TPL (Task Paralrallel Library)

Task

// Метод который будет выполнен как задача.

static void MyTask()

{

Console.WriteLine("MyTask() запущен.");

for (int count = 0; count < 20; count++)

{

Thread.Sleep(500);

Console.WriteLine("В методе MyTask(), счетчик равен: " + count);

}

Console.WriteLine("MyTask() завершен.");

}

static void Main()

{

Console.WriteLine("Основной поток запущен.");

var action = new Action(MyTask);

// Создание экземпляра задачи.

var task = new Task(action);

// Выполнение задачи.

task.Start();

// Метод Main() остается активным до завершения задачи MyTask().

for (int i =0; i < 60; i++)

{

Console.Write(".");

Thread.Sleep(100);

}

Console.WriteLine("Основной поток завершен.");

\*\*\*

Id Curent ID

// Id - Получает уникальный идентификатор данного экземпляра Task.

// CurrentId - Возвращает уникальный идентификатор выполняющейся в настоящее время задачи Task.

namespace TPL

{

class Program

{

// Метод который будет выполнен как задача.

static void MyTask()

{

Console.WriteLine("MyTask() # " + Task.CurrentId + " запущен.");

for (int count = 0; count < 10; count++)

{

Thread.Sleep(500);

Console.WriteLine("В методе MyTask() #" + Task.CurrentId + " счетчик равен: " + count);

}

Console.WriteLine("MyTask() # " + Task.CurrentId + " завершен.");

}

static void Main()

{

Console.WriteLine("Основной поток запущен.");

var task1 = new Task(MyTask);

var task2 = new Task(MyTask);

// Выполнение задач.

task1.Start();

task2.Start();

Console.WriteLine("Идентификатор задачи task1: " + task1.Id);

Console.WriteLine("Идентификатор задачи task2: " + task2.Id);

// Метод Main() остается активным до завершения задачи MyTask().

for (int i = 0; i < 60; i++)

{

Console.Write(".");

Thread.Sleep(100);

}

Console.WriteLine("Основной поток завершен. Id основной задчи равен null? - {0}.", Task.CurrentId == null);

// Delay.

Console.ReadKey();

}

\*\*\*

Task.Wait() – ждет завершения фоновых потоков

static void Main()

{

Console.WriteLine("Основной поток запущен.");

var task1 = new Task(MyTask);

var task2 = new Task(MyTask);

// Выполнение задач.

task1.Start();

task2.Start();

Console.WriteLine("Идентификатор задачи task1: " + task1.Id);

Console.WriteLine("Идентификатор задачи task2: " + task2.Id);

// Метод Main() остается активным до завершения задачи MyTask().

task1.Wait();

task2.Wait();

Console.WriteLine("Основной поток завершен.");

\*\*\*

WaitAll()

/ WaitAll() - Ожидает завершения выполнения всех указанных объектов Task.

// WaitAny() - Ожидает завершения выполнения любого из указанных объектов Task.

Task.WaitAll(task1, task2);

Task.WaitAny(task1, task2);

\*\*\*

TaskFactory

Console.WriteLine("Основной поток запущен.");

// Создание экземпляра задачи с использованием свойства Factory, типа TaskFactory.

Task task = Task.Factory.StartNew(MyTask);

// В случае запуска задачи через TaskFactory, вызов метода task.Start() не требуется.

//task.Start();

// Метод Main() остается активным до завершения задачиMyTask().

for (int i = 0; i < 60; i++)

{

Console.Write(".");

Thread.Sleep(100);

}

Console.WriteLine("Основной поток завершен.");

\*\*\*

Лямбда-оператор

static void Main()

{

Console.WriteLine("Основной поток запущен.");

// Использование лямбда-оператора для определения задачи.

Task task = Task.Factory.StartNew(new Action(() =>

{

Console.WriteLine("Задача запущена.");

for (int count = 0; count < 10; count++)

{

Thread.Sleep(500);

Console.WriteLine("В лямбда-операторе, счетчик равен: " + count);

}

Console.WriteLine("Задача завершена.");

}));

// Ожидание завершения задачи.

task.Wait();

// Освобождение задачи.

task.Dispose();

Console.WriteLine("Основной поток завершен.");

// Delay.

