**Async 与 Await**

**Async**

1. 当将方法用async标识时, 返回值只能为void或者Task或者Task<TReuslt>， 不能是其他类型。
2. 在async标识的方法体里面，在await关键字出现之前，还是主线程顺序调用的，直到await关键字的出现才会出现线程阻塞。
3. await关键字可以理解为等待方法执行完毕，除了可以标记有async关键字的方法外，还能标记Task对象，表示等待该线程执行完毕。所以await关键字并不是针对于async的方法，而是针对async方法所返回给我们的Task。异步方法的返回类型必须为void、Task或者Task<T>类型。也就是说async要么是void，要么和Task关联

**Task对象:**

**构造函数：**

Task(Action action) - 方法无参数，无返回值

Task(Action<object> action, object stateObj) - 方法只带一个object参数，无返回值

Task<int>(Func<int> func) - 方法无参数，返回值int

Task<int>(Func<object, int> func, object stateObj) - 方法只带一个object参数，返回值int

中规中矩的使用 Task

Task tsk1 = new Task(

() => {

Console.WriteLine("Tsk1 Thread: {0}", Thread.CurrentThread.ManagedThreadId);

}

);

tsk1.Start();

await tsk1;

Task tsk2 = new Task(

a => {

Console.WriteLine("Tsk2 a={0}, Thread:{1}", a, Thread.CurrentThread.ManagedThreadId);

}, 200);

tsk2.Start();

await tsk2;

Task<string> tsk3 = new Task<string>(

() => {

Console.WriteLine("Tsk3 Thread:{0}", Thread.CurrentThread.ManagedThreadId);

return "Return Tsk3";

});

tsk3.Start();

string result3 = await tsk3;

Console.WriteLine("Result tsk3 = {0}", result3);

Task<int> tsk4 = new Task<int>(

a => { Console.WriteLine("Tsk3 a={0}, Thread:{1}", a, Thread.CurrentThread.ManagedThreadId);

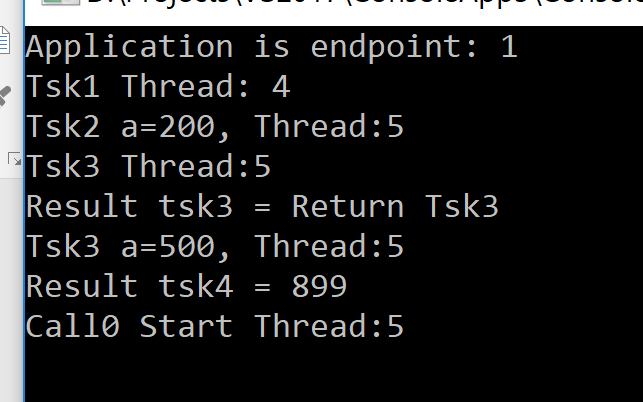
return (int)a + 399;

}, 500);

tsk4.Start();

Console.WriteLine("Result tsk4 = {0}", tsk4.Result);

Console.WriteLine("Call0 Start Thread:{0}", Thread.CurrentThread.ManagedThreadId);



public void call1()

{

var sum = 0;

for (int i = 1; i < 10; i++)

{

sum \*= i;

Console.WriteLine("Times i={0}, sum={1} Thread:{2}", i, sum, Thread.CurrentThread.ManagedThreadId);

}

}

public void call2(int a)

{

var sum = 0;

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

{

sum += i;

//await Task.Delay(500);

Console.WriteLine("i={0}, sum={1}, Thread:{2}", i, sum, Thread.CurrentThread.ManagedThreadId);

}

}

public string call3()

{

return "Call3 Return";

}

public int call4(int a)

{

return a + 299;

}

// Task tsk1 = new Task(() => this.call1()); -- 可以这么写

Task tsk1 = new Task(this.call1); -- 也可以这么写

tsk1.Start();

await tsk1;

