EBook Gratis

APRENDIZAJE .NET Framework

Free unaffiliated eBook created from
Stack Overflow contributors.

#.net

Tabla de contenido

Acerca de

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

It is an unofficial and free .NET Framework 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 .NET Framework.

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 .NET Framework

Observaciones

.NET Framework es un conjunto de bibliotecas y un tiempo de ejecución, diseñado originalmente por Microsoft. Todos los programas .NET se compilan en un código de bytes llamado Microsoft Intermediate Language (MSIL). MSIL se ejecuta mediante Common Language Runtime (CLR).

A continuación puede encontrar varios ejemplos de "Hello World" en varios idiomas que son compatibles con .NET Framework. "Hello World" es un programa que muestra "Hello World" en el dispositivo de pantalla. Se utiliza para ilustrar la sintaxis básica para construir un programa de trabajo. También se puede usar como prueba de cordura para asegurarse de que el compilador de un idioma, el entorno de desarrollo y el entorno de ejecución estén funcionando correctamente.

Lista de idiomas soportados por .NET

Versiones

.RED

Versión	Fecha de lanzamiento
1.0	2002-02-13
1.1	2003-04-24
2.0	2005-11-07
3.0	2006-11-06
3.5	2007-11-19
3.5 SP1	2008-08-11
4.0	2010-04-12
4.5	2012-08-15
4.5.1	2013-10-17
4.5.2	2014-05-05
4.6	2015-07-20
4.6.1	2015-11-17
4.6.2	2016-08-02

Versión	Fecha de lanzamiento
4.7	2017-04-05

Marco Compacto

Versión	Fecha de lanzamiento
1.0	2000-01-01
2.0	2005-10-01
3.5	2007-11-19
3.7	2009-01-01
3.9	2013-06-01

Micro marco

Versión	Fecha de lanzamiento
4.2	2011-10-04
4.3	2012-12-04
4.4	2015-10-20

Examples

Hola mundo en c

```
using System;

class Program
{
    // The Main() function is the first function to be executed in a program
    static void Main()
    {
        // Write the string "Hello World to the standard out
        Console.WriteLine("Hello World");
    }
}
```

`Console.WriteLine` tiene varias sobrecargas. En este caso, la cadena "Hello World" es el parámetro, y emitirá "Hello World" al flujo de salida estándar durante la ejecución. Otras sobrecargas pueden llamar al `.ToString` del argumento antes de escribir en la secuencia. Consulte

la documentación de .NET Framework para obtener más información.

Demo en vivo en acción en .NET Fiddle

Introducción a C #

Hola mundo en Visual Basic .NET

```
Imports System

Module Program
    Public Sub Main()
        Console.WriteLine("Hello World")
    End Sub
End Module
```

Demo en vivo en acción en .NET Fiddle

Introducción a Visual Basic .NET

Hola mundo en F

```
open System

[<EntryPoint>]
let main argv =
    printfn "Hello World"
    0
```

Demo en vivo en acción en .NET Fiddle

Introducción a F #

Hola Mundo en C ++ / CLI

```
using namespace System;

int main(array<String^>^ args)
{
    Console::WriteLine("Hello World");
}
```

Hola mundo en PowerShell

```
Write-Host "Hello World"
```

Introducción a PowerShell

Hola Mundo en Nemerle

```
System.Console.WriteLine("Hello World");
```

Hola Mundo en Oxygene

```
namespace HelloWorld;

interface

type
  App = class
  public
    class method Main(args: array of String);
  end;

implementation

class method App.Main(args: array of String);
begin
  Console.WriteLine('Hello World');
end;

end.
```

Hola mundo en boo

```
print "Hello World"
```

Hola Mundo en Python (IronPython)

```
print "Hello World"
```

```
import clr
from System import Console
Console.WriteLine("Hello World")
```

Hola Mundo en IL

```
.class public auto ansi beforefieldinit Program
       extends [mscorlib]System.Object
{
  .method public hidebysig static void  Main() cil managed
  {
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello World"
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  }

  .method public hidebysig specialname rtspecialname
         instance void  .ctor() cil managed
  {
```

```
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  }

}
```

Lea Empezando con .NET Framework en línea: https://riptutorial.com/es/dot-net/topic/14/empezando-con--net-framework

Capítulo 2: .NET Core

Introducción

.NET Core es una plataforma de desarrollo de propósito general mantenida por Microsoft y la comunidad .NET en GitHub. Es multiplataforma, es compatible con Windows, macOS y Linux, y puede usarse en dispositivos, en la nube y en escenarios integrados / IoT.

Cuando piense en .NET Core lo siguiente debería venir a la mente (implementación flexible, multiplataforma, herramientas de línea de comandos, código abierto).

Otra gran cosa es que incluso si es de código abierto, Microsoft lo está apoyando activamente.

Observaciones

Por sí mismo, .NET Core incluye un único modelo de aplicación, aplicaciones de consola, que es útil para herramientas, servicios locales y juegos basados en texto. Se han construido modelos de aplicaciones adicionales sobre .NET Core para ampliar su funcionalidad, como por ejemplo:

- ASP.NET Core
- Windows 10 Universal Windows Platform (UWP)
- Xamarin.Formas

Además, .NET Core implementa la biblioteca estándar de .NET y, por lo tanto, es compatible con las bibliotecas estándar de .NET.

La biblioteca estándar de .NET es una especificación de API que describe el conjunto consistente de API de .NET que los desarrolladores pueden esperar en cada implementación de .NET. Las implementaciones de .NET necesitan implementar esta especificación para ser consideradas compatibles con la biblioteca estándar de .NET y para admitir las bibliotecas que tienen como destino la biblioteca estándar de .NET.

Examples

Aplicación de consola básica

```
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("\nWhat is your name? ");
        var name = Console.ReadLine();
        var date = DateTime.Now;
        Console.WriteLine("\nHello, {0}, on {1:d} at {1:t}", name, date);
        Console.Write("\nPress any key to exit...");
        Console.ReadKey(true);
    }
```

```
}
```

Lea .NET Core en línea: https://riptutorial.com/es/dot-net/topic/9059/-net-core

Capítulo 3: Acrónimo de glosario

Examples

.Net acrónimos relacionados

Tenga en cuenta que algunos términos como JIT y GC son lo suficientemente genéricos como para aplicarse a muchos entornos de lenguaje de programación y tiempos de ejecución.

CLR: Common Language Runtime
IL: lenguaje intermedio
EE: motor de ejecución
JIT: compilador Just-in-time
GC: recolector de basura
OOM: Sin memoria
STA: apartamento de un solo hilo
MTA: apartamento multihilo

Lea Acrónimo de glosario en línea: https://riptutorial.com/es/dot-net/topic/10939/acronimo-de-glosario

Capítulo 4: ADO.NET

Introducción

ADO (ActiveX Data Objects) .Net es una herramienta proporcionada por Microsoft que proporciona acceso a fuentes de datos como SQL Server, Oracle y XML a través de sus componentes. Las aplicaciones de front-end de .Net pueden recuperar, crear y manipular datos, una vez que se conectan a una fuente de datos a través de ADO.Net con los privilegios adecuados.

ADO.Net proporciona una arquitectura sin conexión. Es un enfoque seguro para interactuar con una base de datos, ya que la conexión no tiene que mantenerse durante toda la sesión.

Observaciones

Una nota sobre la parametrización de SQL con `Parameters.AddWithValue`: `AddWithValue` nunca es un buen punto de partida. Ese método se basa en inferir el tipo de datos de lo que se pasa. Con esto, puede terminar en una situación en la que la conversión impide que su consulta utilice un índice . Tenga en cuenta que algunos tipos de datos de SQL Server, como `char` / `varchar` (sin la "n") o la `date` , no tienen un tipo de datos .NET correspondiente. En esos casos, se debe usar `Add` con el tipo de datos correcto .

Examples

Ejecutando sentencias SQL como un comando

```
// Uses Windows authentication. Replace the Trusted_Connection parameter with
// User Id=...;Password=...; to use SQL Server authentication instead. You may
// want to find the appropriate connection string for your server.
string connectionString =
@"Server=myServer\myInstance;Database=myDataBase;Trusted_Connection=True;"

string sql = "INSERT INTO myTable (myDateTimeField, myIntField) " +
    "VALUES (@someDateTime, @someInt);";

// Most ADO.NET objects are disposable and, thus, require the using keyword.
using (var connection = new SqlConnection(connectionString))
using (var command = new SqlCommand(sql, connection))
{
    // Use parameters instead of string concatenation to add user-supplied
    // values to avoid SQL injection and formatting issues. Explicitly supply datatype.

    // System.Data.SqlDbType is an enumeration. See Note1
    command.Parameters.Add("@someDateTime", SqlDbType.DateTime).Value = myDateTimeVariable;
    command.Parameters.Add("@someInt", SqlDbType.Int).Value = myInt32Variable;

    // Execute the SQL statement. Use ExecuteScalar and ExecuteReader instead
    // for query that return results (or see the more specific examples, once
    // those have been added).
```

```
    connection.Open();
    command.ExecuteNonQuery();
}
```

Nota 1: Consulte la enumeración de SqlDbType para la variación específica de MSFT SQL Server.

Nota 2: Consulte la enumeración de MySqlDbType para la variación específica de MySQL.

Buenas prácticas - Ejecución de sentencias SQL

```
public void SaveNewEmployee(Employee newEmployee)
{
    // best practice - wrap all database connections in a using block so they are always
closed & disposed even in the event of an Exception
    // best practice - retrieve the connection string by name from the app.config or
web.config (depending on the application type) (note, this requires an assembly reference to
System.configuration)
    using(SqlConnection con = new
SqlConnection(System.Configuration.ConfigurationManager.ConnectionStrings["MyConnectionName"].Connectio

    {
        // best practice - use column names in your INSERT statement so you are not dependent
on the sql schema column order
        // best practice - always use parameters to avoid sql injection attacks and errors if
malformed text is used like including a single quote which is the sql equivalent of escaping
or starting a string (varchar/nvarchar)
        // best practice - give your parameters meaningful names just like you do variables in
your code
        using(SqlCommand sc = new SqlCommand("INSERT INTO employee (FirstName, LastName,
DateOfBirth /*etc*/) VALUES (@firstName, @lastName, @dateOfBirth /*etc*/)", con))
        {
            // best practice - always specify the database data type of the column you are
using
            // best practice - check for valid values in your code and/or use a database
constraint, if inserting NULL then use System.DbNull.Value
            sc.Parameters.Add(new SqlParameter("@firstName", SqlDbType.VarChar, 200){Value =
newEmployee.FirstName ?? (object) System.DBNull.Value});
            sc.Parameters.Add(new SqlParameter("@lastName", SqlDbType.VarChar, 200){Value =
newEmployee.LastName ?? (object) System.DBNull.Value});

            // best practice - always use the correct types when specifying your parameters,
Value is assigned to a DateTime instance and not a string representation of a Date
            sc.Parameters.Add(new SqlParameter("@dateOfBirth", SqlDbType.Date){ Value =
newEmployee.DateOfBirth });

            // best practice - open your connection as late as possible unless you need to
verify that the database connection is valid and wont fail and the proceeding code execution
takes a long time (not the case here)
            con.Open();
            sc.ExecuteNonQuery();
        }

        // the end of the using block will close and dispose the SqlConnection
        // best practice - end the using block as soon as possible to release the database
connection
    }
```

```
}

// supporting class used as parameter for example
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
}
```

Mejores prácticas para trabajar con ADO.NET

- La regla de oro es abrir la conexión por un tiempo mínimo. Cierre la conexión explícitamente una vez que finalice la ejecución del procedimiento, el objeto de conexión volverá al grupo de conexiones. Tamaño máximo del conjunto de conexiones predeterminado = 100. Como el conjunto de conexiones mejora el rendimiento de la conexión física a SQL Server. Conexión de agrupación en SQL Server
- Envuelva todas las conexiones de la base de datos en un bloque de uso para que siempre estén cerradas y eliminadas incluso en el caso de una Excepción. Consulte Uso de la declaración (Referencia de C #) para obtener más información sobre el uso de declaraciones
- Recupere las cadenas de conexión por nombre desde app.config o web.config (dependiendo del tipo de aplicación)
    - Esto requiere una referencia de ensamblaje a `System.configuration`
    - Consulte Cadenas de conexión y archivos de configuración para obtener información adicional sobre cómo estructurar su archivo de configuración.
- Siempre use parámetros para los valores entrantes a
    - Evitar los ataques de inyección sql
    - Evite los errores si se usa texto con formato incorrecto, como incluir una comilla simple que es el equivalente en sql de escapar o iniciar una cadena (varchar / nvarchar)
    - Permitir que el proveedor de la base de datos reutilice los planes de consulta (no es compatible con todos los proveedores de la base de datos) lo que aumenta la eficiencia
- Al trabajar con parámetros.
    - El tipo de parámetros de SQL y la falta de coincidencia de tamaño es una causa común de error de inserción / actualización / selección
    - Dale a tus parámetros de Sql nombres significativos al igual que lo haces con las variables en tu código
    - Especifique el tipo de datos de la base de datos de la columna que está utilizando, esto garantiza que no se usen los tipos de parámetros incorrectos, lo que podría dar lugar a resultados inesperados
    - Valide los parámetros entrantes antes de pasarlos al comando (como dice el dicho, " basura dentro, basura fuera "). Valide los valores entrantes lo antes posible en la pila
    - Utilice los tipos correctos cuando asigne sus valores de parámetros, por ejemplo: no asigne el valor de cadena de DateTime, en su lugar, asigne una instancia de DateTime real al valor del parámetro
    - Especifique el tamaño de los parámetros de tipo cadena. Esto se debe a que SQL

Server puede reutilizar los planes de ejecución si los parámetros coinciden en tipo *y* tamaño. Usa -1 para MAX
    - No utilice el método AddWithValue , la razón principal es que es muy fácil olvidar especificar el tipo de parámetro o la precisión / escala cuando sea necesario. Para información adicional, vea ¿Podemos dejar de usar AddWithValue ya?
- Cuando se utilizan conexiones de base de datos
    - Abra la conexión lo más tarde posible y ciérrela tan pronto como sea posible. Esta es una guía general cuando se trabaja con cualquier recurso externo
    - Nunca comparta instancias de conexión de base de datos (ejemplo: tener un host singleton una instancia compartida de tipo `SqlConnection`). Haga que su código siempre cree una nueva instancia de conexión de base de datos cuando sea necesario y luego haga que el código de llamada lo elimine y "deséchelo" cuando termine. La razón de esto es
        - La mayoría de los proveedores de bases de datos tienen algún tipo de agrupación de conexiones, por lo que crear nuevas conexiones administradas es barato
        - Elimina cualquier error futuro si el código comienza a trabajar con varios subprocesos

Usando interfaces comunes para abstraer clases específicas

```
var providerName = "System.Data.SqlClient";    //Oracle.ManagedDataAccess.Client, IBM.Data.DB2
var connectionString = "{your-connection-string}";
//you will probably get the above two values in the ConnectionStringSettings object from
.config file

var factory = DbProviderFactories.GetFactory(providerName);
using(var connection = factory.CreateConnection()) {    //IDbConnection
    connection.ConnectionString = connectionString;
    connection.Open();

    using(var command = connection.CreateCommand()) {    //IDbCommand
        command.CommandText = "{query}";

        using(var reader = command.ExecuteReader()) {    //IDataReader
            while(reader.Read()) {
                ...
            }
        }
    }
}
```

Lea ADO.NET en línea: https://riptutorial.com/es/dot-net/topic/3589/ado-net

Capítulo 5: Ajustes

Examples

AppSettings de ConfigurationSettings en .NET 1.x

Uso en desuso

La clase ConfigurationSettings fue la forma original de recuperar la configuración de un ensamblaje en .NET 1.0 y 1.1. Ha sido reemplazado por la clase ConfigurationManager y la clase WebConfigurationManager .

Si tiene dos claves con el mismo nombre en la sección de `appSettings` de la aplicación del archivo de configuración, se utilizará la última.

app.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

Programa.cs

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationSettings.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            string twoKeys = ConfigurationSettings.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

Leyendo AppSettings desde ConfigurationManager en .NET 2.0 y versiones posteriores

La clase ConfigurationManager es compatible con la propiedad `AppSettings` , que le permite continuar leyendo la configuración de la sección de configuración de la `appSettings` de un archivo de configuración de la misma manera que con .NET 1.x.

app.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

Programa.cs

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationManager.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            var twoKeys = ConfigurationManager.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

Introducción al soporte de configuración de usuario y aplicación fuertemente tipado de Visual Studio

Visual Studio ayuda a administrar la configuración de usuarios y aplicaciones. El uso de este enfoque tiene estos beneficios sobre el uso de la sección de `appSettings` de aplicaciones del archivo de configuración.

1. Los ajustes se pueden hacer fuertemente tipados. Cualquier tipo que pueda ser serializado puede usarse para un valor de configuración.

2. La configuración de la aplicación se puede separar fácilmente de la configuración del usuario. La configuración de la aplicación se almacena en un único archivo de configuración: `web.config` para sitios web y aplicaciones web, y app.config, renombrado como *ensamblado* .exe.config, donde *ensamblaje* es el nombre del ejecutable. La configuración del usuario (no utilizada por los proyectos web) se almacena en un archivo

`user.config` en la carpeta de datos de la aplicación del usuario (que varía con la versión del sistema operativo).

3. La configuración de la aplicación de las bibliotecas de clase se puede combinar en un solo archivo de configuración sin riesgo de colisiones de nombres, ya que cada biblioteca de clase puede tener su propia sección de configuración personalizada.

En la mayoría de los tipos de proyectos, el Diseñador de propiedades del proyecto tiene una pestaña Configuración, que es el punto de partida para crear aplicaciones personalizadas y configuraciones de usuario. Inicialmente, la pestaña Configuración estará en blanco, con un solo enlace para crear un archivo de configuración predeterminado. Al hacer clic en los resultados del enlace en estos cambios:

1. Si no existe un archivo de configuración (`app.config` o `web.config`) para el proyecto, se creará uno.

2. La pestaña Configuración se reemplazará con un control de cuadrícula que le permite crear, editar y eliminar entradas de configuración individuales.

3. En el Explorador de soluciones, se agrega un elemento de `Settings.settings` en la carpeta especial Propiedades. Al abrir este elemento se abrirá la pestaña Configuración.

4. Se agrega un nuevo archivo con una nueva clase parcial en la carpeta `Properties` en la carpeta del proyecto. Este nuevo archivo se llama `Settings.Designer.__` (.cs, .vb, etc.), y la clase se llama `Settings` . La clase es generada por código, por lo que no debe editarse, pero la clase es una clase parcial, por lo que puede extender la clase poniendo miembros adicionales en un archivo separado. Además, la clase se implementa utilizando el patrón Singleton, exponiendo la instancia de singleton con la propiedad denominada `Default` .

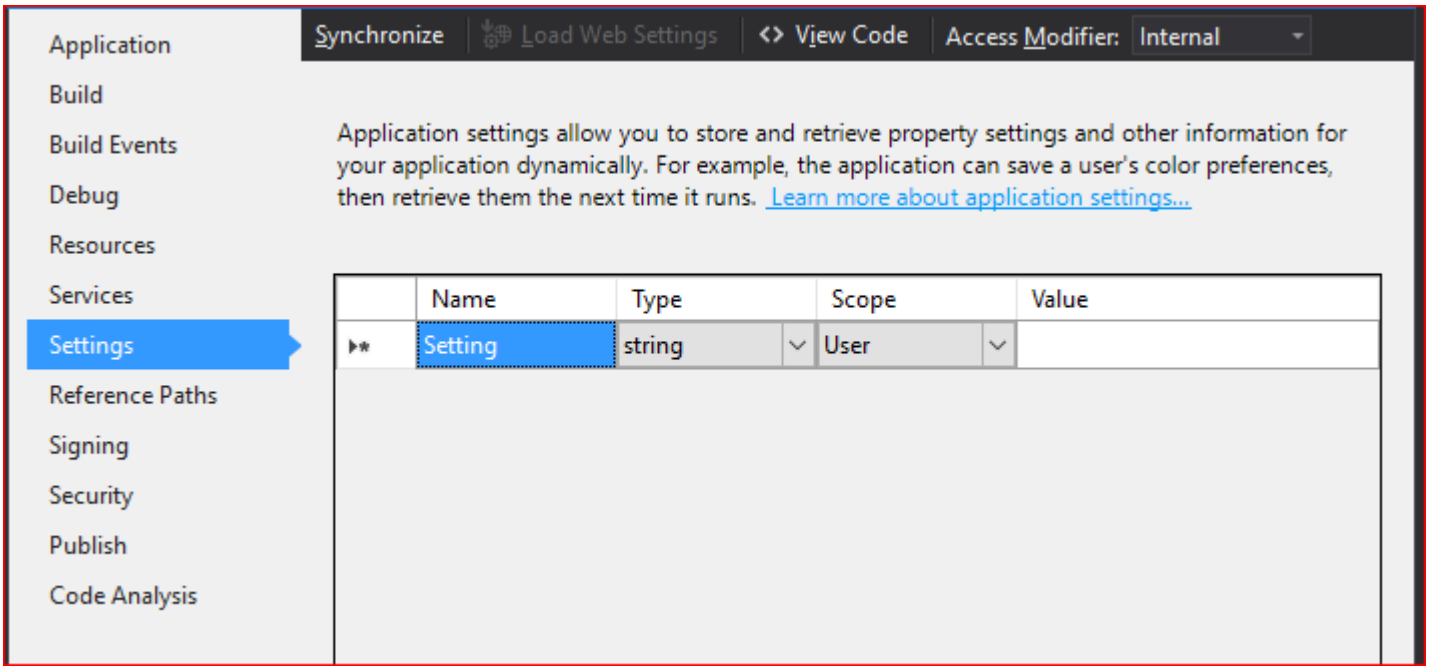
A medida que agrega cada nueva entrada a la pestaña Configuración, Visual Studio hace estas dos cosas:

1. Guarda la configuración en el archivo de configuración, en una sección de configuración personalizada diseñada para ser administrada por la clase Configuración.

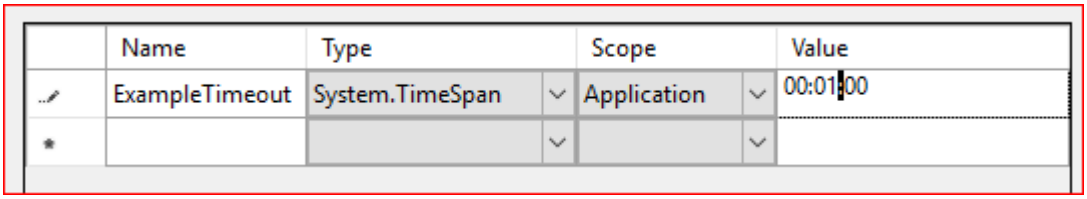
2. Crea un nuevo miembro en la clase Configuración para leer, escribir y presentar la configuración en el tipo específico seleccionado en la pestaña Configuración.

Lectura de configuraciones fuertemente tipadas de la sección personalizada del archivo de configuración

A partir de una nueva clase de configuración y una sección de configuración personalizada:



Agregue una configuración de aplicación llamada ExampleTimeout, usando la hora System.Timespan, y establezca el valor en 1 minuto:



Guarde las Propiedades del proyecto, que guarda las entradas de la pestaña Configuración, y vuelve a generar la clase de Configuración personalizada y actualiza el archivo de configuración del proyecto.

Use la configuración del código (C #):

Programa.cs

```
using System;
using System.Diagnostics;
using ConsoleApplication1.Properties;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            TimeSpan exampleTimeout = Settings.Default.ExampleTimeout;
            Debug.Assert(TimeSpan.FromMinutes(1).Equals(exampleTimeout));

            Console.ReadKey();
        }
    }
}
```

Debajo de las sábanas

Busque en el archivo de configuración del proyecto para ver cómo se ha creado la entrada de configuración de la aplicación:

app.config (Visual Studio actualiza esto automáticamente)

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="applicationSettings"
type="System.Configuration.ApplicationSettingsGroup, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" >
      <section name="ConsoleApplication1.Properties.Settings"
type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
    </sectionGroup>
  </configSections>
  <appSettings />
  <applicationSettings>
    <ConsoleApplication1.Properties.Settings>
      <setting name="ExampleTimeout" serializeAs="String">
        <value>00:01:00</value>
      </setting>
    </ConsoleApplication1.Properties.Settings>
  </applicationSettings>
</configuration>
```

Tenga en cuenta que la sección de `appSettings` no se utiliza. La sección `applicationSettings` contiene una sección personalizada calificada para el espacio de nombres que tiene un elemento de `setting` para cada entrada. El tipo de valor no se almacena en el archivo de configuración; Sólo es conocido por la clase de `Settings` .

Mire en la clase `Settings` para ver cómo usa la clase `ConfigurationManager` para leer esta sección personalizada.

Settings.designer.cs (para proyectos C #)

```
...
    [global::System.Configuration.ApplicationScopedSettingAttribute()]
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.Configuration.DefaultSettingValueAttribute("00:01:00")]
    public global::System.TimeSpan ExampleTimeout {
        get {
            return ((global::System.TimeSpan)(this["ExampleTimeout"]));
        }
    }
...
```

Observe que se creó un `DefaultSettingValueAttribute` para almacenar el valor ingresado en la pestaña Configuración del Diseñador de propiedades del proyecto. Si falta la entrada del archivo de configuración, en su lugar se usa este valor predeterminado.

Lea Ajustes en línea: https://riptutorial.com/es/dot-net/topic/54/ajustes

Capítulo 6: Análisis DateTime

Examples

ParseExact

```
var dateString = "2015-11-24";

var date = DateTime.ParseExact(dateString, "yyyy-MM-dd", null);
Console.WriteLine(date);
```

24/11/2015 12:00:00 a.m.

Tenga en cuenta que pasar `CultureInfo.CurrentCulture` como tercer parámetro es idéntico a pasar `null` . O bien, puede pasar una cultura específica.

Formato de cadenas

La cadena de entrada puede estar en cualquier formato que coincida con la cadena de formato

```
var date = DateTime.ParseExact("24|201511", "dd|yyyyMM", null);
Console.WriteLine(date);
```

24/11/2015 12:00:00 a.m.

Los caracteres que no son especificadores de formato se tratan como literales

```
var date = DateTime.ParseExact("2015|11|24", "yyyy|MM|dd", null);
Console.WriteLine(date);
```

24/11/2015 12:00:00 a.m.

El caso importa para los especificadores de formato

```
var date = DateTime.ParseExact("2015-01-24 11:11:30", "yyyy-mm-dd hh:MM:ss", null);
Console.WriteLine(date);
```

24/11/2015 11:01:30 AM

Tenga en cuenta que los valores de mes y minuto se analizaron en los destinos incorrectos.

Las cadenas de formato de un solo carácter deben ser uno de los formatos estándar.

```
var date = DateTime.ParseExact("11/24/2015", "d", new CultureInfo("en-US"));
var date = DateTime.ParseExact("2015-11-24T10:15:45", "s", null);
var date = DateTime.ParseExact("2015-11-24 10:15:45Z", "u", null);
```

Excepciones

ArgumentNullException

```
var date = DateTime.ParseExact(null, "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", null, null);
```

FormatException

```
var date = DateTime.ParseExact("", "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", "", null);
var date = DateTime.ParseExact("2015-0C-24", "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", "yyyy-QQ-dd", null);

// Single-character format strings must be one of the standard formats
var date = DateTime.ParseExact("2015-11-24", "q", null);

// Format strings must match the input exactly* (see next section)
var date = DateTime.ParseExact("2015-11-24", "d", null); // Expects 11/24/2015 or 24/11/2015
for most cultures
```

Manejando múltiples formatos posibles.

```
var date = DateTime.ParseExact("2015-11-24T10:15:45",
  new [] { "s", "t", "u", "yyyy-MM-dd" }, // Will succeed as long as input matches one of
these
  CultureInfo.CurrentCulture, DateTimeStyles.None);
```

Manejando las diferencias culturales.

```
var dateString = "10/11/2015";
var date = DateTime.ParseExact(dateString, "d", new CultureInfo("en-US"));
Console.WriteLine("Day: {0}; Month: {1}", date.Day, date.Month);
```

Día: 11; Mes: 10

```
date = DateTime.ParseExact(dateString, "d", new CultureInfo("en-GB"));
Console.WriteLine("Day: {0}; Month: {1}", date.Day, date.Month);
```

Día: 10; Mes: 11

TryParse

Este método acepta una cadena como entrada, intenta analizarla en un `DateTime` y devuelve un resultado booleano que indica éxito o fracaso. Si la llamada se realiza correctamente, la variable que se pasa como parámetro de `out` se completa con el resultado analizado.

Si el análisis falla, la variable que se pasa como parámetro de `out` se establece en el valor predeterminado, `DateTime.MinValue` .

TryParse (cadena, fecha y hora)

```
DateTime parsedValue;
```

```
if (DateTime.TryParse("monkey", out parsedValue))
{
   Console.WriteLine("Apparently, 'monkey' is a date/time value. Who knew?");
}
```

Este método intenta analizar la cadena de entrada según la configuración regional del sistema y los formatos conocidos, como ISO 8601 y otros formatos comunes.

```
DateTime.TryParse("11/24/2015 14:28:42", out parsedValue); // true
DateTime.TryParse("2015-11-24 14:28:42", out parsedValue); // true
DateTime.TryParse("2015-11-24T14:28:42", out parsedValue); // true
DateTime.TryParse("Sat, 24 Nov 2015 14:28:42", out parsedValue); // true
```

Dado que este método no acepta información de cultura, utiliza la configuración regional del sistema. Esto puede llevar a resultados inesperados.

```
// System set to en-US culture
bool result = DateTime.TryParse("24/11/2015", out parsedValue);
Console.WriteLine(result);
```

Falso

```
// System set to en-GB culture
bool result = DateTime.TryParse("11/24/2015", out parsedValue);
Console.WriteLine(result);
```

Falso

```
// System set to en-GB culture
bool result = DateTime.TryParse("10/11/2015", out parsedValue);
Console.WriteLine(result);
```

Cierto

Tenga en cuenta que si se encuentra en EE. UU., Le sorprenderá que el resultado analizado sea el 10 de noviembre, no el 11 de octubre.

TryParse (cadena, IFormatProvider, DateTimeStyles, out DateTime)

```
if (DateTime.TryParse(" monkey ", new CultureInfo("en-GB"),
    DateTimeStyles.AllowLeadingWhite | DateTimeStyles.AllowTrailingWhite, out parsedValue)
{
    Console.WriteLine("Apparently, ' monkey ' is a date/time value. Who knew?");
}
```

A diferencia de su método de hermanos, esta sobrecarga permite especificar una cultura y un estilo específicos. El paso `null` para el parámetro `IFormatProvider` utiliza la cultura del sistema.

Excepciones

Tenga en cuenta que es posible que este método arroje una excepción bajo ciertas condiciones. Estos se relacionan con los parámetros introducidos para esta sobrecarga: `IFormatProvider` y `DateTimeStyles` .

- `NotSupportedException` : `IFormatProvider` especifica una cultura neutral
- `ArgumentException` : `DateTimeStyles` no es una opción válida, o contiene indicadores incompatibles como `AssumeLocal` y `AssumeUniversal` .

TryParseExact

Este método se comporta como una combinación de `TryParse` y `ParseExact` : permite que se especifiquen formatos personalizados y devuelve un resultado booleano que indica el éxito o el fracaso en lugar de lanzar una excepción si el análisis falla.

TryParseExact (cadena, cadena, IFormatProvider, DateTimeStyles, out DateTime)

Esta sobrecarga intenta analizar la cadena de entrada en un formato específico. La cadena de entrada debe coincidir con ese formato para poder ser analizada.

```
DateTime.TryParseExact("11242015", "MMddyyyy", null, DateTimeStyles.None, out parsedValue); // true
```

TryParseExact (cadena, cadena [], IFormatProvider, DateTimeStyles, out DateTime)

Esta sobrecarga intenta analizar la cadena de entrada en una matriz de formatos. La cadena de entrada debe coincidir con al menos un formato para ser analizada.

```
DateTime.TryParseExact("11242015", new [] { "yyyy-MM-dd", "MMddyyyy" }, null,
DateTimeStyles.None, out parsedValue); // true
```

Lea Análisis DateTime en línea: https://riptutorial.com/es/dot-net/topic/58/analisis-datetime

Capítulo 7: Apilar y Montar

Observaciones

Vale la pena señalar que al declarar un tipo de referencia, su valor inicial será `null` . Esto se debe a que aún no apunta a una ubicación en la memoria, y es un estado perfectamente válido.
Sin embargo, con la excepción de los tipos anulables, los tipos de valor normalmente siempre deben tener un valor.

Examples

Tipos de valor en uso

Los tipos de valor simplemente contienen un ***valor*** .

Todos los tipos de valor se derivan de la clase System.ValueType , y esto incluye la mayoría de los tipos integrados.

Al crear un nuevo tipo de valor, se utiliza el área de memoria denominada ***pila*** .
La pila crecerá en consecuencia, según el tamaño del tipo declarado. Así, por ejemplo, a un int siempre se le asignarán 32 bits de memoria en la pila. Cuando el tipo de valor ya no está en el alcance, el espacio en la pila se desasignará.

El siguiente código muestra un tipo de valor que se asigna a una nueva variable. Se está utilizando una estructura como una forma conveniente de crear un tipo de valor personalizado (la clase System.ValueType no se puede ampliar de otra manera).

Lo importante a entender es que al asignar un tipo de valor, el valor se ***copia*** a la nueva variable, lo que significa que tenemos dos instancias distintas del objeto, que no pueden afectarse entre sí.

```
struct PersonAsValueType
{
    public string Name;
}

class Program
{
    static void Main()
    {
        PersonAsValueType personA;

        personA.Name = "Bob";

        var personB = personA;

        personA.Name = "Linda";

        Console.WriteLine(                // Outputs 'False' - because
            object.ReferenceEquals(       // personA and personB are referencing
                personA,                  // different areas of memory
```

```
                personB));

        Console.WriteLine(personA.Name);  // Outputs 'Linda'
        Console.WriteLine(personB.Name);  // Outputs 'Bob'
    }
}
```

Tipos de referencia en uso

Los tipos de referencia se componen de una ***referencia*** a un área de memoria y un ***valor*** almacenado dentro de esa área.
Esto es análogo a los punteros en C / C ++.

Todos los tipos de referencia se almacenan en lo que se conoce como ***el montón*** .
El montón es simplemente un área gestionada de memoria donde se almacenan los objetos. Cuando se crea una instancia de un nuevo objeto, una parte del montón se asignará para su uso por ese objeto, y se devolverá una referencia a esa ubicación del montón. El montón es administrado y mantenido por el *recolector de basura* , y no permite la intervención manual.

Además del espacio de memoria requerido para la instancia en sí, se requiere espacio adicional para almacenar la referencia en sí, junto con la información temporal adicional requerida por el CLR de .NET.

El siguiente código muestra un tipo de referencia que se asigna a una nueva variable. En este caso, estamos usando una clase, todas las clases son tipos de referencia (incluso si son estáticas).

Cuando se asigna un tipo de referencia a otra variable, es la ***referencia*** al objeto que se copia, **no** el valor en sí. Esta es una distinción importante entre los tipos de valor y los tipos de referencia.

Las implicaciones de esto son que ahora tenemos *dos* referencias al mismo objeto.
Cualquier cambio en los valores dentro de ese objeto será reflejado por ambas variables.

```
class PersonAsReferenceType
{
    public string Name;
}

class Program
{
    static void Main()
    {
        PersonAsReferenceType personA;

        personA = new PersonAsReferenceType { Name = "Bob" };

        var personB = personA;

        personA.Name = "Linda";

        Console.WriteLine(              // Outputs 'True' - because
            object.ReferenceEquals(     // personA and personB are referencing
                personA,                // the *same* memory location
```

```
                personB));

        Console.WriteLine(personA.Name); // Outputs 'Linda'
        Console.WriteLine(personB.Name); // Outputs 'Linda'
    }
```

Lea Apilar y Montar en línea: https://riptutorial.com/es/dot-net/topic/9358/apilar-y-montar

Capítulo 8: Arboles de expresion

Observaciones

Los árboles de expresiones son estructuras de datos que se utilizan para representar expresiones de código en .NET Framework. Se pueden generar por código y atravesar programáticamente para traducir el código a otro idioma o ejecutarlo. El generador más popular de Expression Trees es el compilador de C #. El compilador de C # puede generar árboles de expresión si una expresión lambda se asigna a una variable de tipo Expresión <Func <... >>. Normalmente esto sucede en el contexto de LINQ. El consumidor más popular es el proveedor LINQ de Entity Framework. Consume los árboles de expresión dados a Entity Framework y genera un código SQL equivalente que luego se ejecuta contra la base de datos.

Examples

Árbol de expresión simple generado por el compilador de C

Considere el siguiente código C #

```
Expression<Func<int, int>> expression = a => a + 1;
```

Debido a que el compilador de C # ve que la expresión lambda está asignada a un tipo de expresión en lugar de un tipo de delegado, genera un árbol de expresión aproximadamente equivalente a este código

```
ParameterExpression parameterA = Expression.Parameter(typeof(int), "a");
var expression = (Expression<Func<int, int>>)Expression.Lambda(
                                                Expression.Add(
                                                    parameterA,
                                                    Expression.Constant(1)),
                                                parameterA);
```

La raíz del árbol es la expresión lambda que contiene un cuerpo y una lista de parámetros. La lambda tiene 1 parámetro llamado "a". El cuerpo es una expresión única del tipo CLR BinaryExpression y NodeType of Add. Esta expresión representa la suma. Tiene dos subexpresiones denotadas como Izquierda y Derecha. La izquierda es la expresión de parámetro para el parámetro "a" y la derecha es una expresión constante con el valor 1.

El uso más simple de esta expresión es imprimirla:

```
Console.WriteLine(expression); //prints a => (a + 1)
```

Que imprime el código C # equivalente.

El árbol de expresiones se puede compilar en un delegado de C # y ejecutarlo CLR

```
Func<int, int> lambda = expression.Compile();
Console.WriteLine(lambda(2)); //prints 3
```

Por lo general, las expresiones se traducen a otros idiomas como SQL, pero también se pueden usar para invocar miembros privados, protegidos e internos de tipos públicos o no públicos como alternativa a Reflection.

construyendo un predicado de campo de formulario == valor

Para construir una expresión como `_ => _.Field == "VALUE"` en tiempo de ejecución.

Dado un predicado `_ => _.Field` y un valor de cadena `"VALUE"` , cree una expresión que compruebe si el predicado es verdadero o no.

La expresión es adecuada para:

- `IQueryable<T>` , `IEnumerable<T>` para probar el predicado.
- Entidad marco o `Linq` a `SQL` para crear una cláusula `Where` que prueba el predicado.

Este método construirá una expresión `Equal` apropiada que compruebe si el `Field` es o no igual a `"VALUE"` .

```
public static Expression<Func<T, bool>> BuildEqualPredicate<T>(
    Expression<Func<T, string>> memberAccessor,
    string term)
{
    var toString = Expression.Convert(Expression.Constant(term), typeof(string));
    Expression expression = Expression.Equal(memberAccessor.Body, toString);
    var predicate = Expression.Lambda<Func<T, bool>>(
        expression,
        memberAccessor.Parameters);
    return predicate;
}
```

El predicado se puede usar al incluir el predicado en un método de extensión `Where` .

```
var predicate = PredicateExtensions.BuildEqualPredicate<Entity>(
    _ => _.Field,
    "VALUE");
var results = context.Entity.Where(predicate).ToList();
```

Expresión para recuperar un campo estático

Teniendo un tipo de ejemplo como este:

```
public TestClass
{
    public static string StaticPublicField = "StaticPublicFieldValue";
}
```

Podemos recuperar el valor de StaticPublicField:

```
var fieldExpr = Expression.Field(null, typeof(TestClass), "StaticPublicField");
var labmda = Expression.Lambda<Func<string>>(fieldExpr);
```

Entonces puede ser compilado en un delegado para recuperar el valor del campo.

```
Func<string> retriever = lambda.Compile();
var fieldValue = retriever();
```

// el resultado de fieldValue es StaticPublicFieldValue

Clase de expresión de invocación

La clase InvocationExpression permite la invocación de otras expresiones lambda que forman parte del mismo árbol de expresiones.

Los creas con el método de `Expression.Invoke` estático.

Problema Queremos incluir los elementos que tienen "automóvil" en su descripción. Necesitamos verificarlo en nulo antes de buscar una cadena interna, pero no queremos que se llame en exceso, ya que el cálculo podría ser costoso.

```
using System;
using System.Linq;
using System.Linq.Expressions;

public class Program
{
    public static void Main()
    {
        var elements = new[] {
            new Element { Description = "car" },
            new Element { Description = "cargo" },
            new Element { Description = "wheel" },
            new Element { Description = null },
            new Element { Description = "Madagascar" },
        };

        var elementIsInterestingExpression = CreateSearchPredicate(
            searchTerm: "car",
            whereToSearch: (Element e) => e.Description);

        Console.WriteLine(elementIsInterestingExpression.ToString());

        var elementIsInteresting = elementIsInterestingExpression.Compile();
        var interestingElements = elements.Where(elementIsInteresting);
        foreach (var e in interestingElements)
        {
            Console.WriteLine(e.Description);
        }

        var countExpensiveComputations = 0;
        Action incCount = () => countExpensiveComputations++;
        elements
            .Where(
                CreateSearchPredicate(
                    "car",
```

```
                    (Element e) => ExpensivelyComputed(
                        e, incCount
                    )
                ).Compile()
            )
            .Count();

        Console.WriteLine("Property extractor is called {0} times.",
countExpensiveComputations);
    }

    private class Element
    {
        public string Description { get; set; }
    }

    private static string ExpensivelyComputed(Element source, Action count)
    {
        count();
        return source.Description;
    }

    private static Expression<Func<T, bool>> CreateSearchPredicate<T>(
            string searchTerm,
            Expression<Func<T, string>> whereToSearch)
    {
        var extracted = Expression.Parameter(typeof(string), "extracted");

        Expression<Func<string, bool>> coalesceNullCheckWithSearch =
            Expression.Lambda<Func<string, bool>>(
                Expression.AndAlso(
                    Expression.Not(
                        Expression.Call(typeof(string), "IsNullOrEmpty", null, extracted)
                    ),
                    Expression.Call(extracted, "Contains", null,
Expression.Constant(searchTerm))
                ),
                extracted);

        var elementParameter = Expression.Parameter(typeof(T), "element");

        return Expression.Lambda<Func<T, bool>>(
            Expression.Invoke(
                coalesceNullCheckWithSearch,
                Expression.Invoke(whereToSearch, elementParameter)
            ),
            elementParameter
        );
    }
}
```

Salida

```
element => Invoke(extracted => (Not(IsNullOrEmpty(extracted)) AndAlso
extracted.Contains("car")), Invoke(e => e.Description, element))
car
cargo
Madagascar
Predicate is called 5 times.
```

Lo primero que se debe tener en cuenta es cómo el acceso real a la propiedad, envuelto en una Invocación:

```
Invoke(e => e.Description, element)
```

, y esta es la única parte que toca `e.Description` , y en lugar de ello, el parámetro `extracted` de tipo `string` se pasa a la siguiente:

```
(Not(IsNullOrEmpty(extracted)) AndAlso extracted.Contains("car"))
```

Otra cosa importante a tener en cuenta aquí es `AndAlso` . Calcula solo la parte izquierda, si la primera parte devuelve "falso". Es un error común utilizar el operador bit a bit 'And' en lugar de este, que siempre calcula ambas partes y fallaría con una NullReferenceException en este ejemplo.

Lea Arboles de expresion en línea: https://riptutorial.com/es/dot-net/topic/2657/arboles-de-expresion

Capítulo 9: Archivo de entrada / salida

Parámetros

Parámetro	Detalles
camino de cadena	Ruta del archivo a comprobar. (relativo o totalmente calificado)

Observaciones

Devuelve verdadero si el archivo existe, falso de lo contrario.

Examples

VB WriteAllText

```
Imports System.IO

Dim filename As String = "c:\path\to\file.txt"
File.WriteAllText(filename, "Text to write" & vbCrLf)
```

VB StreamWriter

```
Dim filename As String = "c:\path\to\file.txt"
If System.IO.File.Exists(filename) Then
    Dim writer As New System.IO.StreamWriter(filename)
    writer.Write("Text to write" & vbCrLf) 'Add a newline
    writer.close()
End If
```

C # StreamWriter

```
using System.Text;
using System.IO;

string filename = "c:\path\to\file.txt";
//'using' structure allows for proper disposal of stream.
using (StreamWriter writer = new StreamWriter(filename"))
{
    writer.WriteLine("Text to Write\n");
}
```

C # WriteAllText ()

```
using System.IO;
using System.Text;
```

```
string filename = "c:\path\to\file.txt";
File.writeAllText(filename, "Text to write\n");
```

C # File.Exists ()

```
using System;
using System.IO;

public class Program
{
    public static void Main()
    {
        string filePath = "somePath";

        if(File.Exists(filePath))
        {
            Console.WriteLine("Exists");
        }
        else
        {
            Console.WriteLine("Does not exist");
        }
    }
}
```

También se puede utilizar en un operador ternario.