Console.ReadKey();

}

}

\*\*\*

ContinueWith

class Program

{

// Метод который будет выполнен как задача.

static void MyTask()

{

Console.WriteLine("MyTask() запущен.");

for (int count = 0; count < 5; count++)

{

Thread.Sleep(500);

Console.WriteLine("В методе MyTask(), счетчик равен: " + count);

}

Console.WriteLine("MyTask() завершен.");

}

// Метод исполняемый как продолжение задачи.

static void ContinuationTask(Task task)

{

Console.WriteLine("Продолжение запущено.");

for (int count = 0; count < 5; count++)

{

Thread.Sleep(500);

Console.WriteLine("В продолжении, счетчик равен: " + count);

}

Console.WriteLine("Продолжение завершено.");

}

static void Main()

{

Console.WriteLine("Основной поток запущен.");

// Создание экземпляра задачи.

var action = new Action(MyTask);

var task = new Task(action);

// Создание продолжения задачи.

// var continuationAction = new Action<Task>(ContinuationTask);

Task taskContinuation = task.ContinueWith(ContinuationTask);

// Выполнение последовательности задач.

task.Start();

// Ожидание завершения продолжения.

taskContinuation.Wait();

//task.Wait();

task.Dispose();

taskContinuation.Dispose();

Console.WriteLine("Основной поток завершен.");

\*\*\*

Возвращение результата

class Program

{

// Метод который будет возвращать результат.

static bool MyTask()

{

// throw new Exception();

return true;

}

// Метод возвращает сумму.

static int SumIt(object v)

{

int x = (int)v, sum = 0;

for (; x > 0; x--)

sum += x;

Thread.Sleep(3000);

return sum;

}

static void Main()

{

Console.WriteLine("Основной поток запущен.");

// Создание экземпляра первой задачи.

Task<bool> task1 = Task<bool>.Factory.StartNew(MyTask);

Console.WriteLine("Результат выполнения задачи MyTask: " + task1.Result);

// Создание экземпляра второй задачи.

Task<int> task2 = Task<int>.Factory.StartNew(SumIt, 3);

Console.WriteLine("Результат выполнения задачи SumIt: " + task2.Result);

task1.Dispose();

task2.Dispose();

Console.WriteLine("Основной поток завершен.");

\*\*\*

CancellationTokenSource

Отмена задачи

class Program

{

// Метод который будет выполнен как задача.

static void MyTask(object ct)

{

var cancelTok = (CancellationToken)ct;

// Прверить, отменена ли задача, прежде чем ее запускать.

cancelTok.ThrowIfCancellationRequested();

Console.WriteLine("MyTask() запущен.");

for (int count = 0; count < 10; count++)

{

// Для отслеживая отмены задачи применяется опрос.

if (cancelTok.IsCancellationRequested)

{

Console.WriteLine("Получен запрос на отмену задачи.");

cancelTok.ThrowIfCancellationRequested();

}

Thread.Sleep(500);

Console.WriteLine("В методе MyTask(), счетчик равен: " + count);

}

Console.WriteLine("MyTask() завершен.");

}

static void Main()

{

Console.WriteLine("Основной поток запущен.");

// Создать объект источника признаков отмены.

var cancelTokSrc = new CancellationTokenSource();

// Запустить задачу, передав признак отмены ей самой и делегату.

Task task = Task.Factory.StartNew(MyTask, cancelTokSrc.Token, cancelTokSrc.Token);

// Дать задаче возможность исполняться вплоть до ее отмены.

Thread.Sleep(2000);

try

{

// Отменить задачу.

cancelTokSrc.Cancel();

// Приостановить выполнение метода Main() до тех пор,

// пока не завершится задача - task.

task.Wait();

}

catch (AggregateException e)

{

if (task.IsCanceled)

Console.WriteLine("\nЗадача task отменена.\n");

Console.WriteLine("- " + e.InnerException.Message); // Для просмотра исключения снять комментарий.

}

finally

{

task.Dispose();

cancelTokSrc.Dispose();

}

Console.WriteLine("Основной поток завершен.");

// Delay.

Console.ReadKey();

}

\*\*\*

static class Program

{

static void Main()

{

MultipleContinueWith(Sum,100000);

Thread.Sleep(2000);

}

static Task<T> MultipleContinueWith<T>(Func<object,T> func, T arg)

{

var t = new Task<T>(func,arg);

t.ContinueWith(task => Console.WriteLine("The sum is: " + task.Result),

TaskContinuationOptions.OnlyOnRanToCompletion);

t.ContinueWith(task => Console.WriteLine("Sum threw: " + task.Exception.InnerException.Message),

TaskContinuationOptions.OnlyOnFaulted);

t.Start();

return t;

}

static Int32 Sum(object arg)

{

Int32 sum = 0, x = (Int32)arg;

for (; x > 0; x--)

