Бесплатная электронная книга

УЧУСЬ .NET Framework

Free unaffiliated eBook created from
Stack Overflow contributors.

#.net

Около

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

глава 1: Начало работы с .NET Framework

замечания

.NET Framework - это набор библиотек и среда выполнения, первоначально разработанная Microsoft. Все .NET-программы компилируются в байт-код под названием Microsoft Intermediate Language (MSIL). MSIL управляется Common Language Runtime (CLR).

Ниже вы можете найти несколько примеров «Hello World» на разных языках, поддерживающих .NET Framework. «Hello World» - это программа, которая отображает « Hello World» на устройстве отображения. Он используется для иллюстрации базового синтаксиса для построения рабочей программы. Его также можно использовать в качестве теста на здравомыслие, чтобы убедиться, что компилятор языка, среда разработки и среда выполнения работают правильно.

Список языков, поддерживаемых .NET.

Версии

.СЕТЬ

Версия	Дата выхода
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

Версия	Дата выхода
4.6.1	2015-11-17
4.6.2	2016-08-02
4,7	2017-04-05

Компактная структура

Версия	Дата выхода
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

Микроструктура

Версия	Дата выхода
4,2	2011-10-04
4,3	2012-12-04
4,4	2015-10-20

Examples

Привет, мир в 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` имеет несколько перегрузок. В этом случае строка «Hello World» является параметром, и во время выполнения она выведет «Hello World» в стандартный поток. Другие перегрузки могут вызывать `.ToString` аргумента перед записью в поток. Дополнительную информацию см. В документации по .NET Framework .

Живая демонстрация в действии на .NET Fiddle

Введение в C #

Hello World в Visual Basic .NET

```
Imports System

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

Живая демонстрация в действии на .NET Fiddle

Введение в Visual Basic .NET

Hello World in F

```
open System

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

Живая демонстрация в действии на .NET Fiddle

Введение в F #

Hello World в C ++ / CLI

```
using namespace System;

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

Привет, мир в PowerShell

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

Введение в PowerShell

Привет, мир в Nemerle

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

Привет, мир в 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.
```

Привет, мир в Бу

```
print "Hello World"
```

Hello World в Python (IronPython)

```
print "Hello World"
```

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

Привет мир в Ил

```
.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
  }

}
```

Прочитайте Начало работы с .NET Framework онлайн: https://riptutorial.com/ru/dot-net/topic/14/начало-работы-с--net-framework

глава 2: .NET Core

Вступление

.NET Core - платформа разработки общего назначения, поддерживаемая Microsoft и сообществом .NET на GitHub. Это кросс-платформенный, поддерживающий Windows, macOS и Linux, и может использоваться в сценариях устройств, облачных и встроенных / IoT.

Когда вы думаете о .NET Core, следует помнить следующее (гибкое развертывание, кросс-платформенные, средства командной строки, с открытым исходным кодом).

Другое замечательно, что даже если это открытый исходный код, Microsoft активно поддерживает его.

замечания

Сам по себе .NET Core включает в себя одну модель приложения - консольные приложения, которая полезна для инструментов, локальных служб и текстовых игр. Для расширения функциональности были созданы дополнительные модели приложений поверх .NET Core, такие как:

- Ядро ASP.NET
- Windows 10 Universal Windows Platform (UWP)
- Xamarin.Forms

Кроме того, .NET Core реализует стандартную библиотеку .NET и, следовательно, поддерживает стандартные библиотеки .NET.

Стандартная библиотека .NET - это спецификация API, которая описывает согласованный набор .NET API, которые разработчики могут ожидать в каждой реализации .NET. Реализация .NET должна реализовывать эту спецификацию, чтобы ее можно было считать совместимой с .NET Standard Library и поддерживать библиотеки, предназначенные для стандартной библиотеки .NET.

Examples

Базовое консольное приложение

```
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);
    }
}
```

Прочитайте .NET Core онлайн: https://riptutorial.com/ru/dot-net/topic/9059/-net-core

глава 3: ADO.NET

Вступление

ADO (объекты данных ActiveX) .Net - это инструмент, предоставляемый Microsoft, который обеспечивает доступ к источникам данных, таким как SQL Server, Oracle и XML через свои компоненты. .Net-приложения могут извлекать, создавать и обрабатывать данные, как только они подключены к источнику данных через ADO.Net с соответствующими привилегиями.

ADO.Net предоставляет архитектуру без подключения. Это безопасный подход к взаимодействию с базой данных, поскольку соединение не обязательно должно поддерживаться в течение всего сеанса.

замечания

Замечание о параметризации SQL с `Parameters.AddWithValue` `AddWithValue` : `AddWithValue` никогда не является хорошей отправной точкой. Этот метод основан на выводе типа данных из того, что передается. При этом вы можете оказаться в ситуации, когда преобразование не позволяет вашему запросу использовать индекс . Обратите внимание, что некоторые типы данных SQL Server, такие как `char` / `varchar` (без предшествующих «n») или `date` , не имеют соответствующего типа данных .NET. В таких случаях вместо этого следует использовать `Add` с правильным типом данных .

Examples

Выполнение операторов SQL как команды

```
// 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();
}
```

Примечание 1: см. Перечисление SqlDbType для варианта MSFT SQL Server.

Примечание 2: см. Перечисление MySqlDbType для варианта, специфичного для MySQL.

Лучшие практики - выполнение заявлений 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; }
}
```

Лучшая практика работы с ADO.NET

- Правило большого пальца - это открыть соединение для минимального времени. Закройте соединение явным образом после завершения процедуры. Это вернет объект подключения обратно в пул соединений. Пул соединений по умолчанию max size = 100. Поскольку объединение пулов повышает производительность физического подключения к SQL Server. Пул соединений в SQL Server
- Оберните все подключения к базе данных в используемом блоке, чтобы они всегда закрывались и удалялись даже в случае исключения. См. Использование Statement (C # Reference) для получения дополнительной информации об использовании операторов
- Извлеките строки подключения по имени из app.config или web.config (в зависимости от типа приложения)
    - Для этого требуется ссылка на сборку `System.configuration`
    - См. « Строки подключения и файлы конфигурации» для получения дополнительной информации о том, как структурировать файл конфигурации
- Всегда используйте параметры для входящих значений для
    - Избегайте атак sql injection
    - Избегайте ошибок, если используется неверный текст, например, включая одиночную кавычку, которая является эквивалентом sql для экранирования или запуска строки (varchar / nvarchar)
    - Предоставление поставщику базы данных повторного использования планов запросов (не поддерживаемых всеми поставщиками баз данных), что повышает эффективность
- При работе с параметрами
    - Тип параметра Sql и несоответствие размера являются общей причиной отказа вставки / обновления / выбора
    - Дайте параметры Sql значимым именам так же, как и переменные в коде.
    - Укажите тип данных базы данных используемого столбца, это гарантирует, что неправильные типы параметров не используются, что может привести к неожиданным результатам

    - Подтвердите свои входящие параметры, прежде чем передавать их в команду (как говорится, « мусор, мусор »). Проверять входящие значения как можно раньше в стеке
    - Используйте правильные типы при назначении значений параметров, например: не назначайте строковое значение DateTime, вместо этого присвойте фактическому экземпляру DateTime значение параметра
    - Укажите размер параметров строкового типа. Это связано с тем, что SQL Server может повторно использовать планы выполнения, если параметры соответствуют типу *и* размеру. Использовать -1 для MAX
    - Не используйте метод AddWithValue , основная причина заключается в том, что очень легко забыть указать тип параметра или точность / масштаб, когда это необходимо. Для получения дополнительной информации см. Можно ли уже остановить использование AddWithValue?
- При использовании соединений с базой данных
    - Откройте соединение как можно позже и закройте его как можно скорее. Это общее руководство при работе с любым внешним ресурсом
    - Никогда не обменивайтесь экземплярами подключения к базе данных (пример: наличие одного хост-узла является общим экземпляром типа `SqlConnection`). Если ваш код всегда создает новый экземпляр подключения базы данных, тогда необходимо, чтобы код вызова удалил его и «выбросил», когда это будет сделано. Причиной этого является
        - У большинства поставщиков баз данных есть своего рода объединение пулов, поэтому создание новых управляемых соединений дешево
        - Он устраняет любые будущие ошибки, если код начинает работать с несколькими потоками

Использование общих интерфейсов для абстрагирования отдельных классов поставщиков

```
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()) {
                ...
            }
        }
    }
```

```
}
```

Прочитайте ADO.NET онлайн: https://riptutorial.com/ru/dot-net/topic/3589/ado-net

глава 4: CLR

Examples

Введение в Common Language Runtime

Common Language Runtime (CLR) - это среда виртуальной машины и часть .NET Framework. Это содержит:

- Портативный язык байт-кода под названием **Common Intermediate Language** (сокращенный CIL или IL)
- Компилятор Just-In-Time, который генерирует машинный код
- Трассировочный сборщик мусора, который обеспечивает автоматическое управление памятью
- Поддержка легких подпроцессов под названием AppDomains
- Механизмы безопасности посредством концепций проверяемого кода и уровней доверия

Код, который выполняется в CLR, называется *управляемым кодом,* чтобы отличать его от кода, выполняемого вне CLR (обычно собственного кода), который называется *неуправляемым кодом* . Существуют различные механизмы, которые облегчают взаимодействие между управляемым и неуправляемым кодом.

Прочитайте CLR онлайн: https://riptutorial.com/ru/dot-net/topic/3942/clr

глава 5: HTTP-клиенты

замечания

В настоящее время соответствующие HTTP / 1.1 RFC:

- 7230: Синтаксис сообщений и маршрутизация
- 7231: семантика и содержание
- 7232: Условные запросы
- 7233: Запросы диапазона
- 7234: Кэширование
- 7235: Аутентификация
- 7239: перенаправленный HTTP-расширение
- 7240: Предпочитать заголовок для HTTP

Существуют также следующие информационные RFC:

- 7236: Регистрация схем аутентификации
- 7237: Регистрация методов

И экспериментальный RFC:

- 7238: Код статуса протокола гипертекстовой передачи 308 (Перманентный переадресация)

Связанные протоколы:

- 4918: HTTP-расширения для веб-распределенного создания и управления версиями (WebDAV)
- 4791: Календарные расширения для WebDAV (CalDAV)

Examples

Чтение ответа GET как строки с использованием 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();
}
```

Чтение ответа GET как строки с использованием System.Net.WebClient

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

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

Чтение ответа GET как строки с использованием System.Net.HttpClient

`HttpClient` доступен через NuGet: клиентские библиотеки Microsoft HTTP .

```
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;
    }
}
```

Отправка запроса POST с помощью полезной нагрузки строки с помощью 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();
```

Отправка запроса POST с помощью командной строки с использованием 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);
}
```

Отправка запроса POST с помощью полезной нагрузки строки с использованием System.Net.HttpClient

`HttpClient` доступен через NuGet: клиентские библиотеки Microsoft HTTP .

```
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();
```

Основной HTTP-загрузчик с использованием 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!");
    }
}
```

Прочитайте HTTP-клиенты онлайн: https://riptutorial.com/ru/dot-net/topic/32/http-клиенты

глава 6: HTTP-серверы

Examples

Основной файловый сервер HTTP только для чтения (HttpListener)

Заметки:

Этот пример должен быть запущен в административном режиме.

Поддерживается только один одновременный клиент.

Для простоты имена файлов считаются все ASCII (для части *имени файла* в заголовке *Content-Disposition*), и ошибки доступа к файлу не обрабатываются.

```
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);
    }
}
```

Основной файловый сервер HTTP только для чтения (ASP.NET Core)

1 - Создайте пустую папку, она будет содержать файлы, созданные на следующих шагах.

2 - Создайте файл с именем `project.json` со следующим содержимым (при необходимости отрегулируйте номер порта и `rootDirectory`):

```
{
  "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 - Создайте файл с именем `Startup.cs` со следующим кодом:

```
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 - Откройте командную строку, перейдите к папке и выполните:

```
dnvm use 1.0.0-rc1-final -r coreclr -p
dnu restore
```

Примечание. Эти команды нужно запускать только один раз. Используйте `dnvm list` для проверки фактического номера последней установленной версии базовой среды CLR.

5 - Запустите сервер с помощью: `dnx web` . Теперь файлы можно запрашивать по адресу `http://localhost:60000/path/to/file.ext` .

Для простоты имена файлов считаются все ASCII (для части имени файла в заголовке Content-Disposition), и ошибки доступа к файлу не обрабатываются.

Прочитайте HTTP-серверы онлайн: https://riptutorial.com/ru/dot-net/topic/53/http-серверы

глава 7: JIT-компилятор

Вступление

Компиляция JIT или компиляция «точно в срок» - это альтернативный подход к интерпретации кода или компиляции с опережением времени. Компиляция JIT используется в .NET framework. Код CLR (C #, F #, Visual Basic и т. Д.) Сначала скомпилирован в нечто, называемое Interpreted Language, или IL. Это код нижнего уровня, который ближе к машинным кодам, но не относится к платформе. Скорее, во время выполнения этот код компилируется в машинный код для соответствующей системы.

замечания

Зачем использовать компиляцию JIT?

- Лучшая совместимость: каждому языку CLR требуется только один компилятор для IL, и этот IL может работать на любой платформе, на которой он может быть преобразован в машинный код.
- Скорость: JIT-компиляция пытается совместить скорость работы скомпилированного кода в будущем, а также гибкость интерпретации (может анализировать код, который будет выполняться для потенциальной оптимизации перед компиляцией)

Страница Википедии для получения дополнительной информации о компиляции JIT в целом: https://en.wikipedia.org/wiki/Just-in-time_compilation

Examples

Пример компиляции IL

Простое приложение Hello World:

```
using System;

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

Эквивалентный код IL (который будет составлен 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
```

Сгенерировано с помощью инструмента MS ILDASM (IL дизассемблер)

Прочитайте JIT-компилятор онлайн: https://riptutorial.com/ru/dot-net/topic/9222/jit-компилятор

глава 8: JSON в .NET с Newtonsoft.Json

Вступление

Пакет NuGet `Newtonsoft.Json` стал стандартом defacto для использования и обработки форматированного текста и объектов JSON в .NET. Это надежный инструмент, быстрый и простой в использовании.

Examples

Сериализовать объект в JSON

```
using Newtonsoft.Json;

var obj = new Person
{
    Name = "Joe Smith",
    Age = 21
};
var serializedJson = JsonConvert.SerializeObject(obj);
```

Это приводит к этому JSON: `{"Name":"Joe Smith","Age":21}`

Удаление десериализации объекта из текста JSON

```
var json = "{\"Name\":\"Joe Smith\",\"Age\":21}";
var person = JsonConvert.DeserializeObject<Person>(json);
```

Это дает объект `Person` с именем «Joe Smith» и Age 21.

Прочитайте JSON в .NET с Newtonsoft.Json онлайн: https://riptutorial.com/ru/dot-net/topic/8746/json-в--net-с-newtonsoft-json

глава 9: LINQ

Вступление

LINQ (Language Integrated Query) - это выражение, которое извлекает данные из источника данных. LINQ упрощает эту ситуацию, предлагая согласованную модель для работы с данными в различных источниках и форматах данных. В запросе LINQ вы всегда работаете с объектами. Вы используете одни и те же базовые шаблоны кодирования для запроса и преобразования данных в документы XML, базы данных SQL, наборы данных ADO.NET, коллекции .NET и любой другой формат, доступный для провайдера. LINQ может использоваться в C # и VB.

Синтаксис

- public static TSource Aggregate <TSource> (этот источник IEnumerable <TSource>, Func <TSource, TSource, TSource> func)
- public static TAccumulate Aggregate <TSource, TAccumulate> (этот источник IEnumerable <TSource>, TAccumulate seed, Func <TAccumulate, TSource, TAccumulate> func)
- public static TResult Aggregate <TSource, TAccumulate, TResult> (этот источник IEnumerable <TSource>, TAccumulate seed, Func <TAccumulate, TSource, TAccumulate> func, Func <TAccumulate, TResult> resultSelector)
- public static Boolean Все <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)
- public static Boolean Любой <TSource> (этот источник IEnumerable <TSource>)
- public static Boolean Любой <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)
- public static IEnumerable <TSource> AsEnumerable <TSource> (этот источник IEnumerable <TSource>)
- public static Decimal Average (этот IEnumerable <десятичный> источник)
- public static Double Average (этот источник IEnumerable <Double>)
- public static Double Average (этот источник IEnumerable <Int32>)
- public static Double Average (этот источник IEnumerable <Int64>)
- public static Nullable <Decimal> Average (этот IEnumerable <Nullable <Decimal >> source)
- public static Nullable <Double> Среднее (этот IEnumerable <Nullable <Double >> source)
- public static Nullable <Double> Средний (этот IEnumerable <Nullable <Int32 >> source)
- public static Nullable <Double> Средний (этот IEnumerable <Nullable <Int64 >> source)
- public static Nullable <Single> Average (этот IEnumerable <Nullable <Single >> source)
- public static Single Average (этот IEnumerable <Single> source)
- public static Decimal Average <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Decimal>)
- public static Double Average <TSource> (этот источник IEnumerable <TSource>,

селектор Func <TSource, Double>)

- public static Double Average <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Int32>)
- public static Double Average <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Int64>)
- public static Nullable <Decimal> Average <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Decimal >> selector)
- public static Nullable <Double> Average <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Double >> selector)
- public static Nullable <Double> Средний <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Int32 >> selector)
- public static Nullable <Double> Average <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Nullable <Int64 >>)
- public static Nullable <Single> Average <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Single >> selector))
- public static Single Average <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Single>)
- public static IEnumerable <TResult> Cast <TResult> (этот источник IEnumerable)
- public static IEnumerable <TSource> Concat <TSource> (этот IEnumerable <TSource> сначала, IEnumerable <TSource> второй)
- public static Boolean Содержит <TSource> (этот источник IEnumerable <TSource>, значение TSource)
- public static Boolean Содержит <TSource> (этот источник IEnumerable <TSource>, значение TSource, сравнитель IEqualityComparer <TSource>)
- public static Int32 Count <TSource> (этот источник IEnumerable <TSource>)
- public static Int32 Count <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)
- public static IEnumerable <TSource> DefaultIfEmpty <TSource> (этот источник IEnumerable <TSource>)
- public static IEnumerable <TSource> DefaultIfEmpty <TSource> (этот источник IEnumerable <TSource>, TSource defaultValue)
- public static IEnumerable <TSource> Distinct <TSource> (этот источник IEnumerable <TSource>)
- public static IEnumerable <TSource> Distinct <TSource> (этот источник IEnumerable <TSource>, сравнительный файл IEqualityComparer <TSource>)
- public static TSource ElementAt <TSource> (этот источник IEnumerable <TSource>, индекс Int32)
- public static TSource ElementAtOrDefault <TSource> (этот источник IEnumerable <TSource>, индекс Int32)
- public static IEnumerable <TResult> Пусто <TResult> ()
- public static IEnumerable <TSource> За исключением <TSource> (этот IEnumerable <TSource> сначала, IEnumerable <TSource> второй)
- public static IEnumerable <TSource> За исключением <TSource> (этот IEnumerable

<TSource> сначала, IEnumerable <TSource> второй, IEqualityComparer <TSource> Comparer)

- public static TSource Первый <TSource> (этот источник IEnumerable <TSource>)
- public static TSource First <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)
- public static TSource FirstOrDefault <TSource> (этот источник IEnumerable <TSource>)
- public static TSource FirstOrDefault <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)
- public static IEnumerable <IGrouping <TKey, TSource >> GroupBy <TSource, TKey> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static IEnumerable <IGrouping <TKey, TSource >> GroupBy <TSource, TKey> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, IEqualityComparer <TKey> Comparer)
- public static IEnumerable <IGrouping <TKey, TElement >> GroupBy <TSource, TKey, TElement> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- публичный статический IEnumerable <IGrouping <TKey, TElement >> GroupBy <TSource, TKey, TElement> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> Comparer)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TResult> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TKey, IEnumerable <TSource>, TResult> resultSelector)
- public static IEnumerable <TResult> GroupBy <TSource, TKey, TResult> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TKey, IEnumerable <TSource>, TResult> resultSelector, IEqualityComparer <TKey> Comparer)
- публичный статический IEnumerable <TResult> GroupBy <TSource, TKey, TElement, TResult> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, Func <TKey, IEnumerable <TElement>, TResult > resultSelector)
- публичный статический IEnumerable <TResult> GroupBy <TSource, TKey, TElement, TResult> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, Func <TKey, IEnumerable <TElement>, TResult > resultSelector, IEqualityComparer <TKey> Comparer)
- публичный статический IEnumerable <TResult> GroupJoin <TOuter, TInner, TKey, TResult> (этот IEnumerable <TOuter> внешний, IEnumerable <TInner> внутренний, Func <TOuter, TKey> внешнийKeySelector, Func <TInner, TKey> innerKeySelector, Func <TOuter, IEnumerable <TInner>, TResult> resultSelector)
- публичный статический IEnumerable <TResult> GroupJoin <TOuter, TInner, TKey, TResult> (этот IEnumerable <TOuter> внешний, IEnumerable <TInner> внутренний, Func <TOuter, TKey> внешнийKeySelector, Func <TInner, TKey> innerKeySelector, Func <TOuter, IEnumerable <TInner>, TResult> resultSelector, IEqualityComparer <TKey> Comparer)
- public static IEnumerable <TSource> Intersect <TSource> (этот IEnumerable <TSource>

сначала, IEnumerable <TSource> второй)

- public static IEnumerable <TSource> Intersect <TSource> (этот IEnumerable <TSource> сначала, IEnumerable <TSource> второй, IEqualityComparer <TSource> Comparer)
- публичный статический IEnumerable <TResult> Join <TOuter, TInner, TKey, TResult> (этот IEnumerable <TOuter> внешний, IEnumerable <TInner> внутренний, Func <TOuter, TKey> внешнийKeySelector, Func <TInner, TKey> innerKeySelector, Func <TOuter, TInner, TResult> resultSelector)
- публичный статический IEnumerable <TResult> Join <TOuter, TInner, TKey, TResult> (этот IEnumerable <TOuter> внешний, IEnumerable <TInner> внутренний, Func <TOuter, TKey> внешнийKeySelector, Func <TInner, TKey> innerKeySelector, Func <TOuter, TInner, TResult> resultSelector, IEqualityComparer <TKey> Comparer)
- public static TSource Last <TSource> (этот источник IEnumerable <TSource>)
- public static TSource Last <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)
- public static TSource LastOrDefault <TSource> (этот источник IEnumerable <TSource>)
- public static TSource LastOrDefault <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)
- public static Int64 LongCount <TSource> (этот источник IEnumerable <TSource>)
- public static Int64 LongCount <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)
- public static Decimal Max (этот источник IEnumerable <Decimal>)
- public static Double Max (этот источник IEnumerable <Double>)
- public static Int32 Max (этот источник IEnumerable <Int32>)
- public static Int64 Max (этот источник IEnumerable <Int64>)
- public static Nullable <Decimal> Max (этот IEnumerable <Nullable <Decimal >> source)
- public static Nullable <Double> Max (этот IEnumerable <Nullable <Double >> источник)
- public static Nullable <Int32> Max (этот IEnumerable <Nullable <Int32 >> source)
- public static Nullable <Int64> Max (этот IEnumerable <Nullable <Int64 >> источник)
- public static Nullable <Single> Max (этот IEnumerable <Nullable <Single >> source)
- public static Single Max (этот IEnumerable <Single> source)
- public static TSource Max <TSource> (этот источник IEnumerable <TSource>)
- public static Decimal Max <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Decimal>)
- public static Double Max <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Double>)
- public static Int32 Max <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Int32>)
- public static Int64 Max <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Int64>)
- public static Nullable <Decimal> Max <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Decimal >> selector)
- public static Nullable <Double> Max <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Double >> selector)

- public static Nullable <Int32> Max <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Nullable <Int32 >>)
- public static Nullable <Int64> Max <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Nullable <Int64 >>)
- public static Nullable <Single> Max <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Single >> selector))
- public static Single Max <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Single>)
- public static TResult Max <TSource, TResult> (этот источник IEnumerable <TSource>, селектор Func <TSource, TResult>)
- public static Decimal Min (этот IEnumerable <Decimal> source)
- public static Double Min (этот источник IEnumerable <Double>)
- public static Int32 Min (этот источник IEnumerable <Int32>)
- public static Int64 Min (этот источник IEnumerable <Int64>)
- public static Nullable <Decimal> Min (этот IEnumerable <Nullable <Decimal >> source)
- public static Nullable <Double> Min (этот IEnumerable <Nullable <Double >> source)
- public static Nullable <Int32> Min (этот IEnumerable <Nullable <Int32 >> source)
- public static Nullable <Int64> Min (этот IEnumerable <Nullable <Int64 >> source)
- public static Nullable <Single> Min (этот IEnumerable <Nullable <Single >> source)
- public static Single Min (этот IEnumerable <Single> source)
- public static TSource Min <TSource> (этот источник IEnumerable <TSource>)
- public static Decimal Min <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, десятичный>)
- public static Double Min <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Double>)
- public static Int32 Min <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Int32>)
- public static Int64 Min <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Int64>)
- public static Nullable <Decimal> Min <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Decimal >> selector)
- public static Nullable <Double> Min <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Double >> selector)
- public static Nullable <Int32> Min <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Nullable <Int32 >>)
- public static Nullable <Int64> Min <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Nullable <Int64 >>)
- public static Nullable <Single> Min <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Single >> selector))
- public static Single Min <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Single>)
- public static TResult Min <TSource, TResult> (этот источник IEnumerable <TSource>, селектор Func <TSource, TResult>)

- public static IEnumerable <TResult> OfType <TResult> (этот источник IEnumerable)
- public static IOrderedEnumerable <TSource> OrderBy <TSource, TKey> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> OrderBy <TSource, TKey> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> Comparer)
- public static IOrderedEnumerable <TSource> OrderByDescending <TSource, TKey> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> OrderByDescending <TSource, TKey> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> Comparer)
- public static IEnumerable <Int32> Range (Int32 start, Int32 count)
- public static IEnumerable <TResult> Repeat <TResult> (элемент TResult, число Int32)
- public static IEnumerable <TSource> Обратный <TSource> (этот источник IEnumerable <TSource>)
- public static IEnumerable <TResult> Выберите <TSource, TResult> (этот источник IEnumerable <TSource>, селектор Func <TSource, TResult>)
- public static IEnumerable <TResult> Выберите <TSource, TResult> (этот источник IEnumerable <TSource>, селектор Func <TSource, Int32, TResult>)
- public static IEnumerable <TResult> SelectMany <TSource, TResult> (этот источник IEnumerable <TSource>, селектор Func <TSource, IEnumerable <TResult >>)
- public static IEnumerable <TResult> SelectMany <TSource, TResult> (этот источник IEnumerable <TSource>, Func <TSource, Int32, IEnumerable <селектор TResult >>)
- public static IEnumerable <TResult> SelectMany <TSource, TCollection, TResult> (этот источник IEnumerable <TSource>, Func <TSource, IEnumerable <TCollection >> collectionSelector, Func <TSource, TCollection, TResult> resultSelector)
- public static IEnumerable <TResult> SelectMany <TSource, TCollection, TResult> (этот источник IEnumerable <TSource>, Func <TSource, Int32, IEnumerable <TCollection >> collectionSelector, Func <TSource, TCollection, TResult> resultSelector)
- public static Boolean SequenceEqual <TSource> (этот IEnumerable <TSource> сначала, IEnumerable <TSource> второй)
- public static Boolean SequenceEqual <TSource> (этот IEnumerable <TSource> сначала, IEnumerable <TSource> второй, IEqualityComparer <TSource> Comparer)
- public static TSource Single <TSource> (этот источник IEnumerable <TSource>)
- public static TSource Single <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)
- public static TSource SingleOrDefault <TSource> (этот источник IEnumerable <TSource>)
- public static TSource SingleOrDefault <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)
- public static IEnumerable <TSource> Пропустить <TSource> (этот источник IEnumerable <TSource>, количество Int32)
- public static IEnumerable <TSource> SkipWhile <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)

- public static IEnumerable <TSource> SkipWhile <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Int32, Boolean>)
- public static Decimal Sum (этот IEnumerable <десятичный> источник)
- public static Double Sum (этот IEnumerable <Double> source)
- public static Int32 Sum (этот источник IEnumerable <Int32>)
- public static Int64 Sum (этот источник IEnumerable <Int64>)
- public static Nullable <Decimal> Sum (этот IEnumerable <Nullable <Decimal >> source)
- public static Nullable <Double> Sum (этот IEnumerable <Nullable <Double >> источник)
- public static Nullable <Int32> Sum (этот IEnumerable <Nullable <Int32 >> source)
- public static Nullable <Int64> Sum (этот IEnumerable <Nullable <Int64 >> источник)
- public static Nullable <Single> Sum (этот IEnumerable <Nullable <Single >> source)
- public static Single Sum (этот IEnumerable <Single> source)
- public static Decimal Sum <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Decimal>)
- public static Double Sum <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Double>)
- public static Int32 Sum <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Int32>)
- public static Int64 Sum <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Int64>)
- public static Nullable <Decimal> Sum <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Decimal >> selector)
- public static Nullable <Double> Sum <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Double >> selector)
- public static Nullable <Int32> Sum <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Nullable <Int32 >>)
- public static Nullable <Int64> Sum <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Int64 >> селектор)
- public static Nullable <Single> Sum <TSource> (этот источник IEnumerable <TSource>, Func <TSource, Nullable <Single >> selector)
- public static Single Sum <TSource> (этот источник IEnumerable <TSource>, селектор Func <TSource, Single>)
- public static IEnumerable <TSource> Возьмите <TSource> (этот источник IEnumerable <TSource>, количество Int32)
- public static IEnumerable <TSource> TakeWhile <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)
- public static IEnumerable <TSource> TakeWhile <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Int32, Boolean>)
- public static IOrderedEnumerable <TSource> ThenBy <TSource, TKey> (этот источник IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> ThenBy <TSource, TKey> (этот источник IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> Comparer)

- public static IOrderedEnumerable <TSource> ThenByDescending <TSource, TKey> (этот источник IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static IOrderedEnumerable <TSource> ThenByDescending <TSource, TKey> (этот источник IOrderedEnumerable <TSource>, Func <TSource, TKey> keySelector, IComparer <TKey> comparer)
- public static TSource [] ToArray <TSource> (этот источник IEnumerable <TSource>)
- public static Dictionary <TKey, TSource> ToDictionary <TSource, TKey> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- публичный статический словарь <TKey, TSource> ToDictionary <TSource, TKey> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, IEqualityComparer <TKey> Comparer)
- публичный статический словарь <TKey, TElement> ToDictionary <TSource, TKey, TElement> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- публичный статический словарь <TKey, TElement> ToDictionary <TSource, TKey, TElement> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> Comparer)
- public static List <TSource> ToList <TSource> (этот источник IEnumerable <TSource>)
- public static ILookup <TKey, TSource> ToLookup <TSource, TKey> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector)
- public static ILookup <TKey, TSource> ToLookup <TSource, TKey> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, IEqualityComparer <TKey> Comparer)
- public static ILookup <TKey, TElement> ToLookup <TSource, TKey, TElement> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector)
- публичный статический ILookup <TKey, TElement> ToLookup <TSource, TKey, TElement> (этот источник IEnumerable <TSource>, Func <TSource, TKey> keySelector, Func <TSource, TElement> elementSelector, IEqualityComparer <TKey> comparer)
- public static IEnumerable <TSource> Union <TSource> (этот IEnumerable <TSource> сначала, IEnumerable <TSource> второй)
- public static IEnumerable <TSource> Union <TSource> (этот IEnumerable <TSource> во-первых, IEnumerable <TSource> второй, IEqualityComparer <TSource> Comparer)
- public static IEnumerable <TSource> Где <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Boolean>)
- public static IEnumerable <TSource> Где <TSource> (этот источник IEnumerable <TSource>, предикат Func <TSource, Int32, Boolean>)
- public static IEnumerable <TResult> Zip <TFirst, TSecond, TResult> (этот IEnumerable <TFirst> сначала, IEnumerable <TSecond> второй, Func <TFirst, TSecond, TResult> resultSelector)

замечания

- См. Также LINQ .

Встроенные методы LINQ - это методы расширения для интерфейса `IEnumerable<T>` которые живут в классе `System.Linq.Enumerable` в сборке `System.Core` . Они доступны в .NET Framework 3.5 и более поздних версиях.

LINQ позволяет просто изменять, преобразовывать и комбинировать различные `IEnumerable` с использованием запроса или функционального синтаксиса.

Хотя стандартные методы LINQ могут работать с любым `IEnumerable<T>` , включая простые массивы и `List<T>` s, их также можно использовать для объектов базы данных, где множество выражений LINQ во многих случаях может быть преобразовано в SQL, если объект данных поддерживает его. См. LINQ to SQL .

Для методов, которые сравнивают объекты (такие как `Contains` и `Except`), `IEquatable<T>.Equals` используется, если тип T коллекции реализует этот интерфейс. В противном случае используются стандартные `Equals` и `GetHashCode` типа (возможно, переопределенные из реализаций `Object` по умолчанию). Существуют также перегрузки для этих методов, которые позволяют указать пользовательский `IEqualityComparer<T>` .

Для методов `...OrDefault` `default(T)` используется для создания значений по умолчанию.

Официальная ссылка: Перечислимый класс

Ленивая оценка

Практически каждый запрос, возвращающий `IEnumerable<T>` , сразу не оценивается; вместо этого логика задерживается до тех пор, пока запрос не будет повторен. Одним из следствий является то, что каждый раз, когда кто-то выполняет итерацию над `IEnumerable<T>` созданным с помощью одного из этих запросов, например, `.Where()` , повторяется полная логика запроса. Если предикат длительный, это может быть причиной проблем с производительностью.

Одно простое решение (когда вы знаете или можете контролировать приблизительный размер результирующей последовательности) состоит в том, чтобы полностью `.ToArray()` результаты с использованием `.ToArray()` или `.ToList()` . `.ToDictionary()` или `.ToLookup()` могут выполнять ту же роль. Можно также, конечно, перебрать всю последовательность и буферизировать элементы в соответствии с другой пользовательской логикой.

`ToArray()` или `ToList()` ?

Оба `.ToArray()` и `.ToList()` все элементы последовательности `IEnumerable<T>` и сохраняют результаты в коллекции, хранящейся в памяти. Используйте следующие рекомендации, чтобы определить, какой из них выбрать:

- Для некоторых API может потребоваться `T[]` или `List<T>` .
- `.ToList()` обычно работает быстрее и генерирует меньше мусора, чем `.ToArray()` , потому что последний должен копировать все элементы в новую коллекцию фиксированного размера еще раз, чем первый, почти в каждом случае.
- Элементы могут быть добавлены или удалены из `List<T>` возвращаемого `.ToList()` , тогда как `T[]` возвращаемый из `.ToArray()` остается фиксированным размером на протяжении всего его жизненного `.ToArray()` . Другими словами, `List<T>` является изменяемым, а `T[]` является неизменным.
- `T[]` возвращаемый из `.ToArray()` использует меньше памяти, чем `List<T>` возвращаемый из `.ToList()` , поэтому, если результат будет храниться в течение длительного времени, предпочитайте `.ToArray()` . Calling `List<T>.TrimExcess()` сделает разницу в памяти строго академической ценой за счет исключения относительного преимущества скорости `.ToList()` .

Examples

Выбрать (карта)

