Capítulo 6
Serialización

1. Serialización en C#

Hoy en día, el lenguaje C# es uno de los más usados para desarrollar aplicaciones web. Uno de los problemas más frecuentes durante la comunicación en programación web es conseguir transferir objetos hacia y desde una aplicación. Para eso, el objeto se debe transformar en un formato universal. Esta transformación se llama serialización. Este proceso permite recuperar el objeto representado bajo un formato intercambiable; con frecuencia este formato tiene una forma textual.

Para que los datos puedan transitar, hay que usar un flujo de datos (llamado *stream* en inglés, de ahí el nombre de la clase base en C#: `Stream`). Estos flujos pueden tomar varias formas, como por ejemplo un flujo de datos en memoria (representado por la clase `MemoryStream` en C#) o incluso un flujo de datos hacia un archivo en el disco (representado por la clase `FileStream` en C#).

Hay varias maneras de serializar un objeto; estas son algunas de ellas:

- La serialización binaria, que permite representar un objeto en un formato binario.
- La serialización XML, que transforma el objeto en cadena de caracteres al formato XML.
- La serialización JSON, que transforma el objeto en cadena de caracteres al formato JSON.

En este capítulo vamos a abordar tres modos de serialización para transformar un objeto a un formato dado, pero también para poder recuperar un objeto desde una fuente de datos en el formato retenido.

2. Serialización binaria

El enfoque binario es el modo de serialización más simple para implantar. También es el que permite almacenar mayor cantidad de información en el tipo del objeto y tiene más fiabilidad de conversión de los valores y tipos almacenados. Sin embargo, su formato es propietario: no es portable ni compatible con otros lenguajes y soluciones, de manera que solo lo usaremos si el emisor y el destinatario son programas en C#.

Observación

Con la llegada de .NET 6 y C# 10, no se recomienda en absoluto usar este modo de serialización. Las explicaciones y el funcionamiento descritos en esta sección solo se proponen con fines de seguimiento, para los lectores que necesiten hacer el mantenimiento de un sistema usando este sistema de serialización. No se recomienda usar los objetos vistos en esta sección para aplicaciones nuevas porque están marcados como obsoletos en .NET 6 y se eliminarán en .NET 7.

Dos planteamientos permiten activar la serialización binaria:

– Usando los atributos apropiados en un tipo dado.

– Implementando la interfaz `ISerializable`.

Los atributos son más fáciles y rápidos de implantar, pero menos flexibles que la implementación de la interfaz `ISerializable`. Es recomendable elegir la mejor solución en función del objetivo.

2.1 Uso de los atributos

Este planteamiento es el más sencillo y rápido de implantar. Considerando la siguiente clase, solo hay que añadir el atributo `[Serializable]` encima de la declaración de la clase:

```
[Serializable]
public class Persona
{
    public string Nombre { get; set; }
    public string Apellido { get; set; }
}
```

De este modo, cuando se pida serializar una variable de tipo `Persona`, se informará al serializador, mediante la presencia del atributo, de que debe considerar la serialización de cada dato contenido en el interior del tipo, en forma de campos. Cada uno de los datos debe ser serializable; en caso contrario, se devolverá una excepción durante el proceso.

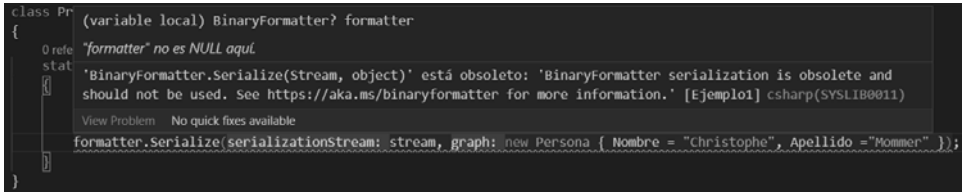
Una vez marcado el objeto, se puede usar el serializador binario, `BinaryFormatter`, que se encuentra en el espacio de nombres `System.Runtime.Serialization.Formatters.Binary`, para serializar el objeto en un stream de datos. El stream se puede guardar en memoria o en el sistema de archivos (este último caso es bastante habitual para guardar objetos en el disco entre dos ejecuciones de un programa). Se usa el método `Serialize` de este objeto para precisar el stream de destino, así como el objeto que se ha de serializar:

```
var formatter = new BinaryFormatter();
using(var stream = new FileStream("person.bin", FileMode. Create))
{
    formatter.Serialize(stream, new Persona { Nombre = "Christophe",
Apellido ="Mommer" });
}
```

Observación

La instrucción using delante de una variable se estudiará en el capítulo Conceptos avanzados, en la sección Gestión de la memoria.

