Programmazione asincrona

Implementazione di operazioni asincrone e gestione di scenari multithread

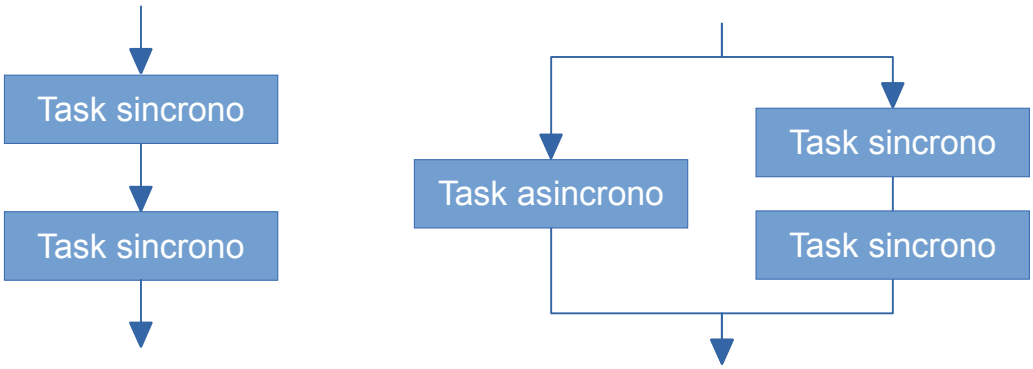
Ambiente: .NET 4.0+

Anno 2013/2014

Indice generale

1 Operazioni asincrone vs operazioni sincrone

Il termine *operazione asincrona* designa un task che può essere eseguito parallelamente ad altri task. All'opposto, un'*operazione sincrona* viene eseguita sequenzialmente (prima o dopo) rispetto alle altre operazioni.



1.1 Operazioni sincrone

L'esecuzione sequenziale delle operazioni sincrone rende semplice implementare:

1. lo scambio di dati;
2. l'accesso a variabili e risorse condivise;
3. la sincronizzazione, e cioè l'ordine con il quale vengono eseguite le operazioni.

Ma l'uso di operazioni sincrone ha anche degli svantaggi:

1. Sono **bloccanti**: l'operazione successiva non comincia finché non termina l'operazione precedente;
2. Non consentono di ottimizzare l'uso della CPU, che resta inattiva mentre attende il completamento di operazioni di I/O.
3. Non consentono di gestire più richieste contemporaneamente.
4. Non consentono di sfruttare il *parallelismo* dell'hardware. (In un certo istante soltanto un'operazione è in esecuzione, indipendentemente dal numero di *core* presenti nel sistema).

1.2 Operazioni asincrone

Le operazioni asincrone sono fondamentali nei seguenti scenari:

1. Eseguire task che non blocchino la UI. (*offloading task*)
2. Gestire contemporaneamente più task, indipendentemente dalla presenza di una architettura parallela (*multicore*);
3. Migliorare le performance del codice eseguito nelle architetture parallele.

1.3 Supporto alle operazioni asincrone in .NET

.NET fornisce un supporto completo alla programmazione asincrona, mettendo a disposizione un insieme di classi progettate allo scopo. Esistono inoltre alcune classi che implementano dei procedimenti sia mediante metodi sincroni, che attraverso metodi asincroni.

Classi d'uso generale per la gestione di scenari asincroni

Thread, Task, TaskFactory, TaskCompletionSource, ...

Classi per la sincronizzazione

Monitor, Mutex, AutoResetEvent, ManualResetEvent, ...

Classi per l'esecuzione parallela: Parallel LINQ (PLINQ, o PFX)

Esecuzioni di query parallele, che sfruttano l'architettura multicore.

Collezioni concorrenti (thread-safe)

ConcurrentBag<>, ConcurrentStack<>, ConcurrentQueue<>, BlockingCollection<>, ...

Metodi per operazioni asincrone "awaitable"

HttpClient: GetAsync()

WebClient: DownloadTaskAsync

Stream: ReadAsync

Pattern APM: Asynchronous Programming Model (obsoleto)

Stream: BeginRead/EndRead

TcpClient: BeginConnect/EndConnect

Pattern EAP: Event-based Asynchronous Pattern (attualmente poco utilizzato)

WebClient: DownloadAsync (evento: DownloadCompleted)

Classe BackgroundWorker

> **Pattern APM e EAP**
>
> Entrambi i pattern di programmazione asincrona sono ormai superati dalla *Task Parallel Library*, introdotta nella versione .NET 4.0. Comunque, sono ancora implementati in molti componenti.

2 Task Parallel Library (TPL)

La **TPL** è un insieme di classi progettate per la programmazione asincrona. Al centro di tutto c'è la classe **Task**.

2.1 Introduzione alla classe Task

Un task è *un'unità di lavoro che può essere svolto in parallelo*. L'unità di lavoro corrisponde ad un metodo (che può essere anche un metodo anonimo o una *lambda expression*).

Un task può essere creato specificando il metodo da eseguire nel costruttore:

```
Task t = new Task(DoSomething);     // DoSomething è metodo da eseguire nel task
t.Start();                          // Esegue il task
```

Lo stesso risultato può essere ottenuto più concisamente utilizzando **Factory.StartNew()**, dove **Factory** è una proprietà statica della classe Task:

```
Task t = Task.Factory.StartNew(DoSomething);  // Crea ed esegue il task
```

Infine, è possibile utilizzare il metodo statico **Run()**:

```
Task t = Task.Run( ()=>DoSomething() ); // viene usata un'espressione lambda
```

> **Run() vs StartNew()**
>
> Il metodo **Run()** richiede espressamente una *lambda expression* e rappresenta una versione semplificata di **Factory.StartNew()**.