```
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
```

Этот тип функции обычно называется `map` в языках функционального программирования.

Где (фильтр)

Этот метод возвращает IEnumerable со всеми элементами, которые соответствуют выражению лямбда

пример

```
var personNames = new[]
{
    "Foo", "Bar", "Fizz", "Buzz"
};

var namesStartingWithF = personNames.Where(p => p.StartsWith("F"));
Console.WriteLine(string.Join(",", namesStartingWithF));
```

Выход:

Foo, Fizz

Посмотреть демо

Сортировать по

```
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
```

Содержит

```
var numbers = new[] {1,2,3,4,5};
Console.WriteLine(numbers.Contains(3)); //True
Console.WriteLine(numbers.Contains(34)); //False
```

Кроме

```
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
```

пересекаться

```
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
```

Обратите внимание, что дубликаты сохраняются в результате. Если это нежелательно, используйте `Union` .

Сначала (найти)

```
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
```

Следующие `InvalidOperationException` с сообщением «Последовательность не содержит соответствующего элемента»:

```
var firstNegativeNumber = numbers.First(n => n < 0);
```

не замужем

```
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
```

Следующий `InvalidOperationException` вызывает `InvalidOperationException` поскольку в последовательности содержится более одного элемента:

```
var theOnlyNumberInNumbers = numbers.Single();
var theOnlyNegativeNumber = numbers.Single(n => n < 0);
```

Прошлой

```
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
```

Следующие `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
```

Следующие `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
```

любой

Возвращает `true` если в коллекции есть элементы, которые удовлетворяют условию в выражении лямбда:

```
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
```

Все

```
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
```

Обратите внимание, что функция « `All` работает, проверяя, что первый элемент оценивается как `false` соответствии с предикатом. Следовательно, метод вернет `true` для *любого* предиката в случае, когда множество пусто:

```
var numbers = new int[0];
var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

SelectMany (плоская карта)

`Enumerable.Select` возвращает выходной элемент для каждого элемента ввода. В то время как `Enumerable.SelectMany` производит переменное количество выходных элементов для каждого элемента ввода. Это означает, что выходная последовательность может содержать больше или меньше элементов, чем во входной последовательности.

`Lambda expressions` передаются в `Enumerable.Select` должны возвращать один элемент. Лямбда-выражения передаются в `Enumerable.SelectMany` должны генерировать дочернюю последовательность. Эта дочерняя последовательность может содержать различное количество элементов для каждого элемента во входной последовательности.

пример

```
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()));
```

Выход:

1,2,3,4,5,6

Посмотреть демо

`Enumerable.SelectMany` также может быть достигнут с помощью синтаксического запроса, используя два последовательных `from` предложений:

```
var allInvoicesFromAllCustomers
    = from customer in customers
      from invoice in customer.Invoices
      select invoice;
```

сумма

```
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
```

Пропускать

Skip будет перечислять первые N элементов, не возвращая их. Как только номер позиции N

+ 1 достигнут, Skip начинает возвращать каждый перечислимый элемент:

```
var numbers = new[] {1,2,3,4,5};

var allNumbersExceptFirstTwo = numbers.Skip(2);
Console.WriteLine(string.Join(",", allNumbersExceptFirstTwo.ToArray()));

//3,4,5
```

принимать

Этот метод берет первые `n` элементов из перечислимого.

```
var numbers = new[] {1,2,3,4,5};

var threeFirstNumbers = numbers.Take(3);
Console.WriteLine(string.Join(",", threeFirstNumbers.ToArray()));

//1,2,3
```

SequenceEqual

```
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
```

Задний ход

```
var numbers = new[] {1,2,3,4,5};
var reversed = numbers.Reverse();

Console.WriteLine(string.Join(",", reversed.ToArray()));

//5,4,3,2,1
```

OfType

```
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
```

Максимум

```
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
```

Средний

```
var numbers = new[] {1,2,3,4};

var averageNumber = numbers.Average();
Console.WriteLine(averageNumber);
// 2,5
```

Этот метод вычисляет среднее число перечислимых чисел.

```
var cities = new[] {
    new {Population = 1000},
    new {Population = 2000},
    new {Population = 4000}
};

var averagePopulation = cities.Average(c => c.Population);
Console.WriteLine(averagePopulation);
// 2333,33
```

Этот метод вычисляет среднее значение перечислимого с помощью делегированной функции.

застежка-молния

.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
```

отчетливый

```
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
```

Группа по

```
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
```

Групповые счета по странам, генерирующие новый объект с количеством записей, общей оплатой и средней оплатой

```
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) });
```

Если мы хотим только итоговые суммы, ни одна группа

```
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) });
```

Если нам нужно несколько подсчетов

```
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) });
```

ToDictionary

Возвращает новый словарь из исходного `IEnumerable` используя предоставленную функцию keySelector для определения ключей. Выбросит `ArgumentException` если keySelector не является инъективным (возвращает уникальное значение для каждого члена исходной коллекции). Существуют перегрузки, которые позволяют указывать значение, которое нужно сохранить, а также ключ.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
};
```

Указание только функции выбора ключа создаст `Dictionary<TKey,TVal>` с `TKey` return Тип селектора клавиш, `TVal` - исходный тип объекта, а исходный объект - как сохраненное значение.

```
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
```

Указание функции селектора значений также приведет к созданию `Dictionary<TKey,TVal>` когда `TKey` прежнему является типом возврата селектора клавиш, но `TVal` теперь возвращает тип функции выбора значения и возвращаемое значение в качестве хранимого значения.

```
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
```

Как указано выше, ключи, возвращаемые селектором ключей, должны быть уникальными. Следующее выдает исключение.

```
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)
```

Если уникальный уникальный ключ не может быть предоставлен для исходной коллекции, попробуйте вместо этого использовать ToLookup. На первый взгляд, ToLookup ведет себя аналогично ToDictionary, однако в полученном Lookup каждый ключ сопряжен с набором значений с соответствующими ключами.

союз

```
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
```

Обратите внимание, что дубликаты удаляются из результата. Если это нежелательно, используйте `Concat` вместо этого.

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[]
```

К списку

```
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
```

подсчитывать

```
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
```

ElementAt

```
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
```

Совокупный (складной)

Создание нового объекта на каждом шаге:

```
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,
```

Использование одного и того же объекта на всех этапах:

```
var commaSeparatedElements2 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"));

Console.WriteLine(commaSeparatedElements2.ToString());  //1,2,3,4,5,
```

Использование селектора результатов:

```
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,
```

Если семя опущено, первым элементом становится семя:

```
var seedAndElements = elements.Select(n=>n.ToString());
var commaSeparatedElements4 = seedAndElements.Aggregate(
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements4);  //12,3,4,5,
```

ToLookup

```
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
```

Присоединиться

```
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
```

GroupJoin

```
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
```

В ролях

`Cast` отличается от других методов `Enumerable` тем, что это метод расширения для `IEnumerable` , а не для `IEnumerable<T>` . Таким образом, его можно использовать для преобразования экземпляров первого в экземпляры позже.

Это не компилируется, поскольку `ArrayList` не реализует `IEnumerable<T>` :

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.First());
```

Это работает так, как ожидалось:

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.Cast<int>().First()); //1
```

`Cast` **не** выполняет преобразования. Следующие компиляции, но `InvalidCastException` во время выполнения:

```
var numbers = new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Cast<decimal>().ToArray();
```

Правильный способ преобразования конвертирования в коллекцию выглядит следующим образом:

```
var numbers= new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Select(n => (decimal)n).ToArray();
```

пустой

Чтобы создать пустой IEnumerable из int:

```
IEnumerable<int> emptyList = Enumerable.Empty<int>();
```

Этот пустой IEnumerable кэшируется для каждого типа T, так что:

```
Enumerable.Empty<decimal>() == Enumerable.Empty<decimal>(); // This is True
Enumerable.Empty<int>() == Enumerable.Empty<decimal>();     // This is False
```

ThenBy

`ThenBy` может использоваться только после предложения `OrderBy` , позволяющего заказать несколько критериев

```
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
```

Спектр

Два параметра `Range` - это *первое* число и *количество* элементов для создания (а не последнее число).

```
// 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)));
```

Левая внешняя связь

```
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}");
}
```

Повторение

`Enumerable.Repeat` генерирует последовательность повторяющегося значения. В этом примере он генерирует «Привет» 4 раза.

```
var repeats = Enumerable.Repeat("Hello", 4);

foreach (var item in repeats)
{
    Console.WriteLine(item);
}
```

```
/* output:
    Hello
    Hello
    Hello
    Hello
*/
```

Прочитайте LINQ онлайн: https://riptutorial.com/ru/dot-net/topic/34/linq

глава 10: ReadOnlyCollections

замечания

`ReadOnlyCollection` предоставляет `ReadOnlyCollection` для чтения к существующей коллекции («коллекция источников»).

Элементы не добавляются непосредственно или удаляются из `ReadOnlyCollection` . Вместо этого они добавляются и удаляются из исходной коллекции, и `ReadOnlyCollection` будет отражать эти изменения в источнике.

Число и порядок элементов внутри `ReadOnlyCollection` не могут быть изменены, но свойства элементов могут быть и методы могут быть вызваны, если предположить, что они находятся в области видимости.

Используйте `ReadOnlyCollection` если вы хотите разрешить внешнему коду просматривать свою коллекцию, не изменяя ее, но все же можете самостоятельно модифицировать коллекцию.

Смотрите также

- `ObservableCollection<T>`
- `ReadOnlyObservableCollection<T>`

ReadOnlyCollections vs ImmutableCollection

`ReadOnlyCollection` отличается от `ImmutableCollection` тем, что вы не можете редактировать `ImmutableCollection` после его создания - он всегда будет содержать `n` элементов, и их нельзя заменить или переупорядочить. С другой стороны, `ReadOnlyCollection` нельзя редактировать напрямую, но элементы все равно могут быть добавлены / удалены / переупорядочены с использованием исходной коллекции.

Examples

Создание ReadOnlyCollection

Использование конструктора

`ReadOnlyCollection` создается путем передачи существующего объекта `IList` в конструктор:

```
var groceryList = new List<string> { "Apple", "Banana" };
var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);
```

Использование LINQ

Дополнительно, LINQ предоставляет метод расширения `AsReadOnly()` для объектов `IList` :

```
var readOnlyVersion = groceryList.AsReadOnly();
```

Заметка

Как правило, вы хотите сохранить исходную коллекцию конфиденциально и разрешить публичный доступ к `ReadOnlyCollection` . Хотя вы можете создать `ReadOnlyCollection` из встроенного списка, вы не сможете изменить коллекцию после ее создания.

```
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!
```

Если вы это сделаете, вы можете захотеть использовать другую структуру данных, такую как `ImmutableCollection` .

Обновление ReadOnlyCollection

`ReadOnlyCollection` нельзя редактировать напрямую. Вместо этого исходная коллекция обновляется, и `ReadOnlyCollection` будет отражать эти изменения. Это ключевая особенность `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"
```

Посмотреть демо

Предупреждение. Элементы в ReadOnlyCollection не являются неотъемлемо доступными для чтения

Если исходная коллекция имеет тип, который не является неизменным, элементы, доступные через `ReadOnlyCollection` могут быть изменены.

```
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);
}
```

Посмотреть демо

Прочитайте ReadOnlyCollections онлайн: https://riptutorial.com/ru/dot-net/topic/6906/readonlycollections

глава 11: System.Diagnostics

Examples

Секундомер

В этом примере показано, как `Stopwatch` можно использовать для сравнения блока кода.

```
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");
        }
    }
}
```

Запуск команд оболочки

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

Это нужно, чтобы скрыть окно 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();
```

Отправить команду CMD и получить выход

Этот метод позволяет отправить `command` на `Cmd.exe` и возвращает стандартный вывод (включая стандартную ошибку) в виде строки:

```
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;
}
```

использование

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

Выход

> 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 Активен. 25/07 / 2016 08:33 Andy.McAnderson ica-cgp # 2 4 Актив. 25/07/2016 08:54 John.Smith ica-cgp # 4 5 Актив 14 25/07/2016 08:57 Bob.Bobbington ica-cgp # 5 6 Active 24692 + 13: 29 25/07/2016 09:05 Tim.Tom ica-cgp # 6 7 Активен. 25/07/2016 09:08 Bob.Joges ica-cgp # 7 8 Актив 24692 + 13: 29 25 / 07/2016 09:13 "

В некоторых случаях доступ к указанному серверу может быть ограничен определенными пользователями. Если у вас есть учетные данные для этого пользователя, можно отправить запросы с помощью этого метода:

```
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;
}
```

Получение пароля:

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

    return ss;
}
```

Заметки

Оба вышеуказанных метода возвращают конкатенацию STDOUT и STDERR, так как `OutputDataReceived` и `ErrorDataReceived` присоединяются к одной и той же переменной - `cmdOut` .

Прочитайте System.Diagnostics онлайн: https://riptutorial.com/ru/dot-net/topic/3143/system-diagnostics

глава 12: System.IO

Examples

Чтение текстового файла с помощью StreamReader

```
string fullOrRelativePath = "testfile.txt";

string fileData;

using (var reader = new StreamReader(fullOrRelativePath))
{
    fileData = reader.ReadToEnd();
}
```

Обратите внимание, что эта `StreamReader` конструктора `StreamReader` делает некоторое обнаружение автоматического кодирования , которое может или не может соответствовать фактической кодировке, используемой в файле.

Обратите внимание, что есть некоторые удобные методы, которые читают весь текст из файла, доступного в классе `System.IO.File` , а именно `File.ReadAllText(path)` и `File.ReadAllLines(path)` .

Чтение / запись данных с использованием System.IO.File

Сначала давайте посмотрим три разных способа извлечения данных из файла.

```
string fileText = File.ReadAllText(file);
string[] fileLines = File.ReadAllLines(file);
byte[] fileBytes = File.ReadAllBytes(file);
```

- В первой строке мы считываем все данные в файле в виде строки.
- Во второй строке мы считываем данные в файле в массив строк. Каждая строка в файле становится элементом в массиве.
- На третьем мы читаем байты из файла.

Затем давайте посмотрим три разных метода **добавления** данных в файл. Если указанный вами файл не существует, каждый метод автоматически создаст файл, прежде чем пытаться добавить к нему данные.

```
 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.");
}
```

- В первой строке мы просто добавляем строку в конец указанного файла.
- Во второй строке мы добавляем каждый элемент массива в новую строку в файле.
- Наконец, в третьей строке мы используем `File.AppendText` чтобы открыть потоковый блок, который добавит все данные, которые будут записаны на него.

И, наконец, давайте посмотрим три разных способа **записи** данных в файл. Разница между *добавлением* и *записью* заключается в том, что запись **поверх записи** данных в файл при добавлении **добавляет** данные в файл. Если указанный вами файл не существует, каждый метод автоматически создаст файл, прежде чем пытаться записать данные на него.

```
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 });
```

- Первая строка записывает строку в файл.
- Вторая строка записывает каждую строку в массив в своей собственной строке в файле.
- Третья строка позволяет вам написать массив байтов в файл.

Последовательные порты, использующие System.IO.SerialPorts

Итерация по подключенным последовательным портам

```
using System.IO.Ports;
string[] ports = SerialPort.GetPortNames();
for (int i = 0; i < ports.Length; i++)
{
    Console.WriteLine(ports[i]);
}
```

Создание объекта System.IO.SerialPort

```
using System.IO.Ports;
SerialPort port = new SerialPort();
SerialPort port = new SerialPort("COM 1"); ;
SerialPort port = new SerialPort("COM 1", 9600);
```

ПРИМЕЧАНИЕ . Это всего лишь три из семи перегрузок конструктора для типа SerialPort.

Чтение / запись данных через SerialPort

Самый простой способ - использовать методы `SerialPort.Read` и `SerialPort.Write` . Однако вы также можете получить объект `System.IO.Stream` который можно использовать для потоковой передачи данных через SerialPort. Для этого используйте `SerialPort.BaseStream` .

