**[async & await 的前世今生（Updated）](http://www.cnblogs.com/jesse2013/p/async-and-await.html)**

**http://www.cnblogs.com/jesse2013/p/async-and-await.html**

async 和 await 出现在C# 5.0之后，给并行编程带来了不少的方便，特别是当在MVC中的Action也变成async之后，有点开始什么都是async的味道了。但是这也给我们编程埋下了一些隐患，有时候可能会产生一些我们自己都不知道怎么产生的Bug，特别是如果连线程基础没有理解的情况下，更不知道如何去处理了。那今天我们就来好好看看这两兄弟和他们的叔叔（Task）爷爷(Thread)们到底有什么区别和特点，本文将会对Thread 到 Task 再到 .NET 4.5的 async和 await，这三种方式下的并行编程作一个概括性的介绍包括：开启线程，线程结果返回，线程中止，线程中的异常处理等。

## 内容索引

* [创建线程](http://www.cnblogs.com/jesse2013/p/async-and-await.html#creation)
* [线程池](http://www.cnblogs.com/jesse2013/p/async-and-await.html#threadpool)
* [参数](http://www.cnblogs.com/jesse2013/p/async-and-await.html#parameters)
* [返回值](http://www.cnblogs.com/jesse2013/p/async-and-await.html#returnvalue)
* [共享数据](http://www.cnblogs.com/jesse2013/p/async-and-await.html#datasharing)
* [线程安全](http://www.cnblogs.com/jesse2013/p/async-and-await.html#threadsafety)
* [锁](http://www.cnblogs.com/jesse2013/p/async-and-await.html#locking)
* [Semaphore](http://www.cnblogs.com/jesse2013/p/async-and-await.html#semaphore)
* [异常处理](http://www.cnblogs.com/jesse2013/p/async-and-await.html#exceptionhandling)
* [一个小例子认识async & await](http://www.cnblogs.com/jesse2013/p/async-and-await.html#awaitsample)
* [await的原形](http://www.cnblogs.com/jesse2013/p/async-and-await.html#realawait)

## 创建

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | static void Main(){      new Thread(Go).Start();  // .NET 1.0开始就有的      Task.Factory.StartNew(Go); // .NET 4.0 引入了 TPL      Task.Run(new Action(Go)); // .NET 4.5 新增了一个Run的方法  }    public static void Go(){      Console.WriteLine("我是另一个线程");  } |

　　这里面需要注意的是，创建Thread的实例之后，需要**手动调用**它的Start方法将其启动。但是对于Task来说，StartNew和Run的同时，既会创建新的线程，并且会**立即启动**它。

## 线程池

　　线程的创建是比较占用资源的一件事情，.NET 为我们提供了线程池来帮助我们创建和管理线程。Task是默认会直接使用线程池，但是Thread不会。如果我们不使用Task，又想用线程池的话，可以使用ThreadPool类。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | static void Main() {      Console.WriteLine("我是主线程：Thread Id {0}", Thread.CurrentThread.ManagedThreadId);      ThreadPool.QueueUserWorkItem(Go);        Console.ReadLine();  }    public static void Go(object data) {      Console.WriteLine("我是另一个线程:Thread Id {0}",Thread.CurrentThread.ManagedThreadId);  } |

## 传入参数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | static void Main() {      new Thread(Go).Start("arg1"); // 没有匿名委托之前，我们只能这样传入一个object的参数        new Thread(delegate(){  // 有了匿名委托之后...          GoGoGo("arg1", "arg2", "arg3");      });        new Thread(() => {  // 当然,还有 Lambada          GoGoGo("arg1","arg2","arg3");      }).Start();        Task.Run(() =>{  // Task能这么灵活，也是因为有了Lambda呀。          GoGoGo("arg1", "arg2", "arg3");      });  }    public static void Go(object name){      // TODO  }    public static void GoGoGo(string arg1, string arg2, string arg3){      // TODO  } |

## 返回值

　　Thead是不能返回值的，但是作为更高级的Task当然要弥补一下这个功能。

|  |  |
| --- | --- |
| 1  2  3  4  5 | static void Main() {      // GetDayOfThisWeek 运行在另外一个线程中      var dayName = Task.Run<string>(() => { return GetDayOfThisWeek(); });      Console.WriteLine("今天是：{0}",dayName.Result);  } |

## 共享数据

　　上面说了参数和返回值，我们来看一下线程之间共享数据的问题。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | private static bool \_isDone = false;  static void Main(){      new Thread(Done).Start();      new Thread(Done).Start();  }    static void Done(){      if (!\_isDone) {          \_isDone = true; // 第二个线程来的时候，就不会再执行了(也不是绝对的，取决于计算机的CPU数量以及当时的运行情况)          Console.WriteLine("Done");      }  } |