2.1.1 Esempio di creazione di un task

Il codice che segue crea un task e lo esegue.

```
static void Main(string[] args)
{
    Task t = new Task(DoSomething);         // crea il task
    t.Start();                              // esegue il task
    Console.WriteLine("Main Thread: " + Thread.CurrentThread.ManagedThreadId);
    Console.ReadKey();                      // necessaria per bloccare il main thread
}

private static void DoSomething()
{
    Console.WriteLine("Task Something: " + Thread.CurrentThread.ManagedThreadId);
}
```

Per dimostrare l'esecuzione asincrona del metodo, il codice visualizza **l'Id** del thread principale e di quello creato con il task:

Output

```
Main Thread: 1
Task Something: 3
```

Nota bene: **l'Id** del thread è diverso nei due casi; ciò dimostra che **Main()** e **DoSomething()** sono eseguiti in thread separati. (il valore 1 corrisponde al thread principale.)

> ### Avvio con debug
>
> Se il programma viene eseguito in modalità "Avvia debug" i due **Id** potrebbero essere diversi da quelli mostrati.
>
> Inoltre, poiché il tempo di avvio dei task non è costante, potrebbe accadere che **DoSomething()** sia eseguito prima che venga visualizzato il messaggio in **Main()**.

2.2 Task vs Thread

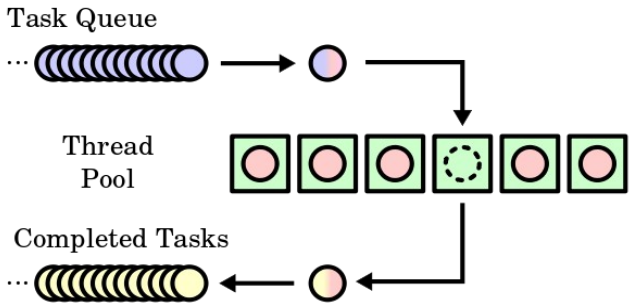
La classi **Task** e **Thread** sono similari; entrambe consentono di eseguire parallelamente un'unità di lavoro. Il precedente codice può essere riscritto usando la classe **Thread** invece di **Task**:

```
static void Main(string[] args)
{
    Thread t = new Thread(DoSomething);           // crea il thread
    t.Start();                                    // esegue il thread
    Console.WriteLine("Main Thread: " + Thread.CurrentThread.ManagedThreadId);

    Console.ReadLine();
}

private static void DoSomething()
{
    Console.WriteLine("Task Something: " + Thread.CurrentThread.ManagedThreadId);
}
```

Task è da preferire, poiché è più efficiente, sfrutta l'eventuale parallelismo dell'architettura ed è al centro della *Task Parallel Library*.

2.2.1 Thread pool

I thread creati dalla TPL sono gestiti dal ***thread-pool***, uno *schedulatore* il cui compito è quello di ottimizzare le risorse impiegate per gestire i thread dell'applicazione.



Il *thread-pool* mantiene una lista di (oggetti) thread già creati, ma non ancora in esecuzione. Quando è necessario eseguire un'operazione asincrona, il *thread-pool* assegna il metodo a uno dei thread disponibili. Dopo che il metodo è terminato, il *thread-pool* "ricicla" il thread e lo rende disponibile per un'altra operazione.

È possibile utilizzare direttamente il *thread-pool* utilizzando il metodo statico **ThreadPool.QueueUserWorkItem()**:

```
static void Main(string[] args)
{
    ThreadPool.QueueUserWorkItem(DoSomething);
    Console.ReadLine();
}
private static void DoSomething(object o)
{
    Console.WriteLine("Thread del thread pool " +
                      Thread.CurrentThread.IsThreadPoolThread);
}
```

Nota bene: la proprietà **IsThreadPoolThread** indica che il thread non è stato creato dall'utente, ma appartiene al *thread-pool*.

Output

```
Thread del thread pool: true
```

> **Task e thread pool**
>
> I task creati mediante la classe Task fanno parte del thread-pool e dunque non c'è alcun vantaggio nell'usare il metodo statico **QueueUserWorkItem()**.

2.3 Creazione di un long running task

Un task può riferirsi a un'operazione asincrona di breve durata, a un procedimento che impiega molto tempo a essere completato, oppure a un procedimento che non termina mai (un servizio).

È possibile specificare la natura del task mediante il tipo **TaskCreationOptions:**

```
Task.Factory.StartNew(DoSomething, TaskCreationOptions.LongRunning);
```

Oppure:

```
Task t = new Task(DoSomething, TaskCreationOptions.LongRunning);
t.Start();
```

2.4 Passaggio di valori al task

Durante la costruzione del task è possibile passare un valore al metodo eseguito, purché questo definisca un solo parametro di tipo **object**:

```
static void Main(string[] args)
{
    Task.Factory.StartNew(DoSomething, "ARGOMENTO");   // crea ed esegue il task
    //...
    Console.ReadLine();
}

private static void DoSomething(object arg)   // riceve il valore "ARGOMENTO"
{
    Console.WriteLine(arg.ToString() + Thread.CurrentThread.ManagedThreadId);
}
```

2.4.1 Passaggio di argomenti mediante lambda expression

Utilizzando una LE è possibile invocare metodi con un numero qualsiasi di parametri. Il precedente codice può essere scritto come segue:

```
Task.Factory.StartNew(() => DoSomething("ARGOMENTO"));
```

Oppure:

```
Task.Run(() => DoSomething("ARGOMENTO"));
```

Nota bene: il *delegate* corrispondente alla LE è **Action**, e cioè un *delegate* che non accetta parametri e restituisce **void**.

2.5 Task che ritornano un valore

La TPL definisce una classe generica, **Task<>**, che consente a un task di ritornare un valore, accessibile attraverso la proprietà **Result**.

L'esempio seguente usa il metodo **DownloadString()** della classe **WebClient** per scaricare la pagina web di un sito. Il metodo viene eseguito all'interno di un task; quest'ultimo ritorna una stringa contenente l'intera pagina.

```
private static void DownloadDemo()
{
    Task<string> t = Task.Factory.StartNew<string>(DownloadPage);
    //...
    string pageText = t.Result;     // accesso al risultato del task (HTML della pagina)
    Console.WriteLine(pageText);
}

private static string DownloadPage()
{
    WebClient cli = new WebClient();
    return cli.DownloadString("http://www.isisfermi.it");
}
```

Il task è di tipo **Task<string>**, poiché **string** è il tipo ritornato dal metodo **DownloadPage()**;

Il precedente esempio introduce una questione molto importante, che affronteremo in seguito: l'accesso al valore prodotto da un altro thread. Questa problematica riguarda in generale la sincronizzazione tra operazioni asincrone.