// Task tsk2 = new Task(this.call2, 5); - 出错，不可以这样写

Task tsk2 = new Task(a=>{ this.call2((int)a); }, 5);

tsk2.Start();

await tsk2;

Task<string> tsk3 = new Task<string>(() => this.call3());

tsk3.Start();

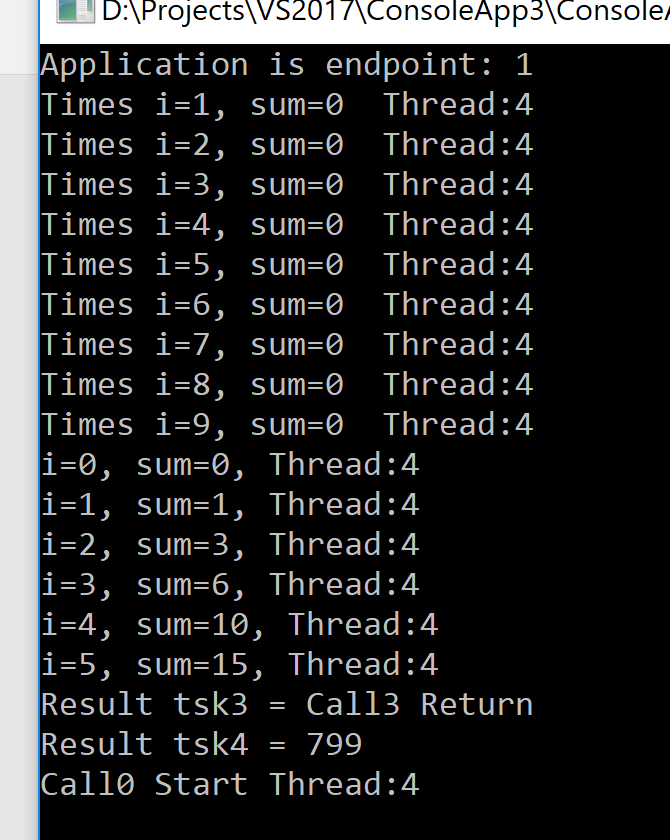
string result3 = await tsk3;

Console.WriteLine("Result tsk3 = {0}", result3);

Task<int> tsk4 = new Task<int>(a => this.call4((int)a) , 500);

tsk4.Start();

Console.WriteLine("Result tsk4 = {0}", tsk4.Result);



Task.Run

通过实例化一个Task对象，然后Start，这种方式中规中矩，但是实践中，通常采用更方便快捷的方式

这种方式直接运行了Task，不像上面的方法还需要调用Start();

Task.Run方法是Task类中的静态方法，接受的参数是委托。返回值是为该Task对象

Task.Run(Action)

Task.Run<TResult>(Func<Task<TResult>>)

Task构造方法还有一个重载函数如下：

Task 构造函数 (Action, TaskCreationOptions)，对应的Task泛型版本也有类似构造函数。TaskCreationOptions参数指示Task创建和执行的可选行为。常用的参数LongRunning

Task.Run(Action)

Task.Run(Func<Task>)

Task.Run<TResult>(Func<TResult>)

Task.Run<TResult>(Func<Task<TResult>>)

Action

Task tsk1 = Task.Run( () => { Console.WriteLine("Call0 : {0}", Thread.CurrentThread.ManagedThreadId); });

await tsk1;

Func<Task>

//这是正确的放回， 启动以后的 Task , 如果返回时没有启动的 Task , 则处于等待状态

Task tsk2 = Task.Run( ()=> { Task tt = new Task(this.call1); tt.Start(); return tt; } );

await tsk2;

Console.WriteLine("Call0 Start Thread:{0}", Thread.CurrentThread.ManagedThreadId);

一直等待， 因为返回的 Task 没有启动

Task tsk2 = Task.Run( ()=> { return new Task(this.call1); } );

await tsk2;

