.NET Training Slides

Async und Parellel

in .NET 4.5

Rainer Stropek
software architects gmbh

Web http://www.timecockpit.com
Mail rainer@timecockpit.com
Twitter @rstropek

time cockpit
Saves the day.

Inhalt

Die große Verbesserung in der BCL (Base Class Library), die mit .NET 4.5 kam, ist die Vereinfachung von **paralleler und asynchroner Programmierung**.

- Grundlagen in der **TPL** (Task Parallel Library)
- optimierte **Datenstrukturen**
- auf Parallelisierung ausgerichtete **Spracherweiterungen**
- **Beispiele**, wie man von diesen .NET 4.5-Innovationen in der Praxis profitieren kann

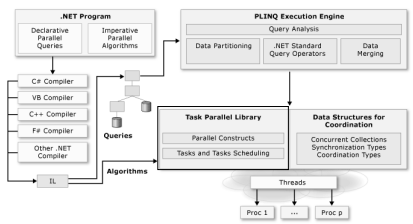
Goals

- Understand Tasks → foundation for `async/await`
- Take a close look at C# 4.5's stars `async/await`
- Present enhancements in .NET 4.5 BCL: TPL Dataflow Library

Task Parallel Library

Recommended Reading

- Joseph Albahari, Threading in C#
  (from his O'Reilly book C# 4.0 in a Nutshell)
- Patterns of Parallel Programming
- Task-based Asynchronous Pattern
- A technical introduction to the Async CTP
- Using Async for File Access
- Async Performance: Understanding the Costs of Async and Await (MSDN Magazine)



See also Patterns of Parallel Programming

Kurzer Überblick über Tas

- Starten
  `Parallel.Invoke(…)`
  `Task.Factory.StartNew(…)`
- Warten
  `myTask.Wait()`
  `Task.WaitAll`
  `Task.WaitAny`
  `Task.Factory.ContinueWhenAll(…)`
  `Task.Factory.ContinueWhenAny(…)`
- Verknüpfen
  ```
  Task.Factory.StartNew(…,
      TaskCreationOptions.AttachedToParent);
  Task.ContinueWith(…)
  ```
- Abbrechen
  Cancellation Tokens

MSDN

```
private static void DoSomething()
{
   Action<Action> measure = (body) =>
   {
      var startTime = DateTime.Now;
      body();
      Console.WriteLine("{0} {1}",
         Thread.CurrentThread.ManagedThreadId,
         DateTime.Now - startTime);
   };

   Action calcProcess = () =>
      { for (int i = 0; i < 100000000; i++);};

   measure(() =>
      Task.WaitAll(Enumerable.Range(0, 10)
         .Select(i => Task.Run(() => measure(calcProcess)))
         .ToArray()));
}
```

This process will run in parallel

Note that we use the new `Task.Run` function here; previously you had to use `Task.Factory.StartNew`

```
Action<Action> measure = (body) => {
    var startTime = DateTime.Now;
    body();
    Console.WriteLine("{0} {1}",
      Thread.CurrentThread.ManagedThreadId,
      DateTime.Now - startTime);
};

Action calcProcess = () =>
   { for (int i = 0; i < 350000000; i++);};
Action ioProcess = () =>
   { Thread.Sleep(1000); };

// ThreadPool.SetMinThreads(5, 5);
measure(() =>{
    Task.WaitAll(Enumerable.Range(0, 10)
        .Select(i => Task.Run(() => measure(ioProcess)))
        .ToArray());
});
```