3 Sincronizzare i task

La sincronizzazione tra operazioni è un problema centrale della programmazione asincrona. Qui ci occupiamo di un ambito ristretto e cioè quello di attendere la fine di un'operazione asincrona, eventualmente per accedere al suo risultato.

3.1 Attendere la fine di un task: metodo Wait()

La classe **Task** definisce un metodo, **Wait()**, che blocca il thread chiamante in attesa che il task sia terminato.

Il seguente codice crea un task, visualizza un messaggio che simula un lavoro e quindi attende la terminazione del task. Quest'ultimo ritarda la propria terminazione di tre secondi.

```
static void Main(string[] args)
{
    Task t = new Task.Factory.StartNew(DoSomething);
    Console.WriteLine("Main thread: simulazione lavoro eseguito in parallelo" );
    t.Wait();    // qui il metodo attente
    Console.WriteLine("Main thread: attesa sul task terminata");
    Console.ReadLine();
}

private static void DoSomething()
{
    Console.WriteLine("Task DoSomething");
    Thread.Sleep(3000);
}
```

L'esecuzione provoca l'immediata visualizzazione del primo messaggio sul thread principale, seguito dal messaggio prodotto dal metodo `DoSomething()`. Sulla chiamata di **Wait()** il main thread si blocca, attendendo che il task termini; ciò avviene dopo 3 secondi.

Output

```
Main thread: simulazione lavoro eseguito in parallelo
Task DoSomething
Main thread: attesa sul task terminata
```

3.2 Ottenere il risultato di un task: proprietà Result

Riprendiamo il codice di download della pagina web in una stringa:

```
private static void DownloadDemo()
{
    Task<string> t = Task.Factory.StartNew<string>(DownloadPage);
    //...
    string pageText = t.Result;     // accesso al risultato del task: attende che termini
    Console.WriteLine(pageText);
}
```

L'accesso alla proprietà **Result** equivale all'invocazione del metodo **Wait():** blocca il chiamante fino a quando il risultato non è disponibile. (Se il risultato è già disponibile, il chiamante non viene bloccato.)

3.2.1 Proprietà Result: accesso thread-safe

Nonostante la proprietà **Result** sia valorizzata in un thread diverso da quello chiamante, il suo uso è *thread-safe*.

3.3 Errori nel codice asincrono

Una caratteristica fondamentale delle operazioni asincrone è quella di non propagare le eccezioni al codice chiamante. Questo rappresenta un problema, poiché un task potrebbe fallire senza che il codice chiamante ne venga a conoscenza.

Wait() e **Result** risolvono questo problema, poiché restituiscono al codice chiamante eventuali eccezioni che si sono verificate durante l'esecuzione del task.

Nell'esempio seguente, il metodo `DoSomethingError()` genera volutamente un'eccezione di tipo **NotImplementedException**:

```
private static void DemoDownloadErrori()
{
    Task t = Task.Factory.StartNew(DoSomethingError);
    Console.WriteLine("Main thread: nessun errore per ora.\n\n");
    //...
    t.Wait();            // qui viene propagata l'eccezione al main thread
}

private static void DoSomethingError()
{
    throw new NotImplementedException("Metodo non implementato");
}
```

L'esecuzione del metodo `DemoDownloadErrori()` provoca il seguente risultato:

Output

```
Main thread: nessun errore per ora.
Eccezione non gestita: System.AggregateException: Si sono verificati uno o
più errori.--->System.NotImplementedException: Metodo non implementato
```

L'eccezione propagata al codice chiamante è di tipo **AggregateException**. Si tratta di un tipo creato apposta per "aggregare" altre eccezioni. Per accedere all'eccezione realmente generata occorre utilizzare la proprietà **InnerException**.

> **AggregateException**
>
> Viene usato il tipo **AggregateException** perché un task potrebbe generare più di un'eccezione. Infatti, un task può avere dei *child-task* i quali possono a loro volta generare delle eccezioni.

3.4 Stato finale di un task

La classe **Task** definisce tre proprietà che informano sulla modalità di terminazione di un task:

Proprietà	Descrizione
IsCompleted	Il task è terminato normalmente.
IsCanceled	Il task è stato cancellato dal codice chiamante
IsFaulted	Il task è terminato con errore.

3.5 Continuazione di un task

Un ***continuation task*** viene eseguito soltanto come continuazione di un altro task, chiamato ***antecedent task***. Possono esistere vari motivi per eseguire un procedimento come *continuation task*, tra i quali:

1. eseguirlo in base alla modalità di terminazione dell'antecedente (completato, con errore, cancellato);
2. eseguirlo in un contesto diverso dall'antecedente. (Utile per sincronizzare un'operazione asincrona con il thread che gestisce la UI).

Si stabilisce la modalità di esecuzione del nuovo task mediante l'enum **TaskContinuationOptions**.

3.5.1 Esecuzione di un continuation task

Il codice seguente crea un task e un *continuation task*; il secondo viene eseguito dal metodo **ContinueWith()**. Il codice eseguito nel *continuation task* riceve come argomento un riferimento al task antecedente.

```
private static void ContinuationTaskDemo()
{
    Task t = Task.Factory.StartNew(DoSomethingAnt);
    t.ContinueWith(DoSomethingCont); // esegue il task di continuazione
                                     // viene eseguito soltanto dopo che "t" è terminato
}
private static void DoSomethingAnt()
{
    Console.WriteLine("Id task antecedente: " + Thread.CurrentThread.ManagedThreadId);
}
```

```
private static void DoSomethingCont(Task t)
{
    Console.WriteLine("Task antecedente completato: " + t.IsCompleted);
    Console.WriteLine("Id task di continuazione: "+Thread.CurrentThread.ManagedThreadId);
}
```

L'output dimostra che i due task vengono eseguiti in thread distinti e che il task antecedente è stato completato prima di eseguire il task di continuazione.

