova attività, è sufficiente aggiungerla alla coda.

Un pool di thread può eseguire più istanze parallele di un'attività, quindi è necessario assicurarsi che il codice sia thread-safe, racchiudendo e proteggendo le variabili a cui possono accedere più di un thread in un blocco sincronizzato. Questo approccio impedirà a un thread di leggere la variabile mentre un altro sta scrivendo su di essa.

Per istanziare un `ThreadPoolExecutor`, sono necessari i seguenti valori:

- Dimensione iniziale del pool e dimensione massima del pool, il numero iniziale di thread da allocare al pool e il numero massimo consentito. Il numero di thread che è possibile avere in un pool dipende principalmente dal numero di core disponibili per il dispositivo
- Tempo di permanenza e unità di tempo, la durata per cui un thread rimarrà inattivo prima di spegnersi, è interpretata dall'unità di tempo, uno dei valori costanti definite in `TimeUnit`
- Coda delle attività, la coda da cui `ThreadPoolExecutor` prende gli oggetti `Runnable`. Per avviare il codice su un thread, un gestore di pool di thread prende un oggetto `Runnable` dalla coda FIFO(che implementa l'interfaccia BlockingQueue) e lo associa al thread

```
'Schedule a new Task
// A queue of Runnables
private final BlockingQueue<Runnable> workQueue;
// Instantiates the queue of Runnables as a LinkedBlockingQueue
workQueue = new LinkedBlockingQueue<Runnable>(); •
/* * Gets the number of available cores
* (not always the same as the maximum number of cores)
* */
private static int NUMBER_OF_CORES = Runtime.getRuntime().availableProcessors();

// Sets the amount of time an idle thread waits before terminating
private static final int KEEP_ALIVE_TIME = 1;
// Sets the Time Unit to seconds
private static final TimeUnit KEEPALIVE_TIME_UNIT = TimeUnit.SECONDS;
// Creates a thread pool manager
myThreadPool = new ThreadPoolExecutor(
NUMBER_OF_CORES,// Initial pool size
NUMBER_OF_CORES, // Max pool size
KEEP_ALIVE_TIME,
KEEP_ALIVE_TIME_UNIT,
workQueue);

myThreadPool.execute(new MyTaskRunnable()),
```

Handler

L'SDK fornisce una classe di supporto che consente allo sviluppatore di eseguire codice su un altro thread, la classe `Handler` permette di eseguire un pezzo di codice su un thread di destinazione dove p stata istaziata la classe.

> Nota
> Questa classe consente, ad esempio di aggiornare l'interfaccia utente dell'applicazione da un thread diverso che sta eseguendo un'attività in background, come il download di un'immagine da un sito web remoto

Per utilizzare un handler, è necessario sottoclassificarlo e sovrascrivere il metodo `handleMessage()` per elaborare i messaggi, per elaborare un `Runnable`, è possibile utilizzare il metodo `post()`.

> Nota
> È sufficiente avere una sola istanza di Handler nella activity

Si può istanziare un `Handler` nella dichiarazione di una classe classe utilizzando, ad esempio, il seguente codice:

```
// Defines a Handler object that's attached to the UI thread
Handler handler = new Handler(Looper.getMainLooper()) {
        /*
        * handleMessage() defines the operations to perform when
        * the Handler receives a new Message to process.
        */
        @Override
        public void handleMessage(Message inputMessage) {
                //Handle an incoming Message request
        }
        ...
}
```

Sending & receiving handler messages

```
/* Creates a message for the Handler
 * with the state and the task object
 **/

int myState = 1;
Message myMessage = handler.obtainMessage(myState, myDataObject);
myMessage.sendToTarget();

Handler handler = new Handler(Looper.getmainLooper()) {
        @Override public void handlemessage(message inputmessage) {
        // Retrieves data from the Message
        mynataobject mynatacbject = (mynataobject) inputmessage.obj;
        switch (inputmessage.what) {
                case SUCCESS:
                        updateUI(myDataObject)
                        break;
                case ERROR:
                        showErrormessageonta();
                        break;
                default:
                        super.handlemessage(inputmessage);
        }
}
```