```
Console.WriteLine(File.Exists(pathToFile) ? "Exists" : "Does not exist");
```

Lea Archivo de entrada / salida en línea: https://riptutorial.com/es/dot-net/topic/1376/archivo-de-entrada---salida

Capítulo 10: Biblioteca paralela de tareas (TPL)

Observaciones

Propósito y casos de uso

El propósito de la biblioteca paralela de tareas es simplificar el proceso de escritura y mantenimiento de código multiproceso y paralelo.

Algunos casos de uso *:

- Mantener una interfaz de usuario sensible ejecutando el trabajo en segundo plano en una tarea separada
- Distribuyendo carga de trabajo
- Permitir que una aplicación cliente envíe y reciba solicitudes al mismo tiempo (resto, TCP / UDP, ect)
- Leyendo y / o escribiendo múltiples archivos a la vez

* El código debe considerarse caso por caso para multiproceso. Por ejemplo, si un bucle solo tiene unas pocas iteraciones o solo una pequeña parte del trabajo, la sobrecarga para el paralelismo puede superar los beneficios.

TPL con .Net 3.5

El TPL también está disponible para .Net 3.5 incluido en un paquete NuGet, se llama Task Parallel Library.

Examples

Bucle básico productor-consumidor (BlockingCollection)

```
var collection = new BlockingCollection<int>(5);
var random = new Random();

var producerTask = Task.Run(() => {
    for(int item=1; item<=10; item++)
    {
        collection.Add(item);
        Console.WriteLine("Produced: " + item);
        Thread.Sleep(random.Next(10,1000));
    }
    collection.CompleteAdding();
    Console.WriteLine("Producer completed!");
});
```

Vale la pena señalar que si no llama a `collection.CompleteAdding();` , puede seguir agregando a la colección incluso si su tarea de consumidor se está ejecutando. Simplemente llame a `collection.CompleteAdding();` Cuando estés seguro no hay más adiciones. Esta funcionalidad se puede usar para hacer un patrón de Productor múltiple a un Consumidor único en el que tenga múltiples fuentes que alimenten elementos en la Colección de Bloqueo y un consumidor individual extraiga los elementos y haga algo con ellos. Si su BlockingCollection está vacía antes de completar la adición, Enumerable from `collection.GetConsumingEnumerable()` se bloqueará hasta que se agregue un nuevo elemento a la colección o BlockingCollection.CompleteAdding (); se llama y la cola está vacía.

```
var consumerTask = Task.Run(() => {
    foreach(var item in collection.GetConsumingEnumerable())
    {
        Console.WriteLine("Consumed: " + item);
        Thread.Sleep(random.Next(10,1000));
    }
    Console.WriteLine("Consumer completed!");
});

Task.WaitAll(producerTask, consumerTask);

Console.WriteLine("Everything completed!");
```

Tarea: instanciación básica y espera.

Se puede crear una tarea directamente creando una instancia de la clase de `Task` ...

```
var task = new Task(() =>
{
    Console.WriteLine("Task code starting...");
    Thread.Sleep(2000);
    Console.WriteLine("...task code ending!");
});

Console.WriteLine("Starting task...");
task.Start();
task.Wait();
Console.WriteLine("Task completed!");
```

... o usando el método estático `Task.Run` :

```
Console.WriteLine("Starting task...");
var task = Task.Run(() =>
{
    Console.WriteLine("Task code starting...");
    Thread.Sleep(2000);
    Console.WriteLine("...task code ending!");
});
task.Wait();
Console.WriteLine("Task completed!");
```

Tenga en cuenta que solo en el primer caso es necesario invocar explícitamente el `Start` .

Tarea: WaitAll y captura de variables.

```
var tasks = Enumerable.Range(1, 5).Select(n => new Task<int>(() =>
{
    Console.WriteLine("I'm task " + n);
    return n;
})).ToArray();

foreach(var task in tasks) task.Start();
Task.WaitAll(tasks);

foreach(var task in tasks)
    Console.WriteLine(task.Result);
```

Tarea: WaitAny

```
var allTasks = Enumerable.Range(1, 5).Select(n => new Task<int>(() => n)).ToArray();
var pendingTasks = allTasks.ToArray();

foreach(var task in allTasks) task.Start();

while(pendingTasks.Length > 0)
{
    var finishedTask = pendingTasks[Task.WaitAny(pendingTasks)];
    Console.WriteLine("Task {0} finished", finishedTask.Result);
    pendingTasks = pendingTasks.Except(new[] {finishedTask}).ToArray();
}

Task.WaitAll(allTasks);
```

Nota: el `WaitAll` final es necesario porque `WaitAny` no hace que se observen excepciones.

Tarea: manejo de excepciones (usando espera)

```
var task1 = Task.Run(() =>
{
    Console.WriteLine("Task 1 code starting...");
    throw new Exception("Oh no, exception from task 1!!");
});

var task2 = Task.Run(() =>
{
    Console.WriteLine("Task 2 code starting...");
    throw new Exception("Oh no, exception from task 2!!");
});

Console.WriteLine("Starting tasks...");
try
{
    Task.WaitAll(task1, task2);
}
catch(AggregateException ex)
{
    Console.WriteLine("Task(s) failed!");
    foreach(var inner in ex.InnerExceptions)
        Console.WriteLine(inner.Message);
```

```
}

Console.WriteLine("Task 1 status is: " + task1.Status); //Faulted
Console.WriteLine("Task 2 status is: " + task2.Status); //Faulted
```

Tarea: manejo de excepciones (sin usar Espera)

```
var task1 = Task.Run(() =>
{
    Console.WriteLine("Task 1 code starting...");
    throw new Exception("Oh no, exception from task 1!!");
});

var task2 = Task.Run(() =>
{
    Console.WriteLine("Task 2 code starting...");
    throw new Exception("Oh no, exception from task 2!!");
});

var tasks = new[] {task1, task2};

Console.WriteLine("Starting tasks...");
while(tasks.All(task => !task.IsCompleted));

foreach(var task in tasks)
{
    if(task.IsFaulted)
        Console.WriteLine("Task failed: " +
            task.Exception.InnerExceptions.First().Message);
}

Console.WriteLine("Task 1 status is: " + task1.Status); //Faulted
Console.WriteLine("Task 2 status is: " + task2.Status); //Faulted
```

Tarea: cancelar usando CancelToken

```
var cancellationTokenSource = new CancellationTokenSource();
var cancellationToken = cancellationTokenSource.Token;

var task = new Task((state) =>
    {
        int i = 1;
        var myCancellationToken = (CancellationToken)state;
        while(true)
        {
            Console.Write("{0} ", i++);
            Thread.Sleep(1000);
            myCancellationToken.ThrowIfCancellationRequested();
        }
    },
    cancellationToken: cancellationToken,
    state: cancellationToken);

Console.WriteLine("Counting to infinity. Press any key to cancel!");
task.Start();
Console.ReadKey();
```

```
cancellationTokenSource.Cancel();
try
{
    task.Wait();
}
catch(AggregateException ex)
{
    ex.Handle(inner => inner is OperationCanceledException);
}

Console.WriteLine($"{Environment.NewLine}You have cancelled! Task status is: {task.Status}");
//Canceled
```

Como alternativa a `ThrowIfCancellationRequested` , la solicitud de cancelación se puede detectar con `IsCancellationRequested` y se puede lanzar una `OperationCanceledException` manualmente:

```
//New task delegate
int i = 1;
var myCancellationToken = (CancellationToken)state;
while(!myCancellationToken.IsCancellationRequested)
{
    Console.Write("{0} ", i++);
    Thread.Sleep(1000);
}
Console.WriteLine($"{Environment.NewLine}Ouch, I have been cancelled!!");
throw new OperationCanceledException(myCancellationToken);
```

Observe cómo el token de cancelación se pasa al constructor de tareas en el parámetro `cancellationToken` . Esto es necesario para que la tarea pase al estado `Canceled` , no al estado Con `Faulted` , cuando se invoca `ThrowIfCancellationRequested` . Además, por el mismo motivo, el token de cancelación se proporciona explícitamente en el constructor de `OperationCanceledException` en el segundo caso.

Tarea.cuando

```
var random = new Random();
IEnumerable<Task<int>> tasks = Enumerable.Range(1, 5).Select(n => Task.Run(async() =>
{
    Console.WriteLine("I'm task " + n);
    await Task.Delay(random.Next(10,1000));
    return n;
}));

Task<Task<int>> whenAnyTask = Task.WhenAny(tasks);
Task<int> completedTask = await whenAnyTask;
Console.WriteLine("The winner is: task " + await completedTask);

await Task.WhenAll(tasks);
Console.WriteLine("All tasks finished!");
```

Tarea.Cuando

```
var random = new Random();
IEnumerable<Task<int>> tasks = Enumerable.Range(1, 5).Select(n => Task.Run(() =>
```

```
{
    Console.WriteLine("I'm task " + n);
    return n;
}));

Task<int[]> task = Task.WhenAll(tasks);
int[] results = await task;

Console.WriteLine(string.Join(",", results.Select(n => n.ToString())));
// Output: 1,2,3,4,5
```

Paralelo.Invocar

```
var actions = Enumerable.Range(1, 10).Select(n => new Action(() =>
{
    Console.WriteLine("I'm task " + n);
    if((n & 1) == 0)
        throw new Exception("Exception from task " + n);
})).ToArray();

try
{
    Parallel.Invoke(actions);
}
catch(AggregateException ex)
{
    foreach(var inner in ex.InnerExceptions)
        Console.WriteLine("Task failed: " + inner.Message);
}
```

Paralelo.para cada

Este ejemplo utiliza `Parallel.ForEach` para calcular la suma de los números entre 1 y 10000 mediante el uso de varios subprocesos. Para lograr la seguridad de subprocesos, `Interlocked.Add` se utiliza para sumar los números.

```
using System.Threading;

int Foo()
{
    int total = 0;
    var numbers = Enumerable.Range(1, 10000).ToList();
    Parallel.ForEach(numbers,
        () => 0, // initial value,
        (num, state, localSum) => num + localSum,
        localSum => Interlocked.Add(ref total, localSum));
    return total; // total = 50005000
}
```

Paralelo.para

Este ejemplo utiliza `Parallel.For` para calcular la suma de los números entre 1 y 10000 mediante el uso de varios subprocesos. Para lograr la seguridad de subprocesos, `Interlocked.Add` se utiliza para sumar los números.

```
using System.Threading;

int Foo()
{
    int total = 0;
    Parallel.For(1, 10001,
        () => 0, // initial value,
        (num, state, localSum) => num + localSum,
        localSum => Interlocked.Add(ref total, localSum));
    return total; // total = 50005000
}
```

Contexto de ejecución fluida con AsyncLocal

Cuando necesite pasar algunos datos de la tarea principal a sus tareas `AsyncLocal` , para que fluya lógicamente con la ejecución, use la clase `AsyncLocal` :

```
void Main()
{
    AsyncLocal<string> user = new AsyncLocal<string>();
    user.Value = "initial user";

    // this does not affect other tasks - values are local relative to the branches of
execution flow
    Task.Run(() => user.Value = "user from another task");

    var task1 = Task.Run(() =>
    {
        Console.WriteLine(user.Value); // outputs "initial user"
        Task.Run(() =>
        {
            // outputs "initial user" - value has flown from main method to this task without
being changed
            Console.WriteLine(user.Value);
        }).Wait();

        user.Value = "user from task1";

        Task.Run(() =>
        {
            // outputs "user from task1" - value has flown from main method to task1
            // than value was changed and flown to this task.
            Console.WriteLine(user.Value);
        }).Wait();
    });

    task1.Wait();

    // ouputs "initial user" - changes do not propagate back upstream the execution flow
    Console.WriteLine(user.Value);
}
```

Nota: Como puede verse en el ejemplo anterior, `AsynLocal.Value` tiene `copy on read` semántica, pero si fluye algún tipo de referencia y cambia sus propiedades, afectará a otras tareas. Por lo tanto, la mejor práctica con `AsyncLocal` es usar tipos de valor o tipos inmutables.

Parallel.ForEach en VB.NET

```
For Each row As DataRow In FooDataTable.Rows
    Me.RowsToProcess.Add(row)
Next

Dim myOptions As ParallelOptions = New ParallelOptions()
myOptions.MaxDegreeOfParallelism = environment.processorcount

Parallel.ForEach(RowsToProcess, myOptions, Sub(currentRow, state)
                                               ProcessRowParallel(currentRow, state)
                                           End Sub)
```

Tarea: Devolver un valor

La tarea que devuelve un valor tiene un tipo de retorno `Task< TResult >` donde TResult es el tipo de valor que debe devolverse. Puede consultar el resultado de una tarea por su propiedad Resultado.

```
Task<int> t = Task.Run(() =>
    {
        int sum = 0;

        for(int i = 0; i < 500; i++)
            sum += i;

        return sum;
    });

Console.WriteLine(t.Result); // Outuput 124750
```

Si la Tarea se ejecuta de forma asíncrona, la espera de la Tarea devuelve su resultado.

```
public async Task DoSomeWork()
{
    WebClient client = new WebClient();
    // Because the task is awaited, result of the task is assigned to response
    string response = await client.DownloadStringTaskAsync("http://somedomain.com");
}
```

Lea Biblioteca paralela de tareas (TPL) en línea: https://riptutorial.com/es/dot-net/topic/55/biblioteca-paralela-de-tareas--tpl-

Capítulo 11: Cargar archivo y datos POST al servidor web

Examples

Subir archivo con WebRequest

Para enviar un archivo y datos de formulario en una sola solicitud, el contenido debe tener un tipo de datos multiparte / formulario .

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Threading.Tasks;

public async Task<string> UploadFile(string url, string filename,
    Dictionary<string, object> postData)
{
    var request = WebRequest.CreateHttp(url);
    var boundary = $"{Guid.NewGuid():N}"; // boundary will separate each parameter
    request.ContentType = $"multipart/form-data; {nameof(boundary)}={boundary}";
    request.Method = "POST";

    using (var requestStream = request.GetRequestStream())
    using (var writer = new StreamWriter(requestStream))
    {
        foreach (var data in postData)
            await writer.WriteAsync( // put all POST data into request
                $"\r\n--{boundary}\r\nContent-Disposition: " +
                $"form-data; name=\"{data.Key}\"\r\n\r\n{data.Value}");

        await writer.WriteAsync( // file header
            $"\r\n--{boundary}\r\nContent-Disposition: " +
            $"form-data; name=\"File\"; filename=\"{Path.GetFileName(filename)}\"\r\n" +
            "Content-Type: application/octet-stream\r\n\r\n");

        await writer.FlushAsync();
        using (var fileStream = File.OpenRead(filename))
            await fileStream.CopyToAsync(requestStream);

        await writer.WriteAsync($"\r\n--{boundary}--\r\n");
    }

    using (var response = (HttpWebResponse) await request.GetResponseAsync())
    using (var responseStream = response.GetResponseStream())
    {
        if (responseStream == null)
            return string.Empty;
        using (var reader = new StreamReader(responseStream))
            return await reader.ReadToEndAsync();
    }
}
```

Uso:

```
var response = await uploader.UploadFile("< YOUR URL >", "< PATH TO YOUR FILE >",
    new Dictionary<string, object>
    {
        {"Comment", "test"},
        {"Modified", DateTime.Now }
    });
```

Lea Cargar archivo y datos POST al servidor web en línea: https://riptutorial.com/es/dot-net/topic/10845/cargar-archivo-y-datos-post-al-servidor-web

Capítulo 12: Clase System.IO.File

Sintaxis

- fuente de cadena
- cadena de destino;

Parámetros

Parámetro	Detalles
`source`	El archivo que se va a mover a otra ubicación.
`destination`	El directorio al que desea mover la `source` (esta variable también debe contener el nombre (y la extensión del archivo) del archivo).

Examples

Borrar un archivo

Eliminar un archivo (si tiene permisos requeridos) es tan simple como:

```
File.Delete(path);
```

Sin embargo, muchas cosas pueden ir mal:

- No tiene los permisos necesarios (se `UnauthorizedAccessException`).
- El archivo puede estar en uso por otra persona (se produce una `IOException`).
- El archivo no se puede eliminar debido a un error de bajo nivel o los medios son de solo lectura (se produce una `IOException`).
- El archivo ya no existe (se lanza `IOException`).

Tenga en cuenta que el último punto (el archivo no existe) generalmente se *evita* con un fragmento de código como este:

```
if (File.Exists(path))
    File.Delete(path);
```

Sin embargo, no es una operación atómica y el archivo puede ser eliminado por otra persona entre la llamada a `File.Exists()` y antes de `File.Delete()` . El enfoque correcto para manejar la operación de E / S requiere un manejo de excepciones (asumiendo que se puede tomar un curso alternativo de acciones cuando falla la operación):

```
if (File.Exists(path))
```

```
{
    try
    {
        File.Delete(path);
    }
    catch (IOException exception)
    {
        if (!File.Exists(path))
            return; // Someone else deleted this file

        // Something went wrong...
    }
    catch (UnauthorizedAccessException exception)
    {
        // I do not have required permissions
    }
}
```

Tenga en cuenta que estos errores de E / S a veces son transitorios (archivo en uso, por ejemplo) y si se trata de una conexión de red, puede recuperarse automáticamente sin ninguna acción por nuestra parte. Entonces es común *volver* a *intentar* una operación de E / S varias veces con un pequeño retraso entre cada intento:

```
public static void Delete(string path)
{
    if (!File.Exists(path))
        return;

    for (int i=1; ; ++i)
    {
        try
        {
            File.Delete(path);
            return;
        }
        catch (IOException e)
        {
            if (!File.Exists(path))
                return;

            if (i == NumberOfAttempts)
                throw;

            Thread.Sleep(DelayBetweenEachAttempt);
        }

        // You may handle UnauthorizedAccessException but this issue
        // will probably won't be fixed in few seconds...
    }
}

private const int NumberOfAttempts = 3;
private const int DelayBetweenEachAttempt = 1000; // ms
```

Nota: en el entorno de Windows, el archivo no se eliminará realmente cuando llame a esta función, si alguien más abre el archivo con `FileShare.Delete` , el archivo se puede eliminar, pero solo ocurrirá cuando el propietario cierre el archivo.

Eliminar líneas no deseadas de un archivo de texto

Cambiar un archivo de texto no es fácil porque su contenido debe moverse. Para *los* archivos *pequeños* , el método más sencillo es leer su contenido en la memoria y luego volver a escribir el texto modificado.

En este ejemplo leemos todas las líneas de un archivo y soltamos todas las líneas en blanco, luego escribimos de nuevo en la ruta original:

```
File.WriteAllLines(path,
    File.ReadAllLines(path).Where(x => !String.IsNullOrWhiteSpace(x)));
```

Si el archivo es demasiado grande para cargarlo en la memoria y la ruta de salida es diferente de la ruta de entrada:

```
File.WriteAllLines(outputPath,
    File.ReadLines(inputPath).Where(x => !String.IsNullOrWhiteSpace(x)));
```

Convertir codificación de archivos de texto

El texto se guarda codificado (consulte también el tema Cadenas). A veces, es posible que deba cambiar su codificación. En este ejemplo se supone (para simplificar) que el archivo no es demasiado grande y se puede leer por completo en la memoria:

```
public static void ConvertEncoding(string path, Encoding from, Encoding to)
{
    File.WriteAllText(path, File.ReadAllText(path, from), to);
}
```

Al realizar conversiones, no olvide que el archivo puede contener BOM (Byte Order Mark), para comprender mejor cómo se gestiona, consulte Encoding.UTF8.GetString no tiene en cuenta el Preámbulo / BOM .

"Toque" una gran cantidad de archivos (para actualizar el último tiempo de escritura)

Este ejemplo actualiza el último tiempo de escritura de una gran cantidad de archivos (usando `System.IO.Directory.EnumerateFiles` lugar de `System.IO.Directory.GetFiles()`). Opcionalmente, puede especificar un patrón de búsqueda (el valor predeterminado es `"*.*"` Y, finalmente, buscar en un árbol de directorios (no solo el directorio especificado):

```
public static void Touch(string path,
                         string searchPattern = "*.*",
                         SearchOptions options = SearchOptions.None)
{
    var now = DateTime.Now;

    foreach (var filePath in Directory.EnumerateFiles(path, searchPattern, options))
    {
```

```
        File.SetLastWriteTime(filePath, now);
    }
}
```

Enumerar archivos anteriores a una cantidad especificada

Este fragmento de código es una función auxiliar para enumerar todos los archivos anteriores a una edad especificada, es útil, por ejemplo, cuando tiene que eliminar archivos de registro antiguos o datos en caché antiguos.

```
static IEnumerable<string> EnumerateAllFilesOlderThan(
                                TimeSpan maximumAge,
                                string path,
                                string searchPattern = "*.*",
                                SearchOption options = SearchOption.TopDirectoryOnly)
{
    DateTime oldestWriteTime = DateTime.Now - maximumAge;

    return Directory.EnumerateFiles(path, searchPattern, options)
        .Where(x => Directory.GetLastWriteTime(x) < oldestWriteTime);
}
```

Utilizado de esta manera:

```
var oldFiles = EnumerateAllFilesOlderThan(TimeSpan.FromDays(7), @"c:\log", "*.log");
```

Algunas cosas a tener en cuenta:

- La búsqueda se realiza utilizando `Directory.EnumerateFiles()` lugar de `Directory.GetFiles()` . La enumeración está *activa* , por lo que no tendrá que esperar hasta que se hayan recuperado todas las entradas del sistema de archivos.
- Estamos comprobando el último tiempo de escritura, pero puede usar el tiempo de creación o el último tiempo de acceso (por ejemplo, para eliminar los archivos en caché *no utilizados* , tenga en cuenta que el tiempo de acceso puede estar deshabilitado).
- La granularidad no es uniforme para todas esas propiedades (tiempo de escritura, tiempo de acceso, tiempo de creación), consulte MSDN para obtener detalles sobre esto.

Mueve un archivo de una ubicación a otra

Archivo.Mover

Para mover un archivo de una ubicación a otra, una simple línea de código puede lograr esto:

```
File.Move(@"C:\TemporaryFile.txt", @"C:\TemporaryFiles\TemporaryFile.txt");
```

Sin embargo, hay muchas cosas que podrían salir mal con esta simple operación. Por ejemplo, ¿qué sucede si el usuario que ejecuta su programa no tiene una unidad que tiene la etiqueta 'C'? ¿Qué pasaría si lo hicieran, pero decidieron cambiarle el nombre a 'B' o 'M'?

¿Qué sucede si el archivo de origen (el archivo en el que desea mover) se ha movido sin que usted lo sepa, o si simplemente no existe?

Esto puede evitarse comprobando primero si el archivo fuente existe:

```
string source = @"C:\TemporaryFile.txt", destination = @"C:\TemporaryFiles\TemporaryFile.txt";
if(File.Exists("C:\TemporaryFile.txt"))
{
    File.Move(source, destination);
}
```

Esto asegurará que, en ese mismo momento, el archivo exista y se pueda mover a otra ubicación. Puede haber ocasiones en que una simple llamada a `File.Exists` no sea suficiente. Si no es así, verifique nuevamente, transmita al usuario que la operación falló, o maneje la excepción.

Una `FileNotFoundException` no es la única excepción que es probable que encuentre.

Vea abajo para posibles excepciones:

Tipo de excepción	Descripción
`IOException`	El archivo ya existe o el archivo de origen no se pudo encontrar.
`ArgumentNullException`	El valor de los parámetros Origen y / o Destino es nulo.
`ArgumentException`	El valor de los parámetros Origen y / o Destino están vacíos o contienen caracteres no válidos.
`UnauthorizedAccessException`	No tiene los permisos necesarios para realizar esta acción.
`PathTooLongException`	La Fuente, el Destino o las rutas especificadas exceden la longitud máxima. En Windows, la longitud de una ruta debe tener menos de 248 caracteres, mientras que los nombres de los archivos deben tener menos de 260 caracteres.
`DirectoryNotFoundException`	No se pudo encontrar el directorio especificado.
`NotSupportedException`	Las rutas de origen o destino o los nombres de los archivos tienen un formato no válido.

Lea Clase System.IO.File en línea: https://riptutorial.com/es/dot-net/topic/5395/clase-system-io-file

Capítulo 13: Clientes HTTP

Observaciones

Los RFC HTTP / 1.1 actualmente relevantes son:

- 7230: Sintaxis de mensajes y enrutamiento
- 7231: Semántica y Contenido
- 7232: Peticiones Condicionales
- 7233: solicitudes de rango
- 7234: almacenamiento en caché
- 7235: Autentica
- 7239: Extensión HTTP reenviada
- 7240: Prefiere el encabezado para HTTP

También están los siguientes RFCs informativos:

- 7236: Registros de Esquemas de Autentificación
- 7237: Registros de Método

Y el experimental RFC:

- 7238: Código de estado del protocolo de transferencia de hipertexto 308 (Redireccionamiento permanente)

Protocolos relacionados:

- 4918: Extensiones HTTP para la creación y control de versiones distribuidas en la web (WebDAV)
- 4791: Extensiones de calendario a WebDAV (CalDAV)

Examples

Leyendo la respuesta GET como una cadena usando System.Net.HttpWebRequest

```
string requestUri = "http://www.example.com";
string responseData;

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(parameters.Uri);
WebResponse response = request.GetResponse();

using (StreamReader responseReader = new StreamReader(response.GetResponseStream()))
{
    responseData = responseReader.ReadToEnd();
}
```

Leyendo la respuesta GET como una cadena usando System.Net.WebClient

```
string requestUri = "http://www.example.com";
string responseData;

using (var client = new WebClient())
{
    responseData = client.DownloadString(requestUri);
}
```

Leyendo la respuesta GET como una cadena usando System.Net.HttpClient

`HttpClient` está disponible a través de NuGet: Microsoft HTTP Client Libraries .

```
string requestUri = "http://www.example.com";
string responseData;

using (var client = new HttpClient())
{
    using(var response = client.GetAsync(requestUri).Result)
    {
       response.EnsureSuccessStatusCode();
       responseData = response.Content.ReadAsStringAsync().Result;
    }
}
```

Enviar una solicitud POST con una carga útil de cadena utilizando System.Net.HttpWebRequest

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri)
{
  Method = requestMethod,
  ContentType = contentType,
};

byte[] bytes = Encoding.UTF8.GetBytes(requestBodyString);
Stream stream = request.GetRequestStream();
stream.Write(bytes, 0, bytes.Length);
stream.Close();

HttpWebResponse response = (HttpWebResponse)request.GetResponse();
```

Enviar una solicitud POST con una carga útil de cadena utilizando System.Net.WebClient

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
```

```
string requestMethod = "POST";

byte[] responseBody;
byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBodyString);

using (var client = new WebClient())
{
    client.Headers[HttpRequestHeader.ContentType] = contentType;
    responseBody = client.UploadData(requestUri, requestMethod, requestBodyBytes);
}
```

Enviar una solicitud POST con una carga útil de cadena utilizando System.Net.HttpClient

`HttpClient` está disponible a través de NuGet: Microsoft HTTP Client Libraries .

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

var request = new HttpRequestMessage
{
    RequestUri = requestUri,
    Method = requestMethod,
};

byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBodyString);
request.Content = new ByteArrayContent(requestBodyBytes);

request.Content.Headers.ContentType = new MediaTypeHeaderValue(contentType);

HttpResponseMessage result = client.SendAsync(request).Result;
result.EnsureSuccessStatusCode();
```

Descargador HTTP básico utilizando System.Net.Http.HttpClient

```
using System;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

class HttpGet
{
    private static async Task DownloadAsync(string fromUrl, string toFile)
    {
        using (var fileStream = File.OpenWrite(toFile))
        {
            using (var httpClient = new HttpClient())
            {
                Console.WriteLine("Connecting...");
                using (var networkStream = await httpClient.GetStreamAsync(fromUrl))
                {
                    Console.WriteLine("Downloading...");
                    await networkStream.CopyToAsync(fileStream);
                    await fileStream.FlushAsync();
```

```
                }
            }
        }
    }

    static void Main(string[] args)
    {
        try
        {
            Run(args).Wait();
        }
        catch (Exception ex)
        {
            if (ex is AggregateException)
                ex = ((AggregateException)ex).Flatten().InnerExceptions.First();

            Console.WriteLine("--- Error: " +
                (ex.InnerException?.Message ?? ex.Message));
        }
    }
    static async Task Run(string[] args)
    {
        if (args.Length < 2)
        {
            Console.WriteLine("Basic HTTP downloader");
            Console.WriteLine();
            Console.WriteLine("Usage: httpget <url>[<:port>] <file>");
            return;
        }

        await DownloadAsync(fromUrl: args[0], toFile: args[1]);

        Console.WriteLine("Done!");
    }
}
```

Lea Clientes HTTP en línea: https://riptutorial.com/es/dot-net/topic/32/clientes-http

Capítulo 14: CLR

Examples

Una introducción a Common Language Runtime

El **Common Language Runtime (CLR)** es un entorno de máquina virtual y parte de .NET Framework. Contiene:

- Un lenguaje de bytecode portátil llamado **Common Intermediate Language** (CIL abreviado, o IL)
- Un compilador Just-In-Time que genera código de máquina
- Un recolector de basura de rastreo que proporciona administración de memoria automática
- Soporte para subprocesos ligeros llamados AppDomains
- Mecanismos de seguridad a través de los conceptos de código verificable y niveles de confianza.

El código que se ejecuta en el CLR se conoce como *código administrado* para distinguirlo del código que se ejecuta fuera del CLR (generalmente código nativo) que se conoce como *código no administrado* . Existen varios mecanismos que facilitan la interoperabilidad entre el código administrado y el no administrado.

Lea CLR en línea: https://riptutorial.com/es/dot-net/topic/3942/clr

Capítulo 15: Colecciones

Observaciones

Hay varios tipos de colección:

- `Array`
- `List`
- `Queue`
- `SortedList`
- `Stack`
- Diccionario

Examples

Creando una lista inicializada con tipos personalizados

```
public class Model
{
    public string Name { get; set; }
    public bool? Selected { get; set; }
}
```

Aquí tenemos una Clase sin constructor con dos propiedades: `Name` y una propiedad booleana anulable `Selected` . Si quisiéramos inicializar una `List<Model>` , hay varias maneras diferentes de ejecutar esto.

```
var SelectedEmployees = new List<Model>
 {
      new Model() {Name = "Item1", Selected = true},
      new Model() {Name = "Item2", Selected = false},
      new Model() {Name = "Item3", Selected = false},
      new Model() {Name = "Item4"}
 };
```

Aquí, estamos creando varias instancias `new` de nuestra clase de `Model` , y las estamos inicializando con datos. ¿Qué pasa si agregamos un constructor?

```
public class Model
{

    public Model(string name, bool? selected = false)
    {
        Name = name;
        selected = Selected;
    }
    public string Name { get; set; }
    public bool? Selected { get; set; }
}
```

Esto nos permite inicializar nuestra Lista de manera un *poco* diferente.

```
var SelectedEmployees = new List<Model>
{
    new Model("Mark", true),
    new Model("Alexis"),
    new Model("")
};
```

¿Qué pasa con una clase donde una de las propiedades es una clase en sí misma?

```
public class Model
{
    public string Name { get; set; }
    public bool? Selected { get; set; }
}

public class ExtendedModel : Model
{
    public ExtendedModel()
    {
        BaseModel = new Model();
    }

    public Model BaseModel { get; set; }
    public DateTime BirthDate { get; set; }
}
```

Observe que revertimos el constructor en la clase `Model` para simplificar un poco el ejemplo.

```
var SelectedWithBirthDate = new List<ExtendedModel>
{
    new ExtendedModel()
    {
        BaseModel = new Model { Name = "Mark", Selected = true},
        BirthDate = new DateTime(2015, 11, 23)
    },
                    new ExtendedModel()
    {
        BaseModel = new Model { Name = "Random"},
        BirthDate = new DateTime(2015, 11, 23)
    }
};
```

Tenga en cuenta que podemos intercambiar nuestra `List<ExtendedModel>` con `Collection<ExtendedModel>` , `ExtendedModel[]` , `object[]` , o incluso simplemente `[]` .

Cola

Hay una colección en .Net que se utiliza para administrar valores en una `Queue` que utiliza el concepto FIFO (primero en entrar, primero en salir) . Los conceptos básicos de las colas es el método `Enqueue(T item)` que se usa para agregar elementos en la cola y `Dequeue()` que se usa para obtener el primer elemento y eliminarlo de la cola. La versión genérica se puede utilizar como el siguiente código para una cola de cadenas.

Primero, agregue el espacio de nombres:

```
using System.Collections.Generic;
```

y úsalo:

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("John");
queue.Enqueue("Paul");
queue.Enqueue("George");
queue.Enqueue("Ringo");

string dequeueValue;
dequeueValue = queue.Dequeue(); // return John
dequeueValue = queue.Dequeue(); // return Paul
dequeueValue = queue.Dequeue(); // return George
dequeueValue = queue.Dequeue(); // return Ringo
```

Existe una versión no genérica del tipo, que funciona con objetos.

El espacio de nombres es:

```
using System.Collections;
```

Adn un ejemplo de código para la cola no genérica:

```
Queue queue = new Queue();
queue.Enqueue("Hello World"); // string
queue.Enqueue(5); // int
queue.Enqueue(1d); // double
queue.Enqueue(true); // bool
queue.Enqueue(new Product()); // Product object

object dequeueValue;
dequeueValue = queue.Dequeue(); // return Hello World (string)
dequeueValue = queue.Dequeue(); // return 5 (int)
dequeueValue = queue.Dequeue(); // return 1d (double)
dequeueValue = queue.Dequeue(); // return true (bool)
dequeueValue = queue.Dequeue(); // return Product (Product type)
```

También hay un método llamado Peek () que devuelve el objeto al principio de la cola sin eliminar los elementos.

```
Queue<int> queue = new Queue<int>();
queue.Enqueue(10);
queue.Enqueue(20);
queue.Enqueue(30);
queue.Enqueue(40);
queue.Enqueue(50);

foreach (int element in queue)
{
    Console.WriteLine(i);
}
```

La salida (sin quitar):

```
10
20
30
40
50
```

Apilar

Hay una colección en .Net utilizada para administrar valores en una `Stack` que usa el concepto LIFO (último en entrar, primero en salir) . Los conceptos básicos de las pilas es el método `Push(T item)` que se usa para agregar elementos en la pila y `Pop()` que se usa para obtener el último elemento agregado y eliminarlo de la pila. La versión genérica se puede utilizar como el siguiente código para una cola de cadenas.

Primero, agregue el espacio de nombres:

```
using System.Collections.Generic;
```

y úsalo:

```
Stack<string> stack = new Stack<string>();
stack.Push("John");
stack.Push("Paul");
stack.Push("George");
stack.Push("Ringo");

string value;
value = stack.Pop(); // return Ringo
value = stack.Pop(); // return George
value = stack.Pop(); // return Paul
value = stack.Pop(); // return John
```

Existe una versión no genérica del tipo, que funciona con objetos.

El espacio de nombres es:

```
using System.Collections;
```

Y un ejemplo de código de pila no genérica:

```
Stack stack = new Stack();
stack.Push("Hello World"); // string
stack.Push(5); // int
stack.Push(1d); // double
stack.Push(true); // bool
stack.Push(new Product()); // Product object

object value;
value = stack.Pop(); // return Product (Product type)
value = stack.Pop(); // return true (bool)
```

```
value = stack.Pop(); // return 1d (double)
value = stack.Pop(); // return 5 (int)
value = stack.Pop(); // return Hello World (string)
```

También hay un método llamado Peek () que devuelve el último elemento agregado pero sin eliminarlo de la `Stack` .

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);

var lastValueAdded = stack.Peek(); // 20
```

Es posible iterar sobre los elementos en la pila y respetará el orden de la pila (LIFO).

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);
stack.Push(30);
stack.Push(40);
stack.Push(50);

foreach (int element in stack)
{
   Console.WriteLine(element);
}
```

La salida (sin quitar):

```
50
40
30
20
10
```

Usando inicializadores de colección

Algunos tipos de colección se pueden inicializar en el momento de la declaración. Por ejemplo, la siguiente declaración crea e inicializa los `numbers` con algunos enteros:

```
List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};
```

Internamente, el compilador de C # en realidad convierte esta inicialización en una serie de llamadas al método Add. Por consiguiente, puede usar esta sintaxis solo para las colecciones que realmente admiten el método `Add` .

> Las clases `Stack<T>` y `Queue<T>` no lo admiten.

Para colecciones complejas como la clase `Dictionary<TKey, TValue>` , que toman pares clave / valor, puede especificar cada par clave / valor como un tipo anónimo en la lista de inicializadores.

```
Dictionary<int, string> employee = new Dictionary<int, string>()
     {{44, "John"}, {45, "Bob"}, {47, "James"}, {48, "Franklin"}};
```

El primer elemento de cada par es la clave, y el segundo es el valor.

Lea Colecciones en línea: https://riptutorial.com/es/dot-net/topic/30/colecciones

Capítulo 16: Compilador JIT

Introducción

La compilación JIT, o compilación justo a tiempo, es un enfoque alternativo para la interpretación del código o la compilación anticipada. La compilación JIT se utiliza en el marco .NET. El código CLR (C #, F #, Visual Basic, etc.) primero se compila en algo llamado Lenguaje interpretado o IL. Este es un código de nivel inferior que está más cerca del código de la máquina, pero no es específico de la plataforma. Más bien, en tiempo de ejecución, este código se compila en código de máquina para el sistema relevante.

Observaciones

¿Por qué usar la compilación JIT?

- Mejor compatibilidad: cada lenguaje CLR necesita solo un compilador para IL, y esta IL puede ejecutarse en cualquier plataforma en la que se pueda convertir en código de máquina.
- Velocidad: la compilación JIT intenta combinar la velocidad de ejecución del código compilado anticipado y la flexibilidad de interpretación (puede analizar el código que se ejecutará para posibles optimizaciones antes de compilar)

Página de Wikipedia para obtener más información sobre la compilación JIT en general: https://en.wikipedia.org/wiki/Just-in-time_compilation

Examples

Muestra de compilación de IL

Sencilla aplicación Hello World:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

Código de IL equivalente (que se compilará con JIT)

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 4.6.1055.0
```

```
//  Copyright (c) Microsoft Corporation.  All rights reserved.



// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
  .ver 4:0:0:0
}
.assembly HelloWorld
{
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.ctor(int32) = ( 01
00 08 00 00 00 00 00 )
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ctor() = ( 01 00 01
00 54 02 16 57 72 61 70 4E 6F 6E 45 78   // ....T..WrapNonEx

63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )       // ceptionThrows.

  // --- The following custom attribute is added automatically, do not uncomment -------
  //  .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribute::.ctor(valuetype
[mscorlib]System.Diagnostics.DebuggableAttribute/DebuggingModes) = ( 01 00 07 01 00 00 00 00 )


  .custom instance void [mscorlib]System.Reflection.AssemblyTitleAttribute::.ctor(string) = (
01 00 0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
  .custom instance void
[mscorlib]System.Reflection.AssemblyDescriptionAttribute::.ctor(string) = ( 01 00 00 00 00 )
  .custom instance void
[mscorlib]System.Reflection.AssemblyConfigurationAttribute::.ctor(string) = ( 01 00 00 00 00 )

  .custom instance void [mscorlib]System.Reflection.AssemblyCompanyAttribute::.ctor(string) =
( 01 00 00 00 00 )
  .custom instance void [mscorlib]System.Reflection.AssemblyProductAttribute::.ctor(string) =
( 01 00 0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
  .custom instance void [mscorlib]System.Reflection.AssemblyCopyrightAttribute::.ctor(string)
= ( 01 00 12 43 6F 70 79 72 69 67 68 74 20 C2 A9 20   // ...Copyright ..