Note that this task is not compute-bound

```
Action<Action> measure = (body) =>{
   var startTime = DateTime.Now;
   body();
   Console.WriteLine("{0} {1}", Thread.CurrentThread.ManagedThreadId,
                     DateTime.Now - startTime);
};

Action calcProcess = () => { for (int i = 0; i < 350000000; i++);};
Action ioProcess = () => { Thread.Sleep(1000); };

ThreadPool.SetMinThreads(5, 5);
measure(() => Enumerable.Range(0, 10)
   .AsParallel()
   .WithDegreeOfParallelism(5)
   .ForAll(i => measure(ioProcess)));
```

```
private static void DoSomethingElse()
{
   Func<int, int> longRunningFunc = (prevResult) =>
      {
         Thread.Sleep(1000);
         return prevResult + 42;
      };

   var task = Task.Run(() => longRunningFunc(0))
      .ContinueWith(t => longRunningFunc(t.Result))
      .ContinueWith(t => longRunningFunc(t.Result));
   task.Wait();
   Console.WriteLine(task.Result);
}
```

Concat tasks using ContinueWith

Wait for completion of a task.

Schleifen - `Parallel.For`

```
var source = new double[Program.Size];
var destination = new double[Program.Size];

Console.WriteLine(MeasuringTools.Measure(() => {
        for (int i = 0; i < Program.Size; i++) {
            source[i] = (double)i;
        }

        for (int i = 0; i < Program.Size; i++) {
            destination[i] = Math.Pow(source[i], 2);
        }
    }));

Console.WriteLine(MeasuringTools.Measure(() => {
        Parallel.For(0, Program.Size, (i) => source[i] = (double)i);
        Parallel.For(0, Program.Size,
            (i) => destination[i] = Math.Pow(source[i], 2));
    }));
```

Schleifen - `Parallel.For`

- Unterstützung für Exception Handling
- `Break` und `Stop` Operationen
  `Stop`: Keine weiteren Iterationen
  `Break`: Keine Iterationen nach dem aktuellen Index mehr
  Siehe dazu auch `ParallelLoopResult`
- `Int32` und `Int64` Laufvariablen
- Konfigurationsmöglichkeiten (z.B. Anzahl an Threads)
- Schachtelbar
  Geteilte Threading-Ressourcen
- Effizientes Load Balancing
- U.v.m.

Nicht selbst entwickeln!

Excursus - PLINQ

- Use `.AsParallel` to execute LINQ query in parallel
- Be careful if you care about ordering
  Use `.AsOrdered` if necessary
- Use `.WithDegreeOfParallelism` in case of IO-bound tasks
- Use `.WithCancellation` to enable cancelling

Performancetipps für PLINQ

- Allokieren von Speicher in parallelem Lambdaausdruck vermeiden
  Sonst kann Speicher + GC zum Engpass werden
  Wenn am Server: Server GC
- False Sharing vermeiden
- Bei zu kurzen Delegates ist Koordinationsaufwand für Parallelisierung oft höher als Performancegewinn
  → Expensive Delegates
  Generell: Auf richtige Granularität der Delegates achten
- `AsParallel()` kann an jeder Stelle im LINQ Query stehen
  → Teilweise serielle, teilweise parallele Ausführung möglich
- Über `Environment.ProcessorCount` kann Anzahl an Kernen ermittelt werden
- Messen, Messen, Messen!

Thread Synchronisation

- Use C# `lock` statement to control access to shared variables
  Under the hoods `Monitor.Enter` and `Monitor.Exit` is used
  Quite fast, usually fast enough
  Only care for lock-free algorithms if really necessary
- Note that a thread can lock the same object in a nested fashion

```
// Source: C# 4.0 in a Nutshell, O'Reilly Media
class ThreadSafe
{
  static readonly object _locker = new object();
  static int _val1, _val2;

  static void Go()
  {
    lock (_locker)
    {
      if (_val2 != 0) Console.WriteLine (_val1 / _val2);
      _val2 = 0;
    }
  }
}

// This is what happens behind the scenes
bool lockTaken = false;
try
{
  Monitor.Enter(_locker, ref lockTaken);
  // Do your stuff...
}
finally
{
   if (lockTaken) Monitor.Exit(_locker);
}
```

Alternatives For `lock`

- `Mutex`
- `Semaphore(Slim)`
- `ReaderWriterLock(Slim)`
- Not covered here in details

