BCSCHOOL

Empowering Digital Skills For The Jobs Of The Future

by

BC Soft
consulenza sistemi informativi

AiRED
Artificial Intelligence Ricerca e Sviluppo

Academy .NET

Multithreading

Sommario

- Threading
- Concorrenza e sincronizzazione
- I task
- Programmazione asincrona in C#
- Programmazione parallela
- PLINQ

Threading

Le applicazioni generalmente hanno la necessità di dover gestire ed eseguire più compiti in contemporanea.

Una applicazione lenta e che si blocca ad aspettare il completamento di un'operazione finirebbe presto per stancare l'utente, soprattutto se si tratta di un'applicazione con interfaccia grafica, che al clic su un pulsante avvia un compito particolarmente gravoso e non permette di fare nient'altro nell'attesa.

È possibile superare questa limitazione con l'utilizzo di un pattern di programmazione chiamato **multithreading**

Threading Concorrenza e sincronizzazione

Un **thread** è un percorso di esecuzione di un processo all'interno di una stessa applicazione eseguibile. Le applicazioni possono funzionare correttamente anche con un singolo Thread, quello primario creato dal sistema operativo all'avvio dell'applicazione stessa, cioè quando si entra nel main.

Il thread principale può creare e avviare Thread secondari per eseguire più compiti in parallelo, ossia fornisce la funzionalità di **multithreading**.

Threading Concorrenza e sincronizzazione

Un thread è rappressentato dalla classe **Thread** del namespace System.Threading.

Si può ottenere la istanza del thread corrente per mezzo della proprietà statica di Thread chiamata **CurrentThread**.

```
static void Main()
{
    Thread.CurrentThread.Name=“Primary”;
…
}
```

Threading Concorrenza e sincronizzazione

Per creare un nuovo Thread è necessario creare una istanza della classe **Thread** e avviarla usando il metodo **Start**.

Per indicare il codice che dovrà essere eseguito da parte del nuovo thread si usa un delegate ThreadStart, che punterà al metodo da eseguire all'avvio del thread secondario.

```
public delegate void ThreadStart();
```

Threading Concorrenza e sincronizzazione

Il codice da eseguire all'interno del nuovo Thread sarà di conseguenza un metodo con tipo di ritorno **void** senza parametri di input:

```
public void CodiceSecondario()
{
    Console.WriteLine(Thread.CurrentThread.Name);

    for(int i=0;i<10;i++)
    {
        Console.WriteLine(i);
        Thread.Sleep(100);
    }
}
```

Threading Concorrenza e sincronizzazione

Il metodo statico Sleep() interrompe il thread attuale per il numero di millisecondi indicato. Nell'esempio precedente il thread che esegue il metodo si interromperà per 100 millisecondi ad ogni ciclo del for.

Avvio del thread:

```
Thread th1=new Thread(new Thread(CodiceSecondario));
th1.Start();
```

Threading Concorrenza e sincronizzazione

Con espressione lambda:

```
Thread th3 = new Thread(() =>
            {
                Console.WriteLine(Thread.CurrentThread.Name);
                for (int i = 0; i < 10; i++)
                {
                    Console.WriteLine(i);
                }
            });
th3.Start();
```

Threading Concorrenza e sincronizzazione

Nel momento in cui un Thread viene avviato, la sua propreità **IsAlive** sarà true.

È possibile attendere che un Thread secondario finisca la sua esecuzione prima di continuare quella del thread che lo creato, utilizzando il metodo Join.

```
Thread th=new Thread(CodiceSecondario);
th.Start();
th.Join();
Console.WriteLine("th terminato");
```

Threading Concorrenza e sincronizzazione

Il metodo Join può accettare un parametro come valore di timeout in modo che se il thread non termina pirma di tale valore, l'esecuzione del thread prima rimasto bloccato in attesa riprenderà ugualmente.

Priorità dei Thread

La proprietà **Priority** della classe Thread permette di impostare un livello di priorità assegnabile al thread. Determina qunto tempo di esecuzione il sistema operativo dedicherà ad esso, relativamente ad alti Thread.

