.NET Profiler

Un **profiler .NET** è uno strumento che consente di analizzare le prestazioni di un'applicazione .NET, identificando *colli di bottiglia* e problemi di utilizzo delle risorse [1]. Il profiler raccoglie dati su **CPU, memoria e chiamate di funzioni**, offrendo una vista dettagliata del comportamento runtime. Ad esempio, Visual Studio include un profiler integrato per misurare il tempo di CPU, l'uso di memoria e le allocazioni di oggetti .NET [2]. Queste informazioni aiutano lo sviluppatore C# a capire quali metodi consumano più risorse e dove ottimizzare il codice. Nel caso non si disponga di un profiler grafico, si possono usare classi come `Stopwatch` o chiamate a `GC.GetTotalMemory` per effettuare un *profiling manuale* direttamente nel codice, misurando il tempo di esecuzione e la memoria utilizzata da specifiche operazioni.

```
using System;
using System.Diagnostics;

 // Esempio di profiling manuale: misurazione tempo di esecuzione
Stopwatch sw = Stopwatch.StartNew();
long somma = 0;
for (int i = 0; i < 10000000; i++)
{
    somma += i;  // Operazione CPU-intensiva da profilare
}
sw.Stop();
Console.WriteLine($"Tempo impiegato: {sw.ElapsedMilliseconds}
ms");  // Tempo in millisecondi

// Misurazione dell'utilizzo di memoria
long memoriaPrima = GC.GetTotalMemory(false);
byte[] data = new byte[10 * 1024 * 1024];       // Allocazione di ~10 MB
long memoriaDopo = GC.GetTotalMemory(false);
Console.WriteLine($"Byte allocati: {memoriaDopo - memoriaPrima}");   //
Differenza di memoria allocata

// Pulizia (rilascia la memoria allocata dall'array)
data = null;
GC.Collect();
```

Gestione della Memoria

In .NET la memoria è gestita automaticamente attraverso il *managed heap* e lo *stack*. **Lo stack** viene utilizzato per variabili locali e parametri: è una struttura LIFO che alloca memoria in modo molto rapido e la libera automaticamente quando si esce dal metodo [3]. **L'heap**, invece, è l'area da cui proviene la memoria per gli oggetti allocati con `new` (tipi reference). A differenza dello stack, l'heap ha una gestione più complessa ed è qui che interviene il *Garbage Collector* [4]. In termini pratici, i tipi valore (*struct*, ad esempio un `int`) vivono tipicamente sullo stack (o all'interno di oggetti), mentre i tipi

reference (*class*, ad esempio una `string` o un oggetto personalizzato) sono allocati sull'heap. C# si occupa di liberare la memoria degli oggetti non più utilizzati, ma è importante comprendere queste differenze per scrivere codice efficiente ed evitare comportamenti inattesi (ad esempio, aliasing di riferimento vs copia di valori). Nell'esempio seguente vediamo come un tipo valore viene copiato completamente (occupando due spazi di memoria distinti nello stack), mentre un tipo reference copia solo il riferimento (puntando allo stesso oggetto su heap):

```
using System;

// Definizione di un tipo valore (struct) e di un tipo reference (class)
struct Punto { public int X; public int Y; }
class PuntoClass { public int X; public int Y; }

// Esempio di allocazione e copia su stack vs heap
Punto a = new Punto { X = 5, Y = 5 };       // `a` è allocato nello stack
Punto b = a;                                // `b` è una copia indipendente
di `a`
b.X = 10;
Console.WriteLine(a.X);  // Output: 5  (il valore originale non cambia, copia
per valore)

PuntoClass c1 = new PuntoClass { X = 5, Y = 5 };  // Oggetto `PuntoClass`
allocato nell'heap
PuntoClass c2 = c1;                               // `c2` copia il
riferimento, punta allo *stesso* oggetto heap
c2.X = 10;
Console.WriteLine(c1.X);  // Output: 10 (c1 e c2 riferiscono lo stesso
oggetto, modificato tramite c2)
```