Task tsk2 = Task.Run( ()=> { return new Task(this.call1); } );

tsk2.Start(); -- 这样也是不能启动的

await tsk2;

Func<TResult>

Task<string> tsk3 = Task.Run( ()=>"hello world" );

Console.WriteLine("Task2 Result: {0}", tsk3.Result);

Func<Task<TResult>>

Task<string> tsk4 = Task.Run(() => Task.Run(() => "Good Day"));

Console.WriteLine("Task2 Result: {0}", tsk4.Result);

public async Task<int> call5()

{

return await Task.FromResult<int>(5050);

}

Task<int> tsk4 = Task.Run( ()=>{ return this.call5(); } );

Console.WriteLine("Task4 Result: {0}", tsk4.Result);

Task.FromResult<TResult> return Task<TResult>

Async Task , Async Task<TResult>

Return await Task.Run( ()=>{ … ; no return; } );

Async Task<TResult>

public int add (int a, int b)

{

Console.WriteLine("{0} + {1} = {2} Thread:{3}", a, b, a + b, Thread.CurrentThread.ManagedThreadId);

return a + b;

}

public async Task<int> call5(int a, int b)

{

return await Task.FromResult<int>( this.add(a, b) );

}

// 也可以这样写

public async Task<int> call5(int a, int b)

{

return await Task.Run<int>( ()=>this.add(a,b) );

}

Task<int> tsk4 = Task.Run( ()=>{ return this.call5(230, 140); } );

Console.WriteLine("Task4 Result: {0} Thread:{1}", tsk4.Result, Thread.CurrentThread.ManagedThreadId);

public async Task<int> call5(int a, int b)

{

await Task.Run<int>( ()=>this.add(a,b) );

return 10500; - 可以返回任何值

}

Async Task

public async Task call6(int a, int b)

{

int sum = await Task.Run<int>(() => this.add(a, b) );

Console.WriteLine("Add Result: {0} Thread:{1}", sum, Thread.CurrentThread.ManagedThreadId);

// No return

}

await this.call6(100, 230);

[**C# 5.0 Async函数的提示和技巧**](http://www.cnblogs.com/TianFang/archive/2012/12/24/2831341.html)

这篇文章主要内容来自于文章[C# Async Tips and Tricks Part 2 : Async Void](http://www.jaylee.org/post/2012/07/08/c-sharp-async-tips-and-tricks-part-2-async-void.aspx)，我本想直接翻译的，无奈由于水平有限，因此这里给的是参考原文结合自己的理解的一篇随笔。

**一、创建Async函数**

Async是C# 5.0中新增的关键字，通过语法糖的形式简化异步编程，它有如下三种方式：

1. async Task<T> MyReturningMethod { return default(T); }
2. async Task MyMethod() { }
3. async void MyFireAndForgetMethod() { }

从功能上来看方式2和方式3非常类似，都是无返回值的，区别仅仅是方式3无法等待。既然有功能更加强大的async Task的形式，为什么还要支持一个async void呢？

**二、async void函数**

async void函数存在的唯一目的就是和就是用于兼容现有的事件分发函数，MS在BCL库中提供了大量void类型的事件，基本形式如下：

    private void Button1\_Click(object sender, EventArgs args) { }

这个和方式2中async Task的方法签名是不兼容的，因此，就增加了async void来实现对现有BCL库中的事件或委托兼容。

    private async void Button1\_Click(object sender, EventArgs args) { }

更进一步，通过ILSpy反编译异步函数可以发现，async void和async Task类型的函数的实现是不一样的，前者是[AsyncVoidMethodBuilder](http://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.asyncvoidmethodbuilder(v=vs.110).aspx)类，而后者是 [AsyncTaskMethodBuilder](http://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.asynctaskmethodbuilder(v=vs.110).aspx)类。不过，它们的功能和处理方式都差不多，唯一的区别就是异常处理。