20 32 30 31 37 00 00 )                            //  2017..
  .custom instance void [mscorlib]System.Reflection.AssemblyTrademarkAttribute::.ctor(string)
= ( 01 00 00 00 00 )
  .custom instance void
[mscorlib]System.Runtime.InteropServices.ComVisibleAttribute::.ctor(bool) = ( 01 00 00 00 00 )

  .custom instance void [mscorlib]System.Runtime.InteropServices.GuidAttribute::.ctor(string)
= ( 01 00 24 33 30 38 62 33 64 38 36 2D 34 31 37 32   // ..$308b3d86-4172

2D 34 30 32 32 2D 61 66 63 63 2D 33 66 38 65 33   // -4022-afcc-3f8e3

32 33 33 63 35 62 30 00 00 )                      // 233c5b0..
  .custom instance void
[mscorlib]System.Reflection.AssemblyFileVersionAttribute::.ctor(string) = ( 01 00 07 31 2E 30
2E 30 2E 30 00 00 )             // ...1.0.0.0..
  .custom instance void
[mscorlib]System.Runtime.Versioning.TargetFrameworkAttribute::.ctor(string) = ( 01 00 1C 2E 4E
45 54 46 72 61 6D 65 77 6F 72 6B   // ....NETFramework

2C 56 65 72 73 69 6F 6E 3D 76 34 2E 35 2E 32 01   // ,Version=v4.5.2.
```

```
00 54 0E 14 46 72 61 6D 65 77 6F 72 6B 44 69 73   // .T..FrameworkDis

70 6C 61 79 4E 61 6D 65 14 2E 4E 45 54 20 46 72   // playName..NET Fr

61 6D 65 77 6F 72 6B 20 34 2E 35 2E 32 )          // amework 4.5.2
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module HelloWorld.exe
// MVID: {2A7E1D59-1272-4B47-85F6-D7E1ED057831}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00020003    //  ILONLY 32BITPREFERRED
// Image base: 0x0000021C70230000


// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit HelloWorld.Program
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  Main(string[] args) cil managed
  {
    .entrypoint
    // Code size       13 (0xd)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello World"
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  } // end of method Program::Main

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       8 (0x8)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  nop
    IL_0007:  ret
  } // end of method Program::.ctor

} // end of class HelloWorld.Program
```

Generado con la herramienta MS ILDASM (desensamblador IL)

Lea Compilador JIT en línea: https://riptutorial.com/es/dot-net/topic/9222/compilador-jit

Capítulo 17: Contextos de sincronización

Observaciones

Un Contexto de Sincronización es una abstracción que permite consumir código para pasar unidades de trabajo a un programador, sin necesidad de conocer cómo se programará el trabajo.

Los contextos de sincronización se usan tradicionalmente para garantizar que el código se ejecute en un subproceso específico. En las aplicaciones WPF y Winforms, el marco de presentación proporciona un `SynchronizationContext` representa el subproceso de la interfaz de usuario. De esta manera, `SynchronizationContext` se puede considerar como un patrón productor-consumidor para los delegados. Un subproceso de trabajo *producirá un* código ejecutable (el delegado) y lo pondrá en cola o *consumirá* en el bucle de mensajes de la interfaz de usuario.

La biblioteca paralela de tareas proporciona funciones para capturar y utilizar automáticamente los contextos de sincronización.

Examples

Ejecutar código en el hilo de la interfaz de usuario después de realizar un trabajo en segundo plano

Este ejemplo muestra cómo actualizar un componente de la interfaz de usuario desde un subproceso en segundo plano utilizando un `SynchronizationContext`

```
void Button_Click(object sender, EventArgs args)
{
    SynchronizationContext context = SynchronizationContext.Current;
    Task.Run(() =>
    {
        for(int i = 0; i < 10; i++)
        {
            Thread.Sleep(500); //simulate work being done
            context.Post(ShowProgress, "Work complete on item " + i);
        }
    }
}

void UpdateCallback(object state)
{
    // UI can be safely updated as this method is only called from the UI thread
    this.MyTextBox.Text = state as string;
}
```

En este ejemplo, si intentara actualizar directamente `MyTextBox.Text` dentro del bucle `for` , obtendría un error de subprocesamiento. Al publicar la acción `UpdateCallback` en `SynchronizationContext` , el cuadro de texto se actualiza en el mismo hilo que el resto de la interfaz de usuario.

En la práctica, las actualizaciones de progreso deben realizarse utilizando una instancia de `System.IProgress<T>` . La implementación predeterminada `System.Progress<T>` captura automáticamente el contexto de sincronización en el que se crea.

Lea Contextos de sincronización en línea: https://riptutorial.com/es/dot-net/topic/5407/contextos-de-sincronizacion

Capítulo 18: Contratos de código

Observaciones

Los contratos de código permiten la compilación o el análisis en tiempo de ejecución de las condiciones pre / post de los métodos y las condiciones invariables para los objetos. Estas condiciones se pueden usar para garantizar que las personas que llaman y el valor de retorno coincidan con los estados válidos para el procesamiento de la aplicación. Otros usos de los contratos de código incluyen la generación de documentación.

Examples

Precondiciones

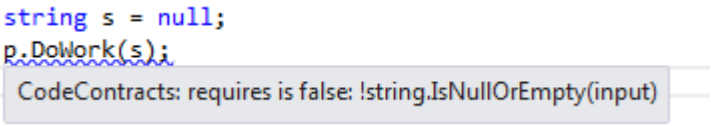
Las condiciones previas permiten que los métodos proporcionen valores mínimos requeridos para los parámetros de entrada

Ejemplo...

```
void DoWork(string input)
{
    Contract.Requires(!string.IsNullOrEmpty(input));

    //do work
}
```

Resultado del análisis estático ...



Postcondiciones

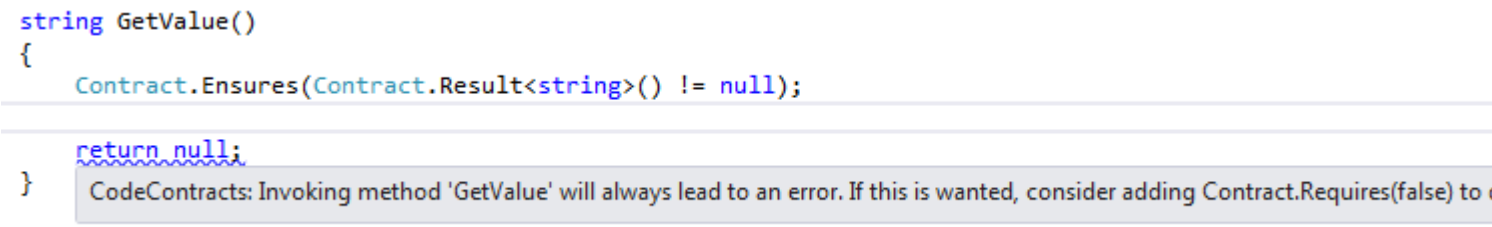
Las condiciones posteriores aseguran que los resultados devueltos de un método coincidirán con la definición proporcionada. Esto proporciona a la persona que llama una definición del resultado esperado. Las condiciones posteriores pueden permitir implementaciones simplificadas, ya que el analizador estático puede proporcionar algunos resultados posibles.

Ejemplo...

```
string GetValue()
{
    Contract.Ensures(Contract.Result<string>() != null);

    return null;
}
```

Analisis Estático Resultado ...



Contratos para interfaces

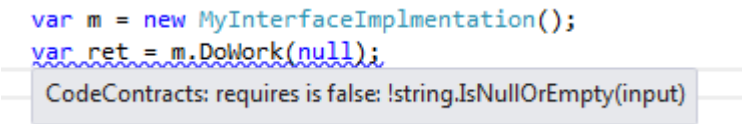
Usando Contratos de Código es posible aplicar un contrato a una interfaz. Esto se hace declarando una clase abstracta que implementa las interfaces. La interfaz debe estar etiquetada con el `ContractClassAttribute` y la definición del contrato (la clase abstracta) debe estar etiquetada con el `ContractClassForAttribute`

C # Ejemplo ...

```
[ContractClass(typeof(MyInterfaceContract))]
public interface IMyInterface
{
    string DoWork(string input);
}
//Never inherit from this contract defintion class
[ContractClassFor(typeof(IMyInterface))]
internal abstract class MyInterfaceContract : IMyInterface
{
    private MyInterfaceContract() { }

    public string DoWork(string input)
    {
        Contract.Requires(!string.IsNullOrEmpty(input));
        Contract.Ensures(!string.IsNullOrEmpty(Contract.Result<string>()));
        throw new NotSupportedException();
    }
}
public class MyInterfaceImplmentation : IMyInterface
{
    public string DoWork(string input)
    {
        return input;
    }
}
```

Resultado del análisis estático ...

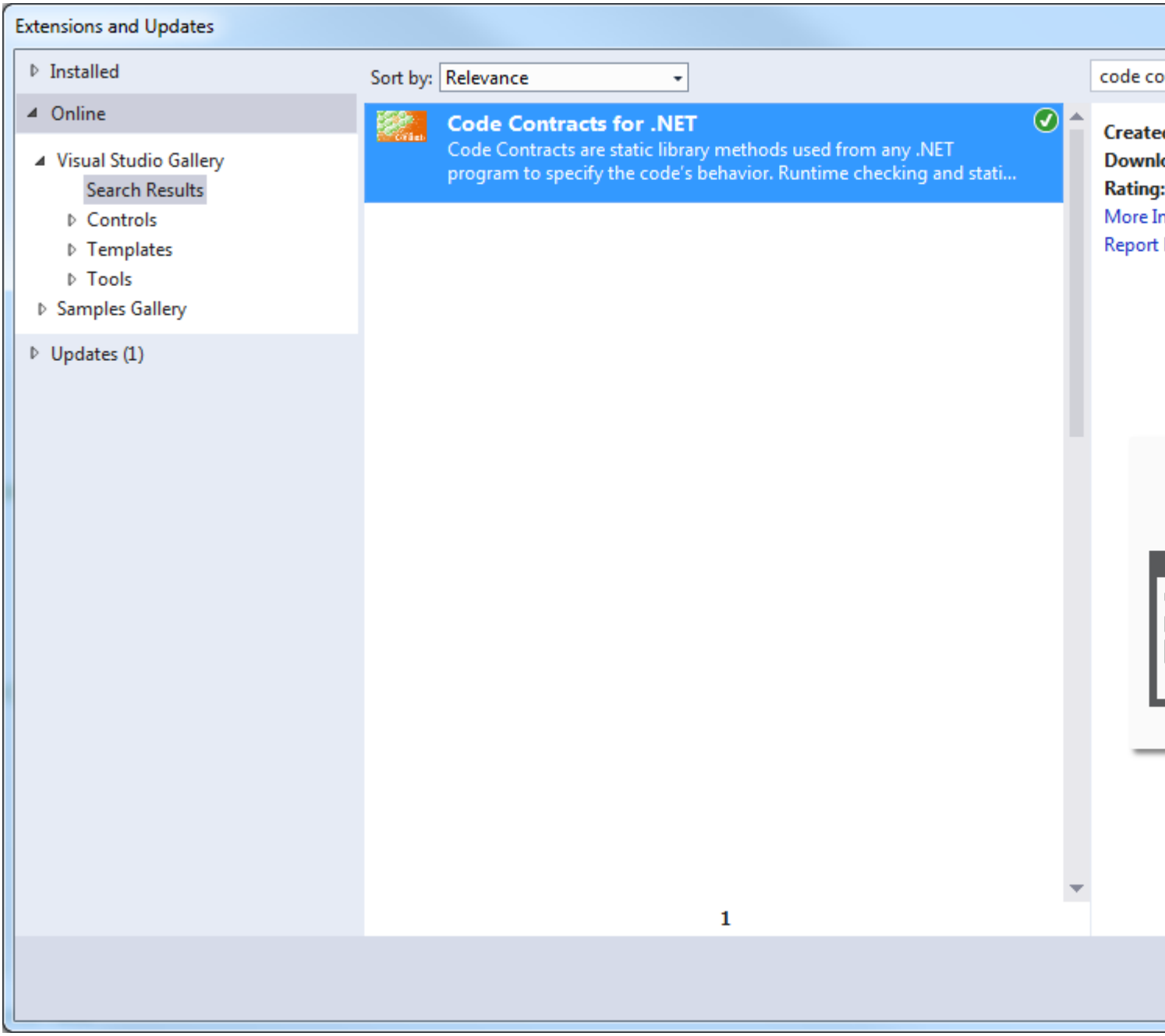


Instalación y habilitación de contratos de código

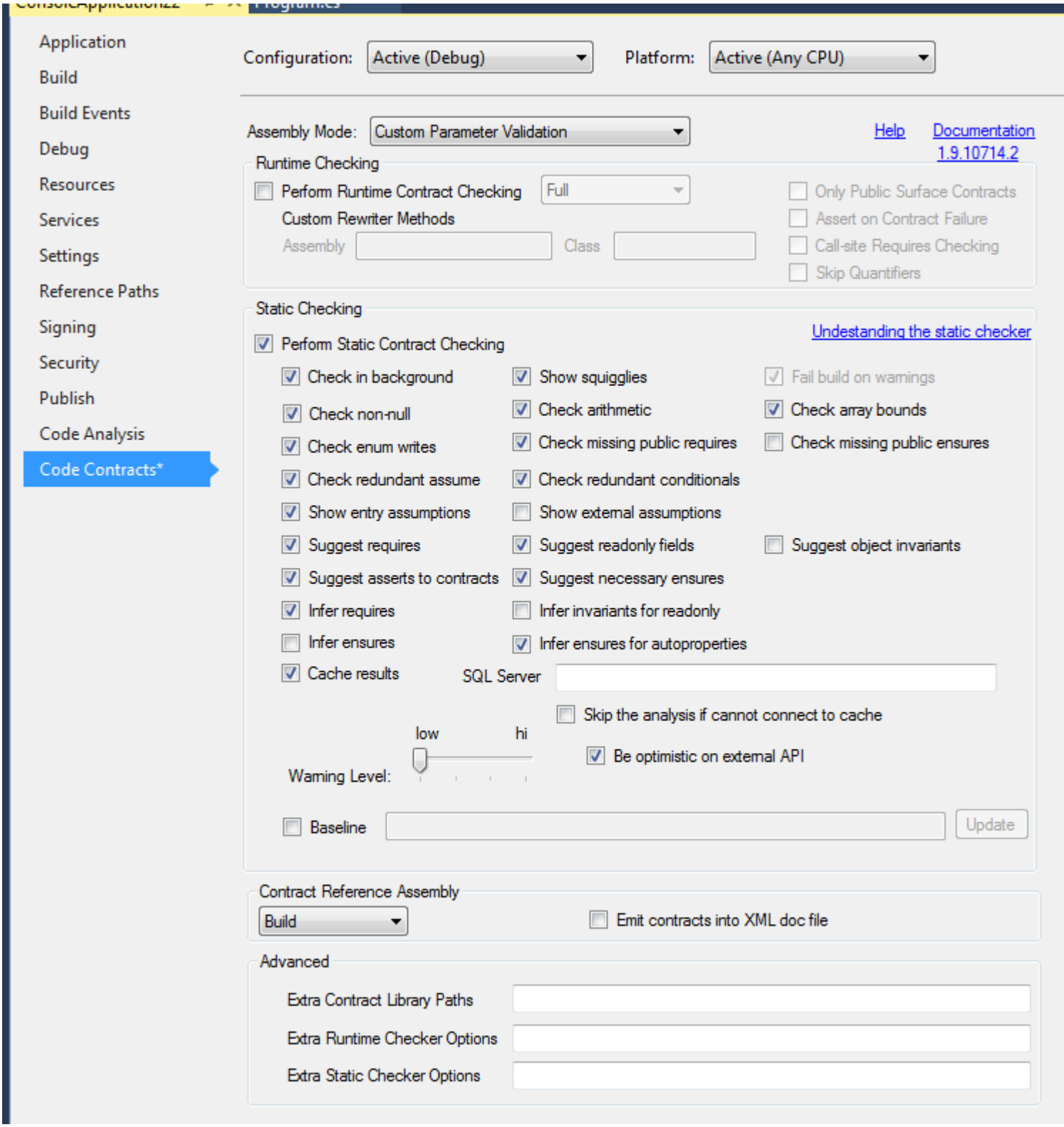
Mientras que `System.Diagnostics.Contracts` está incluido dentro de .Net Framework. Para utilizar

los contratos de código, debe instalar las extensiones de Visual Studio.

En `Extensions and Updates` busque `Code Contracts` luego instale las `Code Contracts Tools`



Después de instalar las herramientas, debe habilitar los `Code Contracts` dentro de su solución de Proyecto. Como mínimo, es probable que desee habilitar la `Static Checking` (comprobación después de la compilación). Si está implementando una biblioteca que será utilizada por otras soluciones, puede considerar también habilitar la `Runtime Checking` .



Lea Contratos de código en línea: https://riptutorial.com/es/dot-net/topic/1937/contratos-de-codigo

Capítulo 19: Descripción general de la API de la biblioteca paralela de tareas (TPL)

Observaciones

La biblioteca paralela de tareas es un conjunto de tipos públicos y API que simplifican drásticamente el proceso de agregar paralelismo y concurrencia a una aplicación. .Red. TPL se introdujo en .Net 4 y es la forma recomendada de escribir código de subprocesos múltiples y paralelo.

TPL se encarga de la programación del trabajo, la afinidad de subprocesos, el soporte de cancelación, la administración del estado y el equilibrio de carga para que el programador pueda concentrarse en resolver problemas en lugar de dedicar tiempo a detalles comunes de bajo nivel.

Examples

Realice el trabajo en respuesta a un clic del botón y actualice la interfaz de usuario

Este ejemplo demuestra cómo puede responder a un clic de botón realizando algún trabajo en un subproceso de trabajo y luego actualizar la interfaz de usuario para indicar que se ha completado.

```
void MyButton_OnClick(object sender, EventArgs args)
{
    Task.Run(() => // Schedule work using the thread pool
        {
            System.Threading.Thread.Sleep(5000); // Sleep for 5 seconds to simulate work.
        })
    .ContinueWith(p => // this continuation contains the 'update' code to run on the UI thread
    {
        this.TextBlock_ResultText.Text = "The work completed at " + DateTime.Now.ToString()
    },
    TaskScheduler.FromCurrentSynchronizationContext()); // make sure the update is run on the
UI thread.

}
```

Lea Descripción general de la API de la biblioteca paralela de tareas (TPL) en línea: https://riptutorial.com/es/dot-net/topic/5164/descripcion-general-de-la-api-de-la-biblioteca-paralela-de-tareas--tpl-

Capítulo 20: Diagnostico del sistema

Examples

Cronógrafo

Este ejemplo muestra cómo se puede usar el `Stopwatch` para evaluar un bloque de código.

```
using System;
using System.Diagnostics;

public class Benchmark : IDisposable
{
    private Stopwatch sw;

    public Benchmark()
    {
        sw = Stopwatch.StartNew();
    }

    public void Dispose()
    {
        sw.Stop();
        Console.WriteLine(sw.Elapsed);
    }
}

public class Program
{
    public static void Main()
    {
        using (var bench = new Benchmark())
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

Ejecutar comandos de shell

```
string strCmdText = "/C copy /b Image1.jpg + Archive.rar Image2.jpg";
System.Diagnostics.Process.Start("CMD.exe",strCmdText);
```

Esto es para ocultar la ventana de cmd.

```
System.Diagnostics.Process process = new System.Diagnostics.Process();
System.Diagnostics.ProcessStartInfo startInfo = new System.Diagnostics.ProcessStartInfo();
startInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden;
startInfo.FileName = "cmd.exe";
startInfo.Arguments = "/C copy /b Image1.jpg + Archive.rar Image2.jpg";
process.StartInfo = startInfo;
process.Start();
```

Enviar comando a CMD y recibir salida

Este método permite enviar un `command` a `Cmd.exe` y devuelve la salida estándar (incluido el error estándar) como una cadena:

```
private static string SendCommand(string command)
{
    var cmdOut = string.Empty;

    var startInfo = new ProcessStartInfo("cmd", command)
    {
        WorkingDirectory = @"C:\Windows\System32", // Directory to make the call from
        WindowStyle = ProcessWindowStyle.Hidden,   // Hide the window
        UseShellExecute = false,                   // Do not use the OS shell to start the
process
        CreateNoWindow = true,                     // Start the process in a new window
        RedirectStandardOutput = true,             // This is required to get STDOUT
        RedirectStandardError = true               // This is required to get STDERR
    };

    var p = new Process {StartInfo = startInfo};

    p.Start();

    p.OutputDataReceived += (x, y) => cmdOut += y.Data;
    p.ErrorDataReceived += (x, y) => cmdOut += y.Data;
    p.BeginOutputReadLine();
    p.BeginErrorReadLine();
    p.WaitForExit();
    return cmdOut;
}
```

Uso

```
var servername = "SVR-01.domain.co.za";
var currentUsers = SendCommand($"/C QUERY USER /SERVER:{servername}")
```

Salida

> string currentUsers = "USERNAME SESSIONNAME ID STATE IDLE TIME LOGON TIME Joe.Bloggs ica-cgp # 0 2 Active 24692 + 13: 29 25/07/2016 07:50 Jim.McFlannegan ica-cgp # 1 3 Active. 25/07 / 2016 08:33 Andy.McAnderson ica-cgp # 2 4 activo. 25/07/2016 08:54 John.Smith ica-cgp # 4 5 activo 14 25/07/2016 08:57 Bob.Bobbington ica-cgp # 5 6 Activo 24692 + 13: 29 25/07/2016 09:05 Tim.Tom ica-cgp # 6 7 Activo. 25/07/2016 09:08 Bob.Joges ica-cgp # 7 8 Activo 24692 + 13: 29 25 / 07/2016 09:13 "

En algunas ocasiones, el acceso al servidor en cuestión puede estar restringido a ciertos usuarios. Si tiene las credenciales de inicio de sesión para este usuario, es posible enviar consultas con este método:

```
private static string SendCommand(string command)
```

```
{
    var cmdOut = string.Empty;

    var startInfo = new ProcessStartInfo("cmd", command)
    {
        WorkingDirectory = @"C:\Windows\System32",
        WindowStyle = ProcessWindowStyle.Hidden,    // This does not actually work in
conjunction with "runas" - the console window will still appear!
        UseShellExecute = false,
        CreateNoWindow = true,
        RedirectStandardOutput = true,
        RedirectStandardError = true,

        Verb = "runas",
        Domain = "doman1.co.za",
        UserName = "administrator",
        Password = GetPassword()
    };

    var p = new Process {StartInfo = startInfo};

    p.Start();

    p.OutputDataReceived += (x, y) => cmdOut += y.Data;
    p.ErrorDataReceived += (x, y) => cmdOut += y.Data;
    p.BeginOutputReadLine();
    p.BeginErrorReadLine();
    p.WaitForExit();
    return cmdOut;
}
```

Obteniendo la contraseña:

```
static SecureString GetPassword()
{
    var plainText = "password123";
    var ss = new SecureString();
    foreach (char c in plainText)
    {
        ss.AppendChar(c);
    }

    return ss;
}
```

Notas

Los dos métodos anteriores devolverán una concatenación de STDOUT y STDERR, ya que `OutputDataReceived` y `ErrorDataReceived` se `ErrorDataReceived` a la misma variable, `cmdOut` .

Lea Diagnostico del sistema en línea: https://riptutorial.com/es/dot-net/topic/3143/diagnostico-del-sistema

Capítulo 21: Encriptación / Criptografía

Observaciones

.NET Framework proporciona la implementación de muchos algoritmos criptográficos. Incluyen básicamente algoritmos simétricos, algoritmos asimétricos y hashes.

Examples

RijndaelManaged

Espacio de nombres requerido: `System.Security.Cryptography`

```
private class Encryption {

          private const string SecretKey = "topSecretKeyusedforEncryptions";

          private const string SecretIv = "secretVectorHere";

          public string Encrypt(string data) {
            return string.IsNullOrEmpty(data) ? data :
Convert.ToBase64String(this.EncryptStringToBytesAes(data, this.GetCryptographyKey(),
this.GetCryptographyIv()));
          }

          public string Decrypt(string data) {
            return string.IsNullOrEmpty(data) ? data :
this.DecryptStringFromBytesAes(Convert.FromBase64String(data), this.GetCryptographyKey(),
this.GetCryptographyIv());
          }

          private byte[] GetCryptographyKey() {
            return Encoding.ASCII.GetBytes(SecretKey.Replace('e', '!'));
          }

          private byte[] GetCryptographyIv() {
            return Encoding.ASCII.GetBytes(SecretIv.Replace('r', '!'));
          }

          private byte[] EncryptStringToBytesAes(string plainText, byte[] key, byte[] iv) {
            MemoryStream encrypt;
            RijndaelManaged aesAlg = null;
            try {
              aesAlg = new RijndaelManaged {
                Key = key,
                IV = iv
              };
              var encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
              encrypt = new MemoryStream();
              using (var csEncrypt = new CryptoStream(encrypt, encryptor,
CryptoStreamMode.Write)) {
                using (var swEncrypt = new StreamWriter(csEncrypt)) {
                  swEncrypt.Write(plainText);
                }
```

```
            }
          } finally {
            aesAlg?.Clear();
          }
          return encrypt.ToArray();
        }

        private string DecryptStringFromBytesAes(byte[] cipherText, byte[] key, byte[] iv) {
          RijndaelManaged aesAlg = null;
          string plaintext;
          try {
            aesAlg = new RijndaelManaged {
              Key = key,
              IV = iv
            };
            var decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);
            using (var msDecrypt = new MemoryStream(cipherText)) {
              using (var csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read)) {
                using (var srDecrypt = new StreamReader(csDecrypt))
                  plaintext = srDecrypt.ReadToEnd();
              }
            }
          } finally {
            aesAlg?.Clear();
          }
          return plaintext;
        }
      }
```

Uso

```
var textToEncrypt = "hello World";

 var encrypted = new Encryption().Encrypt(textToEncrypt); //-> zBmW+FUxOvdbpOGm9Ss/vQ==

 var decrypted = new Encryption().Decrypt(encrypted); //-> hello World
```

Nota:

- Rijndael es el predecesor del algoritmo criptográfico simétrico estándar AES.

Cifrar y descifrar datos usando AES (en C #)

```
using System;
using System.IO;
using System.Security.Cryptography;

namespace Aes_Example
{
    class AesExample
    {
        public static void Main()
        {
            try
            {
                string original = "Here is some data to encrypt!";
```

```
            // Create a new instance of the Aes class.
            // This generates a new key and initialization vector (IV).
            using (Aes myAes = Aes.Create())
            {
                // Encrypt the string to an array of bytes.
                byte[] encrypted = EncryptStringToBytes_Aes(original,
                                                            myAes.Key,
                                                            myAes.IV);

                // Decrypt the bytes to a string.
                string roundtrip = DecryptStringFromBytes_Aes(encrypted,
                                                              myAes.Key,
                                                              myAes.IV);

                //Display the original data and the decrypted data.
                Console.WriteLine("Original:   {0}", original);
                Console.WriteLine("Round Trip: {0}", roundtrip);
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: {0}", e.Message);
        }
    }

    static byte[] EncryptStringToBytes_Aes(string plainText, byte[] Key, byte[] IV)
    {
        // Check arguments.
        if (plainText == null || plainText.Length <= 0)
            throw new ArgumentNullException("plainText");
        if (Key == null || Key.Length <= 0)
            throw new ArgumentNullException("Key");
        if (IV == null || IV.Length <= 0)
            throw new ArgumentNullException("IV");

        byte[] encrypted;

        // Create an Aes object with the specified key and IV.
        using (Aes aesAlg = Aes.Create())
        {
            aesAlg.Key = Key;
            aesAlg.IV = IV;

            // Create a decrytor to perform the stream transform.
            ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key,
                                                                aesAlg.IV);

            // Create the streams used for encryption.
            using (MemoryStream msEncrypt = new MemoryStream())
            {
                using (CryptoStream csEncrypt = new CryptoStream(msEncrypt,
                                                                 encryptor,
                                                                 CryptoStreamMode.Write))
                {
                    using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))
                    {
                        //Write all data to the stream.
                        swEncrypt.Write(plainText);
                    }
```

```
                        encrypted = msEncrypt.ToArray();
                    }
                }
            }

            // Return the encrypted bytes from the memory stream.
            return encrypted;
        }

        static string DecryptStringFromBytes_Aes(byte[] cipherText, byte[] Key, byte[] IV)
        {
            // Check arguments.
            if (cipherText == null || cipherText.Length <= 0)
                throw new ArgumentNullException("cipherText");
            if (Key == null || Key.Length <= 0)
                throw new ArgumentNullException("Key");
            if (IV == null || IV.Length <= 0)
                throw new ArgumentNullException("IV");

            // Declare the string used to hold the decrypted text.
            string plaintext = null;

            // Create an Aes object with the specified key and IV.
            using (Aes aesAlg = Aes.Create())
            {
                aesAlg.Key = Key;
                aesAlg.IV = IV;

                // Create a decrytor to perform the stream transform.
                ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key,
                                                                    aesAlg.IV);

                // Create the streams used for decryption.
                using (MemoryStream msDecrypt = new MemoryStream(cipherText))
                {
                    using (CryptoStream csDecrypt = new CryptoStream(msDecrypt,
                                                                     decryptor,
                                                                     CryptoStreamMode.Read))
                    {
                        using (StreamReader srDecrypt = new StreamReader(csDecrypt))
                        {

                            // Read the decrypted bytes from the decrypting stream
                            // and place them in a string.
                            plaintext = srDecrypt.ReadToEnd();
                        }
                    }
                }
            }

            return plaintext;
        }
    }
}
```

Este ejemplo es de MSDN .

Es una aplicación de demostración de consola, que muestra cómo cifrar una cadena utilizando el cifrado **AES** estándar y cómo descifrarla después.

(**AES = Advanced Encryption Standard** , una especificación para el cifrado de datos electrónicos establecida por el Instituto Nacional de Estándares y Tecnología (NIST) de EE. UU. En 2001, que sigue siendo el estándar de facto para el cifrado simétrico)

Notas:

- En un escenario de cifrado real, debe elegir un modo de cifrado adecuado (se puede asignar a la propiedad `Mode` seleccionando un valor de la enumeración `CipherMode`). **Nunca** use el `CipherMode.ECB` (modo de libro de códigos electrónico), ya que esto produce un flujo de cifrado débil
- Para crear una `Key` buena (y no débil), use un generador criptográfico aleatorio o use el ejemplo anterior (**Crear una clave a partir de una contraseña**). El **tamaño de clave** recomendado es de 256 bits. Los tamaños de clave admitidos están disponibles a través de la propiedad `LegalKeySizes` .
- Para inicializar el vector de inicialización `IV` , puede usar una SALT como se muestra en el ejemplo anterior (**SAL aleatoria**)
- Los tamaños de bloque admitidos están disponibles a través de la propiedad `SupportedBlockSizes` , el tamaño de bloque puede asignarse a través de la propiedad `BlockSize`

Uso: ver el método principal ().

Crear una clave a partir de una contraseña / SALT aleatoria (en C #)

```
using System;
using System.Security.Cryptography;
using System.Text;

public class PasswordDerivedBytesExample
{
    public static void Main(String[] args)
    {
        // Get a password from the user.
        Console.WriteLine("Enter a password to produce a key:");

        byte[] pwd = Encoding.Unicode.GetBytes(Console.ReadLine());

        byte[] salt = CreateRandomSalt(7);

        // Create a TripleDESCryptoServiceProvider object.
        TripleDESCryptoServiceProvider tdes = new TripleDESCryptoServiceProvider();

        try
        {
            Console.WriteLine("Creating a key with PasswordDeriveBytes...");

            // Create a PasswordDeriveBytes object and then create
            // a TripleDES key from the password and salt.
            PasswordDeriveBytes pdb = new PasswordDeriveBytes(pwd, salt);

            // Create the key and set it to the Key property
            // of the TripleDESCryptoServiceProvider object.
```

```
            tdes.Key = pdb.CryptDeriveKey("TripleDES", "SHA1", 192, tdes.IV);

            Console.WriteLine("Operation complete.");
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
            // Clear the buffers
            ClearBytes(pwd);
            ClearBytes(salt);

            // Clear the key.
            tdes.Clear();
        }

        Console.ReadLine();
    }

    #region Helper methods

    /// <summary>
    /// Generates a random salt value of the specified length.
    /// </summary>
    public static byte[] CreateRandomSalt(int length)
    {
        // Create a buffer
        byte[] randBytes;

        if (length >= 1)
        {
            randBytes = new byte[length];
        }
        else
        {
            randBytes = new byte[1];
        }

        // Create a new RNGCryptoServiceProvider.
        RNGCryptoServiceProvider rand = new RNGCryptoServiceProvider();

        // Fill the buffer with random bytes.
        rand.GetBytes(randBytes);

        // return the bytes.
        return randBytes;
    }

    /// <summary>
    /// Clear the bytes in a buffer so they can't later be read from memory.
    /// </summary>
    public static void ClearBytes(byte[] buffer)
    {
        // Check arguments.
        if (buffer == null)
        {
            throw new ArgumentNullException("buffer");
        }
```

```
        // Set each byte in the buffer to 0.
        for (int x = 0; x < buffer.Length; x++)
        {
            buffer[x] = 0;
        }
    }

    #endregion
}
```

Este ejemplo está tomado de MSDN.

Es una demostración de la consola, y muestra cómo crear una clave segura basada en una contraseña definida por el usuario, y cómo crear una SALT aleatoria basada en el generador criptográfico aleatorio.

Notas:

- La función incorporada `PasswordDeriveBytes` utiliza el algoritmo estándar PBKDF1 para generar una clave a partir de la contraseña. Por defecto, usa 100 iteraciones para generar la clave para ralentizar los ataques de fuerza bruta. La SAL generada al azar refuerza aún más la clave.
- La función `CryptDeriveKey` convierte la clave generada por `PasswordDeriveBytes` en una clave compatible con el algoritmo de cifrado especificado (aquí "TripleDES") utilizando el algoritmo hash especificado (aquí "SHA1"). El tamaño de clave en este ejemplo es de 192 bytes, y el vector de inicialización IV se toma del proveedor de cifrado de triple DES
- Por lo general, este mecanismo se usa para proteger una clave generada aleatoriamente más fuerte mediante una contraseña, que cifra una gran cantidad de datos. También puede usarlo para proporcionar múltiples contraseñas de diferentes usuarios para dar acceso a los mismos datos (está protegido por una clave aleatoria diferente).
- Desafortunadamente, `CryptDeriveKey` actualmente no es compatible con AES. Ver aqui **NOTA:** Como solución alternativa, puede crear una clave AES aleatoria para el cifrado de los datos que se protegerán con AES y almacenar la clave AES en un contenedor TripleDES que utiliza la clave generada por `CryptDeriveKey` . Pero eso limita la seguridad a TripleDES, no aprovecha el tamaño de clave más grande de AES y crea una dependencia de TripleDES.

Uso: Ver el método principal ().

Cifrado y descifrado mediante criptografía (AES)

Codigo de desencriptacion

```
    public static string Decrypt(string cipherText)
    {
        if (cipherText == null)
            return null;
```

```
        byte[] cipherBytes = Convert.FromBase64String(cipherText);
        using (Aes encryptor = Aes.Create())
        {
            Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(CryptKey, new byte[] { 0x49, 0x76,
0x61, 0x6e, 0x20, 0x4d, 0x65, 0x64, 0x76, 0x65, 0x64, 0x65, 0x76 });
            encryptor.Key = pdb.GetBytes(32);
            encryptor.IV = pdb.GetBytes(16);

            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor.CreateDecryptor(),
CryptoStreamMode.Write))
                {
                    cs.Write(cipherBytes, 0, cipherBytes.Length);
                    cs.Close();
                }

                cipherText = Encoding.Unicode.GetString(ms.ToArray());
            }
        }

        return cipherText;
    }
```

Codigo de cifrado

```
    public static string Encrypt(string cipherText)
    {
        if (cipherText == null)
            return null;

        byte[] clearBytes = Encoding.Unicode.GetBytes(cipherText);
        using (Aes encryptor = Aes.Create())
        {
            Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(CryptKey, new byte[] { 0x49, 0x76,
0x61, 0x6e, 0x20, 0x4d, 0x65, 0x64, 0x76, 0x65, 0x64, 0x65, 0x76 });
            encryptor.Key = pdb.GetBytes(32);
            encryptor.IV = pdb.GetBytes(16);

            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor.CreateEncryptor(),
CryptoStreamMode.Write))
                {
                    cs.Write(clearBytes, 0, clearBytes.Length);
                    cs.Close();
                }

                cipherText = Convert.ToBase64String(ms.ToArray());
            }
        }
        return cipherText;
    }
```

Uso

```
var textToEncrypt = "TestEncrypt";

var encrypted = Encrypt(textToEncrypt);
```

```
var decrypted = Decrypt(encrypted);
```

Lea Encriptación / Criptografía en línea: https://riptutorial.com/es/dot-net/topic/7615/encriptacion---criptografia

Capítulo 22: Enhebrado

Examples

Accediendo a los controles de formulario desde otros hilos.

Si desea cambiar un atributo de un control, como un cuadro de texto o una etiqueta de otro hilo que no sea el hilo de la GUI que creó el control, deberá invocarlo o, de lo contrario, podría aparecer un mensaje de error que indique:

> "Operación de subprocesos no válida: control 'nombre_control' al que se accede desde un hilo que no sea el hilo en el que se creó."

El uso de este código de ejemplo en un formulario system.windows.forms emitirá una excepción con ese mensaje:

```
private void button4_Click(object sender, EventArgs e)
{
    Thread thread = new Thread(updatetextbox);
    thread.Start();
}

private void updatetextbox()
{
    textBox1.Text = "updated"; // Throws exception
}
```

En cambio, cuando desea cambiar el texto de un cuadro de texto dentro de un hilo que no lo posee, use Control.Invoke o Control.BeginInvoke. También puede usar Control.InvokeRequired para verificar si es necesario invocar el control.

```
private void updatetextbox()
{
    if (textBox1.InvokeRequired)
        textBox1.BeginInvoke((Action)(() => textBox1.Text = "updated"));
    else
        textBox1.Text = "updated";
}
```

Si necesita hacer esto a menudo, puede escribir una extensión para objetos invocables para reducir la cantidad de código necesario para realizar esta comprobación:

```
public static class Extensions
{
    public static void BeginInvokeIfRequired(this ISynchronizeInvoke obj, Action action)
    {
        if (obj.InvokeRequired)
            obj.BeginInvoke(action, new object[0]);
        else
            action();
    }
```

```
}
```

Y actualizar el cuadro de texto desde cualquier hilo se vuelve un poco más simple:

```
private void updatetextbox()
{
    textBox1.BeginInvokeIfRequired(() => textBox1.Text = "updated");
}
```

Tenga en cuenta que Control.BeginInvoke como se usa en este ejemplo es asíncrono, lo que significa que el código que viene después de una llamada a Control.BeginInvoke se puede ejecutar inmediatamente después, ya sea que el delegado pasado se haya ejecutado o no.

Si necesita asegurarse de que textBox1 se actualice antes de continuar, use Control.Invoke en su lugar, lo que bloqueará el subproceso de llamada hasta que se haya ejecutado su delegado. Tenga en cuenta que este enfoque puede ralentizar significativamente su código si realiza muchas llamadas de invocación y tenga en cuenta que bloqueará su aplicación si su hilo de GUI está esperando a que el hilo de llamada se complete o libere un recurso retenido.

Lea Enhebrado en línea: https://riptutorial.com/es/dot-net/topic/3098/enhebrado

Capítulo 23: Escribir y leer desde StdErr stream

Examples

Escribir en la salida de error estándar utilizando la consola

```
var sourceFileName = "NonExistingFile";
try
{
    System.IO.File.Copy(sourceFileName, "DestinationFile");
}
catch (Exception e)
{
    var stdErr = Console.Error;
    stdErr.WriteLine($"Failed to copy '{sourceFileName}': {e.Message}");
}
```

Leer de error estándar de proceso hijo

```
var errors = new System.Text.StringBuilder();
var process = new Process
{
    StartInfo = new ProcessStartInfo
    {
        RedirectStandardError = true,
        FileName = "xcopy.exe",
        Arguments = "\"NonExistingFile\" \"DestinationFile\"",
        UseShellExecute = false
    },

};
process.ErrorDataReceived += (s, e) => errors.AppendLine(e.Data);
process.Start();
process.BeginErrorReadLine();
process.WaitForExit();

if (errors.Length > 0) // something went wrong
    System.Console.Error.WriteLine($"Child process error: \r\n {errors}");
```

Lea Escribir y leer desde StdErr stream en línea: https://riptutorial.com/es/dot-net/topic/10779/escribir-y-leer-desde-stderr-stream

Capítulo 24: Examen de la unidad

Examples

Agregar el proyecto de prueba de unidad MSTest a una solución existente

- Haga clic derecho en la solución, Agregar nuevo proyecto
- En la sección Prueba, seleccione un Proyecto de prueba unitaria.
- Elija un nombre para el ensamblaje: si está probando el proyecto `Foo` , el nombre puede ser `Foo.Tests`
- Agregue una referencia al proyecto probado en las referencias del proyecto de prueba unitaria

Creación de un método de prueba de muestra

MSTest (el marco de prueba predeterminado) requiere que sus clases de prueba `[TestClass]` decoradas por un atributo `[TestClass]` y los métodos de prueba con un atributo `[TestMethod]` , y que sea pública.

```
[TestClass]
public class FizzBuzzFixture
{
    [TestMethod]
    public void Test1()
    {
        //arrange
        var solver = new FizzBuzzSolver();
        //act
        var result = solver.FizzBuzz(1);
        //assert
        Assert.AreEqual("1",result);
    }
}
```

Lea Examen de la unidad en línea: https://riptutorial.com/es/dot-net/topic/5365/examen-de-la-unidad

Capítulo 25: Excepciones

Observaciones

Relacionado:

- MSDN: manejo de excepciones y excepciones (Guía de programación de C #)
- MSDN: Manejar y lanzar excepciones
- MSDN: CA1031: no detectar tipos de excepciones generales
- MSDN: try-catch (referencia de C #)

Examples

Atrapando una excepción

El código puede y debe lanzar excepciones en circunstancias excepcionales. Ejemplos de esto incluyen:

- Intentando leer más allá del final de una secuencia
- No tener los permisos necesarios para acceder a un archivo.
- Intentar realizar una operación no válida, como dividir por cero
- Un tiempo de espera que se produce al descargar un archivo de Internet

La persona que llama puede manejar estas excepciones "atrapándolas", y solo debe hacerlo cuando:

- Realmente puede resolver la circunstancia excepcional o recuperarse adecuadamente, o;
- Puede proporcionar un contexto adicional a la excepción que sería útil si la excepción necesita ser relanzada (las excepciones de relanzamiento son capturadas por los controladores de excepciones que se encuentran más arriba en la pila de llamadas)

Se debe tener en cuenta que elegir *no* capturar una excepción es perfectamente válido si la intención es que se maneje en un nivel superior.

La captura de una excepción se realiza envolviendo el código potencialmente lanzador en un bloque `try { ... }` siguiente manera, y capturando las excepciones que puede manejar en un bloque `catch (ExceptionType) { ... }` :

```
Console.Write("Please enter a filename: ");
string filename = Console.ReadLine();

Stream fileStream;

try
{
    fileStream = File.Open(filename);
}
catch (FileNotFoundException)
```

```
{
    Console.WriteLine("File '{0}' could not be found.", filename);
}
```

Usando un bloque Finalmente

El bloque `finally { ... }` de `try-finally` o `try-catch-finally` siempre se ejecutará, independientemente de si se produjo una excepción o no (excepto cuando se ha lanzado una excepción `StackOverflowException` o se ha realizado una llamada a `Environment.FailFast()`).

Puede utilizarse para liberar o limpiar los recursos adquiridos en el bloque `try { ... }` forma segura.

```
Console.Write("Please enter a filename: ");
string filename = Console.ReadLine();

Stream fileStream = null;

try
{
    fileStream = File.Open(filename);
}
catch (FileNotFoundException)
{
    Console.WriteLine("File '{0}' could not be found.", filename);
}
finally
{
    if (fileStream != null)
    {
        fileStream.Dispose();
    }
}
```

Atrapar y volver a captar excepciones.

Cuando desea capturar una excepción y hacer algo, pero no puede continuar la ejecución del bloque de código actual debido a la excepción, es posible que desee volver a emitir la excepción al siguiente controlador de excepciones en la pila de llamadas. Hay buenas y malas maneras de hacer esto.

```
private static void AskTheUltimateQuestion()
{
    try
    {
        var x = 42;
        var y = x / (x - x); // will throw a DivideByZeroException

        // IMPORTANT NOTE: the error in following string format IS intentional
        // and exists to throw an exception to the FormatException catch, below
        Console.WriteLine("The secret to life, the universe, and everything is {1}", y);
    }
    catch (DivideByZeroException)
    {
```

```
        // we do not need a reference to the exception
        Console.WriteLine("Dividing by zero would destroy the universe.");

        // do this to preserve the stack trace:
        throw;
    }
    catch (FormatException ex)
    {
        // only do this if you need to change the type of the Exception to be thrown
        // and wrap the inner Exception

        // remember that the stack trace of the outer Exception will point to the
        // next line

        // you'll need to examine the InnerException property to get the stack trace
        // to the line that actually started the problem

        throw new InvalidOperationException("Watch your format string indexes.", ex);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Something else horrible happened. The exception: " + ex.Message);

        // do not do this, because the stack trace will be changed to point to
        // this location instead of the location where the exception
        // was originally thrown:
        throw ex;
    }
}

static void Main()
{
    try
    {
        AskTheUltimateQuestion();
    }
    catch
    {
        // choose this kind of catch if you don't need any information about
        // the exception that was caught

        // this block "eats" all exceptions instead of rethrowing them
    }
}
```

Puede filtrar por tipo de excepción e incluso por propiedades de excepción (nuevo en C # 6.0, un poco más disponible en VB.NET (citación necesaria)):

Documentación / C # / nuevas funcionalidades

Filtros de excepción

Dado que las excepciones de C # 6.0 se pueden filtrar usando el operador `when` .