чтение

```
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);
```

Вы также можете прочитать все доступные данные:

```
string curData = port.ReadExisting();
```

Или просто прочитайте первую новую строку, встречающуюся во входящих данных:

```
string line = port.ReadLine();
```

Пишу

Самый простой способ записи данных через SerialPort:

```
port.Write("here is some text to be sent over the serial port.");
```

Однако вы также можете отправлять данные по мере необходимости:

```
//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);
```

Прочитайте System.IO онлайн: https://riptutorial.com/ru/dot-net/topic/5259/system-io

глава 13: System.Net.Mail

замечания

Важно Dispose System.Net.MailMessage, потому что каждое отдельное приложение содержит Stream, и эти потоки должны быть освобождены как можно скорее. Оператор using гарантирует, что объект Disposable Disposed также в случае Исключения

Examples

MailMessage

Вот пример создания почтового сообщения с вложениями. После создания мы отправляем это сообщение с помощью класса `SmtpClient` . Здесь используется порт 25 по умолчанию.

```
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;
        }
    }
}
```

Почта с приложением

`MailMessage` представляет почтовое сообщение, которое может быть отправлено далее с использованием класса `SmtpClient` . В почтовое сообщение можно добавить несколько вложений (файлов).

```
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

}
```

Прочитайте System.Net.Mail онлайн: https://riptutorial.com/ru/dot-net/topic/7440/system-net-mail

глава 14: System.Runtime.Caching.MemoryCache (ObjectCache)

Examples

Добавление элемента в кеш (Set)

Функция set вставляет запись кэша в кеш с помощью экземпляра CacheItem для предоставления ключа и значения для записи в кэш.

Эта функция переопределяет `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)

Эта функция получает существующий кеш формы элемента, и если элемент не существует в кеше, он будет извлекать элемент на основе функции 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);
        }
    }
```

Прочитайте System.Runtime.Caching.MemoryCache (ObjectCache) онлайн:
https://riptutorial.com/ru/dot-net/topic/76/system-runtime-caching-memorycache--objectcache-

глава 15: XmlSerializer

замечания

Не используйте `XmlSerializer` для анализа `HTML` . Для этого доступны специальные библиотеки, такие как HTML Agility Pack

Examples

Сериализовать объект

```
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);
    }
}
```

Дезамеризовать объект

```
public Foo DeserializeFoo(string fileName)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.OpenRead(fileName))
    {
        return (Foo)serializer.Deserialize(stream);
    }
}
```

Поведение: имя элемента карты для свойства

```
<Foo>
    <Dog/>
</Foo>
```

-

```
public class Foo
{
    // Using XmlElement
    [XmlElement(Name="Dog")]
    public Animal Cat { get; set; }
}
```

Поведение: имя массива массива для свойства (XmlArray)

```
<Store>
    <Articles>
        <Product/>
        <Product/>
    </Articles>
</Store>
```

-

```
public class Store
{
    [XmlArray("Articles")]
    public List<Product> Products {get; set; }
}
```

Форматирование: пользовательский формат DateTime

```
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); }
    }
}
```

Эффективное создание нескольких сериализаторов с производными типами, заданными динамически

Откуда мы пришли

Иногда мы не можем предоставить все необходимые метаданные, необходимые для инфраструктуры XmlSerializer в атрибуте. Предположим, что у нас есть базовый класс сериализованных объектов, а некоторые из производных классов неизвестны базовому классу. Мы не можем поместить атрибут для всех классов, которые не известны во время разработки базового типа. У нас может быть другая команда, разрабатывающая некоторые из производных классов.

Что мы можем сделать

Мы можем использовать `new XmlSerializer(type, knownTypes)` , но это будет операция O (N ^ 2) для N сериализаторов, по крайней мере, для обнаружения всех типов, представленных в

аргументах:

```
// 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])
```

В этом примере базовый тип не знает о его производных типах, что является нормальным в ООП.

Эффективное выполнение

К счастью, существует метод, который решает эту конкретную проблему - эффективно использовать известные типы для нескольких сериализаторов:

Метод System.Xml.Serialization.XmlSerializer.FromTypes (Тип [])

> Метод FromTypes позволяет эффективно создавать массив объектов XmlSerializer для обработки массива объектов Type.

```
var allSerializers = XmlSerializer.FromTypes(allTypes);
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary(i => allTypes[i], i => allSerializers[i]);
```

Вот полный пример кода:

```
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]);
    }
}
```

Выход:

```
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!
```

Что в выходе

Это сообщение об ошибке рекомендует, что мы пытались избежать (или то, что мы не можем сделать в некоторых сценариях) - ссылки на производные типы из базового класса:

```
Use the XmlInclude or SoapInclude attribute to specify types that are not known statically.
```

Вот как мы получаем наш производный класс в XML:

```
<Base xsi:type="Derived">
```

`Base` соответствует типу свойства, объявленному в типе `Container` , и `Derived` является типом экземпляра, фактически поставленного.

Вот рабочий пример скрипки

Прочитайте XmlSerializer онлайн: https://riptutorial.com/ru/dot-net/topic/31/xmlserializer

глава 16: Ввод / вывод файлов

параметры

параметр	подробности
строка пути	Путь файла для проверки. (относительный или полностью квалифицированный)

замечания

Возвращает true, если файл существует, в противном случае - false.

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");
        }
    }
}
```

Может также использоваться в тройном операторе.

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

Прочитайте Ввод / вывод файлов онлайн: https://riptutorial.com/ru/dot-net/topic/1376/ввод---вывод-файлов

глава 17: Внедрение зависимости

замечания

Проблемы, решаемые путем инъекций зависимостей

Если бы мы не использовали инъекцию зависимостей, класс `Greeter` мог бы выглядеть примерно так:

```
public class ControlFreakGreeter
{
    public void Greet()
    {
        var greetingProvider = new SqlGreetingProvider(
            ConfigurationManager.ConnectionStrings["myConnectionString"].ConnectionString);
        var greeting = greetingProvider.GetGreeting();
        Console.WriteLine(greeting);
    }
}
```

Это «контрольный урод», потому что он контролирует создание класса, предоставляющего приветствие, он контролирует, откуда приходит строка подключения SQL, и он контролирует вывод.

Используя инъекцию зависимостей, класс `Greeter` отказывается от этих обязанностей в пользу единой ответственности, написав приветствие, предоставленное ему.

Принцип инверсии зависимостей предполагает, что классы должны зависеть от абстракций (например, интерфейсов), а не от других конкретных классов. Прямые зависимости (сцепление) между классами могут затруднить техническое обслуживание. В зависимости от абстракций можно уменьшить эту связь.

Инъекционная инъекция помогает нам достичь этой инверсии зависимостей, поскольку она приводит к написанию классов, которые зависят от абстракций. Класс `Greeter` «ничего не знает» о деталях реализации `IGreetingProvider` и `IGreetingWriter` . Он знает только, что введенные зависимости реализуют эти интерфейсы. Это означает, что изменения в конкретных классах, которые реализуют `IGreetingProvider` и `IGreetingWriter` , не будут влиять на `Greeter` . Они не заменят их совершенно разными реализациями. Только изменения в интерфейсах будут. `Greeter` развязан.

`ControlFreakGreeter` невозможно правильно протестировать. Мы хотим протестировать одну небольшую единицу кода, но вместо этого наш тест будет включать подключение к SQL и выполнение хранимой процедуры. Это также будет включать тестирование вывода консоли. Поскольку ControlFreakGreeter делает так много, невозможно проверить изолированно от других классов.

`Greeter` легко тестируется на единицу, потому что мы можем вводить в заблуждение реализацию зависимостей, которые легче выполнять и проверять, чем вызов хранимой процедуры или чтение выходных данных консоли. Он не требует строки подключения в app.config.

Конкретные реализации `IGreetingProvider` и `IGreetingWriter` могут стать более сложными. У них, в свою очередь, могут быть свои собственные зависимости, которые вводятся в них. (Например, мы бы `SqlGreetingProvider` строку подключения SQL в `SqlGreetingProvider` .) Но эта сложность «скрыта» от других классов, которые зависят только от интерфейсов. Это облегчает изменение одного класса без «эффекта пульсации», который требует от нас внести соответствующие изменения в другие классы.

Examples

Инъекция зависимостей - простой пример

Этот класс называется `Greeter` . Его обязанность - выпустить приветствие. Он имеет две *зависимости* . Ему нужно что-то, что даст ему приветствие для вывода, а затем ему нужен способ вывода этого приветствия. Эти зависимости описываются как интерфейсы, `IGreetingProvider` и `IGreetingWriter` . В этом примере эти две зависимости «вводятся» в `Greeter` . (Дальнейшее объяснение по примеру.)

```
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);
}
```

Класс `Greeting` зависит как от `IGreetingProvider` и от `IGreetingWriter` , но он не несет

ответственности за создание экземпляров. Вместо этого он требует их в своем конструкторе. Все, что создает экземпляр `Greeting` должно обеспечивать эти две зависимости. Мы можем назвать это «инъекцией» зависимостей.

Потому что зависимости предоставляются классу в его конструкторе, это также называется «инъекцией конструктора».

Несколько общих соглашений:

- Конструктор сохраняет зависимости как `private` поля. Как только экземпляр класса создается, эти зависимости доступны для всех других нестатических методов класса.
- `private` поля - только для `readonly` . Как только они установлены в конструкторе, они не могут быть изменены. Это означает, что эти поля не должны (и не могут) быть изменены вне конструктора. Это гарантирует, что эти зависимости будут доступны для жизни класса.
- Зависимости - это интерфейсы. Это не является строго необходимым, но является общим, потому что упрощает замену одной реализации зависимости другой. Он также позволяет предоставлять издеваемую версию интерфейса для целей модульного тестирования.

Как инъекция зависимостей упрощает тестирование единиц

Это основывается на предыдущем примере класса `Greeter` который имеет две зависимости: `IGreetingProvider` и `IGreetingWriter` .

Фактическая реализация `IGreetingProvider` может извлекать строку из вызова API или базы данных. Реализация `IGreetingWriter` может отображать приветствие в консоли. Но поскольку `Greeter` имеет свои зависимости, введенные в его конструктор, легко написать единичный тест, который вводит издевательские версии этих интерфейсов. В реальной жизни мы могли бы использовать фреймворк вроде Moq , но в этом случае я напишу эти издевательства.

```
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);
    }
}
```

Поведение `IGreetingProvider` и `IGreetingWriter` не относится к этому тесту. Мы хотим проверить, что `Greeter` получает приветствие и записывает его. Конструкция `Greeter` (с использованием инъекции зависимостей) позволяет вводить насмешливые зависимости без каких-либо сложных движущихся частей. Все, что мы тестируем, это то, что `Greeter` взаимодействует с этими зависимостями, как мы ожидаем.

Почему мы используем контейнеры для инъекций зависимостей (контейнеры IoC)

Инъекция зависимостей означает написание классов, чтобы они не контролировали свои зависимости - вместо этого их зависимости предоставляются («впрыскивается»).

Это не то же самое, что использование рамки инъекции зависимостей (часто называемой «контейнером DI», «контейнером IoC» или просто «контейнером»), например, Castle Windsor, Autofac, SimpleInjector, Ninject, Unity или другими.

Контейнер просто упрощает инъекцию зависимостей. Например, предположим, что вы пишете несколько классов, которые полагаются на инъекцию зависимостей. Один класс зависит от нескольких интерфейсов, классы, которые реализуют эти интерфейсы, зависят от других интерфейсов и т. Д. Некоторые зависят от конкретных значений. И только для удовольствия некоторые из этих классов реализуют `IDisposable` и должны быть удалены.

Каждый отдельный класс хорошо написан и легко тестируется. Но теперь есть другая проблема: создание экземпляра класса стало намного сложнее. Предположим, мы создаем экземпляр класса `CustomerService` . Он имеет зависимости, а зависимости зависят. Построение экземпляра может выглядеть примерно так:

```
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);
        }
    }
}
```

Вы можете удивиться, почему бы не поставить целую гигантскую конструкцию в отдельную функцию, которая просто возвращает `CustomerService` ? Одна из причин заключается в том, что из-за того, что в него вводятся зависимости для каждого класса, класс не несет ответственности за знание того, являются ли эти зависимости `IDisposable` или их удаление. Он просто использует их. Поэтому, если у нас была `GetCustomerService()` которая вернула полностью сконструированную `CustomerService` , этот класс может содержать несколько одноразовых ресурсов и не иметь доступа к ним или распоряжаться ими.

И помимо утилизации `IDisposable` , кто хочет вызвать серию вложенных конструкторов, подобных этому, когда-либо? Это короткий пример. Это может стать намного хуже. Опять же, это не значит, что мы неправильно написали классы. Классы могут быть индивидуально идеальными. Задача состоит в их объединении.

Контейнер инъекции зависимостей упрощает это. Это позволяет нам указать, какой класс или значение следует использовать для выполнения каждой зависимости. В этом слегка упрощенном примере используется 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"))
);
```

Мы называем это «регистрирующими зависимостями» или «настройкой контейнера». В переводе это говорит нашему `WindsorContainer` :

- Если для класса требуется `ILogWriter` , создайте экземпляр `LogWriter` . `LogWriter` требуется путь к файлу. Используйте это значение из `AppSettings` .

- Если для класса требуется `IAuthorizationRepository` , создайте экземпляр `SqlAuthorizationRepository` . Для этого требуется строка подключения. Используйте это значение в разделе `ConnectionStrings` .
- Если для класса требуется `ICustomerDataProvider` , создайте `CustomerApiClient` и `AppSettings` строку из `AppSettings` .

Когда мы запрашиваем зависимость от контейнера, мы называем это «разрешающей» зависимость. Это плохая практика, чтобы сделать это непосредственно с помощью контейнера, но это совсем другая история. Для демонстрационных целей мы могли бы теперь сделать это:

```
var customerService = container.Resolve<CustomerService>();
var data = customerService.GetCustomerData(customerNumber);
container.Release(customerService);
```

Контейнер знает, что `CustomerService` зависит от `IAuthorizationRepository` и `ICustomerDataProvider` . Он знает, какие классы необходимо создать для выполнения этих требований. Эти классы, в свою очередь, имеют больше зависимостей, и контейнер знает, как их выполнить. Он создаст каждый класс, в котором он нуждается, пока не сможет вернуть экземпляр `CustomerService` .

Если он `IDoesSomethingElse` точки, в которой класс требует зависимости, которую мы не зарегистрировали, например, `IDoesSomethingElse` , тогда, когда мы пытаемся разрешить `CustomerService` он выдаст явное исключение, сообщив нам, что мы ничего не зарегистрировали для выполнения этого требования.

Каждая структура DI ведет себя несколько иначе, но обычно они дают нам некоторый контроль над тем, как создаются определенные классы. Например, хотим ли мы создать один экземпляр `LogWriter` и предоставить его каждому классу, который зависит от `ILogWriter` , или мы хотим, чтобы он каждый раз создавал новый? В большинстве контейнеров есть способ указать это.

Что относительно классов, которые реализуют `IDisposable` ? Вот почему мы называем `container.Release(customerService);` в конце. Большинство контейнеров (включая Windsor) будут отступать от всех созданных зависимостей и `Dispose` те, которые нуждаются в утилизации. Если `CustomerService` является `IDisposable` он тоже удалит это.

Регистрация зависимостей, как видно выше, может выглядеть как больше кода для записи. Но когда у нас много классов с большим количеством зависимостей, тогда это действительно окупается. И если бы нам пришлось писать те же классы *без* использования инъекции зависимостей, то такое же приложение с большим количеством классов стало бы трудно поддерживать и тестировать.

Это царапины поверхности, *почему* мы используем контейнеры для инъекций зависимостей. *Как* мы настроим наше приложение для использования одного (и правильно

его используем), это не только одна тема - это несколько тем, так как инструкции и примеры варьируются от одного контейнера к другому.

Прочитайте Внедрение зависимости онлайн: https://riptutorial.com/ru/dot-net/topic/5085/внедрение-зависимости

глава 18: Вывоз мусора

Вступление

В .Net объекты, созданные с помощью new (), выделяются в управляемой куче. Эти объекты никогда не будут окончательно завершены программой, которая их использует; вместо этого этот процесс контролируется сборщиком мусора .Net.

Некоторые из приведенных ниже примеров - это «лабораторные случаи», показывающие сборщик мусора на работе и некоторые существенные детали его поведения, в то время как другие сосредоточены на том, как подготовить классы для правильной обработки Сборщиком мусора.

замечания

Сборщик мусора нацелен на снижение стоимости программы с точки зрения выделенной памяти, но при этом имеет стоимость с точки зрения времени обработки. Чтобы достичь хорошего общего компромисса, существует ряд оптимизаций, которые следует учитывать при программировании с помощью сборщика мусора:

- Если метод Collect () должен быть явно вызван (что не так часто бывает так), рассмотрите использование «оптимизированного» режима, который завершает работу с мертвым объектом только тогда, когда на самом деле нужна память
- Вместо вызова метода Collect () используйте методы AddMemoryPressure () и RemoveMemoryPressure (), которые запускают коллекцию памяти только в случае необходимости
- Сбор памяти не гарантируется, чтобы завершить все мертвые объекты; вместо этого сборщик мусора управляет 3 «поколениями», объект иногда «выживает» от поколения к следующему
- В зависимости от различных факторов, включая настройку тонкой настройки, могут применяться несколько моделей потоков, что приводит к разным степеням помех между резьбой коллектора мусора и другими прикладными нитями (нитями)

Examples

Основной пример сбора (мусора)

Учитывая следующий класс:

```
public class FinalizableObject
{
    public FinalizableObject()
```

```
    {
        Console.WriteLine("Instance initialized");
    }

    ~FinalizableObject()
    {
        Console.WriteLine("Instance finalized");
    }
}
```

Программа, которая создает экземпляр, даже не используя его:

```
new FinalizableObject(); // Object instantiated, ready to be used
```

Производит следующий вывод:

```
<namespace>.FinalizableObject initialized
```

Если ничего не происходит, объект не дорабатывается до окончания программы (что освобождает все объекты в управляемой куче, завершая их в процессе).

Можно заставить сборщик мусора работать в данной точке, как показано ниже:

```
new FinalizableObject(); // Object instantiated, ready to be used
GC.Collect();
```

Что дает следующий результат:

```
<namespace>.FinalizableObject initialized
<namespace>.FinalizableObject finalized
```

На этот раз, как только был вызван сборщик мусора, неиспользуемый (он же «мертвый») объект был доработан и освобожден от управляемой кучи.

Живые объекты и мертвые объекты - основы

Правило: когда происходит сбор мусора, «живые объекты» - это те, которые все еще используются, тогда как «мертвые объекты» - это те, которые больше не используются (любая переменная или поле, ссылающееся на них, если таковые имеются, вышло за рамки до того, как произойдет сбор) ,

В следующем примере (для удобства FinalizableObject1 и FinalizableObject2 являются подклассами FinalizableObject из приведенного выше примера и, таким образом, наследуют поведение сообщения инициализации / завершения):

```
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();
```

Выход будет:

```
<namespace>.FinalizableObject1 initialized
<namespace>.FinalizableObject2 initialized
<namespace>.FinalizableObject1 finalized
```

В то время, когда вызывается сборщик мусора, FinalizableObject1 является мертвым объектом и завершается, а FinalizableObject2 - это живой объект и хранится в управляемой куче.

Несколько мертвых объектов

Что, если два (или несколько) иначе мертвых объекта ссылаются друг на друга? Это показано в примере ниже, если предположить, что OtherObject является общедоступным свойством 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();
```

Это дает следующий результат:

```
<namespace>.FinalizedObject1 initialized
<namespace>.FinalizedObject2 initialized
<namespace>.FinalizedObject1 finalized
<namespace>.FinalizedObject2 finalized
```

Эти два объекта завершены и освобождены от управляемой кучи, несмотря на то, что они ссылаются друг на друга (поскольку никакая другая ссылка не существует для любого из них из реального объекта).

Слабые ссылки

Слабые ссылки - это ссылки на другие объекты (иначе называемые «цели»), но «слабые», поскольку они не мешают собирать мусор. Другими словами, слабые ссылки не учитываются, когда сборщик мусора оценивает объекты как «живые» или «мертвые».

Следующий код:

```
var weak = new WeakReference<FinalizableObject>(new FinalizableObject());
GC.Collect();
```

Производит вывод:

```
<namespace>.FinalizableObject initialized
<namespace>.FinalizableObject finalized
```

Объект освобождается от управляемой кучи, несмотря на то, что ссылается на переменную WeakReference (все еще в области, когда был вызван сборщик мусора).

Следствие №1: в любое время небезопасно предполагать, что цель WeakReference по-прежнему распределяется по управляемой куче или нет.

Следствие №2: всякий раз, когда программе необходимо получить доступ к объекту Weakreference, код должен быть предоставлен для обоих случаев, а цель еще выделена или нет. Метод доступа к целевому объекту - 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
}
```

Общая версия WeakReference доступна с .Net 4.5. Все версии фреймворка предоставляют нестандартную, нетипизированную версию, которая построена таким же образом и проверяется следующим образом:

```
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
}
```

Утилизировать () против финализаторов

Внедрите метод Dispose () (и объявите содержащийся класс как IDisposable) в качестве средства обеспечения освобождения ресурсов памяти, как только объект больше не будет использоваться. «Уловка» заключается в том, что нет надежной гарантии того, что метод Dispose () будет когда-либо вызываться (в отличие от финализаторов, которые всегда вызываются в конце жизни объекта).

Один из сценариев - это программа, вызывающая Dispose () для объектов, которые она явно создает:

```
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
}
```

Другой сценарий - объявить класс, который будет создан в рамках структуры. В этом случае новый класс обычно наследует базовый класс, например, в MVC создается класс контроллера в качестве подкласса System.Web.Mvc.ControllerBase. Когда базовый класс реализует интерфейс IDisposable, это хороший намек на то, что утилита Dispose () будет правильно использована инфраструктурой, но опять же нет сильной гарантии.

Таким образом Dispose () не является заменой финализатора; вместо этого они должны использоваться для разных целей:

- Финализатор в конечном итоге освобождает ресурсы, чтобы избежать утечек памяти, которые могли бы произойти иначе
- Dispose () освобождает ресурсы (возможно, те же самые), как только они больше не нужны, чтобы уменьшить давление на общее распределение памяти.

Надлежащее удаление и завершение объектов

Поскольку Dispose () и финализаторы нацелены на разные цели, класс, управляющий ресурсами с большой памятью, должен реализовать оба из них. Следствием является запись класса, чтобы он хорошо справлялся с двумя возможными сценариями:

- Когда вызывается только финализатор

- Когда Dispose () вызывается сначала и позже, вызывается также финализатор

Одно из решений записывает код очистки таким образом, что запуск его один или два раза приведет к тому же результату, что и его запуск только один раз. Технико-экономическое обоснование зависит от характера очистки, например:

- Закрытие уже закрытого соединения с базой данных, вероятно, не будет иметь никакого эффекта, поэтому оно будет работать
- Обновление некоторого «количества использования» опасно и приведет к неправильному результату при вызове дважды, а не в один раз.

Более безопасное решение обеспечивает по дизайну, что код очистки вызывается один раз и только один раз, независимо от внешнего контекста. Это может быть достигнуто «классическим способом» с использованием выделенного флага:

```
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;
        }
    }
}
```

В качестве альтернативы сборщик мусора предоставляет специальный метод SuppressFinalize (), который позволяет пропустить финализатор после вызова Dispose:

```
public class DisposableFinalizable2 : IDisposable
{
    ~DisposableFinalizable2() { Cleanup(); }

    public void Dispose()
    {
        Cleanup();
        GC.SuppressFinalize(this);
    }

    private void Cleanup()
    {
        // Actual code to release resources gets here
    }
}
```

Прочитайте Вывоз мусора онлайн: https://riptutorial.com/ru/dot-net/topic/9636/вывоз-мусора

глава 19: Вызов платформы

Синтаксис

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

Examples

Вызов функции 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);
    }
}
```

Объявить функцию как `static extern` stting `DllImportAttribute` с свойством `Value` установленным в .dll name. Не забудьте использовать пространство имен `System.Runtime.InteropServices` . Затем назовите его как обычный статический метод.

Службы Invocation Services позаботятся о загрузке .dll и поиске нужного результата. P / Invoke в большинстве простых случаев также будет маршалировать параметры и возвращать значение в .dll и обратно. Т.е. преобразовать из .NET типов данных в Win32 и наоборот.

Использование Windows API

Используйте pinvoke.net .

Прежде чем объявлять `extern` функцию Windows API в вашем коде, попробуйте найти его на pinvoke.net . Они, скорее всего, уже имеют подходящую декларацию со всеми поддерживающими типами и хорошими примерами.

Маршевые массивы

Массивы простого типа

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

Массивы строк

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

Структуры маршалинга

Простая структура

Подпись C ++:

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

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

Определение 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 с неизвестными размерами массива. Передача

Подпись C ++

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

void SetVector(VECTOR &vector);
```

При переходе от управляемого к неуправляемому коду это

Массив `data` должен быть определен как IntPtr, и память должна быть явно выделена `Marshal.AllocHGlobal()` (и освобождена с помощью `Marshal.FreeHGlobal()` послесловия):

```
[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 с неизвестными размерами массива. получающий

Подпись C ++:

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

bool GetCurrentUser(USER *user);
```

Когда такие данные передаются из неуправляемого кода, а память выделяется неуправляемыми функциями, управляемый вызывающий `IntPrt` должен получать его в переменную `IntPrt` и преобразовывать буфер в управляемый массив. В случае строк есть удобный `Marshal.PtrToStringAnsi()` :

```
[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);
```

Маршалинские союзы

Только поля типа значения

Декларация C ++

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

Объявление C #

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

Смешивание значений и ссылочных полей

Перекрытие ссылочного значения одним типом значения не допускается, поэтому вы не можете просто использовать ~~`FieldOffset(0) text; FieldOffset(0) i;`~~-не будет компилироваться для

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

и, как правило, вам придется использовать настраиваемый маршалинг. Однако в таких случаях, как эта более простая техника, можно использовать:

```
[StructLayout(LayoutKind.Sequential)]
public struct TextOrInt
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 128)]
```

```
    public byte[] text;
    public int i { get { return BitConverter.ToInt32(text, 0); } }
}
```

Прочитайте Вызов платформы онлайн: https://riptutorial.com/ru/dot-net/topic/1643/вызов-платформы

глава 20: Глобализация в ASP.NET MVC с использованием интеллектуальной интернационализации для ASP.NET

замечания

Интеллектуальная интернационализация для страницы ASP.NET

Преимущество этого подхода в том, что вам не нужно загромождать контроллеры и другие классы с кодом для поиска значений из .resx-файлов. Вы просто размещаете текст в [[[тройные скобки.]]] (Разделитель настраивается.) `HttpModule` ищет перевод в вашем файле .po, чтобы заменить разделительный текст. Если перевод найден, `HttpModule` заменяет перевод. Если перевод не найден, он удаляет тройные скобки и отображает страницу с оригинальным нетранслируемым текстом.

.po файлы являются стандартным форматом для доставки переводов для приложений, поэтому для их редактирования имеется ряд приложений. Легко отправить файл .po нетехническому пользователю, чтобы они могли добавлять переводы.

Examples

Базовая конфигурация и настройка

1. Добавьте пакет nuget I18N в проект MVC.
2. В web.config добавьте `i18n.LocalizingModule` в `i18n.LocalizingModule` `<httpModules>` или `<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. Добавьте папку с именем «locale» в корень вашего сайта. Создайте подпапку для каждой культуры, которую вы хотите поддержать. Например, `/locale/fr/` .
4. В каждой папке с конкретными культурами создайте текстовый файл с именем `messages.po` .

5. Для тестирования введите следующие строки текста в файл `messages.po` :

```
#: Translation test
msgid "Hello, world!"
msgstr "Bonjour le monde!"
```

6. Добавьте контроллер в свой проект, который возвращает текст для перевода.