Se comprueba que Visual Studio Code lanza un aviso de escritura de este código en .NET 6 porque el tipo ha sido devaluado y se eliminará en .NET 7, según la hoja de ruta de Microsoft:



Aviso en el editor con el uso del tipo BinaryFormatter

La clase `BinaryFormatter` también permite leer contenido desde un stream dado, gracias al método `Deserialize`. Se puede observar que este método devuelve una instancia de tipo `object`; por eso es necesario hacer un cast:

```
using(var reader = new FileStream("person.bin", FileMode.Open))
{
    var person = (Persona)formatter.Deserialize(reader);
    Console.WriteLine("Hola " + person.Nombre + " " + person.Apellido);
}
```

Dentro de la clase, se puede especificar que no se quiere serializar un dato concreto añadiendo el atributo `[NonSerialized]` encima del dato afectado.

Observación

Este atributo solo funciona en los campos y no en las propiedades, lo que demuestra una cierta debilidad del planteamiento de la serialización binaria.

```
[Serializable]
public class Persona
{
    public string Nombre { get; set; }
    public string Apellido { get; set; }
    [NonSerialized]
    public int Edad;
}

var formatter = new BinaryFormatter();
using (var stream = new FileStream("person.bin", FileMode.Create))
{
```

```
    formatter.Serialize(stream, new Persona { Nombre = " Christophe",
Apellido = "Mommer", Edad = 33 });
}

using (var reader = new FileStream("person.bin", FileMode.Open))
{
    var person = (Persona)formatter.Deserialize(reader);
    Console.WriteLine("Hola " + person.Nombre + " " + person.Apellido
+ ". Tiene " + person.Edad + " años"); // Mostrará Hola Christophe
Mommer. Tiene 0 años
}
```

El valor de un campo marcado como `NonSerialized` siempre será igual a `null` (en caso de tipo de referencia) o al valor predeterminado (en caso de tipo de valor), incluso si el constructor de la clase especifica un valor. El serializador binario es el único que no invoca al constructor de una clase en la deserialización.

Sin embargo, se puede definir un método al que se llamará durante la etapa de deserialización binaria. Este método no debe devolver nada y toma un único parámetro de tipo `StreamingContext`. Se añade el atributo `[OnDeserializing]` encima del método afectado:

```
[Serializable]
public class Persona
{
    public string Nombre { get; set; }
    public string Apellido { get; set; }
    [NonSerialized]
    public int Edad;

    [OnDeserializing]
    public void OnDeserializing(StreamingContext context)
    {
        Edad = 33; // aquí, se fija el valor 33 de manera permanente
en la deserialización
    }
}
```

También existen los siguientes atributos para crear métodos que se invocarán durante la (de)serialización:

– `[OnSerializing]`: se define en el método que se invoca durante la serialización.
– `[OnSerialized]`: se define en el método que se invoca cuando la serialización ha terminado.
– `[OnDeserialized]`: se define en el método que se invoca cuando la deserialización ha terminado.

Para finalizar la gestión por atributo, hay que recordar que el serializador binario puede ser sensible en caso de cambio del tipo serializado. En efecto, si el tipo de datos cambia entre la descripción de la clase y los datos serializados, se produce un error en la ejecución.

De la misma manera, si se ha añadido un dato, hay que mencionárselo al serializador porque, para él, deben estar presentes todos los datos no marcados como `[NonSerialized]`. Podemos usar el atributo `[OptionalField]` para especificar que el dato es en efecto opcional (dado que no estaba en una versión anterior). Este atributo también permite almacenar la versión donde se ha añadido el dato:

```
[Serializable]
public class Persona
{
    public string Nombre { get; set; }
    public string Apellido { get; set; }
    [NonSerialized]
    public int Edad;
    [OptionalField(VersionAdded = 2)]
    public DateTime FechaDeNacimiento;
}
```

2.2 Uso de la interfaz ISerializable

El otro planteamiento para serializar un objeto de manera binaria es hacer implementar la interfaz `ISerializable` mediante el objeto. Esta interfaz solo contiene un método:

```
void GetObjectData(SerializationInfo info, StreamingContext context);
```

Por lo tanto, hay que implementar este método para describir lo que se quiere serializar y cómo. El enfoque, aunque es más flexible, también es más complejo que el uso de los atributos.