// checked

{

sum += x;

}

return sum;

}

}

\*\*\*

Parallel Invoke

// Применение метода Parallel.Invoke() для параллельного выполнения двух методов.

namespace TPL

{

static class Program

{

// Метод который будет выполнен как задача.

static void Method1()

{

Console.WriteLine("Method1() запущен.");

for (int count = 0; count < 5; count++)

{

Thread.Sleep(500);

Console.WriteLine("В методе Method1(), счетчик равен: " + count);

}

Console.WriteLine("Method1() завершен.");

}

// Метод который будет выполнен как задача.

static void Method2()

{

Console.WriteLine("Method2() запущен.");

for (int count = 0; count < 5; count++)

{

Thread.Sleep(500);

Console.WriteLine("В методе Method2(), счетчик равен: " + count);

}

Console.WriteLine("Method2() завершен.");

}

static void Main()

{

Console.WriteLine("Основной поток запущен.");

var options = new ParallelOptions

{

MaxDegreeOfParallelism = Environment.ProcessorCount>2

? Environment.ProcessorCount-1:1

};

Console.WriteLine("Количество логических ядер центрального процессора:"+Environment.ProcessorCount);

// Выполнить параллельно два метода.

//Parallel.Invoke(Method1,Method2);

Parallel.Invoke(options, Method1, Method2,Method1,Method2);

// Внимание!

// Выполнение метода Main() приостанавливается,

// пока не произойдет возврат из метода Invoke().

Console.WriteLine("Основной поток завершен.");

// Delay.

Console.ReadKey();

}

\*\*\*

Parallel For

static int[] data;

// Метод служащий в качестве тела параллельно выполняемого цикла.

// Операторы этого цикла просто расходуют время ЦП для целей демонстрации.

static void MyTransform(int i)

{

data[i] = data[i] / 10;

if (data[i] < 10000) data[i] = 0;

if (data[i] > 10000 && data[i] < 20000) data[i] = 100;

if (data[i] > 20000 && data[i] < 30000) data[i] = 200;

if (data[i] > 30000) data[i] = 300;

}

static void Main()

{

Console.WriteLine("Основной поток запущен.");

data = new int[100000000];

// Инициализация данных в обычном цикле for.

for (int i = 0; i < data.Length; i++)

data[i] = i;

// Распараллелить цикл методом For().

Parallel.For(0, data.Length, new Action<int>(MyTransform));

// Внимание!

// Выполнение метода Main() приостанавливается,

// пока не произойдет возврат из метода For().

Console.WriteLine("Основной поток завершен.");

// Delay.

// Console.ReadKey();

}

\*\*\*

Parallel Loop State

static int[] data;

// Метод служащий в качестве тела параллельно выполняемого цикла.

// Операторы этого цикла просто расходуют время ЦП для целей демонстрации.

// ParallelLoopState - Позволяет итерациям циклов Parallel взаимодействовать с другими итерациями.

// Экземпляр этого класса предоставляется каждому циклу параллельным классом;

// невозможно создавать экземпляры в пользовательском коде.

static void MyTransform(int i, ParallelLoopState state)

{

// Прервать цикл при обнаружении отрицательного значения.

if (data[i] < 0)

state.Break();

data[i] = data[i] / 10;

if (data[i] < 10000) data[i] = 0;

if (data[i] > 10000 && data[i] < 20000) data[i] = 100;

if (data[i] > 20000 && data[i] < 30000) data[i] = 200;

if (data[i] > 30000) data[i] = 300;

Console.WriteLine(i);

}

static void Main()

{

Console.WriteLine("Основной поток запущен.");

data = new int[100000000];

// Инициализация массива в цикле.

for (int i = 0; i < data.Length; i++)

data[i] = i;

// Поместить отрицательное значение в массив data.

data[1000] = -10;

// Параллельный вариант инициализации массива в цикле.

// ParallelLoopResult - Предоставляет состояние выполнения цикла Parallel.

ParallelLoopResult loopResult = Parallel.For(0, data.Length, MyTransform);

// Проверить, завершился ли цикл.

if (!loopResult.IsCompleted)

Console.WriteLine("\nЦикл завершился преждевременно." +

" На шаге {0} обнаружено отрицательное значение.\n", loopResult.LowestBreakIteration);

Console.WriteLine("Основной поток завершен.");

// Delay.