```
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. Запустите приложение MVC и перейдите на маршрут, соответствующий действию вашего контроллера, например http: // localhost: [yourportnumber] / default . Обратите внимание, что URL-адрес изменен, чтобы отразить вашу культуру по умолчанию, например http: // localhost: [yourportnumber] / en / default .
8. Замените `/en/` в URL с `/fr/` (или любой культурой, которую вы выбрали.) На странице теперь должна отображаться переведенная версия вашего текста.
9. Измените язык своего браузера, чтобы предпочесть альтернативную культуру и снова просмотреть `/default` . Обратите внимание, что URL-адрес изменен, чтобы отразить вашу альтернативную культуру, и появится переведенный текст.
10. В web.config добавьте обработчики, чтобы пользователи не могли просматривать вашу папку `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>
```

Прочитайте Глобализация в ASP.NET MVC с использованием интеллектуальной интернационализации для ASP.NET онлайн: https://riptutorial.com/ru/dot-net/topic/5086/глобализация-в-asp-net-mvc-с-использованием-интеллектуальной-интернационализации-для-asp-net

глава 21: Деревья выражений

замечания

Деревья выражений - это структуры данных, используемые для представления выражений кода в .NET Framework. Они могут быть сгенерированы с помощью кода и программно переведены, чтобы перевести код на другой язык или выполнить его. Самым популярным генератором деревьев выражений является сам компилятор C #. Компилятор C # может генерировать деревья выражений, если выражение lambda назначается переменной типа Expression <Func <... >>. Обычно это происходит в контексте LINQ. Наиболее популярным потребителем является поставщик LINQ от Entity Framework. Он потребляет деревья выражений, заданные Entity Framework, и генерирует эквивалентный код SQL, который затем выполняется в базе данных.

Examples

Простое дерево выражений, созданное компилятором C

Рассмотрим следующий код C #

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

Поскольку компилятор C # видит, что выражение lambda назначается типу Expression, а не тип делегата, он генерирует дерево выражений, примерно эквивалентное этому коду

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

Корнем дерева является выражение лямбда, которое содержит тело и список параметров. Лямбда имеет 1 параметр под названием «a». Тело - это одно выражение выражения BinaryExpression CLR и NodeType для Add. Это выражение представляет собой дополнение. Он имеет два подвыражения, обозначаемых как «левый» и «правый». Слева - это Параметрическое выражение для параметра «a», а Right - константное выражение со значением 1.

Простейшим использованием этого выражения является его печать:

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

Что печатает эквивалентный код C #.

Дерево выражений можно скомпилировать в делегат C # и выполнить с помощью CLR

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

Обычно выражения переводятся на другие языки, такие как SQL, но могут также использоваться для вызова частных, защищенных и внутренних членов публичных или непубличных типов в качестве альтернативы Reflection.

построение предиката поля формы == значение

Чтобы создать выражение, подобное `_ => _.Field == "VALUE"` во время выполнения.

Учитывая предикат `_ => _.Field` и строковое значение `"VALUE"` , создайте выражение, которое проверяет, является ли предикат истинным.

Выражение подходит для:

- `IQueryable<T>` , `IEnumerable<T>` для проверки предиката.
- сущности или `Linq` to `SQL` для создания предложения `Where` , которое проверяет предикат.

Этот метод построит соответствующее выражение `Equal` которое проверяет, равно ли `Field "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;
}
```

Предикат можно использовать, включив предикат в метод расширения `Where` .

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

Выражение для извлечения статического поля

Пример такого типа:

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

Мы можем получить значение StaticPublicField:

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

Затем он может быть скомпилирован в делегат для получения значения поля.

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

// результат fieldValue - StaticPublicFieldValue

Класс InvocationExpression

Класс InvocationExpression позволяет вызывать другие лямбда-выражения, которые являются частями одного и того же дерева выражений.

Вы создаете их со статическим методом `Expression.Invoke` .

Проблема Мы хотим получить предметы, которые имеют «автомобиль» в своем описании. Нам нужно проверить его на нуль, прежде чем искать строку внутри, но мы не хотим, чтобы ее вызывали чрезмерно, так как вычисление могло быть дорогостоящим.

```
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
        );
    }
}
```

Выход

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

Первое, что нужно отметить, - это то, как фактический доступ к собственности, завернутый в Invoke:

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

, и это единственная часть, `e.Description` , а вместо нее `extracted` параметр `string` типа передается следующему:

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

Еще одно важное замечание - `AndAlso` . Он вычисляет только левую часть, если первая часть возвращает «false». Ошибочно использовать побитовый оператор «A» вместо него, который всегда вычисляет обе части, и в этом примере завершится с ошибкой NullReferenceException.

Прочитайте Деревья выражений онлайн: https://riptutorial.com/ru/dot-net/topic/2657/деревья-выражений

глава 22: Для каждого

замечания

Использовать его вообще?

Вы можете утверждать, что намерение платформы .NET состоит в том, что запросы не имеют побочных эффектов, а метод `ForEach` по определению вызывает побочный эффект. Вы можете обнаружить, что ваш код более удобен в обслуживании и проще тестировать, если вместо этого вы используете простой `foreach` .

Examples

Вызов метода для объекта в списке

```
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());
```

Метод расширения для IEnumerable

`ForEach()` определяется в классе `List<T>` , но не в `IQueryable<T>` или `IEnumerable<T>` . В этих случаях у вас есть два варианта:

ToList первый

Нумерация (или запрос) будет оцениваться, копирование результатов в новый список или вызов базы данных. Затем этот метод вызывается для каждого элемента.

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ToList().ForEach(c => c.SendEmail());
```

Этот метод имеет очевидные издержки использования памяти, поскольку создается промежуточный список.

Метод расширения

Напишите метод расширения:

```
public static void ForEach<T>(this IEnumerable<T> enumeration, Action<T> action)
{
    foreach(T item in enumeration)
    {
        action(item);
    }
}
```

Использование:

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ForEach(c => c.SendEmail());
```

Внимание: методы LINQ Framework разработаны с целью быть *чистыми* , что означает, что они не производят побочных эффектов. Единственной целью метода `ForEach` является создание побочных эффектов и отклонение от других методов в этом аспекте. Вместо этого вы можете просто использовать простой цикл `foreach` .

Прочитайте Для каждого онлайн: https://riptutorial.com/ru/dot-net/topic/2225/для-каждого

глава 23: Единичное тестирование

Examples

Добавление проекта тестирования модуля MSTest к существующему решению

- Щелкните правой кнопкой мыши на решении, добавьте новый проект
- В разделе «Тестирование» выберите проект «Единичный тест»
- Выберите имя для сборки - если вы тестируете проект `Foo` , имя может быть `Foo.Tests`
- Добавить ссылку на тестируемый проект в ссылках на единичные тестовые проекты

Создание тестового метода

MSTest (платформа тестирования по умолчанию) требует, чтобы ваши тестовые классы были украшены атрибутом `[TestClass]` и тестовыми методами с атрибутом `[TestMethod]` и были общедоступными.

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

Прочитайте Единичное тестирование онлайн: https://riptutorial.com/ru/dot-net/topic/5365/единичное-тестирование

глава 24: Загрузка файлов и данных POST на веб-сервер

Examples

Загрузить файл с помощью WebRequest

Чтобы отправить файл и сформировать данные в одном запросе, содержимое должно иметь тип multipart / form-data .

```
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();
    }
}
```

Использование:

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

Прочитайте Загрузка файлов и данных POST на веб-сервер онлайн:
https://riptutorial.com/ru/dot-net/topic/10845/загрузка-файлов-и-данных-post-на-веб-сервер

глава 25: Запись и чтение из потока StdErr

Examples

Запись на стандартный вывод ошибки с помощью консоли

```
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}");
}
```

Чтение стандартной ошибки дочернего процесса

```
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}");
```

Прочитайте Запись и чтение из потока StdErr онлайн: https://riptutorial.com/ru/dot-net/topic/10779/запись-и-чтение-из-потока-stderr

глава 26: Исключения

замечания

Связанные с:

- MSDN: Исключения и обработка исключений (Руководство по программированию на C #)
- MSDN: Исключения для обработки и метания
- MSDN: CA1031: не использовать общие типы исключений
- MSDN: try-catch (ссылка C #)

Examples

Ловля исключения

Код может и должен вызывать исключения в исключительных обстоятельствах. Примеры этого включают:

- Попытка прочитать конец конца потока
- Отсутствие необходимых разрешений для доступа к файлу
- Попытка выполнить недопустимую операцию, например деление на ноль
- Тайм-аут, возникающий при загрузке файла из Интернета

Вызывающий может обрабатывать эти исключения, «ловя их», и должен делать это только тогда, когда:

- Он может фактически разрешить исключительные обстоятельства или восстановить надлежащим образом;
- Это может обеспечить дополнительный контекст для исключения, которое было бы полезно, если бы исключение нужно было перебросить (повторные выбросы исключений улавливаются обработчиками исключений далее в стеке вызовов)

Следует отметить, что выбор *не* для того, чтобы поймать исключение, является вполне допустимым, если намерение заключается в том, чтобы его обрабатывали на более высоком уровне.

Захват исключения выполняется путем переноса потенциально метательного кода в блок `try { ... }` следующим образом и улавливания исключений, которые он может обрабатывать в блоке `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);
}
```

Использование блока finally

`finally { ... }` блок `try-finally` или `try-catch-finally` всегда будет выполняться независимо от того, произошло ли исключение или нет (кроме случаев, когда `StackOverflowException` было выбрано или вызвано в `Environment.FailFast()`).

Его можно использовать для бесплатной или очистки ресурсов, полученных в блоке `try { ... }` .

```
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();
    }
}
```

Ловля и реорганизация пойманных исключений

Если вы хотите поймать исключение и что-то сделать, но вы не можете продолжить выполнение текущего блока кода из-за исключения, вы можете захотеть перебросить исключение в следующий обработчик исключений в стеке вызовов. Есть хорошие способы и плохие способы сделать это.

```
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
    }
}
```

Вы можете фильтровать по типу исключений и даже по свойствам исключений (новый в C # 6.0, более длинный доступный в VB.NET (ссылка)):

Документация / C # / новые функции

Исключительные фильтры

Поскольку исключения C # 6.0 могут быть отфильтрованы с использованием оператора `when` .

Это похоже на использование простого `if` но не разматывает стек, если условие внутри `when` не выполняется.

пример

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

Такую информацию можно найти в функциях C # 6.0 здесь: Исключительные фильтры

Повторное исключение в блоке catch

В блоке `catch` ключевое слово `throw` может использоваться самостоятельно, без указания значения исключения, для *восстановления* только что пойманного исключения. Повторное исключение позволяет исходному исключению продолжать цепочку обработки исключений, сохраняя стек вызовов или связанные данные:

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

Общим анти-шаблоном является вместо этого `throw ex` , что приводит к ограничению зрения следующего обработчика исключения для трассировки стека:

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

В общем случае использование `throw ex` нежелательно, так как будущие обработчики исключений, которые проверяют трассировку стека, смогут видеть вызовы еще до `throw ex` . Опуская переменную `ex` и используя ключевое слово `throw` исходное исключение будет « пузыряться» .

Выбрасывание исключения из другого метода при сохранении его информации

Иногда вы хотите поймать исключение и выбросить его из другого потока или метода, сохраняя исходный стек исключений. Это можно сделать с помощью `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();
    }
}
```

Прочитайте Исключения онлайн: https://riptutorial.com/ru/dot-net/topic/33/исключения

глава 27: Использование прогресса и IProgress

Examples

Отчет о прогрессе

`IProgress<T>` может использоваться для отчета о ходе выполнения некоторой процедуры для другой процедуры. В этом примере показано, как вы можете создать базовый метод, который сообщает о его прогрессе.

```
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);
    }
}
```

Выход:

```
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
```

Обратите внимание, что когда вы запускаете этот код, вы можете видеть, что номера выводятся не по порядку. Это связано с тем, что метод `IProgress<T>.Report()` запускается асинхронно и, следовательно, не подходит для ситуаций, когда прогресс должен сообщаться по порядку.

Использование IProgress

Важно отметить, что `System.Progress<T>` не имеет доступного к нему метода `Report()` . Этот метод был реализован явно из интерфейса `IProgress<T>` и поэтому должен быть вызван в `Progress<T>` когда он `IProgress<T>` на `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
```

Прочитайте Использование прогресса и IProgress онлайн: https://riptutorial.com/ru/dot-net/topic/5628/использование-прогресса--t--и-iprogress--t-

глава 28: Класс SpeechRecognitionEngine для распознавания речи

Синтаксис

- SpeechRecognitionEngine ()
- SpeechRecognitionEngine.LoadGrammar (грамматика грамматики)
- SpeechRecognitionEngine.SetInputToDefaultAudioDevice ()
- SpeechRecognitionEngine.RecognizeAsync (режим RecognizeMode)
- GrammarBuilder ()
- GrammarBuilder.Append (варианты выбора)
- Выбор (params string [])
- Грамматика (построитель GrammarBuilder)

параметры

`LoadGrammar` : **параметры**	**подробности**
грамматика	Грамматика для загрузки. Например, объект `DictationGrammar` чтобы разрешить бесплатную текстовую диктовку.
`RecognizeAsync` : **параметры**	**подробности**
Режим	`RecognizeMode` для текущего распознавания: `Single` для одного распознавания, `Multiple` чтобы разрешить несколько.
`GrammarBuilder.Append` : **параметры**	**подробности**
выбор	Добавляет некоторые варианты для создателя грамматики. Это означает, что, когда пользователь вводит речь, распознаватель может следовать за разными «ветвями» из грамматики.
`Choices` **конструктора:** **Параметры**	**подробности**
выбор	Множество вариантов для построителя грамматики. См. `GrammarBuilder.Append` .

`LoadGrammar` : параметры	подробности
`Grammar` конструктор: параметр	подробности
строитель	`GrammarBuilder` для построения `Grammar` .

замечания

Чтобы использовать `SpeechRecognitionEngine` , ваша версия Windows должна включать распознавание речи.

Вы должны добавить ссылку на `System.Speech.dll` прежде чем сможете использовать классы речи.

Examples

Асинхронное распознавание речи для бесплатной текстовой диктовки

```
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);
```

Асинхронное распознавание речи на основе ограниченного набора фраз

```
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);
```

Прочитайте Класс SpeechRecognitionEngine для распознавания речи онлайн: https://riptutorial.com/ru/dot-net/topic/69/класс-speechrecognitionengine-для-распознавания-

речи

глава 29: Класс System.IO.File

Синтаксис

- источник строки;
- назначение строки;

параметры

параметр	подробности
`source`	Файл, который нужно переместить в другое место.
`destination`	Каталог, в который вы хотите переместить `source` (эта переменная также должна содержать имя (и расширение файла) файла.

Examples

Удалить файл

Чтобы удалить файл (если у вас есть необходимые разрешения), просто:

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

Однако многие вещи могут пойти не так:

- У вас нет необходимых разрешений (вызывается `UnauthorizedAccessException`).
- Файл может быть использован кем-то другим (вызывается `IOException`).
- Файл не может быть удален из-за ошибки низкого уровня или носитель доступен только для чтения (`IOException`).
- Файл больше не существует (вызывается `IOException`).

Обратите внимание, что последняя точка (файл не существует) обычно *обходит* фрагмент кода следующим образом:

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

Однако это не атомная операция, и файл может быть `File.Exists()` кем-то другим между вызовом `File.Exists()` и перед `File.Delete()` . Правильный подход к обработке операций ввода-вывода требует обработки исключений (при условии, что при выполнении операции может быть предпринят альтернативный курс действий):

```
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
    }
}
```

Обратите внимание, что эти ошибки ввода-вывода иногда являются временными (например, файл используется), и если задействовано сетевое подключение, оно может автоматически восстанавливаться без каких-либо действий с нашей стороны. Затем обычно *повторяется* операция ввода-вывода несколько раз с небольшой задержкой между каждой попыткой:

```
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
```

Примечание: в среде Windows файл не будет действительно удален при вызове этой

функции, если кто-то еще откроет файл с помощью `FileShare.Delete` тогда файл можно удалить, но это будет эффективно, только когда владелец закроет файл.

Снимайте ненужные строки из текстового файла

Изменить текстовый файл непросто, потому что его содержимое нужно перемещать. Для *небольших* файлов самым простым способом является чтение его содержимого в памяти, а затем запись измененного текста.

В этом примере мы читаем все строки из файла и удаляем все пустые строки, а затем возвращаем исходный путь:

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

Если файл слишком велик, чтобы загрузить его в память, и путь вывода отличается от пути ввода:

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

Преобразование кодировки текстовых файлов

Текст сохраняется в кодировке (см. Также раздел «Строки»), и иногда вам может понадобиться изменить его кодировку, этот пример предполагает (для простоты), что файл не слишком велик, и его можно полностью прочитать в памяти:

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

При выполнении преобразований не забывайте, что файл может содержать спецификацию (знак байтового заказа), чтобы лучше понять, как он управляется, ссылаясь на Encoding.UTF8.GetString не принимает во внимание преамбулу / спецификацию .

«Коснитесь» большого количества файлов (чтобы обновить последнее время записи)

Этот пример обновляет последнее время записи огромного количества файлов (используя `System.IO.Directory.EnumerateFiles` вместо `System.IO.Directory.GetFiles()`). При желании вы можете указать шаблон поиска (по умолчанию это `"*.*"` И, в конечном итоге, искать через дерево каталогов (не только указанный каталог):

```
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);
    }
}
```

Перечислить файлы старше указанной суммы

Этот фрагмент является вспомогательной функцией для перечисления всех файлов старше указанного возраста, это полезно - например, когда вам приходится удалять старые файлы журналов или старые кешированные данные.

```
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);
}
```

Используется следующим образом:

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

Мало что нужно отметить:

- Поиск выполняется с использованием `Directory.EnumerateFiles()` вместо `Directory.GetFiles()` . Перечисление *живое,* тогда вам не нужно ждать, пока все записи в файловой системе не будут извлечены.
- Мы проверяем последнее время записи, но вы можете использовать время создания или время последнего доступа (например, для удаления *неиспользуемых* кешированных файлов, обратите внимание, что время доступа может быть отключено).
- Гранулярность не является однородной для всех этих свойств (время записи, время доступа, время создания), проверьте MSDN, чтобы узнать подробности об этом.

Перемещение файла из одного места в другое

File.Move

Чтобы переместить файл из одного места в другое, одна простая строка кода может это

сделать:

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

Тем не менее, есть много вещей, которые могут пойти не так с этой простой операцией. Например, что, если у пользователя, запускающего вашу программу, нет Диска с меткой « C»? Что делать, если они это сделали, но решили переименовать его в «B» или «M»?

Что делать, если исходный файл (файл, в который вы хотите переместить) был перемещен без вашего ведома - или что, если он просто не существует.

Это можно обойти, предварительно проверив, существует ли исходный файл:

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

Это гарантирует, что в этот момент файл действительно существует и может быть перемещен в другое место. Бывают случаи, когда простого вызова `File.Exists` будет недостаточно. Если это не так, проверьте еще раз, передайте пользователю, что операция завершилась неудачно - или обработайте исключение.

Исключение `FileNotFoundException` - это не единственное исключение, с которым вы, вероятно, столкнетесь.

Ниже приведены возможные исключения:

Тип исключения	Описание
`IOException`	Файл уже существует или исходный файл не найден.
`ArgumentNullException`	Значение параметров Source и / или Destination равно null.
`ArgumentException`	Значение параметров Source и / или Destination пусто или содержит недопустимые символы.
`UnauthorizedAccessException`	У вас нет необходимых разрешений для выполнения этого действия.
`PathTooLongException`	Исходный, целевой или заданный путь (ы) превышают максимальную длину. В Windows длина пути должна быть меньше 248 символов, а имена файлов должны быть менее 260 символов.
`DirectoryNotFoundException`	Указанный каталог не найден.
`NotSupportedException`	Пути источника или назначения или имена файлов

Тип исключения	Описание
	находятся в недопустимом формате.

Прочитайте Класс System.IO.File онлайн: https://riptutorial.com/ru/dot-net/topic/5395/класс-system-io-file

глава 30: Кодовые контракты

замечания

Кодовые контракты позволяют компилировать или анализировать временные условия методов и инвариантных условий для объектов. Эти условия могут использоваться для обеспечения того, чтобы вызывающие и возвращаемые значения соответствовали действительным состояниям для обработки приложений. Другие виды использования кодовых контрактов включают в себя создание документации.

Examples

Предпосылками

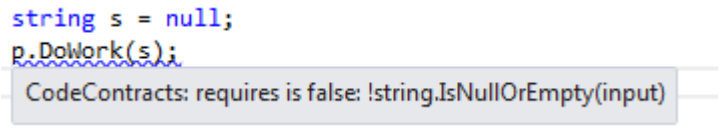
Предварительные условия позволяют методам предоставлять минимальные требуемые значения для входных параметров

Пример...

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

    //do work
}
```

Результат статического анализа ...



Постусловия

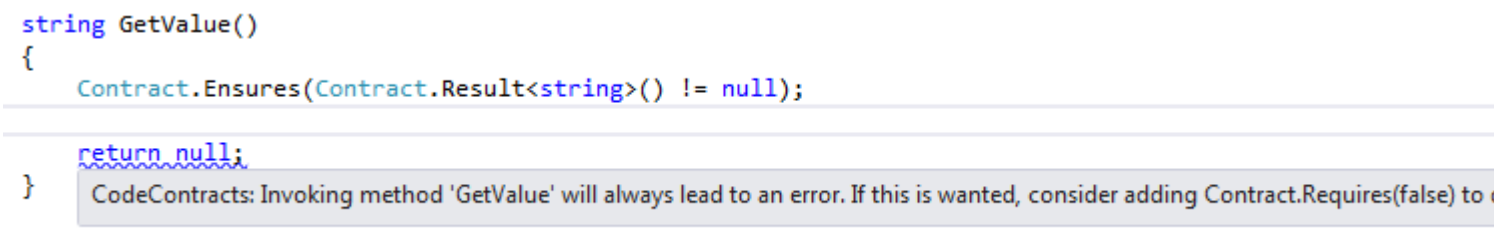
Постусловия гарантируют, что возвращаемые результаты метода будут соответствовать предоставленному определению. Это предоставляет вызывающему абоненту определение ожидаемого результата. Постусловия могут допускать упрощенные импликации, поскольку некоторые возможные результаты могут быть предоставлены статическим анализатором.

Пример...

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

```
    return null;
}
```

Результат статического анализа ...



Контракты для интерфейсов

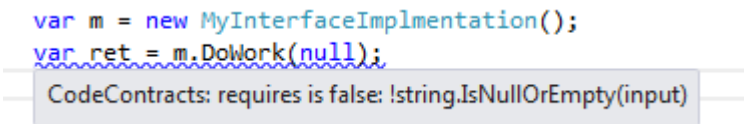
Используя Кодовые контракты, можно заключить договор с интерфейсом. Это делается путем объявления абстрактного класса, который связывает интерфейсы. Интерфейс должен быть помечен атрибутом `ContractClassAttribute` а определение контракта (абстрактный класс) должно быть помечено атрибутом `ContractClassForAttribute`

Пример C # ...

```
[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;
    }
}
```

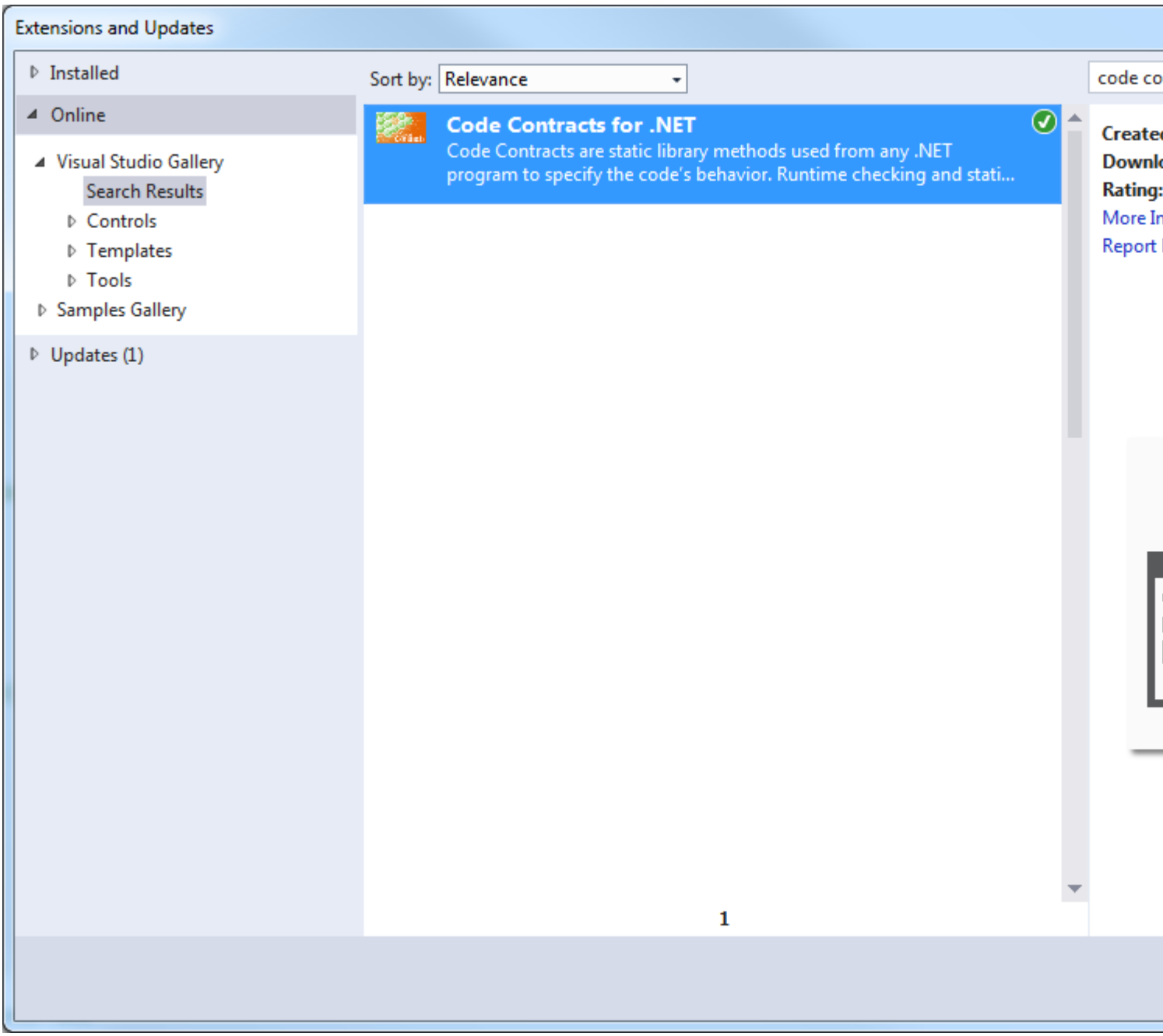
Результат статического анализа ...