Observación

A pesar de la implementación de la interfaz, sigue siendo necesario conservar el atributo `[Serializable]` encima de la clase.

El objeto `SerializationInfo` es un tipo de diccionario que permite asignar los datos de la clase que se va a serializar mediante clave/valor. Por ejemplo, si se retoma nuestra clase Persona con este enfoque, el código sería el siguiente:

```
[Serializable]
public class Persona2 : ISerializable
{
    public string Nombre { get; set; }
    public string Apellido { get; set; }
    public int Edad { get; set; }
    public DateTime FechaDeNacimiento { get; set; }

      public void GetObjectData(SerializationInfo info,
StreamingContext context)
    {
        info.AddValue("Nombre", Nombre);
        info.AddValue("Apellido", Apellido);
    }
}
```

Se constata que se han eliminado todos los atributos de gestión interna de la serialización, ya que la función del método `GetObjectData` es definir qué miembros se deben serializar.

Capítulo 5
Presentación de las herramientas

1. Elegir las herramientas

Hay dos maneras de abordar la cuestión de las herramientas:

– Presentar una lista de varios perfiladores y soluciones de benchmarking para que el lector se haga una idea general de las posibilidades disponibles en el mercado.

– Enfocarse directamente sobre un conjunto de herramientas que se utilizarán en el resto del libro.

Aunque sería interesante disponer de una lista para poder comparar sus respectivas funcionalidades, esto no sería necesariamente interesante en la medida en que no se detallarían las herramientas enumeradas, lo que obligaría a realizar una investigación adicional. Por ello, se decidió que este capítulo se concentre en la presentación de las herramientas de perfilado y de benchmarking que se utilizarán en los capítulos siguientes, para que el lector tenga ya una introducción a las mismas antes de utilizarlas en mayor profundidad.

La elección de las herramientas en este capítulo, y por tanto en este libro, se ha realizado en función del estado actual del mercado, para que el lector pueda equiparse de forma rápida y económica. Cabe destacar que la selección de herramientas es casi exclusiva de Windows, debido a la madurez del marco de trabajo en este sistema operativo. No obstante, existen variantes para Linux y macOS, pero simplemente se mencionarán y no se estudiarán en detalle. Por lo tanto, es necesario que el lector disponga al menos de una máquina virtual con Windows para poder seguir las actividades prácticas.