Il tipo della proprietà **Priority** è un'enumerazione, **ThreadPriority**, che può essumere i seguenti valori:

Lowest, BelowNormal, Normal, AboveNormal, Highest

Threading Concorrenza e sincronizzazione

I thread secondari, per conportamento predefinito, sono dei Thread di **foreground**. I thread di foreground mantengono viva l'applicazione fino a quando sono tutti terminati.

Per far in modo che i thread creati siano, al contrario, thread da eseguire in background, è necessario impostare esplicitamente la proprietà IsBackground = true.

```
thread.IsBackground=true;
```

Threading Concorrenza e sincronizzazione

Un thread di background viene terminato a qualunque punto sia esso giunto nel suo compito, quando anche il thread principale dell'applicazion termina.

Threading Concorrenza e sincronizzazione

Avviando un nuovo Thread è possibile passare parametri ad esso: un secondo costruttore di Thread prevede l'utilizzo di un delegate **ParameterizedThreadStart**, che consente di passare un Object come argomento del metodo eseguito nel thread, che quindi esso potrà utilizzare al suo interno:

```
string hello = "hello";
Thread th5=new Thread(new ParameterizedThreadStart( (object parameter)=>
{
Console.WriteLine(parameter);
    for(int j=0;j<10;j++)
    {
        Console.Write(".");
        Thread.Sleep(100);
    }
}));
th5.Start(hello);
```

Threading Concorrenza e sincronizzazione

Passagio di più parametri grazie ad espressione lambda:

```
string hello = “hello”;
int n=5;
int interval=1000;

Thread th6=new Thread( ()=> {
    for(int i=0;i<n;i++)
    {
        Console.WriteLine(hello);
        Thread.Sleep(interval);
    }
});
```

Threading Concorrenza e sincronizzazione

In applicazioni multithread è necessario assicurarsi che i dati utilizzati siano protetti dal possibile accesso da parte di thread differenti.

La condizione in cui più thread possono aver letto e modificato una risorsa è detta **race condition**.

Un'altra condizione che può portare al blocco dell'applicazione perché diversi thread restano in attesa uno dell'altro, è detta deadlock

Threading Concorrenza e sincronizzazione

L’istruzione **lock** permette di sincronizzare gli accessi alle risorse condivise da parte di più thread.

La parola chiave lock definisce un blocco di istruzioni , detto sezione critica, che viene sincronizzato in modo che un thread che inizia ad eseguire tale blocco non possa essere interrotto da un altro prima che esso lo completi.

Threading Concorrenza e sincronizzazione

lock è basato su un oggetto di tipo riferimento (in modo da evitare blocchi di sole copie dell'oggetto) che viene utilizzato come **token** da acquisire e bloccare per poter entrare all'interno della sezione critica di istruzioni.

```
lock(obj)
{
//sezione critica
}
```

Un ulteriore Thread non potrà acquisire l'oggetto finchè non verrà rilasciato dal thread che lo utilizza. Il thread verrà bloccato una volta giunti al blocco lock e resterà fermo in attesa che il precedente thread rilasci l'oggetto.

Threading Concorrenza e sincronizzazione

L'istruzione **lock** viene interpretata dal compilatore trasformandola nell'uso della classe **Monitor.**

La classe Monitor può essere controllata esplicitamente dallo sviluppatore come nel seguente esempio:

```
Monitor.Enter(locker);
    try
    {
    //sezione critica
    } finally
    {
    Monitor.Exit(locker);
    }
```

```
lock(locker)
{
//sezione critica
}
```

Pool di thread

La creazione diretta di istanze di Thread non è l'unica opzione per eseguire thread secondari. Tale procedura infatti è particolarmente dispendiosa di risorse. Per ovviare a questo è possibile utilizzare un **pool di thread**, gestito dal CLR.

Un thread pool è un gestore che crea e mantiene un insieme di thread pronti all'uso, riutilizzando quando possibile quelli creati in precedenza.