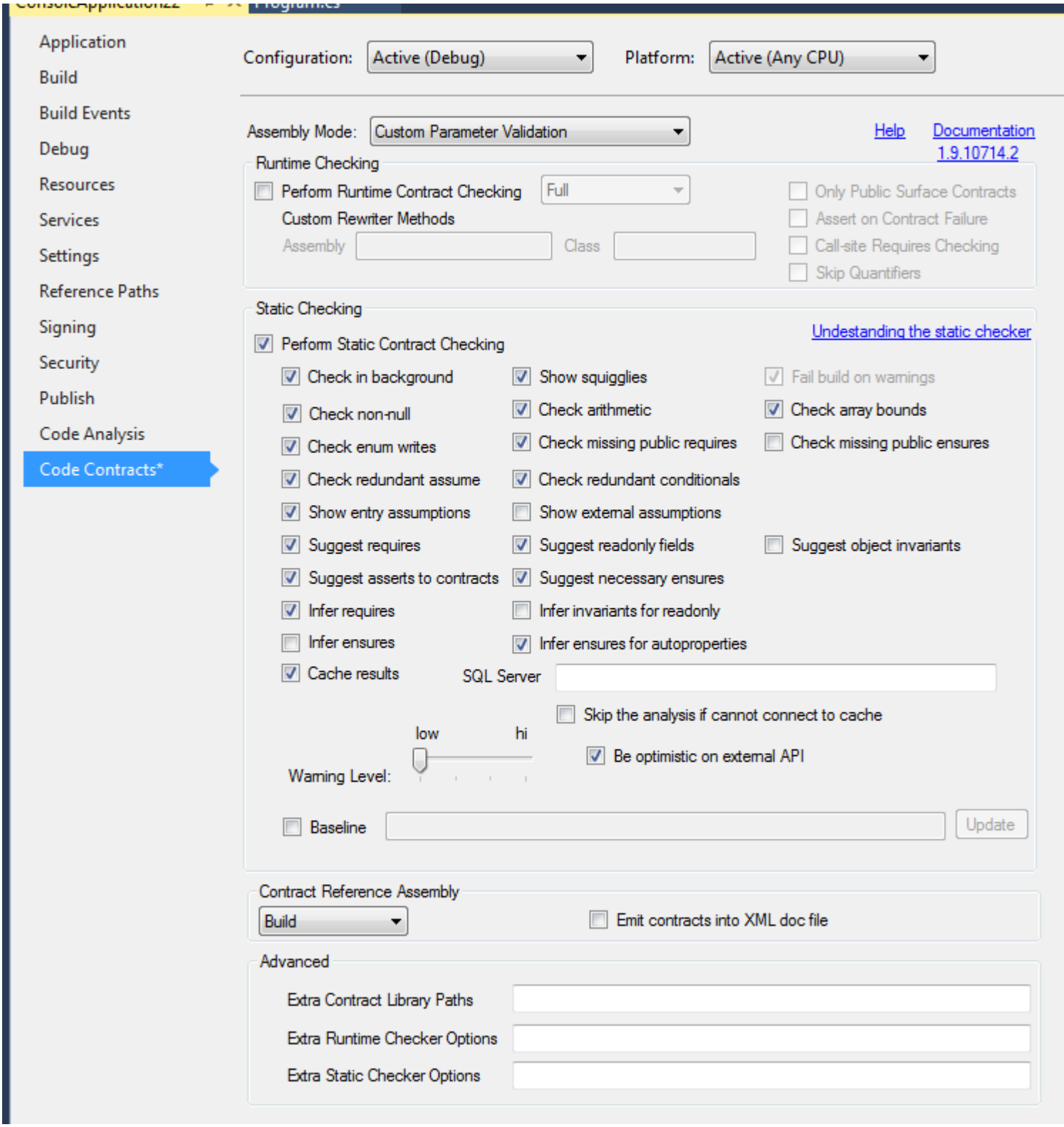
Установка и включение кодовых контрактов

Хотя `System.Diagnostics.Contracts` включен в .Net Framework. Чтобы использовать Кодовые контракты, вы должны установить расширения Visual Studio.

В разделе « `Extensions and Updates` найдите « `Code Contracts` затем установите `Code Contracts Tools` « `Code Contracts Tools`



После установки инструментов вы должны включить `Code Contracts` в свое решение Project. Как минимум, вы, вероятно, захотите включить `Static Checking` (проверьте после сборки). Если вы реализуете библиотеку, которая будет использоваться другими решениями, вы можете также рассмотреть возможность включения `Runtime Checking` .



Прочитайте Кодовые контракты онлайн: https://riptutorial.com/ru/dot-net/topic/1937/кодовые-контракты

глава 31: Коллекции

замечания

Существует несколько видов коллекций:

- `Array`
- `List`
- `Queue`
- `SortedList`
- `Stack`
- толковый словарь

Examples

Создание инициализированного списка с использованием пользовательских типов

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

Здесь у нас есть класс без конструктора с двумя свойствами: `Name` и nullable boolean property `Selected` . Если мы хотим инициализировать `List<Model>` , для его выполнения существует несколько разных способов.

```
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"}
 };
```

Здесь мы создаем несколько `new` экземпляров нашего класса `Model` и инициализируем их данными. Что делать, если мы добавили конструктор?

```
public class Model
{

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

```
    public bool? Selected { get; set; }
}
```

Это позволяет нам *немного* инициализировать наш список.

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

Что относительно класса, в котором одним из свойств является сам класс?

```
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; }
}
```

Обратите внимание, что мы вернули конструктор класса `Model` чтобы немного упростить пример.

```
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)
    }
};
```

Обратите внимание, что мы можем поменять наш `List<ExtendedModel>` с помощью `Collection<ExtendedModel>` , `ExtendedModel[]` , `object[]` или даже просто `[]` .

Очередь

В .Net используется коллекция для управления значениями в `Queue` , использующей

концепцию FIFO (first-in first-out) . `Enqueue(T item)` очередей является метод `Enqueue(T item)` который используется для добавления элементов в очередь и `Dequeue()` который используется для получения первого элемента и удаления его из очереди. Общая версия может использоваться как следующий код для очереди строк.

Сначала добавьте пространство имен:

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

и использовать его:

```
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
```

Существует не общий вариант типа, который работает с объектами.

Пространство имен:

```
using System.Collections;
```

Добавьте образец кода для не общей очереди:

```
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)
```

Существует также метод Peek (), который возвращает объект в начале очереди, не удаляя его.

```
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);
}
```

Выход (без снятия):

```
10
20
30
40
50
```

стек

Существует коллекция в .Net, используемая для управления значениями в `Stack` , использующем концепцию LIFO (last-in first-out) . Основами стеков является метод `Push(T item)` который используется для добавления элементов в стек и `Pop()` который используется для добавления последнего элемента и удаления его из стека. Общая версия может использоваться как следующий код для очереди строк.

Сначала добавьте пространство имен:

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

и использовать его:

```
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
```

Существует не общий вариант типа, который работает с объектами.

Пространство имен:

```
using System.Collections;
```

И образец кода не общего набора:

```
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)
```

Существует также метод Peek (), который возвращает последний добавленный элемент, но не удаляет его из `Stack` .

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

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

Можно выполнить итерацию по элементам в стеке, и он будет соблюдать порядок стека (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);
}
```

Выход (без снятия):

```
50
40
30
20
10
```

Использование инициализаторов коллекции

Некоторые типы коллекций могут быть инициализированы во время объявления. Например, следующий оператор создает и инициализирует `numbers` с целыми числами:

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

Внутренне компилятор C # фактически преобразует эту инициализацию в серию вызовов метода Add. Следовательно, вы можете использовать этот синтаксис только для коллекций, которые фактически поддерживают метод `Add` .

> Классы `Stack<T>` и `Queue<T>` не поддерживают его.

Для сложных наборов, таких как класс `Dictionary<TKey, TValue>` , которые принимают пары ключ / значение, вы можете указать каждую пару ключ / значение как анонимный тип в списке инициализаторов.

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

Первый элемент в каждой паре - это ключ, а второй - значение.

Прочитайте Коллекции онлайн: https://riptutorial.com/ru/dot-net/topic/30/коллекции

глава 32: Контексты синхронизации

замечания

Контекст синхронизации - это абстракция, которая позволяет потреблять код для передачи единиц работы планировщику, не требуя понимания того, как будет запланирована работа.

Контексты синхронизации традиционно используются для обеспечения того, чтобы код запускался в определенном потоке. В приложениях WPF и Winforms структура представления `SynchronizationContext` представляющая поток пользовательского интерфейса, предоставляется каркасом представления. Таким образом, `SynchronizationContext` можно рассматривать как шаблон-производитель-потребитель для делегатов. Рабочий поток *создает* исполняемый код (делегат) и ставит его в очередь или *потребляет* контур сообщения пользовательского интерфейса.

Параллельная библиотека задач предоставляет функции автоматического захвата и использования контекстов синхронизации.

Examples

Выполнить код в потоке пользовательского интерфейса после выполнения фоновой работы

В этом примере показано, как обновить компонент пользовательского интерфейса из фонового потока с помощью `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;
}
```

В этом примере, если вы попытались напрямую обновить `MyTextBox.Text` внутри цикла `for` ,

вы получите ошибку потоковой `MyTextBox.Text` . `UpdateCallback` действие `UpdateCallback` в `SynchronizationContext` , текстовое поле обновляется в том же потоке, что и в остальном пользовательском интерфейсе.

На практике обновления прогресса должны выполняться с использованием экземпляра `System.IProgress<T>` . Реализация по умолчанию `System.Progress<T>` автоматически фиксирует контекст синхронизации, на котором он создан.

Прочитайте Контексты синхронизации онлайн: https://riptutorial.com/ru/dot-net/topic/5407/контексты-синхронизации

глава 33: Многопоточность

Examples

Доступ к элементам управления формы из других потоков

Если вы хотите изменить атрибут элемента управления, такого как текстовое поле или ярлык из другого потока, кроме потока GUI, создавшего элемент управления, вам придется его вызывать, иначе вы можете получить сообщение об ошибке:

> «Неверная операция поперечного потока: Control 'control_name' обращается из потока, отличного от потока, в котором он был создан."

Используя этот примерный код в форме system.windows.forms, вы получите исключение с этим сообщением:

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

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

Вместо этого, если вы хотите изменить текст текстового поля из потока, который не владеет им, используйте Control.Invoke или Control.BeginInvoke. Вы также можете использовать Control.InvokeRequired для проверки необходимости использования элемента управления.

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

Если вам нужно делать это часто, вы можете написать расширение для вызываемых объектов, чтобы уменьшить количество кода, необходимого для выполнения этой проверки:

```
public static class Extensions
{
    public static void BeginInvokeIfRequired(this ISynchronizeInvoke obj, Action action)
    {
```

```
        if (obj.InvokeRequired)
            obj.BeginInvoke(action, new object[0]);
        else
            action();
    }
}
```

И обновление текстового поля из любого потока становится немного проще:

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

Имейте в виду, что Control.BeginInvoke, используемый в этом примере, является асинхронным, что означает, что код, поступивший после вызова Control.BeginInvoke, может быть запущен сразу после того, был ли еще выполнен переданный делегат.

Если вам нужно убедиться, что textBox1 обновлен до продолжения, вместо этого используйте Control.Invoke, который заблокирует вызывающий поток до тех пор, пока ваш делегат не будет выполнен. Обратите внимание, что этот подход может значительно снизить ваш код, если вы делаете много вызовов с вызовом и отмечаете, что он затормозит ваше приложение, если ваш поток GUI ждет, когда вызывающий поток завершит или освободит сохраненный ресурс.

Прочитайте Многопоточность онлайн: https://riptutorial.com/ru/dot-net/topic/3098/многопоточность

глава 34: Настройка привязки процесса и потока

параметры

параметр	подробности
сродство	целое число, описывающее набор процессоров, на которых разрешен запуск процесса. Например, в 8-процессорной системе, если вы хотите, чтобы ваш процесс выполнялся только на процессорах 3 и 4, вы выбираете аффинность следующим образом: 00001100, которая равна 12

замечания

Совместимость процессора с потоком - это набор процессоров, к которым он имеет отношение. Другими словами, это может быть запланировано для запуска.

Аффинность процессора представляет каждый процессор как бит. Бит 0 представляет процессор один, бит 1 представляет второй процессор и так далее.

Examples

Получить маску слияния процесса

```
    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');
    }
}
```

Пример использования:

```
private static void Main(string[] args)
{
    GetProcessAffinityMask();

    Console.ReadKey();
}
// Output:
// Process Test.vshost Affinity Mask is : 11111111
```

Установить маску сродства процесса

```
    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));
    }
```

Пример использования:

```
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
```

Прочитайте Настройка привязки процесса и потока онлайн: https://riptutorial.com/ru/dot-net/topic/4431/настройка-привязки-процесса-и-потока

глава 35: настройки

Examples

AppSettings из ConfigurationSettings в .NET 1.x

Устаревшее использование

Класс ConfigurationSettings был исходным способом получения настроек для сборки в .NET 1.0 и 1.1. Он был заменен классом ConfigurationManager и классом WebConfigurationManager .

Если у вас есть два ключа с тем же именем в разделе `appSettings` файла конфигурации, используется последний.

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>
```

Program.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();
        }
    }
}
```

Чтение AppSettings из ConfigurationManager в .NET 2.0 и более поздних

версиях

Класс ConfigurationManager поддерживает свойство `AppSettings` , которое позволяет продолжить чтение параметров из раздела `appSettings` файла конфигурации так же, как и . 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>
```

Program.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();
        }
    }
}
```

Введение в строго типизированную поддержку приложений и пользовательских настроек из Visual Studio

Visual Studio помогает управлять настройками пользователя и приложения. Использование этого подхода имеет эти преимущества перед использованием раздела `appSettings` файла конфигурации.

1. Настройки можно сделать строго типизированными. Любой тип, который может быть сериализован, может использоваться для значения настроек.

2. Настройки приложения можно легко отделить от пользовательских настроек. Параметры приложения хранятся в одном файле конфигурации: `web.config` для веб-

сайтов и веб-приложений и app.config, переименованных в *сборку* .exe.config, где *сборка* - это имя исполняемого файла. Пользовательские настройки (не используемые веб-проектами) хранятся в файле `user.config` в папке Application Data (в зависимости от версии операционной системы).

3. Параметры приложения из библиотек классов могут быть объединены в один файл конфигурации без риска конфликтов имен, поскольку каждая библиотека классов может иметь свой собственный раздел настроек.

В большинстве типов проектов конструктор свойств проекта имеет вкладку « Параметры », которая является отправной точкой для создания пользовательских приложений и пользовательских настроек. Первоначально вкладка «Параметры» будет пустой, с одной ссылкой для создания файла настроек по умолчанию. Щелчок по ссылке приводит к следующим изменениям:

1. Если файл конфигурации (`app.config` или `web.config`) не существует для проекта, он будет создан.

2. Вкладка «Настройки» будет заменена элементом управления сеткой, который позволит вам создавать, редактировать и удалять записи отдельных настроек.

3. В обозревателе решений в элементе свойств Properties добавлен элемент `Settings.settings` . Открытие этого элемента откроет вкладку «Настройки».

4. В папке « `Properties` » в папке проекта добавлен новый файл с новым частичным классом. Этот новый файл называется `Settings.Designer.__` (.cs, .vb и т. Д.), А класс называется `Settings` . Класс генерируется кодом, поэтому его нельзя редактировать, но класс является частичным классом, поэтому вы можете расширить класс, добавив дополнительные элементы в отдельный файл. Кроме того, класс реализуется с использованием шаблона Singleton, выставляя экземпляр singleton с свойством `Default` .

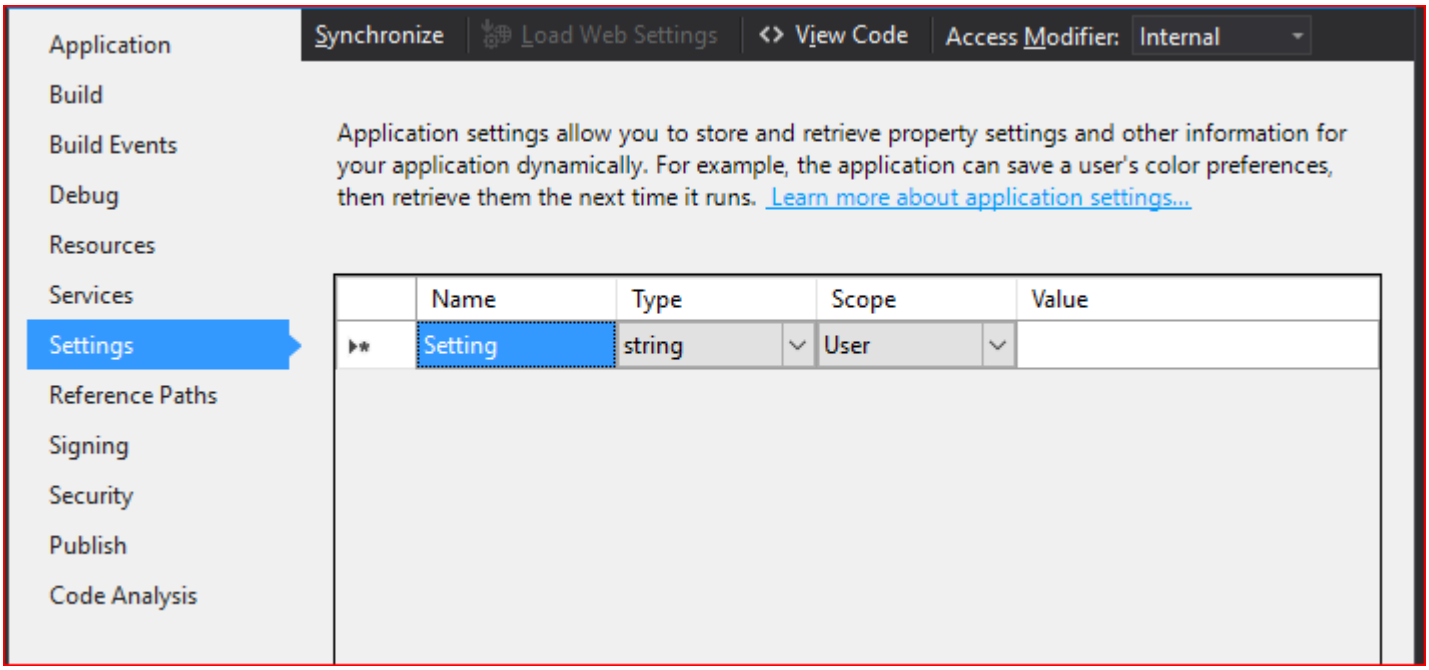
Когда вы добавляете каждую новую запись на вкладку «Настройки», Visual Studio выполняет следующие две задачи:

1. Сохраняет настройку в файле конфигурации в настраиваемом разделе конфигурации, предназначенном для управления классом Settings.

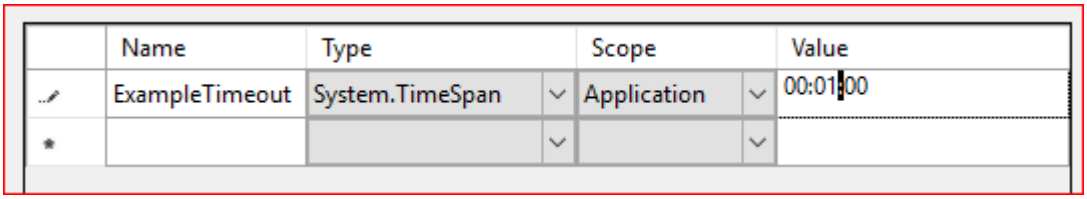
2. Создает новый член в классе «Параметры» для чтения, записи и представления настроек в определенном типе, выбранном на вкладке «Настройки».

Чтение строго типизированных параметров из пользовательского раздела файла конфигурации

Начиная с нового класса настроек и настраиваемой конфигурации:



Добавьте параметр приложения с именем ExampleTimeout, используя время System.Timespan и установите значение 1 минута:



Сохраните свойства проекта, в котором сохраняются записи вкладки «Настройки», а также повторно создается собственный класс «Параметры» и обновляется файл конфигурации проекта.

Используйте настройку из кода (C #):

Program.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();
        }
    }
}
```

Под крышками

Просмотрите файл конфигурации проекта, чтобы узнать, как была создана запись параметра приложения:

app.config (Visual Studio автоматически обновляет)

```
<?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>
```

Обратите внимание, что раздел `appSettings` не используется. Раздел `applicationSettings` содержит настраиваемый раздел, соответствующий определенному пространству имен, который имеет элемент `setting` для каждой записи. Тип значения не сохраняется в файле конфигурации; он известен только классу `Settings` .

Посмотрите в классе « `Settings` », чтобы узнать, как он использует класс `ConfigurationManager` для чтения этого настраиваемого раздела.

Settings.designer.cs (для проектов 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"]));
        }
    }
...
```

Обратите внимание, что был создан `DefaultSettingValueAttribute` для сохранения значения, введенного на вкладке «Параметры» в «Дизайнере свойств проекта». Если в файле конфигурации отсутствует запись, это значение по умолчанию используется.

Прочитайте настройки онлайн: https://riptutorial.com/ru/dot-net/topic/54/настройки

глава 36: Обзор параллельной библиотеки задач (TPL)

замечания

Параллельная библиотека задач представляет собой набор общедоступных типов и API, которые значительно упрощают процесс добавления параллелизма и параллелизма в приложение. .Сеть. TPL был введен в .Net 4 и является рекомендуемым способом написания многопоточного и параллельного кода.

TPL заботится о планировании работы, близости потоков, поддержке отмены, управлении состоянием и балансировке нагрузки, чтобы программист мог сосредоточиться на решении проблем, а не тратить время на общие детали низкого уровня.

Examples

Выполните работу в ответ на нажатие кнопки и обновите пользовательский интерфейс.

В этом примере показано, как вы можете ответить на нажатие кнопки, выполнив некоторую работу над рабочим потоком, а затем обновите интерфейс пользователя, чтобы указать завершение

```
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.

}
```

Прочитайте Обзор параллельной библиотеки задач (TPL) онлайн:
https://riptutorial.com/ru/dot-net/topic/5164/обзор-параллельной-библиотеки-задач--tpl-

глава 37: отражение

Examples

Что такое Ассамблея?

Ассемблы являются строительным блоком любого приложения Common Language Runtime (CLR) . Каждый тип, который вы определяете вместе со своими методами, свойствами и их байт-кодом, компилируется и упаковывается внутри сборки.

```
using System.Reflection;
```

```
Assembly assembly = this.GetType().Assembly;
```

Ассемблы самодокументируются: они содержат не только типы, методы и их код IL, но также и метаданные, необходимые для их проверки и использования, как при компиляции, так и во время выполнения:

```
Assembly assembly = Assembly.GetExecutingAssembly();

foreach (var type in assembly.GetTypes())
{
    Console.WriteLine(type.FullName);
}
```

Ассемблы имеют имена, которые описывают их полную, уникальную идентичность:

```
Console.WriteLine(typeof(int).Assembly.FullName);
// Will print: "mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

Если это имя содержит `PublicKeyToken` , оно называется *сильным именем* . Сильным обозначением сборки является процесс создания подписи с использованием закрытого ключа, который соответствует открытому ключу, распространяемому вместе с сборкой. Эта подпись добавляется в манифест Ассамблеи, который содержит имена и хэши всех файлов, составляющих сборку, и его `PublicKeyToken` становится частью имени. Ассемблы, имеющие такое же сильное имя, должны быть одинаковыми; сильные имена используются для управления версиями и для предотвращения конфликтов сборки.