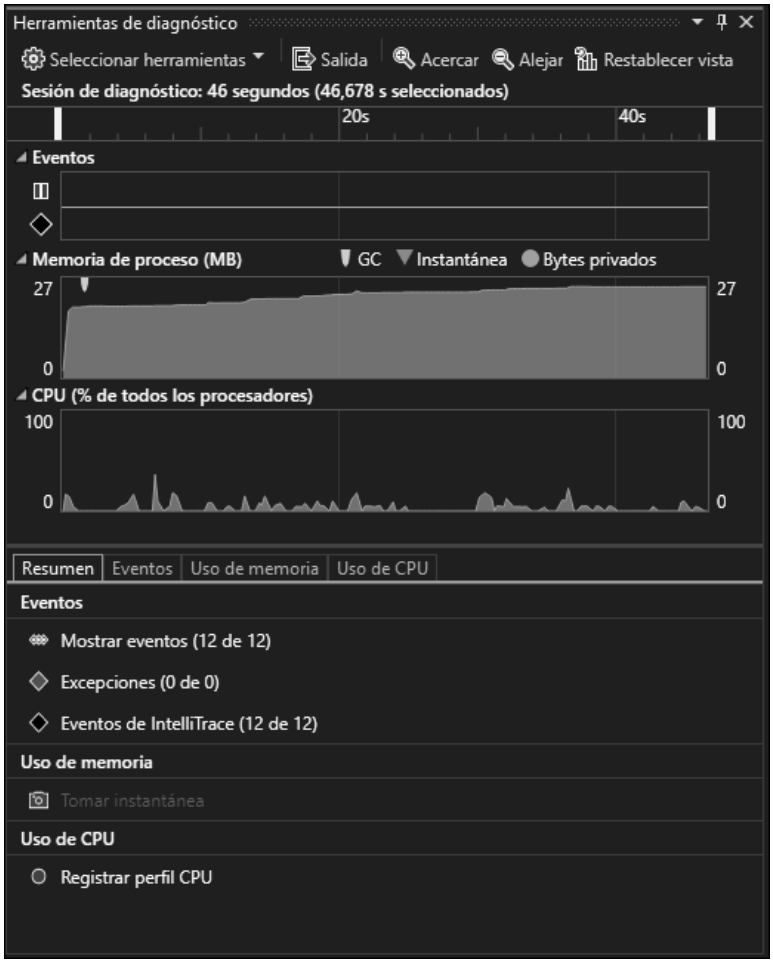
2. Visual Studio 2022

Visual Studio es muy a menudo la principal herramienta de trabajo del desarrollador .NET. Hoy en día existen alternativas viables, más o menos avanzadas, como Visual Studio Code o Rider de JetBrains; pero como la mayoría de los desarrolladores de .NET utilizan Visual Studio cuando trabajan en Windows, fue este último el que se eligió para el análisis de rendimiento.

La ventaja de utilizar Visual Studio es que ofrece una integración directa del conjunto de herramientas sin tener que cambiar de entorno conforme se avanza a lo largo del proyecto. Esto significa que se puede realizar una sesión de perfilado a medida que se desarrolla una funcionalidad, si esta se requiere. Además, es la propia Microsoft la que produce Visual Studio. Como tal, el IDE está lógicamente muy correlacionado y actualizado con el .NET Framework.

2.1 Ventana de diagnóstico

En su configuración predefinida, Visual Studio 2022 muestra directamente las herramientas de diagnóstico durante una sesión de depuración:



Ventana de diagnóstico al depurar con Visual Studio

Observación

*Si esta ventana no está visible, puede mostrarla al entrar en el modo de Depuración (Debug) yendo al menú **Depurar**, luego a **Ventanas** y seleccionando **Mostrar herramientas de diagnóstico**. El método abreviado predefinido es [Ctrl][Alt][F2].*

Aunque esta ventana da una idea de cómo la utilización de la aplicación está consumiendo recursos y comportándose, no es la principal fuente de información para una sesión de perfilado. Sin embargo, le permite hacer un seguimiento de algunos de los principales eventos, así como obtener una visión general del porcentaje de CPU consumido y del consumo de RAM.

En ella encontrará los siguientes elementos:

– Los momentos en que el GC pasa, simbolizados por pequeñas marcas amarillas en el consumo de memoria.
– Las excepciones encontradas en el flujo de código, simbolizadas por un rombo rojo.
– Instantáneas de la memoria tomadas por el desarrollador (se trata de tomar una foto de la memoria en un instante T, para compararla con otra foto tomada antes o después, o simplemente para explorar el estado de la memoria en ese momento).

También hay varias pestañas que permiten navegar a otras fuentes de información:

– La pestaña **Eventos** enumera los sucesos más destacados que han ocurrido mientras la aplicación estaba en ejecución.
– La pestaña **Uso de memoria** lista las instantáneas tomadas por el desarrollador durante la ejecución y permite explotarlas.
– La pestaña **Uso de CPU** permite observar el consumo de la CPU. De forma predefinida, el perfilado de la CPU está desactivado (y hay que ir a la pestaña para activarlo explícitamente) porque consume recursos que, de hecho, no están asignados a la aplicación.

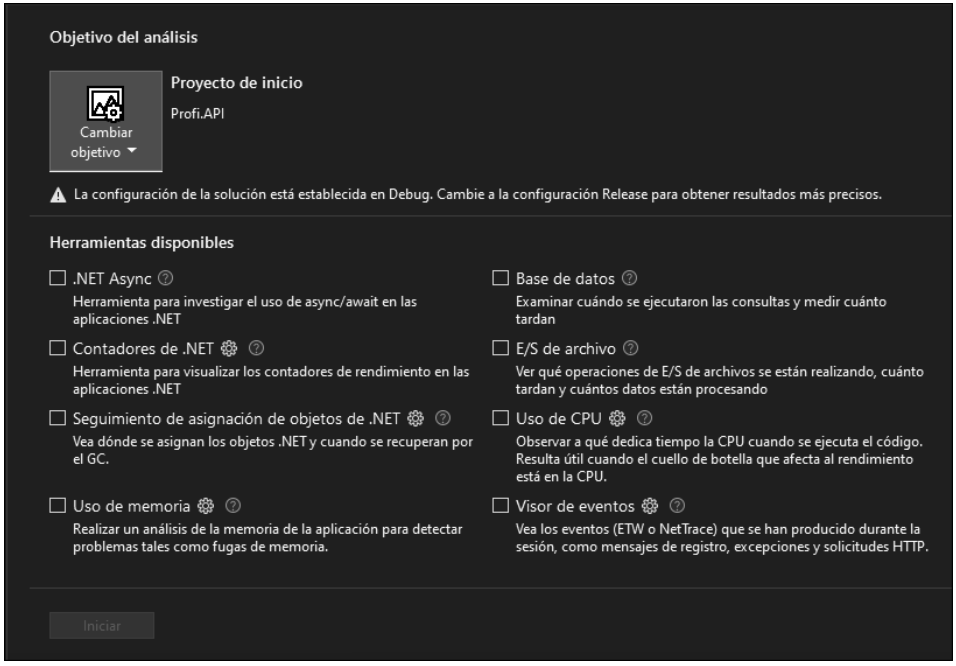
Lo que contiene esta ventana de diagnóstico es una visión general de las herramientas de perfilado, ya que se trata de una evaluación de información sobre la marcha y no de una sesión dedicada.