Task Parallel Library

In .NET 4.0 è stata introdotta la **Task Parallel Library.** Un insieme di tipi e API per facilitare lo sviluppo di applicazioni che fanno uso di attività parallele e concorrenti.

È stata introdotta la classe **Task** che consente di avviare una operazione e di gestire anche eventuali attività aggiuntive da iniziare al completamento di quelle precedenti. È inpoltre possibile farsi restituire un valore di ritorno quando tale operazione è stata completata.

Task Parallel Library

Per avviare un Task in un Thread secondario, servendosi di un pool di thread, si può utilizzare il metodo seguente:

```
Task.Factory.StartNew(() => Console.WriteLine("Hello tasks"));
```

La proprietà Factory restituisce un oggetto **TaskFactory** per la creazione di nuovi task.

StartNew() crea un nuovo task e lo accoda per l'esecuzione.

Task Parallel Library

Da .NET 4.5 è disponibile il metodo run() per semplificare ulteriormente creazione ed esecuzione del Task:

```
Task.Run(() => Console.WriteLine("Hello tasks 2"));
```

Per essere certi che un task sia stato completato, e solo dopo questo evento proseguire con il resto del codice, si utilizza il metodo **Wait**:

```
Task.Run(()=>
{
OperazioneLunga();
});
task.Wait();
```

Task Parallel Library

Il metodo Wait blocca l'esecuzione del codice fino a quando il task in esame non è terminato. Un suo overload permette di indicare un timeout, trascorso il quale l'attesa viene interrotta anche se il task non ha ancora completato il suo compito.

```
Task longTask=Task.Run( ()=>{
Console.WriteLine("Start task…");
Thread.Sleep(3000);
Console.WriteLine("End task");
});
longTask.Wait(2000);
Console.WriteLine("after wait…" );
```

Task Parallel Library

Se il Task deve restituire un valore al suo termine, è possibile utilizzare la versione generica **Task<TResult>** , che specifica come parametro di tipo, il tipo del valore di ritorno.

```
Task<string> task = Task<string>.Run(() =>
{
String url = “http://www.microsoft.com”;
WebClient wc = new WebClient();
return wc.DownloadString(url);
});
string result = task.Result;
```

La lettura della proprietà Result, se il task non è ancora stato completato, bloccherà l’esecuzione del thread corrente fino al suo completamento.

Programmazione Asincrona

Grazie alla **programmazione asincrona** è possibile eseguire più metodi in parallelo utilizzando i **thread**.

Le parole chiave **async** e **wait** permettono di gestire più **thread** in modo semplice.

async e await

- **async** marca un metodo come **asincrono**
- Un **metodo asincrono** può utilizzare nel suo **corpo** la parola chiave **await.**
- Se la parola chiave **async** non è presente, **await** non può essere utilizzato.

async e await

Metodo asincrono:

```
public async void AsynchronousMethod()

{
await Method();
istruzione successiva...
}
```

await permette la sospensione di esecuzione del metodo **async** ed esegue l'operazione marcata da **await** in un **Thread** separato.

Al termine dell'operazione il metodo passa all'istruzione successiva.

Task

L'espressione usata con **await** restituisce un oggetto **awaitable**, generalmente un **Task** o **Task<T>**.

```
public static async Task<string> HelloWorldAsync()
{
    await Task.Delay(1000);
    return "Hello World";
}
```

try – catch async

Da C# 6 è possibile utilizzare l'istruzione **await** all'interno di blocchi try – catch.

```
try
{
    ...
}
catch (Exception ex)
{
    await SaveExceptionAsync(ex);
}
finally
{
    await LogAsync(ex);
}
```

Programmazione parallela

La **Task Parallel Library** oltre a fornire le classi per la gestione di Task, include anche una classe **Parallel** che è un'ulteriore astrazione dei thread, dedicata all'esecuzione di codice in parallelo.

Programmazione parallela

La classe **Parallel** permette di eseguire una stessa operazione in parallelo sugli elementi di un array o di una collezione.

