EBook Gratis

APRENDIZAJE async-await

Free unaffiliated eBook created from
Stack Overflow contributors.

#async-await

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: async-await

It is an unofficial and free async-await ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official async-await.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con async-await

Observaciones

`async-await` permite la ejecución de código asíncrono (significa no bloqueo, paralelo). Ayuda a mantener su interfaz de usuario receptiva en todo momento, mientras ejecuta operaciones potencialmente largas en segundo plano.

Es especialmente útil para operaciones de E / S (como descargar desde un servidor o leer un archivo desde el disco duro), pero también se puede usar para ejecutar cálculos intensivos de CPU sin congelar la aplicación.

Examples

uso simple

Se necesitan tres cosas para usar `async-await` :

- El objeto `Task` : este objeto se devuelve mediante un método que se ejecuta de forma asíncrona. Te permite controlar la ejecución del método.
- La palabra clave `await` : "espera" una `Task` . Coloque esta palabra clave antes de la `Task` para esperar asincrónicamente a que finalice
- La palabra clave `async` : todos los métodos que utilizan la palabra clave `await` deben marcarse como `async`

Un pequeño ejemplo que demuestra el uso de estas palabras clave.

```
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);
    }
}
```

Algunas cosas a tener en cuenta:

- Puede especificar un valor de retorno de una operación asíncrona con la `Task<string>` o similar. El `await` palabra clave espera hasta que la ejecución de los acabados de método y devuelve la `string` .
- El objeto de `Task` simplemente contiene el estado de la ejecución del método, se puede usar como cualquier otra variable.
- si se lanza una excepción (por ejemplo, por el `WebClient`), la primera vez que se utiliza la palabra clave `await` (en este ejemplo, en la línea `var result (...)`), se `var result (...)` .
- Se recomienda nombrar los métodos que devuelven el objeto `Task` como `MethodNameAsync`

Ejecutar código síncrono asíncrono

Si desea ejecutar un código síncrono asíncrono (por ejemplo, cálculos extensos de la CPU), puede usar `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;
        }
    });
}
```

el objeto de tarea

El objeto `Task` es un objeto como cualquier otro si quita las palabras clave `async-await` .

Considera este ejemplo:

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

private async Task WaitAsync()
{
```

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

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

La diferencia entre estos dos métodos es simple:

- `WaitAsync` espera a que finalice `Task.Delay` y luego regresa.
- `WaitDirectlyAsync` no espera, y simplemente devuelve el objeto `Task` instante.

Cada vez que utiliza la palabra clave `await` , el compilador genera un código para tratar con ella (y el objeto `Task` que espera).

- Al llamar, `await WaitAsync()` esto sucede dos veces: una vez en el método de llamada y una vez en el método mismo.
- En la llamada, `await WaitDirectlyAsync` esto sucede solo una vez (en el método de llamada). Por lo tanto, archivaría un poco de aceleración en comparación con `await WaitAsync()` .

**Cuidado con las `exceptions` **: las `Exceptions` surgirán la primera vez que se `await` una `Task` . Ejemplo:

```
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;
    }
}
```

vacío asíncrono

Puede usar `void` (en lugar de `Task`) como un tipo de retorno de un método asíncrono. Esto resultará en una acción de "disparar y olvidar":

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

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

Cuando regresa `void` , no puede `await FireAndForgetAsync` . No podrá saber cuándo finaliza el método, y cualquier excepción provocada dentro del método de `async void` se tragará.

Lea Empezando con async-await en línea: https://riptutorial.com/es/async-await/topic/5658/empezando-con-async-await

Capítulo 2: Mejores prácticas

Examples

Evitar async void

- El único lugar donde puede usar `async void` forma segura es en los controladores de eventos. Considere el siguiente código:

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

  Una vez que se completa la llamada `async` , espera a que `SynchronizationContext` esté disponible. Sin embargo, el controlador de eventos mantiene el `SynchronizationContext` mientras espera que se `SomeFuncAsync` método `SomeFuncAsync` ; causando así una espera circular (punto muerto).

  Para arreglar esto necesitamos modificar el controlador de eventos para:

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

- Cualquier excepción desechada de un método de `async void` se generará directamente en el `SynchronizationContext` que estaba activo cuando se inició el método de `async void` .

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

  la excepción nunca es capturada por el bloque catch en `SomeOtherFunc` .

- `async void` métodos de `async void` no proporcionan una manera fácil de notificar al código de llamada que han completado

- `async void` métodos de `async void` son difíciles de probar. El soporte de pruebas asíncronas de MSTest solo funciona para métodos `async` que devuelven `Task` o `Task<T>` .

Lea Mejores prácticas en línea: https://riptutorial.com/es/async-await/topic/9055/mejores-practicas

Creditos

S. No	Capítulos	Contributors
1	Empezando con async-await	Community, Florian Moser, Kirill Mehtiev
2	Mejores prácticas	user2321864