EBook Gratuito

APPRENDIMENTO async-await

Free unaffiliated eBook created from
Stack Overflow contributors.

#async-await

Sommario

Di

Capitolo 1: Iniziare con async-await

Osservazioni

`async-await` consente l'esecuzione asincrona (significa non bloccante, parallela) del codice. Ti aiuta a mantenere la tua interfaccia utente reattiva in ogni momento, mentre esegui operazioni potenzialmente lunghe in background.

È particolarmente utile per operazioni di I / O (come il download da un server o la lettura di un file dall'HDD), ma può anche essere utilizzato per eseguire calcoli intensivi della CPU senza congelare l'applicazione.

Examples

uso semplice

Tre cose sono necessarie per usare `async-await` :

- L'oggetto `Task` : questo oggetto viene restituito da un metodo che viene eseguito in modo asincrono. Ti permette di controllare l'esecuzione del metodo.
- La parola chiave `await` : "attende" un `Task` . Inserisci questa parola chiave prima che l' `Task` attenda in modo asincrono
- La parola chiave `async` : tutti i metodi che utilizzano la parola chiave `await` devono essere contrassegnati come `async`

Un piccolo esempio che dimostra l'utilizzo di queste parole chiave

```
public async Task DoStuffAsync()
{
    var result = await DownloadFromWebpageAsync(); //calls method and waits till execution
finished
    var task = WriteTextAsync(@"temp.txt", result); //starts saving the string to a file,
continues execution right await
    Debug.Write("this is executed parallel with WriteTextAsync!"); //executed parallel with
WriteTextAsync!
    await task; //wait for WriteTextAsync to finish execution
}

private async Task<string> DownloadFromWebpageAsync()
{
    using (var client = new WebClient())
    {
        return await client.DownloadStringTaskAsync(new Uri("http://stackoverflow.com"));
    }
}

private async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);
```

```
    using (FileStream sourceStream = new FileStream(filePath, FileMode.Append))
    {
        await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
    }
}
```

Alcune cose da notare:

- È possibile specificare un valore di ritorno da un'operazione asincrona con `Task<string>` o simile. La parola chiave `await` attende che l'esecuzione del metodo finisca e restituisca la `string` .
- l'oggetto `Task` contiene semplicemente lo stato dell'esecuzione del metodo, può essere utilizzato come qualsiasi altra variabile.
- se viene lanciata un'eccezione (ad esempio da `WebClient`), viene visualizzata la prima volta che viene utilizzata la parola chiave `await` (in questo esempio alla riga `var result (...)`)
- Si consiglia di assegnare un nome ai metodi che restituiscono l'oggetto `Task` come `MethodNameAsync`

eseguire il codice sincrono asincrono

Se si desidera eseguire il codice sincrono asincrono (ad esempio calcoli estesi della CPU), è possibile utilizzare `Task.Run(() => {})` .

```
public async Task DoStuffAsync()
{
    await DoCpuBoundWorkAsync();
}


private async Task DoCpuBoundWorkAsync()
{
    await Task.Run(() =>
    {
        for (long i = 0; i < Int32.MaxValue; i++)
        {
            i = i ^ 2;
        }
    });
}
```

l'oggetto Task

L'oggetto `Task` è un oggetto come qualsiasi altro se togli le parole chiave `async-await` .

Considera questo esempio:

```
public async Task DoStuffAsync()
{
    await WaitAsync();
    await WaitDirectlyAsync();
}

private async Task WaitAsync()
```

```
{
    await Task.Delay(1000);
}

private Task WaitDirectlyAsync()
{
    return Task.Delay(1000);
}
```

La differenza tra questi due metodi è semplice:

- `WaitAsync` attende che `Task.Delay` , quindi restituisce.
- `WaitDirectlyAsync` non attende e restituisce immediatamente l'oggetto `Task` .

Ogni volta che si utilizza la parola chiave `await` , il compilatore genera il codice per gestirlo (e l'oggetto `Task` che attende).

- Alla chiamata `await WaitAsync()` questo accade due volte: una volta nel metodo chiamante e una volta nel metodo stesso.
- Alla chiamata `await WaitDirectlyAsync` questo accade solo una volta (nel metodo di chiamata). Si `await WaitAsync()` quindi una piccola accelerazione rispetto `await WaitAsync()` .

Attenzione con le `exceptions` : le `Exceptions` peggioreranno la prima volta che un `Task` è `await` . Esempio:

```
private async Task WaitAsync()
{
    try
    {
        await Task.Delay(1000);
    }
    catch (Exception ex)
    {
        //this might execute
        throw;
    }
}

private Task WaitDirectlyAsync()
{
    try
    {
        return Task.Delay(1000);
    }
    catch (Exception ex)
    {
        //this code will never execute!
        throw;
    }
}
```

vuoto asincrono

È possibile utilizzare `void` (anziché `Task`) come tipo di ritorno di un metodo asincrono. Ciò si tradurrà in un'azione "fire-and-forget":

```
public void DoStuff()
{
    FireAndForgetAsync();
}

private async void FireAndForgetAsync()
{
    await Task.Delay(1000);
    throw new Exception(); //will be swallowed
}
```

Poiché stai tornando a `void` , non puoi `await FireAndForgetAsync` . Non sarai in grado di sapere quando il metodo finisce, e tutte le eccezioni sollevate all'interno del metodo del `async void` saranno inghiottite.

Leggi Iniziare con async-await online: https://riptutorial.com/it/async-await/topic/5658/iniziare-con-async-await

Capitolo 2: Migliori pratiche

Examples

Evita il vuoto asincrono

- L'unico posto in cui è possibile utilizzare in modo sicuro il `async void` è nei gestori di eventi. Considera il seguente codice:

```
private async Task<bool> SomeFuncAsync() {
  ...
  await ...
}
public void button1_Click(object sender,  EventArgs e) {
  var result = SomeFuncAsync().Result;
  SomeOtherFunc();
}
```

  Una volta completata la chiamata `async` , attende che `SynchronizationContext` diventi disponibile. Tuttavia, il gestore eventi mantiene il `SynchronizationContext` mentre è in attesa del `SomeFuncAsync` metodo `SomeFuncAsync` ; causando così un'attesa circolare (stallo).

  Per risolvere questo problema, è necessario modificare il gestore eventi per:

```
public async void button1_Click(object sender,  EventArgs e) {
  var result = await SomeFuncAsync();
  SomeOtherFunc();
}
```

- Qualsiasi eccezione eliminata da un metodo di `async void` verrà sollevata direttamente sul `SynchronizationContext` che era attivo all'avvio del metodo del `async void` .

```
private async void SomeFuncAsync() {
  throw new InvalidOperationException();
}
public void SomeOtherFunc() {
  try {
    SomeFuncAsync();
  }
  catch (Exception ex) {
    Console.WriteLine(ex);
    throw;
  }
}
```

  l'eccezione non viene mai catturata dal blocco catch in `SomeOtherFunc` .

- `async void` metodi `async void` non forniscono un modo semplice per notificare il codice chiamante che hanno completato

- `async void`

metodi di `async void` sono difficili da testare. Il supporto di test asincroni MST funziona solo con metodi `async` che restituiscono `Task` o `Task<T>` .

Leggi Migliori pratiche online: https://riptutorial.com/it/async-await/topic/9055/migliori-pratiche

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con async-await	Community, Florian Moser, Kirill Mehtiev
2	Migliori pratiche	user2321864