Esto es similar a usar un simple `if` pero no desenrolla la pila si no se cumple la condición dentro de `when` .

Ejemplo

```
try
{
  // ...
}
catch (Exception e) when (e.InnerException != null) // Any condition can go in here.
{
  // ...
}
```

La misma información se puede encontrar en las características de C # 6.0 aquí: Filtros de excepción

Recorriendo una excepción dentro de un bloque catch

Dentro de un bloque `catch` , la palabra clave `throw` se puede usar sola, sin especificar un valor de excepción, para *volver* a *emitir* la excepción que se acaba de capturar. La repetición de una excepción permite que la excepción original continúe en la cadena de manejo de excepciones, preservando su pila de llamadas o datos asociados:

```
try {...}
catch (Exception ex) {
  // Note: the ex variable is *not* used
  throw;
}
```

Un antipatrón común es `throw ex` , que tiene el efecto de limitar la siguiente vista del controlador de excepción de la traza de pila:

```
try {...}
catch (Exception ex) {
  // Note: the ex variable is thrown
  //  future stack traces of the exception will not see prior calls
  throw ex;
}
```

En general, no es deseable utilizar la función `throw ex` , ya que los futuros controladores de excepciones que inspeccionan el seguimiento de la pila solo podrán ver las llamadas tan atrás como la `throw ex` . Al omitir la variable `ex` , y al usar solo la palabra clave `throw` , la excepción original se "disparará" .

Lanzar una excepción a partir de un método diferente y preservar su información.

Ocasionalmente, querría atrapar una excepción y lanzarla desde un hilo o método diferente mientras se conserva la pila de excepciones original. Esto se puede hacer con `ExceptionDispatchInfo` :

```
using System.Runtime.ExceptionServices;

void Main()
{
```

```
    ExceptionDispatchInfo capturedException = null;
    try
    {
        throw new Exception();
    }
    catch (Exception ex)
    {
        capturedException = ExceptionDispatchInfo.Capture(ex);
    }

    Foo(capturedException);
}

void Foo(ExceptionDispatchInfo exceptionDispatchInfo)
{
    // Do stuff

    if (capturedException != null)
    {
        // Exception stack trace will show it was thrown from Main() and not from Foo()
        exceptionDispatchInfo.Throw();
    }
}
```

Lea Excepciones en línea: https://riptutorial.com/es/dot-net/topic/33/excepciones

Capítulo 26: Expresiones regulares (System.Text.RegularExpressions)

Examples

Compruebe si el patrón coincide con la entrada

```
public bool Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. W.rld!";

    // true
    return Regex.IsMatch(input, pattern);
}
```

Opciones de paso

```
public bool Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. W.rld!";

    // true
    return Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase | RegexOptions.Singleline);
}
```

Sencilla combinación y reemplazo

```
public string Check()
{
    string input = "Hello World!";
    string pattern = @"W.rld";

    // Hello Stack Overflow!
    return Regex.Replace(input, pattern, "Stack Overflow");
}
```

Partido en grupos

```
public string Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. (?<Subject>W.rld)!";

    Match match = Regex.Match(input, pattern);

    // World
    return match.Groups["Subject"].Value;
```

```
}
```

Eliminar caracteres no alfanuméricos de la cadena

```
public string Remove()
{
    string input = "Hello./!";

    return Regex.Replace(input, "[^a-zA-Z0-9]", "");
}
```

Encontrar todos los partidos

Utilizando

```
using System.Text.RegularExpressions;
```

Código

```
static void Main(string[] args)
{
    string input = "Carrot Banana Apple Cherry Clementine Grape";
    // Find words that start with uppercase 'C'
    string pattern = @"\bC\w*\b";

    MatchCollection matches = Regex.Matches(input, pattern);
    foreach (Match m in matches)
        Console.WriteLine(m.Value);
}
```

Salida

```
Carrot
Cherry
Clementine
```

Lea Expresiones regulares (System.Text.RegularExpressions) en línea: https://riptutorial.com/es/dot-net/topic/6944/expresiones-regulares--system-text-regularexpressions-

Capítulo 27: Flujo de datos TPL

Observaciones

Bibliotecas utilizadas en ejemplos

`System.Threading.Tasks.Dataflow`

`System.Threading.Tasks`

`System.Net.Http`

`System.Net`

Diferencia entre Post y SendAsync

Para agregar elementos a un bloque, puede usar `Post` o `SendAsync` .

`Post` intentará agregar el elemento de forma sincrónica y devolverá un `bool` que `bool` si tuvo éxito o no. Puede que no tenga éxito cuando, por ejemplo, un bloque ha alcanzado su `BoundedCapcity` y no tiene aún más espacio para nuevos elementos. `SendAsync` por otro lado, devolverá una `Task<bool>` incompleta que puede `await` . Esa tarea se completará en el futuro con un resultado `true` cuando el bloque borre su cola interna y pueda aceptar más elementos o con un resultado `false` si se está reduciendo permanentemente (por ejemplo, como resultado de la cancelación).

Examples

Publicar en un ActionBlock y esperar a que se complete

```
// Create a block with an asynchronous action
var block = new ActionBlock<string>(async hostName =>
{
    IPAddress[] ipAddresses = await Dns.GetHostAddressesAsync(hostName);
    Console.WriteLine(ipAddresses[0]);
});

block.Post("google.com"); // Post items to the block's InputQueue for processing
block.Post("reddit.com");
block.Post("stackoverflow.com");

block.Complete(); // Tell the block to complete and stop accepting new items
await block.Completion; // Asynchronously wait until all items completed processingu
```

Enlace de bloques para crear una tubería.

```
var httpClient = new HttpClient();
```

```
// Create a block the accepts a uri and returns its contents as a string
var downloaderBlock = new TransformBlock<string, string>(
    async uri => await httpClient.GetStringAsync(uri));

// Create a block that accepts the content and prints it to the console
var printerBlock = new ActionBlock<string>(
    contents => Console.WriteLine(contents));

// Make the downloaderBlock complete the printerBlock when its completed.
var dataflowLinkOptions = new DataflowLinkOptions {PropagateCompletion = true};

// Link the block to create a pipeline
downloaderBlock.LinkTo(printerBlock, dataflowLinkOptions);

// Post urls to the first block which will pass their contents to the second one.
downloaderBlock.Post("http://youtube.com");
downloaderBlock.Post("http://github.com");
downloaderBlock.Post("http://twitter.com");

downloaderBlock.Complete(); // Completion will propagate to printerBlock
await printerBlock.Completion; // Only need to wait for the last block in the pipeline
```

Productor / Consumidor Sincrónico con BufferBlock

```
public class Producer
{
    private static Random random = new Random((int)DateTime.UtcNow.Ticks);
    //produce the value that will be posted to buffer block
    public double Produce ( )
    {
        var value = random.NextDouble();
        Console.WriteLine($"Producing value: {value}");
        return value;
    }
}

public class Consumer
{
    //consume the value that will be received from buffer block
    public void Consume (double value) => Console.WriteLine($"Consuming value: {value}");
}

class Program
{
    private static BufferBlock<double> buffer = new BufferBlock<double>();
    static void Main (string[] args)
    {
        //start a task that will every 1 second post a value from the producer to buffer block
        var producerTask = Task.Run(async () =>
        {
            var producer = new Producer();
            while(true)
            {
                buffer.Post(producer.Produce());
                await Task.Delay(1000);
            }
        });
        //start a task that will recieve values from bufferblock and consume it
        var consumerTask = Task.Run(() =>
```

```
        {
            var consumer = new Consumer();
            while(true)
            {
                consumer.Consume(buffer.Receive());
            }
        });

        Task.WaitAll(new[] { producerTask, consumerTask });
    }
}
```

Productor asíncrono consumidor con un bloqueo de búfer acotado

```
var bufferBlock = new BufferBlock<int>(new DataflowBlockOptions
{
    BoundedCapacity = 1000
});

var cancellationToken = new CancellationTokenSource(TimeSpan.FromSeconds(10)).Token;

var producerTask = Task.Run(async () =>
{
    var random = new Random();

    while (!cancellationToken.IsCancellationRequested)
    {
        var value = random.Next();
        await bufferBlock.SendAsync(value, cancellationToken);
    }
});

var consumerTask = Task.Run(async () =>
{
    while (await bufferBlock.OutputAvailableAsync())
    {
        var value = bufferBlock.Receive();
        Console.WriteLine(value);
    }
});

await Task.WhenAll(producerTask, consumerTask);
```

Lea Flujo de datos TPL en línea: https://riptutorial.com/es/dot-net/topic/784/flujo-de-datos-tpl

Capítulo 28: Formas VB

Examples

Hola Mundo en Formas VB.NET

Para mostrar un cuadro de mensaje cuando se ha mostrado el formulario:

```
Public Class Form1
    Private Sub Form1_Shown(sender As Object, e As EventArgs) Handles MyBase.Shown
        MessageBox.Show("Hello, World!")
    End Sub
End Class
To show a message box before the form has been shown:

Public Class Form1
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        MessageBox.Show("Hello, World!")
    End Sub
End Class
```

Load () se llamará primero, y solo una vez, cuando se carga el formulario por primera vez. Se llamará a Show () cada vez que el usuario inicie el formulario. Activate () se llamará cada vez que el usuario active el formulario.

Load () se ejecutará antes de llamar a Show (), pero tenga cuidado: llamar a msgBox () en show puede hacer que se ejecute msgBox () antes de que finalice Load (). **En general, es una mala idea depender del orden de los eventos entre Load (), Show () y similares.**

Para principiantes

Algunas cosas que todos los principiantes deben saber / hacer que les ayudarán a tener un buen comienzo con VB .Net:

Establece las siguientes opciones:

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

End Class
```

Utilice &, no + para la concatenación de cadenas. Las cuerdas deben estudiarse con cierto detalle ya que son ampliamente utilizadas.

Dedica algo de tiempo a entender los tipos de valor y referencia .

Nunca utilice Application.DoEvents . Preste atención a la 'Precaución'. Cuando llegue a un punto en el que esto parezca algo que debe usar, pregunte.

La documentación es tu amiga.

Temporizador de formularios

El componente Windows.Forms.Timer se puede usar para proporcionar información de usuario que **no** es crítica en el tiempo. Cree un formulario con un botón, una etiqueta y un componente de temporizador.

Por ejemplo, podría utilizarse para mostrar al usuario la hora del día periódicamente.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Timer1.Interval = 60 * 1000 'one minute intervals
        'start timer
        Timer1.Start()
        Label1.Text = DateTime.Now.ToLongTimeString
    End Sub

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        Label1.Text = DateTime.Now.ToLongTimeString
    End Sub
End Class
```

Pero este temporizador no es adecuado para la temporización. Un ejemplo lo estaría usando para una cuenta atrás. En este ejemplo simularemos una cuenta regresiva a tres minutos. Este puede muy bien ser uno de los ejemplos más aburridos importantes aquí.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        ctSecs = 0 'clear count
        Timer1.Interval = 1000 'one second in ms.
        'start timers
        stpw.Reset()
        stpw.Start()
        Timer1.Start()
    End Sub
```

```
    Dim stpw As New Stopwatch
    Dim ctSecs As Integer

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        ctSecs += 1
        If ctSecs = 180 Then 'about 2.5 seconds off on my PC!
            'stop timing
            stpw.Stop()
            Timer1.Stop()
            'show actual elapsed time
            'Is it near 180?
            Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```

Después de hacer clic en el botón 1, pasan aproximadamente tres minutos y label1 muestra los resultados. ¿Label1 muestra 180? Probablemente no. ¡En mi máquina mostraba 182.5!

El motivo de la discrepancia se encuentra en la documentación, "El componente de Windows Forms Timer es de un solo hilo y está limitado a una precisión de 55 milisegundos". Esta es la razón por la que no debe utilizarse para la sincronización.

Al usar el temporizador y el cronómetro de manera un poco diferente, podemos obtener mejores resultados.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Timer1.Interval = 100 'one tenth of a second in ms.
        'start timers
        stpw.Reset()
        stpw.Start()
        Timer1.Start()
    End Sub

    Dim stpw As New Stopwatch
    Dim threeMinutes As TimeSpan = TimeSpan.FromMinutes(3)

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        If stpw.Elapsed >= threeMinutes Then '0.1 off on my PC!
            'stop timing
            stpw.Stop()
            Timer1.Stop()
            'show actual elapsed time
            'how close?
            Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```

Hay otros temporizadores que pueden usarse según sea necesario. Esta búsqueda debería ayudar en ese sentido.

Lea Formas VB en línea: https://riptutorial.com/es/dot-net/topic/2197/formas-vb

Capítulo 29: Gestión de la memoria

Observaciones

Las aplicaciones críticas para el rendimiento en aplicaciones administradas .NET pueden verse gravemente afectadas por el GC. Cuando se ejecuta el GC, todos los demás subprocesos se suspenden hasta que se completa. Por esta razón, se recomienda evaluar cuidadosamente los procesos de GC y determinar cómo minimizar cuando se ejecuta.

Examples

Recursos no gestionados

Cuando hablamos sobre el GC y el "montón", realmente estamos hablando de lo que se llama el *montón administrado* . Los objetos en el *montón administrado* pueden acceder a recursos que no están en el montón administrado, por ejemplo, al escribir o leer desde un archivo. Se puede producir un comportamiento inesperado cuando se abre un archivo para su lectura y luego se produce una excepción, lo que impide que el identificador del archivo se cierre como lo haría normalmente. Por esta razón, .NET requiere que los recursos no administrados implementen la interfaz `IDisposable` . Esta interfaz tiene un solo método llamado `Dispose` sin parámetros:

```
public interface IDisposable
{
    Dispose();
}
```

Cuando maneje recursos no administrados, debe asegurarse de que estén correctamente dispuestos. Puede hacerlo llamando explícitamente a `Dispose()` en un bloque `finally` o con una instrucción `using` .

```
StreamReader sr;
string textFromFile;
string filename = "SomeFile.txt";
try
{
    sr = new StreamReader(filename);
    textFromFile = sr.ReadToEnd();
}
finally
{
    if (sr != null) sr.Dispose();
}
```

o

```
string textFromFile;
string filename = "SomeFile.txt";
```

```
using (StreamReader sr = new Streamreader(filename))
{
    textFromFile = sr.ReadToEnd();
}
```

Este último es el método preferido y se expande automáticamente al primero durante la compilación.

Utilice SafeHandle cuando ajuste recursos no administrados

Al escribir envoltorios para recursos no administrados, debe `SafeHandle` subclase de `SafeHandle` lugar de intentar implementar `IDisposable` y un finalizador. Su subclase `SafeHandle` debe ser lo más pequeña y simple posible para minimizar la posibilidad de una fuga en el mango. Esto probablemente significa que su implementación de SafeHandle sería un detalle de implementación interna de una clase que lo envuelve para proporcionar una API utilizable. Esta clase garantiza que, incluso si un programa `SafeHandle` su instancia de `SafeHandle` , se libera su identificador no administrado.

```
using System.Runtime.InteropServices;

class MyHandle : SafeHandle
{
    public override bool IsInvalid => handle == IntPtr.Zero;
    public MyHandle() : base(IntPtr.Zero, true)
    { }

    public MyHandle(int length) : this()
    {
        SetHandle(Marshal.AllocHGlobal(length));
    }

    protected override bool ReleaseHandle()
    {
        Marshal.FreeHGlobal(handle);
        return true;
    }
}
```

Descargo de responsabilidad: este ejemplo es un intento de mostrar cómo proteger un recurso administrado con `SafeHandle` que implementa `IDisposable` para usted y configura los finalizadores de manera adecuada. Es muy artificial y probablemente no tenga sentido asignar una porción de memoria de esta manera.

Lea Gestión de la memoria en línea: https://riptutorial.com/es/dot-net/topic/59/gestion-de-la-memoria

Capítulo 30: Globalización en ASP.NET MVC utilizando la internacionalización inteligente para ASP.NET

Observaciones

Inteligente internacionalización para la página ASP.NET

El beneficio de este enfoque es que no tiene que saturar los controladores y otras clases con código para buscar valores de archivos .resx. Simplemente rodee el texto entre [[[corchetes triples.]]] (El delimitador es configurable.) Un `HttpModule` busca una traducción en su archivo .po para reemplazar el texto delimitado. Si se encuentra una traducción, `HttpModule` sustituye a la traducción. Si no se encuentra ninguna traducción, elimina los corchetes triples y renderiza la página con el texto original sin traducir.

Los archivos .po son un formato estándar para suministrar traducciones para aplicaciones, por lo que hay una serie de aplicaciones disponibles para editarlas. Es fácil enviar un archivo .po a un usuario no técnico para que puedan agregar traducciones.

Examples

Configuración básica y configuración

1. Agregue el paquete nuget I18N a su proyecto MVC.
2. En web.config, agregue `i18n.LocalizingModule` a su `<httpModules>` o `<modules>` .

```
<!-- IIS 6 -->
<httpModules>
  <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
</httpModules>

<!-- IIS 7 -->
<system.webServer>
  <modules>
    <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
  </modules>
</system.webServer>
```

3. Agregue una carpeta llamada "locale" a la raíz de su sitio. Cree una subcarpeta para cada cultura que desee apoyar. Por ejemplo, `/locale/fr/` .
4. En cada carpeta específica de cada cultura, cree un archivo de texto llamado `messages.po` .
5. Para propósitos de prueba, ingrese las siguientes líneas de texto en su archivo `messages.po` :

```
#: Translation test
msgid "Hello, world!"
```

```
msgstr "Bonjour le monde!"
```

6. Agregue un controlador a su proyecto que devuelva texto para traducir.

```
using System.Web.Mvc;

namespace I18nDemo.Controllers
{
    public class DefaultController : Controller
    {
        public ActionResult Index()
        {
            // Text inside [[[triple brackets]]] must precisely match
            // the msgid in your .po file.
            return Content("[[[Hello, world!]]]");
        }
    }
}
```

7. Ejecute su aplicación MVC y busque la ruta correspondiente a la acción de su controlador, como http: // localhost: [su número de puerto] / predeterminado .
   Observe que la URL se ha cambiado para reflejar su cultura predeterminada, como http: // localhost: [yourportnumber] / es / default .
8. Reemplace `/en/` en la URL con `/fr/` (o la cultura que haya seleccionado). La página ahora debe mostrar la versión traducida de su texto.
9. Cambie la configuración de idioma de su navegador para preferir su cultura alternativa y busque `/default` nuevamente. Observe que la URL se ha cambiado para reflejar su cultura alternativa y aparece el texto traducido.
10. En web.config, agregue controladores para que los usuarios no puedan navegar a su carpeta de `locale` .

```
<!-- IIS 6 -->
<system.web>
  <httpHandlers>
    <add path="*" verb="*" type="System.Web.HttpNotFoundHandler"/>
  </httpHandlers>
</system.web>

<!-- IIS 7 -->
<system.webServer>
  <handlers>
    <remove name="BlockViewHandler"/>
   <add name="BlockViewHandler" path="*" verb="*" preCondition="integratedMode"
type="System.Web.HttpNotFoundHandler"/>
  </handlers>
</system.webServer>
```

Lea Globalización en ASP.NET MVC utilizando la internacionalización inteligente para ASP.NET en línea: https://riptutorial.com/es/dot-net/topic/5086/globalizacion-en-asp-net-mvc-utilizando-la-internacionalizacion-inteligente-para-asp-net

Capítulo 31: Instrumentos de cuerda

Observaciones

En las cadenas .NET `System.String` son secuencias de caracteres `System.Char` , cada carácter es una unidad de código codificada en UTF-16. Esta distinción es importante porque la definición de *caracteres en el lenguaje hablado y la* definición de *caracteres en* .NET (y en muchos otros idiomas) son diferentes.

Un *carácter* , que debería llamarse correctamente grafema , se muestra como un glifo y está definido por uno o más puntos de código Unicode. Cada punto de código se codifica en una secuencia de unidades de código . Ahora debería estar claro por qué un solo `System.Char` no siempre representa un grafema, veamos en el mundo real cómo son diferentes:

- Un grafema, debido a la combinación de caracteres , puede dar lugar a dos o más puntos de código: `à` está compuesto por dos puntos de código: *U + 0061 LETRA PEQUEÑA LETRA A* y *U + 0300 COMBINACIÓN DEL ACUMENTO GRAVE* . Este es el error más común porque `"à ".Length == 2` mientras que usted puede esperar `1` .
- Hay caracteres duplicados, por ejemplo `à` puede ser un único punto de código *U + 00E0 LETRA A LATINA PEQUEÑA CON GRAVE* o dos puntos de código como se explicó anteriormente. Obviamente, deben comparar lo mismo: `"\u00e0" == "\u0061\u0300"` (incluso si `"\u00e0".Length != "\u0061\u0300".Length`). Esto es posible debido a la *normalización* de la *cadena* realizada por el método `String.Normalize()` .
- Una secuencia Unicode puede contener una secuencia compuesta o descompuesta, por ejemplo, el carácter □ *U + D55C HAN CHARACTER* puede ser un único punto de código (codificado como una sola unidad de código en UTF-16) o una secuencia descompuesta de sus sílabas □ , □ y □ . Deben ser comparados iguales.
- Un punto de código puede codificarse en más de una unidad de código: el carácter □ *U + 2008A HAN CHARACTER* se codifica como dos `System.Char` (`"\ud840\udc8a"`) incluso si es solo un punto de código: UTF-16 La codificación no es de tamaño fijo! Esta es una fuente de innumerables errores (también errores graves de seguridad), si, por ejemplo, su aplicación aplica una longitud máxima y trunca ciegamente la cadena, entonces puede crear una cadena no válida.
- Algunos idiomas tienen digraph y trigraphs, por ejemplo, en `ch ch ch` es una letra independiente (después de `h` y antes de `i` , cuando ordene una lista de cadenas, tendrá *fyzika* antes de *chemie* .

Hay muchos más problemas relacionados con el manejo del texto. Consulte, por ejemplo, ¿Cómo puedo realizar una comparación de caracteres por caracteres en Unicode? para una introducción más amplia y más enlaces a argumentos relacionados.

En general, cuando se trata de texto *internacional* , puede usar esta función simple para enumerar elementos de texto en una cadena (evitando romper los sustitutos y la codificación de Unicode):

```
public static class StringExtensions
{
```

```
    public static IEnumerable<string> EnumerateCharacters(this string s)
    {
        if (s == null)
            return Enumerable.Empty<string>();

        var enumerator = StringInfo.GetTextElementEnumerator(s.Normalize());
        while (enumerator.MoveNext())
            yield return (string)enumerator.Value;
    }
}
```

Examples

Contar personajes distintos

Si necesita contar caracteres distintos, entonces, por las razones explicadas en la sección *Comentarios* , no puede simplemente usar la propiedad `Length` porque es la longitud de la matriz de `System.Char` que no son caracteres sino unidades de código (no puntos de código Unicode). ni grafemas). Si, por ejemplo, simplemente escribe `text.Distinct().Count()` obtendrá resultados incorrectos, código correcto:

```
int distinctCharactersCount = text.EnumerateCharacters().Count();
```

Un paso más es **contar las ocurrencias de cada personaje** , si el rendimiento no es un problema, simplemente puede hacerlo así (en este ejemplo, independientemente del caso):

```
var frequencies = text.EnumerateCharacters()
    .GroupBy(x => x, StringComparer.CurrentCultureIgnoreCase)
    .Select(x => new { Character = x.Key, Count = x.Count() };
```

Contar personajes

Si necesita contar *caracteres* , entonces, por las razones explicadas en la sección *Comentarios* , no puede simplemente usar la propiedad Longitud porque es la longitud de la matriz de `System.Char` que no son caracteres sino unidades de código (no son puntos de código Unicode ni grafemas). El código correcto es entonces:

```
int length = text.EnumerateCharacters().Count();
```

Una pequeña optimización puede reescribir el método de extensión `EnumerateCharacters()` específicamente para este propósito:

```
public static class StringExtensions
{
    public static int CountCharacters(this string text)
    {
        if (String.IsNullOrEmpty(text))
            return 0;

        int count = 0;
```

```
        var enumerator = StringInfo.GetTextElementEnumerator(text);
        while (enumerator.MoveNext())
            ++count;

        return count;
    }
}
```

Contar las ocurrencias de un personaje.

Debido a las razones explicadas en la sección de *Comentarios* , no puede simplemente hacer esto (a menos que quiera contar las ocurrencias de una unidad de código específica):

```
int count = text.Count(x => x == ch);
```

Necesitas una función más compleja:

```
public static int CountOccurrencesOf(this string text, string character)
{
    return text.EnumerateCharacters()
        .Count(x => String.Equals(x, character, StringComparer.CurrentCulture));
}
```

Tenga en cuenta que la comparación de cadenas (en contraste con la comparación de caracteres que es invariante de la cultura) siempre debe realizarse de acuerdo con las reglas de una cultura específica.

Dividir la cadena en bloques de longitud fija

No podemos dividir una cadena en puntos arbitrarios (porque un `System.Char` puede no ser válido solo porque es un carácter de combinación o parte de un sustituto), entonces el código debe tener eso en cuenta (tenga en cuenta que con la *longitud* me refiero al número de *grafemas,* no al número de *unidades* de *código*):

```
public static IEnumerable<string> Split(this string value, int desiredLength)
{
    var characters = StringInfo.GetTextElementEnumerator(value);
    while (characters.MoveNext())
        yield return String.Concat(Take(characters, desiredLength));
}

private static IEnumerable<string> Take(TextElementEnumerator enumerator, int count)
{
    for (int i = 0; i < count; ++i)
    {
        yield return (string)enumerator.Current;

        if (!enumerator.MoveNext())
            yield break;
    }
}
```

Convertir cadena a / desde otra codificación

Las cadenas .NET contienen `System.Char` (unidades de código UTF-16). Si desea guardar (o administrar) texto con otra codificación, debe trabajar con una matriz de `System.Byte` .

Las conversiones se realizan por clases derivadas de `System.Text.Encoder` y `System.Text.Decoder` que, juntas, pueden convertir a / desde otra codificación (de un byte *X* byte byte codificado `byte[]` a un sistema codificado en UTF- `System.String` y vice -versa).

Debido a que el codificador / decodificador generalmente funciona muy cerca uno del otro, se agrupan en una clase derivada de `System.Text.Encoding` , las clases derivadas ofrecen conversiones a / desde codificaciones populares (UTF-8, UTF-16 y así sucesivamente).

Ejemplos:

Convertir una cadena a UTF-8

```
byte[] data = Encoding.UTF8.GetBytes("This is my text");
```

Convertir datos UTF-8 a una cadena

```
var text = Encoding.UTF8.GetString(data);
```

Cambiar la codificación de un archivo de texto existente

Este código leerá el contenido de un archivo de texto codificado en UTF-8 y lo guardará nuevamente codificado como UTF-16. Tenga en cuenta que este código no es óptimo si el archivo es grande porque leerá todo su contenido en la memoria:

```
var content = File.ReadAllText(path, Encoding.UTF8);
File.WriteAllText(content, Encoding.UTF16);
```

Object.ToString () método virtual

Todo en .NET es un objeto, por lo tanto, cada tipo tiene el método `ToString()` definido en la clase `Object` que puede ser anulado. La implementación predeterminada de este método solo devuelve el nombre del tipo:

```
public class Foo
{
}

var foo = new Foo();
```

```
Console.WriteLine(foo); // outputs Foo
```

`ToString()` se llama implícitamente cuando concatina un valor con una cadena:

```
public class Foo
{
    public override string ToString()
    {
        return "I am Foo";
    }
}

var foo = new Foo();
Console.WriteLine("I am bar and "+foo);// outputs I am bar and I am Foo
```

El resultado de este método también es ampliamente utilizado por las herramientas de depuración. Si, por alguna razón, no desea anular este método, pero desea personalizar cómo el depurador muestra el valor de su tipo, use el atributo DebuggerDisplay (MSDN):

```
// [DebuggerDisplay("Person = FN {FirstName}, LN {LastName}")]
[DebuggerDisplay("Person = FN {"+nameof(Person.FirstName)+"}, LN
{"+nameof(Person.LastName)+"}")]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set;}
    // ...
}
```

Inmutabilidad de las cuerdas.

Las cuerdas son inmutables. Simplemente no puedes cambiar la cadena existente. Cualquier operación en la cadena crea una nueva instancia de la cadena con un nuevo valor. Esto significa que si necesita reemplazar un solo carácter en una cadena muy larga, la memoria se asignará para un nuevo valor.

```
string veryLongString = ...
// memory is allocated
string newString = veryLongString.Remove(0,1); // removes first character of the string.
```

Si necesita realizar muchas operaciones con valor de cadena, use la clase `StringBuilder` que está diseñada para la manipulación eficiente de cadenas:

```
var sb = new StringBuilder(someInitialString);
foreach(var str in manyManyStrings)
{
    sb.Append(str);
}
var finalString = sb.ToString();
```

Cuerdas

A pesar de que `String` es un tipo de referencia, el operador `==` compara los valores de cadena en lugar de las referencias.

Como sabrás, la `string` es solo una matriz de caracteres. Pero si crees que la verificación y comparación de la igualdad de las cadenas se hace carácter por carácter, estás equivocado. Esta operación es específica de la cultura (ver Comentarios más abajo): algunas secuencias de caracteres pueden tratarse como iguales dependiendo de la cultura .

Piense dos veces antes de la verificación de la igualdad en el cortocircuito comparando las propiedades de `Length` de dos cadenas.

Use sobrecargas del método `String.Equals` que acepte el valor de enumeración `StringComparison` adicional, si necesita cambiar el comportamiento predeterminado.

Lea Instrumentos de cuerda en línea: https://riptutorial.com/es/dot-net/topic/2227/instrumentos-de-cuerda

Capítulo 32: Invocación de plataforma

Sintaxis

- [DllImport ("Example.dll")] static extern void SetText (string inString);
- [DllImport ("Example.dll")] static extern void GetText (StringBuilder outString);
- Texto de cadena [MarshalAs (UnmanagedType.ByValTStr, SizeConst = 32)];
- [MarshalAs (UnmanagedType.ByValArray, SizeConst = 128)] byte [] byteArr;
- [StructLayout (LayoutKind.Sequential)] public struct PERSON {...}
- [StructLayout (LayoutKind.Explicit)] public struct MarshaledUnion {[FieldOffset (0)] ...}

Examples

Llamando a una función dll Win32

```
using System.Runtime.InteropServices;

class PInvokeExample
{
    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    public static extern uint MessageBox(IntPtr hWnd, String text, String caption, int
options);

    public static void test()
    {
        MessageBox(IntPtr.Zero, "Hello!", "Message", 0);
    }
}
```

Declare una función como `static extern` stting `DllImportAttribute` con su propiedad `Value` establecida en .dll nombre. No olvide utilizar el `System.Runtime.InteropServices` nombres `System.Runtime.InteropServices` . Entonces llámalo como un método estático regular.

Los Servicios de Invocación de Plataforma se encargarán de cargar el archivo .dll y encontrar la función deseada. El P / Invoke en la mayoría de los casos simples también calculará los parámetros y devolverá el valor ay desde .dll (es decir, convertir de tipos de datos .NET a Win32 y viceversa).

Usando la API de Windows

Utilice pinvoke.net .

Antes de declarar una función API `extern` Windows en su código, considere buscarla en pinvoke.net . Lo más probable es que ya tengan una declaración adecuada con todos los tipos de soporte y buenos ejemplos.

Arreglando matrices

Arreglos de tipo simple.

```
[DllImport("Example.dll")]
static extern void SetArray(
    [MarshalAs(UnmanagedType.LPArray, SizeConst = 128)]
    byte[] data);
```

Matrices de cuerda

```
[DllImport("Example.dll")]
static extern void SetStrArray(string[] textLines);
```

Estructuras de cálculo

Estructura simple

Firma C ++:

```
typedef struct _PERSON
{
    int age;
    char name[32];
} PERSON, *LP_PERSON;

void GetSpouse(PERSON person, LP_PERSON spouse);
```

Definición de C #

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct PERSON
{
    public int age;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
    public string name;
}

[DllImport("family.dll", CharSet = CharSet.Auto)]
public static extern bool GetSpouse(PERSON person, ref PERSON spouse);
```

Struct con campos de matriz de tamaño desconocido. Pasando en

Firma C ++

```
typedef struct
{
    int length;
    int *data;
} VECTOR;

void SetVector(VECTOR &vector);
```

Cuando se pasa de código administrado a no administrado, este

La matriz de `data` debe definirse como IntPtr y la memoria debe asignarse explícitamente con `Marshal.AllocHGlobal()` (y liberarse con las `Marshal.FreeHGlobal()` palabras `Marshal.FreeHGlobal()`):

```
[StructLayout(LayoutKind.Sequential)]
public struct VECTOR : IDisposable
{
    int length;
    IntPtr dataBuf;

    public int[] data
    {
        set
        {
            FreeDataBuf();
            if (value != null && value.Length > 0)
            {
                dataBuf = Marshal.AllocHGlobal(value.Length * Marshal.SizeOf(value[0]));
                Marshal.Copy(value, 0, dataBuf, value.Length);
                length = value.Length;
            }
        }
    }
    void FreeDataBuf()
    {
        if (dataBuf != IntPtr.Zero)
        {
            Marshal.FreeHGlobal(dataBuf);
            dataBuf = IntPtr.Zero;
        }
    }
    public void Dispose()
    {
        FreeDataBuf();
    }
}

[DllImport("vectors.dll")]
public static extern void SetVector([In]ref VECTOR vector);
```

Struct con campos de matriz de tamaño desconocido. Recepción

Firma C ++:

```
typedef struct
{
    char *name;
} USER;

bool GetCurrentUser(USER *user);
```

Cuando dichos datos pasan de código no administrado y la memoria es asignada por las funciones no administradas, el llamante administrado debe recibirla en una variable `IntPrt` y convertir el búfer en una matriz administrada. En el caso de cadenas, hay un método `Marshal.PtrToStringAnsi()` conveniente:

```
[StructLayout(LayoutKind.Sequential)]
```

```
public struct USER
{
    IntPtr nameBuffer;
    public string name { get { return Marshal.PtrToStringAnsi(nameBuffer); } }
}

[DllImport("users.dll")]
public static extern bool GetCurrentUser(out USER user);
```

Uniendo las uniones

Solo campos de valor

Declaración de C ++

```
typedef union
{
    char c;
    int i;
} CharOrInt;
```

Declaración de C #

```
[StructLayout(LayoutKind.Explicit)]
public struct CharOrInt
{
    [FieldOffset(0)]
    public byte c;
    [FieldOffset(0)]
    public int i;
}
```

Mezcla de tipo valor y campos de referencia.

La superposición de un valor de referencia con un tipo de valor uno no está permitida, por lo que no puede simplemente usar el ~~`FieldOffset(0) text; FieldOffset(0) i;`~~-no compilará para

```
typedef union
{
    char text[128];
    int i;
} TextOrInt;
```

y en general, tendría que emplear el cálculo de personalización personalizado. Sin embargo, en casos particulares como esta se pueden usar técnicas más simples:

```
[StructLayout(LayoutKind.Sequential)]
public struct TextOrInt
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 128)]
    public byte[] text;
    public int i { get { return BitConverter.ToInt32(text, 0); } }
}
```

Lea Invocación de plataforma en línea: https://riptutorial.com/es/dot-net/topic/1643/invocacion-de-plataforma

Capítulo 33: Inyección de dependencia

Observaciones

Problemas resueltos por inyección de dependencia

Si no utilizáramos la inyección de dependencia, la clase `Greeter` podría verse más como esto:

```
public class ControlFreakGreeter
{
    public void Greet()
    {
        var greetingProvider = new SqlGreetingProvider(
            ConfigurationManager.ConnectionStrings["myConnectionString"].ConnectionString);
        var greeting = greetingProvider.GetGreeting();
        Console.WriteLine(greeting);
    }
}
```

Es un "control freak" porque controla la creación de la clase que proporciona el saludo, controla de dónde proviene la cadena de conexión SQL y controla la salida.

Al usar la inyección de dependencia, la clase `Greeter` renuncia a esas responsabilidades en favor de una sola responsabilidad, escribiendo un saludo que se le proporciona.

El principio de inversión de dependencia sugiere que las clases deberían depender de abstracciones (como interfaces) en lugar de otras clases concretas. Las dependencias directas (acoplamiento) entre clases pueden dificultar progresivamente el mantenimiento. Dependiendo de las abstracciones se puede reducir ese acoplamiento.

La inyección de dependencia nos ayuda a lograr esa inversión de dependencia porque lleva a escribir clases que dependen de abstracciones. La clase `Greeter` "no sabe" nada en absoluto sobre los detalles de implementación de `IGreetingProvider` e `IGreetingWriter` . Solo se sabe que las dependencias inyectadas implementan esas interfaces. Eso significa que los cambios en las clases concretas que implementan `IGreetingProvider` e `IGreetingWriter` no afectarán a `Greeter` . Tampoco los reemplazará con implementaciones completamente diferentes. Solo los cambios en las interfaces lo harán. `Greeter` está desacoplado.

`ControlFreakGreeter` es imposible realizar una prueba de unidad correctamente. Queremos probar una pequeña unidad de código, pero en lugar de eso, nuestra prueba incluiría conectarse a SQL y ejecutar un procedimiento almacenado. También incluiría probar la salida de la consola. Debido a que ControlFreakGreeter hace tanto, es imposible realizar pruebas aisladas de otras clases.

`Greeter` es fácil de realizar una prueba unitaria porque podemos inyectar implementaciones simuladas de sus dependencias que son más fáciles de ejecutar y verificar que llamar a un procedimiento almacenado o leer la salida de la consola. No requiere una cadena de conexión en app.config.

Las implementaciones concretas de `IGreetingProvider` y `IGreetingWriter` pueden ser más complejas. Ellos, a su vez, pueden tener sus propias dependencias que se inyectan en ellos. (Por ejemplo, inyectaríamos la cadena de conexión SQL en `SqlGreetingProvider`). Pero esa complejidad está "oculta" de otras clases que solo dependen de las interfaces. Eso hace que sea más fácil modificar una clase sin un "efecto dominó" que requiere que realicemos los cambios correspondientes a otras clases.

Examples

Inyección de dependencia - Ejemplo simple

Esta clase se llama `Greeter` . Su responsabilidad es dar un saludo. Tiene dos *dependencias* . Necesita algo que le dé el saludo de salida, y luego necesita una forma de emitir ese saludo. Esas dependencias se describen como interfaces, `IGreetingProvider` e `IGreetingWriter` . En este ejemplo, esas dos dependencias se "inyectan" en `Greeter` . (Explicación adicional siguiendo el ejemplo).

```
public class Greeter
{
    private readonly IGreetingProvider _greetingProvider;
    private readonly IGreetingWriter _greetingWriter;

    public Greeter(IGreetingProvider greetingProvider, IGreetingWriter greetingWriter)
    {
        _greetingProvider = greetingProvider;
        _greetingWriter = greetingWriter;
    }

    public void Greet()
    {
        var greeting = _greetingProvider.GetGreeting();
        _greetingWriter.WriteGreeting(greeting);
    }
}

public interface IGreetingProvider
{
    string GetGreeting();
}

public interface IGreetingWriter
{
    void WriteGreeting(string greeting);
}
```

La clase de `Greeting` depende tanto de `IGreetingProvider` como de `IGreetingWriter` , pero no es responsable de crear instancias de ninguno de los dos. En su lugar los requiere en su constructor. Cualquier cosa que cree una instancia de `Greeting` debe proporcionar esas dos dependencias. Podemos llamar a eso "inyectar" las dependencias.

Debido a que las dependencias se proporcionan a la clase en su constructor, esto también se denomina "inyección de constructor".

Algunas convenciones comunes:

- El constructor guarda las dependencias como campos `private` . Tan pronto como se crea una instancia de la clase, esas dependencias están disponibles para todos los otros métodos no estáticos de la clase.
- Los campos `private` son de `readonly` . Una vez que se establecen en el constructor, no se pueden cambiar. Esto indica que esos campos no deben (y no pueden) ser modificados fuera del constructor. Eso asegura aún más que esas dependencias estarán disponibles durante toda la vida de la clase.
- Las dependencias son interfaces. Esto no es estrictamente necesario, pero es común porque facilita la sustitución de una implementación de la dependencia por otra. También permite proporcionar una versión simulada de la interfaz para propósitos de prueba unitaria.

Cómo la inyección de dependencia hace que las pruebas unitarias sean más fáciles

Esto se basa en el ejemplo anterior de la clase `Greeter` que tiene dos dependencias, `IGreetingProvider` e `IGreetingWriter` .

La implementación real de `IGreetingProvider` podría recuperar una cadena de una llamada a la API o una base de datos. La implementación de `IGreetingWriter` puede mostrar el saludo en la consola. Pero como `Greeter` tiene sus dependencias inyectadas en su constructor, es fácil escribir una prueba de unidad que inyecta versiones simuladas de esas interfaces. En la vida real podríamos usar un marco como Moq , pero en este caso escribiré esas implementaciones burladas.