　　线程之间可以通过static变量来共享数据。

## 线程安全

 　　我们先把上面的代码小小的调整一下，就知道什么是线程安全了。我们把Done方法中的两句话对换了一下位置 。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | private static bool \_isDone = false;  static void Main(){      new Thread(Done).Start();      new Thread(Done).Start();      Console.ReadLine();  }    static void Done(){      if (!\_isDone) {         Console.WriteLine("Done"); // 猜猜这里面会被执行几次？          \_isDone = true;      }  } |

　　上面这种情况不会一直发生，但是如果你运气好的话，就会中奖了。因为第一个线程还没有来得及把\_isDone设置成true，第二个线程就进来了，而这不是我们想要的结果，在多个线程下，结果不是我们的预期结果，这就是线程不安全。

## 锁

　　要解决上面遇到的问题，我们就要用到锁。锁的类型有独占锁，互斥锁，以及读写锁等，我们这里就简单演示一下独占锁。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | private static bool \_isDone = false;  private static object \_lock = new object();  static void Main(){      new Thread(Done).Start();      new Thread(Done).Start();      Console.ReadLine();  }    static void Done(){      lock (\_lock){          if (!\_isDone){              Console.WriteLine("Done"); // 猜猜这里面会被执行几次？              \_isDone = true;          }      }  } |

　　再我们加上锁之后，被锁住的代码在同一个时间内只允许一个线程访问，其它的线程会被阻塞，只有等到这个锁被释放之后其它的线程才能执行被锁住的代码。

## Semaphore 信号量

　　我实在不知道这个单词应该怎么翻译，从官方的解释来看，我们可以这样理解。它可以控制对某一段代码或者对某个资源访问的线程的数量，超过这个数量之后，其它的线程就得等待，只有等现在有线程释放了之后，下面的线程才能访问。这个跟锁有相似的功能，只不过不是独占的，它允许一定数量的线程同时访问。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | static SemaphoreSlim \_sem = new SemaphoreSlim(3);    // 我们限制能同时访问的线程数量是3  static void Main(){      for (int i = 1; i <= 5; i++) new Thread(Enter).Start(i);      Console.ReadLine();  }    static void Enter(object id){      Console.WriteLine(id + " 开始排队...");      \_sem.Wait();      Console.WriteLine(id + " 开始执行！");      Thread.Sleep(1000 \* (int)id);      Console.WriteLine(id + " 执行完毕，离开！");      \_sem.Release();  } |

在最开始的时候，前3个排队之后就立即进入执行，但是4和5，只有等到有线程退出之后才可以执行。

## 异常处理

　　其它线程的异常，主线程可以捕获到么？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public static void Main(){      try{          new Thread(Go).Start();      }      catch (Exception ex){          // 其它线程里面的异常，我们这里面是捕获不到的。          Console.WriteLine("Exception!");      }  }  static void Go() { throw null; } |

　　那么升级了的Task呢？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public static void Main(){      try{          var task = Task.Run(() => { Go(); });          task.Wait();  // 在调用了这句话之后，主线程才能捕获task里面的异常            // 对于有返回值的Task, 我们接收了它的返回值就不需要再调用Wait方法了          // GetName 里面的异常我们也可以捕获到          var task2 = Task.Run(() => { return GetName(); });          var name = task2.Result;      }      catch (Exception ex){          Console.WriteLine("Exception!");      }  }  static void Go() { throw null; }  static string GetName() { throw null; } |

## 一个小例子认识async & await

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | static void Main(string[] args){      Test(); // 这个方法其实是多余的, 本来可以直接写下面的方法      // await GetName()      // 但是由于控制台的入口方法不支持async,所有我们在入口方法里面不能 用 await        Console.WriteLine("Current Thread Id :{0}", Thread.CurrentThread.ManagedThreadId);  }    static async Task Test(){      // 方法打上async关键字，就可以用await调用同样打上async的方法      // await 后面的方法将在另外一个线程中执行      await GetName();  }    static async Task GetName(){      // Delay 方法来自于.net 4.5      await Task.Delay(1000);  // 返回值前面加 async 之后，方法里面就可以用await了      Console.WriteLine("Current Thread Id :{0}", Thread.CurrentThread.ManagedThreadId);      Console.WriteLine("In antoher thread.....");  } |

## await 的原形

**await后的的执行顺序**

     感谢 locus的指正， await 之后不会开启新的线程(await 从来不会开启新的线程)，所以上面的图是有一点问题的。

　　await 不会开启新的线程，当前线程会一直往下走直到遇到真正的Async方法（比如说HttpClient.GetStringAsync），这个方法的内部会用Task.Run或者Task.Factory.StartNew 去开启线程。也就是如果方法不是.NET为我们提供的Async方法，我们需要**自己创建Task，才会真正的去创建线程**。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | static void Main(string[] args)  {      Console.WriteLine("Main Thread Id: {0}\r\n", Thread.CurrentThread.ManagedThreadId);      Test();      Console.ReadLine();  }    static async Task Test()  {      Console.WriteLine("Before calling GetName, Thread Id: {0}\r\n", Thread.CurrentThread.ManagedThreadId);      var name = GetName();   //我们这里没有用 await,所以下面的代码可以继续执行      // 但是如果上面是 await GetName()，下面的代码就不会立即执行，输出结果就不一样了。      Console.WriteLine("End calling GetName.\r\n");      Console.WriteLine("Get result from GetName: {0}", await name);  }    static async Task<string> GetName()  {      // 这里还是主线程      Console.WriteLine("Before calling Task.Run, current thread Id is: {0}", Thread.CurrentThread.ManagedThreadId);      return await Task.Run(() =>      {          Thread.Sleep(1000);          Console.WriteLine("'GetName' Thread Id: {0}", Thread.CurrentThread.ManagedThreadId);          return "Jesse";      });  } |