// Console.ReadKey();

}

\*\*\*

PLinq

class Program

{

static void Main()

{

var data = new int[10000000];

// Инициализация массива данных положительными значениями.

for (int i = 0; i < data.Length; i++)

data[i] = i;

// Заполнение массива отрицательными значениями.

data[1000] = -100;

data[14000] = -2;

data[15000] = -3;

data[676000] = -4;

data[8024540] = -5;

data[9908000] = -6;

// Запрос PLINQ для поиска отрицательных значений.

var negatives = from val in data.AsParallel() // ParallelEnumerable.AsParallel<int>(data)

where val < 0

select val;

IEnumerable<int> en = data.Where<int>(x => x < 0).AsParallel();

foreach (var v in en)

Console.Write(v + " ");

\*\*\*

// Отмена параллельного запроса. (Выполняться через Ctrl + F5)

namespace PLINQ

{

class Program

{

static void Main()

{

var token = new CancellationTokenSource();

var data = new int[10000000];

// Инициализация массива данных положительными значениями.

for (int i = 0; i < data.Length; i++) data[i] = i;

// Заполнение массива отрицательными значениями.

data[1000] = -1;

data[14000] = -2;

data[15000] = -3;

data[676000] = -4;

data[8024540] = -5;

data[9908000] = -6;

// Запрос PLINQ для поиска отрицательных значений.

ParallelQuery<int> negatives = from val in data

.AsParallel()

.WithCancellation(token.Token)

.WithExecutionMode(ParallelExecutionMode.ForceParallelism)

where val < 0

select val;

// Создание задачи для отмены запроса по истечении 100 миллисекунд.

token.CancelAfter(50);

try

{

foreach (var v in negatives)

Console.Write(v + " ");

Console.WriteLine("Enumeration completed succesfully!");

}

catch (OperationCanceledException ex)

{

Console.WriteLine(ex.Message);

}

finally

{

token.Dispose();

}

// Delay.

Console.ReadKey();

}

\*\*\*

Async & Await

Синхронная загрузка данных страницы

private void GetButtonClick(object sender, RoutedEventArgs e)

{

dataTextBox.Text = "Starting sync download\n";

var req = (HttpWebRequest) WebRequest.Create("http://www.microsoft.com/");

req.Timeout = 2000;

req.Method = "GET";

try

{

var resp = (HttpWebResponse) req.GetResponse();

dataTextBox.Text += "Sync completed\n";

string headersText = resp.Headers.ToString();

dataTextBox.Text += headersText;

}

catch (WebException exception)

{

MessageBox.Show(exception.Message);

}

}

}

\*\*\*

Старый вариант асинхронной модели (не сработает)

private void GetButtonClick(object sender, RoutedEventArgs e)

{

dataTextBox.Text += "Beginning download\n";

var req = (HttpWebRequest)WebRequest.Create("http://www.google.com");

req.Method = "GET";

req.BeginGetResponse(

asyncResult =>

{

var resp = (HttpWebResponse)req.EndGetResponse(asyncResult);

string headersText = resp.Headers.ToString();

dataTextBox.Text += headersText;

},

null);

dataTextBox.Text += "Download started async\n";

}

}

\*\*\*

private void GetButtonClick(object sender, RoutedEventArgs e)

{

dataTextBox.Text += "Beginning download\n";

var sync = SynchronizationContext.Current;

var req = (HttpWebRequest) WebRequest.Create("http://www.microsoft.com");

req.Method = "GET";

req.BeginGetResponse(

asyncResult =>

{

var resp = (HttpWebResponse) req.EndGetResponse(asyncResult);

string headersText = resp.Headers.ToString();

sync.Post(

state => dataTextBox.Text += headersText,

null);

},

null);

dataTextBox.Text += "Download started async\n";

}

\*\*\*

Async & Await

public partial class MainWindow

{

public MainWindow()

{

InitializeComponent();

}

private void GetButtonClick(object sender, RoutedEventArgs e)

{

dataTextBox.Text +="Staring async download\n";

DoDownload();

dataTextBox.Text +="Async download started\n";

}

async Task DoDownload()

{

var req = (HttpWebRequest) WebRequest.Create("http://www.microsoft.com");

req.Method = "GET";

var resp = (HttpWebResponse) await req.GetResponseAsync();

dataTextBox.Text += resp.Headers.ToString();

dataTextBox.Text +="Async download completed";

}

async void DoDownloadFromAsync()

{

var req = (HttpWebRequest)WebRequest.Create("http://www.microsoft.com");

req.Method = "GET";