Garbage Collection

Il **Garbage Collector (GC)** di .NET gestisce automaticamente la memoria sull'heap, **individuando e liberando gli oggetti non più referenziati** dall'applicazione [5]. Ciò significa che lo sviluppatore non deve esplicitamente deallocare la memoria (come invece accade in C/C++); il runtime rileva quando un oggetto non ha più riferimenti attivi e recupera la memoria occupata, prevenendo *memory leak*. Per efficienza, .NET adotta una strategia **generazionale**: gli oggetti appena creati appartengono alla *Generazione 0* (raccolta frequente per gli oggetti a breve vita), quelli che sopravvivono a più cicli di GC vengono promossi in *Gen 1* e infine in *Gen 2* per gli oggetti longevi [6]. Questo approccio riduce il più possibile le pause di raccolta concentrandosi spesso sugli oggetti più giovani. In casi normali, il GC decide autonomamente *quando* effettuare una raccolta in base alle esigenze di memoria. È possibile comunque forzare la raccolta manualmente con `GC.Collect()`, ad esempio per liberare memoria dopo un grande carico allocativo, ma farlo è **sconsigliato** salvo necessità specifiche [7], poiché interrompe l'esecuzione e può peggiorare le prestazioni se usato impropriamente. Nel codice seguente creiamo e poi eliminiamo un grande oggetto per illustrare l'azione del GC:

```
using System;

Console.WriteLine($"Memoria iniziale: {GC.GetTotalMemory(false)} byte");
byte[] buffer = new byte[10 * 1024 * 1024];      // Allochiamo ~10 MB
```

```
sull'heap
Console.WriteLine($"Dopo allocazione: {GC.GetTotalMemory(false)} byte");
buffer = null;                                  // L'oggetto non è più
referenziato
GC.Collect();                                   // Forziamo il Garbage
Collector a eseguire la raccolta
Console.WriteLine($"Dopo GC.Collect: {GC.GetTotalMemory(false)} byte");
```

Commento: nell'output, la memoria dopo il GC dovrebbe diminuire considerevolmente rispetto a dopo l'allocazione (può non tornare esattamente al valore iniziale a causa di overhead interni). Il `GC.Collect()` forza la raccolta di **tutte** le generazioni e libera il buffer non più utilizzato. In generale, però, si lascia al runtime la decisione di quando fare pulizia, concentrandosi come sviluppatori sull'allocare oggetti solo quando serve e liberare riferimenti (es. impostandoli a null o usandoli in scope locali) quando non più necessari.

Ottimizzazione del Codice e JIT

Per ottenere codice C# più **performante** è importante sia scrivere in modo efficiente sia comprendere il ruolo del *Just-In-Time compiler (JIT)*. .NET non esegue il codice IL (Intermediate Language) direttamente, ma lo **compila a runtime in codice macchina nativo** metodo per metodo, alla prima esecuzione di ciascun metodo [8]. Questo implica che la **prima chiamata** a un metodo include un overhead di compilazione JIT, mentre le chiamate successive risultano più veloci perché il codice nativo viene cacheato in memoria e riutilizzato [9]. Ad esempio, picchi di latenza all'avvio di un'applicazione sono spesso dovuti alla compilazione JIT iniziale dei metodi più usati. Per migliorare le prestazioni, è consigliabile compilare il progetto in modalità **Release** (abilitando le ottimizzazioni del JIT, come inlining e loop unrolling, che in Debug sono disattivate) e, se necessario, "scaldare" in anticipo i percorsi critici chiamando alcuni metodi durante l'inizializzazione (per pagarne il costo JIT prima che servano risultati rapidi). Dal lato del codice, l'**ottimizzazione** consiste nell'eliminare operazioni inutili e ridurre il lavoro del runtime: ad esempio, evitare allocazioni superflue (specialmente *dentro loop*), che aumentano la pressione sul GC e possono rallentare l'applicazione [10]. Inoltre, scegliere strutture dati e algoritmi efficienti (es. usare una `Dictionary` per ricerche veloci anziché scansionare una lista) aiuta il JIT e il runtime a fare meno lavoro.