Как создать объект T с помощью Reflection

Использование конструктора по умолчанию

```
T variable = Activator.CreateInstance(typeof(T));
```

Использование параметризованного конструктора

```
T variable = Activator.CreateInstance(typeof(T), arg1, arg2);
```

Создание объекта и настройка свойств с использованием отражения

Допустим, у нас есть класс `Classy` который обладает свойством Propertua

```
public class Classy
{
    public string Propertua {get; set;}
}
```

для установки `Propertua` с использованием отражения:

```
var typeOfClassy = typeof (Classy);
var classy = new Classy();
var prop = typeOfClassy.GetProperty("Propertua");
prop.SetValue(classy, "Value");
```

Получение атрибута enum с отражением (и кэширование)

Атрибуты могут быть полезны для обозначения метаданных на перечислениях. Получение значения этого может быть медленным, поэтому важно кэшировать результаты.

```
    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;
        }
    }
```

Сравните два объекта с отражением

```
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;
    }
}
```

Примечание: в этом примере выполняется сравнение на основе полей (игнорирование статических полей и свойств) для простоты

Прочитайте отражение онлайн: https://riptutorial.com/ru/dot-net/topic/44/отражение

глава 38: Параллельная библиотека задач (TPL)

замечания

Назначение и использование случаев

Цель параллельной библиотеки задач - упростить процесс написания и поддержки многопоточного и параллельного кода.

Некоторые примеры использования *:

- Сохранение пользовательского интерфейса при выполнении фоновой работы по отдельной задаче
- Распределение рабочей нагрузки
- Разрешить клиентскому приложению отправлять и получать запросы одновременно (отдых, TCP / UDP, ect)
- Чтение и / или запись нескольких файлов одновременно

* Кодекс следует рассматривать в каждом конкретном случае для многопоточности. Например, если в цикле имеется несколько итераций или выполняется только небольшая часть работы, накладные расходы для параллелизма могут перевесить преимущества.

TPL с .Net 3.5

TPL также доступен для .Net 3.5, входящего в пакет NuGet, он называется Task Parallel Library.

Examples

Базовая цепочка производителей-потребителей (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!");
```

```
});
```

Стоит отметить, что если вы не вызываете `collection.CompleteAdding();` , вы можете продолжать добавлять коллекцию, даже если ваша потребительская задача запущена. Просто вызовите `collection.CompleteAdding();` когда вы уверены, что больше нет дополнений. Эта функциональность может быть использована для того, чтобы сделать несколько производителей одним шаблоном потребителя, где у вас есть несколько источников, которые подают элементы в BlockingCollection, и один потребитель вытаскивает предметы и что-то делает с ними. Если ваш BlockingCollection пуст перед вызовом полного добавления, Enumerable from `collection.GetConsumingEnumerable()` будет блокироваться до добавления нового элемента в коллекцию или BlockingCollection.CompleteAdding (); и очередь пуста.

```
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!");
```

Задача: базовая инстанция и Wait

Задача может быть создана путем непосредственного создания класса `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!");
```

... или используя статический метод `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!");
```

Обратите внимание, что только в первом случае необходимо явно вызвать `Start` .

Задача: WaitAll и захват переменных

```
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);
```

Задача: 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);
```

Примечание: окончательный `WaitAll` необходим, `WaitAny` что `WaitAny` не вызывает исключений.

Задача: обработка исключений (с использованием Wait)

```
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
```

Задача: обработка исключений (без использования Wait)

```
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
```

Задача: отменить с помощью CancellationToken

```
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
```

В качестве альтернативы `ThrowIfCancellationRequested` запрос аннулирования может быть обнаружен с помощью `IsCancellationRequested` а `OperationCanceledException` может быть `IsCancellationRequested` вручную:

```
//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);
```

Обратите внимание, как маркер отмены передается конструктору задачи в параметре `cancellationToken` . Это необходимо для того , что задача переходит в `Canceled` состоянии, а не в `Faulted` состояния, когда `ThrowIfCancellationRequested` вызываются. Кроме того, по той же причине маркер отмены явно указывается в конструкторе `OperationCanceledException` во втором случае.

Task.WhenAny

```
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!");
```

Task.WhenAll

```
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
```

Parallel.Invoke

```
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);
}
```

Parallel.ForEach

В этом примере `Parallel.ForEach` использует для вычисления суммы чисел от 1 до 10000 с помощью нескольких потоков. Для обеспечения безопасности потоков `Interlocked.Add` используется для суммирования чисел.

```
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
```

```
}
```

Parallel.For

В этом примере используется `Parallel.For` для вычисления суммы чисел от 1 до 10000 с использованием нескольких потоков. Для обеспечения безопасности потоков `Interlocked.Add` используется для суммирования чисел.

```
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
}
```

Исходный контекст выполнения с AsyncLocal

Когда вам нужно передать некоторые данные из родительской задачи в свои дочерние задачи, чтобы она логически `AsyncLocal` с выполнением, используйте класс `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);
}
```

Примечание. Как видно из приведенного выше примера, `AsynLocal.Value` имеет `copy on read` семантике `copy on read` , но если вы используете некоторый ссылочный тип и меняете его свойства, вы будете влиять на другие задачи. Следовательно, наилучшая практика с `AsyncLocal` заключается в использовании типов значений или неизменяемых типов.

Parallel.ForEach в 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)
```

Задача: возврат значения

Задача, возвращающая значение, возвращает тип `Task< TResult >` где TResult - тип значения, которое необходимо вернуть. Вы можете запросить результат задачи по свойству Result.

```
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
```

Если Задача выполняется асинхронно, чем ожидание задачи, возвращает результат.

```
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");
}
```

Прочитайте Параллельная библиотека задач (TPL) онлайн: https://riptutorial.com/ru/dot-net/topic/55/параллельная-библиотека-задач--tpl-

глава 39: Параллельная обработка с использованием .Net framework

Вступление

Эта тема посвящена многоядерному программированию с использованием параллельной библиотеки задач с платформой .NET. Параллельная библиотека задач позволяет вам писать код, который читается человеком и настраивается с количеством доступных ячеек. Поэтому вы можете быть уверены, что ваше программное обеспечение автоматически обновит себя с помощью среды обновления.

Examples

Параллельные расширения

Параллельные расширения были добавлены параллельно с параллельной библиотекой задач для достижения параллелизма данных. Параллелизм данных относится к сценариям, в которых одна и та же операция выполняется параллельно (то есть параллельно) с элементами в исходном наборе или массиве. .NET предоставляет новые конструкции для достижения параллелизма данных с использованием конструкций Parallel.For и Parallel.Foreach.

```
//Sequential version

foreach (var item in sourcecollection){

Process(item);

}

// Parallel equivalent

Parallel.foreach(sourcecollection, item => Process(item));
```

Вышеупомянутая конструкция Parallel.ForEach использует несколько ядер и, таким образом, повышает производительность таким же образом.

Прочитайте Параллельная обработка с использованием .Net framework онлайн: https://riptutorial.com/ru/dot-net/topic/8085/параллельная-обработка-с-использованием--net-framework

глава 40: Пользовательские типы

замечания

Обычно `struct` используется только тогда, когда производительность очень важна. Поскольку типы значений живут в стеке, их можно получить гораздо быстрее, чем классы. Однако в стеке гораздо меньше места, чем куча, поэтому структуры должны быть небольшими (Microsoft рекомендует, чтобы `struct` s занимала не более 16 байт).

`class` является наиболее часто используемым типом (из этих трех) в C #, и, как правило, вы должны идти первым.

`enum` используется всякий раз, когда вы можете иметь четко определенный, отличный список элементов, которые нужно определить только один раз (во время компиляции). Перечисления полезны для программистов как облегченная ссылка на какое-то значение: вместо определения списка `constant` переменных для сравнения вы можете использовать перечисление и получать поддержку Intellisense, чтобы убедиться, что вы случайно не используете неправильное значение.

Examples

Определение структуры

Структуры, наследуемые от System.ValueType, являются типами значений и живут в стеке. Когда типы значений передаются как параметр, они передаются по значению.

```
Struct MyStruct
{
    public int x;
    public int y;
}
```

Передано по значению означает, что значение параметра *копируется* для метода, и любые изменения, внесенные в параметр в методе, не отражаются вне метода. Например, рассмотрим следующий код, который вызывает метод с именем `AddNumbers` , передающий переменные `a` и `b` , которые имеют тип `int` , который является типом Value.

```
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
}
```

Даже если мы добавили 5 к `x` внутри метода, значение остается неизменным, потому что это тип значения, и это означает , что `a x` была *копия* `a` «значение с, но на самом деле не . `a`

Помните, что типы значений живут в стеке и передаются по значению.

Определение класса

Классы, наследуемые от System.Object, являются ссылочными типами и живут в куче. Когда ссылочные типы передаются в качестве параметра, они передаются по ссылке.

```
public Class MyClass
{
    public int a;
    public int b;
}
```

Передано по ссылке означает, что *ссылка* на параметр передается методу, и любые изменения параметра будут отражаться вне метода, когда он вернется, потому что ссылка относится *к одному и тому же объекту в памяти* . Давайте используем тот же пример, что и раньше, но сначала будем «обертывать» `int` s в классе.

```
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
}
```

На этот раз, когда мы изменили `sample.a` на `10` , значение `instanceOfMyClass.a` *также* изменяется, потому что оно было *передано по ссылке* . Передано ссылкой означает, что *ссылка* (также иногда называемая *указателем*) на объект передавалась в метод вместо копии самого объекта.

Помните, что типы ссылок живут в куче и передаются по ссылке.

Определение Enum

Перечисление - это особый тип класса. Ключевое слово `enum` сообщает компилятору, что этот класс наследуется от абстрактного класса System.Enum. Перечисления используются для отдельных списков элементов.

```
public enum MyEnum
{
    Monday = 1,
    Tuesday,
    Wednesday,
    //...
}
```

Вы можете представить себе перечисление как удобный способ сопоставления констант с некоторым базовым значением. Перечисленное выше, объявляет значения для каждого дня недели и начинается с `1` . `Tuesday` автоматически будет отображаться на `2` , в `Wednesday` до `3` и т. Д.

По умолчанию перечисления используют `int` как базовый тип и начинаются с 0, но вы можете использовать любой из следующих *интегральных типов* : `byte, sbyte, short, ushort, int, uint, long, or ulong` и можете указывать явные значения для любого вещь. Если некоторые элементы явно указаны, но некоторые из них не указаны, каждый элемент после последнего определенного будет увеличен на 1.

Мы хотели бы использовать этот пример *литья* другого значения для *MyEnum* как так:

```
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
```

Другой полезный, хотя и более сложный тип перечисления называется `Flags` . При *украшении* перечисления с на `Flags` атрибутов, вы можете присвоить переменное более одного значения в то время. Обратите внимание: при этом вы *должны* явно определять значения в представлении базы 2.

```
[Flags]
public enum MyEnum
{
    Monday = 1,
    Tuesday = 2,
```

```
    Wednesday = 4,
    Thursday = 8,
    Friday = 16,
    Saturday = 32,
    Sunday = 64
}
```

Теперь вы можете сравнивать более одного значения за раз, используя *побитовое сравнение* или, если вы используете .NET 4.0 или новее, встроенный метод `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
}
```

Поскольку класс Enum подклассифицирован из `System.ValueType` , он рассматривается как тип значения и передается по значению, а не по ссылке. Базовый объект создается в куче, но когда вы передаете значение перечисления в вызов функции, копия значения, использующего базовый тип значения Enum (обычно System.Int32), помещается в стек. Компилятор отслеживает связь между этим значением и базовым объектом, который был создан в стеке. Дополнительную информацию см. В разделе ValueType Class (System) (MSDN) .

Прочитайте Пользовательские типы онлайн: https://riptutorial.com/ru/dot-net/topic/57/пользовательские-типы

глава 41: Последовательные порты

Examples

Основная операция

```
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();
```

Список доступных имен портов

```
string[] portNames = SerialPort.GetPortNames();
```

Асинхронное считывание

```
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;
        }
    };
```

Синхронный текстовый эхо-сервис

```
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();
        }
    }
}
```

Асинхронный приемник сообщений

```
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;
        }
    }
}
```

Эта программа ждет сообщений, заключенных в байты `STX` и `ETX` и выводит текст, идущий между ними. Все остальное отбрасывается. При переполнении буфера записи он останавливается. При других ошибках он перезапускает входные и выходные буферы и ждет дополнительных сообщений.

Код иллюстрирует:

- Чтение асинхронного последовательного порта (см. Использование `SerialPort.DataReceived`).
- Обработка ошибок последовательного порта (см. Использование `SerialPort.ErrorReceived`).
- Внеконтекстная реализация протокола.
- Чтение частичных сообщений.
    - Событие `SerialPort.DataReceived` может произойти раньше всего сообщения (до `ETX`). Все сообщение также может быть недоступно во входном буфере (SerialPort.Read (..., ..., port.BytesToRead) читает только часть сообщения). В этом случае мы запишем полученную часть (`unprocessedBuffer`) и продолжаем ждать дальнейших данных.
- Работа с несколькими сообщениями происходит за один раз.
    - Событие `SerialPort.DataReceived` может произойти только после отправки нескольких сообщений другим концом.

Прочитайте Последовательные порты онлайн: https://riptutorial.com/ru/dot-net/topic/5366/последовательные-порты

глава 42: Поток данных TPL

замечания

Библиотеки, используемые в примерах

`System.Threading.Tasks.Dataflow`

`System.Threading.Tasks`

`System.Net.Http`

`System.Net`

Разница между Post и SendAsync

Чтобы добавить элементы в блок, вы можете использовать `Post` или `SendAsync` .

`Post` будет пытаться добавить элемент синхронно и вернуть `bool` говоря, удалось ли это или нет. Это может быть неудачно, если, например, блок достиг своего `BoundedCapcity` и не имеет больше места для новых элементов. `SendAsync` другой стороны `SendAsync` вернет незавершенную `Task<bool>` которую вы можете `await` . Эта задача завершится в будущем с `true` результатом, когда блок очистит свою внутреннюю очередь и сможет принимать больше элементов или с `false` результатом, если он постоянно отказывается (например, в результате отмены).

Examples

Проводка в ActionBlock и ожидание завершения

```
// 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
```

Связывание блоков для создания конвейера

```
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
```

Синхронный производитель / потребитель с буфером

```
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 });
    }
}
```

Потребитель асинхронного производителя с ограниченным буфером

```
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);
```

Прочитайте Поток данных TPL онлайн: https://riptutorial.com/ru/dot-net/topic/784/поток-данных-tpl

глава 43: Пространство имен System.Reflection.Emit

Examples

Создание сборки динамически

```
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
    }
}
```

Прочитайте Пространство имен System.Reflection.Emit онлайн: https://riptutorial.com/ru/dot-net/topic/74/пространство-имен-system-reflection-emit

глава 44: Работа с SHA1 в C

Вступление

в этом проекте вы видите, как работать с криптографической хэш-функцией SHA1. например, получить хэш из строки и как взломать хэш SHA1. источник на git-хабе: https://github.com/mahdiabasi/SHA1Tool

Examples

Генерация контрольной суммы SHA1 файловой функции

Сначала вы добавляете System.Security.Cryptography и System.IO в свой проект

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
            SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

Прочитайте Работа с SHA1 в C # онлайн: https://riptutorial.com/ru/dot-net/topic/9457/работа-с-sha1-в-c-sharp

глава 45: Работа с SHA1 в C

Вступление

в этом проекте вы видите, как работать с криптографической хэш-функцией SHA1. например, получить хэш из строки и как взломать хэш SHA1.

исходный компилятор на github: https://github.com/mahdiabasi/SHA1Tool

Examples

Генерация контрольной суммы SHA1 файла

> сначала вы добавляете пространство имен System.Security.Cryptography в свой проект

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
            SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

Генерируемый хэш текста

```
  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();
    }
```

Прочитайте Работа с SHA1 в C # онлайн: https://riptutorial.com/ru/dot-net/topic/9458/работа-с-sha1-в-c-sharp

глава 46: Регулярные выражения (System.Text.RegularExpressions)

Examples

Проверьте, соответствует ли шаблон вводам

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

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

Варианты передачи

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

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

Простое совпадение и замена

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

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

Сопоставление по группам

```
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;
```

```
}
```

Удаление не буквенно-цифровых символов из строки

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

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

Найти все совпадения

С помощью

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

Код

```
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);
}
```

Выход

```
Carrot
Cherry
Clementine
```

Прочитайте Регулярные выражения (System.Text.RegularExpressions) онлайн: https://riptutorial.com/ru/dot-net/topic/6944/регулярные-выражения--system-text-regularexpressions-

глава 47: Сериализация JSON

замечания

JavaScriptSerializer против Json.NET

Класс `JavaScriptSerializer` был введен в .NET 3.5 и используется внутренне асинхронным уровнем связи .NET для приложений с поддержкой AJAX. Он может использоваться для работы с JSON в управляемом коде.

Несмотря на существование класса `JavaScriptSerializer` , Microsoft рекомендует использовать библиотеку Json.NET с открытым исходным кодом для сериализации и десериализации. Json.NET обеспечивает более высокую производительность и более дружественный интерфейс для отображения JSON на пользовательские классы (пользовательские `JavaScriptConverter` объект будет нужно сделать то же самое с `JavaScriptSerializer`).

Examples

Десериализация с использованием System.Web.Script.Serialization.JavaScriptSerializer

Метод `JavaScriptSerializer.Deserialize<T>(input)` пытается десериализовать строку допустимого JSON в объект указанного типа `<T>` , используя сопоставления по умолчанию, поддерживаемые `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"]
```

Примечание. Объект `JavaScriptSerializer` был представлен в версии .NET 3.5.

Десериализация с использованием 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);
```

Для получения дополнительной информации см. Официальный сайт Json.NET .

Примечание. Json.NET поддерживает .NET версии 2 и выше.

Сериализация с использованием 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}
```

Обратите внимание, что свойства (и классы) могут быть украшены атрибутами, чтобы изменить их внешний вид в результирующей строке json или вообще удалить их из json-строки (JsonIgnore).

Более подробную информацию о атрибутах сериализации Json.NET можно найти здесь .

В C # общедоступные идентификаторы записываются в *PascalCase* по соглашению. В JSON соглашение заключается в использовании *camelCase* для всех имен. Вы можете использовать контрактный преобразователь для преобразования между ними.

```
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}
}
```

Сериализация-десериализация с использованием Newtonsoft.Json

В отличие от других помощников, этот использует помощники статического класса для сериализации и десериализации, поэтому он немного проще, чем другие.

```
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);
```

Динамическое связывание

Json.NET от Newtonsoft позволяет вам динамически связывать json (используя объекты ExpandoObject / Dynamic) без необходимости явно создавать тип.

Сериализация

```
dynamic jsonObject = new ExpandoObject();
jsonObject.Title   = "Merchent of Venice";
jsonObject.Author  = "William Shakespeare";
Console.WriteLine(JsonConvert.SerializeObject(jsonObject));
```

Десериализация

```
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);
```

Обратите внимание, что ключи в объекте rawJson были превращены в переменные-члены в динамическом объекте.

Это полезно в тех случаях, когда приложение может принимать / производить различные форматы JSON. Тем не менее предлагается использовать дополнительный уровень проверки для строки Json или динамического объекта, сгенерированного в результате сериализации / де-сериализации.

Сериализация с использованием Json.NET с настройками JsonSerializerSettings

Этот сериализатор имеет некоторые приятные функции, которые не имеет сериализатора

по умолчанию .net json serializer, например обработка нулевого значения, вам просто нужно создать `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
NullValueHandling = NullValueHandling.Ignore});
   return result;
}
```

Еще одна серьезная проблема с сериализатором в .net - это цикл саморегуляции. В случае учащегося, участвующего в курсе, его экземпляр имеет свойство курса, а курс имеет набор учеников, что означает `List<Student>` который создаст опорный цикл. Вы можете справиться с этим с помощью `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
ReferenceLoopHandling = ReferenceLoopHandling.Ignore});
   return result;
}
```

Вы можете поместить следующий вариант сериализации следующим образом:

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
NullValueHandling = NullValueHandling.Ignore, ReferenceLoopHandling =
ReferenceLoopHandling.Ignore});
   return result;
}
```

Прочитайте Сериализация JSON онлайн: https://riptutorial.com/ru/dot-net/topic/183/сериализация-json

глава 48: сетей

замечания

См. Также: HTTP-клиенты

Examples

Основной TCP-чат (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);
            }
        }
    }
}
```

Основной клиент SNTP (UdpClient)

См. RFC 2030 для получения подробной информации о протоколе 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.#;.}");
    }
}
```

Прочитайте сетей онлайн: https://riptutorial.com/ru/dot-net/topic/35/сетей

глава 49: Синхронизация DateTime

Examples

ParseExact

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

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

24.11.2013 12:00:00

Обратите внимание, что передача `CultureInfo.CurrentCulture` в качестве третьего параметра идентична передаче `null` . Или вы можете передать определенную культуру.

Форматировать строки

Строка ввода может быть в любом формате, соответствующем строке формата

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

24.11.2013 12:00:00

Любые символы, не являющиеся спецификаторами формата, рассматриваются как литералы

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

24.11.2013 12:00:00

Вопросы для спецификаторов формата

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

24.11.2011 11:01:30

Обратите внимание, что значения месяца и минуты были проанализированы в неправильные адресаты.

Строки с одним символьным символом должны быть одним из стандартных форматов

```
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);
```

Исключения

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
```

Обработка нескольких возможных форматов

```
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);
```

Обработка различий в культуре

```
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);
```

День: 11; Месяц: 10

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

День: 10; Месяц: 11

TryParse

Этот метод принимает строку как входную, пытается проанализировать ее в `DateTime` и возвращает логический результат, указывающий на успех или неудачу. Если вызов преуспевает, переменная, переданная как параметр `out` заполняется анализируемым результатом.

Если синтаксический анализ не выполняется, переменная, переданная как параметр `out` устанавливается на значение по умолчанию `DateTime.MinValue` .

TryParse (строка, вне DateTime)

```
DateTime parsedValue;

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

Этот метод пытается проанализировать входную строку на основе региональных настроек системы и известных форматов, таких как ISO 8601 и других распространенных форматов.

```
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
```

Поскольку этот метод не принимает информацию о культуре, он использует языковой стандарт системы. Это может привести к неожиданным результатам.

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

Ложь

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

Ложь

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

Правда

Обратите внимание: если вы находитесь в США, вы можете быть удивлены, что результат анализа будет 10 ноября, а не 11 октября.

TryParse (строка, IFormatProvider, DateTimeStyles, outTimeTime)

```
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?");
```

```
}
```

В отличие от метода sibling, эта перегрузка позволяет указать определенную культуру и стиль (ы). Передача `null` для параметра `IFormatProvider` использует культуру системы.

Исключения

Обратите внимание, что этот метод может генерировать исключение при определенных условиях. Они относятся к параметрам, введенным для этой перегрузки: `IFormatProvider` и `DateTimeStyles` .

- `NotSupportedException` : `IFormatProvider` указывает нейтральную культуру
- `ArgumentException` : `DateTimeStyles` не является допустимым параметром или содержит несовместимые флаги, такие как `AssumeLocal` и `AssumeUniversal` .

TryParseExact

Этот метод ведет себя как комбинация `TryParse` и `ParseExact` : он позволяет `ParseExact` настраиваемый формат (ы) и возвращает логический результат, указывающий на успех или неудачу, а не на выброс исключения, если синтаксический анализ не выполняется.

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

Эта перегрузка пытается проанализировать входную строку в определенном формате. Строка ввода должна соответствовать этому формату для анализа.

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

TryParseExact (строка, строка [], IFormatProvider, DateTimeStyles, outTimeTime)

Эта перегрузка пытается разобрать входную строку на массив форматов. Строка ввода должна соответствовать хотя бы одному формату для анализа.

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

Прочитайте Синхронизация DateTime онлайн: https://riptutorial.com/ru/dot-net/topic/58/синхронизация-datetime

глава 50: Словари

Examples

Перечисление словаря

Вы можете перечислить словарь через один из трех способов:

Использование пар 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);
}
```

Использование клавиш

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(int key in dict.Keys)
{
    Console.WriteLine("Key : " + key.ToString() + ", Value : " + dict[key]);
}
```

Использование значений

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(string s in dict.Values)
{
    Console.WriteLine("Value : " + s);
}
```

Инициализация словаря с помощью инициализатора коллекции

```
// 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"
};
```

Добавление в словарь

```
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");
}
```

В качестве альтернативы они могут быть добавлены / установлены с помощью индексатора. (Индексатор внутренне выглядит как свойство, имеющее get и set, но принимает параметр любого типа, который указан между скобками):

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict[1] = "First";
dict[2] = "Second";
dict[3] = "Third";
```

В отличие от метода `Add` который генерирует исключение, если ключ уже содержится в словаре, индексщик просто заменяет существующее значение.

Для использования в потоковом словаре слова `ConcurrentDictionary<TKey, TValue>` :

```
var dict = new ConcurrentDictionary<int, string>();
dict.AddOrUpdate(1, "First", (oldKey, oldValue) => "First");
```

Получение значения из словаря

Учитывая этот код установки:

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};
```

Вы можете прочитать значение для записи с ключом 1. Если ключ не существует, значение будет `KeyNotFoundException` , поэтому вы можете сначала проверить его с помощью `ContainsKey` :

```
if (dict.ContainsKey(1))
    Console.WriteLine(dict[1]);
```

У этого есть один недостаток: вы будете искать через свой словарь дважды (один раз, чтобы проверить наличие и один, чтобы прочитать значение). Для большого словаря это

может повлиять на производительность. К счастью, обе операции могут выполняться вместе:

```
string value;
if (dict.TryGetValue(1, out value))
    Console.WriteLine(value);
```

Сделать словарь с ключами Case-Insensivitve.

```
var MyDict = new Dictionary<string,T>(StringComparison.InvariantCultureIgnoreCase)
```

ConcurrentDictionary (из .NET 4.0)

> Представляет потокобезопасную коллекцию пар ключ / значение, к которым одновременно можно обращаться несколькими потоками.