Task<WebResponse> getResponseTask = Task.Factory.FromAsync<WebResponse>(

req.BeginGetResponse, req.EndGetResponse, null);

var resp = (HttpWebResponse) await getResponseTask;

string headersText = resp.Headers.ToString();

dataTextBox.Text += headersText;

dataTextBox.Text += "Async download completed";

}

}

\*\*\*

Синхронная загрузка страниц

public partial class MainWindow

{

public MainWindow()

{

InitializeComponent();

}

private void StartButtonClick(object sender, RoutedEventArgs e)

{

resultsTextBox.Clear();

SumPageSizes();

resultsTextBox.Text += "\r\nControl returned to startButton\_Click.";

}

private void SumPageSizes()

{

IEnumerable<string> urlList = SetUpURLList();

var total = 0;

foreach (var url in urlList)

{

byte[] urlContents = GetURLContents(url);

DisplayResults(url, urlContents);

total += urlContents.Length;

}

resultsTextBox.Text +=

string.Format("\r\n\r\nTotal bytes returned: {0}\r\n", total);

}

private IEnumerable<string> SetUpURLList()

{

var urls = new List<string>

{

"http://msdn.microsoft.com/library/windows/apps/br211380.aspx",

"http://msdn.microsoft.com",

"http://msdn.microsoft.com/en-us/library/hh290136.aspx",

"http://msdn.microsoft.com/en-us/library/ee256749.aspx",

"http://msdn.microsoft.com/en-us/library/hh290138.aspx",

"http://msdn.microsoft.com/en-us/library/hh290140.aspx",

"http://msdn.microsoft.com/en-us/library/dd470362.aspx",

"http://msdn.microsoft.com/en-us/library/aa578028.aspx",

"http://msdn.microsoft.com/en-us/library/ms404677.aspx",

"http://msdn.microsoft.com/en-us/library/ff730837.aspx"

};

return urls;

}

private byte[] GetURLContents(string url)

{

var content = new MemoryStream();

var webReq = (HttpWebRequest)WebRequest.Create(url);

using (var response = webReq.GetResponse())

using (var responseStream = response.GetResponseStream())

{

if (responseStream != null)

{

responseStream.CopyTo(content);

}

}

return content.ToArray();

}

private void DisplayResults(string url, byte[] content)

{

var bytes = content.Length;

var displayURL = url.Replace("http://", "");

resultsTextBox.Text += string.Format("\n{0,-58} {1,8}", displayURL, bytes);

}

}

\*\*\*

Асинхронная загрузка страниц

private async void StartButtonClick(object sender, RoutedEventArgs e)

{

resultsTextBox.Clear();

await SumPageSizesAsync();

resultsTextBox.Text += "\r\nControl returned to startButton\_Click.\r\n";

}

private async Task SumPageSizesAsync()

{

// Make a list of web addresses.

IEnumerable<string> urlList = SetUpURLList();

var total = 0;

foreach (var url in urlList)

{

byte[] urlContents = await GetURLContentsAsync(url);

DisplayResults(url, urlContents);

total += urlContents.Length;

}

resultsTextBox.Text +=

string.Format("\r\n\r\nTotal bytes returned: {0}\r\n", total);

}

private IEnumerable<string> SetUpURLList()

{

var urls = new List<string>

{

"http://msdn.microsoft.com/library/windows/apps/br211380.aspx",

"http://msdn.microsoft.com",

"http://msdn.microsoft.com/en-us/library/hh290136.aspx",

"http://msdn.microsoft.com/en-us/library/ee256749.aspx",

"http://msdn.microsoft.com/en-us/library/hh290138.aspx",

"http://msdn.microsoft.com/en-us/library/hh290140.aspx",

"http://msdn.microsoft.com/en-us/library/dd470362.aspx",

"http://msdn.microsoft.com/en-us/library/aa578028.aspx",

"http://msdn.microsoft.com/en-us/library/ms404677.aspx",

"http://msdn.microsoft.com/en-us/library/ff730837.aspx"

};

return urls;

}

private async Task<byte[]> GetURLContentsAsync(string url)

{

var content = new MemoryStream();

var webReq = (HttpWebRequest)WebRequest.Create(url);

using (WebResponse response = await webReq.GetResponseAsync())

{

using (Stream responseStream = response.GetResponseStream())

{

if (responseStream != null)

await responseStream.CopyToAsync(content);

}

}

return content.ToArray();

}

private void DisplayResults(string url, byte[] content)

{

// Display the length of each website.

var bytes = content.Length;

// Strip off the "http://".

var displayURL = url.Replace("http://", "");

resultsTextBox.Text += string.Format("\n{0,-58} {1,8}", displayURL, bytes);

