Asincronismo

Introducción

La programación multi-thread implica el reparto de tareas (thread) de una misma aplicación, de forma que se realicen de manera independiente unas de otras. De este modo se consigue un uso óptimo del tiempo del procesador. Las tareas no se realizan en realidad de forma paralela. El procesador asigna un tiempo de procesamiento a cada tarea en función de su importancia. Este cambio de contexto implica que el procesador memoriza la pila de la tarea actual antes de restaurar la de la tarea a la que le da paso. C# 5 introduce de nuevo palabras clave (`await` y `async`) para facilitar el desarrollo asíncrono. El desarrollo síncrono implica que una función que se invoca bloquea la ejecución del programa hasta que éste termina. Cuando una función se invoca de forma asíncrona, la ejecución del programa principal continúa. Por tanto, existe una noción de ejecución en paralelo y de concurrencia, como ocurre con la programación multi-thread.

Funciones asíncronas

La versión 5 del lenguaje C# introdujo las nuevas palabras clave `async` y `await` que sirven para facilitar la implementación de funciones asíncronas de manera que se parezca a la implementación de un método síncrono.
A continuación, se muestra el ejemplo de una función que necesita una cantidad de tiempo importante para ejecutarse:

```
public static double TimeConsumingFunction()
{
        double x = 1;
        for (int i = 1; i < 100000000; i++)
        {
                x += Math.Tan(x) / i;
        }
        return x;
}
```

Esta función bloquea la ejecución del programa hasta que termina y devuelve su resultado a la llamada:

```
Console.WriteLine("Ejecución de una función larga");

DateTime inicio = DateTime.Now;
double resultado = TimeConsumingFunction();
DateTime fin = DateTime.Now;

Console.Write("Resultado: ");
Console.WriteLine(resultado);

Console.Write("Tiempo de ejecución: ");
```

```
Console.WriteLine(TimeSpan.FromTicks(fin.Ticks - inicio.Ticks).
TotalSeconds);
```

```
Ejecución de una función larga
Resultado: -24,4605663988507
Tiempo de ejecución: 5,3490973
```

1. Task y Task<TResult>

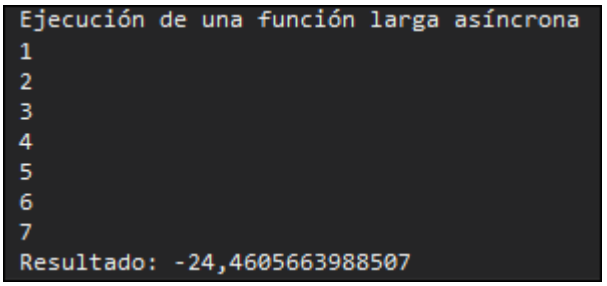
El espacio de nombres `System.Threading.Tasks` define una clase `Task` y una clase `Task<TResult>` que representan un método que se completará más adelante. Puede obtener un objeto de tipo `Task<TResult>` como retorno de la ejecución del método estático `Run` de la clase `Task`:

```
public static Task<double> TimeConsumingFunctionAsync()
{
         return Task.Run(() => TimeConsumingFunction());
}
```

Este método es asíncrono, ya que vuelve al método que lo invoca inmediatamente mientras se ejecuta. La clase `TaskAwaiter` del espacio de nombres `System.Runtime.CompilerServices` permite especificar las acciones que hay que realizar cuando la tarea se termina. La clase `Task<TResult>` expone el método `GetAwaiter` que devuelve una instancia de objeto de tipo `TaskAwaiter`.
Es preciso relacionar un delegado con el método `OnCompleted` para indicar las acciones que hay que realizar:

```
Console.WriteLine("Ejecución de una función larga asíncrona");

int i = 0;

Task<double> task = TimeConsumingFunctionAsync();

TaskAwaiter<double> awaiter = task.GetAwaiter();
awaiter.OnCompleted(() =>
        {
            i = 20;
            double resultado2 = awaiter.GetResult();
            Console.Write("Resultado: ");
            Console.WriteLine(resultado2);
        });

for (i = 1; i < 15; i++)
{
        Console.WriteLine(i);
        Thread.Sleep(1000);
}
```

Durante la ejecución del cálculo más largo, el programa continúa su ejecución de manera normal, entrando en el bucle `for`. Cuando el método asíncrono se termina, el delegado del método `OnCompleted` se ejecuta. Dentro de este delegado, es posible recuperar el resultado de la función mediante el método `GetResult` del objeto de tipo `TaskAwaiter`. Observe que también es posible modificar la variable `i` en el delegado, sin generar una excepción. Esto tiene como efecto que el programa salga del bucle `for`:



La clase `Task` representa la versión no genérica de la clase `Task<TResult>`. Tiene las mismas funcionalidades, excepto que no proporciona ningún resultado a través del método `GetResult`.