Thread Synchronization

- `AutoResetEvent`
  Unblocks a thread once when it receives a signal from another thread
- `ManualResetEvent(Slim)`
  Like a door, opens and closes again
- `CountdownEvent`
  New in .NET 4
  Unblocks if a certain number of signals have been received
- `Barrier` class
  New in .NET 4
  Not covered here
- `Wait` and `Pulse`
  Not covered here

async/await

Spracherweiterungen für asynchrones Programmieren

```
private static void DownloadSomeTextSync()
{
   using (var client = new WebClient())
   {
      Console.WriteLine(
         client.DownloadString(new Uri(string.Format(
            "http://{0}",
            (Dns.GetHostAddresses("www.basta.net"))[0])));
   }
}
```

Synchronous version of the code; would block UI thread

```
private static void DownloadSomeText()
{
    var finishedEvent = new AutoResetEvent(false);

    // Notice the IAsyncResult-pattern here
    Dns.BeginGetHostAddresses("www.basta.net", GetHostEntryFinished,
        finishedEvent);
    finishedEvent.WaitOne();
}

private static void GetHostEntryFinished(IAsyncResult result)
{
    var hostEntry = Dns.EndGetHostAddresses(result);
    using (var client = new WebClient())
    {
        // Notice the Event-based asynchronous pattern here
        client.DownloadStringCompleted += (s, e) =>
        {
            Console.WriteLine(e.Result);
            ((AutoResetEvent)result.AsyncState).Set();
        };
        client.DownloadStringAsync(new Uri(string.Format(
            "http://{0}",
            hostEntry[0].ToString())));
    }
}
```

Notice that control flow is not clear any more.

```
private static void DownloadSomeText()
{
    var finishedEvent = new AutoResetEvent(false);

    // Notice the IAsyncResult-pattern here
    Dns.BeginGetHostAddresses(
        "www.basta.net",
        (result) =>
        {
            var hostEntry = Dns.EndGetHostAddresses(result);
            using (var client = new WebClient())
            {
                // Notice the Event-based asynchronous pattern here
                client.DownloadStringCompleted += (s, e) =>
                {
                    Console.WriteLine(e.Result);
                    ((AutoResetEvent)result.AsyncState).Set();
                };
                client.DownloadStringAsync(new Uri(string.Format(
                    "http://{0}",
                    hostEntry[0].ToString())));
            }
        },
        finishedEvent);
    finishedEvent.WaitOne();
}
```

Notice how lambda expression can make control flow clearer

Notice the use of the new Task Async Pattern APIs in .NET 4.5 here

```
private static void DownloadSomeTextUsingTask()
{
    Dns.GetHostAddressesAsync("www.basta.net")
        .ContinueWith(t =>
        {
            using (var client = new WebClient())
            {
                return client.DownloadStringTaskAsync(new Uri(string.Format(
                        "http://{0}",
                        t.Result[0].ToString())));
            }
        })
        .ContinueWith(t2 => Console.WriteLine(t2.Unwrap().Result))
        .Wait();
}
```

Notice the use of lambda expressions all over the methods

Notice how code has become shorter and more readable

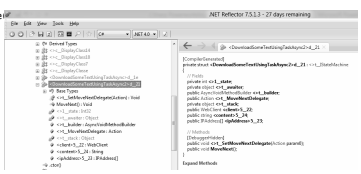
Rules For Async Method Signatures

- Method name ends with `Async`
- Return value
  `Task` if sync version has return type `void`
  `Task<T>` if sync version has return type `T`
- Avoid `out` and `ref` parameters
  Use e.g. `Task<Tuple<T1, T2, ...>>` instead

```
// Synchronous version
private static void DownloadSomeTextSync()
{
   using (var client = new WebClient())
   {
      Console.WriteLine(
         client.DownloadString(new Uri(string.Format(
            "http://{0}",
            (Dns.GetHostAddresses("www.basta.net"))[0]))));
   }
}

// Asynchronous version
private static async void DownloadSomeTextUsingTaskAsync()
{
   using (var client = new WebClient())
   {
      Console.WriteLine(
         await client.DownloadStringTaskAsync(new Uri(string.Format(
            "http://{0}",
            (await Dns.GetHostAddressesAsync("www.basta.net"))[0]))));
   }
}
```