}

}

\*\*\*

public partial class MainWindow

{

CancellationTokenSource cts;

public MainWindow()

{

InitializeComponent();

}

async void StartButtonClick(object sender, RoutedEventArgs e)

{

resultsTextBox.Clear();

cts = new CancellationTokenSource();

try

{

await AccessTheWebAsync(cts.Token);

resultsTextBox.Text += "\r\nDownloads complete.";

}

catch (OperationCanceledException)

{

resultsTextBox.Text += "\r\nDownloads canceled.\r\n";

}

catch (Exception)

{

resultsTextBox.Text += "\r\nDownloads failed.\r\n";

}

cts = null;

}

void CancelButtonClick(object sender, RoutedEventArgs e)

{

if (cts != null)

{

cts.Cancel();

}

}

async Task AccessTheWebAsync(CancellationToken ct)

{

var client = new HttpClient();

IEnumerable<string> urlList = SetUpURLList();

IEnumerable<Task<int>> downloadTasksQuery =

from url in urlList select ProcessURL(url, client, ct);

List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

while (downloadTasks.Count > 0)

{

Task<int> firstFinishedTask = await Task.WhenAny(downloadTasks);

downloadTasks.Remove(firstFinishedTask);

int length = firstFinishedTask.Result;

resultsTextBox.Text += String.Format("\r\nLength of the download: {0}", length);

}

}

private IEnumerable<string> SetUpURLList()

{

var urls = new List<string>

{

"http://msdn.microsoft.com",

"http://msdn.microsoft.com/library/windows/apps/br211380.aspx",

"http://msdn.microsoft.com/en-us/library/hh290136.aspx",

"http://msdn.microsoft.com/en-us/library/dd470362.aspx",

"http://msdn.microsoft.com/en-us/library/aa578028.aspx",

"http://msdn.microsoft.com/en-us/library/ms404677.aspx",

"http://msdn.microsoft.com/en-us/library/ff730837.aspx"

};

return urls;

}

async Task<int> ProcessURL(string url, HttpClient client, CancellationToken ct)

{

HttpResponseMessage response = await client.GetAsync(url, ct);

var task = response.Content.ReadAsByteArrayAsync();

var taskDebug = task.ContinueWith((task1) => Debug.WriteLine("URL:{0}, Thread:{1}",url,Thread.CurrentThread.ManagedThreadId));

byte[] urlContents = await task;

return urlContents.Length;

}

}

\*\*\*

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

// Func< (аргумент - int), (возвращаемое значение - double) >.

Func<int, double> expression = x => x / 2; // Лямбда-Выражение сообщенное с делегатом.

Func<int, double> func = x => x / 2;

int number = 9;

Console.WriteLine("Результат: {0}", func(number));

\*\*\*

// (аргумент - double) (возвращаемое значение - double)

Func<double, double> expression = x => x / 2;

// Func<double, int> expression = x => x / 2;

int number = 9;

Console.WriteLine("Результат: {0}", expression(number));

\*\*\*

Передача методу делегата в параметр

static void Main()

{

int number = 0;

// Анонимный метод - как аргумент метода.

WriteStream(() => number++) ;

Console.WriteLine("Результат: {0}", number);

// Задержка.

Console.ReadKey();

}

// Делегат в качестве формального аргумента.

static void WriteStream(Func<int> counter)

{

for (int i = 0; i < 10; ++i)

Console.Write("{0}, ", counter.Invoke());

}

}

\*\*\*

static void Main()

{

var members = new List<string>

{

"Один - One",

"Два - Two",

"Три - Three",

"Одиннадцать - Eleven",

};

WriteStream(members, "а", (x, y) => x.ToLower().Contains(y));

// Задержка.

Console.ReadKey();

}

// Метод.

static void WriteStream(IEnumerable<string> members, string name, Func<string, string, bool> predicate)

{

foreach (string member in members)

if (predicate(member, name)) // Делегат-Предикат.

Console.WriteLine(member);

}

}

\*\*\*

Expression

static void Main()

{

// Expression<Func<int, int>> expression = n => n + 1;

// Строки ниже заменяют единственную строку Лямбда-Выражения из предыдущего примера.

// Представляем параметр в списке параметров лямбда-выражения.

// Эта строка говорит, что нам нужна переменная n типа int.