Sin embargo, esta ventana puede utilizarse durante el desarrollo de una funcionalidad clásica para vigilar las principales métricas relacionadas con el rendimiento.

2.2 Sesión de perfilado

Este es el corazón de las herramientas de supervisión del rendimiento que ofrece Visual Studio. Esta herramienta permite ejecutar una sesión de perfilado dedicada sin utilizar el depurador. Para ello, utilice el asistente que se ofrece, disponible en el menú **Depurar**, y luego elija **Generador de perfiles de rendimiento** (método abreviado predefinido [Alt][F2]).

Cuando se inicia la sesión, se propone un asistente que enumera al mismo tiempo todo lo que se puede rastrear:



Ventana del asistente de las herramientas de perfilado

Observación

Esta pantalla oculta de forma predefinida las herramientas no disponibles para la aplicación actual. Es posible mostrarlas haciendo clic en el enlace ***Mostrar todas las herramientas*** *en la parte superior derecha de la ventana.*

A continuación, puede marcar y configurar las herramientas que le interesen para supervisar el rendimiento. En una sesión típica de perfilado, se suele elegir **Uso de CPU** y, en menor medida, el **Uso de memoria**.

La herramienta nos lo recuerda bastante, pero se realiza una sesión de perfilado en una versión de la aplicación final (Release), ya que así se eliminan todas las instrucciones IL innecesarias (**nop**), colocadas con fines de depuración, y también se activan las optimizaciones realizadas por el compilador, que es lo más parecido a una aplicación desplegada en producción (lista para su comercialización).

El botón de la parte superior de la ventana nos permite elegir el objetivo en el que las herramientas deben ejecutarse. El problema es que nuestra solución tiene dos aplicaciones separadas, que deben ejecutarse simultáneamente para que nuestro proyecto funcione. Por lo tanto, hay dos posibilidades:

– Elegir perfilar solo una de las dos aplicaciones y ejecutar la otra directamente en segundo plano.
– Iniciar una nueva instancia de Visual Studio que permita perfilar la otra aplicación y ejecutar las herramientas de perfilado en ambas instancias simultáneamente.

Observación

En cualquier caso, lo ideal es ejecutar primero la API para evitar que el cliente se quede sin ella y no pueda funcionar correctamente.

Tenga cuidado, si cambia el objetivo de inicio directamente desde Visual Studio, tendrá que «refrescar» el asistente para que tenga en cuenta el nuevo proyecto de inicio, bien cerrándolo y volviéndolo a abrir, o bien haciendo clic en el botón ***Cambiar objetivo*** *y luego haciendo clic en* ***Proyecto de inicio****. El proyecto que se va a analizar y se muestra junto al botón.*

Una vez seleccionadas y configuradas las herramientas, basta con hacer clic en el botón **Iniciar** situado en la parte inferior del asistente. Este se encargará de ejecutar el proceso y Visual Studio se adjuntará a él para comenzar a recopilar las métricas.

Cuando el usuario detiene la aplicación (normalmente cerrando la consola asociada) o se pide explícitamente a Visual Studio que deje de recolectar datos (haciendo clic en el enlace para detener dicha acción en la ventana principal de Visual Studio), Visual Studio comenzará a preparar el resumen de la sesión y lo presentará bajo la forma de un recapitulativo:

Recapitulativo de una sesión de perfilado

Observación

En este caso, el mejor enfoque es ejecutar los dos programas compilados en modo Release de forma autónoma y utilizar el perfilador de Visual Studio para adjuntarlo a un proceso externo.

Visual Studio podrá destacar las rutas clasificadas como calientes, es decir, las que ocupan más CPU en la sesión de perfilado que se está ejecutando.