Создание экземпляра

Создание экземпляра работает почти так же, как с `Dictionary<TKey, TValue>` , например:

```
var dict = new ConcurrentDictionary<int, string>();
```

Добавление или обновление

Вы можете быть удивлены, что нет метода `Add` , но вместо этого есть `AddOrUpdate` с 2 перегрузками:

(1) `AddOrUpdate(TKey key, TValue, Func<TKey, TValue, TValue> addValue)` - *добавляет пару ключ / значение, если ключ еще не существует, или обновляет пару ключ / значение с помощью указанной функции, если ключ уже существует.*

(2) `AddOrUpdate(TKey key, Func<TKey, TValue> addValue, Func<TKey, TValue, TValue> updateValueFactory)` - *использует указанные функции, чтобы добавить пару ключ / значение к ключу, если ключ еще не существует, или обновить пару ключ / значение, если ключ уже существует.*

Добавление или обновление значения, независимо от того, что было значением, если оно уже присутствовало для данного ключа (1):

```
string addedValue = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => "First");
```

Добавление или обновление значения, но теперь изменение значения в обновлении на основе предыдущего значения (1):

```
string addedValue2 = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => $"{valueOld}
Updated");
```

Используя перегрузку (2), мы также можем добавить новое значение с использованием фабрики:

```
string addedValue3 = dict.AddOrUpdate(1, (key) => key == 1 ? "First" : "Not First",
(updateKey, valueOld) => $"{valueOld} Updated");
```

Получение ценности

Получение значения аналогично значению `Dictionary<TKey,TValue>` :

```
string value = null;
bool success = dict.TryGetValue(1, out value);
```

Получение или добавление значения

Есть две перегрузки mehod, которые будут **получать или добавлять** значение поточно-безопасным способом.

Получите значение с помощью ключа 2 или добавьте значение «Второе», если ключ отсутствует:

```
string theValue = dict.GetOrAdd(2, "Second");
```

Использование фабрики для добавления значения, если значение отсутствует:

```
string theValue2 = dict.GetOrAdd(2, (key) => key == 2 ? "Second" : "Not Second." );
```

IEnumerable to Dictionary (≥ .NET 3.5)

Создайте словарь <TKey, TValue> из 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);
```

Удаление из словаря

Учитывая этот код установки:

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};
```

Используйте метод `Remove` для удаления ключа и связанного с ним значения.

```
bool wasRemoved = dict.Remove(2);
```

Выполнение этого кода удаляет ключ `2` и его значение из словаря. `Remove` возвращает логическое значение, указывающее, был ли найден и удален указанный ключ из словаря. Если ключ не существует в словаре, ничего не удаляется из словаря, а возвращается false (исключение не генерируется).

Неправильно пытаться удалить ключ, установив для ключа значение `null` .

```
dict[2] = null; // WRONG WAY TO REMOVE!
```

Это не приведет к удалению ключа. Он просто заменит предыдущее значение на значение `null` .

Чтобы удалить все ключи и значения из словаря, используйте метод `Clear` .

```
dict.Clear();
```

После выполнения `Clear` в словаря `Count` будет 0, а внутренняя емкость остается неизменной.

ContainsKey (TKey)

Чтобы проверить, имеет ли `Dictionary` специальный ключ, вы можете вызвать метод `ContainsKey(TKey)` и предоставить ключ типа `TKey` . Метод возвращает значение `bool` когда ключ существует в словаре. Для образца:

```
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", ... } },
};
```

И проверьте, существует ли `C2` в словаре:

```
if (dictionary.ContainsKey("C2"))
{
   // exists
}
```

Метод ContainsKey доступен в стандартной версии `Dictionary<TKey, TValue>` .

Словарь в список

Создание списка KeyValuePair:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<KeyValuePair<int, int>> list = new List<KeyValuePair<int, int>>();
list.AddRange(dictionary);
```

Создание списка ключей:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Keys);
```

Создание списка значений:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Values);
```

ConcurrentDictionary, дополненный Lazy'1, уменьшает дублирование вычислений

проблема

ConcurrentDictionary сияет, когда дело доходит до моментального возврата существующих ключей из кеша, в основном блокировки и борьбы на гранулированном уровне. Но что, если создание объекта действительно дорого, перевешивает стоимость переключения контекста, и некоторые промахи промахов происходят?

Если один и тот же ключ запрашивается из нескольких потоков, один из объектов,

возникающих в результате встречных операций, в конечном итоге будет добавлен в коллекцию, а остальные будут выброшены, теряя ресурсы ЦП для создания ресурса объекта и памяти для временного хранения объекта , Другие ресурсы также могут быть потрачены впустую. Это действительно плохо.

Решение

Мы можем комбинировать `ConcurrentDictionary<TKey, TValue>` с `Lazy<TValue>` . Идея заключается в том, что метод ConcurrentDictionary GetOrAdd может вернуть значение, фактически добавленное в коллекцию. В этом случае теряются и потери Lazy-объектов, но это не так уж сложно, так как сам Lazy-объект относительно невысок. Свойство Value проигрывающего Lazy никогда не запрашивается, потому что мы умны, чтобы запрашивать только свойство Value того, которое фактически добавлено в коллекцию, - тот, который возвращается из метода 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;
    }
}
```

Кэширование объектов XmlSerializer может быть особенно дорогостоящим, и в старте приложения также много споров. И есть еще кое-что: если это пользовательские сериализаторы, утечка памяти также будет продолжаться до конца жизненного цикла процесса. Единственное преимущество ConcurrentDictionary в этом случае заключается в том, что для остальной части жизненного цикла процесса не будет блокировок, но запуск приложения и использование памяти будут неприемлемыми. Это работа для нашего ConcurrentDictionary, дополненная 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");
```

```
}
```

Прочитайте Словари онлайн: https://riptutorial.com/ru/dot-net/topic/45/словари

глава 51: Сокращение Глоссарий

Examples

.Net Связанные сокращения

Обратите внимание, что некоторые термины, такие как JIT и GC, достаточно универсальны для применения во многих языковых средах программирования и времени автономной работы.

CLR: Common Language Runtime
IL: Промежуточный язык
EE: Механизм выполнения
JIT: компилятор Just-in-time
GC: сборщик мусора
OOM: Недостаточно памяти
STA: Однопоточная квартира
MTA: Многопоточная квартира

Прочитайте Сокращение Глоссарий онлайн: https://riptutorial.com/ru/dot-net/topic/10939/сокращение-глоссарий

глава 52: Стек и куча

замечания

Стоит отметить, что при объявлении ссылочного типа его начальное значение будет равно `null` . Это связано с тем, что он еще не указывает на местоположение в памяти и является совершенно правильным состоянием.
Однако, за исключением типов с нулевым значением, типы значений обычно должны иметь значение.

Examples

Используемые типы значений

Типы значений просто содержат ***значение*** .

Все типы значений производятся из класса System.ValueType , и это включает большинство встроенных типов.

При создании нового типа значения используется область памяти, называемая ***стеком*** . Соответственно, стек будет расти по размеру объявленного типа. Так, например, int всегда будет выделено 32 бита памяти в стеке. Когда тип значения больше не находится в области видимости, пространство в стеке будет освобождено.

В приведенном ниже коде показано, что тип значения присваивается новой переменной. Структура используется как удобный способ создания пользовательского типа значений (класс System.ValueType не может быть иным образом расширен).

Важно понять, что при назначении типа значения само значение ***копируется*** в новую переменную, то есть мы имеем два разных экземпляра объекта, которые не могут влиять друг на друга.

```
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'
    }
}
```

Используемые типы ссылок

Типы ссылок состоят как из ***ссылки*** на область памяти, так и из ***значения,*** хранящегося в этой области.
Это аналогично указателям на C / C ++.

Все ссылочные типы хранятся на так называемой ***куче*** .
Куча - это просто управляемая область памяти, где хранятся объекты. Когда создается новый объект, часть кучи будет выделена для использования этим объектом, и будет возвращена ссылка на это место кучи. Куча управляется и поддерживается *сборщиком мусора* и не допускает ручного вмешательства.

Помимо пространства памяти, необходимого для самого экземпляра, требуется дополнительное пространство для хранения самой ссылки, а также дополнительная временная информация, требуемая .NET CLR.

В приведенном ниже коде показано, что ссылочный тип присваивается новой переменной. В этом случае мы используем класс, все классы являются ссылочными типами (даже если они статичны).

Когда ссылочный тип присваивается другой переменной, это ***ссылка*** на объект, который скопирован, а **не** само значение. Это важное различие между типами значений и ссылочными типами.

Последствия этого заключаются в том, что теперь у нас есть *две* ссылки на один и тот же объект.
Любые изменения значений внутри этого объекта будут отражаться обеими переменными.

```
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'
    }
```

Прочитайте Стек и куча онлайн: https://riptutorial.com/ru/dot-net/topic/9358/стек-и-куча

глава 53: Струны

замечания

В .NET-строках `System.String` - это последовательность символов `System.Char` , каждый символ - кодированный код UTF-16. Это различие важно , потому что *говорят* определение *языка характера* и .NET (и многих других языков) определение характера различны.

Один *символ* , который должен быть правильно назван grapheme , отображается в виде глифа и определяется одним или несколькими кодовыми точками Unicode. Каждая кодовая точка затем кодируется в последовательности блоков кода . Теперь должно быть понятно, почему один `System.Char` не всегда представляет собой графем, давайте посмотрим в реальном мире, как они отличаются:

- Одна графема из-за сочетания символов может приводить к двум или более кодовым точкам: `à` состоит из двух кодовых точек: *U + 0061 LATIN SMALL LETTER A* и *U + 0300 COMBINING GRAVE ACCENT* . Это самая распространенная ошибка, потому что `"à ".Length == 2` то время как вы можете ожидать `1` .
- Существуют дублированные символы, например, `à` может быть одной кодовой точкой *U + 00E0 ЛАТИНСКОЕ МАЛОЕ ПИСЬМО A C GRAVE* или двумя кодовыми точками, как описано выше. Очевидно, что они должны сравнивать одно и то же: `"\u00e0" == "\u0061\u0300"` (даже если `"\u00e0".Length != "\u0061\u0300".Length`). Это возможно из-за *нормализации строки,* выполняемой методом `String.Normalize()` .
- Последовательность Unicode может содержать *сгруппированную* или разложенную последовательность, например символ □ *U + D55C HAN CHARACTER* может быть одной кодовой точкой (закодированной как единый блок кода в UTF-16) или разложенной последовательностью его слогов □ , □ и □ . Их нужно сравнивать равными.
- Одна кодовая точка может быть закодирована в несколько кодовых единиц: символ □ *U + 2008A HAN CHARACTER* кодируется как два `System.Char` (`"\ud840\udc8a"`), даже если это всего лишь одна кодовая точка: UTF-16 кодировка не фиксированный размер! Это источник бесчисленных ошибок (также серьезных ошибок безопасности), если, например, ваше приложение применяет максимальную длину и слепо обрезает строку, тогда вы можете создать недопустимую строку.
- Некоторые языки имеют орграф и триграфы, например , в Чехии `ч` является автономным письмом (после `часов` и , прежде чем `я` тогда при заказе списка строк вы будете иметь *fyzika* перед тем *Chemie.*

Есть гораздо больше проблем с обработкой текста, см., Например, Как я могу использовать символ Unicode для сравнения символов? для более широкого введения и дополнительных ссылок на соответствующие аргументы.

В общем случае при работе с *международным* текстом вы можете использовать эту простую функцию для перечисления текстовых элементов в строке (избегая прерывания суррогатов и кодирования 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

Подсчет отдельных символов

Если вам нужно подсчитать разные символы, то по причинам, описанным в разделе « *Примечания* », вы не можете просто использовать свойство `Length` потому что это длина массива `System.Char` которые не являются символами, а единицами кода (а не кодами Unicode ни графемы). Если, например, вы просто пишете `text.Distinct().Count()` вы получите неверные результаты, исправьте код:

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

Еще один шаг - **подсчет вхождений каждого символа** , если производительность не является проблемой, вы можете просто сделать это так (в этом примере, независимо от случая):

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

Количество символов

Если вам нужно подсчитать *символы,* то по причинам, описанным в разделе « *Замечания* », вы не можете просто использовать свойство «Длина», потому что это длина массива `System.Char` которые не являются символами, а единицами кода (а не кодами Unicode и графемы). Правильный код:

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

Небольшая оптимизация может переписать метод расширения `EnumerateCharacters()`

специально для этой цели:

```
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;
    }
}
```

Количество экземпляров символа

Из-за причин, объясненных в разделе « *Замечания* », вы не можете просто сделать это (если вы не хотите подсчитывать вхождения определенного блока кода):

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

Вам нужна более сложная функция:

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

Обратите внимание, что сравнение строк (в отличие от сравнения символов, которое является инвариантом культуры) всегда должно выполняться в соответствии с правилами конкретной культуре.

Разделить строку на блоки фиксированной длины

Мы не можем разбить строку на произвольные точки (потому что `System.Char` может быть недействительным в одиночку, потому что это комбинационный символ или часть суррогата), тогда код должен учитывать это (обратите внимание, что с *длиной* я имею в виду количество *графем, а* не количество *кодовых единиц*):

```
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;
    }
}
```

Преобразование строки в / из другой кодировки

Строки .NET содержат `System.Char` (кодовые модули UTF-16). Если вы хотите сохранить (или управлять) текст с другой кодировкой, вам придется работать с массивом `System.Byte` .

Преобразования выполняются по классам , полученных из `System.Text.Encoder` и `System.Text.Decoder` , которые вместе могут конвертировать в / из другого кодирования (от байта *X* закодированного массива `byte[]` к UTF-16 закодированных `System.String` и порока - versa).

Поскольку кодировщик / декодер обычно работает очень близко друг к другу, они группируются вместе в классе, производном от `System.Text.Encoding` , производные классы предлагают преобразования в / из популярных кодировок (UTF-8, UTF-16 и т. Д.).

Примеры:

Преобразование строки в UTF-8

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

Преобразование данных UTF-8 в строку

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

Изменить кодировку существующего текстового файла

Этот код будет считывать содержимое текстового файла с кодировкой UTF-8 и сохранять его обратно в кодировке UTF-16. Обратите внимание, что этот код не является оптимальным, если файл большой, потому что он будет считывать весь его контент в память:

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

Виртуальный метод Object.ToString ()

Все в .NET является объектом, поэтому каждый тип имеет метод `ToString()` определенный в классе `Object` который может быть переопределен. По умолчанию реализация этого метода возвращает имя типа:

```
public class Foo
{
}

var foo = new Foo();
Console.WriteLine(foo); // outputs Foo
```

`ToString()` неявно вызывается при объединении значения со строкой:

```
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
```

Результат этого метода также широко используется средствами отладки. Если по какой-то причине вы не хотите переопределять этот метод, но хотите настроить, как отладчик показывает значение вашего типа, используйте атрибут 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;}
    // ...
}
```

Неизменяемость струн

Строки неизменяемы. Вы просто не можете изменить существующую строку. Любая операция над строкой создает новый экземпляр строки, имеющей новое значение. Это означает, что если вам нужно заменить один символ в очень длинной строке, память будет выделена для нового значения.

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

Если вам нужно выполнить множество операций со строковым значением, используйте класс `StringBuilder` который предназначен для эффективной обработки строк:

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

Сочетание строк

Несмотря на то, что `String` является ссылочным типом `==` оператор сравнивает строковые значения, а не ссылки.

Как вы знаете, `string` - это всего лишь массив символов. Но если вы считаете, что проверка равенства строк и сравнение производится персонажем по характеру, вы ошибаетесь. Эта операция специфична для культуры (см. Примечания ниже): некоторые последовательности символов можно рассматривать как равные в зависимости от культуры .

Подумайте дважды перед коротким замыканием проверки равенства, сравнив свойства `Length` двух строк!

Используйте перегрузки метода `String.Equals` которые принимают дополнительное значение перечисления `StringComparison` , если вам нужно изменить поведение по умолчанию.

Прочитайте Струны онлайн: https://riptutorial.com/ru/dot-net/topic/2227/струны

глава 54: Упаковочная система NuGet

замечания

NuGet.org :

> NuGet - это менеджер пакетов для платформы разработки Microsoft, включая . NET. Клиентские инструменты NuGet предоставляют возможность производить и потреблять пакеты. Галерея NuGet - это центральный репозиторий пакетов, используемый всеми авторами и потребителями пакетов.

Изображения в примерах предоставлены NuGet.org .

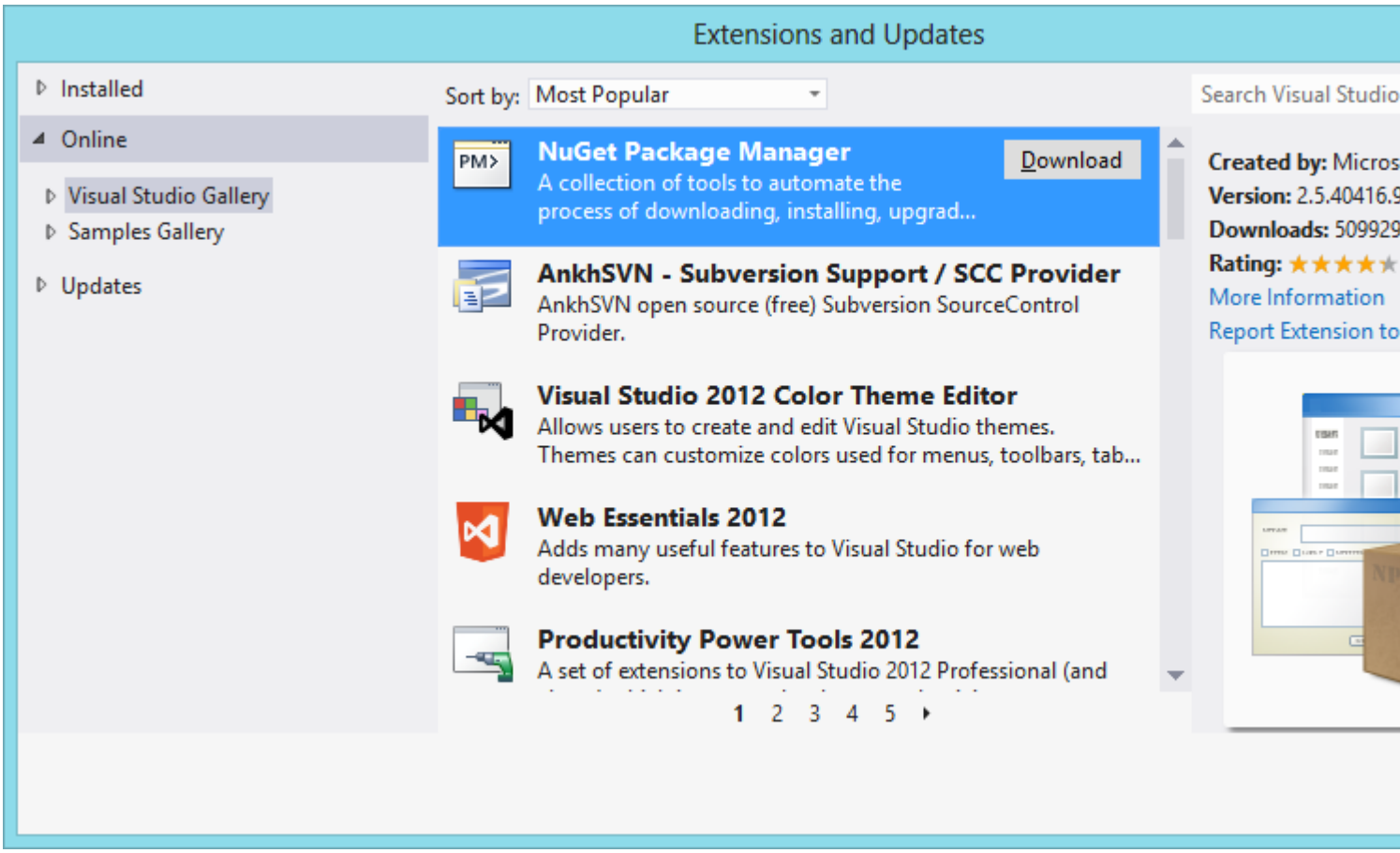
Examples

Установка диспетчера пакетов NuGet

Чтобы иметь возможность управлять пакетами ваших проектов, вам нужен диспетчер пакетов NuGet. Это расширение Visual Studio, описанное в официальных документах: Установка и обновление NuGet Client .

Начиная с Visual Studio 2012, NuGet входит в каждую редакцию и может быть использован из: Tools -> NuGet Package Manager -> Package Manager Console.

Вы делаете это через меню «Инструменты» Visual Studio, нажав «Расширения и обновления»:



Это устанавливает как GUI:

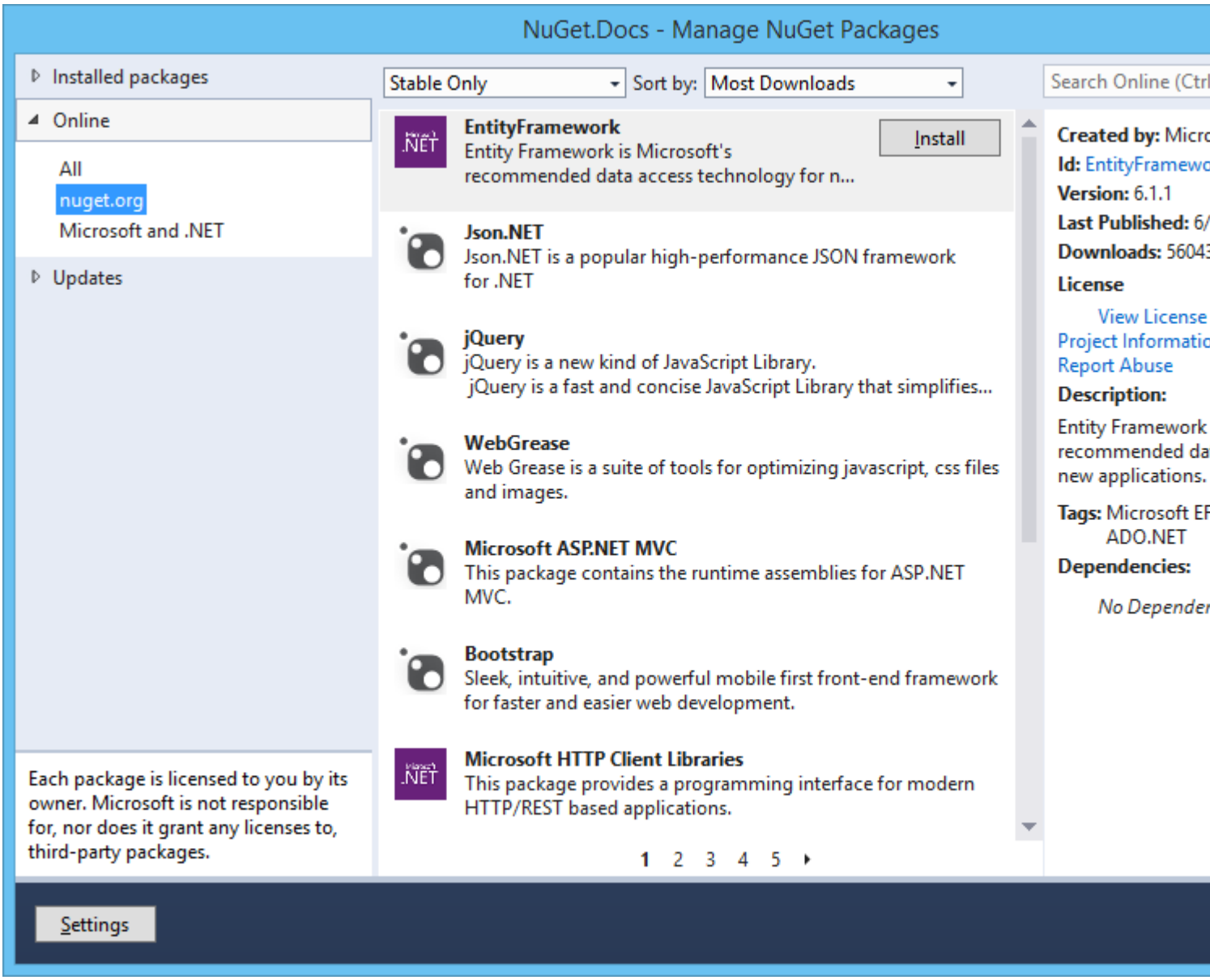
- Доступно с помощью нажатия «Управление пакетами NuGet ...» в проекте или в папке «Ссылки»

И консоль диспетчера пакетов:

- Инструменты -> Диспетчер пакетов NuGet -> Консоль диспетчера пакетов.

Управление пакетами через пользовательский интерфейс

Когда вы щелкните правой кнопкой мыши проект (или его папку «Ссылки»), вы можете нажать «Управление пакетами NuGet ...». Появится диалоговое окно диспетчера пакетов .



Управление пакетами через консоль

Нажмите меню «Инструменты» -> «Диспетчер пакетов NuGet» -> «Диспетчер пакетов», чтобы отобразить консоль в вашей среде IDE. Официальная документация здесь .

Здесь вы можете указать, в частности, команды `install-package` которые устанавливают введенный пакет в текущий выбранный «Проект по умолчанию»:

```
Install-Package Elmah
```

Вы также можете предоставить проект для установки пакета, переопределив выбранный проект в раскрывающемся меню «Проект по умолчанию»:

```
Install-Package Elmah -ProjectName MyFirstWebsite
```

Обновление пакета

Чтобы обновить пакет, используйте следующую команду:

```
PM> Update-Package EntityFramework
```

где EntityFramework - это имя пакета для обновления. Обратите внимание, что обновление будет выполняться для всех проектов, и оно отличается от `Install-Package EntityFramework` которое будет установлено только в «Проект по умолчанию».