Notice how similar the sync and async versions are!

```
private static async void DownloadSomeTextUsingTaskAsync2()
{
   using (var client = new WebClient())
   {
      try
      {
         var ipAddress = await Dns.GetHostAddressesAsync("www.basta.net");
         var content = await client.DownloadStringTaskAsync(
            new Uri(string.Format("htt://{0}", ipAddress[0])));
         Console.WriteLine(content);
      }
      catch (Exception)
      {
         Console.WriteLine(
      }
   }
}
```



Let's check the generated code and debug the async code

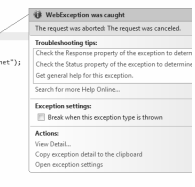
Guidelines for `async/await`

- If `Task` ended in `Canceled` state, `OperationCanceledException` will be thrown

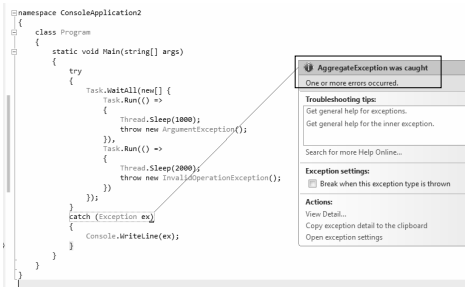
```
private async static void CancelTask()
{
    try
    {
        var cancelSource = new CancellationTokenSource();
        var result = await DoSomethingCancelledAsync(cancelSource.Token);
        Console.WriteLine(result);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Cancelled!");
    }
}

private static Task<int> DoSomethingCancelledAsync(CancellationToken token)
{
    // For demo purposes we ignore token and always return a cancelled task
    var result = new TaskCompletionSource<int>();
    result.SetCanceled();
    return result.Task;
}
```

Note usage of `TaskCompletionSource<T>` here

```
private static async void DownloadSomeTextUsingTaskAsync2()
{
    using (var client = new WebClient())
    {
        try
        {
            var ipAddress = await Dns.GetHostAddressesAsync("www.basta.net");
            new Thread(() =>
                {
                    Thread.Sleep(100);
                    client.CancelAsync();
                }).Start();
            var content = await client.DownloadStringTaskAsync(
                new Uri(string.Format("http://{0}", ipAddress[0])));
            Console.WriteLine(content);
        }
        catch (Exception)
        {
            Console.WriteLine("Exception!");
        }
    }
}
```



Note that async API of `WebClient` uses existing cancellation logic instead of `CancellationTokenSource`



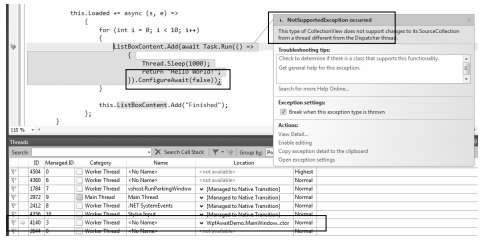
Guidelines for `async/await`

- Caller runs in parallel to awaited methods
- Async methods sometimes do not run async (e.g. if task is already completed when `async` is reached)

Guidelines for `async/await` (UI Layer)

- `async/await` use `SynchronizationContext` to execute the awaiting method → UI thread in case of UI layer
- Use `Task.ConfigureAwait` to disable this behavior
  E.g. inside library to enhance performance

```
public partial class MainWindow : Window
{
public MainWindow()
{
    this.DataContext = this;
    this.ListBoxContent = new ObservableCollection<string>();
    this.InitializeComponent();
    this.ListBoxContent.Add("Started");

    this.Loaded += async (s, e) =>
        {
            for (int i = 0; i < 10; i++)
            {
               ListBoxContent.Add(await Task.Run(() =>
                    {
                        Thread.Sleep(1000);
                        return "Hello World!";
                    }));
            }

            this.ListBoxContent.Add("Finished");
        };
}

public ObservableCollection<string> ListBoxContent { get; private set; }
```