**三、TPL中的未处理异常**

由于async函数和TPL存在非常大的关联，在分析async函数异常处理方式前，首先来复习下TPL中对于异常是如何处理的，以如下代码为例：

    static void Main(string[] args)  
    {  
        TaskScheduler.UnobservedTaskException += TaskScheduler\_UnobservedTaskException;  
        Test();  
  
        Console.ReadLine();        //等待Test任务执行完成  
  
        GC.Collect();  
        GC.WaitForPendingFinalizers();  
  
        Console.ReadLine();  
    }  
  
    static void TaskScheduler\_UnobservedTaskException(object sender, UnobservedTaskExceptionEventArgs e)  
    {  
        Console.WriteLine(e.Exception);  
    }  
  
    public static Task Test()  
    {  
        return Task.Run(() => { throw new Exception(); });  
    }

执行上面代码后，我们可以发现：

1. Test任务抛异常后，程序能仍然能正常运行
2. GC执行时，可以通过TaskScheduler.UnobservedTaskException捕获到Test任务抛异常的信息。

从上面代码执行结果，我们可以大致了解Tpl对未处理异常的处理方式：

1. Task未处理异常不会继续往上抛导致程序异常终止
2. Task中未处理异常可以通过TaskScheduler.UnobservedTaskException事件捕获
3. TaskScheduler.UnobservedTaskException事件并不是在抛异常时立即的，而是GC时从Finalizer线程里触发并执行的。

简单的说，TaskScheduler中处理了Task中抛出的异常，不会导致程序异常终止。老赵的Blog[关于C#中async/await中的异常处理](http://blog.zhaojie.me/2012/04/exception-handling-in-csharp-async-await-1.html)中详细描述了这一过程，感兴趣的朋友可以看看。

不过，Task中的未处理异常不会导致程序异常终止在另一方面也掩盖了代码中存在bug的隐患，因此建议注册TaskScheduler.UnobservedTaskException事件，对未处理异常记录日志，方便后续跟踪分析。

**四、async Task函数中的未处理异常**

复习完TPL的处理过程后，我们再来看看async Task中对异常处理的方式，还是前面的那个代码，只不过这次把Test函数替换成如下形式：

    public static async Task Test()  
    {  
        throw new Exception();  
    }

编译这段代码的时候，会发现如下告警：  
warning CS4014: 由于不等待此调用，因此会在此调用完成前继续执行当前方法。请考虑向此调用的结果应用"await"运算符。

这个告警确实很有帮助，可以有效的提示忘记等待异步函数的执行完成，不过不知道为什么没有async标记的异步函数不提示这个告警。

当执行这段代码时，执行结果和前面TPL中一致：Test函数中的异常并不终止程序，异常信息在TaskScheduler.UnobservedTaskException事件中可以获取。由于[AsyncTaskMethodBuilder](http://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.asynctaskmethodbuilder(v=vs.110).aspx)内部就是调用Task来处理的，这个也就不难理解了（两段代码并不等价，async标记了的函数是对UI线程是特殊处理了的）。

**五、async void函数中的未处理异常**

下面我们再来看看async void函数未处理异常，这次我们把Test函数替换为如下形式：

    public static async void Test()  
    {  
        throw new Exception();  
    }

这次和上面有几点不同：

1. 编译的时候没有CS4014告警
2. Test函数执行时抛异常直接终止了程序
3. 在TaskScheduler.UnobservedTaskException中查看不到异常信息

这几点主要的不同在于[AsyncVoidMethodBuilder](http://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.asyncvoidmethodbuilder(v=vs.110).aspx)内部并没有使用TaskScheduler，线程池中的未处理异常便一直向上抛，导致程序异常终止（CLR中的处理方式可以参看[Exceptions in Managed Threads](http://msdn.microsoft.com/en-us/library/ms228965.aspx)这篇文章。）。

这个异常信息在桌面程序中可以通过AppDomain.UnhandledException事件查看，但该回调是没有处理异常的功能，因此一旦出现该异常，程序仍然将终止，不过可以记录个出错的原因，方便错误定位。