ParameterExpression n = Expression.Parameter(typeof(int), "n");

ParameterExpression t = Expression.Parameter(typeof(int), "t");

var cons1 = Expression.Constant(1);

var c2 = Expression.Constant(2);

// Прибавляем число 1 к параметру n.

var addBody = Expression.Add(n, cons1);

var multBody = Expression.Multiply(addBody, t);

var divBody = Expression.Divide(multBody, c2);

var expression = Expression.Lambda<Func<int, int, int>>(divBody,t,n);

Console.WriteLine(expression);

// Компилируем выражение в делегат.

Func<int, int,int> func = expression.Compile();

for (int i = 0; i < 10; i++ )

Console.WriteLine("Результат: {0}", func(i,i));

// Задержка.

Console.ReadKey();

}

\*\*\*

static void Main()

{

Expression<Func<int, int>> expression = n => n + 1;

// Теперь присвоим expression значение исходного выражения умноженное на 2.

expression = Expression<Func<int, int>>.Lambda<Func<int, int>>

(Expression.Multiply(expression.Body, Expression.Constant(2)),

expression.Parameters);

// Компилируем выражение в делегат.

Func<int, int> func = expression.Compile();

Console.WriteLine(expression);

for (int i = 0; i < 10; i++ )

Console.WriteLine("Результат: {0}", func.Invoke(i));

// Задержка.

Console.ReadKey();

}

\*\*\*

static void Main()

{

#region MethodInfo

MethodInfo consoleWriteLine = typeof(Console).GetMethod("WriteLine", new[] { typeof(String) });

MethodInfo consoleReadLine = typeof(Console).GetMethod("ReadLine");

MethodInfo stringConcat = typeof(String).GetMethod("Concat", new[] { typeof(String), typeof(String) });

MethodInfo convertToTnt = typeof(Convert).GetMethod("ToInt32", new[] { typeof(String) });

MethodInfo convertToString = typeof(Convert).GetMethod("ToString", new[] { typeof(Int32) });

#endregion

#region Constants and Parameters

var enterA = Expression.Constant("Enter a:");

var enterB = Expression.Constant("Enter b:");

var theSumIs = Expression.Constant("The sum of a and b is : ");

var exceptionMessage = Expression.Constant("Exception: ");

var parameterA = Expression.Parameter(typeof(Int32), "a");

var parameterB = Expression.Parameter(typeof(Int32), "b");

var parameterResult = Expression.Parameter(typeof(Int32), "sum");

var message = Expression.Parameter(typeof(String), "message");

var exception = Expression.Parameter(typeof(Exception), "ex");

#endregion

var callReadLine = Expression.Call(consoleReadLine);

var block = Expression.Block(new[]

{

parameterA, parameterB, parameterResult, message

},

// Console.WriteLine(enterA);

Expression.Call(consoleWriteLine, enterA),

// int parameterA = Convert.ToInt32(Console.ReadLine());

Expression.Assign(parameterA, Expression.Call(convertToTnt, callReadLine)),

// Console.WriteLine(enterB);

Expression.Call(consoleWriteLine, enterB),

// int parameterB = Convert.ToInt32(Console.ReadLine());

Expression.Assign(parameterB, Expression.Call(convertToTnt, callReadLine)),

//int parameterResult = parameterA + parameterB;

Expression.Assign(parameterResult, Expression.Add(parameterA, parameterB)),

// string message = String.Concat(theSumIs, Convert.ToString(parameterResult));

Expression.Assign(message, Expression.Call(stringConcat, theSumIs, Expression.Call(convertToString, parameterResult))),

// Console.WriteLine(message);

Expression.Call(consoleWriteLine, message)

);

//Let's make it safe :)

// catch (Exception ex)

//{

// Console.WriteLine(String.Concat(exceptionMessage, ex.Message));

//}

var catchBlock = Expression.Catch(exception,

Expression.Call(consoleWriteLine,

Expression.Call(stringConcat, exceptionMessage, Expression.Property(exception, "Message"))));

////try

//{

// \*block

//}

////catch

//{

// \*catchBlock

//}

var safeBlock = Expression.TryCatch(block, catchBlock);

//-----------------------------------------------------

#region Print out the Code!

Console.WriteLine(safeBlock);

foreach (var expression in block.Expressions)

{

Console.WriteLine("\t" + expression);

}

Console.WriteLine(safeBlock.Handlers[0]);

Console.WriteLine("\t" + safeBlock.Handlers[0].Body);

Console.WriteLine(new string('-', 40));

#endregion

Expression<Action> expressionCw = Expression.Lambda<Action>(safeBlock);

var lambda = expressionCw.Compile();

//Voila!

lambda.Invoke();

}

\*\*\*

{

// Расширяющий метод.

public static IEnumerable<TItem> MakeCustomIterator<TCollection, TCursor, TItem>(this TCollection collection,

TCursor cursor,

Func<TCollection, TCursor, TItem> getCurrent,

Func<TCollection, TCursor, bool> isFinished,

Func<TCursor, TCursor> advanceCursor)

{

while (!isFinished(collection, cursor))

{

yield return getCurrent(collection, cursor);

cursor = advanceCursor(cursor);

}

}

}

class Program

{

static void Main()

{

// Матрица.