2. async y await

Las palabras clave `await` y `async` aportan una mayor simplicidad en la creación de funciones asíncronas. La palabra clave `async` es un modificador de método que indica al compilador que debe tratar `await` como una palabra clave, en lugar de como un identificador. De esta manera se evita cualquier ambigüedad que se podría producir al migrar una aplicación que contuviera `await` como identificador. La palabra clave `async` solo se puede usar en métodos, expresiones lambda y funciones que devuelven un objeto `Task` o `Task<TResult>`. La palabra clave `async` no tiene ningún efecto sobre la declaración de un método. Solo afecta a los elementos ubicados dentro del método.

De esta manera es posible invocar al método `TimeConsumingFunction Async` de la siguiente manera:

```
double resultado = await TimeConsumingFunctionAsync();
```

Para que el compilador no devuelva ninguna excepción, hay que encapsular esta instrucción en un método marcado con la palabra clave `async`:

```
public static async void TimeConsumingFunctionAsync2()
{
                              double    resultado       =    await
TimeConsumingFunctionAsync();
        Console.Write("Resultado: ");
        Console.WriteLine(resultado);
}
```

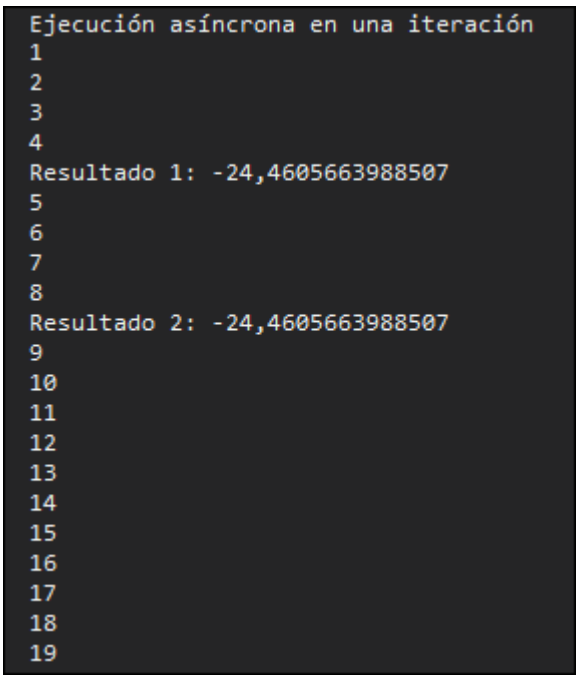
Cuando el proceso de ejecución de la aplicación encuentra la palabra clave `await`, la ejecución devuelve la llamada de la misma manera que con la palabra clave `yield return` en una iteración. Antes de devolver el control al método que invoca, se adjunta un anchor al método cuya llamada está marcada con la palabra clave `await` para que, cuando termine, la ejecución del método marcado con la palabra clave `async` pueda recomenzar y terminar.

Las expresiones `await` se pueden insertar en el código en cualquier lugar, excepto en un bloque `catch` o `finally`, en una expresión con la palabra clave `lock`, en un contexto `unsafe` o en un punto de entrada de la aplicación como un método `Main`.
Por ejemplo, puede utilizar métodos asíncronos dentro de una iteración de la siguiente manera:

```
public static async void Iteration()
{
        for (int i = 0; i < 2; i++)
        {
                                double resultado = await
TimeConsumingFunctionAsync();
            Console.Write("Resultado " + (i + 1) + ": ");
            Console.WriteLine(resultado);
        }
}
```

Llamando de esta manera:

```
Console.WriteLine("Ejecución asíncrona en una iteración");
Iteration();

for (int j = 1; j < 20; j++)
{
        Console.WriteLine(j);
        Thread.Sleep(1000);
}
```

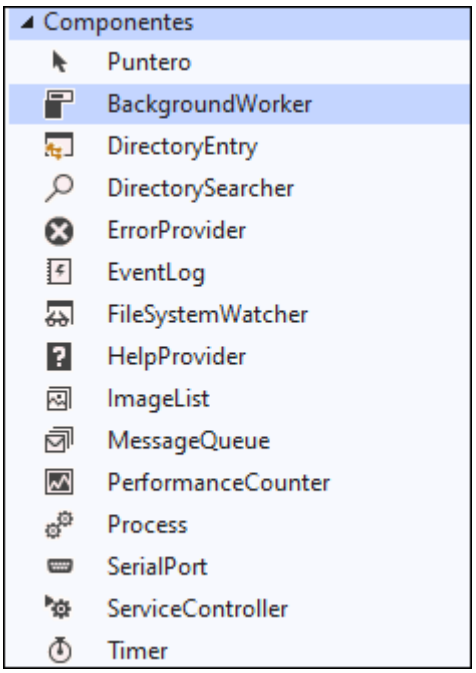


Tan pronto como se produce la primera llamada al método asíncrono, la ejecución vuelve al método que invoca. Cuando el método asíncrono termina, la ejecución se retoma donde se detuvo, dentro de la iteración. Las variables locales de la iteración y, en particular, el contador conserva sus valores.