Nel seguente esempio, misuriamo il tempo di esecuzione di un metodo costoso alla **prima** invocazione vs. le successive, evidenziando l'effetto del JIT. La prima chiamata include sia l'esecuzione del metodo sia il tempo impiegato dal JIT per compilare il metodo, mentre la seconda chiamata esegue direttamente il codice nativo già compilato:

```
using System;
using System.Diagnostics;

void EseguiCalcolo()
{
    // Simula un calcolo intensivo
    long somma = 0;
    for (int i = 0; i < 1000000; i++)
    {
        somma += i;
    }
```

```
}

var sw = Stopwatch.StartNew();
EseguiCalcolo();
sw.Stop();
Console.WriteLine($"Prima chiamata: {sw.ElapsedMilliseconds} ms (include
overhead JIT)");
sw.Restart();
EseguiCalcolo();
sw.Stop();
Console.WriteLine($"Seconda chiamata: {sw.ElapsedMilliseconds} ms (codice già
compilato)");
```

Nella prima esecuzione di `EseguiCalcolo()`, il metodo viene compilato JIT e poi eseguito, portando a un tempo leggermente maggiore. *Nella seconda esecuzione*, il JIT non è più coinvolto per quel metodo, e si nota un tempo inferiore. In scenari reali, questo significa che i metodi più invocati beneficiano nel tempo della compilazione JIT. Inoltre, il JIT in .NET Core/5+ adotta una **Tiered Compilation** (compilazione a livelli): inizialmente compila velocemente con ottimizzazioni minime per ridurre la latenza, poi se un metodo viene chiamato molte volte, il runtime può ricompilarlo con ottimizzazioni più aggressive. Questi dettagli del JIT sono gestiti automaticamente dal runtime, ma sapere che esistono aiuta a interpretare comportamenti prestazionali (ad esempio, perché la prima iterazione di un algoritmo può essere più lenta delle successive). In conclusione, per ottimizzare il codice C# conviene: (1) scrivere codice pulito ed efficiente (evitando operazioni ridondanti e allocazioni inutili), (2) sfruttare le ottimizzazioni del compilatore/JIT con le build di Release, e (3) utilizzare profiler e strumenti di benchmarking per misurare i miglioramenti e individuare i veri colli di bottiglia. [11] [12]

---

[1] Strumenti di profilatura in .NET - .NET | Microsoft Learn
https://learn.microsoft.com/it-it/dotnet/core/diagnostics/profilers

[2] Profiling del Codice in C# - Documentazione csharp | Codegrind
https://codegrind.it/documentazione/csharp/profiling-codice?srsltid=AfmBOorvMzV2T98kgDY8i2zkkYuwyBkHR1RhmlKjiSc2cjx_pGEBOWpo

[3] [4] [6] [7] Gestione della Memoria in C#: Guida Completa - Documentazione csharp | Codegrind
https://codegrind.it/documentazione/csharp/memory-management?srsltid=AfmBOoq9tcCn9kM1FU28zqPmJdAgGtr0fidHddpdUnyD4AQ9POjgA7_A

[5] [12] Garbage Collection in C#: Guida Completa - Documentazione csharp | Codegrind
https://codegrind.it/documentazione/csharp/garbage-collection

[8] [9] Processo di esecuzione gestita - .NET | Microsoft Learn
https://learn.microsoft.com/it-it/dotnet/standard/managed-execution-process

[10] Ottimizzazione del Codice in C# - Documentazione csharp | Codegrind
https://www.codegrind.it/documentazione/csharp/ottimizzazione-codice

[11] Why C# Programs Are Slower When Run for the First Time | by Viktor Ponamarev | Medium
https://medium.com/@vikpoca/why-c-programs-are-slower-when-run-for-the-first-time-1970311a67d2