但是，对于WinRT程序来说就悲催了，由于不支持AppDomain，并且程序是直接crash的，都不抛个对话框挂调试器，对于那些不必现的问题连定位都不容易。在我以前的文章[WinRT中的UnhandledException不能捕获异步函数的异常](http://www.cnblogs.com/TianFang/archive/2012/11/23/2784496.html)中就描述过这一问题。

我最初写WinRT程序的时候，为了消除CS4014告警，对于无需等待的函数，就直接写成了async void的形式，导致后续定位时欲哭不能。

因此，强烈建议严格限制async void的使用范围，尽量使用async Task来替换；对于CS4014告警，也不要无视，无需等待的任务通过下列扩展函数来清除告警。

    static class AsyncExtension  
    {  
        public static void IgnorCompletion(this Task task)  
        {  
        }  
    }

**六、Async lambda表达式**

综合前面的分析，async void函数抛的异常非常难以定位，因此要严格限制使用。不过仍有一个非常隐蔽的async void类型函数非常容易被忽略，那就是async lambda表达式。

首先看一下如下代码：

    Enumerable.Range(0, 3).ToList().ForEach(async (i) => { throw new Exception(); });

这段代码就隐式生成了async void函数，直接导致了程序的crash。

另外，编译器是优先生成async Task形式的匿名函数的，这一点比较好。例如对于如下两个重载函数：

    public void ForEach(Action<T> action);  
    public void ForEach(Func<T, Task> action);

对于如下代码：

    ForEach(async i => { });

编译器是使用ForEach(Func<T, Task> action);生成匿名函数的，而不是async void类型，但对于那些没有Func<T, Task>的重载的函数（例如前面的List.Foreach），仍然会生成async void匿名函数，需要注意。

**七、编程建议：**

使用async异步编程时，请注意如下事项：

1. async void函数只能在UI Event回调中使用。
2. async void函数中一定要用try-catch捕获所有异常，否则会很容易导致程序崩溃。
3. async void类型的lambda表达式非常隐蔽，并且容易在无意中编写出来，尤其需要注意。
4. 不要忽视CS4014告警，更不要为了消除CS4014告警而改用async void函数。  
   确实无需等待的async Task函数用我前面写的扩展函数IgnorCompletion消除这个告警。
5. 注册TaskScheduler.UnobservedTaskException事件，记录Task中未处理异常信息，方便分析及错误定位。（注意，这个回调里面不能进行耗时操作，具体原因参看前面的老赵的那篇Blog）

**总结起来一句话：**async void函数能不用就不用，用的时候也要捕获所有异常再用。

不过，这个做起来还是有些难度的，有的时候会在不经意间写了async void函数（例如在lambda表达式中），并且不容易发现。最好还是需要一个工具来分析下程序集，检查函数是否只用于UI Event回调。原文作者说会提供一个基于这个原则的fxcop的静态分析规则，但目前还并没有给出，不过感觉并不难，有空的话我写一个试试。

最后，该作者的这系列文章一共有三篇，写的都非常不错，这里强烈推荐下：

1. [C# 5.0 Async Tips and Tricks, Part 1](http://www.jaylee.org/post/2012/06/18/CSharp-5-0-Async-Tips-and-Tricks-Part-1.aspx)
2. [C# Async Tips and Tricks Part 2 : Async Void](http://www.jaylee.org/post/2012/07/08/c-sharp-async-tips-and-tricks-part-2-async-void.aspx)
3. [C# Async Tips and Tricks, Part 3: Tasks and the Synchronization Context](http://jaylee.org/post/2012/09/29/C-Async-Tips-and-Tricks-Part-3-Tasks-and-the-Synchronization-Context.aspx)

我这篇随笔主要是基于第二篇文章写的，如果有空的话剩下两篇放在一篇随笔中一并介绍下。