El componente BackgroundWorker

El componente `BackgroundWorker` se utiliza en el formulario Send de la solución de ejemplo.

La caja de herramientas de Visual Studio proporciona el componente `BackgroundWorker`. Resulta muy útil para las aplicaciones de Windows que deben gestionar una interfaz de usuario y, al mismo tiempo, se tienen que refrescar para poder realizar de esta manera, las operaciones complicadas.



Como indica la interfaz, el componente `BackgroundWorker` permite ejecutar una operación en un thread separado. Basta con instanciar un nuevo objeto `BackgroundWorker` desde el diseñador de pantallas o mediante código e indicarle el método a ejecutar en un thread separado:

```
BackgroundWorker bw = new BackgroundWorker();
bw.DoWork += new DoWorkEventHandler(bw_DoWork);
```

Cuando desee iniciar la ejecución de las tareas en segundo plano, llame al método `RunWorkerAsync` del objeto `BackgroundWorker`:

```
bw.RunWorkerAsync();
```

El componente se ha simplificado al máximo para facilitar su uso. Contiene dos propiedades, `WorkerReportsProgress` y `WorkerSupportsCancellation`, que permiten respectivamente especificar si el componente indica su progresión a los objetos suscritos a su evento `ProgressChanged` y si la ejecución de la tarea desatendida se puede anular mediante el método `CancelAsync`:

```
bw.WorkerReportsProgress = true;
bw.WorkerSupportsCancellation = true;
```

El componente también tiene tres eventos:

- `DoWork`: este evento se desencadena mediante el método `RunWorkerAsync` del objeto. Lanza la ejecución del método asociado a su delegado en un thread separado:

```
bw.DoWork += new DoWorkEventHandler(bw_DoWork);
bw.RunWorkerAsync();

void bw_DoWork(object sender, DoWorkEventArgs e)
{
    Library.MailerTools.Send();
```

```
}
```

El controlador del evento recibe como argumento un objeto del tipo `DoWorkEventArgs` con una propiedad `Argument` que permite transferir datos para la ejecución de la tarea desatendida gracias a una sobrecarga del método `RunWorkerAsync`.

- `ProgressChanged`: este evento se desencadena durante la llamada al método `ReportProgress` del objeto. Recibe como argumentos un valor entero, que permite indicar el porcentaje de avance de la tarea, y un objeto que permite transferir cualquier tipo de datos a los controladores del evento:

```
bw.ProgressChanged +=
          new ProgressChangedEventHandler(bw_ProgressChanged);

void bw_DoWork(object sender, DoWorkEventArgs e)
{
    bw.ReportProgress(0, "Envío en curso ");
    Library.MailerTools.Send();
}
void bw_ProgressChanged(object sender,
                  ProgressChangedEventArgs e)
{
    this.lblStatus.Text = e.UserState.ToString();
}
```

El controlador del evento recibe como argumento un objeto del tipo `ProgressChanged EventArgs`, que expone los valores asignados durante la llamada al método `ReportProgress`, es decir, `ProgressPercentage` y `UserState`.

- `RunWorkerCompleted`: este evento se desencadena automáticamente cuando termina la ejecución de la tarea desatendida y los controladores de este evento pueden realizar las operaciones deseadas, como la actualización de la interfaz de usuario:

```
bw.RunWorkerCompleted +=
      new
RunWorkerCompletedEventHandler(bw_RunWorkerCompleted);

void bw_RunWorkerCompleted(object sender,
                           RunWorkerCompletedEventArgs e)
{
    this.lblStatus.Text = "Envío terminado ";
}
```

El controlador del evento recibe como argumento un objeto del tipo `RunWorkerCompletedEventArgs`, que expone las propiedades `Result`, de tipo

`object`, que permiten transferir los resultados de la tarea desatendida, y `UserState`, que permite transferir los datos originales.

El componente `BackgroundWorker` expone también dos propiedades en modo de solo lectura: `CancellationPending` e `IsBusy`.

- `CancellationPending` indica que se ha invocado al método `CancelAsync`. La tarea desatendida debe implementar una manera de anular su ejecución comprobando el valor de esta propiedad, ya que el thread no se eliminará y la ejecución continuará.
- `IsBusy` permite saber si la tarea en segundo plano se está ejecutando o no. Su valor es igual a `true` cuando se invoca al método `RunWorkerAsync` y `false` cuando el trabajo termina y se desencadena el evento `RunWorkerCompleted`.