Output

```
Id task antecedente: 3
Task antecedente completato: True
Id task di continuazione: 4
```

3.5.2 Esecuzione condizionata di un continuation task

Una possibilità interessante è quella di eseguire un *continuation task* in base al fatto che il task antecedente sia terminato con successo oppure no, come mostra il seguente codice:

```
private static void ContinuationTaskDemoCond()
{
    Task t = Task.Factory.StartNew(DoSomethingException);
    t.ContinueWith(ContinuationOnCompleted,
                   TaskContinuationOptions.OnlyOnRanToCompletion);
    t.ContinueWith(ContinuationOnError, TaskContinuationOptions.OnlyOnFaulted);
}

private static void DoSomethingException()
{
    throw null;       // genera un'eccezione
}

// eseguito soltanto se il task antecedente completa correttamente
private static void ContinuationOnCompleted(Task t)
{
    Console.WriteLine("Task antecedente completato: " + t.IsCompleted);
}

// eseguito soltanto se il task antecedente genera un'eccezione
private static void ContinuationOnError(Task t)
{
    Console.WriteLine("Task antecedente fallito: " + t.IsFaulted);
    Console.WriteLine("Eccezione: " + t.Exception.InnerException.Message);
}
```

Le costanti **OnlyOnRanToCompletition** e **OnlyOnFaulted** stabiliscono in quale condizione debba essere eseguito il task di continuazione.

Nell'esempio, il codice eseguito nel task genera un'eccezione e dunque viene eseguito `ContinuationOnError()`. Quest'ultimo, attraverso il riferimento al task antecedente, visualizza il tipo di eccezione generato.

4 Pattern di programmazione asincrona

Di seguito sono introdotti alcuni scenari di programmazione asincrona allo scopo di esaminare varie tecniche di sincronizzazione. Contemporaneamente sarà approfondito il supporto offerto da .NET alla programmazione asincrona.

La sincronizzazione di operazioni asincrone segue alcuni modelli:

1. **Fire&Forget** (nessuna sincronizzazione).
2. **Polling**.
3. **APM**: Asynchronous Programming Model.
   - Delegati asincroni.
4. **EAP**: Event-based Asynchronous Pattern.
5. **async/await** (uso della TPL per produrre metodi "awaitable")
6. **Producer/Consumer**.

4.1 Fire&Forget

Fire&Forget (letteralmente: *spara e dimentica*) non implica alcuna forma di sincronizzazione. Il chiamante esegue l'operazione asincrona e poi si disinteressa del suo andamento e/o risultato.

4.2 Polling

Nel ***polling*** il chiamante esegue l'operazione asincrona e quindi ne verifica periodicamente il completamento o il fallimento. Perché ciò sia possibile è necessario che sia disponibile un modo per conoscere lo stato del task asincrono (completato, errore, etc).

Il *polling* è raramente la scelta ideale, e può risultare poco efficiente, poiché il chiamante usa tempo di CPU soltanto per verificare se l'operazione asincrona è stata completata.

4.3 APM: Asynchronous Programming Model

Nel pattern **APM** l'operazione asincrona viene implementata mediante una coppia di metodi, per convenzione denominati **BeginXXX()** e **EndXXX()**, dove "XXX" definisce il nome dell'operazione.

Per compatibilità verso quei componenti che supportano ancora questo pattern, il metodo **Task.Factory.FromAsync()** è in grado di "trasformare" un'operazione APM in un task.

4.3.1 Delegati asincroni

Nel momento in cui si definisce un *delegate*, .NET ne implementa automaticamente una versione asincrona che consente di invocare il metodo in un thread diverso dal chiamante. L'esecuzione segue il pattern APM ed è realizzato mediante una coppia di metodi **BeginInvoke**/**EndInvoke**.

Analogamente all'APM, anche i delegati asincroni sono ormai superati.

4.4 EAP: Event-based Asynchronous Pattern

L'**EAP** è implementato da un *metodo asincrono* e un *evento di completamento*. Questi consentono di avviare un'operazione asincrona e di ricevere una notifica del suo completamento, o fallimento.

Esempi di implementazione del pattern sono le classi **BackgroundWorker** e **WebClient**.

Il codice seguente scarica una pagina web utilizzando il metodo asincrono **DownloadFileAsync()** della classe **WebClient**. Il parametro `e` fornisce le informazioni sullo stato di completamento dell'operazione:

```
private void btnDownload_Click(object sender, EventArgs e)
{
    WebClient cli = new WebClient();
    var uri = new Uri("http://www.google.it");
    cli.DownloadFileCompleted += cli_DownloadFileCompleted;  //sottoscrive l'evento
    cli.DownloadFileAsync(uri, @"d:\page.html");             //chiamata asincrona
}

void cli_DownloadFileCompleted(object sender, AsyncCompletedEventArgs e)
{
    if (e.Cancelled)                // verifica se l'operazione è stata cancellata
        return;

    if (e.Error == null)            // verifica se c'è stato un errore
        MessageBox.Show("Pagina scaricata");
    else
        MessageBox.Show("Errore download: " + e.Error.Message);
}
```

4.4.1 Accesso thread-safe alla UI nell'evento di completamento

L'evento di completamento accede alla UI in modo *thread-safe*.