Guidelines For Implementing Methods Ready For `async/await`

- Return `Task/Task<T>`
- Use postfix `Async`
- If method support cancelling, add parameter of type `System.Threading.CancellationToken`
- If method support progress reporting, add `IProgress<T>` parameter
- Only perform very limited work before returning to the caller (e.g. check arguments)
- Directly throw exception only in case of *usage* errors

```
    private static Task<int> CalculateValueAsync(
        int startingValue,
        CancellationToken cancellationToken,
        IProgress<int> progress)
    {
        if (startingValue < 0)
        {
            // Usage error
            throw new ArgumentOutOfRangeException("startingValue");
        }

        return Task.Run(() =>
            {
                int result = startingValue;
                for (int outer = 0; outer < 10; outer++)
                {
                    cancellationToken.ThrowIfCancellationRequested();

                    // Do some calculation
                    Thread.Sleep(500);
                    result += 42;

                    progress.Report(outer + 1);
                }

                return result;
            });
    }
}
```

Note that this pattern is good for compute-bound jobs

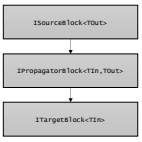
TPL Dataflow Library

Overview

- System.Threading.Tasks.Dataflow
  You need to install the Microsoft.Tpl.Dataflow NuGet package to get it
- For parallelizing applications with high throughput and low latency

Sources and Targets

- Sources, Propagators, and Targets
- Use `LinkTo` method to connect
  Optional filtering
- Use `Complete` method after completing work
- Message passing
  `Post/SendAsync` to send
  `Receive/ReceiveAsync/ TryReceive` to receive



Buffering Blocks

```
// Create a BufferBlock<int> object.
var bufferBlock = new BufferBlock<int>();

// Post several messages to the block.
for (int i = 0; i < 3; i++)
{
   bufferBlock.Post(i);
}

// Receive the messages back from the block.
for (int i = 0; i < 3; i++)
{
   Console.WriteLine(bufferBlock.Receive());
}

/* Output:
   0
   1
   2
 */
```

- `BufferBlock<T>`
- `BroadcastBlock<T>`
- `WriteOnceBlock<T>`

Execution Blocks

```
// Create an ActionBlock<int> object that prints values
// to the console.
var actionBlock = new ActionBlock<int>(n => Console.WriteLine(n));

// Post several messages to the block.
for (int i = 0; i < 3; i++)
{
   actionBlock.Post(i * 10);
}

// Set the block to the completed state and wait for all
// tasks to finish.
actionBlock.Complete();
actionBlock.Completion.Wait();

/* Output:
   0
   10
   20
 */
```

- `ActionBlock<T>`
- `TransformBlock<T>`
- `TransformManyBlock <T>`

Grouping Blocks

```
// Create a BatchBlock<int> object that holds ten
// elements per batch.
var batchBlock = new BatchBlock<int>(10);

// Post several values to the block.
for (int i = 0; i < 13; i++)
{
   batchBlock.Post(i);
}
// Set the block to the completed state. This causes
// the block to propagate out any any remaining
// values as a final batch.
batchBlock.Complete();

// Print the sum of both batches.

Console.WriteLine("The sum of the elements in batch 1 is {0}.",
   batchBlock.Receive().Sum());

Console.WriteLine("The sum of the elements in batch 2 is {0}.",
   batchBlock.Receive().Sum());

/* Output:
   The sum of the elements in batch 1 is 45.
   The sum of the elements in batch 2 is 33.
 */
```

- `BatchBlock<T>`
- `JoinBlock<T>`
- `BatchedJoinBlock<T>`

.NET Training Slides

Q&A

Thank your for coming!

Rainer Stropek
software architects gmbh

Mail rainer@timecockpit.com
Web http://www.timecockpit.com
Twitter @rstropek

time cockpit
Saves the day.