```
public class TestGreetingProvider : IGreetingProvider
{
    public const string TestGreeting = "Hello!";

    public string GetGreeting()
    {
        return TestGreeting;
    }
}

public class TestGreetingWriter : List<string>, IGreetingWriter
{
    public void WriteGreeting(string greeting)
    {
        Add(greeting);
    }
}

[TestClass]
public class GreeterTests
{
    [TestMethod]
    public void Greeter_WritesGreeting()
    {
        var greetingProvider = new TestGreetingProvider();
        var greetingWriter = new TestGreetingWriter();
        var greeter = new Greeter(greetingProvider, greetingWriter);
```

```
        greeter.Greet();
        Assert.AreEqual(greetingWriter[0], TestGreetingProvider.TestGreeting);
    }
}
```

El comportamiento de `IGreetingProvider` y `IGreetingWriter` no son relevantes para esta prueba. Queremos probar que `Greeter` recibe un saludo y lo escribe. El diseño de `Greeter` (mediante inyección de dependencia) nos permite inyectar dependencias simuladas sin partes móviles complicadas. Todo lo que estamos probando es que `Greeter` interactúa con esas dependencias como esperamos.

Por qué usamos contenedores de inyección de dependencia (contenedores IoC)

Inyección de dependencia significa clases de escritura para que no controlen sus dependencias; en cambio, se les proporcionan sus dependencias ("inyectadas").

Esto no es lo mismo que usar un marco de inyección de dependencias (a menudo denominado "contenedor DI", "contenedor IoC" o simplemente "contenedor") como Castle Windsor, Autofac, SimpleInjector, Ninject, Unity u otros.

Un contenedor simplemente facilita la inyección de dependencia. Por ejemplo, suponga que escribe una cantidad de clases que dependen de la inyección de dependencia. Una clase depende de varias interfaces, las clases que implementan esas interfaces dependen de otras interfaces, etc. Algunos dependen de valores específicos. Y solo por diversión, algunas de esas clases implementan `IDisposable` y deben eliminarse.

Cada clase individual está bien escrita y es fácil de evaluar. Pero ahora hay un problema diferente: crear una instancia de una clase se ha vuelto mucho más complicado. Supongamos que estamos creando una instancia de una clase `CustomerService` . Tiene dependencias y sus dependencias tienen dependencias. Construir una instancia puede verse algo como esto:

```
public CustomerData GetCustomerData(string customerNumber)
{
    var customerApiEndpoint =
ConfigurationManager.AppSettings["customerApi:customerApiEndpoint"];
    var logFilePath = ConfigurationManager.AppSettings["logwriter:logFilePath"];
    var authConnectionString =
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString;
    using(var logWriter = new LogWriter(logFilePath ))
    {
        using(var customerApiClient = new CustomerApiClient(customerApiEndpoint))
        {
            var customerService = new CustomerService(
                new SqlAuthorizationRepository(authorizationConnectionString, logWriter),
                new CustomerDataRepository(customerApiClient, logWriter),
                logWriter
            );

            // All this just to create an instance of CustomerService!
            return customerService.GetCustomerData(string customerNumber);
        }
    }
```

```
}
```

Podría preguntarse, ¿por qué no poner toda la construcción gigante en una función separada que simplemente devuelve `CustomerService` ? Una razón es que debido a que las dependencias para cada clase se inyectan en ella, una clase no es responsable de saber si esas dependencias son `IDisposable` o desecharlas. Simplemente los usa. Entonces, si tuviéramos una función `GetCustomerService()` que devolviera un `CustomerService` completamente construido, esa clase podría contener una cantidad de recursos desechables y ninguna forma de acceder o desecharlos.

Y aparte de desechar `IDisposable` , ¿quién quiere llamar a una serie de constructores anidados como ese, alguna vez? Ese es un pequeño ejemplo. Podría ser mucho, mucho peor. Nuevamente, eso no significa que escribimos las clases de manera incorrecta. Las clases pueden ser individualmente perfectas. El reto es componerlos juntos.

Un contenedor de inyección de dependencia simplifica eso. Nos permite especificar qué clase o valor debe usarse para cumplir con cada dependencia. Este ejemplo ligeramente simplificado utiliza Castle Windsor:

```
var container = new WindsorContainer()
container.Register(
    Component.For<CustomerService>(),
    Component.For<ILogWriter, LogWriter>()
        .DependsOn(Dependency.OnAppSettingsValue("logFilePath", "logWriter:logFilePath")),
    Component.For<IAuthorizationRepository, SqlAuthorizationRepository>()
        .DependsOn(Dependency.OnValue(connectionString,
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString)),
    Component.For<ICustomerDataProvider, CustomerApiClient>()
         .DependsOn(Dependency.OnAppSettingsValue("apiEndpoint",
"customerApi:customerApiEndpoint"))
);
```

A esto lo llamamos "registrar dependencias" o "configurar el contenedor". Traducido, esto le dice a nuestro `WindsorContainer` :

- Si una clase requiere `ILogWriter` , cree una instancia de `LogWriter` . `LogWriter` requiere una ruta de archivo. Utilice este valor de `AppSettings` .
- Si una clase requiere `IAuthorizationRepository` , cree una instancia de `SqlAuthorizationRepository` . Requiere una cadena de conexión. Utilice este valor de la sección `ConnectionStrings` .
- Si una clase requiere `ICustomerDataProvider` , cree un `CustomerApiClient` y proporcione la cadena que necesita desde `AppSettings` .

Cuando solicitamos una dependencia del contenedor, llamamos a eso "resolver" una dependencia. Es una mala práctica hacer eso directamente usando el contenedor, pero esa es una historia diferente. Para propósitos de demostración, ahora podríamos hacer esto:

```
var customerService = container.Resolve<CustomerService>();
var data = customerService.GetCustomerData(customerNumber);
container.Release(customerService);
```

El contenedor sabe que `CustomerService` depende de `IAuthorizationRepository` y `ICustomerDataProvider` . Sabe qué clases necesita crear para cumplir con esos requisitos. Esas clases, a su vez, tienen más dependencias, y el contenedor sabe cómo cumplirlas. Creará todas las clases que necesite hasta que pueda devolver una instancia de `CustomerService` .

Si llega a un punto en el que una clase requiere una dependencia que no hemos registrado, como `IDoesSomethingElse` , cuando intentemos resolver el servicio al `CustomerService` se producirá una excepción clara que nos indica que no hemos registrado nada para cumplir con ese requisito.

Cada marco DI se comporta de manera un poco diferente, pero generalmente nos dan cierto control sobre cómo se ejemplifican ciertas clases. Por ejemplo, ¿queremos que cree una instancia de `LogWriter` y la proporcione a todas las clases que dependen de `ILogWriter` , o queremos que cree una nueva cada vez? La mayoría de los contenedores tienen una manera de especificar eso.

¿Qué pasa con las clases que implementan `IDisposable` ? Por eso llamamos `container.Release(customerService);` al final. La mayoría de los contenedores (incluyendo Windsor), avanzará hacia atrás a través de todas las dependencias creadas y `Dispose` los que necesitan de desecharlas. Si `CustomerService` es `IDisposable` también lo eliminará.

Registrar dependencias como se ve arriba puede parecer más código para escribir. Pero cuando tenemos muchas clases con muchas dependencias, entonces realmente vale la pena. Y si tuviéramos que escribir esas mismas clases *sin* usar la inyección de dependencia, entonces esa misma aplicación con muchas clases sería difícil de mantener y probar.

Esto araña la superficie de *por* qué usamos contenedores de inyección de dependencia. *La forma en* que configuramos nuestra aplicación para usar una (y usarla correctamente) no es solo un tema, sino una serie de temas, ya que las instrucciones y los ejemplos varían de un contenedor a otro.

Lea Inyección de dependencia en línea: https://riptutorial.com/es/dot-net/topic/5085/inyeccion-de-dependencia

Capítulo 34: JSON en .NET con Newtonsoft.Json

Introducción

El paquete `Newtonsoft.Json` ha convertido en el estándar de facto para usar y manipular texto y objetos con formato JSON en .NET. Es una herramienta robusta, rápida y fácil de usar.

Examples

Serializar objeto en JSON

```
using Newtonsoft.Json;

var obj = new Person
{
    Name = "Joe Smith",
    Age = 21
};
var serializedJson = JsonConvert.SerializeObject(obj);
```

Esto da como resultado este JSON: `{"Name":"Joe Smith","Age":21}`

Deserializar un objeto desde texto JSON

```
var json = "{\"Name\":\"Joe Smith\",\"Age\":21}";
var person = JsonConvert.DeserializeObject<Person>(json);
```

Esto produce un objeto `Person` con el nombre "Joe Smith" y 21 años.

Lea JSON en .NET con Newtonsoft.Json en línea: https://riptutorial.com/es/dot-net/topic/8746/json-en--net-con-newtonsoft-json

Capítulo 35: Leer y escribir archivos zip

Introducción

La clase **ZipFile** vive en el espacio de nombres **System.IO.Compression** . Se puede utilizar para leer y escribir en archivos Zip.

Observaciones

- También puede utilizar un MemoryStream en lugar de un FileStream.
- Excepciones

Excepción	Condición
ArgumentException	La secuencia ya se ha cerrado, o las capacidades de la secuencia no coinciden con el modo (por ejemplo, tratando de escribir en una secuencia de solo lectura)
ArgumentNullException	el *flujo de* entrada es nulo
ArgumentOutOfRangeException	*el modo* tiene un valor inválido
InvalidDataException	Ver lista abajo

Cuando se lanza una **InvalidDataException** , puede tener 3 causas:

- El contenido del flujo no se puede interpretar como un archivo zip
- *el modo* es Actualizar y falta una entrada del archivo o está dañada y no se puede leer
- *el modo* es Actualizar y una entrada es demasiado grande para caber en la memoria

Toda la información ha sido tomada de esta página de MSDN.

Examples

Listado de contenidos ZIP

Este fragmento de código mostrará una lista de todos los nombres de archivos de un archivo zip. Los nombres de archivo son relativos a la raíz zip.

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    for (int i = 0; i < archive.Entries.Count; i++)
    {
        Console.WriteLine($"{i}: {archive.Entries[i]}");
```

```
    }
}
```

Extraer archivos de archivos ZIP

Extraer todos los archivos en un directorio es muy fácil:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    archive.ExtractToDirectory(AppDomain.CurrentDomain.BaseDirectory);
}
```

Cuando el archivo ya exista, se lanzará una **excepción System.IO.IOException** .

Extraer archivos específicos:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    // Get a root entry file
    archive.GetEntry("test.txt").ExtractToFile("test_extracted_getentries.txt", true);

    // Enter a path if you want to extract files from a subdirectory
    archive.GetEntry("sub/subtest.txt").ExtractToFile("test_sub.txt", true);

    // You can also use the Entries property to find files
    archive.Entries.FirstOrDefault(f => f.Name ==
"test.txt")?.ExtractToFile("test_extracted_linq.txt", true);

    // This will throw a System.ArgumentNullException because the file cannot be found
    archive.GetEntry("nonexistingfile.txt").ExtractToFile("fail.txt", true);
}
```

Cualquiera de estos métodos producirá el mismo resultado.

Actualizando un archivo ZIP

Para actualizar un archivo ZIP, el archivo debe abrirse con ZipArchiveMode.Update en su lugar.

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Update))
{
    // Add file to root
    archive.CreateEntryFromFile("test.txt", "test.txt");

    // Add file to subfolder
    archive.CreateEntryFromFile("test.txt", "symbols/test.txt");
}
```

También existe la opción de escribir directamente en un archivo dentro del archivo:

```
var entry = archive.CreateEntry("createentry.txt");
```

```
using(var writer = new StreamWriter(entry.Open()))
{
    writer.WriteLine("Test line");
}
```

Lea Leer y escribir archivos zip en línea: https://riptutorial.com/es/dot-net/topic/9943/leer-y-escribir-archivos-zip

Capítulo 36: LINQ

Introducción

LINQ (Language Integrated Query) es una expresión que recupera datos de un origen de datos. LINQ simplifica esta situación al ofrecer un modelo consistente para trabajar con datos a través de varios tipos de fuentes de datos y formatos. En una consulta LINQ, siempre está trabajando con objetos. Utiliza los mismos patrones de codificación básicos para consultar y transformar datos en documentos XML, bases de datos SQL, conjuntos de datos ADO.NET, colecciones .NET y cualquier otro formato para el que un proveedor esté disponible. LINQ se puede utilizar en C # y VB.

Sintaxis

- agregación estática pública de TSource <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, TSource, TSource> func)
- agregación estática pública TAccumulate <TSource, TAccumulate> (esta fuente IEnumerable <TSource>, TAccumulate seed, Func <TAccumulate, TSource, TAccumulate> func)
- agregación estática pública de TResult <TSource, TAccumulate, TResult> (esta fuente IEnumerable <TSource>, fuente de TAccumulate, Func <TAccumulate, TSource, TAccumulate> func, Func <TAccumulate, TResult> resultadoSelector)
- Public static Boolean All <TSource> (este IEnumerable <TSource> source, Func <TSource, Boolean> predicate)
- Public static Boolean Any <TSource> (esta fuente IEnumerable <TSource>)
- Public static Boolean Any <TSource> (este IEnumerable <TSource> source, Func <TSource, Boolean> predicate)
- IEnumerable estático público <TSource> AsEnumerable <TSource> (esta fuente <TSource> IEnumerable)
- Promedio Decimal estático público (esta fuente <Decimal> IEnumerable)
- Promedio doble estático público (esta fuente <Double> de IEnumerable)
- Promedio público doble estático (esta fuente IEnumerable <Int32>)
- Promedio público doble estático (esta fuente IEnumerable <Int64>)
- Promedio público nullable <Decimal> (este IEnumerable <Nullable <Decimal>> fuente)
- Nullable estático público <Double> Promedio (este IEnumerable <Nullable <Double>> fuente)
- Nullable estático público <Double> Promedio (este IEnumerable <Nullable <Int32>> fuente)
- Nullable estático público <Double> Promedio (este IEnumerable <Nullable <Int64>> fuente)
- Nullable estático público <Single> Promedio (este IEnumerable <Nullable <Single>> fuente)
- Promedio único estático público (esta fuente <Single> de IEnumerable)
- Promedio Decimal estático público <TSource> (este IEnumerable <TSource> source, Func <TSource, Decimal> selector)
- Promedio público doble estático <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Double> selector)

- Promedio público doble estático <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Int32> selector)
- Promedio público doble estático <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Int64> selector)
- Nullable estático público <Decimal> Promedio <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Nullable <Decimal>> selector)
- Nullable estático público <Double> Promedio <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Nullable <Double>> selector)
- Nullable estático público <Double> Promedio <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Nullable <Int32>> selector)
- Nullable estático público <Double> Promedio <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Nullable <Int64>> selector)
- Nullable estático público <Single> Promedio <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Nullable <Single>> selector)
- Promedio único público estático <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Single> selector)
- Public static IEnumerable <TResult> Cast <TResult> (esta fuente IEnumerable)
- pública <Entrada> ITenumo> Concat <Toma> (este <EnTienda <Entrada> IEnumerable <Testidad> en segundo lugar)
- public static Boolean Contiene <TSource> (esta fuente IEnumerable <TSource>, valor de TSource)
- public static Boolean Contiene <TSource> (este IEnumerable <TSource> fuente, TSource valor, IEqualityComparer <TSource> comparador)
- cuenta estática pública Int32 <TSource> (esta fuente IEnumerable <TSource>)
- cuenta estática pública Int32 Count <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Boolean> predicate)
- Public static IEnumerable <TSource> DefaultIfEmpty <TSource> (esta fuente IEnumerable <TSource>)
- public static IEnumerable <TSource> DefaultIfEmpty <TSource> (esta fuente IEnumerable <TSource>, TSource defaultValue)
- Public static IEnumerable <TSource> Distinct <TSource> (esta fuente IEnumerable <TSource>)
- IEnumerable estático público <TSource> Distintivo <TSource> (este IEnumerable <TSource> fuente, IEqualityComparer <TSource> comparador)
- TSA estática pública ElementAt <TSource> (esta IEnumerable <TSource> fuente, índice Int32)
- TSource estático público ElementAtOrDefault <TSource> (este IEnumerable <TSource> fuente, índice Int32)
- Public static IEnumerable <TResult> Empty <TResult> ()
- IEnumerable estático público <TSource> Excepto <TSource> (este IEnumerable <TSource> primero, IEnumerable <TSource> segundo)
- IEnumerable estático público <TSource> Excepto <TSource> (este IEnumerable <TSource> primero, IEnumerable <TSource> segundo, IEqualityComparer <TSource> comparador)
- fuente estática pública TSource First <TSource> (esta fuente IEnumerable <TSource>)
- Public static TSource First <TSource> (este IEnumerable <TSource> source, Func <TSource, Boolean> predicate)

- público estático TSource FirstOrDefault <TSource> (esta fuente IEnumerable <TSource>)
- public static TSource FirstOrDefault <TSource> (este IEnumerable <TSource> source, Func <TSource, Boolean> predicate)
- IEnumerable estático público <IGrouping <TKey, TSource >> GroupBy <TSource, TKey> (este IEnumerable <TSource> source, Func <TSource, TKey> keySelector)
- IEnumerable estático público <IGrouping <TKey, TSource >> GroupBy <TSource, TKey> (este IEnumerable <TSource> fuente, Func <TSource, TKey> keySelector, IEqualityComparer <TKey> comparador)
- público IEnumerable estático <IGrouping <TKey, TElement >> GroupBy <TSource, TKey, TElement> (esta fuente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- public static IEnumerable <IGrouping <TKey, TElement >> GroupBy <TSource, TKey, TElement> (esta fuente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> comparer)
- público Estática IEnumerable <TResult> GrupoBy <TSource, TKey, TResult> (esta fuente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TKey, IEnumerable <TSource>, TResult> resultSelector)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TResult> (esta fuente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TKey, IEnumerable <TSource>, TResult> resultSelector, IEqualityComparer <TKey> comparer)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TElement, TResult> (esta fuente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, Func <TKey, IEnumerable <TElement>, TResult > resultSelector)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TElement, TResult> (esta fuente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, Func <TKey, IEnumerable <TElement>, TResult > resultSelector, comparador IEqualityComparer <TKey>)
- public static IEnumerable <TResult> GroupJoin <TOuter, TInner, TKey, TResult> (este IEnumerable <TOuter> external, IEnumerable <TInner> inner, Func <Touter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <Touter, IEnumerable <TInner>, TResult> resultSelector)
- public static IEnumerable <TResult> GroupJoin <TOuter, TInner, TKey, TResult> (este IEnumerable <TOuter> external, IEnumerable <TInner> inner, Func <Touter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <Touter, IEnumerable <TInner>, TResult> resultSelector, IEqualityComparer <TKey> comparer)
- IEnumerable estático público <TSource> Intersect <TSource> (este IEnumerable <TSource> primero, IEnumerable <TSource> segundo)
- IEnumerable estático público <TSource> Intersect <TSource> (este IEnumerable <TSource> primero, IEnumerable <TSource> segundo, IEqualityComparer <TSource> comparador)
- public static IEnumerable <TResult> Join <TOuter, TInner, TKey, TResult> (este IEnumerable <TOuter> external, IEnumerable <TInner> inner, Func <TOuter, TKey> outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <Touter, TInner, TResult> resultSelector)
- public static IEnumerable <TResult> Join <TOuter, TInner, TKey, TResult> (este IEnumerable <TOuter> external, IEnumerable <TInner> inner, Func <TOuter, TKey>

outerKeySelector, Func <TInner, TKey> innerKeySelector, Func <Touter, TInner, TResult> resultSelector, IEqualityComparer <TKey> comparer)

- Último <TSource> de esta fuente estática pública (esta fuente <TSource> de IEnumerable)
- Último recurso público de TSource <TSource> (este IEnumerable <TSource> source, Func <TSource, Boolean> predicate)
- público estático TSource LastOrDefault <TSource> (esta fuente IEnumerable <TSource>)
- público estático TSource LastOrDefault <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Boolean> predicado)
- pública estática Int64 LongCount <TSource> (esta fuente IEnumerable <TSource>)
- pública estática Int64 LongCount <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Boolean> predicate)
- Máximo decimal estático público (esta fuente <Decimal> IEnumerable)
- doble estática pública (esta fuente <Double> de IEnumerable)
- pública estática Int32 Max (esta fuente IEnumerable <Int32>)
- pública estática Int64 Max (esta fuente IEnumerable <Int64>)
- Nullable estática pública <Decimal> Máx (este IEnumerable <Nullable <Decimal>> fuente)
- Nullable estático público <Double> Max (este IEnumerable <Nullable <Double>> fuente)
- Nullable estática pública <Int32> Máx (este IEnumerable <Nullable <Int32>> fuente)
- Nullable estática pública <Int64> Máx (este IEnumerable <Nullable <Int64>> fuente)
- Nullable estático público <Single> Máx (este IEnumerable <Nullable <Single>> fuente)
- único estático público (esta fuente <Single> de IEnumerable)
- public static TSource Max <TSource> (esta fuente IEnumerable <TSource>)
- public static Decimal Max <TSource> (este IEnumerable <TSource> source, Func <TSource, Decimal> selector)
- public static Double Max <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Double> selector)
- public static Int32 Max <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Int32> selector)
- pública estática Int64 Max <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Int64> selector)
- pública estática Nullable <Decimal> Max <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Nullable <Decimal>> selector)
- pública estática Nullable <Double> Max <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Nullable <Double>> selector)
- Public static Nullable <Int32> Max <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Nullable <Int32>> selector)
- Public static Nullable <Int64> Max <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Nullable <Int64>> selector)
- pública estática Nullable <Single> Max <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Nullable <Single>> selector)
- public static Single Max <TSource> (este IEnumerable <TSource> source, Func <TSource, Single> selector)
- public static TResult Max <TSource, TResult> (esta fuente IEnumerable <TSource>, Func <TSource, TResult> selector)
- Minimal Decimal estático público (esta fuente <Decimal> IEnumerable)
- Double Min estática pública (esta fuente <Double> de IEnumerable)

- Int32 Min público estático (esta fuente IEnumerable <Int32>)
- Int64 público estático (esta fuente IEnumerable <Int64>)
- Nullable estático público <Decimal> Min (este IEnumerable <Nullable <Decimal>> fuente)
- Nullable estático público <Double> Min (este IEnumerable <Nullable <Double>> fuente)
- Nullable estática pública <Int32> Min (este IEnumerable <Nullable <Int32>> fuente)
- Nullable estática pública <Int64> Min (este IEnumerable <Nullable <Int64>> fuente)
- Nullable estático público <Single> Min (este IEnumerable <Nullable <Single>> fuente)
- pública (solo) estática mínima (esta fuente <Single> de IEnumerable)
- <<source Source> (esta fuente IEnumerable <TSource>) estática pública
- Minimal Decimal estático público <TSource> (este IEnumerable <TSource> source, Func <TSource, Decimal> selector)
- pública estática Double Min <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Double> selector)
- <> Fuente pública estática Int32 Min <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Int32> selector)
- Int64 público estático Int64 Min <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Int64> selector)
- Nullable estática pública <Decimal> Min <TSource> (este <Entrada <EntradaDeIerable <Funcional, Func <TSource, <Decimal> Nullable>> selector))
- Public static Nullable <Double> Min <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Nullable <Double>> selector)
- Nullable estática pública <Int32> Min <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Nullable <Int32>> selector)
- Nullable estático público <Int64> Min <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Nullable <Int64>> selector)
- Nullable estática pública <Single> Min <TSource> (esta IEnumerable <TSource> fuente, Func <TSource, Nullable <Single>> selector)
- Public static Single Min <TSource> (este IEnumerable <TSource> source, Func <TSource, Single> selector)
- público estático TResult Min <TSource, TResult> (este IEnumerable <TSource> fuente, Func <TSource, TResult> selector)
- IEnumerable estático público <TResult> OfType <TResult> (esta fuente IEnumerable)
- Public static IOrderedEnumerable <TSource> OrderBy <TSource, TKey> (esta fuente IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- Public static IOrderedEnumerable <TSource> OrderBy <TSource, TKey> (esta fuente IEnumerable <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> comparador)
- Public static IOrderedEnumerable <TSource> OrderByDescending <TSource, TKey> (esta fuente IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- Public static IOrderedEnumerable <TSource> OrderByDescending <TSource, TKey> (esta fuente IEnumerable <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> comparador)
- Intervalo IEnumerable estático público <Int32> (Int32 start, Int32 count)
- Public static IEnumerable <TResult> Repetir <TResult> (elemento TResult, conteo Int32)
- Public static IEnumerable <TSource> Reverse <TSource> (esta fuente IEnumerable <TSource>)

- IEnumerable estático público <TResult> Seleccione <TSource, TResult> (este <Entrada> <En este fuente IEnumerable, Func <TSource, TResult> selector)
- IEnumerable estático público <TResult> Seleccione <TSource, TResult> (esta fuente <TSource> IEnumerable>, Func <TSource, Int32, TResult> selector)
- public static IEnumerable <TResult> SelectMany <TSource, TResult> (esta fuente IEnumerable <TSource>, Func <TSource, IEnumerable <TResult>> selector)
- public static IEnumerable <TResult> SelectMany <TSource, TResult> (esta fuente IEnumerable <TSource>, Func <TSource, Int32, IEnumerable <TResult>> selector)
- public static IEnumerable <TResult> SelectMany <TSource, TCollection, TResult> (esta fuente IEnumerable <TSource>, Func <TSource, IEnumerable <TCollection>> collectionSelector, Func <TSource, TCollection, TResult> resultadoSelector)
- public static IEnumerable <TResult> SelectMany <TSource, TCollection, TResult> (esta fuente IEnumerable <TSource>, Func <TSource, Int32, IEnumerable <TCollection>> collectionSelector, Func <TSource, TCollection, TResult> resultSelector)
- Public static Boolean SequenceEqual <TSource> (este IEnumerable <TSource> primero, IEnumerable <TSource> segundo)
- Public static Boolean SequenceEqual <TSource> (este IEnumerable <TSource> primero, IEnumerable <TSource> segundo, IEqualityComparer <TSource> comparer)
- Public static TSource Single <TSource> (esta fuente IEnumerable <TSource>)
- Fuente estática pública TSource Single <TSource> (esta fuente <Entrada>, <Entrada>, Func <TSource, Boolean> predicado)
- público estático TSource SingleOrDefault <TSource> (esta fuente IEnumerable <TSource>)
- público estático TSource SingleOrDefault <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Boolean> predicado)
- ITenumerable <TSource> estático público Omitir <TSource> (esta fuente <TSource> IEnumerable, cuenta Int32)
- público IEnumerable I <> <> <> Fuente> Omitir> <<<Fuente> (<En este <I> IEnumerable <TSource>, Func <TSource, Boolean> predicate)
- público IEnumerable I <> <> <> Fuente> Omitir> <<<Fuente> (<En este <I> <I> Fuente <En <> <> <<<<Source Source>, Func <TSource, Int32, Boolean> predicate)
- Suma decimal estática pública (esta fuente <Decimal> IEnumerable)
- Suma estática pública doble (esta fuente <Double> de IEnumerable)
- suma estática pública Int32 (esta fuente IEnumerable <Int32>)
- Suma estática pública Int64 (esta fuente IEnumerable <Int64>)
- Public static Nullable <Decimal> Sum (este IEnumerable <Nullable <Decimal>> source)
- Public static Nullable <Double> Sum (este IEnumerable <Nullable <Double>> source)
- Public static Nullable <Int32> Sum (este IEnumerable <Nullable <Int32>> source)
- Nullable estática pública <Int64> Suma (esta IEnumerable <Nullable <Int64>> fuente)
- Public static Nullable <Single> Sum (este IEnumerable <Nullable <Single>> fuente)
- Suma única estática pública (esta fuente IEnumerable <Single>)
- Suma Decimal estática pública <TSource> (esta fuente <Entrada> IEnumerable, Func <TSource, Decimal> selector)
- Public static Double Sum <TSource> (este IEnumerable <TSource> source, Func <TSource, Double> selector)
- Public static Int32 Sum <TSource> (este IEnumerable <TSource> source, Func <TSource, Int32> selector)

- Public static Int64 Sum <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Int64> selector)
- Public static Nullable <Decimal> Sum <TSource> (este IEnumerable <TSource> source, Func <TSource, Nullable <Decimal>> selector)
- Public static Nullable <Double> Sum <TSource> (este IEnumerable <TSource> source, Func <TSource, Nullable <Double>> selector)
- Public static Nullable <Int32> Sum <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Nullable <Int32>> selector)
- Public static Nullable <Int64> Sum <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Nullable <Int64>> selector)
- pública estática Nullable <Single> Sum <TSource> (esta fuente IEnumerable <TSource>, Func <TSource, Nullable <Single>> selector)
- Public static Single Sum <TSource> (este IEnumerable <TSource> source, Func <TSource, Single> selector)
- ITenumerable <TSource> estático público Tomar <TSource> (esta fuente <TSource> IEnumerable, cuenta de Int32)
- público IEnumerable <TSource> TakeWhile <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Boolean> predicado)
- público IEnumerable <TSource> TakeWhile <TSource> (este IEnumerable <TSource> fuente, Func <TSource, Int32, Boolean> predicado)
- Public static IOrderedEnumerable <TSource> ThenBy <TSource, TKey> (esta fuente IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector)
- Public static IOrderedEnumerable <TSource> ThenBy <TSource, TKey> (esta fuente IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> comparador)
- public static IOrderedEnumerable <TSource> ThenByDescending <TSource, TKey> (esta fuente IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> ThenByDescending <TSource, TKey> (esta fuente IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> comparer)
- TSource estático público [] ToArray <TSource> (esta fuente IEnumerable <TSource>)
- Diccionario estático público <TKey, TSource> ToDictionary <TSource, TKey> (este <Entrada <Entrada> fuente, Func <TSource, TKey> keySelector)
- Diccionario estático público <TKey, TSource> ToDictionary <TSource, TKey> (esta fuente IEnumerable <TSource>, Func <TSource, TKey> keySelector, IEqualityComparer <TKey> comparador)
- Diccionario estático público <TKey, TElement> ToDictionary <TSource, TKey, TElement> (esta fuente <EntradaTenerable ITenumerable, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- diccionario estático público <TKey, TElement> ToDictionary <TSource, TKey, TElement> (este <Entrada <EntradaTenera <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> comparador)
- Lista estática pública <TSource> ToList <TSource> (esta fuente IEnumerable <TSource>)
- pública ILookup estática <TKey, TSource> ToLookup <TSource, TKey> (esta fuente <Ted> IEnumerable <TSource, Func <TSource, TKey> keySelector)
- pública ILookup estática <TKey, TSource> ToLookup <TSource, TKey> (esta fuente

IEnumerable <TSource>, Func <TSource, TKey> keySelector, IEqualityComparer <TKey> comparador)
- pública ILookup estática <TKey, TElement> ToLookup <TSource, TKey, TElement> (esta fuente IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- pública estática ILookup <TKey, TElement> ToLookup <TSource, TKey, TElement> (esta fuente enumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> comparador)
- Unión <TSource> estática pública <TSource> <TSource> (primero <TSource> IEnumerable <TSource>, <TSource> IEnumerable <TSource)
- Public <B> <TSource> Union <TSource> estática pública (esta <TSource> IEnumerable <TSource> en primer lugar, <TSource> IEnumerable <TSource>, comparador IEqualityComparer <TSource>)
- ITenumerable <TSource> estático público Donde <TSource> (este <Tedource> IEnumerable fuente, Func <TSource, booleano> predicado)
- ITenumerable <TSource> estático público Donde <TSource> (este <Tedource> IEnumerable fuente, Func <TSource, Int32, booleano> predicado)
- público IEnumerable IEnumerable <TResult> Zip <TFirst, TSecond, TResult> (este IEnumerable <TFirst> primero, IEnumerable <TSecond> segundo, Func <TFirst, TSecond, TResult> resultSelector)

Observaciones

- Véase también LINQ .

Los métodos incorporados de LINQ son métodos de extensión para la `IEnumerable<T>` que viven en la clase `System.Linq.Enumerable` en el ensamblado `System.Core` . Están disponibles en .NET Framework 3.5 y versiones posteriores.

LINQ permite la modificación, transformación y combinación `IEnumerable` de varios `IEnumerable` mediante una sintaxis funcional o de tipo consulta.

Mientras que los métodos estándar de LINQ pueden trabajar en cualquier `IEnumerable<T>` , incluyendo las matrices simples y `List<T>` s, también pueden ser utilizados en los objetos de base de datos, donde el conjunto de expresiones LINQ se puede transformar en muchos casos a SQL si la objeto de datos lo soporta. Ver LINQ to SQL .

Para los métodos que comparan objetos (como `Contains` y `Except`), `IEquatable<T>.Equals` se usa si el tipo T de la colección implementa esa interfaz. De lo contrario, se utilizan los `Equals` y `GetHashCode` estándar del tipo (posiblemente anulados de las implementaciones de `Object` predeterminadas). También hay sobrecargas para estos métodos que permiten especificar un `IEqualityComparer<T>` .

Para los métodos `...OrDefault` , el `default(T)` se utiliza para generar valores predeterminados.

Referencia oficial: clase enumerable.

Evaluación perezosa

Prácticamente todas las consultas que devuelven un `IEnumerable<T>` no se evalúan de inmediato; en cambio, la lógica se retrasa hasta que la consulta se repite. Una implicación es que cada vez que alguien itera sobre un `IEnumerable<T>` creado a partir de una de estas consultas, por ejemplo, `.Where()` , se repite la lógica de consulta completa. Si el predicado es de larga ejecución, esto puede ser una causa de problemas de rendimiento.

Una solución simple (cuando sabe o puede controlar el tamaño aproximado de la secuencia resultante) es amortiguar completamente los resultados utilizando `.ToArray()` o `.ToList()` . `.ToDictionary()` o `.ToLookup()` pueden cumplir la misma función. Por supuesto, también se puede iterar en toda la secuencia y almacenar en búfer los elementos de acuerdo con otra lógica personalizada.

`ToArray()` o `ToList()` ?

Tanto `.ToArray()` como `.ToList()` recorren todos los elementos de una `IEnumerable<T>` y guardan los resultados en una colección almacenada en la memoria. Utilice las siguientes pautas para determinar cuál elegir:

- Algunas API pueden requerir una `T[]` o una `List<T>` .
- `.ToList()` normalmente se ejecuta más rápido y genera menos basura que `.ToArray()` , porque este último debe copiar todos los elementos en una nueva colección de tamaño fijo una vez más que la anterior, en casi todos los casos.
- Los elementos se pueden agregar o quitar de la `List<T>` devuelta por `.ToList()` , mientras que la `T[]` devuelta desde `.ToArray()` sigue siendo un tamaño fijo durante toda su vida útil. En otras palabras, la `List<T>` es mutable, y `T[]` es inmutable.
- La `T[]` devuelta desde `.ToArray()` usa menos memoria que la `List<T>` devuelta desde `.ToList()` , por lo tanto, si el resultado se va a almacenar por mucho tiempo, prefiera `.ToArray()` . `List<T>.TrimExcess()` llamadas `List<T>.TrimExcess()` haría que la diferencia de memoria fuera estrictamente académica, al costo de eliminar la ventaja de velocidad relativa de `.ToList()` .

Examples

Seleccione (mapa)

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var names = persons.Select(p => p.Name);
Console.WriteLine(string.Join(",", names.ToArray()));
```

```
//Foo,Bar,Fizz,Buzz
```

Este tipo de función se suele llamar `map` en los lenguajes de programación funcionales.

Donde (filtro)

Este método devuelve un IEnumerable con todos los elementos que cumplen con la expresión lambda

Ejemplo

```
var personNames = new[]
{
    "Foo", "Bar", "Fizz", "Buzz"
};

var namesStartingWithF = personNames.Where(p => p.StartsWith("F"));
Console.WriteLine(string.Join(",", namesStartingWithF));
```

Salida:

Foo, Fizz

Ver demostración

Orden por

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByName = persons.OrderBy(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Id).ToArray()));

//2,4,3,1
```

OrderByDescending

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByNameDescending = persons.OrderByDescending(p => p.Name);
```

```
Console.WriteLine(string.Join(",", personsSortedByNameDescending.Select(p =>
p.Id).ToArray()));

//1,3,4,2
```

Contiene

```
var numbers = new[] {1,2,3,4,5};
Console.WriteLine(numbers.Contains(3)); //True
Console.WriteLine(numbers.Contains(34)); //False
```

Excepto

```
var numbers = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbersBetweenSixAndFourteen = new[] { 6, 8, 10, 12 };

var result = numbers.Except(evenNumbersBetweenSixAndFourteen);

Console.WriteLine(string.Join(",", result));

//1, 2, 3, 4, 5, 7, 9
```

Intersecarse

```
var numbers1to10 = new[] {1,2,3,4,5,6,7,8,9,10};
var numbers5to15 = new[] {5,6,7,8,9,10,11,12,13,14,15};

var numbers5to10 = numbers1to10.Intersect(numbers5to15);

Console.WriteLine(string.Join(",", numbers5to10));

//5,6,7,8,9,10
```

Concat

```
var numbers1to5 = new[] {1, 2, 3, 4, 5};
var numbers4to8 = new[] {4, 5, 6, 7, 8};

var numbers1to8 = numbers1to5.Concat(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,4,5,6,7,8
```

Tenga en cuenta que los duplicados se mantienen en el resultado. Si esto no es deseable, usa `Union` lugar.

Primero (encontrar)

```
var numbers = new[] {1,2,3,4,5};
```

```
var firstNumber = numbers.First();
Console.WriteLine(firstNumber); //1

var firstEvenNumber = numbers.First(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2
```

La siguiente `InvalidOperationException` lanza `InvalidOperationException` con el mensaje "La secuencia no contiene ningún elemento coincidente":

```
var firstNegativeNumber = numbers.First(n => n < 0);
```

Soltero

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.Single();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.Single(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1
```

Lo siguiente lanza la `InvalidOperationException` ya que hay más de un elemento en la secuencia:

```
var theOnlyNumberInNumbers = numbers.Single();
var theOnlyNegativeNumber = numbers.Single(n => n < 0);
```

Último

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.Last();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.Last(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4
```

Lo siguiente lanza `InvalidOperationException` :

```
var lastNegativeNumber = numbers.Last(n => n < 0);
```

LastOrDefault

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.LastOrDefault();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.LastOrDefault(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4
```

```
var lastNegativeNumber = numbers.LastOrDefault(n => n < 0);
Console.WriteLine(lastNegativeNumber); //0

var words = new[] { "one", "two", "three", "four", "five" };

var lastWord = words.LastOrDefault();
Console.WriteLine(lastWord); // five

var lastLongWord = words.LastOrDefault(w => w.Length > 4);
Console.WriteLine(lastLongWord); // three

var lastMissingWord = words.LastOrDefault(w => w.Length > 5);
Console.WriteLine(lastMissingWord); // null
```

SingleOrDefault

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.SingleOrDefault();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.SingleOrDefault(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1

var theOnlyNegativeNumber = numbers.SingleOrDefault(n => n < 0);
Console.WriteLine(theOnlyNegativeNumber);  //0
```

Lo siguiente lanza `InvalidOperationException` :

```
var theOnlyNumberInNumbers = numbers.SingleOrDefault();
```

FirstOrDefault

```
var numbers = new[] {1,2,3,4,5};

var firstNumber = numbers.FirstOrDefault();
Console.WriteLine(firstNumber); //1

var firstEvenNumber = numbers.FirstOrDefault(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2

var firstNegativeNumber = numbers.FirstOrDefault(n => n < 0);
Console.WriteLine(firstNegativeNumber); //0

var words = new[] { "one", "two", "three", "four", "five" };

var firstWord = words.FirstOrDefault();
Console.WriteLine(firstWord); // one

var firstLongWord = words.FirstOrDefault(w => w.Length > 3);
Console.WriteLine(firstLongWord); // three

var firstMissingWord = words.FirstOrDefault(w => w.Length > 5);
Console.WriteLine(firstMissingWord); // null
```

Alguna

Devuelve `true` si la colección tiene algún elemento que cumpla con la condición en la expresión lambda:

```
var numbers = new[] {1,2,3,4,5};

var isNotEmpty = numbers.Any();
Console.WriteLine(isNotEmpty); //True

var anyNumberIsOne = numbers.Any(n => n == 1);
Console.WriteLine(anyNumberIsOne); //True

var anyNumberIsSix = numbers.Any(n => n == 6);
Console.WriteLine(anyNumberIsSix); //False

var anyNumberIsOdd = numbers.Any(n => (n & 1) == 1);
Console.WriteLine(anyNumberIsOdd); //True

var anyNumberIsNegative = numbers.Any(n => n < 0);
Console.WriteLine(anyNumberIsNegative); //False
```

Todos

```
var numbers = new[] {1,2,3,4,5};

var allNumbersAreOdd = numbers.All(n => (n & 1) == 1);
Console.WriteLine(allNumbersAreOdd); //False

var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

Tenga en cuenta que el método `All` funciona al verificar que el primer elemento se evalúe como `false` según el predicado. Por lo tanto, el método devolverá `true` para *cualquier* predicado en el caso de que el conjunto esté vacío:

```
var numbers = new int[0];
var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

SelectMany (mapa plano)

`Enumerable.Select` devuelve un elemento de salida para cada elemento de entrada. Mientras que `Enumerable.SelectMany` produce un número variable de elementos de salida para cada elemento de entrada. Esto significa que la secuencia de salida puede contener más o menos elementos de los que estaban en la secuencia de entrada.

`Lambda expressions` pasadas a `Enumerable.Select` deben devolver un solo elemento. Las expresiones Lambda pasadas a `Enumerable.SelectMany` deben producir una secuencia secundaria. Esta secuencia secundaria puede contener un número variable de elementos para cada elemento en la secuencia de entrada.

Ejemplo

```
class Invoice
{
    public int Id { get; set; }
}

class Customer
{
    public Invoice[] Invoices {get;set;}
}

var customers = new[] {
    new Customer {
        Invoices = new[] {
            new Invoice {Id=1},
            new Invoice {Id=2},
        }
    },
    new Customer {
        Invoices = new[] {
            new Invoice {Id=3},
            new Invoice {Id=4},
        }
    },
    new Customer {
        Invoices = new[] {
            new Invoice {Id=5},
            new Invoice {Id=6},
        }
    }
};

var allInvoicesFromAllCustomers = customers.SelectMany(c => c.Invoices);

Console.WriteLine(
    string.Join(",", allInvoicesFromAllCustomers.Select(i => i.Id).ToArray()));
```

Salida:

1,2,3,4,5,6

Ver demostración

`Enumerable.SelectMany` también se puede lograr con una consulta basada en la sintaxis usando dos cláusulas consecutivas `from` :

```
var allInvoicesFromAllCustomers
    = from customer in customers
      from invoice in customer.Invoices
      select invoice;
```

Suma

```
var numbers = new[] {1,2,3,4};
```

```
var sumOfAllNumbers = numbers.Sum();
Console.WriteLine(sumOfAllNumbers); //10

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var totalPopulation = cities.Sum(c => c.Population);
Console.WriteLine(totalPopulation); //7500
```

Omitir

Omitir enumerará los primeros N elementos sin devolverlos. Una vez que se alcanza el número de artículo N + 1, Skip comienza a devolver cada artículo enumerado:

```
var numbers = new[] {1,2,3,4,5};

var allNumbersExceptFirstTwo = numbers.Skip(2);
Console.WriteLine(string.Join(",", allNumbersExceptFirstTwo.ToArray()));

//3,4,5
```

Tomar

Este método toma los primeros `n` elementos de un enumerable.

```
var numbers = new[] {1,2,3,4,5};

var threeFirstNumbers = numbers.Take(3);
Console.WriteLine(string.Join(",", threeFirstNumbers.ToArray()));

//1,2,3
```

SecuenciaEqual

```
var numbers = new[] {1,2,3,4,5};
var sameNumbers = new[] {1,2,3,4,5};
var sameNumbersInDifferentOrder = new[] {5,1,4,2,3};

var equalIfSameOrder = numbers.SequenceEqual(sameNumbers);
Console.WriteLine(equalIfSameOrder); //True

var equalIfDifferentOrder = numbers.SequenceEqual(sameNumbersInDifferentOrder);
Console.WriteLine(equalIfDifferentOrder); //False
```

Marcha atrás

```
var numbers = new[] {1,2,3,4,5};
var reversed = numbers.Reverse();

Console.WriteLine(string.Join(",", reversed.ToArray()));
```

```
//5,4,3,2,1
```

De tipo

```
var mixed = new object[] {1,"Foo",2,"Bar",3,"Fizz",4,"Buzz"};
var numbers = mixed.OfType<int>();

Console.WriteLine(string.Join(",", numbers.ToArray()));

//1,2,3,4
```

Max

```
var numbers = new[] {1,2,3,4};

var maxNumber = numbers.Max();
Console.WriteLine(maxNumber); //4

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var maxPopulation = cities.Max(c => c.Population);
Console.WriteLine(maxPopulation); //4000
```

Min

```
var numbers = new[] {1,2,3,4};

var minNumber = numbers.Min();
Console.WriteLine(minNumber); //1

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var minPopulation = cities.Min(c => c.Population);
Console.WriteLine(minPopulation); //1000
```

Promedio

```
var numbers = new[] {1,2,3,4};

var averageNumber = numbers.Average();
Console.WriteLine(averageNumber);
// 2,5
```

Este método calcula el promedio de enumerables de números.

```
var cities = new[] {
    new {Population = 1000},
    new {Population = 2000},
    new {Population = 4000}
};

var averagePopulation = cities.Average(c => c.Population);
Console.WriteLine(averagePopulation);
// 2333,33
```

Este método calcula el promedio de enumerable usando la función delegada.