　　我们再来看一下那张图：

1. 进入主线程开始执行
2. 调用async方法，返回一个Task，注意这个时候另外一个线程已经开始运行，也就是**GetName里面的 Task** 已经开始工作了
3. 主线程继续往下走
4. 第3步和第4步是同时进行的，主线程并没有挂起等待
5. 如果另一个线程已经执行完毕，name.IsCompleted=true，主线程仍然不用挂起，直接拿结果就可以了。如果另一个线程还同有执行完毕, name.IsCompleted=false，那么主线程会挂起等待，直到返回结果为止。

**只有async方法在调用前才能加await么？**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | static void Main(){      Test();      Console.ReadLine();  }    static async void Test(){      Task<string> task = Task.Run(() =>{          Thread.Sleep(5000);          return "Hello World";      });      string str = await task;  //5 秒之后才会执行这里      Console.WriteLine(str);  } |

　　答案很明显：**await并不是针对于async的方法，而是针对async方法所返回给我们的Task**，这也是为什么所有的async方法都必须返回给我们Task。所以我们同样可以在Task前面也加上await关键字，这样做实际上是告诉编译器我需要等这个Task的返回值或者等这个Task执行完毕之后才能继续往下走。

**不用await关键字，如何确认Task执行完毕了？**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | static void Main(){      var task = Task.Run(() =>{          return GetName();      });        task.GetAwaiter().OnCompleted(() =>{          // 2 秒之后才会执行这里          var name = task.Result;          Console.WriteLine("My name is: " + name);      });        Console.WriteLine("主线程执行完毕");      Console.ReadLine();  }    static string GetName(){      Console.WriteLine("另外一个线程在获取名称");      Thread.Sleep(2000);      return "Jesse";  } |

**Task.GetAwaiter()和await Task 的区别？**

* 加上await关键字之后，后面的代码会**被挂起等待**，直到task执行完毕有返回值的时候才会继续向下执行，这一段时间主线程会处于挂起状态。
* GetAwaiter方法会返回一个awaitable的对象（继承了INotifyCompletion.OnCompleted方法）我们只是传递了一个委托进去，等task完成了就会执行这个委托，但是并**不会影响主线程**，下面的代码会立即执行。这也是为什么我们结果里面第一句话会是 “主线程执行完毕”！

**Task如何让主线程挂起等待？**

上面的右边是属于没有挂起主线程的情况，和我们的await仍然有一点差别，那么在获取Task的结果前如何挂起主线程呢？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | static void Main(){      var task = Task.Run(() =>{          return GetName();      });        var name = task.GetAwaiter().GetResult();      Console.WriteLine("My name is:{0}",name);        Console.WriteLine("主线程执行完毕");      Console.ReadLine();  }    static string GetName(){      Console.WriteLine("另外一个线程在获取名称");      Thread.Sleep(2000);      return "Jesse";  } |

Task.GetAwait()方法会给我们返回一个awaitable的对象，通过调用这个对象的GetResult方法就会挂起主线程，当然也不是所有的情况都会挂起。还记得我们Task的特性么？ 在一开始的时候就启动了另一个线程去执行这个Task，当我们调用它的结果的时候如果这个Task已经执行完毕，主线程是不用等待可以直接拿其结果的，如果没有执行完毕那主线程就得挂起等待了。

**await 实质是在调用awaitable对象的GetResult方法**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | static async Task Test(){      Task<string> task = Task.Run(() =>{          Console.WriteLine("另一个线程在运行！");  // 这句话只会被执行一次          Thread.Sleep(2000);          return "Hello World";      });        // 这里主线程会挂起等待，直到task执行完毕我们拿到返回结果      var result = task.GetAwaiter().GetResult();      // 这里不会挂起等待，因为task已经执行完了，我们可以直接拿到结果      var result2 = await task;      Console.WriteLine(str);  } |

到此为止，await就真相大白了，欢迎点评。Enjoy Coding! :)

作者：Jesse 出处：<http://jesse2013.cnblogs.com/>

本文版权归作者和博客园共有，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接，否则保留追究法律责任的权利。如果觉得还有帮助的话，可以点一下右下角的[【推荐】](javascript:void(0))，希望能够持续的为大家带来好的技术文章！想跟我一起进步么？那就[【关注】](javascript:void(0))我吧。