//var matrix = new List<List<double>> {new List<double> { 1.0, 1.1, 1.2 },

// new List<double> { 2.0, 2.1, 2.2 },

// new List<double> { 3.0, 3.1, 3.2 }

// };

////IEnumerable<double> iter = matrix.MakeCustomIterator<List<List<double>>, Int32[], double>(...);

//IEnumerable<double> iter = matrix.MakeCustomIterator(new[] { 0, 0 },

// (coll, cur) => coll [cur[0]] [cur[1]],

// (coll, cur)=> cur[0] > coll.Count-1 || cur[1] > coll.,

// cur => new[] { cur[0]+1 , cur[1] + 1 });

//// Вывод на экран.

//foreach (double item in iter)

// Console.WriteLine(item);

var iter2 = "Hello"

.MakeCustomIterator(0,

(col, cur) => col[cur] + " ",

(col, cur) => cur > col.Length-1,

cur => cur + 2);

foreach (var symbol in iter2)

{

Console.Write(symbol);

}

// Задержка.

Console.ReadKey();

}

\*\*\*

static IEnumerable<T> MakeGenerator<T>(T initialValue, Func<T, T> advance)

{

T currentValue = initialValue;

while (true)

{

yield return currentValue;

currentValue = advance(currentValue);

}

}

static void Main()

{

IEnumerable<double> iter = MakeGenerator<double>(1.0, x => x \* 1.2);

var enumerator = iter.GetEnumerator();

for (int i = 0; i < 100; ++i)

{

enumerator.MoveNext();

Console.WriteLine(enumerator.Current);

}

// Задержка.

Console.ReadKey();

}

}

\*\*\*

// Метод принимает два делегата и производит третий делегат, комбинируя первые два.

static Func<T, S> Chain<T, R, S>(Func<T, R> f1, Func<R, S> f2)

{

return x => f2(f1(x));

}

static void Main()

{

Func<int, double> f2f1 = Chain((int x) => x \* 3, x => x + 3.1415);

Console.WriteLine(f2f1(2));

// Задержка.

Console.ReadKey();

}

}

\*\*\*

Memoizer

// Мемоизация - замена делегата другим делегатом.

public static class Memoizers

{

// Расширяющий метод.(Для произведения нового делегата.)

public static Func<T, R> Memoize<T, R>(this Func<T, R> func)

{

var cache = new Dictionary<T, R>(); // для блока истинности тернарного оператора чтоб не вычислять заново фибоначи для 0 и 1

return x =>

{

R result = default(R);

if (cache.TryGetValue(x, out result))

return result;

result = func(x);

cache[x] = result;

return result;

};

}

}

class Program

{

static void Main()

{

Func<UInt32, UInt64> fib = null;

fib = (x) => x > 1 ? fib(x - 1) + fib(x - 2) : x;

fib = fib.Memoize(); // Закомментировать и выполнить! (Ощутимая задержка в расчетах)

for (UInt32 i = 0; i < 95; ++i)

Console.WriteLine("{0:D2}-е число: {1}", i + 1, fib(i));

// Задержка

Console.ReadKey();

}

}

\*\*\*

Каррирование

// Частичное применение преобразует функцию с N параметрами в функцию с N-1 параметрами, применяя один аргумент.

public static class CurryExtensions

{

public static Func<TArg1, TResult> ApplyPartial<TArg1, TArg2, TResult>(this Func<TArg1, TArg2, TResult> func,

TArg2 constant)

{

return (x) => func(x, constant);

}

}

class Program

{

static void Main()

{

var mylist = new List<double> { 1.0, 3.4, 5.4, 6.54 };

// Здесь - исходное выражение.

Func<double, double, double> func = (x, y) => x + y;

Func<double, double> funcBound = func.ApplyPartial(3.2);

var newlist = from el in mylist

select funcBound(el);

foreach (double item in newlist)

Console.Write("{0}, ", item);

// Задержка.

Console.ReadKey();

}

}

\*\*\*

Домены приложений. NT-службы