Cremallera

.NET 4.0

```
var tens = new[] {10,20,30,40,50};
var units = new[] {1,2,3,4,5};

var sums = tens.Zip(units, (first, second) => first + second);

Console.WriteLine(string.Join(",", sums));

//11,22,33,44,55
```

Distinto

```
var numbers = new[] {1, 1, 2, 2, 3, 3, 4, 4, 5, 5};
var distinctNumbers = numbers.Distinct();

Console.WriteLine(string.Join(",", distinctNumbers));

//1,2,3,4,5
```

Agrupar por

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.GroupBy(p => p.Job);

foreach(var theGroup in groupedByJob)
{
    Console.WriteLine(
        "{0} are {1}s",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz,Buzz are Developers
//Foo,Bar are Astronauts
```

Agrupe las facturas por país, generando un nuevo objeto con el número de registro, el total pagado y el promedio pagado

```
var a = db.Invoices.GroupBy(i => i.Country)
          .Select(g => new { Country = g.Key,
                             Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

Si queremos solo los totales, no hay grupo.

```
var a = db.Invoices.GroupBy(i => 1)
          .Select(g => new { Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

Si necesitamos varios recuentos.

```
var a = db.Invoices.GroupBy(g => 1)
          .Select(g => new { High = g.Count(i => i.Paid >= 1000),
                             Low = g.Count(i => i.Paid < 1000),
                             Sum = g.Sum(i => i.Paid) });
```

Al diccionario

Devuelve un nuevo diccionario de la fuente `IEnumerable` usando la función de selector de teclas provista para determinar las claves. Lanzará una `ArgumentException` si keySelector no es inyectivo (devuelve un valor único para cada miembro de la colección de origen). Hay sobrecargas que permiten a uno especificar el valor que se almacenará, así como la clave.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
};
```

Especificar solo una función de selector de clave creará un `Dictionary<TKey,TVal>` con `TKey` el tipo de retorno del selector de clave, `TVal` el tipo de objeto original y el objeto original como valor almacenado.

```
var personsById = persons.ToDictionary(p => p.Id);
// personsById is a Dictionary<int,object>

Console.WriteLine(personsById[1].Name); //Fizz
Console.WriteLine(personsById[2].Name); //Buzz
```

La especificación de una función de selector de valor también creará un `Dictionary<TKey,TVal>` con `TKey` aún el tipo de retorno del selector de teclas, pero `TVal` ahora es el tipo de retorno de la función del selector de valor, y el valor devuelto como el valor almacenado.

```
var namesById = persons.ToDictionary(p => p.Id, p => p.Name);
//namesById is a Dictionary<int,string>

Console.WriteLine(namesById[3]); //Foo
Console.WriteLine(namesById[4]); //Bar
```

Como se indicó anteriormente, las claves devueltas por el selector de claves deben ser únicas. Lo siguiente lanzará una excepción.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
    new { Name="Oops", Id=4}
};

var willThrowException = persons.ToDictionary(p => p.Id)
```

Si no se puede dar una clave única para la colección de origen, considere usar ToLookup en su lugar. En la superficie, ToLookup se comporta de manera similar a ToDictionary, sin embargo, en la búsqueda resultante, cada clave se empareja con una colección de valores con claves coincidentes.

Unión

```
var numbers1to5 = new[] {1,2,3,4,5};
var numbers4to8 = new[] {4,5,6,7,8};

var numbers1to8 = numbers1to5.Union(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,6,7,8
```

Tenga en cuenta que los duplicados se eliminan del resultado. Si esto no es deseable, use `Concat` en `Concat` lugar.

ToArray

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersArray = someNumbers.ToArray();

Console.WriteLine(someNumbersArray.GetType().Name);
//Int32[]
```

Listar

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersList = someNumbers.ToList();

Console.WriteLine(
    someNumbersList.GetType().Name + " - " +
    someNumbersList.GetType().GetGenericArguments()[0].Name);
//List`1 - Int32
```

Contar

```
IEnumerable<int> numbers = new[] {1,2,3,4,5,6,7,8,9,10};

var numbersCount = numbers.Count();
Console.WriteLine(numbersCount); //10

var evenNumbersCount = numbers.Count(n => (n & 1) == 0);
Console.WriteLine(evenNumbersCount); //5
```

Elemento

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAt(2);
Console.WriteLine(thirdName); //Fizz

//The following throws ArgumentOutOfRangeException

var minusOnethName = names.ElementAt(-1);
var fifthName = names.ElementAt(4);
```

ElementAtOrDefault

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAtOrDefault(2);
Console.WriteLine(thirdName); //Fizz

var minusOnethName = names.ElementAtOrDefault(-1);
Console.WriteLine(minusOnethName); //null

var fifthName = names.ElementAtOrDefault(4);
Console.WriteLine(fifthName); //null
```

SkipWhile

```
var numbers = new[] {2,4,6,8,1,3,5,7};

var oddNumbers = numbers.SkipWhile(n => (n & 1) == 0);
```

```
Console.WriteLine(string.Join(",", oddNumbers.ToArray()));

//1,3,5,7
```

TakeWhile

```
var numbers = new[] {2,4,6,1,3,5,7,8};

var evenNumbers = numbers.TakeWhile(n => (n & 1) == 0);

Console.WriteLine(string.Join(",", evenNumbers.ToArray()));

//2,4,6
```

DefaultIfEmpty

```
var numbers = new[] {2,4,6,8,1,3,5,7};

var numbersOrDefault = numbers.DefaultIfEmpty();
Console.WriteLine(numbers.SequenceEqual(numbersOrDefault)); //True

var noNumbers = new int[0];

var noNumbersOrDefault = noNumbers.DefaultIfEmpty();
Console.WriteLine(noNumbersOrDefault.Count()); //1
Console.WriteLine(noNumbersOrDefault.Single()); //0

var noNumbersOrExplicitDefault = noNumbers.DefaultIfEmpty(34);
Console.WriteLine(noNumbersOrExplicitDefault.Count()); //1
Console.WriteLine(noNumbersOrExplicitDefault.Single()); //34
```

Agregado (pliegue)

Generando un nuevo objeto en cada paso:

```
var elements = new[] {1,2,3,4,5};

var commaSeparatedElements = elements.Aggregate(
    seed: "",
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements);  //1,2,3,4,5,
```

Usando el mismo objeto en todos los pasos:

```
var commaSeparatedElements2 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"));

Console.WriteLine(commaSeparatedElements2.ToString());  //1,2,3,4,5,
```

Usando un selector de resultados:

```
var commaSeparatedElements3 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"),
    resultSelector: (seed) => seed.ToString());
Console.WriteLine(commaSeparatedElements3);  //1,2,3,4,5,
```

Si se omite una semilla, el primer elemento se convierte en la semilla:

```
var seedAndElements = elements.Select(n=>n.ToString());
var commaSeparatedElements4 = seedAndElements.Aggregate(
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements4);  //12,3,4,5,
```

Para buscar

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.ToLookup(p => p.Job);

foreach(var theGroup in groupedByJob)
{
    Console.WriteLine(
        "{0} are {1}s",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz,Buzz are Developers
//Foo,Bar are Astronauts
```

Unirse

```
class Developer
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
        Id = 1,
        Name = "Foobuzz"
    },
    new Developer {
```

```
        Id = 2,
        Name = "Barfizz"
    }
};

var projects = new[] {
    new Project {
        DeveloperId = 1,
        Name = "Hello World 3D"
    },
    new Project {
        DeveloperId = 1,
        Name = "Super Fizzbuzz Maker"
    },
    new Project {
        DeveloperId = 2,
        Name = "Citizen Kane - The action game"
    },
    new Project {
        DeveloperId = 2,
        Name = "Pro Pong 2016"
    }
};

var denormalized = developers.Join(
    inner: projects,
    outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, proj) => new {
            ProjectName = proj.Name,
            DeveloperName = dev.Name});

foreach(var item in denormalized)
{
    Console.WriteLine("{0} by {1}", item.ProjectName, item.DeveloperName);
}

//Hello World 3D by Foobuzz
//Super Fizzbuzz Maker by Foobuzz
//Citizen Kane - The action game by Barfizz
//Pro Pong 2016 by Barfizz
```

Grupo unirse a

```
class Developer
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
```

```
        Id = 1,
        Name = "Foobuzz"
    },
    new Developer {
        Id = 2,
        Name = "Barfizz"
    }
};

var projects = new[] {
    new Project {
        DeveloperId = 1,
        Name = "Hello World 3D"
    },
    new Project {
        DeveloperId = 1,
        Name = "Super Fizzbuzz Maker"
    },
    new Project {
        DeveloperId = 2,
        Name = "Citizen Kane - The action game"
    },
    new Project {
        DeveloperId = 2,
        Name = "Pro Pong 2016"
    }
};

var grouped = developers.GroupJoin(
    inner: projects,
    outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, projs) => new {
            DeveloperName = dev.Name,
            ProjectNames = projs.Select(p => p.Name).ToArray()});

foreach(var item in grouped)
{
    Console.WriteLine(
        "{0}'s projects: {1}",
        item.DeveloperName,
        string.Join(", ", item.ProjectNames));
}

//Foobuzz's projects: Hello World 3D, Super Fizzbuzz Maker
//Barfizz's projects: Citizen Kane - The action game, Pro Pong 2016
```

Emitir

`Cast` es diferente de los otros métodos de `Enumerable` en que es un método de extensión para `IEnumerable` , no para `IEnumerable<T>` . Por lo tanto, se puede utilizar para convertir instancias del primero en instancias del último.

Esto no se compila ya que `ArrayList` no implementa `IEnumerable<T>` :

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.First());
```

Esto funciona como se esperaba:

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.Cast<int>().First()); //1
```

`Cast` **no** realiza conversiones de conversión. Lo siguiente compila pero lanza la excepción `InvalidCastException` en tiempo de ejecución:

```
var numbers = new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Cast<decimal>().ToArray();
```

La forma correcta de realizar una conversión de conversión a una colección es la siguiente:

```
var numbers= new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Select(n => (decimal)n).ToArray();
```

Vacío

Para crear un IEnumerable vacío de int:

```
IEnumerable<int> emptyList = Enumerable.Empty<int>();
```

Este IEnumerable vacío se almacena en caché para cada tipo T, de modo que:

```
Enumerable.Empty<decimal>() == Enumerable.Empty<decimal>(); // This is True
Enumerable.Empty<int>() == Enumerable.Empty<decimal>();     // This is False
```

Entonces por

`ThenBy` solo se puede utilizar después de una cláusula `OrderBy` que permite realizar pedidos utilizando varios criterios

```
var persons = new[]
{
    new {Id = 1, Name = "Foo", Order = 1},
    new {Id = 1, Name = "FooTwo", Order = 2},
    new {Id = 2, Name = "Bar", Order = 2},
    new {Id = 2, Name = "BarTwo", Order = 1},
    new {Id = 3, Name = "Fizz", Order = 2},
    new {Id = 3, Name = "FizzTwo", Order = 1},
};

var personsSortedByName = persons.OrderBy(p => p.Id).ThenBy(p => p.Order);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Name)));
//This will display :
//Foo,FooTwo,BarTwo,Bar,FizzTwo,Fizz
```

Distancia

Los dos parámetros para `Range` son el *primer* número y el *conteo* de elementos a producir (no el último número).

```
// prints 1,2,3,4,5,6,7,8,9,10
Console.WriteLine(string.Join(",", Enumerable.Range(1, 10)));

// prints 10,11,12,13,14
Console.WriteLine(string.Join(",", Enumerable.Range(10, 5)));
```

Izquierda combinación externa

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void Main(string[] args)
{
    var magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    var terry = new Person { FirstName = "Terry", LastName = "Adams" };

    var barley = new Pet { Name = "Barley", Owner = terry };

    var people = new[] { magnus, terry };
    var pets = new[] { barley };

    var query =
        from person in people
        join pet in pets on person equals pet.Owner into gj
        from subpet in gj.DefaultIfEmpty()
        select new
        {
            person.FirstName,
            PetName = subpet?.Name ?? "-" // Use - if he has no pet
        };

    foreach (var p in query)
        Console.WriteLine($"{p.FirstName}: {p.PetName}");
}
```

Repetir

`Enumerable.Repeat` genera una secuencia de un valor repetido. En este ejemplo se genera "Hola" 4 veces.

```
var repeats = Enumerable.Repeat("Hello", 4);

foreach (var item in repeats)
{
```

```
    Console.WriteLine(item);
}

/* output:
    Hello
    Hello
    Hello
    Hello
*/
```

Lea LINQ en línea: https://riptutorial.com/es/dot-net/topic/34/linq

Capítulo 37: Los diccionarios

Examples

Enumerar un diccionario

Puedes enumerar a través de un diccionario de una de las siguientes tres maneras:

Usando pares de KeyValue

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(KeyValuePair<int, string> kvp in dict)
{
   Console.WriteLine("Key : " + kvp.Key.ToString() + ", Value : " + kvp.Value);
}
```

Usando las teclas

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(int key in dict.Keys)
{
    Console.WriteLine("Key : " + key.ToString() + ", Value : " + dict[key]);
}
```

Usando valores

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(string s in dict.Values)
{
    Console.WriteLine("Value : " + s);
}
```

Inicializando un diccionario con un inicializador de colección

```
// Translates to `dict.Add(1, "First")` etc.
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};

// Translates to `dict[1] = "First"` etc.
// Works in C# 6.0.
var dict = new Dictionary<int, string>()
{
    [1] = "First",
    [2] = "Second",
    [3] = "Third"
};
```

Agregando a un diccionario

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict.Add(1, "First");
dict.Add(2, "Second");

// To safely add items (check to ensure item does not already exist - would throw)
if(!dict.ContainsKey(3))
{
   dict.Add(3, "Third");
}
```

Alternativamente, se pueden agregar / configurar a través de un indexador. (Un indexador se parece internamente a una propiedad, tiene un get y un set, pero toma un parámetro de cualquier tipo que se especifica entre paréntesis):

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict[1] = "First";
dict[2] = "Second";
dict[3] = "Third";
```

A diferencia del método `Add` que lanza una excepción, si una clave ya está contenida en el diccionario, el indexador simplemente reemplaza el valor existente.

Para un diccionario seguro, use `ConcurrentDictionary<TKey, TValue>` :

```
var dict = new ConcurrentDictionary<int, string>();
dict.AddOrUpdate(1, "First", (oldKey, oldValue) => "First");
```

Obtener un valor de un diccionario

Dado este código de configuración:

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};
```

Es posible que desee leer el valor de la entrada con la clave 1. Si no existe una clave, obtener un valor arrojará la `KeyNotFoundException` , por lo que es posible que primero desee verificarlo con `ContainsKey` :

```
if (dict.ContainsKey(1))
    Console.WriteLine(dict[1]);
```

Esto tiene una desventaja: buscará en su diccionario dos veces (una para verificar la existencia y otra para leer el valor). Para un diccionario grande, esto puede afectar el rendimiento. Afortunadamente ambas operaciones se pueden realizar juntas:

```
string value;
if (dict.TryGetValue(1, out value))
    Console.WriteLine(value);
```

Hacer un diccionario Con llaves Case-Insensivitve.

```
var MyDict = new Dictionary<string,T>(StringComparison.InvariantCultureIgnoreCase)
```

Diccionario concurrente (desde .NET 4.0)

> Representa una colección segura de subprocesos de pares clave / valor a los que se puede acceder mediante varios subprocesos simultáneamente.

Creando una instancia

La creación de una instancia funciona de la misma manera que con el `Dictionary<TKey, TValue>` , por ejemplo:

```
var dict = new ConcurrentDictionary<int, string>();
```

Agregando o Actualizando

Es posible que se sorprenda de que no haya un método `Add` , pero en su lugar hay `AddOrUpdate` con 2 sobrecargas:

(1) `AddOrUpdate(TKey key, TValue, Func<TKey, TValue, TValue> addValue)` - *Agrega un par de clave / valor si la clave aún no existe, o actualiza un par de clave / valor usando la función especificada si la clave ya existe.*

(2) `AddOrUpdate(TKey key, Func<TKey, TValue> addValue, Func<TKey, TValue, TValue> updateValueFactory)` : *utiliza las funciones especificadas para agregar un par de clave / valor a la clave si no existe ya, o para actualice un par clave / valor si la clave ya existe.*

Agregar o actualizar un valor, sin importar cuál era el valor si ya estaba presente para la clave dada (1):

```
string addedValue = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => "First");
```

Agregando o actualizando un valor, pero ahora alterando el valor en la actualización, basado en el valor anterior (1):

```
string addedValue2 = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => $"{valueOld}
Updated");
```

Usando la sobrecarga (2) también podemos agregar un nuevo valor usando una fábrica:

```
string addedValue3 = dict.AddOrUpdate(1, (key) => key == 1 ? "First" : "Not First",
(updateKey, valueOld) => $"{valueOld} Updated");
```

Obteniendo valor

Obtener un valor es el mismo que con el `Dictionary<TKey,TValue>` :

```
string value = null;
bool success = dict.TryGetValue(1, out value);
```

Obtener o agregar un valor

Hay dos sobrecargas de métodos, que **obtendrán o agregarán** un valor de una manera segura para subprocesos.

Obtenga el valor con la clave 2, o agregue el valor "Segundo" si la clave no está presente:

```
string theValue = dict.GetOrAdd(2, "Second");
```

Usar una fábrica para agregar un valor, si el valor no está presente:

```
string theValue2 = dict.GetOrAdd(2, (key) => key == 2 ? "Second" : "Not Second." );
```

IEnumerable al diccionario (≥ .NET 3.5)

Cree un diccionario <TKey, TValue> desde un IEnumerable <T> :

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
public class Fruits
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

```
var fruits = new[]
{
    new Fruits { Id = 8 , Name = "Apple" },
    new Fruits { Id = 3 , Name = "Banana" },
    new Fruits { Id = 7 , Name = "Mango" },
};


// Dictionary<int, string>                  key      value
var dictionary = fruits.ToDictionary(x => x.Id, x => x.Name);
```

Eliminar de un diccionario

Dado este código de configuración:

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};
```

Utilice el método `Remove` para eliminar una clave y su valor asociado.

```
bool wasRemoved = dict.Remove(2);
```

La ejecución de este código elimina la clave `2` y su valor del diccionario. `Remove` devuelve un valor booleano que indica si la clave especificada se encontró y se eliminó del diccionario. Si la clave no existe en el diccionario, no se elimina nada del diccionario y se devuelve falso (no se lanza ninguna excepción).

Es **incorrecto** intentar y eliminar una clave configurando el valor de la clave en `null` .

```
dict[2] = null; // WRONG WAY TO REMOVE!
```

Esto no eliminará la clave. Simplemente reemplazará el valor anterior con un valor `null` .

Para eliminar todas las claves y valores de un diccionario, use el método `Clear` .

```
dict.Clear();
```

Después de ejecutar `Clear` el `Count` del diccionario será 0, pero la capacidad interna permanecerá sin cambios.

ContainsKey (TKey)

Para verificar si un `Dictionary` tiene una clave específica, puede llamar al método `ContainsKey(TKey)` y proporcionar la clave del tipo `TKey` . El método devuelve un valor `bool` cuando la clave existe en el diccionario. Para la muestra:

```
var dictionary = new Dictionary<string, Customer>()
{
   {"F1", new Customer() { FirstName = "Felipe", ... } },
   {"C2", new Customer() { FirstName = "Carl", ... } },
   {"J7", new Customer() { FirstName = "John", ... } },
   {"M5", new Customer() { FirstName = "Mary", ... } },
};
```

Y compruebe si existe un `C2` en el Diccionario:

```
if (dictionary.ContainsKey("C2"))
{
   // exists
}
```

El método ContainsKey está disponible en la versión genérica `Dictionary<TKey, TValue>` .

Diccionario a la lista

Creando una lista de KeyValuePair:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<KeyValuePair<int, int>> list = new List<KeyValuePair<int, int>>();
list.AddRange(dictionary);
```

Creando una lista de claves:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Keys);
```

Creando una lista de valores:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Values);
```

El diccionario simultáneo aumentado con Lazy'1 reduce el cómputo duplicado

Problema

ConcurrentDictionary brilla cuando se trata de devolver instantáneamente las claves existentes de la memoria caché, en su mayoría sin bloqueo, y compitiendo en un nivel granular. Pero, ¿qué sucede si la creación del objeto es realmente costosa y supera el costo del cambio de contexto y se producen algunos errores de caché?

Si se solicita la misma clave de varios subprocesos, uno de los objetos resultantes de las operaciones de colisión se agregará finalmente a la colección, y los otros se tirarán, desperdiciando el recurso de la CPU para crear el objeto y el recurso de memoria para almacenar el objeto temporalmente . También se podrían desperdiciar otros recursos. Esto es realmente malo.

Solución

Podemos combinar `ConcurrentDictionary<TKey, TValue>` con `Lazy<TValue>` . La idea es que el método ConcurrentDictionary GetOrAdd solo puede devolver el valor que realmente se agregó a la colección. Los objetos perezosos perdidos también se pueden desperdiciar en este caso, pero eso no es un gran problema, ya que el objeto perezoso en sí es relativamente barato. La

propiedad Value de la perezosa perdedora nunca se solicita, porque somos inteligentes para solicitar solo la propiedad Value de la que realmente se agregó a la colección, la que se devolvió del método GetOrAdd:

```
public static class ConcurrentDictionaryExtensions
{
    public static TValue GetOrCreateLazy<TKey, TValue>(
        this ConcurrentDictionary<TKey, Lazy<TValue>> d,
        TKey key,
        Func<TKey, TValue> factory)
    {
        return
            d.GetOrAdd(
                key,
                key1 =>
                    new Lazy<TValue>(() => factory(key1),
                    LazyThreadSafetyMode.ExecutionAndPublication)).Value;
    }
}
```

El almacenamiento en caché de objetos XmlSerializer puede ser particularmente costoso, y también hay mucha controversia en el inicio de la aplicación. Y hay más en esto: si esos son serializadores personalizados, también habrá una pérdida de memoria para el resto del ciclo de vida del proceso. El único beneficio del ConcurrentDictionary en este caso es que durante el resto del ciclo de vida del proceso no habrá bloqueos, pero el inicio de la aplicación y el uso de la memoria serán inaceptables. Este es un trabajo para nuestro ConcurrentDictionary, aumentado con Lazy:

```
private ConcurrentDictionary<Type, Lazy<XmlSerializer>> _serializers =
    new ConcurrentDictionary<Type, Lazy<XmlSerializer>>();

public XmlSerializer GetSerialier(Type t)
{
    return _serializers.GetOrCreateLazy(t, BuildSerializer);
}

private XmlSerializer BuildSerializer(Type t)
{
    throw new NotImplementedException("and this is a homework");
}
```

Lea Los diccionarios en línea: https://riptutorial.com/es/dot-net/topic/45/los-diccionarios

Capítulo 38: Marco de Extensibilidad Gestionado

Observaciones

Una de las grandes ventajas de MEF sobre otras tecnologías que admiten el patrón de inversión de control es que admite la resolución de dependencias que no se conocen en el momento del diseño, sin necesidad de mucha configuración (si la hay).

Todos los ejemplos requieren una referencia al ensamblado System.ComponentModel.Composition.

Además, todos los ejemplos (básicos) los utilizan como objetos de negocio de muestra:

```
using System.Collections.ObjectModel;

namespace Demo
{
    public sealed class User
    {
        public User(int id, string name)
        {
            this.Id = id;
            this.Name = name;
        }

        public int Id { get; }
        public string Name { get; }
        public override string ToString() => $"User[Id: {this.Id}, Name={this.Name}]";
    }

    public interface IUserProvider
    {
        ReadOnlyCollection<User> GetAllUsers();
    }
}
```

Examples

Exportando un Tipo (Básico)

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel.Composition;

namespace Demo
{
    [Export(typeof(IUserProvider))]
    public sealed class UserProvider : IUserProvider
    {
```

```
        public ReadOnlyCollection<User> GetAllUsers()
        {
            return new List<User>
            {
                new User(0, "admin"),
                new User(1, "Dennis"),
                new User(2, "Samantha"),
            }.AsReadOnly();
        }
    }
}
```

Esto podría definirse virtualmente en cualquier lugar; todo lo que importa es que la aplicación sepa dónde buscarla (a través de los ComposablePartCatalogs que crea).

Importando (Básico)

```
using System;
using System.ComponentModel.Composition;

namespace Demo
{
    public sealed class UserWriter
    {
        [Import(typeof(IUserProvider))]
        private IUserProvider userProvider;

        public void PrintAllUsers()
        {
            foreach (User user in this.userProvider.GetAllUsers())
            {
                Console.WriteLine(user);
            }
        }
    }
}
```

Este es un tipo que depende de un `IUserProvider` , que podría definirse en cualquier lugar. Al igual que en el ejemplo anterior, todo lo que importa es que la aplicación sabe dónde buscar la exportación correspondiente (a través de los ComposablePartCatalogs que crea).

Conectando (Básico)

Vea los otros ejemplos (básicos) de arriba.

```
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;

namespace Demo
{
    public static class Program
    {
        public static void Main()
        {
            using (var catalog = new ApplicationCatalog())
```

```
            using (var exportProvider = new CatalogExportProvider(catalog))
            using (var container = new CompositionContainer(exportProvider))
            {
                exportProvider.SourceProvider = container;

                UserWriter writer = new UserWriter();

                // at this point, writer's userProvider field is null
                container.ComposeParts(writer);

                // now, it should be non-null (or an exception will be thrown).
                writer.PrintAllUsers();
            }
        }
    }
}
```

Siempre que algo en la ruta de búsqueda del ensamblaje de la aplicación tenga `[Export(typeof(IUserProvider))]` , la importación correspondiente del `UserWriter` cumplirá y los usuarios se imprimirán.

Se pueden usar otros tipos de catálogos (por ejemplo, `DirectoryCatalog`) en lugar de (o además de) `ApplicationCatalog` , para buscar en otros lugares las exportaciones que satisfacen las importaciones.

Lea Marco de Extensibilidad Gestionado en línea: https://riptutorial.com/es/dot-net/topic/62/marco-de-extensibilidad-gestionado

Capítulo 39: Para cada

Observaciones

Utilizarlo en absoluto?

Podría argumentar que la intención de .NET framework es que las consultas no tengan ningún efecto secundario y que el método `ForEach` esté, por definición, causando un efecto secundario. Es posible que su código sea más fácil de mantener y más fácil de probar si, en su lugar, utiliza un simple `foreach` .

Examples

Llamar a un método en un objeto en una lista

```
public class Customer {
   public void SendEmail()
   {
       // Sending email code here
   }
}

List<Customer> customers = new List<Customer>();

customers.Add(new Customer());
customers.Add(new Customer());

customers.ForEach(c => c.SendEmail());
```

Método de extensión para IEnumerable

`ForEach()` se define en la clase `List<T>` , pero no en `IQueryable<T>` o `IEnumerable<T>` . Tienes dos opciones en esos casos:

ToList primero

Se evaluará la enumeración (o consulta), copiando los resultados en una nueva lista o llamando a la base de datos. El método se llama a continuación en cada elemento.

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ToList().ForEach(c => c.SendEmail());
```

Este método tiene una sobrecarga de uso de memoria evidente, ya que se crea una lista intermedia.

Método de extensión

Escribe un método de extensión:

```
public static void ForEach<T>(this IEnumerable<T> enumeration, Action<T> action)
{
    foreach(T item in enumeration)
    {
        action(item);
    }
}
```

Utilizar:

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ForEach(c => c.SendEmail());
```

Precaución: los métodos LINQ de Framework se han diseñado con la intención de ser *puros* , lo que significa que no producen efectos secundarios. El único propósito del método `ForEach` es producir efectos secundarios, y se desvía de los otros métodos en este aspecto. Puede considerar simplemente usar un bucle `foreach` simple en su lugar.

Lea Para cada en línea: https://riptutorial.com/es/dot-net/topic/2225/para-cada

Capítulo 40: Procesamiento paralelo utilizando .Net framework

Introducción

Este tema trata sobre la programación de varios núcleos utilizando la biblioteca paralela de tareas con .NET framework. La biblioteca paralela de tareas le permite escribir código legible por humanos y se ajusta con la cantidad de Cores disponibles. Por lo tanto, puede estar seguro de que su software se actualizará automáticamente con el entorno de actualización.

Examples

Extensiones paralelas

Se han introducido extensiones paralelas junto con la biblioteca paralela de tareas para lograr el paralelismo de datos. El paralelismo de datos se refiere a escenarios en los que la misma operación se realiza de forma simultánea (es decir, en paralelo) en elementos de una colección o matriz de origen. .NET proporciona nuevas construcciones para lograr el paralelismo de datos mediante el uso de construcciones Parallel.For y Parallel.Foreach.

```
//Sequential version

foreach (var item in sourcecollection){

Process(item);

}

// Parallel equivalent

Parallel.foreach(sourcecollection, item => Process(item));
```

La construcción Parallel.ForEach mencionada anteriormente utiliza los múltiples núcleos y, por lo tanto, mejora el rendimiento de la misma manera.

Lea Procesamiento paralelo utilizando .Net framework en línea: https://riptutorial.com/es/dot-net/topic/8085/procesamiento-paralelo-utilizando--net-framework

Capítulo 41: Proceso y ajuste de afinidad del hilo

Parámetros

Parámetro	Detalles
afinidad	entero que describe el conjunto de procesadores en los que el proceso puede ejecutarse. Por ejemplo, en un sistema de 8 procesadores, si desea que su proceso se ejecute solo en los procesadores 3 y 4, elija una afinidad como esta: 00001100 que es igual a 12

Observaciones

La afinidad del procesador de un subproceso es el conjunto de procesadores con los que tiene una relación. En otras palabras, aquellos que pueden ser programados para ejecutarse.

La afinidad del procesador representa cada procesador como un bit. El bit 0 representa el procesador uno, el bit 1 representa el procesador dos y así sucesivamente.

Examples

Obtener máscara de afinidad de proceso

```
    public static int GetProcessAffinityMask(string processName = null)
    {
        Process myProcess = GetProcessByName(ref processName);

        int processorAffinity = (int)myProcess.ProcessorAffinity;
        Console.WriteLine("Process {0} Affinity Mask is : {1}", processName,
FormatAffinity(processorAffinity));

        return processorAffinity;
    }

    public static Process GetProcessByName(ref string processName)
    {
        Process myProcess;
        if (string.IsNullOrEmpty(processName))
        {
            myProcess = Process.GetCurrentProcess();
            processName = myProcess.ProcessName;
        }
        else
        {
            Process[] processList = Process.GetProcessesByName(processName);
            myProcess = processList[0];
        }
```

```
        return myProcess;
    }

    private static string FormatAffinity(int affinity)
    {
        return Convert.ToString(affinity, 2).PadLeft(Environment.ProcessorCount, '0');
    }
}
```

Ejemplo de uso:

```
private static void Main(string[] args)
{
    GetProcessAffinityMask();

    Console.ReadKey();
}
// Output:
// Process Test.vshost Affinity Mask is : 11111111
```

Establecer máscara de afinidad de proceso

```
    public static void SetProcessAffinityMask(int affinity, string processName = null)
    {
        Process myProcess = GetProcessByName(ref processName);

        Console.WriteLine("Process {0} Old Affinity Mask is : {1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));

        myProcess.ProcessorAffinity = new IntPtr(affinity);
        Console.WriteLine("Process {0} New Affinity Mask is : {1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));
    }
```

Ejemplo de uso:

```
private static void Main(string[] args)
{
    int newAffinity = Convert.ToInt32("10101010", 2);
    SetProcessAffinityMask(newAffinity);

    Console.ReadKey();
}
// Output :
// Process Test.vshost Old Affinity Mask is : 11111111
// Process Test.vshost New Affinity Mask is : 10101010
```

Lea Proceso y ajuste de afinidad del hilo en línea: https://riptutorial.com/es/dot-net/topic/4431/proceso-y-ajuste-de-afinidad-del-hilo

Capítulo 42: Puertos seriales

Examples

Operación básica

```
var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
serialPort.Open();
serialPort.WriteLine("Test data");
string response = serialPort.ReadLine();
Console.WriteLine(response);
serialPort.Close();
```

Lista de nombres de puertos disponibles

```
string[] portNames = SerialPort.GetPortNames();
```

Lectura asíncrona

```
void SetupAsyncRead(SerialPort serialPort)
{
    serialPort.DataReceived += (sender, e) => {
        byte[] buffer = new byte[4096];
        switch (e.EventType)
        {
            case SerialData.Chars:
                var port = (SerialPort)sender;
                int bytesToRead = port.BytesToRead;
                if (bytesToRead > buffer.Length)
                    Array.Resize(ref buffer, bytesToRead);
                int bytesRead = port.Read(buffer, 0, bytesToRead);
                // Process the read buffer here
                // ...
                break;
            case SerialData.Eof:
                // Terminate the service here
                // ...
                break;
        }
    };
```

Servicio de eco de texto síncrono.

```
using System.IO.Ports;

namespace TextEchoService
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
            serialPort.Open();
            string message = "";
            while (message != "quit")
            {
                message = serialPort.ReadLine();
                serialPort.WriteLine(message);
            }
            serialPort.Close();
        }
    }
}
```

Receptor asíncrono de mensajes.

```
using System;
using System.Collections.Generic;
using System.IO.Ports;
using System.Text;
using System.Threading;

namespace AsyncReceiver
{
    class Program
    {
        const byte STX = 0x02;
        const byte ETX = 0x03;
        const byte ACK = 0x06;
        const byte NAK = 0x15;
        static ManualResetEvent terminateService = new ManualResetEvent(false);
        static readonly object eventLock = new object();
        static List<byte> unprocessedBuffer = null;

        static void Main(string[] args)
        {
            try
            {
                var serialPort = new SerialPort("COM11", 9600, Parity.Even, 8, StopBits.One);
                serialPort.DataReceived += DataReceivedHandler;
                serialPort.ErrorReceived += ErrorReceivedHandler;
                serialPort.Open();
                terminateService.WaitOne();
                serialPort.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception occurred: {0}", e.Message);
            }
            Console.ReadKey();
        }

        static void DataReceivedHandler(object sender, SerialDataReceivedEventArgs e)
        {
            lock (eventLock)
            {
                byte[] buffer = new byte[4096];
                switch (e.EventType)
                {
                    case SerialData.Chars:
```

```
                    var port = (SerialPort)sender;
                    int bytesToRead = port.BytesToRead;
                    if (bytesToRead > buffer.Length)
                        Array.Resize(ref buffer, bytesToRead);
                    int bytesRead = port.Read(buffer, 0, bytesToRead);
                    ProcessBuffer(buffer, bytesRead);
                    break;
                case SerialData.Eof:
                    terminateService.Set();
                    break;
            }
        }
    }
    static void ErrorReceivedHandler(object sender, SerialErrorReceivedEventArgs e)
    {
        lock (eventLock)
            if (e.EventType == SerialError.TXFull)
            {
                Console.WriteLine("Error: TXFull. Can't handle this!");
                terminateService.Set();
            }
            else
            {
                Console.WriteLine("Error: {0}. Resetting everything", e.EventType);
                var port = (SerialPort)sender;
                port.DiscardInBuffer();
                port.DiscardOutBuffer();
                unprocessedBuffer = null;
                port.Write(new byte[] { NAK }, 0, 1);
            }
    }

    static void ProcessBuffer(byte[] buffer, int length)
    {
        List<byte> message = unprocessedBuffer;
        for (int i = 0; i < length; i++)
            if (buffer[i] == ETX)
            {
                if (message != null)
                {
                    Console.WriteLine("MessageReceived: {0}",
                        Encoding.ASCII.GetString(message.ToArray()));
                    message = null;
                }
            }
            else if (buffer[i] == STX)
                message = null;
            else if (message != null)
                message.Add(buffer[i]);
        unprocessedBuffer = message;
    }
}
}
```

Este programa espera los mensajes incluidos en los bytes `STX` y `ETX` y genera el texto que viene entre ellos. Todo lo demás se desecha. En el desbordamiento del búfer de escritura se detiene. En otros errores, restablece los buffers de entrada y salida y espera otros mensajes.

El código ilustra:

- Lectura asíncrona de puerto serie (ver `SerialPort.DataReceived` use).
- Procesamiento de errores del puerto serie (consulte `SerialPort.ErrorReceived` use).
- Implementación de protocolo no basado en mensajes de texto.
- Lectura parcial de mensajes.
    - El evento `SerialPort.DataReceived` puede suceder antes de que llegue el mensaje completo (hasta `ETX`). Es posible que el mensaje completo tampoco esté disponible en el búfer de entrada (SerialPort.Read (..., ..., port.BytesToRead) lee solo una parte del mensaje). En este caso, guardamos la parte recibida (`unprocessedBuffer`) y seguimos esperando otros datos.
- Tratar con varios mensajes que vienen de una sola vez.
    - El evento `SerialPort.DataReceived` puede suceder solo después de que el otro extremo haya enviado varios mensajes.

Lea Puertos seriales en línea: https://riptutorial.com/es/dot-net/topic/5366/puertos-seriales

Capítulo 43: ReadOnlyCollections

Observaciones

Una `ReadOnlyCollection` proporciona una vista de solo lectura a una colección existente (la 'colección de origen').

Los elementos no se agregan o eliminan directamente de `ReadOnlyCollection` . En su lugar, se agregan y eliminan de la colección de origen y `ReadOnlyCollection` reflejará estos cambios en el origen.

El número y el orden de los elementos dentro de una `ReadOnlyCollection` no pueden modificarse, pero las propiedades de los elementos pueden ser y los métodos pueden llamarse, asumiendo que están dentro del alcance.

Use `ReadOnlyCollection` cuando desee permitir que un código externo vea su colección sin poder modificarla, pero aún así podrá modificar la colección usted mismo.

Ver también

- `ObservableCollection<T>`
- `ReadOnlyObservableCollection<T>`

ReadOnlyCollections vs ImmutableCollection

Una `ReadOnlyCollection` difiere de una `ImmutableCollection` en que no puede editar una `ImmutableCollection` una vez que la creó; siempre contendrá `n` elementos, y no se pueden reemplazar ni reordenar. Una `ReadOnlyCollection` , por otro lado, no se puede editar directamente, pero los elementos todavía se pueden agregar / eliminar / reordenar usando la colección de origen.

Examples

Creando una colección ReadOnly

Usando el constructor

Una `ReadOnlyCollection` se crea al pasar un objeto `IList` existente al constructor:

```
var groceryList = new List<string> { "Apple", "Banana" };
var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);
```

Usando LINQ

Adicionalmente, LINQ proporciona un método de extensión `AsReadOnly()` para objetos `IList` :

```
var readOnlyVersion = groceryList.AsReadOnly();
```

Nota

Normalmente, desea mantener la colección de origen de forma privada y permitir el acceso público a `ReadOnlyCollection` . Si bien puede crear una `ReadOnlyCollection` de una lista en línea, no podrá modificar la colección después de crearla.

```
var readOnlyGroceryList = new List<string> {"Apple", "Banana"}.AsReadOnly();
// Great, but you will not be able to update the grocery list because
//  you do not have a reference to the source list anymore!
```

Si se encuentra haciendo esto, puede considerar usar otra estructura de datos, como `ImmutableCollection` .

Actualizando una ReadOnlyCollection

Una `ReadOnlyCollection` no se puede editar directamente. En su lugar, la colección de origen se actualiza y `ReadOnlyCollection` reflejará estos cambios. Esta es la característica clave de `ReadOnlyCollection` .

```
var groceryList = new List<string> { "Apple", "Banana" };

var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);

var itemCount = readOnlyGroceryList.Count;  // There are currently 2 items

//readOnlyGroceryList.Add("Candy");         // Compiler Error - Items cannot be added to a
ReadOnlyCollection object
groceryList.Add("Vitamins");                // ..but they can be added to the original
collection

itemCount = readOnlyGroceryList.Count;      // Now there are 3 items
var lastItem = readOnlyGroceryList.Last();  // The last item on the read only list is now
"Vitamins"
```

Ver demostración

Advertencia: los elementos de ReadOnlyCollection no son inherentemente de solo lectura

Si la colección de origen es de un tipo que no es inmutable, los elementos a los que se accede a través de `ReadOnlyCollection` se pueden modificar.

```
public class Item
{
    public string Name { get; set; }
    public decimal Price { get; set; }
```

```
}

public static void FillOrder()
{
    // An order is generated
    var order = new List<Item>
    {
        new Item { Name = "Apple", Price = 0.50m },
        new Item { Name = "Banana", Price = 0.75m },
        new Item { Name = "Vitamins", Price = 5.50m }
    };

    // The current sub total is $6.75
    var subTotal = order.Sum(item => item.Price);

    // Let the customer preview their order
    var customerPreview = new ReadOnlyCollection<Item>(order);

    // The customer can't add or remove items, but they can change
    //   the price of an item, even though it is a ReadOnlyCollection
    customerPreview.Last().Price = 0.25m;

    // The sub total is now only $1.50!
    subTotal = order.Sum(item => item.Price);
}
```

Ver demostración

Lea ReadOnlyCollections en línea: https://riptutorial.com/es/dot-net/topic/6906/readonlycollections

Capítulo 44: Recolección de basura

Introducción

En .Net, los objetos creados con new () se asignan en el montón administrado. Estos objetos nunca son finalizados explícitamente por el programa que los usa; en su lugar, este proceso es controlado por el .Net Garbage Collector.

Algunos de los ejemplos a continuación son "casos de laboratorio" para mostrar el recolector de basura en el trabajo y algunos detalles significativos de su comportamiento, mientras que otros se centran en cómo preparar clases para el manejo adecuado por parte del recolector de basura.

Observaciones

El recolector de basura tiene como objetivo reducir el costo del programa en términos de memoria asignada, pero hacerlo tiene un costo en términos de tiempo de procesamiento. Para lograr un buen compromiso general, hay una serie de optimizaciones que deben tenerse en cuenta al programar con el recolector de basura en mente:

- Si se debe invocar explícitamente el método Collect () (que no debería ser el caso), considere usar el modo "optimizado" que finaliza el objeto muerto solo cuando la memoria es realmente necesaria
- En lugar de invocar el método Collect (), considere usar los métodos AddMemoryPressure () y RemoveMemoryPressure (), que activan una colección de memoria solo si es realmente necesario
- No se garantiza que una colección de memoria finalice todos los objetos muertos; en cambio, el recolector de basura gestiona 3 "generaciones", un objeto que a veces "sobrevive" de una generación a la siguiente
- Es posible que se apliquen varios modelos de subprocesos, dependiendo de varios factores, incluida la configuración del ajuste fino, que resulta en diferentes grados de interferencia entre el subproceso del Recolector de basura y los otros subprocesos de la aplicación

Examples

Un ejemplo básico de recolección (basura)

Dada la siguiente clase:

```
public class FinalizableObject
{
    public FinalizableObject()
    {
        Console.WriteLine("Instance initialized");
    }

    ~FinalizableObject()
```

```
    {
        Console.WriteLine("Instance finalized");
    }
}
```

Un programa que crea una instancia, incluso sin usarlo:

```
new FinalizableObject(); // Object instantiated, ready to be used
```

Produce la siguiente salida:

```
<namespace>.FinalizableObject initialized
```

Si no ocurre nada más, el objeto no se finaliza hasta que el programa finaliza (lo que libera todos los objetos en el montón administrado, finalizando estos en el proceso).

Es posible forzar que el recolector de basura se ejecute en un punto determinado, de la siguiente manera:

```
new FinalizableObject(); // Object instantiated, ready to be used
GC.Collect();
```

Lo que produce el siguiente resultado:

```
<namespace>.FinalizableObject initialized
<namespace>.FinalizableObject finalized
```

Esta vez, tan pronto como se invocó el recolector de basura, el objeto no utilizado (también conocido como "muerto") se finalizó y se liberó del montón administrado.