4.4.2 Cancellare un'operazione asincrona

Tipicamente, il pattern EAP prevede che si possa abortire l'operazione. A questo scopo la classe **WebClient** fornisce il metodo **CancelAsync()**.

```
WebClient cli = new WebClient();          // cli deve essere globale
private void btnCancelDownload_Click(object sender, EventArgs e)
{
    cli.CancelAsync();
}
```

4.4.3 Evento “xxxChanged”: avanzamento dell'operazione

EAP prevede la notifica dello stato di avanzamento dell'operazione asincrona. A questo scopo la classe **WebClient** definisce l'evento **DownloadProgressChanged**, che può essere utilizzato per aggiornare la UI sull'avanzamento del download (ad esempio mediante una *progressive bar*)

> **Pattern EAP e classe BackgroundWorker**
>
> La classe **BackgroundWorker** fornisce un'implementazione personalizzabile del pattern. La classe non implementa di per sé alcuna operazione; si limita a fornire un'infrastruttura che facilita l'implementazione del pattern.

4.5 async/await

La versione 5.0 del linguaggio C# introduce un nuovo approccio alla programmazione asincrona, con l'obiettivo di nascondere le problematiche relative all'uso della TPL e della sincronizzazione. A questo scopo esistono le parole chiave **async** e **await**.

Decorando la definizione di un metodo con la parola chiave **async** si stabilisce che può contenere una chiamata asincrona (anche se non obbligatoriamente). Decorando l'invocazione di un metodo con **await** si esegue una chiamata asincrona.

Il linguaggio impone dei vincoli nell'uso di **async/await**: soltanto i metodi decorati con **async** possono utilizzare la parola chiave **await**.

Di seguito è mostrato il download asincrono di una pagina web:

```
private async void btnAsyncAwait_Click(object sender, EventArgs e)
{
    WebClient cli = new WebClient();
    var uri = new Uri("http://www.google.it");
    await cli.DownloadFileTaskAsync(uri, @"d:\page.html");
    MessageBox.Show("File scaricato");
}
```

Ad eccezione delle parole **async** e **await**, il codice è identico a quello che avremmo scritto per una chiamata sincrona. Dietro le quinte, però, il linguaggio genera il codice necessario per:

1. sospendere l'esecuzione del metodo sulla chiamata a **DownloadFileTaskAsync()**;
2. ridare il controllo al thread chiamante (e quindi gestire gli eventi della UI);
3. riprendere l'esecuzione del metodo dopo che l'operazione asincrona è stata completata, eseguendo le restanti istruzioni (la *message dialog* , in questo caso).

4.5.1 Gestione delle eccezioni

L'uso di **await** propaga le eccezioni al thread chiamante. Per gestirle basta proteggere la chiamata asincrona con un **try...catch**:

```
private async void btnAsyncAwait_Click2(object sender, EventArgs e)
{
    WebClient cli = new WebClient();
    try
    {
       var uri = new Uri("http://www.google.it");
       await cli.DownloadFileTaskAsync(uri, @"d:\page.html");
        MessageBox.Show("File scaricato");
    }
    catch (Exception ex)
```

```
    {
        MessageBox.Show("Errore: "+ex.Message);
    }
}
```

4.5.2 await “vs” Wait() e Result

Diversamente dalla chiamata al metodo **Wait()** o all'uso della proprietà **Result** di un task, l'uso di **await**:

- sospende il metodo chiamante, ma *non blocca il thread*;
- propaga l'eccezione effettivamente prodotta e non il tipo **AggregateException**.

4.5.3 Ritardare la sospensione del metodo chiamante

Non è obbligatorio usare **await** direttamente nella chiamata asincrona. È possibile ottenere un task dalla chiamata e attendere successivamente la sua terminazione. Lo scopo è quello di eseguire parallelamente del lavoro prima di sospendere il metodo chiamante.

```
private async void btnAsyncAwait_Click(object sender, EventArgs e)
{
    WebClient cli = new WebClient();
    var uri = new Uri("http://www.google.it");
    Task t = cli.DownloadFileTaskAsync(uri, @"d:\page.html");

    DoSomething();        // qui viene svolto del lavoro nel thread chiamante

    await t;              // il metodo viene sospeso
    MessageBox.Show("File scaricato");
}
```

`DoSomething()` viene eseguito immediatamente, parallelamente all'operazione asincrona. Dopodiché, se l'operazione asincrona non è ancora terminata, il metodo chiamante viene sospeso. Dopo che l'operazione è terminata, il chiamante viene ripreso e viene visualizzata la *message dialog*.

5 Producer/Consumer

Il pattern producer/consumer affronta un classico problema di sincronizzazione:

due processi[1] (chiamati "produttore" e "consumatore") condividono un buffer, di solito di dimensione prefissata (*bounded buffer*). Compito del produttore è inserire dati nel buffer, compito del consumatore è quello di estrarli e processarli.

Il pattern porta con sé diversi problemi di sincronizzazione.

Accesso thread safe al buffer

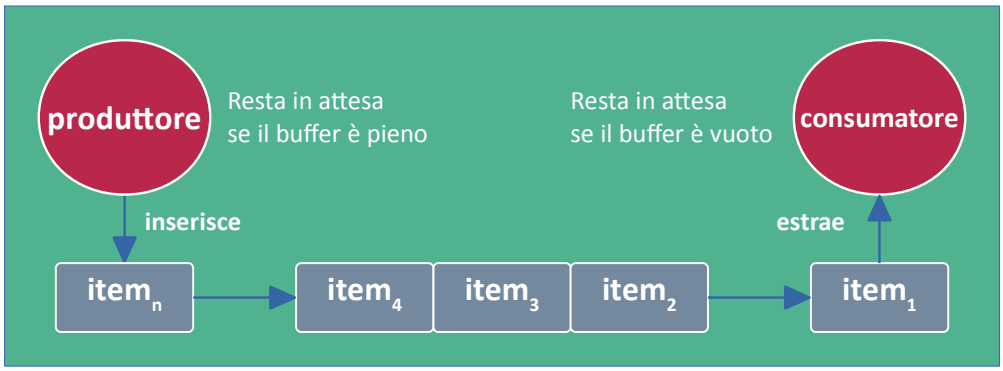
Il buffer è condiviso tra processi eseguiti in modo asincrono uno dall'altro. Pertanto le operazioni di inserimento ed estrazione di un elemento devono essere *thread-safe*.