I dati di origine vengono suddivisi in partizioni, in modo che più thread, eseguiti su core del processore differenti, possano occuparsi contemporanemanete di segmenti di dati differenti.

Programmazione parallela

Parallel espone tre metodi statici :

- **Parallel.For** : ciclo for in parallelo
- **Parallel.ForEach :** ciclo foreach in parallelo
- **Parallel.Invoke :** esegue un array di delegate in parallelo

Programmazione parallela

Parallel.For eseguirà un ciclo for in parallelo su thread differenti. Il numero di thread e la loro creazione sarà automaticamente ottimizzata per ottenere il massimo parallelismo possibile.

Il metodo ha 12 overload, il più frequente è il seguente:

```
public static ParallelLoopResult For(
int fromInclusive,
int toExclusive,
Action<int> body
)
```

Programmazione parallela

Il primo e il secondo parametro rappresentano l'indice di partenza (incluso) e quello finale (escluso) delle iterazioni da eseguire, mentre l'ultimo è un delegate di tipo Action<int> che rappresenta l'azione da iterare.

Stampa dei numeri da 1 a 10:

```
Parallel.For(1, 11, i => Console.WriteLine("Parallel.For {0}", i));
```

L'ordine con cui saranno eseguite le iterazioni non è predicibile a priori.

Programmazione parallela

Un altro interessante overload imposta opzioni per monitorare lo stato del ciclo:

```
Parallel.For(0, 50, (i, loopState) =>
{
Console.WriteLine(i);
if (i > 10)
{
loopState.Break();
}
});
```

L'oggetto **ParallelLoopState** viene utilizzato per fermare il ciclo prematuramente. Il metodo **Break** informa il ciclo che le iterazioni successive a quella corrente devono essere fermate.

Programmazione parallela

È importante ricordare che l'ordine di esecuzione delle iterazioni non è sequenziale, quindi prima di giungere alla numero 10 potrebbero anche essere state avviate iterazioni con i superiore a 10.

Il metodo **Stop** invece forza tutti i thread avviati a terminare la loro esecuzione il prima possibile.

Programmazione parallela

Il metodo **Parallel.ForEach** è l'analogo parallelizzato del costrutto foreach, quindi permette di eseguire un ciclo sugli elementi di una Ienumerable, ma in modo asincrono.

```
List<string> list=new List<string>(){“questa”, “è”, “una”, “frase”, “di”, “poche”,
“parole”};
Parallel.ForEach(list, word => Console.WriteLine(“{0}: {1}”, word, word.Length));
```

Programmazione parallela

Parallel.Invoke permette di eseguire in parallelo un array di delegate Action.

I metodi da eseguire in parallelo possono essere contenuti in un vero arrray, oppure passati come numero variabile di argomenti
Supponiamo i metodi M1, M2, M3:

```
Parallel.Invoke(M1, M2, M3);
```

Se contenuti in un array:

```
Action[] methods = new Action[] { M1, M2, M3 };
Parallel.Invoke(methods);
```

PLINQ

PLINQ è una implementazione in parallelo di LINQ to Objects.

Tutti i metodi di estensione di LINQ sono disponibili in versione parallela e inoltre altri metodi implementano operatori aggiuntivi.

Le query PLINQ sono molto simili alle query LINQ to Objects non parallele. PLINQ tenta di sfruttare al massimo tutti i processori del sistema.

Come avviene con la classe Parallel, viene eseguito il partizionamento dell'origine dati in segmenti e quindi la query può essere eseguita su ogni segmento in thread distinti.

PLINQ

Il funzionamento di PLINQ e possibile grazie alla classe **ParallelEnumerable** e i suoi metodi di estensione

Il metodo **AsParallel()** è invocabile da qualsiasi collezione per ottenerne una versione parallela della stessa, rappresentata dal tipo **ParallelQuery<T>**

```
List<string> stringhe = new List<string>() {"str1", "str2", "str3"};
ParallelQuery<string> parallelStringhe = parole.AsParallel();

parallelStringhe.ForAll((stringa) => Console.WriteLine(stringa));
```

Domande & approfondimenti

Academy .NET