Objetos vivos y objetos muertos - lo básico

Regla de oro: cuando se produce la recolección de basura, los "objetos vivos" son aquellos que aún están en uso, mientras que los "objetos muertos" son aquellos que ya no se usan (cualquier variable o campo que haga referencia a ellos, si los hubiera, ha quedado fuera del alcance antes de que ocurra la recolección) .

En el siguiente ejemplo (por conveniencia, FinalizableObject1 y FinalizableObject2 son subclases de FinalizableObject del ejemplo anterior y, por lo tanto, heredan el comportamiento del mensaje de inicialización / finalización):

```
var obj1 = new FinalizableObject1(); // Finalizable1 instance allocated here
var obj2 = new FinalizableObject2(); // Finalizable2 instance allocated here
obj1 = null; // No more references to the Finalizable1 instance
GC.Collect();
```

La salida será:

```
<namespace>.FinalizableObject1 initialized
```

```
<namespace>.FinalizableObject2 initialized
<namespace>.FinalizableObject1 finalized
```

En el momento en que se invoca el recolector de basura, FinalizableObject1 es un objeto muerto y se finaliza, mientras que FinalizableObject2 es un objeto vivo y se mantiene en el montón administrado.

Múltiples objetos muertos

¿Qué sucede si dos (o varios) objetos muertos se hacen referencia entre sí? Esto se muestra en el ejemplo a continuación, suponiendo que OtherObject es una propiedad pública de FinalizableObject:

```
var obj1 = new FinalizableObject1();
var obj2 = new FinalizableObject2();
obj1.OtherObject = obj2;
obj2.OtherObject = obj1;
obj1 = null; // Program no longer references Finalizable1 instance
obj2 = null; // Program no longer references Finalizable2 instance
// But the two objects still reference each other
GC.Collect();
```

Esto produce el siguiente resultado:

```
<namespace>.FinalizedObject1 initialized
<namespace>.FinalizedObject2 initialized
<namespace>.FinalizedObject1 finalized
<namespace>.FinalizedObject2 finalized
```

Los dos objetos se finalizan y se liberan del montón administrado a pesar de que se hacen referencia entre sí (porque no existe ninguna otra referencia para ninguno de ellos desde un objeto realmente vivo).

Referencias débiles

Las referencias débiles son ... referencias, a otros objetos (también conocidos como "objetivos"), pero "débiles", ya que no evitan que esos objetos se recojan de basura. En otras palabras, las referencias débiles no cuentan cuando el recolector de basura evalúa los objetos como "vivos" o "muertos".

El siguiente código:

```
var weak = new WeakReference<FinalizableObject>(new FinalizableObject());
GC.Collect();
```

Produce la salida:

```
<namespace>.FinalizableObject initialized
<namespace>.FinalizableObject finalized
```

El objeto se libera del montón administrado a pesar de que la variable WeakReference hace referencia a él (aún en el ámbito cuando se invocó el recolector de basura).

Consecuencia n. ° 1: en cualquier momento, no es seguro asumir si un destino WeakReference todavía está asignado en el montón administrado o no.

Consecuencia # 2: siempre que un programa necesite acceder al objetivo de una Referencia débil, se debe proporcionar un código para ambos casos, ya sea que el objetivo aún esté asignado o no. El método para acceder al objetivo es TryGetTarget:

```
var target = new object(); // Any object will do as target
var weak = new WeakReference<object>(target); // Create weak reference
target = null; // Drop strong reference to the target

// ... Many things may happen in-between

// Check whether the target is still available
if(weak.TryGetTarget(out target))
{
    // Use re-initialized target variable
    // To do whatever the target is needed for
}
else
{
    // Do something when there is no more target object
    // The target variable value should not be used here
}
```

La versión genérica de WeakReference está disponible desde .Net 4.5. Todas las versiones de framework proporcionan una versión no genérica, sin tipo, que se construye de la misma manera y se verifica de la siguiente manera:

```
var target = new object(); // Any object will do as target
var weak = new WeakReference(target); // Create weak reference
target = null; // Drop strong reference to the target

// ... Many things may happen in-between

// Check whether the target is still available
if (weak.IsAlive)
{
    target = weak.Target;

    // Use re-initialized target variable
    // To do whatever the target is needed for
}
else
{
    // Do something when there is no more target object
    // The target variable value should not be used here
}
```

Eliminar () vs. finalizadores

Implemente el método Dispose () (y declare la clase contenedora como IDisponible) como un

medio para garantizar que los recursos pesados en memoria se liberen tan pronto como el objeto ya no se use. La "captura" es que no existe una garantía sólida de que se invoque el método Dispose () (a diferencia de los finalizadores que siempre se invocan al final de la vida útil del objeto).

Un escenario es un programa que llama a Dispose () en objetos que crea explícitamente:

```
private void SomeFunction()
{
    // Initialize an object that uses heavy external resources
    var disposableObject = new ClassThatImplementsIDisposable();

    // ... Use that object

    // Dispose as soon as no longer used
    disposableObject.Dispose();

    // ... Do other stuff

    // The disposableObject variable gets out of scope here
    // The object will be finalized later on (no guarantee when)
    // But it no longer holds to the heavy external resource after it was disposed
}
```

Otro escenario es declarar una clase para ser instanciada por el marco. En este caso, la nueva clase generalmente hereda una clase base, por ejemplo en MVC uno crea una clase de controlador como una subclase de System.Web.Mvc.ControllerBase. Cuando la clase base implementa una interfaz IDisposable, este es un buen indicio de que el marco invocaría a Dispose () correctamente, pero nuevamente no hay una garantía sólida.

Por lo tanto, Dispose () no es un sustituto para un finalizador; en cambio, los dos deben ser usados para diferentes propósitos:

- Un finalizador eventualmente libera recursos para evitar pérdidas de memoria que de lo contrario ocurrirían
- Dispose () libera recursos (posiblemente los mismos) tan pronto como ya no sean necesarios, para aliviar la presión sobre la asignación de memoria general.

La correcta disposición y finalización de los objetos.

Como Dispose () y los finalizadores tienen objetivos diferentes, una clase que administre recursos pesados de memoria externa debería implementar ambos. La consecuencia es escribir la clase para que maneje bien dos posibles escenarios:

- Cuando solo se invoca el finalizador
- Cuando se invoca primero Dispose () y luego se invoca también el finalizador

Una solución es escribir el código de limpieza de tal manera que ejecutarlo una o dos veces produzca el mismo resultado que ejecutarlo solo una vez. La viabilidad depende de la naturaleza de la limpieza, por ejemplo:

- El cierre de una conexión de base de datos ya cerrada probablemente no tendría ningún

efecto por lo que funciona
- Actualizar algún "conteo de uso" es peligroso y produciría un resultado incorrecto cuando se lo llama dos veces en lugar de una vez.

Una solución más segura es garantizar por diseño que el código de limpieza se llame una vez y solo una vez, independientemente del contexto externo. Esto se puede lograr de la "manera clásica" usando una bandera dedicada:

```
public class DisposableFinalizable1: IDisposable
{
    private bool disposed = false;

    ~DisposableFinalizable1() { Cleanup(); }

    public void Dispose() { Cleanup(); }

    private void Cleanup()
    {
        if(!disposed)
        {
            // Actual code to release resources gets here, then
            disposed = true;
        }
    }
}
```

Alternativamente, el recolector de basura proporciona un método específico SuppressFinalize () que permite omitir el finalizador después de invocar Dispose:

```
public class DisposableFinalizable2 : IDisposable
{
    ~DisposableFinalizable2() { Cleanup(); }

    public void Dispose()
    {
        Cleanup();
        GC.SuppressFinalize(this);
    }

    private void Cleanup()
    {
        // Actual code to release resources gets here
    }
}
```

Lea Recolección de basura en línea: https://riptutorial.com/es/dot-net/topic/9636/recoleccion-de-basura

Capítulo 45: Redes

Observaciones

Ver también: Clientes HTTP

Examples

Chat TCP básico (TcpListener, TcpClient, NetworkStream)

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpChat
{
    static void Main(string[] args)
    {
        if(args.Length == 0)
        {
            Console.WriteLine("Basic TCP chat");
            Console.WriteLine();
            Console.WriteLine("Usage:");
            Console.WriteLine("tcpchat server <port>");
            Console.WriteLine("tcpchat client <url> <port>");
            return;
        }

        try
        {
            Run(args);
        }
        catch(IOException)
        {
            Console.WriteLine("--- Connection lost");
        }
        catch(SocketException ex)
        {
            Console.WriteLine("--- Can't connect: " + ex.Message);
        }
    }

    static void Run(string[] args)
    {
        TcpClient client;
        NetworkStream stream;
        byte[] buffer = new byte[256];
        var encoding = Encoding.ASCII;

        if(args[0].StartsWith("s", StringComparison.InvariantCultureIgnoreCase))
        {
            var port = int.Parse(args[1]);
            var listener = new TcpListener(IPAddress.Any, port);
```

```
            listener.Start();
            Console.WriteLine("--- Waiting for a connection...");
            client = listener.AcceptTcpClient();
        }
        else
        {
            var hostName = args[1];
            var port = int.Parse(args[2]);
            client = new TcpClient();
            client.Connect(hostName, port);
        }

        stream = client.GetStream();
        Console.WriteLine("--- Connected. Start typing! (exit with Ctrl-C)");

        while(true)
        {
            if(Console.KeyAvailable)
            {
                var lineToSend = Console.ReadLine();
                var bytesToSend = encoding.GetBytes(lineToSend + "\r\n");
                stream.Write(bytesToSend, 0, bytesToSend.Length);
                stream.Flush();
            }

            if (stream.DataAvailable)
            {
                var receivedBytesCount = stream.Read(buffer, 0, buffer.Length);
                var receivedString = encoding.GetString(buffer, 0, receivedBytesCount);
                Console.Write(receivedString);
            }
        }
    }
}
```

Cliente SNTP básico (UdpClient)

Consulte RFC 2030 para obtener detalles sobre el protocolo SNTP.

```
using System;
using System.Globalization;
using System.Linq;
using System.Net;
using System.Net.Sockets;

class SntpClient
{
    const int SntpPort = 123;
    static DateTime BaseDate = new DateTime(1900, 1, 1);

    static void Main(string[] args)
    {
        if(args.Length == 0) {
            Console.WriteLine("Simple SNTP client");
            Console.WriteLine();
            Console.WriteLine("Usage: sntpclient <sntp server url> [<local timezone>]");
            Console.WriteLine();
            Console.WriteLine("<local timezone>: a number between -12 and 12 as hours from
UTC");
```

```
            Console.WriteLine("(append .5 for an extra half an hour)");
            return;
        }

        double localTimeZoneInHours = 0;
        if(args.Length > 1)
            localTimeZoneInHours = double.Parse(args[1], CultureInfo.InvariantCulture);

        var udpClient = new UdpClient();
        udpClient.Client.ReceiveTimeout = 5000;

        var sntpRequest = new byte[48];
        sntpRequest[0] = 0x23; //LI=0 (no warning), VN=4, Mode=3 (client)

        udpClient.Send(
            dgram: sntpRequest,
            bytes: sntpRequest.Length,
            hostname: args[0],
            port: SntpPort);

        byte[] sntpResponse;
        try
        {
            IPEndPoint remoteEndpoint = null;
            sntpResponse = udpClient.Receive(ref remoteEndpoint);
        }
        catch(SocketException)
        {
            Console.WriteLine("*** No response received from the server");
            return;
        }

        uint numberOfSeconds;
        if(BitConverter.IsLittleEndian)
            numberOfSeconds = BitConverter.ToUInt32(
                sntpResponse.Skip(40).Take(4).Reverse().ToArray()
                ,0);
        else
            numberOfSeconds = BitConverter.ToUInt32(sntpResponse, 40);

        var date = BaseDate.AddSeconds(numberOfSeconds).AddHours(localTimeZoneInHours);

        Console.WriteLine(
            $"Current date in server: {date:yyyy-MM-dd HH:mm:ss}
UTC{localTimeZoneInHours:+0.#;-0.#;.}");
    }
}
```

Lea Redes en línea: https://riptutorial.com/es/dot-net/topic/35/redes

Capítulo 46: Reflexión

Examples

¿Qué es una asamblea?

Los ensamblajes son el bloque de construcción de cualquier aplicación de Common Language Runtime (CLR) . Cada tipo que defina, junto con sus métodos, propiedades y su código de bytes, se compila y empaqueta dentro de un ensamblaje.

```
using System.Reflection;
```

```
Assembly assembly = this.GetType().Assembly;
```

Los ensamblajes se autodocumentan: no solo contienen tipos, métodos y su código IL, sino también los metadatos necesarios para inspeccionarlos y consumirlos, tanto en la compilación como en el tiempo de ejecución:

```
Assembly assembly = Assembly.GetExecutingAssembly();

foreach (var type in assembly.GetTypes())
{
    Console.WriteLine(type.FullName);
}
```

Las asambleas tienen nombres que describen su identidad única y completa:

```
Console.WriteLine(typeof(int).Assembly.FullName);
// Will print: "mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

Si este nombre incluye un `PublicKeyToken` , se llama un *nombre* `PublicKeyToken` . Nombrar un ensamblaje en firme es el proceso de crear una firma mediante el uso de la clave privada que corresponde a la clave pública distribuida con el ensamblaje. Esta firma se agrega al manifiesto de ensamblaje, que contiene los nombres y los hashes de todos los archivos que forman el ensamblaje, y su `PublicKeyToken` convierte en parte del nombre. Las asambleas que tienen el mismo nombre seguro deben ser idénticas; Los nombres fuertes se utilizan en las versiones y para evitar conflictos de montaje.

Cómo crear un objeto de T utilizando la reflexión.

Usando el constructor por defecto

```
T variable = Activator.CreateInstance(typeof(T));
```

Usando constructor parametrizado

```
T variable = Activator.CreateInstance(typeof(T), arg1, arg2);
```

Creación de objetos y configuración de propiedades utilizando la reflexión.

Digamos que tenemos una clase con `Classy` que tiene propiedad Propertua

```
public class Classy
{
    public string Propertua {get; set;}
}
```

Para establecer `Propertua` utilizando la reflexión:

```
var typeOfClassy = typeof (Classy);
var classy = new Classy();
var prop = typeOfClassy.GetProperty("Propertua");
prop.SetValue(classy, "Value");
```

Obtención de un atributo de una enumeración con reflexión (y almacenamiento en caché)

Los atributos pueden ser útiles para denotar metadatos en enumeraciones. Obtener el valor de esto puede ser lento, por lo que es importante almacenar en caché los resultados.

```
    private static Dictionary<object, object> attributeCache = new Dictionary<object,
object>();

    public static T GetAttribute<T, V>(this V value)
        where T : Attribute
        where V : struct
    {
        object temp;

        // Try to get the value from the static cache.
        if (attributeCache.TryGetValue(value, out temp))
        {
            return (T) temp;
        }
        else
        {
            // Get the type of the struct passed in.
            Type type = value.GetType();
            FieldInfo fieldInfo = type.GetField(value.ToString());

            // Get the custom attributes of the type desired found on the struct.
            T[] attribs = (T[])fieldInfo.GetCustomAttributes(typeof(T), false);

            // Return the first if there was a match.
            var result = attribs.Length > 0 ? attribs[0] : null;

            // Cache the result so future checks won't need reflection.
            attributeCache.Add(value, result);

            return result;
        }
```

```
    }
```

Compara dos objetos con la reflexión.

```
public class Equatable
{
    public string field1;

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        var type = obj.GetType();
        if (GetType() != type)
            return false;

        var fields = type.GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            if (field.GetValue(this) != field.GetValue(obj))
                return false;

        return true;
    }

    public override int GetHashCode()
    {
        var accumulator = 0;
        var fields = GetType().GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            accumulator = unchecked ((accumulator * 937) ^
field.GetValue(this).GetHashCode());

        return accumulator;
    }
}
```

Nota: este ejemplo realiza una comparación basada en campos (ignora los campos estáticos y las propiedades) para simplificar

Lea Reflexión en línea: https://riptutorial.com/es/dot-net/topic/44/reflexion

Capítulo 47: Serialización JSON

Observaciones

JavaScriptSerializer vs Json.NET

La clase `JavaScriptSerializer` se introdujo en .NET 3.5 y es utilizada internamente por la capa de comunicación asíncrona de .NET para aplicaciones habilitadas para AJAX. Se puede utilizar para trabajar con JSON en código administrado.

A pesar de la existencia de la clase `JavaScriptSerializer` , Microsoft recomienda usar la biblioteca de código abierto Json.NET para la serialización y deserialización. Json.NET ofrece un mejor rendimiento y una interfaz más amigable para asignar JSON a clases personalizadas (se necesitaría un objeto `JavaScriptConverter` personalizado para lograr lo mismo con `JavaScriptSerializer`).

Examples

Deserialización utilizando System.Web.Script.Serialization.JavaScriptSerializer

El método `JavaScriptSerializer.Deserialize<T>(input)` intenta deserializar una cadena de JSON válida en un objeto del tipo especificado `<T>` , utilizando las asignaciones predeterminadas admitidas de forma nativa por `JavaScriptSerializer` .

```
using System.Collections;
using System.Web.Script.Serialization;

// ...

string rawJSON = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

JavaScriptSerializer JSS = new JavaScriptSerializer();
Dictionary<string, object> parsedObj = JSS.Deserialize<Dictionary<string, object>>(rawJSON);

string name = parsedObj["Name"].toString();
ArrayList numbers = (ArrayList)parsedObj["Numbers"]
```

Nota: el objeto `JavaScriptSerializer` se introdujo en .NET versión 3.5

Deserialización utilizando Json.NET

```
internal class Sequence{
    public string Name;
    public List<int> Numbers;
}

// ...
```

```
string rawJSON = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

Sequence sequence = JsonConvert.DeserializeObject<Sequence>(rawJSON);
```

Para obtener más información, consulte el sitio oficial de Json.NET .

Nota: Json.NET admite .NET versión 2 y superior.

Serialización utilizando Json.NET

```
[JsonObject("person")]
public class Person
{
    [JsonProperty("name")]
    public string PersonName { get; set; }
    [JsonProperty("age")]
    public int PersonAge { get; set; }
    [JsonIgnore]
    public string Address { get; set; }
}

Person person = new Person { PersonName = "Andrius", PersonAge = 99, Address = "Some address"
};
string rawJson = JsonConvert.SerializeObject(person);

Console.WriteLine(rawJson); // {"name":"Andrius","age":99}
```

Observe cómo las propiedades (y las clases) se pueden decorar con atributos para cambiar su apariencia en la cadena json resultante o para eliminarlas de la cadena json (JsonIgnore).

Puede encontrar más información sobre los atributos de serialización de Json.NET aquí .

En C #, los identificadores públicos se escriben en *PascalCase* por convención. En JSON, la convención es usar *camelCase* para todos los nombres. Puede utilizar un resolutor de contrato para convertir entre los dos.

```
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    [JsonIgnore]
    public string Address { get; set; }
}

public void ToJson() {
    Person person = new Person { Name = "Andrius", Age = 99, Address = "Some address" };
    var resolver = new CamelCasePropertyNamesContractResolver();
    var settings = new JsonSerializerSettings { ContractResolver = resolver };
    string json = JsonConvert.SerializeObject(person, settings);

    Console.WriteLine(json); // {"name":"Andrius","age":99}
}
```

Serialización-Deserialización utilizando Newtonsoft.Json

A diferencia de los otros ayudantes, este utiliza ayudantes de clase estática para serializar y deserializar, por lo que es un poco más fácil de usar que los demás.

```
using Newtonsoft.Json;

var rawJSON      = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
var fibo         = JsonConvert.DeserializeObject<Dictionary<string, object>>(rawJSON);
var rawJSON2     = JsonConvert.SerializeObject(fibo);
```

Vinculación dinámica

Json.NET de Newtonsoft le permite enlazar json dinámicamente (utilizando objetos dinámicos / ExpandoObject) sin la necesidad de crear el tipo explícitamente.

Publicación por entregas

```
dynamic jsonObject = new ExpandoObject();
jsonObject.Title   = "Merchent of Venice";
jsonObject.Author  = "William Shakespeare";
Console.WriteLine(JsonConvert.SerializeObject(jsonObject));
```

De serialización

```
var rawJson = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
dynamic parsedJson = JObject.Parse(rawJson);
Console.WriteLine("Name: " + parsedJson.Name);
Console.WriteLine("Name: " + parsedJson.Numbers.Length);
```

Observe que las claves en el objeto rawJson se han convertido en variables miembro en el objeto dinámico.

Esto es útil en los casos en que una aplicación puede aceptar / producir diferentes formatos de JSON. Sin embargo, se sugiere utilizar un nivel adicional de validación para la cadena Json o para el objeto dinámico generado como resultado de la serialización / deserialización.

Serialización utilizando Json.NET con JsonSerializerSettings

Este serializador tiene algunas características interesantes que el serializador .net json predeterminado no tiene, como el manejo de valor nulo, solo necesita crear el `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
NullValueHandling = NullValueHandling.Ignore});
   return result;
}
```

Otro problema serio del serializador en .net es el bucle de referencia automática. En el caso de un estudiante que está inscrito en un curso, su instancia tiene una propiedad del curso y un curso tiene una colección de estudiantes que significa una `List<Student>` que creará un bucle de referencia. Puedes manejar esto con `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
ReferenceLoopHandling = ReferenceLoopHandling.Ignore});
   return result;
}
```

Puedes poner varias opciones de serializaciones como esta:

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
NullValueHandling = NullValueHandling.Ignore, ReferenceLoopHandling =
ReferenceLoopHandling.Ignore});
   return result;
}
```

Lea Serialización JSON en línea: https://riptutorial.com/es/dot-net/topic/183/serializacion-json

Capítulo 48: Servidores HTTP

Examples

Servidor de archivos HTTP básico de solo lectura (HttpListener)

Notas:

Este ejemplo debe ejecutarse en modo administrativo.

Sólo se admite un cliente simultáneo.

Para simplificar, se asume que los nombres de los archivos son todos ASCII (para la parte del *nombre* del *archivo* en el encabezado *Content-Disposition*) y los errores de acceso a los archivos no se manejan.

```
using System;
using System.IO;
using System.Net;

class HttpFileServer
{
    private static HttpListenerResponse response;
    private static HttpListener listener;
    private static string baseFilesystemPath;

    static void Main(string[] args)
    {
        if (!HttpListener.IsSupported)
        {
            Console.WriteLine(
                "*** HttpListener requires at least Windows XP SP2 or Windows Server 2003.");
            return;
        }

        if(args.Length < 2)
        {
            Console.WriteLine("Basic read-only HTTP file server");
            Console.WriteLine();
            Console.WriteLine("Usage: httpfileserver <base filesystem path> <port>");
            Console.WriteLine("Request format: http://url:port/path/to/file.ext");
            return;
        }

        baseFilesystemPath = Path.GetFullPath(args[0]);
        var port = int.Parse(args[1]);

        listener = new HttpListener();
        listener.Prefixes.Add("http://*:" + port + "/");
        listener.Start();

        Console.WriteLine("--- Server stated, base path is: " + baseFilesystemPath);
        Console.WriteLine("--- Listening, exit with Ctrl-C");
        try
        {
```

```
            ServerLoop();
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex);
            if(response != null)
            {
                SendErrorResponse(500, "Internal server error");
            }
        }
    }

    static void ServerLoop()
    {
        while(true)
        {
            var context = listener.GetContext();

            var request = context.Request;
            response = context.Response;
            var fileName = request.RawUrl.Substring(1);
            Console.WriteLine(
                "--- Got {0} request for: {1}",
                request.HttpMethod, fileName);

            if (request.HttpMethod.ToUpper() != "GET")
            {
                SendErrorResponse(405, "Method must be GET");
                continue;
            }

            var fullFilePath = Path.Combine(baseFilesystemPath, fileName);
            if(!File.Exists(fullFilePath))
            {
                SendErrorResponse(404, "File not found");
                continue;
            }

            Console.Write("    Sending file...");
            using (var fileStream = File.OpenRead(fullFilePath))
            {
                response.ContentType = "application/octet-stream";
                response.ContentLength64 = (new FileInfo(fullFilePath)).Length;
                response.AddHeader(
                    "Content-Disposition",
                    "Attachment; filename=\"" + Path.GetFileName(fullFilePath) + "\"");
                fileStream.CopyTo(response.OutputStream);
            }

            response.OutputStream.Close();
            response = null;
            Console.WriteLine(" Ok!");
        }
    }

    static void SendErrorResponse(int statusCode, string statusResponse)
    {
        response.ContentLength64 = 0;
        response.StatusCode = statusCode;
        response.StatusDescription = statusResponse;
        response.OutputStream.Close();
```

```
        Console.WriteLine("*** Sent error: {0} {1}", statusCode, statusResponse);
    }
}
```

Servidor de archivos HTTP básico de solo lectura (ASP.NET Core)

1 - Cree una carpeta vacía, contendrá los archivos creados en los siguientes pasos.

2 - Cree un archivo llamado `project.json` con el siguiente contenido (ajuste el número de puerto y `rootDirectory` según corresponda):

```
{
  "dependencies": {
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel --server.urls http://localhost:60000"
  },

  "frameworks": {
    "dnxcore50": { }
  },

  "fileServer": {
    "rootDirectory": "c:\\users\\username\\Documents"
  }
}
```

3 - Crea un archivo llamado `Startup.cs` con el siguiente código:

```
using System;
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.FileProviders;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.StaticFiles;
using Microsoft.Extensions.Configuration;

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        var builder = new ConfigurationBuilder();
        builder.AddJsonFile("project.json");
        var config = builder.Build();
        var rootDirectory = config["fileServer:rootDirectory"];
        Console.WriteLine("File server root directory: " + rootDirectory);

        var fileProvider = new PhysicalFileProvider(rootDirectory);

        var options = new StaticFileOptions();
        options.ServeUnknownFileTypes = true;
        options.FileProvider = fileProvider;
        options.OnPrepareResponse = context =>
        {
            context.Context.Response.ContentType = "application/octet-stream";
```

```
            context.Context.Response.Headers.Add(
                "Content-Disposition",
                $"Attachment; filename=\"{context.File.Name}\"");
        };

        app.UseStaticFiles(options);
    }
}
```

4 - Abra un símbolo del sistema, navegue a la carpeta y ejecute:

```
dnvm use 1.0.0-rc1-final -r coreclr -p
dnu restore
```

Nota: estos comandos deben ejecutarse solo una vez. Use la `dnvm list` para verificar el número real de la última versión instalada del CLR principal.

5 - Inicia el servidor con: `dnx web` . Los archivos ahora se pueden solicitar en `http://localhost:60000/path/to/file.ext` .

Para simplificar, se asume que los nombres de los archivos son todos ASCII (para la parte del nombre del archivo en el encabezado Content-Disposition) y los errores de acceso a los archivos no se manejan.

Lea Servidores HTTP en línea: https://riptutorial.com/es/dot-net/topic/53/servidores-http

Capítulo 49: Sistema de envasado NuGet

Observaciones

NuGet.org :

> NuGet es el administrador de paquetes para la plataforma de desarrollo de Microsoft, incluido .NET. Las herramientas de cliente de NuGet proporcionan la capacidad de producir y consumir paquetes. La Galería NuGet es el repositorio central de paquetes utilizado por todos los autores y consumidores de paquetes.

Imágenes en ejemplos cortesía de NuGet.org .

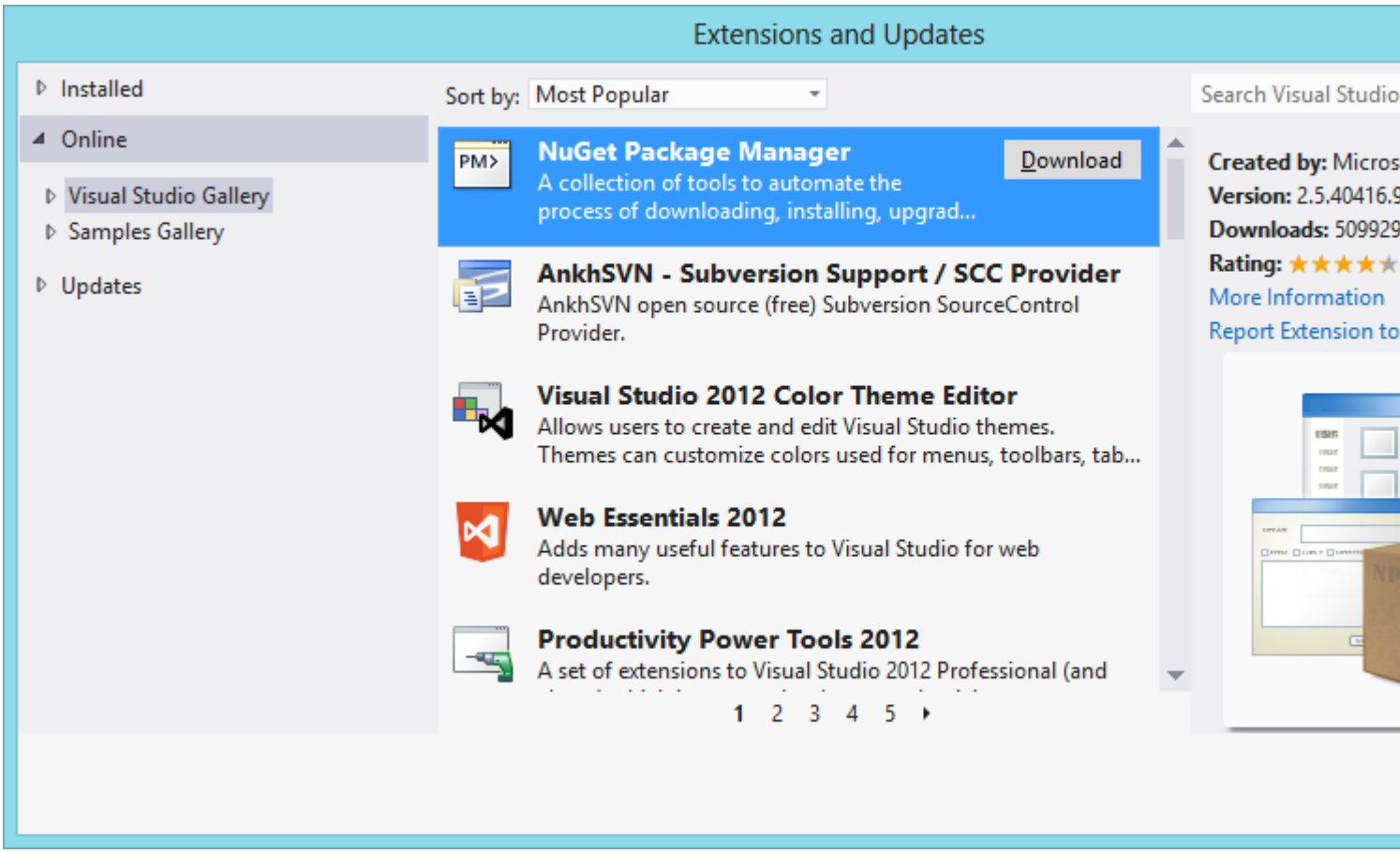
Examples

Instalación del Gestor de paquetes NuGet

Para poder administrar los paquetes de sus proyectos, necesita NuGet Package Manager. Esta es una extensión de Visual Studio, que se explica en los documentos oficiales: Instalación y actualización de NuGet Client .

A partir de Visual Studio 2012, NuGet se incluye en todas las ediciones y se puede usar desde: Herramientas -> NuGet Package Manager -> Package Manager Console.

Lo haces a través del menú Herramientas de Visual Studio, haciendo clic en Extensiones y actualizaciones:



Esto instala tanto la GUI:

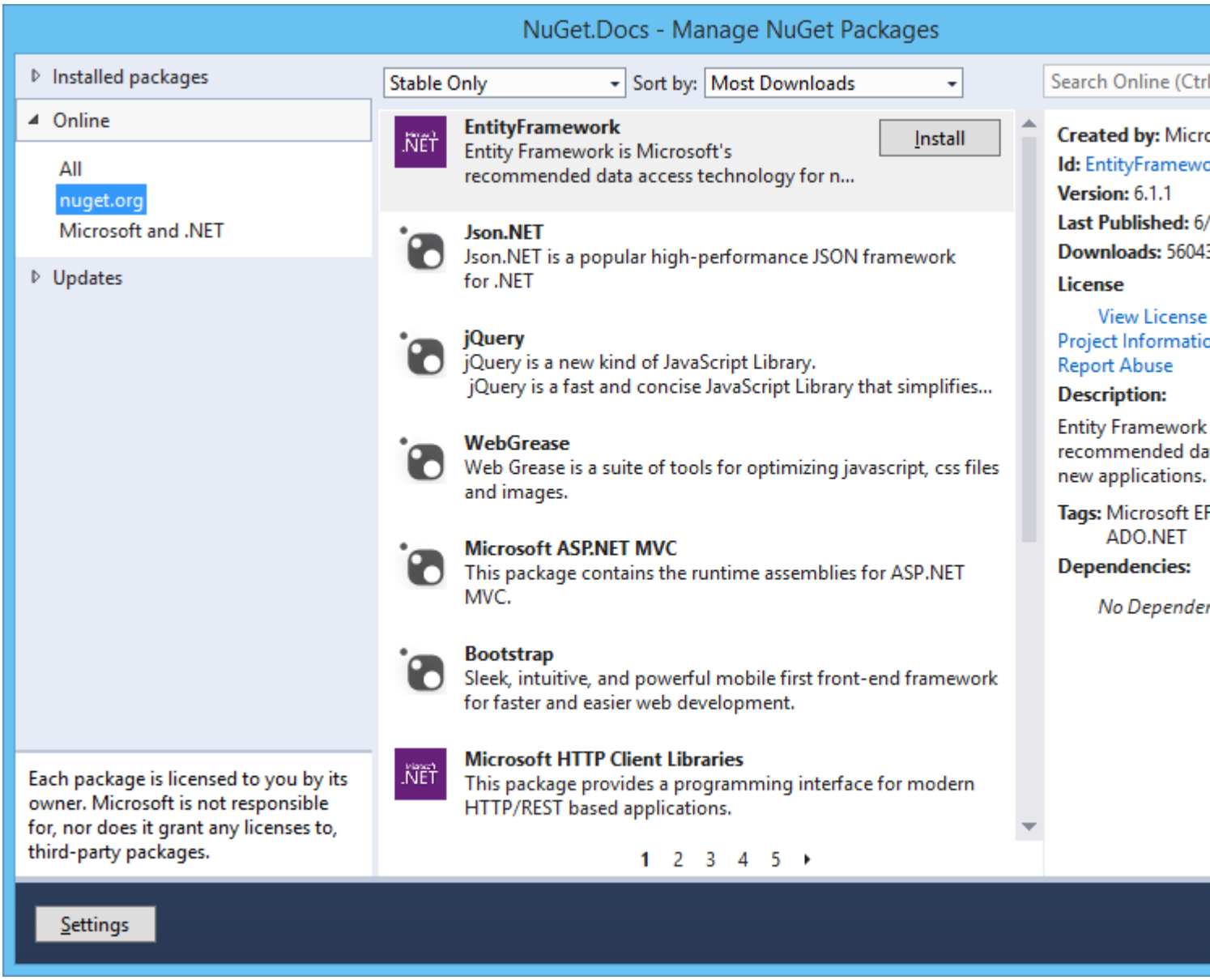
- Disponible haciendo clic en "Administrar paquetes NuGet ..." en un proyecto o en su carpeta Referencias

Y la consola del administrador de paquetes:

- Herramientas -> NuGet Package Manager -> Package Manager Console.

Gestión de paquetes a través de la interfaz de usuario

Cuando hace clic con el botón derecho en un proyecto (o en su carpeta Referencias), puede hacer clic en la opción "Administrar paquetes NuGet ...". Esto muestra el cuadro de diálogo Administrador de paquetes .



Gestionando paquetes a través de la consola.

Haga clic en los menús Herramientas -> NuGet Package Manager -> Package Manager Console para mostrar la consola en su IDE. Documentación oficial aquí .

Aquí puede emitir, entre otros, `install-package` comandos `install-package` que instalan el paquete introducido en el "Proyecto predeterminado" seleccionado actualmente:

```
Install-Package Elmah
```

También puede proporcionar el proyecto para instalar el paquete, anulando el proyecto seleccionado en el menú desplegable "Proyecto predeterminado":

```
Install-Package Elmah -ProjectName MyFirstWebsite
```

Actualizando un paquete

Para actualizar un paquete usa el siguiente comando:

```
PM> Update-Package EntityFramework
```

donde EntityFramework es el nombre del paquete que se actualizará. Tenga en cuenta que la actualización se ejecutará para todos los proyectos, por lo que es diferente de `Install-Package EntityFramework` que se instalaría solo en el "Proyecto predeterminado".

También puede especificar un solo proyecto explícitamente:

```
PM> Update-Package EntityFramework -ProjectName MyFirstWebsite
```

Desinstalar un paquete

```
PM> Uninstall-Package EntityFramework
```

Desinstalar un paquete de un proyecto en una solución

```
PM> Uninstall-Package -ProjectName MyProjectB EntityFramework
```

Instalando una versión específica de un paquete

```
PM> Install-Package EntityFramework -Version 6.1.2
```

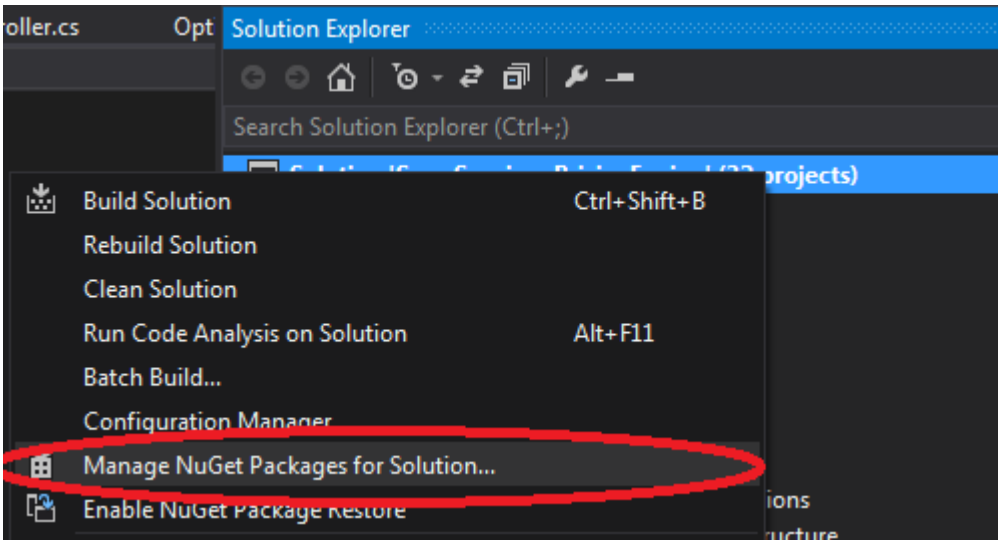
Agregando un feed fuente de paquete (MyGet, Klondike, ect)

```
nuget sources add -name feedname -source http://sourcefeedurl
```

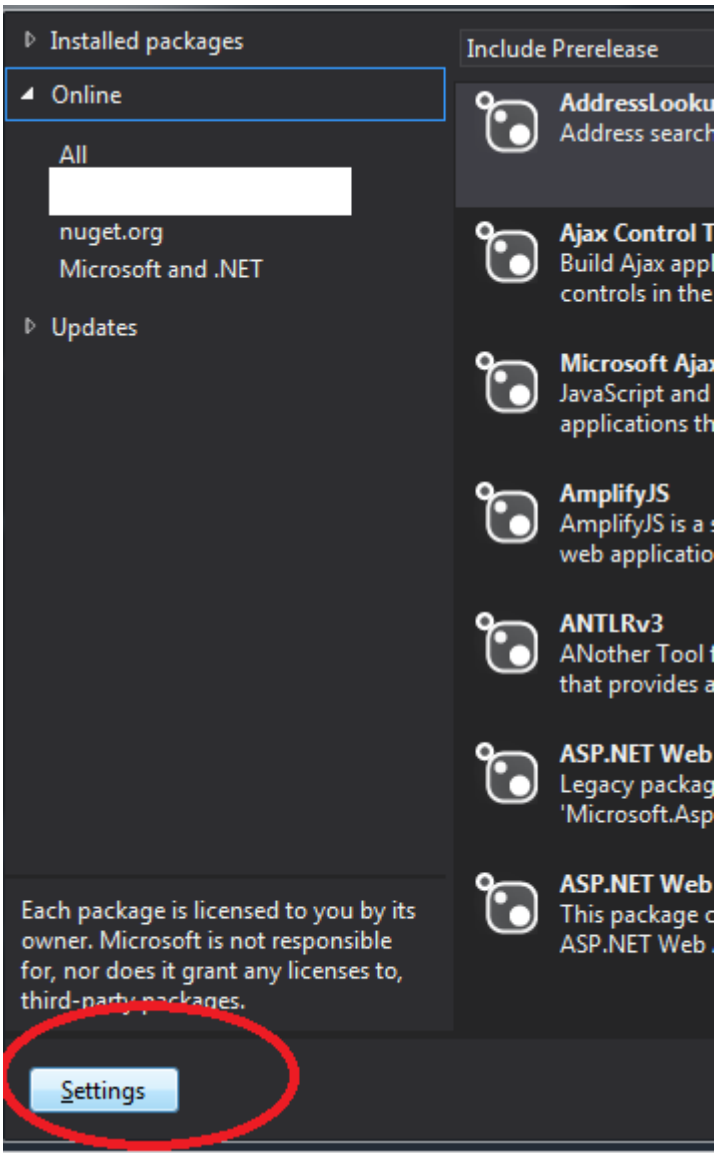
Usando diferentes fuentes de paquetes Nuget (locales) usando la interfaz de usuario

Es común que la compañía configure su propio servidor nuget para la distribución de paquetes en diferentes equipos.

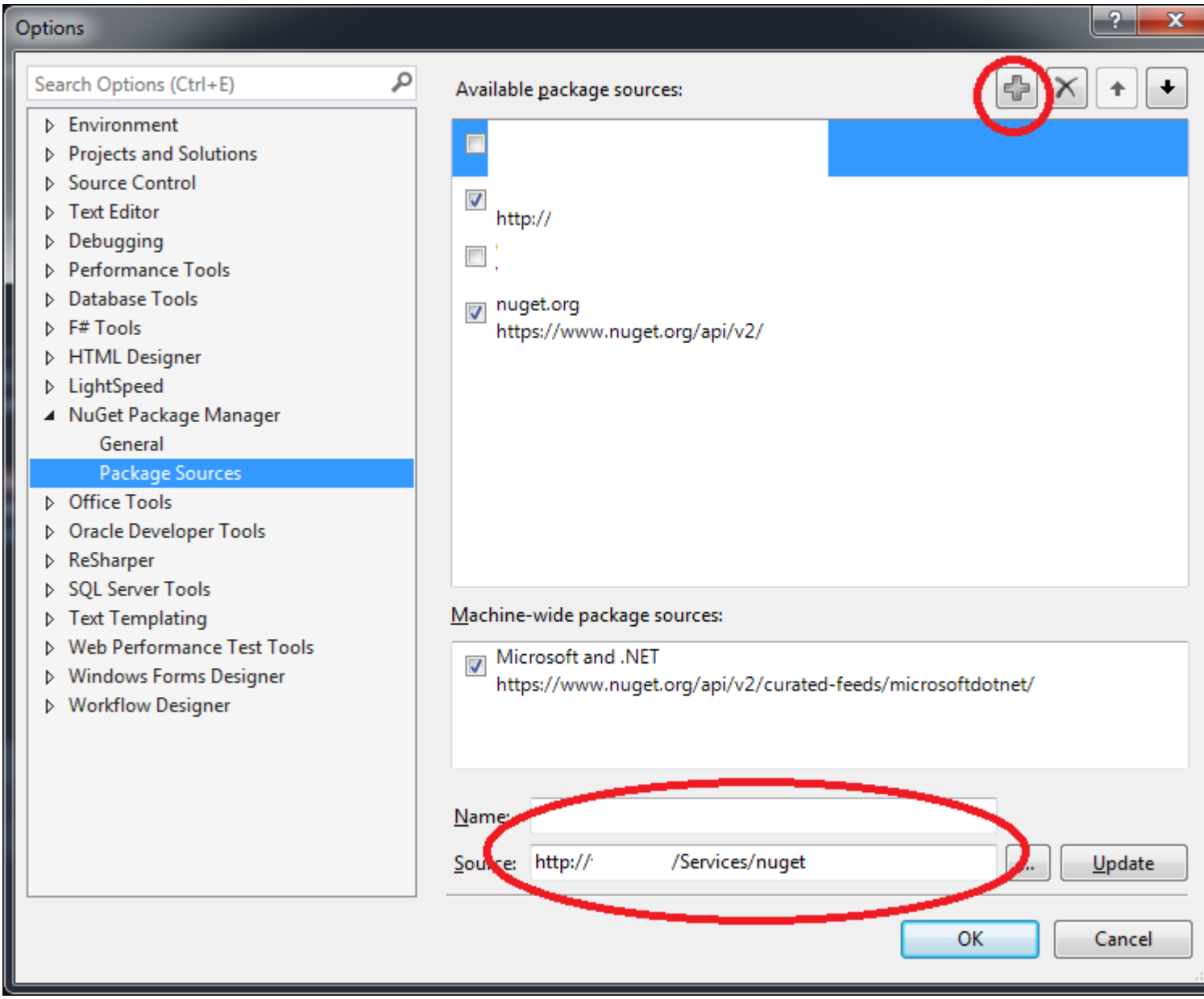
1. Vaya al Explorador de soluciones y haga clic en el botón `derecho del mouse`, luego elija `Manage NuGet Packages for Solution`



2. En la ventana que se abre, haga clic en `Settings`



3. Haga clic en `+` en la esquina superior derecha y luego agregue el nombre y la URL que apunta a su servidor local de nuget.



desinstalar una versión específica del paquete

```
PM> uninstall-Package EntityFramework -Version 6.1.2
```

Lea Sistema de envasado NuGet en línea: https://riptutorial.com/es/dot-net/topic/43/sistema-de-envasado-nuget

Capítulo 50: SpeechRecognitionEngine clase para reconocer el habla

Sintaxis

- SpeechRecognitionEngine ()
- SpeechRecognitionEngine.LoadGrammar (gramática gramatical)
- SpeechRecognitionEngine.SetInputToDefaultAudioDevice ()
- SpeechRecognitionEngine.RecognizeAsync (modo RecognizeMode)
- Constructor de gramática()
- GrammarBuilder.Append (Opciones de opciones)
- Opciones (cadena de parámetros [] opciones)
- Gramática (constructor GrammarBuilder)

Parámetros

`LoadGrammar` : Parámetros	Detalles
gramática	La gramática a cargar. Por ejemplo, un objeto `DictationGrammar` para permitir el dictado de texto libre.
`RecognizeAsync` : Parámetros	**Detalles**
modo	El `RecognizeMode` para el reconocimiento actual: `Single` para un solo reconocimiento, `Multiple` para permitir el múltiple.
`GrammarBuilder.Append` : Parámetros	**Detalles**
elecciones	Anexa algunas opciones al constructor de gramática. Esto significa que, cuando el usuario ingresa el habla, el reconocedor puede seguir diferentes "ramas" de una gramática.
Constructor de `Choices` : Parámetros	**Detalles**
elecciones	Un conjunto de opciones para el constructor de gramática. Ver `GrammarBuilder.Append` .
Constructor de `Grammar` : parámetro	**Detalles**
constructor	El `GrammarBuilder` para construir una `Grammar` de.

Observaciones

Para usar `SpeechRecognitionEngine` , su versión de Windows debe tener habilitado el reconocimiento de voz.

Debe agregar una referencia a `System.Speech.dll` antes de poder usar las clases de voz.

Examples

Reconocimiento asíncrono de voz para dictado de texto libre.