Attesa del produttore in caso di buffer pieno

Se il buffer è pieno il produttore deve mettersi in attesa prima di poter inserire un nuovo elemento. L'attesa deve terminare appena si rende disponibile una posizione libera.

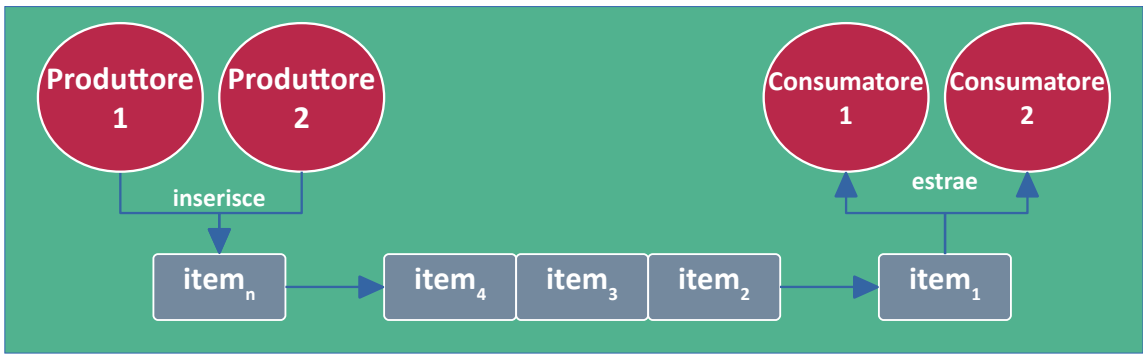
Attesa del consumatore in caso di buffer vuoto

Se il buffer è vuoto il consumatore deve mettersi in attesa. L'attesa deve terminare appena si rende disponibile un elemento da estrarre.



5.1 Generalizzazione del pattern producer/consumer

In realtà non esiste alcun vincolo al numero di produttori e/o consumatori:



1 Qui, il termine "processo" designa in generale un procedimento che viene eseguito in parallelo agli altri.

5.2 Implementazione del pattern P/C: BlockingCollection<>

La classe **BlockingCollection** fornisce una soluzione già pronta all'uso che implementa il pattern P/C. Infatti:

1. garantisce l'accesso *thread-safe* al buffer, gestendolo con una *concurrent collection*: **ConcurrentQueue**, **ConcurrentStack**, o **ConcurrentBag** (la collezione predefinita è di tipo **ConcurrentQueue**);
2. gestisce sia buffer a dimensione limitata che illimitata;
3. gestisce automaticamente la sincronizzazione di *producer* e *consumer*, bloccando quando necessario le operazioni di inserimento/estrazione di un elemento.

5.2.1 Estrazione degli elementi mediante ciclo foreach

BlockingCollection semplifica l'implementazione del *consumer*: il **GetConsumingEnumerable()** consente al *consumer* di estrarre gli elementi dalla collezione mediante un semplice **foreach**.

Negli scenari più semplici, il *consumer* si riduce al seguente pattern:

```
BlockingCollection<tipo> bcList = new BlockingCollection<tipo>();
...
foreach (var item in bcList.GetConsumingEnumerable())
{
    // estrae "item" dalla collezione e lo elabora
}
```

La sincronizzazione viene gestita automaticamente, compreso il blocco del *consumer* se la collezione è vuota, e il suo "risveglio" nel momento in cui si rende disponibile almeno un elemento.

5.2.2 Completare il buffer e sbloccare il consumer

Il pattern appena mostrato impedisce al *consumer* di terminare. È possibile risolvere il problema facendo eseguire al *producer* il metodo **CompleteAdding()**. Ciò impedisce nuovi inserimenti e comunica al *consumer*, se la collezione è vuota, che non vi sono più elementi da processare.

> **Cancellazione dell'attesa del *consumer***
>
> Esiste un'altra opzione per terminare il *consumer*. **GetConsumerEnumerable()** accetta un *cancellation token*, e cioè un oggetto che può essere usato per interrompere il ciclo che tiene bloccato il *consumer*.

5.2.3 Esempio d'uso della BlockingCollection<>

Nel seguente esempio un *producer* ha il compito di caricare una lista di URI da un file di testo, mentre un *consumer* ha quello di scaricare i file corrispondenti agli URI.

```
static BlockingCollection<Uri> bcList = new BlockingCollection<Uri>();
private static void TestBC()
{
    Task.Factory.StartNew(Consumer);        // esegue il consumer in un altro thread
    Producer(@"..\..\URILIST.TXT");         // esegue il producer nel main thread
}
```

```
private static void Producer(string fileName)
{
    string[] list = File.ReadAllLines(fileName);
    foreach (var uriStr in list)
    {
        bcList.Add(new Uri(uriStr));
    }
    bcList.CompleteAdding();
}

private static void Consumer()
{
    WebClient cli = new WebClient();
    int fileCcount = 0;
    foreach (var uri in bcList.GetConsumingEnumerable())
    {
        string downloadFileName = string.Format(@"..\..\download {0}.html",
                                                fileCcount++);
        cli.DownloadFile(uri, downloadFileName);
    }
    Console.Write("consumer terminato");
    cli.Dispose();
}
```

Nota bene: viene prima avviato il *consumer* e soltanto dopo il *producer*, cosa che a prima vista appare contro intuitiva (all'inizio non c'è niente da "consumare"). In realtà è del tutto legittimo, poiché il *consumer* si mette semplicemente in attesa che vengano inseriti degli elementi nella collezione.

> ### Avviare il consumer prima del producer
>
> La prassi non solo è legittima ma anche consigliata. Nell'esempio, con il *producer* eseguito nel main-thread, è praticamente obbligatoria. In caso contrario, il *consumer* comincia a processare i dati soltanto dopo che il *producer* ha completamente finito il proprio lavoro, cosa che impedisce di eseguire i due in parallelo.