Вы также можете явно указать один проект:

```
PM> Update-Package EntityFramework -ProjectName MyFirstWebsite
```

Удаление пакета

```
PM> Uninstall-Package EntityFramework
```

Удаление пакета из одного проекта в решении

```
PM> Uninstall-Package -ProjectName MyProjectB EntityFramework
```

Установка конкретной версии пакета

```
PM> Install-Package EntityFramework -Version 6.1.2
```

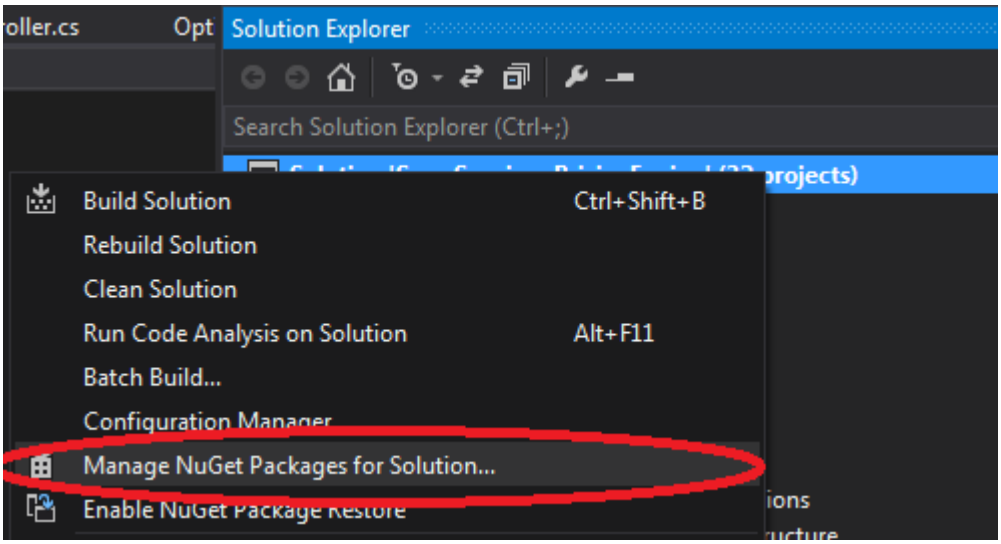
Добавление источника исходного кода пакета (MyGet, Klondike, ect)

```
nuget sources add -name feedname -source http://sourcefeedurl
```

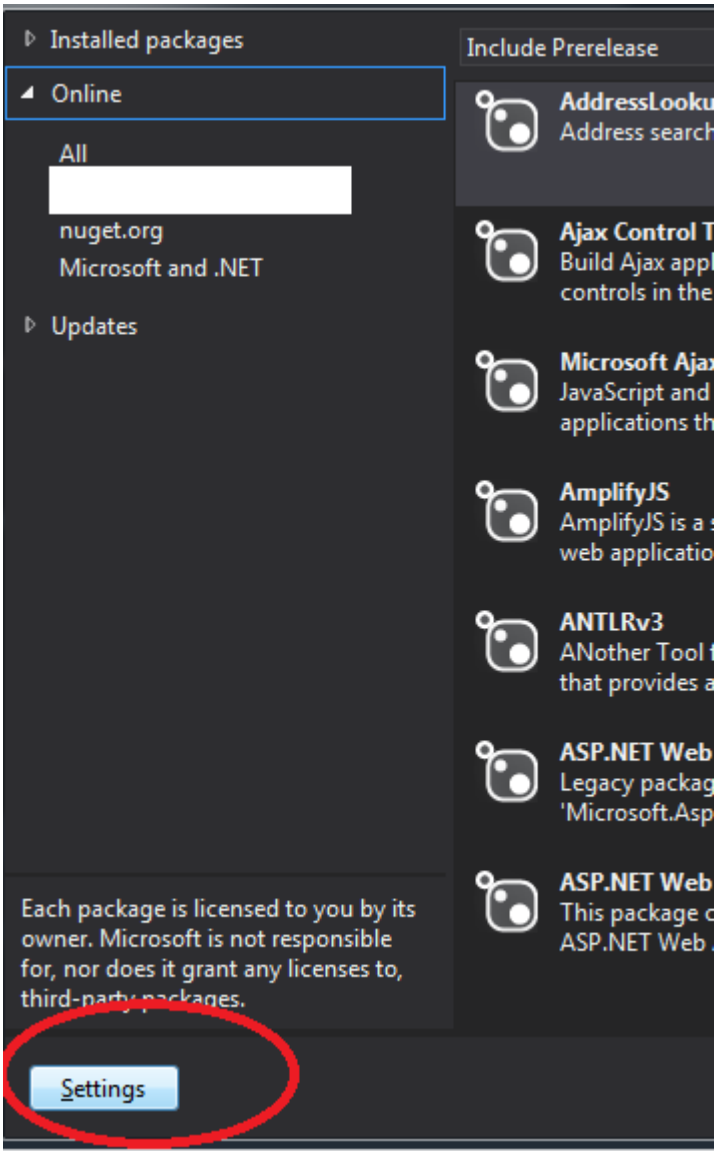
Использование различных (локальных) источников пакета Nuget с использованием пользовательского интерфейса

Обычно компания настраивает собственный сервер nuget для распространения пакетов в разных командах.

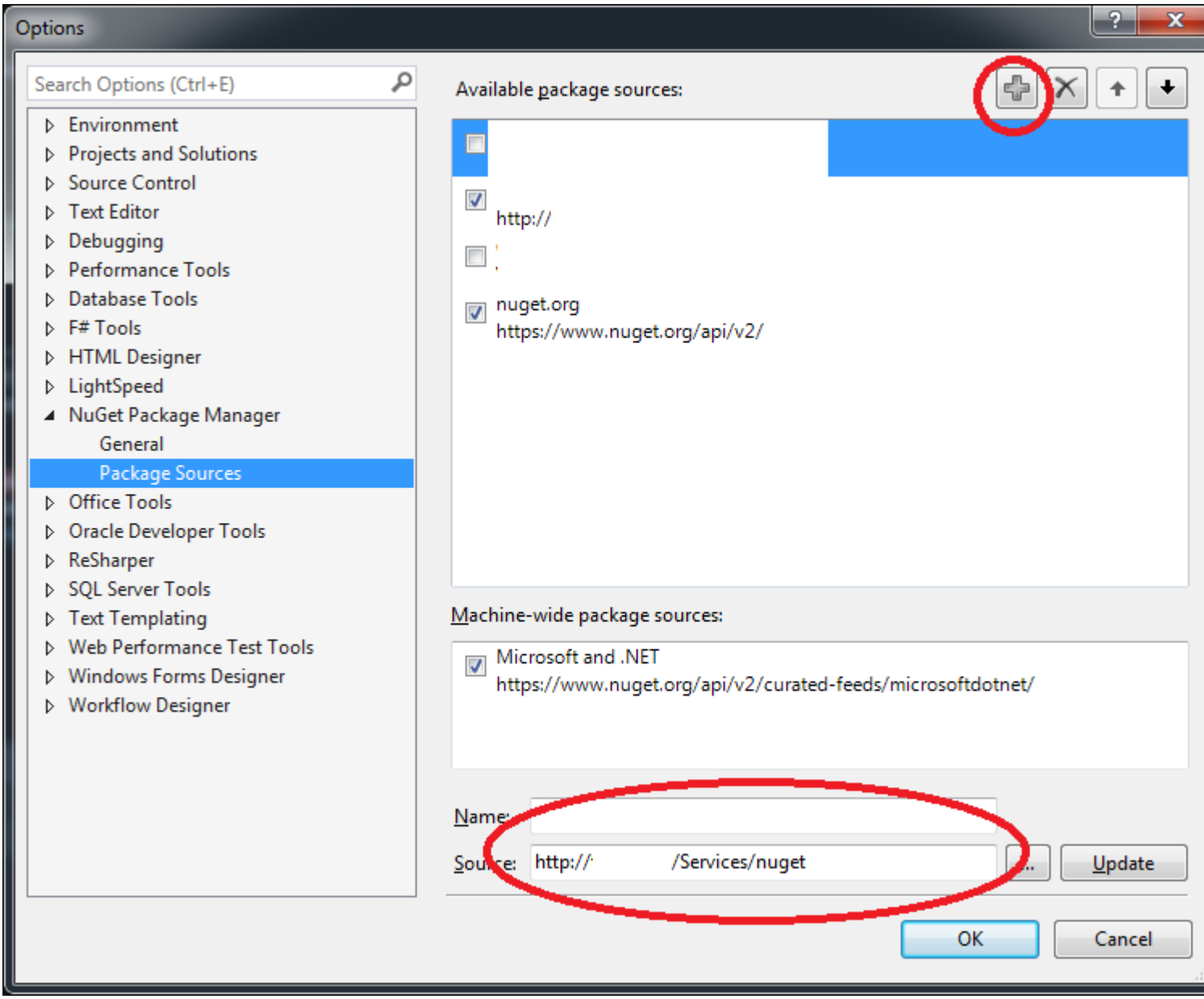
1. Перейдите в Обозреватель решений и нажмите `правую` кнопку `мыши` , затем выберите « `Manage NuGet Packages for Solution`



2. В открывшемся окне нажмите « `Settings`



3. Нажмите `+` в верхнем правом углу, затем добавьте имя и URL-адрес, указывающий на ваш локальный сервер nuget.



удалить определенную версию пакета

```
PM> uninstall-Package EntityFramework -Version 6.1.2
```

Прочитайте Упаковочная система NuGet онлайн: https://riptutorial.com/ru/dot-net/topic/43/упаковочная-система-nuget

глава 55: Управление памятью

замечания

Процессы, зависящие от производительности в управляемых приложениях .NET, могут серьезно пострадать от GC. Когда GC работает, все остальные потоки приостанавливаются до завершения. По этой причине рекомендуется тщательно оценить процессы ГХ и определить, как свести к минимуму при ее запуске.

Examples

Неуправляемые ресурсы

Когда мы говорим о GC и «куче», мы действительно говорим о том, что называется *управляемой кучей* . Объекты в *управляемой куче* могут обращаться к ресурсам не на управляемой куче, например, при записи или чтении из файла. Неожиданное поведение может возникать, когда файл открывается для чтения, а затем возникает исключение, предотвращая закрытие дескриптора файла, как обычно. По этой причине .NET требует, чтобы неуправляемые ресурсы реализовали интерфейс `IDisposable` . Этот интерфейс имеет один метод `Dispose` без параметров:

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

При работе с неуправляемыми ресурсами вы должны убедиться, что они правильно настроены. Вы можете сделать это, явно вызвав `Dispose()` в блоке `finally` или с `using` оператора `using` .

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

или же

```
string textFromFile;
```

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

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

Последний является предпочтительным методом и автоматически расширяется до первого во время компиляции.

Использовать SafeHandle при обертке неуправляемых ресурсов

При написании оберток для неуправляемых ресурсов вы должны подклассифицировать `SafeHandle` а не пытаться самостоятельно реализовать `IDisposable` и финализатор. Подкласс `SafeHandle` должен быть как можно более малым и простым, чтобы свести к минимуму вероятность утечки рукоятки. Вероятно, это означает, что ваша реализация SafeHandle будет внутренней деталью реализации класса, который обертывает его, чтобы предоставить полезный API. Этот класс гарантирует, что даже если программа утечки вашего экземпляра `SafeHandle` , ваш неуправляемый дескриптор будет выпущен.

```
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;
    }
}
```

Отказ от ответственности. Этот пример представляет собой попытку показать, как защитить управляемый ресурс с помощью `SafeHandle` который реализует `IDisposable` для вас и соответствующим образом настраивает финализаторы. Это очень надуманно и, вероятно, бессмысленно выделять кусок памяти таким образом.

Прочитайте Управление памятью онлайн: https://riptutorial.com/ru/dot-net/topic/59/управление-памятью

глава 56: Управляемая расширяемость

замечания

Одним из больших преимуществ MEF по сравнению с другими технологиями, поддерживающими шаблон инверсии управления, является то, что он поддерживает разрешение зависимостей, которые неизвестны во время разработки, без необходимости в значительной (если есть) конфигурации.

Все примеры требуют ссылки на сборку System.ComponentModel.Composition.

Кроме того, все (основные) примеры используют их в качестве примерных бизнес-объектов:

```
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

Экспорт типа (Basic)

```
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();
        }
    }
}
```

Это можно определить практически в любом месте; все, что имеет значение, - это то, что приложение знает, где его искать (через создаваемые им ComposablePartCatalogs).

Импорт (основной)

```
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);
            }
        }
    }
}
```

Это тип, который зависит от `IUserProvider` , который может быть определен где угодно. Как и в предыдущем примере, все, что имеет значение, - это то, что приложение знает, где искать соответствующий экспорт (через создаваемые ComposablePartCatalogs).

Подключение (базовое)

См. Другие (основные) примеры выше.

```
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();
            }
        }
    }
}
```

До тех пор, пока что-то в пути поиска сборки приложения имеет `[Export(typeof(IUserProvider))]` , соответствующий импорт `UserWriter` будет выполнен, и пользователи будут напечатаны.

Другие типы каталогов (например, `DirectoryCatalog`) могут использоваться вместо (или в дополнение к) `ApplicationCatalog` для просмотра в других местах для экспорта, которые удовлетворяют им.

Прочитайте Управляемая расширяемость онлайн: https://riptutorial.com/ru/dot-net/topic/62/управляемая-расширяемость

глава 57: Формы VB

Examples

Hello World в форматах VB.NET

Чтобы показать окно сообщения, когда была показана форма:

```
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 () будет вызываться сначала, и только один раз, когда форма сначала загружается. Show () будет вызываться каждый раз, когда пользователь запускает форму. Активировать () будет вызываться каждый раз, когда пользователь делает форму активной.

Load () будет выполняться до вызова Show (), но будет предупрежден: вызов msgBox () в show может привести к тому, что msgBox () будет выполнен до завершения Load (). **Как правило, плохая идея зависит от упорядочения событий между Load (), Show () и т. Д.**

Для начинающих

Некоторые вещи, которые все новички должны знать / делать, помогут им хорошо начать с VB .Net:

Установите следующие параметры:

```
'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
```

Используйте &, not + для конкатенации строк. Строки следует изучать в деталях,

поскольку они широко используются.

Потратьте некоторое время на понимание значений и ссылочных типов .

Никогда не используйте Application.DoEvents . Обратите внимание на «Предостережение». Когда вы достигнете точки, где это кажется чем-то, что вы должны использовать, спросите.

Документация - ваш друг.

Таймер форм

Компонент Windows.Forms.Timer может использоваться для предоставления информации пользователя, которая **не** критична по времени. Создайте форму с помощью одной кнопки, одной метки и компонента таймера.

Например, он может использоваться, чтобы периодически показывать пользователю время суток.

```
'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
```

Но этот таймер не подходит для синхронизации. Пример будет использовать его для обратного отсчета. В этом примере мы будем моделировать обратный отсчет до трех минут. Это вполне может быть одним из самых скучно важных примеров здесь.

```
'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
```

После нажатия кнопки1 происходит переход на три минуты, а label1 показывает результаты. Является ли label1 показать 180? Возможно нет. На моей машине он показал 182,5!

Причина несоответствия в документации: «Компонент Timer Windows Forms является однопоточным и ограничен точностью до 55 миллисекунд». Вот почему он не должен использоваться для синхронизации.

Используя таймер и секундомер немного иначе, мы можем получить лучшие результаты.

```
'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
```

Существуют другие таймеры, которые можно использовать по мере необходимости. Этот поиск должен помочь в этом отношении.

Прочитайте Формы VB онлайн: https://riptutorial.com/ru/dot-net/topic/2197/формы-vb

глава 58: Чтение и запись Zip-файлов

Вступление

Класс **ZipFile** живет в пространстве имен **System.IO.Compression** . Его можно использовать для чтения и записи в Zip-файлы.

замечания

- Вы также можете использовать MemoryStream вместо FileStream.
- Исключения

исключение	Состояние
ArgumentException	Поток уже закрыт или возможности потока не соответствуют режиму (например: попытка записи в поток только для чтения)
ArgumentNullException	входной *поток* равен нулю
ArgumentOutOfRangeException	*режим* имеет недопустимое значение
InvalidDataException	См. Список ниже

Когда **выбрано InvalidDataException** , оно может иметь 3 причины:

- Содержимое потока не может быть интерпретировано как zip-архив
- *mode* - это обновление, и запись отсутствует в архиве или повреждена и не может быть прочитана
- *mode* - обновление, и запись слишком велика, чтобы вписаться в память

Вся информация была взята с этой страницы MSDN

Examples

Отображение содержимого ZIP

В этом фрагменте будут перечислены все имена файлов zip-архива. Имена файлов относятся к корню 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]}");
    }
}
```

Извлечение файлов из ZIP-файлов

Извлечение всех файлов в каталог очень просто:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    archive.ExtractToDirectory(AppDomain.CurrentDomain.BaseDirectory);
}
```

Когда файл уже существует, будет **выведено** сообщение **System.IO.IOException** .

Извлечение определенных файлов:

```
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);
}
```

Любой из этих методов даст тот же результат.

Обновление ZIP-файла

Чтобы обновить ZIP-файл, файл должен быть открыт с помощью ZipArchiveMode.Update.

```
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");
}
```

Существует также возможность записи непосредственно в файл в архиве:

```
var entry = archive.CreateEntry("createentry.txt");
using(var writer = new StreamWriter(entry.Open()))
{
    writer.WriteLine("Test line");
}
```

Прочитайте Чтение и запись Zip-файлов онлайн: https://riptutorial.com/ru/dot-net/topic/9943/чтение-и-запись-zip-файлов

глава 59: Шифрование / криптография

замечания

.NET Framework обеспечивает реализацию многих криптографических алгоритмов. Они включают в себя в основном симметричные алгоритмы, асимметричные алгоритмы и хеши.

Examples

RijndaelManaged

Требуемое пространство имен: `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;
          }
        }
```

использование

```
var textToEncrypt = "hello World";

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

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

Замечания:

- Rijndael является предшественником стандартного симметричного криптографического алгоритма AES.

Шифровать и расшифровывать данные с использованием AES (в 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;
    }
}
}
```

Этот пример из MSDN .

Это консольное демонстрационное приложение, показывающее, как шифровать строку, используя стандартное шифрование **AES** , и как расшифровать ее впоследствии.

(**AES = Advanced Encryption Standard ,** спецификация для шифрования электронных данных, установленная Национальным институтом стандартов и технологий США (NIST) в 2001 году, которая по-прежнему является стандартом де-факто для симметричного шифрования)

Заметки:

- В реальном сценарии шифрования вам необходимо выбрать правильный режим шифрования (его можно присвоить свойству « `Mode` », выбрав значение из перечисления `CipherMode`). **Никогда не** используйте `CipherMode.ECB` (режим электронной кодовой книги), так как это обеспечивает слабый поток шифров
- Чтобы создать хороший (а не слабый) `Key` , используйте криптографический случайный генератор или используйте пример выше (**Create a Key from** the **Password**). Рекомендуемый **KeySize** - 256 бит. Поддерживаемые размеры ключей доступны через свойство `LegalKeySizes` .
- Чтобы инициализировать вектор инициализации `IV` , вы можете использовать СОЛЬ, как показано в примере выше (**Random SALT**)
- Поддерживаемые размеры блоков доступны через свойство `SupportedBlockSizes` , размер блока может быть назначен через свойство `BlockSize`

Использование: см. Метод Main ().

Создайте ключ из пароля / случайного СОЛЬ (в 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
}
```

Этот пример взят из MSDN.

Это демонстрация консоли, в которой показано, как создать защищенный ключ на основе пользовательского пароля и как создать произвольную OCB на основе криптографического генератора.

Заметки:

- Встроенная функция `PasswordDeriveBytes` использует стандартный алгоритм PBKDF1 для генерации ключа из пароля. По умолчанию он использует 100 итераций для генерации ключа, чтобы замедлить атаки грубой силы. Соль, генерируемая случайным образом, еще более укрепляет ключ.

- Функция `CryptDeriveKey` преобразует ключ, сгенерированный `PasswordDeriveBytes` в ключ, совместимый с указанным алгоритмом шифрования (здесь «TripleDES»), используя указанный хэш-алгоритм (здесь «SHA1»). Ключ в этом примере составляет 192 байта, а вектор инициализации IV берется у поставщика криптографии с тройным DES

- Обычно этот механизм используется для защиты более сильного случайного сгенерированного ключа паролем, который шифрует большой объем данных. Вы также можете использовать его для предоставления нескольких паролей разных пользователей для доступа к тем же данным (их защищает другой случайный ключ).

- К сожалению, `CryptDeriveKey` настоящее время не поддерживает AES. См. Здесь. **ПРИМЕЧАНИЕ.** В качестве обходного пути вы можете создать случайный ключ AES для шифрования данных, которые должны быть защищены AES, и сохранить ключ AES в TripleDES-контейнере, который использует ключ, сгенерированный `CryptDeriveKey` . Но это ограничивает безопасность TripleDES, не использует преимущества больших ключей AES и создает зависимость от TripleDES.

Использование: см. Метод Main ().

Шифрование и дешифрование с использованием криптографии (AES)

Код дешифрования

```
    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;
    }
```

Код шифрования

```
    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;
    }
```

использование

```
var textToEncrypt = "TestEncrypt";

var encrypted = Encrypt(textToEncrypt);

var decrypted = Decrypt(encrypted);
```

Прочитайте Шифрование / криптография онлайн: https://riptutorial.com/ru/dot-net/topic/7615/шифрование---криптография

кредиты

S. No	Главы	Contributors
1	Начало работы с .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	ADO.NET	Akshay Anand, Andrew Morton, Daniel A. White, DavidG, Drew, elmer007, Hamid, Harjot, Heinzi, Igor, user2321864
4	CLR	Gajendra, starbeamrainbowlabs, Theodoros Chatzigiannakis
5	HTTP-клиенты	CodeCaster, Konamiman, MuiBienCarlota
6	HTTP-серверы	Devon Burriss, Konamiman
7	JIT-компилятор	Krikor Ailanjian
8	JSON в .NET с Newtonsoft.Json	DLeh
9	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
10	ReadOnlyCollections	tehDorf
11	System.Diagnostics	Adi Lester, Bassie, Fredou, Ogglas, Ondřej Štorc, RamenChef
12	System.IO	CodeCaster, Daniel A. White, demonplus, Filip Frącz, RoyalPotato
13	System.Net.Mail	demonplus, Steve, vicky
14	System.Runtime.Caching.MemoryCache (ObjectCache)	Guanxi, RamenChef
15	XmlSerializer	Aphelion, George Polevoy, RamenChef, Rowland Shaw, Thomas Levesque, void, Yogi
16	Ввод / вывод файлов	ale10ander, Alexander Mandt, Ingenioushax, Nitram
17	Внедрение зависимости	Phil Thomas, Scott Hannen
18	Вывоз мусора	avat
19	Вызов платформы	Dmitry Egorov, Imran Ali Khan
20	Глобализация в ASP.NET MVC с использованием интеллектуальной интернационализации для ASP.NET	Scott Hannen
21	Деревья выражений	Akshay Anand, George Polevoy, Jim, n.podbielski, Pavel Mayorov, RamenChef, Stephen Leppik, Stilgar, wangengzheng
22	Для каждого	Dr Rob Lang, just.ru, Lucas Trzesniewski
23	Единичное тестирование	Axarydax
24	Загрузка файлов и данных POST на веб-сервер	Aleks Andreev
25	Запись и чтение из потока StdErr	Aleks Andreev
26	Исключения	Adi Lester, Akshay Anand, Alan McBee, Alfred Myers, Arvin Baccay, BananaSft, CodeCaster, Dave R., Kritner, Mafii, Matt,

		Rob, Sean, starbeamrainbowlabs, STW, Yousef Al-Mulla
27	Использование прогресса и IProgress	DLeh
28	Класс SpeechRecognitionEngine для распознавания речи	ProgramFOX, RamenChef
29	Класс System.IO.File	Adriano Repetti, delete me
30	Кодовые контракты	JJS, Matthew Whited, RamenChef
31	Коллекции	Alan McBee, Aman Sharma, Anik Saha, Daniel A. White, demonplus, Felipe Oriani, harriyott, Ian, Mark C., Ravi A., Virtlink
32	Контексты синхронизации	DLeh, Gusdor
33	Многопоточность	Behzad, Martijn Pieters, Mellow
34	Настройка привязки процесса и потока	MSE, RamenChef
35	настройки	Alan McBee
36	Обзор параллельной библиотеки задач (TPL)	Gusdor, Jacobr365
37	отражение	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
38	Параллельная библиотека задач (TPL)	Adi Lester, Aman Sharma, Andrew, i3arnon, Jacobr365, JamyRyals, Konamiman, Mathias Müller, Mert Gülsoy, Mikhail Filimonov, Pavel Mayorov, Pavel Voronin, RamenChef, Thomas Bledsoe, TorbenJ
39	Параллельная обработка с использованием .Net framework	Yahfoufi
40	Пользовательские типы	Alan McBee, DrewJordan, matteeyah
41	Последовательные порты	Dmitry Egorov
42	Поток данных TPL	i3arnon, Jacobr365, Nikola.Lukovic, RamenChef

43	Пространство имен System.Reflection.Emit	Luaan, NikolayKondratyev, RamenChef, toddmo
44	Работа с SHA1 в C #	mahdi abasi
45	Регулярные выражения (System.Text.RegularExpressions)	BrunoLM, Denuath, Matt dc, tehDorf
46	Сериализация JSON	Akshay Anand, Andrius, Eric, hasan, M22an, PedroSouki, Thriggle, Tolga Evcimen
47	сетей	Konamiman
48	Синхронизация DateTime	GalacticCowboy, John
49	Словари	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
50	Сокращение Глоссарий	Tanveer Badar
51	Стек и куча	Hywel Rees
52	Струны	Adriano Repetti, Alexander Mandt, Matt, Pavel Voronin, RamenChef
53	Упаковочная система NuGet	Andrey Shchekin, Anik Saha, Ashtonian, CodeCaster, Daniel A. White, Matas Vaitkevicius, Ozair Kafray
54	Управление памятью	Big Fan, binki, DrewJordan
55	Управляемая расширяемость	Joe Amenta, Kirk Broadhurst, RamenChef
56	Формы VB	ale10ander, dbasnett
57	Чтение и запись Zip-файлов	Arxae
58	Шифрование / криптография	Alexander Mandt, Daniel A. White, demonplus, Jagadisha B S, lokusking, Matt