Post Direct Runnable to Handler

```
Thread imageDownload = new Thread(new Runnable() {
        @Override
        public void run() {
                Drawable d = null;
                try {
                        InputStream is = (InputStream) new
URL("http://farm8.staticflickr.com/7149/6526427657_3c790c2af7_b.jpg"+image).getContent();
                        d = Drawable.createFromStream(is, "image");
                } catch (Exception e) {
                        e.printStackTrace();
                }
                if(d != null) {
                        final Drawable dFinal = d;
                        handler.post(new Runnable() {
                                @Override
                                public void run() {
                                        ImageView imageView =
(ImageView)findViewById(R.id.imageView);
                                        imageView.setImageDrawable(dFinal);
                                }
                        });
                }
        }
});
imageDownload.start();
```

AsyncTask

`AsyncTask` è una classe di supporto astratta per gestire operazioni in background che alla fine vengono riportate nell'interfaccia utente dell'applicazione, creando un'interfaccia più semplice per le operazioni asincrone rispetto alla creazione manuale di una classe Java Thread.

Invece di creare il proprio thread, è possibile sottoclassificare `AsyncTask` implementando i metodi di evento appropriati.

Un'attività asincrona è definita da un calcolo che viene eseguito su un thread in background e il cui risultato viene pubblicato sul thread dell'interfaccia utente (UI).
Un'attività asincrona è definita da 3 tipi generici chiamati:

- Params
- Progress
- Result,

E da 4 fasi chiamate:

- onPreExecute
- doInBackground
- onProgressUpdate
- onPostExecute

Quando un'attività asincrona viene eseguita passa attraverso le 4 fasi sopra:

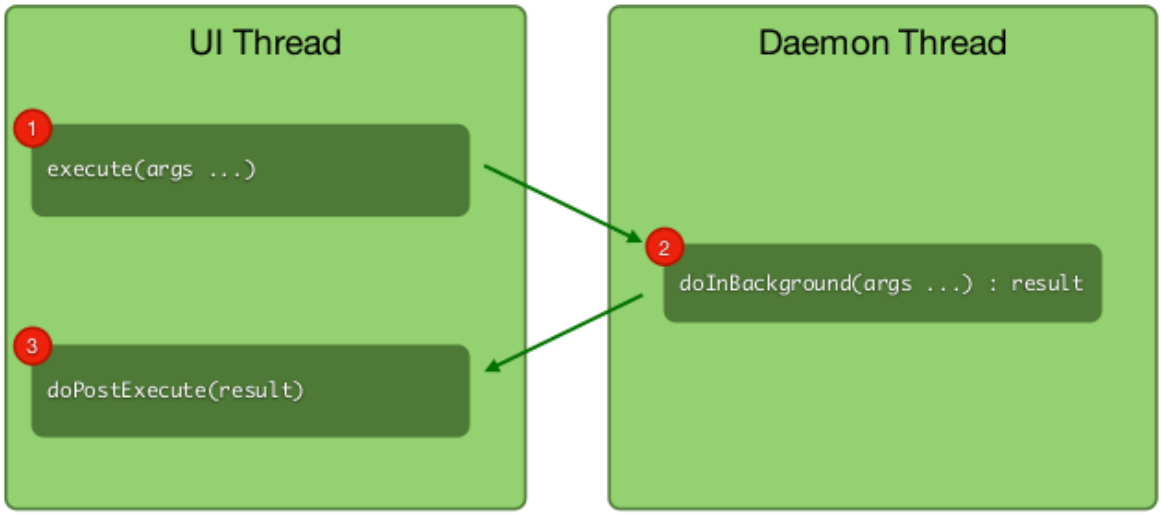
- `onPreExecute()`, viene invocata sul thread dell'interfaccia utente immediatamente dopo l'esecuzione dell'attività. Questa fase viene comunemente utilizzata per configurare l'attività, ad esempio mostrando una barra di avanzamento nell'interfaccia utente
- `doInBackground(Params...)`, viene invocata sul thread in background subito dopo che `onPreExecute()` ha terminato l'esecuzione. Questa fase viene utilizzata per eseguire il calcolo in background che potrebbe richiedere molto tempo, i parametri dell'attività asincrona vengono passati a questa fase. Il risultato del calcolo deve essere restituito da questa fase e sarà passato all'ultima fase. Questa fase può anche utilizzare `publishProgress(Progress...)` per pubblicare uno o più valori di progresso, che vengono pubblicati sul thread dell'interfaccia utente, nella fase `onProgressUpdate(Progress...)`
- `onProgressUpdate(Progress...)`, viene invocata sul thread dell'interfaccia utente dopo una chiamata a `publishProgress(Progress...)`. Il momento dell'esecuzione non è definito. Questo metodo viene utilizzato per visualizzare qualsiasi tipo di progresso nell'interfaccia utente mentre il calcolo in background è ancora in esecuzione, ad esempio può essere utilizzato per animare una barra di avanzamento o mostrare log in un campo di testo
- `onPostExecute(Result)`, viene invocata sul thread dell'interfaccia utente dopo il completamento del calcolo in background. Il risultato del calcolo in background viene passato a questa fase come parametro