6 Cancellare operazioni asincrone: *task cancellation*

Una caratteristica importante delle operazioni asincrone è quella di rendere possibile la loro interruzione. Per loro natura, le operazioni sincrone non forniscono questa possibilità, poiché durante la loro esecuzione il codice chiamante è "sospeso".

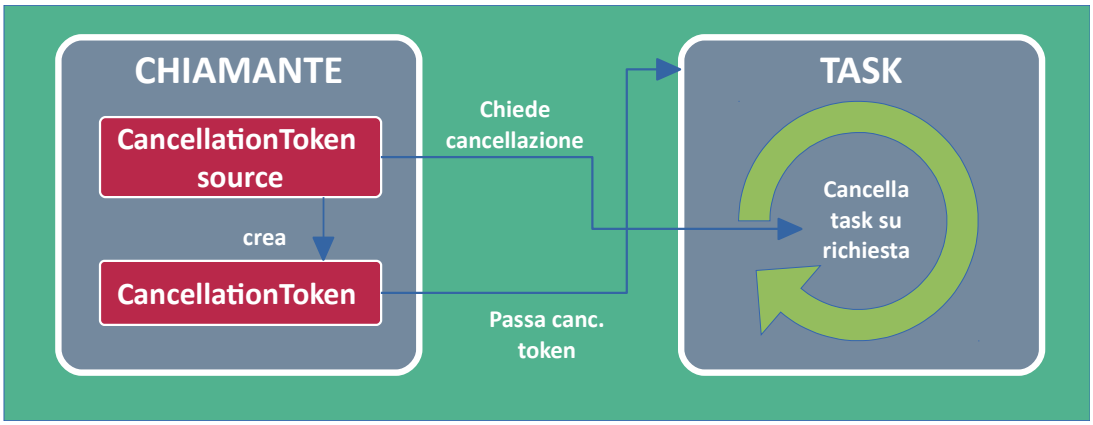
> **Operazioni sincrone cancellabili**
>
> In realtà si possono realizzare operazioni sincrone che, periodicamente, eseguano il codice chiamante, tramite *callback* o eventi. In questo modo si può dare la possibilità al codice chiamante di "comunicare" al codice chiamato che deve terminare.

Di seguito sarà analizzata la tecnica chiamata ***cooperative cancellation***. Pur non essendo l'unica utilizzabile, ha il vantaggio di seguire uno standard ben definito e di utilizzare degli oggetti appositamente implementati per questo scopo.

6.1 "Cancellazione cooperativa" di un task

Si parla di cancellazione cooperativa quando il metodo asincrono è strutturato in modo da poter terminare in risposta a una richiesta esterna. Alla base di questa tecnica sta l'idea che il metodo verifichi periodicamente se c'è una richiesta di cancellazione.

La cancellazione cooperativa viene implementata mediante i tipi **CancellationTokenSouce** e **CancellationToken**.



Il codice chiamante crea un **CancellationTokenSource** e da questo ottiene un **CancellationToken**. Il token viene passato al task, che lo usa per verificare se c'è una richiesta di cancellazione.

Il chiamante può eseguire una richiesta di cancellazione chiamando il metodo **Cancel()** del **CancellationTokenSource**.

6.1.1 Esempio di cancellazione di un task

Il codice crea un task che si limita a simulare un certo lavoro. Il task riceve un **CancellationToken**, attraverso il quale può essere fermato.

```
CancellationTokenSource cancSource = new CancellationTokenSource();

private void btnExecute_Click(object sender, EventArgs e)
{
    CancellationToken cancToken = cancSource.Token;          // crea il cancellation token
    Task.Run(()=> CancellableTask(cancToken), cancToken);  // e lo passa al task
}

private void btnCancel_Click(object sender, EventArgs e)
{
    cancSource.Cancel();            // chiede la cancellazione del task
}

private void CancellableTask(CancellationToken cancToken)
{
    for (int i = 0; i < 200; i++)
    {
        cancToken.ThrowIfCancellationRequested();   //ferma il task se c'è una richiesta
        Thread.Sleep(100);
    }
}
```

Se c'è una richiesta di cancellazione, la chiamata a **ThrowIfCancellationRequested()** solleva un'eccezione di tipo **OperationCanceledException**.

Nota bene: nel metodo **Run()**, il token di cancellazione viene passato sia al metodo sia al task.

> **Passaggio del token di cancellazione**
>
> Se il *cancellation token* viene passato al metodo ma non al task, il meccanismo di arresto funziona ugualmente, ma con un diverso esito. Il metodo viene interrotto come se fosse stata sollevata un'eccezione qualsiasi e lo stato finale del task risulta ***Faulted*** e non ***Canceled***.

6.2 Verificare l'avvenuta cancellazione di un task

Il metodo **ContinueWith()** consente di eseguire un *continuation task* se il task antecedente è stato cancellato.

Il codice seguente visualizza un messaggio se il task viene cancellato, un messaggio diverso se completa normalmente. (In entrambi i casi viene utilizzata una *lambda expression*)

```
private void btnExecute_Click(object sender, EventArgs e)
{
    CancellationToken cancToken = cancSource.Token;
    Task t = Task.Run(()=> CancellableTask(cancToken), cancToken);

    t.ContinueWith((ant) => MessageBox.Show("Cancellato"),
```

```
                    TaskContinuationOptions.OnlyOnCanceled);

    t.ContinueWith((ant) => MessageBox.Show("Completato"),
                    TaskContinuationOptions.OnlyOnRanToCompletion);
}
```

7 Riportare l'avanzamento di un'operazione asincrona

Operazioni che richiedono tempo per il loro completamento dovrebbero riportare il loro stato di avanzamento, in modo che la UI possa fornire un feedback all'utente, ad esempio mediante una *progressive bar*.

La TPL definisce un'interfaccia e una classe in grado di risolvere questo problema: **IProgress<>** e **Progress<>**.