```
using System.Speech.Recognition;

// ...

SpeechRecognitionEngine recognitionEngine = new SpeechRecognitionEngine();
recognitionEngine.LoadGrammar(new DictationGrammar());
recognitionEngine.SpeechRecognized += delegate(object sender, SpeechRecognizedEventArgs e)
{
    Console.WriteLine("You said: {0}", e.Result.Text);
};
recognitionEngine.SetInputToDefaultAudioDevice();
recognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
```

Reconocimiento asíncrono del habla basado en un conjunto restringido de frases

```
SpeechRecognitionEngine recognitionEngine = new SpeechRecognitionEngine();
GrammarBuilder builder = new GrammarBuilder();
builder.Append(new Choices("I am", "You are", "He is", "She is", "We are", "They are"));
builder.Append(new Choices("friendly", "unfriendly"));
recognitionEngine.LoadGrammar(new Grammar(builder));
recognitionEngine.SpeechRecognized += delegate(object sender, SpeechRecognizedEventArgs e)
{
    Console.WriteLine("You said: {0}", e.Result.Text);
};
recognitionEngine.SetInputToDefaultAudioDevice();
recognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
```

Lea SpeechRecognitionEngine clase para reconocer el habla en línea:
https://riptutorial.com/es/dot-net/topic/69/speechrecognitionengine-clase-para-reconocer-el-habla

Capítulo 51: System.IO

Examples

Leyendo un archivo de texto usando StreamReader

```
string fullOrRelativePath = "testfile.txt";

string fileData;

using (var reader = new StreamReader(fullOrRelativePath))
{
    fileData = reader.ReadToEnd();
}
```

Tenga en cuenta que esta sobrecarga del constructor `StreamReader` realiza alguna detección de codificación automática, que puede o no ser compatible con la codificación real utilizada en el archivo.

Tenga en cuenta que hay algunos métodos convenientes que leen todo el texto del archivo disponible en la clase `System.IO.File` , a saber, `File.ReadAllText(path)` y `File.ReadAllLines(path)` .

Lectura / escritura de datos usando System.IO.File

Primero, veamos tres formas diferentes de extraer datos de un archivo.

```
string fileText = File.ReadAllText(file);
string[] fileLines = File.ReadAllLines(file);
byte[] fileBytes = File.ReadAllBytes(file);
```

- En la primera línea, leemos todos los datos en el archivo como una cadena.
- En la segunda línea, leemos los datos del archivo en una matriz de cadenas. Cada línea del archivo se convierte en un elemento de la matriz.
- En el tercero leemos los bytes del archivo.

A continuación, veamos tres métodos diferentes de **agregar** datos a un archivo. Si el archivo que especifica no existe, cada método creará automáticamente el archivo antes de intentar agregarle los datos.

```
 File.AppendAllText(file, "Here is some data that is\nappended to the file.");
 File.AppendAllLines(file, new string[2] { "Here is some data that is", "appended to the
file." });
 using (StreamWriter stream = File.AppendText(file))
 {
     stream.WriteLine("Here is some data that is");
     stream.Write("appended to the file.");
 }
```

- En la primera línea, simplemente agregamos una cadena al final del archivo especificado.
- En la segunda línea, agregamos cada elemento de la matriz a una nueva línea en el archivo.
- Finalmente, en la tercera línea, usamos `File.AppendText` para abrir un streamwriter que agregará los datos que se le escriban.

Y por último, veamos tres métodos diferentes para **escribir** datos en un archivo. La diferencia entre *agregar* y *escribir* es que la escritura **sobrescribe** los datos en el archivo mientras se **agrega agrega** a los datos en el archivo. Si el archivo que especifica no existe, cada método creará automáticamente el archivo antes de intentar escribirle los datos.

```
File.WriteAllText(file, "here is some data\nin this file.");
File.WriteAllLines(file, new string[2] { "here is some data", "in this file" });
File.WriteAllBytes(file, new byte[2] { 0, 255 });
```

- La primera línea escribe una cadena en el archivo.
- La segunda línea escribe cada cadena en la matriz en su propia línea en el archivo.
- Y la tercera línea le permite escribir una matriz de bytes en el archivo.

Puertos serie utilizando System.IO.SerialPorts

Iterando sobre puertos seriales conectados

```
using System.IO.Ports;
string[] ports = SerialPort.GetPortNames();
for (int i = 0; i < ports.Length; i++)
{
    Console.WriteLine(ports[i]);
}
```

Creación de una instancia de un objeto System.IO.SerialPort

```
using System.IO.Ports;
SerialPort port = new SerialPort();
SerialPort port = new SerialPort("COM 1"); ;
SerialPort port = new SerialPort("COM 1", 9600);
```

NOTA : Esas son solo tres de las siete sobrecargas del constructor para el tipo SerialPort.

Lectura / escritura de datos sobre el SerialPort

La forma más sencilla es utilizar los métodos `SerialPort.Read` y `SerialPort.Write` . Sin embargo, también puede recuperar un objeto `System.IO.Stream` que puede usar para transmitir datos a través del SerialPort. Para hacer esto, use `SerialPort.BaseStream` .

Leyendo

```
int length = port.BytesToRead;
//Note that you can swap out a byte-array for a char-array if you prefer.
byte[] buffer = new byte[length];
port.Read(buffer, 0, length);
```

También puede leer todos los datos disponibles:

```
string curData = port.ReadExisting();
```

O simplemente lea la primera nueva línea encontrada en los datos entrantes:

```
string line = port.ReadLine();
```

Escritura

La forma más fácil de escribir datos a través de SerialPort es:

```
port.Write("here is some text to be sent over the serial port.");
```

Sin embargo, también puede enviar datos de esta manera cuando sea necesario:

```
//Note that you can swap out the byte-array with a char-array if you so choose.
byte[] data = new byte[1] { 255 };
port.Write(data, 0, data.Length);
```

Lea System.IO en línea: https://riptutorial.com/es/dot-net/topic/5259/system-io

Capítulo 52: System.Net.Mail

Observaciones

Es importante disponer un System.Net.MailMessage porque todos los archivos adjuntos contienen un Stream y estos Streams se deben liberar lo antes posible. La declaración de uso garantiza que el objeto desechable se deseche también en caso de excepciones

Examples

MailMessage

Aquí está el ejemplo de creación de mensaje de correo con archivos adjuntos. Después de crear enviamos este mensaje con la ayuda de la clase `SmtpClient` . Aquí se utiliza el puerto predeterminado de 25.

```
public class clsMail
{
    private static bool SendMail(string mailfrom, List<string>replytos, List<string> mailtos,
List<string> mailccs, List<string> mailbccs, string body, string subject, List<string>
Attachment)
    {
        try
        {
            using(MailMessage MyMail = new MailMessage())
            {
                MyMail.From = new MailAddress(mailfrom);
                foreach (string mailto in mailtos)
                    MyMail.To.Add(mailto);

                if (replytos != null && replytos.Any())
                {
                    foreach (string replyto in replytos)
                        MyMail.ReplyToList.Add(replyto);
                }

                if (mailccs != null && mailccs.Any())
                {
                    foreach (string mailcc in mailccs)
                        MyMail.CC.Add(mailcc);
                }

                if (mailbccs != null && mailbccs.Any())
                {
                    foreach (string mailbcc in mailbccs)
                        MyMail.Bcc.Add(mailbcc);
                }

                MyMail.Subject = subject;
                MyMail.IsBodyHtml = true;
                MyMail.Body = body;
                MyMail.Priority = MailPriority.Normal;
```

```
                if (Attachment != null && Attachment.Any())
                {
                    System.Net.Mail.Attachment attachment;
                    foreach (var item in Attachment)
                    {
                        attachment = new System.Net.Mail.Attachment(item);
                        MyMail.Attachments.Add(attachment);
                    }
                }

                SmtpClient smtpMailObj = new SmtpClient();
                smtpMailObj.Host = "your host";
                smtpMailObj.Port = 25;
                smtpMailObj.Credentials = new System.Net.NetworkCredential("uid", "pwd");

                smtpMailObj.Send(MyMail);
                return true;
            }
        }
        catch
        {
            return false;
        }
    }
}
```

Correo con archivo adjunto

`MailMessage` representa un mensaje de correo que puede enviarse aún más usando la clase `SmtpClient` . Se pueden agregar varios archivos adjuntos (archivos) al mensaje de correo.

```
using System.Net.Mail;

using(MailMessage myMail = new MailMessage())
{
     Attachment attachment = new Attachment(path);
     myMail.Attachments.Add(attachment);

     // further processing to send the mail message

}
```

Lea System.Net.Mail en línea: https://riptutorial.com/es/dot-net/topic/7440/system-net-mail

Capítulo 53: System.Reflection.Emit namespace

Examples

Creando un ensamblaje dinámicamente.

```
using System;
using System.Reflection;
using System.Reflection.Emit;

class DemoAssemblyBuilder
{
    public static void Main()
    {
        // An assembly consists of one or more modules, each of which
        // contains zero or more types. This code creates a single-module
        // assembly, the most common case. The module contains one type,
        // named "MyDynamicType", that has a private field, a property
        // that gets and sets the private field, constructors that
        // initialize the private field, and a method that multiplies
        // a user-supplied number by the private field value and returns
        // the result. In C# the type might look like this:
        /*
        public class MyDynamicType
        {
            private int m_number;

            public MyDynamicType() : this(42) {}
            public MyDynamicType(int initNumber)
            {
                m_number = initNumber;
            }

            public int Number
            {
                get { return m_number; }
                set { m_number = value; }
            }

            public int MyMethod(int multiplier)
            {
                return m_number * multiplier;
            }
        }
        */

        AssemblyName aName = new AssemblyName("DynamicAssemblyExample");
        AssemblyBuilder ab =
            AppDomain.CurrentDomain.DefineDynamicAssembly(
                aName,
                AssemblyBuilderAccess.RunAndSave);

        // For a single-module assembly, the module name is usually
        // the assembly name plus an extension.
```

```
        ModuleBuilder mb =
            ab.DefineDynamicModule(aName.Name, aName.Name + ".dll");

        TypeBuilder tb = mb.DefineType(
            "MyDynamicType",
             TypeAttributes.Public);

        // Add a private field of type int (Int32).
        FieldBuilder fbNumber = tb.DefineField(
            "m_number",
            typeof(int),
            FieldAttributes.Private);

        // Next, we make a simple sealed method.
        MethodBuilder mbMyMethod = tb.DefineMethod(
            "MyMethod",
            MethodAttributes.Public,
            typeof(int),
            new[] { typeof(int) });

        ILGenerator il = mbMyMethod.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this - always the first argument of any instance
method
        il.Emit(OpCodes.Ldfld, fbNumber);
        il.Emit(OpCodes.Ldarg_1); // Load the integer argument
        il.Emit(OpCodes.Mul); // Multiply the two numbers with no overflow checking
        il.Emit(OpCodes.Ret); // Return

        // Next, we build the property. This involves building the property itself, as well as
the
        // getter and setter methods.
        PropertyBuilder pbNumber = tb.DefineProperty(
            "Number", // Name
            PropertyAttributes.None,
            typeof(int), // Type of the property
            new Type[0]); // Types of indices, if any

        MethodBuilder mbSetNumber = tb.DefineMethod(
            "set_Number", // Name - setters are set_Property by convention
            // Setter is a special method and we don't want it to appear to callers from C#
            MethodAttributes.PrivateScope | MethodAttributes.HideBySig |
MethodAttributes.Public | MethodAttributes.SpecialName,
            typeof(void), // Setters don't return a value
            new[] { typeof(int) }); // We have a single argument of type System.Int32

        // To generate the body of the method, we'll need an IL generator
        il = mbSetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this
        il.Emit(OpCodes.Ldarg_1); // Load the new value
        il.Emit(OpCodes.Stfld, fbNumber); // Save the new value to this.m_number
        il.Emit(OpCodes.Ret); // Return

        // Finally, link the method to the setter of our property
        pbNumber.SetSetMethod(mbSetNumber);

        MethodBuilder mbGetNumber = tb.DefineMethod(
            "get_Number",
            MethodAttributes.PrivateScope | MethodAttributes.HideBySig |
MethodAttributes.Public | MethodAttributes.SpecialName,
            typeof(int),
            new Type[0]);
```

```
        il = mbGetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this
        il.Emit(OpCodes.Ldfld, fbNumber); // Load the value of this.m_number
        il.Emit(OpCodes.Ret); // Return the value

        pbNumber.SetGetMethod(mbGetNumber);

        // Finally, we add the two constructors.
        // Constructor needs to call the constructor of the parent class, or another
constructor in the same class
        ConstructorBuilder intConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis,
new[] { typeof(int) });
        il = intConstructor.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Call, typeof(object).GetConstructor(new Type[0])); // call parent's
constructor
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldarg_1); // our int argument
        il.Emit(OpCodes.Stfld, fbNumber); // store argument in this.m_number
        il.Emit(OpCodes.Ret);

        var parameterlessConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis,
new Type[0]);
        il = parameterlessConstructor.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldc_I4_S, (byte)42); // load 42 as an integer constant
        il.Emit(OpCodes.Call, intConstructor); // call this(42)
        il.Emit(OpCodes.Ret);

        // And make sure the type is created
        Type ourType = tb.CreateType();

        // The types from the assembly can be used directly using reflection, or we can save
the assembly to use as a reference
        object ourInstance = Activator.CreateInstance(ourType);
        Console.WriteLine(ourType.GetProperty("Number").GetValue(ourInstance)); // 42

        // Save the assembly for use elsewhere. This is very useful for debugging - you can
use e.g. ILSpy to look at the equivalent IL/C# code.
        ab.Save(@"DynamicAssemblyExample.dll");
        // Using newly created type
        var myDynamicType = tb.CreateType();
        var myDynamicTypeInstance = Activator.CreateInstance(myDynamicType);

        Console.WriteLine(myDynamicTypeInstance.GetType()); // MyDynamicType

        var numberField = myDynamicType.GetField("m_number", BindingFlags.NonPublic |
BindingFlags.Instance);
        numberField.SetValue (myDynamicTypeInstance, 10);

        Console.WriteLine(numberField.GetValue(myDynamicTypeInstance)); // 10
    }
}
```

Lea System.Reflection.Emit namespace en línea: https://riptutorial.com/es/dot-net/topic/74/system-reflection-emit-namespace

Capítulo 54: System.Runtime.Caching.MemoryCache (ObjectCache)

Examples

Agregar elemento a caché (conjunto)

La función de ajuste inserta una entrada de caché en el caché utilizando una instancia de CacheItem para proporcionar la clave y el valor de la entrada de caché.

Esta función `ObjectCache.Set(CacheItem, CacheItemPolicy)`

```
private static bool SetToCache()
{
    string key = "Cache_Key";
    string value = "Cache_Value";

    //Get a reference to the default MemoryCache instance.
    var cacheContainer = MemoryCache.Default;

    var policy = new CacheItemPolicy()
    {
        AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(DEFAULT_CACHE_EXPIRATION_MINUTES)
     };
     var itemToCache = new CacheItem(key, value); //Value is of type object.
     cacheContainer.Set(itemToCache, policy);
}
```

System.Runtime.Caching.MemoryCache (ObjectCache)

Esta función obtiene el caché de formulario del elemento existente, y si el elemento no existe en el caché, recuperará el elemento según la función valueFetchFactory.

```
    public static TValue GetExistingOrAdd<TValue>(string key, double minutesForExpiration,
Func<TValue> valueFetchFactory)
    {
        try
        {
            //The Lazy class provides Lazy initialization which will evaluate
            //the valueFetchFactory only if item is not in the cache.
            var newValue = new Lazy<TValue>(valueFetchFactory);

            //Setup the cache policy if item will be saved back to cache.
            CacheItemPolicy policy = new CacheItemPolicy()
            {
                AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(minutesForExpiration)
            };

            //returns existing item form cache or add the new value if it does not exist.
```

```
            var cachedItem = _cacheContainer.AddOrGetExisting(key, newValue, policy) as
Lazy<TValue>;

            return (cachedItem ?? newValue).Value;
        }
        catch (Exception excep)
        {
            return default(TValue);
        }
    }
```

Lea System.Runtime.Caching.MemoryCache (ObjectCache) en línea:
https://riptutorial.com/es/dot-net/topic/76/system-runtime-caching-memorycache--objectcache-

Capítulo 55: Tipos personalizados

Observaciones

Normalmente, una `struct` se usa solo cuando el rendimiento es muy importante. Dado que los tipos de valor viven en la pila, se puede acceder a ellos mucho más rápido que las clases. Sin embargo, la pila tiene mucho menos espacio que el montón, por lo que las estructuras deben mantenerse pequeñas (Microsoft recomienda que las `struct` ocupen más de 16 bytes).

Una `class` es el tipo más usado (de estos tres) en C #, y generalmente es lo que debe ir primero.

Se utiliza una `enum` siempre que pueda tener una lista clara y clara de elementos que solo deben definirse una vez (en tiempo de compilación). Las enumeraciones son útiles para los programadores como una referencia liviana a algún valor: en lugar de definir una lista de variables `constant` para comparar, puede usar una enumeración y obtener soporte de Intellisense para asegurarse de que no use accidentalmente un valor incorrecto.

Examples

Definición de Struct

Las estructuras heredan de System.ValueType, son tipos de valor y viven en la pila. Cuando los tipos de valor se pasan como un parámetro, se pasan por valor.

```
Struct MyStruct
{
    public int x;
    public int y;
}
```

Pasado por valor significa que el valor del parámetro se *copia* para el método, y cualquier cambio realizado en el parámetro en el método no se refleja fuera del método. Por ejemplo, considere el siguiente código, que llama a un método llamado `AddNumbers` , que pasa las variables `a` y `b` , que son de tipo `int` , que es un tipo de valor.

```
int a = 5;
int b = 6;

AddNumbers(a,b);

public AddNumbers(int x, int y)
{
    int z = x + y; // z becomes 11
    x = x + 5; // now we changed x to be 10
    z = x + y; // now z becomes 16
```

```
}
```

A pesar de que hemos añadido a 5 `x` dentro del método, el valor de `a` se mantiene sin cambios, porque es un tipo de valor, y eso significa que `x` era una *copia* de `a` 'valor de s, pero en realidad no `a` .

Recuerde, los tipos de valor viven en la pila y se pasan por valor.

Definición de clase

Las clases heredadas de System.Object, son tipos de referencia y viven en el montón. Cuando los tipos de referencia se pasan como un parámetro, se pasan por referencia.

```
public Class MyClass
{
    public int a;
    public int b;
}
```

Pasado por referencia significa que una *referencia* al parámetro se pasa al método, y cualquier cambio en el parámetro se reflejará fuera del método cuando regrese, porque la referencia es *exactamente el mismo objeto en la memoria* . Usemos el mismo ejemplo que antes, pero primero "envolveremos" los `int` en una clase.

```
MyClass instanceOfMyClass = new MyClass();
instanceOfMyClass.a = 5;
instanceOfMyClass.b = 6;

AddNumbers(instanceOfMyClass);

public AddNumbers(MyClass sample)
{
    int z = sample.a + sample.b; // z becomes 11
    sample.a = sample.a + 5; // now we changed a to be 10
    z = sample.a + sample.b; // now z becomes 16
}
```

Esta vez, cuando cambiamos `sample.a` a `10` , el valor de `instanceOfMyClass.a` *también* cambia, porque se *pasó por referencia* . Pasado por referencia significa que se pasó al método una *referencia* (también llamada a veces *puntero*) al objeto, en lugar de una copia del objeto en sí.

Recuerde, los tipos de referencia viven en el montón y se pasan por referencia.

Definición de enumeración

Un enum es un tipo especial de clase. La palabra clave `enum` le

dice al compilador que esta clase hereda de la clase abstracta System.Enum. Las enumeraciones se utilizan para distintas listas de elementos.

```
public enum MyEnum
{
    Monday = 1,
    Tuesday,
    Wednesday,
    //...
}
```

Puede pensar en una enumeración como una forma conveniente de asignar constantes a algún valor subyacente. La enumeración definida anteriormente declara valores para cada día de la semana y comienza con `1` . `Tuesday` se asignaría automáticamente a `2` , el `Wednesday` a `3` , etc.

Por defecto, las enumeraciones usan `int` como el tipo subyacente y comienzan en 0, pero puede usar cualquiera de los siguientes *tipos integrales* : `byte, sbyte, short, ushort, int, uint, long, or ulong` , y puede especificar valores explícitos para cualquier ít. Si algunos elementos se especifican explícitamente, pero otros no, cada elemento después del último definido se incrementará en 1.

Volveremos a utilizar este ejemplo *echando* algún otro valor a un *MyEnum* de este modo:

```
MyEnum instance = (MyEnum)3; // the variable named 'instance' gets a
                             //value of MyEnum.Wednesday, which maps to 3.

int x = 2;
instance = (MyEnum)x; // now 'instance' has a value of MyEnum.Tuesday
```

Otro tipo de enumeración útil, aunque más complejo, se llama `Flags` . Al *decorar* una enumeración con el atributo `Flags` , puede asignar una variable más de un valor a la vez. Tenga en cuenta que al hacer esto *debe* definir los valores explícitamente en la representación de base 2.

```
[Flags]
public enum MyEnum
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 4,
    Thursday = 8,
    Friday = 16,
    Saturday = 32,
    Sunday = 64
}
```

Ahora puede comparar más de un valor a la vez, ya sea utilizando *comparaciones a nivel de bits* o, si está utilizando .NET 4.0 o posterior, el método `Enum.HasFlag` .

```
MyEnum instance = MyEnum.Monday | MyEnum.Thursday; // instance now has a value of
```

```
                                                    // *both* Monday and Thursday,
                                                    // represented by (in binary) 0100.

if (instance.HasFlag(MyEnum.Wednesday))
{
    // it doesn't, so this block is skipped
}
else if (instance.HasFlag(MyEnum.Thursday))
{
    // it does, so this block is executed
}
```

Como la clase Enum se subclasifica de `System.ValueType` , se trata como un tipo de valor y se pasa por valor, no por referencia. El objeto base se crea en el montón, pero cuando pasa un valor de enumeración a una llamada de función, una copia del valor que usa el tipo de valor subyacente de Enum (generalmente System.Int32) se inserta en la pila. El compilador rastrea la asociación entre este valor y el objeto base que se creó en la pila. Consulte ValueType Class (System) (MSDN) para obtener más información.

Lea Tipos personalizados en línea: https://riptutorial.com/es/dot-net/topic/57/tipos-personalizados

Capítulo 56: Trabajar con SHA1 en C

Introducción

En este proyecto, verá cómo trabajar con la función de hash criptográfico SHA1. por ejemplo, obtener hash de la cadena y cómo romper el hash SHA1. fuente en git hub: https://github.com/mahdiabasi/SHA1Tool

Examples

#Generar suma de comprobación SHA1 de una función de archivo

Primero agregue System.Security.Cryptography y System.IO a su proyecto

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
            SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

Lea Trabajar con SHA1 en C # en línea: https://riptutorial.com/es/dot-net/topic/9457/trabajar-con-sha1-en-c-sharp

Capítulo 57: Trabajar con SHA1 en C

Introducción

En este proyecto, verá cómo trabajar con la función de hash criptográfico SHA1. por ejemplo, obtener hash de la cadena y cómo romper el hash SHA1.

compelete fuente en github: https://github.com/mahdiabasi/SHA1Tool

Examples

#Generar la suma de comprobación SHA1 de un archivo

Primero agregue el espacio de nombres System.Security.Cryptography a su proyecto

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
            SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

#Generar hash de un texto

```
  public static string TextToHash(string text)
    {
        var sh = SHA1.Create();
        var hash = new StringBuilder();
        byte[] bytes = Encoding.UTF8.GetBytes(text);
        byte[] b = sh.ComputeHash(bytes);
        foreach (byte a in b)
        {
            var h = a.ToString("x2");
            hash.Append(h);
        }
        return hash.ToString();
    }
```

Lea Trabajar con SHA1 en C # en línea: https://riptutorial.com/es/dot-net/topic/9458/trabajar-con-sha1-en-c-sharp

Capítulo 58: Usando el progreso e IProgress

Examples

Informe de progreso simple

`IProgress<T>` se puede usar para informar el progreso de algún procedimiento a otro procedimiento. Este ejemplo muestra cómo puede crear un método básico que informe de su progreso.

```
void Main()
{
    IProgress<int> p = new Progress<int>(progress =>
    {
        Console.WriteLine("Running Step: {0}", progress);
    });
    LongJob(p);
}

public void LongJob(IProgress<int> progress)
{
    var max = 10;
    for (int i = 0; i < max; i++)
    {
        progress.Report(i);
    }
}
```

Salida:

```
Running Step: 0
Running Step: 3
Running Step: 4
Running Step: 5
Running Step: 6
Running Step: 7
Running Step: 8
Running Step: 9
Running Step: 2
Running Step: 1
```

Tenga en cuenta que cuando ejecute este código, es posible que los números salgan desordenados. Esto se debe a que el `IProgress<T>.Report()` se ejecuta de forma asíncrona y, por lo tanto, no es tan adecuado para situaciones en las que el progreso debe informarse en orden.

Utilizando IProgress

Es importante tener en cuenta que la `System.Progress<T>` no tiene el método `Report()` disponible. Este método se implementó explícitamente desde la `IProgress<T>` y, por lo tanto, debe llamarse en un `Progress<T>` cuando se `IProgress<T>` un `IProgress<T>` .

```
var p1 = new Progress<int>();
p1.Report(1); //compiler error, Progress does not contain method 'Report'

IProgress<int> p2 = new Progress<int>();
p2.Report(2); //works

var p3 = new Progress<int>();
((IProgress<int>)p3).Report(3); //works
```

Lea Usando el progreso e IProgress en línea: https://riptutorial.com/es/dot-net/topic/5628/usando-el-progreso--t--e-iprogress--t-

Capítulo 59: XmlSerializer

Observaciones

No utilice el `XmlSerializer` para analizar `HTML` . Para esto, bibliotecas especiales están disponibles como el HTML Agility Pack.

Examples

Serializar objeto

```
public void SerializeFoo(string fileName, Foo foo)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.Open(fileName, FileMode.Create))
    {
        serializer.Serialize(stream, foo);
    }
}
```

Deserializar objeto

```
public Foo DeserializeFoo(string fileName)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.OpenRead(fileName))
    {
        return (Foo)serializer.Deserialize(stream);
    }
}
```

Comportamiento: Asignar nombre de elemento a propiedad

```
<Foo>
    <Dog/>
</Foo>
```

-

```
public class Foo
{
    // Using XmlElement
    [XmlElement(Name="Dog")]
    public Animal Cat { get; set; }
}
```

Comportamiento: asignar el nombre de la matriz a la propiedad (XmlArray)

```
<Store>
    <Articles>
        <Product/>
        <Product/>
    </Articles>
</Store>
```

-

```
public class Store
{
    [XmlArray("Articles")]
    public List<Product> Products {get; set; }
}
```

Formato: Formato de fecha y hora personalizado

```
public class Dog
{
    private const string _birthStringFormat = "yyyy-MM-dd";

    [XmlIgnore]
    public DateTime Birth {get; set;}

    [XmlElement(ElementName="Birth")]
    public string BirthString
    {
        get { return Birth.ToString(_birthStringFormat); }
        set { Birth = DateTime.ParseExact(value, _birthStringFormat,
CultureInfo.InvariantCulture); }
    }
}
```

Creación eficiente de varios serializadores con tipos derivados especificados dinámicamente

De donde venimos

A veces no podemos proporcionar todos los metadatos necesarios para el marco de XmlSerializer en atributo. Supongamos que tenemos una clase base de objetos serializados, y algunas de las clases derivadas son desconocidas para la clase base. No podemos colocar un atributo para todas las clases que no se conocen en el momento del diseño del tipo base. Podríamos tener otro equipo desarrollando algunas de las clases derivadas.

Qué podemos hacer

Podemos usar el `new XmlSerializer(type, knownTypes)` , pero eso sería una operación O (N ^ 2) para N serializadores, al menos para descubrir todos los tipos proporcionados en los argumentos:

```
// Beware of the N^2 in terms of the number of types.
var allSerializers = allTypes.Select(t => new XmlSerializer(t, allTypes));
```

```
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary (i => allTypes[i], i => allSerializers[i])
```

En este ejemplo, el tipo Base no tiene conocimiento de sus tipos derivados, lo cual es normal en OOP.

Haciéndolo eficientemente

Afortunadamente, hay un método que resuelve este problema en particular: suministrar tipos conocidos para varios serializadores de manera eficiente:

System.Xml.Serialization.XmlSerializer.FromTypes Method (Type [])

> El método FromTypes le permite crear de manera eficiente una matriz de objetos XmlSerializer para procesar una matriz de objetos Tipo.

```
var allSerializers = XmlSerializer.FromTypes(allTypes);
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary(i => allTypes[i], i => allSerializers[i]);
```

Aquí hay un ejemplo completo de código:

```
using System;
using System.Collections.Generic;
using System.Xml.Serialization;
using System.Linq;
using System.Linq;

public class Program
{
    public class Container
    {
        public Base Base { get; set; }
    }

    public class Base
    {
        public int JustSomePropInBase { get; set; }
    }

    public class Derived : Base
    {
        public int JustSomePropInDerived { get; set; }
    }

    public void Main()
    {
        var sampleObject = new Container { Base = new Derived() };
        var allTypes = new[] { typeof(Container), typeof(Base), typeof(Derived) };

        Console.WriteLine("Trying to serialize without a derived class metadata:");
        SetupSerializers(allTypes.Except(new[] { typeof(Derived) }).ToArray());
        try
        {
            Serialize(sampleObject);
```

```
        }
        catch (InvalidOperationException e)
        {
            Console.WriteLine();
            Console.WriteLine("This error was anticipated,");
            Console.WriteLine("we have not supplied a derived class.");
            Console.WriteLine(e);
        }
        Console.WriteLine("Now trying to serialize with all of the type information:");
        SetupSerializers(allTypes);
        Serialize(sampleObject);
        Console.WriteLine();
        Console.WriteLine("Slides down well this time!");
    }

    static void Serialize<T>(T o)
    {
        serializerDictionary[typeof(T)].Serialize(Console.Out, o);
    }

    private static Dictionary<Type, XmlSerializer> serializerDictionary;

    static void SetupSerializers(Type[] allTypes)
    {
        var allSerializers = XmlSerializer.FromTypes(allTypes);
        serializerDictionary = Enumerable.Range(0, allTypes.Length)
            .ToDictionary(i => allTypes[i], i => allSerializers[i]);
    }
}
```

Salida:

```
Trying to serialize without a derived class metadata:
<?xml version="1.0" encoding="utf-16"?>
<Container xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
This error was anticipated,
we have not supplied a derived class.
System.InvalidOperationException: There was an error generating the XML document. --->
System.InvalidOperationException: The type Program+Derived was not expected. Use the
XmlInclude or SoapInclude attribute to specify types that are not known statically.
   at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write2_Base(String
n, String ns, Base o, Boolean isNullable, Boolean needType)
   at
Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write3_Container(String
n, String ns, Container o, Boolean isNullable, Boolean needType)
   at
Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write4_Container(Object
o)
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id)
   --- End of inner exception stack trace ---
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id)
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle)
   at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces)
   at Program.Serialize[T](T o)
   at Program.Main()
```

```
Now trying to serialize with all of the type information:
<?xml version="1.0" encoding="utf-16"?>
<Container xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Base xsi:type="Derived">
    <JustSomePropInBase>0</JustSomePropInBase>
    <JustSomePropInDerived>0</JustSomePropInDerived>
  </Base>
</Container>
Slides down well this time!
```

¿Qué hay en la salida?

Este mensaje de error recomienda lo que intentamos evitar (o lo que no podemos hacer en algunos escenarios): hacer referencia a los tipos derivados de la clase base:

```
Use the XmlInclude or SoapInclude attribute to specify types that are not known statically.
```

Así es como obtenemos nuestra clase derivada en el XML:

```
<Base xsi:type="Derived">
```

`Base` corresponde al tipo de propiedad declarado en el tipo `Container` , y `Derived` es el tipo de la instancia realmente suministrada.

Aquí hay un ejemplo de trabajo.

Lea XmlSerializer en línea: https://riptutorial.com/es/dot-net/topic/31/xmlserializer

Creditos

S. No	Capítulos	Contributors
1	Empezando con .NET Framework	Adriano Repetti, Alan McBee, ale10ander, Andrew Jens, Andrew Morton, Andrey Shchekin, Community, Daniel A. White, Ehsan Sajjad, harriyott, hillary.fraley, Ian, James Thorpe, Jamie Rees, Joel Martinez, Kevin Montrose, Lirrik, MarcinJuraszek, matteeyah, naveen, Nicholas Sizer, Paweł Izdebski, Peter, Peter Gordon, Peter Hommel, PSN, Richard Lander, Rion Williams, Robert Columbia, RubberDuck, SeeuD1, Serg Rogovtsev, Squidward, Stephen Leppik, Steven Doggart, svick, ɬolɐəz əuɬ qoq
2	.NET Core	Mihail Stancescu
3	Acrónimo de glosario	Tanveer Badar
4	ADO.NET	Akshay Anand, Andrew Morton, Daniel A. White, DavidG, Drew, elmer007, Hamid, Harjot, Heinzi, Igor, user2321864
5	Ajustes	Alan McBee
6	Análisis DateTime	GalacticCowboy, John
7	Apilar y Montar	Hywel Rees
8	Arboles de expresion	Akshay Anand, George Polevoy, Jim, n.podbielski, Pavel Mayorov, RamenChef, Stephen Leppik, Stilgar, wangengzheng
9	Archivo de entrada / salida	ale10ander, Alexander Mandt, Ingenioushax, Nitram
10	Biblioteca paralela de tareas (TPL)	Adi Lester, Aman Sharma, Andrew, i3arnon, Jacobr365, JamyRyals, Konamiman, Mathias Müller, Mert Gülsoy, Mikhail Filimonov, Pavel Mayorov, Pavel Voronin, RamenChef, Thomas Bledsoe, TorbenJ
11	Cargar archivo y datos POST al servidor	Aleks Andreev

	web	
12	Clase System.IO.File	Adriano Repetti, delete me
13	Clientes HTTP	CodeCaster, Konamiman, MuiBienCarlota
14	CLR	Gajendra, starbeamrainbowlabs, Theodoros Chatzigiannakis
15	Colecciones	Alan McBee, Aman Sharma, Anik Saha, Daniel A. White, demonplus, Felipe Oriani, harriyott, Ian, Mark C., Ravi A., Virtlink
16	Compilador JIT	Krikor Ailanjian
17	Contextos de sincronización	DLeh, Gusdor
18	Contratos de código	JJS, Matthew Whited, RamenChef
19	Descripción general de la API de la biblioteca paralela de tareas (TPL)	Gusdor, Jacobr365
20	Diagnostico del sistema	Adi Lester, Bassie, Fredou, Ogglas, Ondřej Štorc, RamenChef
21	Encriptación / Criptografía	Alexander Mandt, Daniel A. White, demonplus, Jagadisha B S, lokusking, Matt
22	Enhebrado	Behzad, Martijn Pieters, Mellow
23	Escribir y leer desde StdErr stream	Aleks Andreev
24	Examen de la unidad	Axarydax
25	Excepciones	Adi Lester, Akshay Anand, Alan McBee, Alfred Myers, Arvin Baccay, BananaSft, CodeCaster, Dave R., Kritner, Mafii, Matt, Rob, Sean, starbeamrainbowlabs, STW, Yousef Al-Mulla
26	Expresiones regulares (System.Text.RegularExpressions)	BrunoLM, Denuath, Matt dc, tehDorf
27	Flujo de datos TPL	i3arnon, Jacobr365, Nikola.Lukovic, RamenChef
28	Formas VB	ale10ander, dbasnett
29	Gestión de la memoria	Big Fan, binki, DrewJordan
30	Globalización en ASP.NET MVC	Scott Hannen

	utilizando la internacionalización inteligente para ASP.NET	
31	Instrumentos de cuerda	Adriano Repetti, Alexander Mandt, Matt, Pavel Voronin, RamenChef
32	Invocación de plataforma	Dmitry Egorov, Imran Ali Khan
33	Inyección de dependencia	Phil Thomas, Scott Hannen
34	JSON en .NET con Newtonsoft.Json	DLeh
35	Leer y escribir archivos zip	Arxae
36	LINQ	A. Raza, Adil Mammadov, Akshay Anand, Alexander V., Benjamin Hodgson, Blachshma, Bradley Grainger, Bruno Garcia, Carlos Muñoz, CodeCaster, dbasnett, DoNot , dotctor, Eduardo Molteni, Ehsan Sajjad, GalacticCowboy, H. Pauwelyn, Haney, J3soon, jbtule, jnovo, Joe Amenta, Kilazur, Konamiman, MarcinJuraszek, Mark Hurd, McKay, Mellow, Mert Gülsoy, Mike Stortz, Mr.Mindor, Nate Barbettini, Pavel Voronin, Ruben Steins, Salvador Rubio Martinez, Sammi, Sergio Domínguez, Sidewinder94
37	Los diccionarios	Adriano Repetti, Bjørn-Roger Kringsjå, Daniel Plaisted, Darrel Lee, Felipe Oriani, George Duckett, George Polevoy, hatchet, Hogan, Ian, LegionMammal978, Luke Bearl, Olivier Jacot-Descombes, RamenChef, Ringil, Robert Columbia, Stephen Byrne, the berserker, Tomáš Hübelbauer
38	Marco de Extensibilidad Gestionado	Joe Amenta, Kirk Broadhurst, RamenChef
39	Para cada	Dr Rob Lang, just.ru, Lucas Trzesniewski
40	Procesamiento paralelo utilizando .Net framework	Yahfoufi
41	Proceso y ajuste de afinidad del hilo	MSE, RamenChef
42	Puertos seriales	Dmitry Egorov
43	ReadOnlyCollections	tehDorf
44	Recolección de basura	avat

45	Redes	Konamiman
46	Reflexión	Aleks Andreev, Bjørn-Roger Kringsjå, demonplus, Jean-Baptiste Noblot, Jigar, JJP , Kirk Broadhurst, Lorenzo Dematté, Matas Vaitkevicius, NetSquirrel, Pavel Mayorov, Peter, smdrager, Terry, user1304444, void
47	Serialización JSON	Akshay Anand, Andrius, Eric, hasan, M22an, PedroSouki, Thriggle, Tolga Evcimen
48	Servidores HTTP	Devon Burriss, Konamiman
49	Sistema de envasado NuGet	Andrey Shchekin, Anik Saha, Ashtonian, CodeCaster, Daniel A. White, Matas Vaitkevicius, Ozair Kafray
50	SpeechRecognitionEngine clase para reconocer el habla	ProgramFOX, RamenChef
51	System.IO	CodeCaster, Daniel A. White, demonplus, Filip Frącz, RoyalPotato
52	System.Net.Mail	demonplus, Steve, vicky
53	System.Reflection.Emit namespace	Luaan, NikolayKondratyev, RamenChef, toddmo
54	System.Runtime.Caching.MemoryCache (ObjectCache)	Guanxi, RamenChef
55	Tipos personalizados	Alan McBee, DrewJordan, matteeyah
56	Trabajar con SHA1 en C #	mahdi abasi
57	Usando el progreso e IProgress	DLeh
58	XmlSerializer	Aphelion, George Polevoy, RamenChef, Rowland Shaw, Thomas Levesque, void, Yogi