Un'attività può essere annullata in qualsiasi momento invocando `cancel(boolean)`, l'invocazione causerà che le chiamate successive a `isCancelled()` restituiscano true.
Dopo aver invocato questo metodo, invece di `onPostExecute(Object)`, verrà invocato `onCancelled(Object)` dopo che `doInBackground(Object[])` è terminato.
Per assicurarsi che un'attività venga annullata il più rapidamente possibile, si dovrebbe sempre controllare periodicamente il valore restituito di `isCancelled()` da `doInBackground(Object[]),` se possibile (ad esempio, all'interno di un ciclo).

Ci sono alcune regole di threading che devono essere seguite affinché questa classe funzioni correttamente:

- L'istanza dell'attività deve essere creata sul thread dell'interfaccia utente
- `execute(Params...)` deve essere invocato sul thread dell'interfaccia utente
- `onPreExecute()`, `onPostExecute(Result)`, `doInBackground(Params...)`, `onProgressUpdate(Progress...)` non devono essere invocati manualmente.
- L'attività può essere eseguita solo una volta (verrà lanciata un'eccezione se si tenta una seconda esecuzione)

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
        protected Long doInBackground(URL... urls) {
                int count = urls.length;
                long totalSize = 0;
                for (int i = 0; i < count; i++) {
                        totalSize += Downloader.downloadFile(urls[i]);
                        publishProgress((int) ((i / (float) count) * 100));
                }
                return totalSize;
        }
        protected void onProgressUpdate(Integer... progress) {
                setProgressPercent(progress[0]);
        }
        protected void onPostExecute(Long result) {
                showDialog("Downloaded " + result + " bytes");
        }
}
new DownloadFilesTask().execute(url1, url2, url3);
```



Mentre AsyncTask semplifica l'implementazione di elaborazioni concorrenti contemporanee, impone forti vincoli che non possono essere verificati automaticamente.

> Nota
> La violazione di tali vincoli causerà bug di concorrenza molto difficili da individuare.

Il vincolo più importante è che il metodo `doInBackground` venga eseguito su un thread diverso, per questo motivo, dovrebbe fare solo riferimenti thread-safe alle variabili ereditate nel suo ambito.

> Nota
> Bisognerebbe evitare che una o più variabili vengano accedute da due thread diversi senza sincronizzazione

> Nota
> La soluzione migliore a questo problema è rendere gli argomenti di AsyncTask immutabili. Se non possono essere modificati, sono thread-safe e non necessitano ulteriori precauzioni