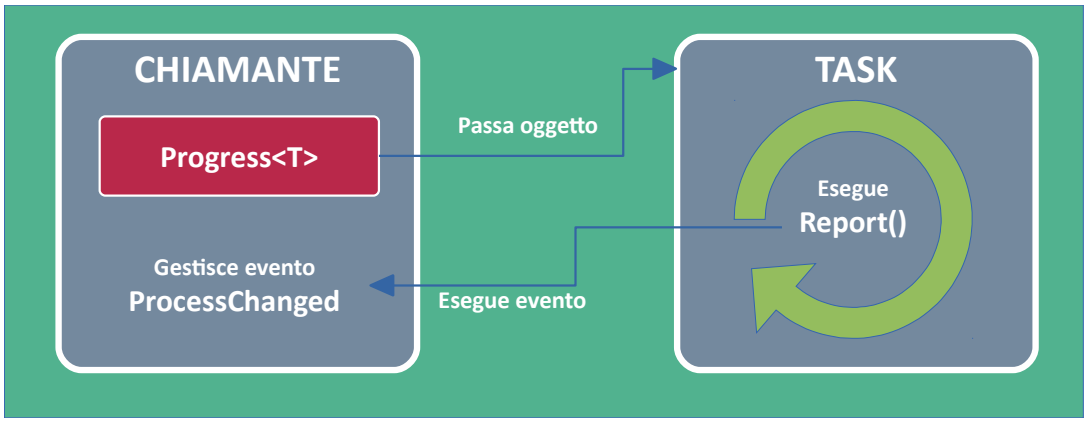
7.1 *Progress reporting*

Il meccanismo di *progress reporting* si basa sulla cooperazione tra chiamante e codice asincrono. Il primo crea un oggetto che il secondo utilizzerà per segnalare l'avanzamento dell'operazione.

L'oggetto deve implementare l'interfaccia **IProgress**, che definisce il metodo generico **Report()**.

7.1.1 Classe Progress

Progress rappresenta un componente già pronto che implementa l'interfaccia **IProgress**. La classe definisce un evento, **ProgressChanged**, attraverso il quale il codice chiamante può ottenere lo stato di avanzamento dell'operazione asincrona. L'evento viene eseguito nel thread chiamante e dunque non solleva problemi di sincronizzazione.



7.2 Esempio di *progress reporting*

Il codice seguente esegue un metodo che si limita a incrementare un contatore. Il metodo riceve un oggetto **IProgress**, attraverso il quale riporta il valore del contatore.

Il codice chiamante crea un oggetto **Progress** prima di creare il task e si sottoscrive all'evento **ProgressChanged**:

```
private void btnExecute_Click(object sender, EventArgs e)
{
    Progress<int> prog = new Progress<int>();
    prog.ProgressChanged += prog_ProgressChanged;
    Task.Run(() => ProgressTask(prog));
}
```

```
// gestisce l'evento sollevato dall'oggetto “prog”
void prog_ProgressChanged(object sender, int value)
{
    lblProgressValue.Text = value.ToString();
}

private void ProgressTask(IProgress<int> prog)
{
    for (int i = 0; i < 20; i++)
    {
        Thread.Sleep(200);
        if (prog != null)
            prog.Report(i);          // indirettamente, esegue " prog_ProgressChanged"
    }
}
```

Nota bene: il *parametro di tipo* dell'interfaccia e della classe **Progress** devono coincidere con il tipo del valore passato al metodo **Report()**.

8 Accesso sincronizzato ai controlli della UI

L'accesso alla UI da un thread diverso da quello principale è un problema generale. Esistono varie tecniche per risolvere questo problema, (**Progress** ne rappresenta un esempio) tra le quali l'uso di uno ***schedulatore di task***.

8.1 Sincronizzare le chiamate alla UI con uno schedulatore di task

Il metodo statico **FromCurrentSyncronizationContext()** della classe **TaskScheduler** ritorna uno oggetto in grado di sincronizzare il codice asincrono con la UI.

Nello scenario seguente un task aggiorna il valore di una label senza sincronizzarsi con il thread della UI:

```
private void btnStart_Click(object sender, EventArgs e)
{
    Task.Run(() => WorkerTask());
}

private void WorkerTask()
{
    int i = 0;
    while(true)
    {
        Thread.Sleep(200);
        lblValue.Text = i++.ToString();     // -> cross-threading!
    }
}
```

Il codice evidenziato produce un accesso *thread-unsafe* alla UI, che può destabilizzare il funzionamento del programma. (Visual Studio produce una *cross-thread exception*)

Una soluzione è quella di eseguire il codice mediante uno schedulatore:

```
TaskScheduler sc;
private void Form1_Load(object sender, EventArgs e)
{
    sc = TaskScheduler.FromCurrentSynchronizationContext();
}

private void btnStart_Click(object sender, EventArgs e)
{
    Task.Run(() => WorkerTask());
}

private void WorkerTask()
{
    int i = 0;
    while(true)
    {
        Thread.Sleep(200);
        Task t = new Task(() => lblValue.Text = i++.ToString());
```

```
        t.Start(sc);        // esegue accesso thread-safe alla UI
    }
}
```

8.1.1 Creazione ed esecuzione del task

Il task creato nell'esempio precedente ha l'unica funzione di eseguire il codice nella UI; pertanto non è necessario memorizzare l'oggetto in una variabile.

Il codice seguente crea il task ed esegue immediatamente il metodo **Start()**, senza assegnarlo prima ad una variabile:

```
private void WorkerTask()
{
    int i = 0;
    while(true)
    {
        Thread.Sleep(200);
        new Task(() => lblValue.Text = i++.ToString()).Start(sc);
    }
}
```

8.1.2 Creazione dello schedulatore

Lo schedulatore deve essere creato *dopo* che la UI è stata caricata. Dunque, è corretto crearlo nell'evento **Form_Load**, ma non direttamente nella dichiarazione:

```
TaskScheduler sc = TaskScheduler.FromCurrentSynchronizationContext(); //-> errore!
private void Form1_Load(object sender, EventArgs e)
{
    //...
}
```

Una volta creato, lo schedulatore può essere utilizzato quante volte si vuole.