Обработка исключений при работе с Task

В последних своих постах я активно использую класс Task для асинхронного выполнения задач. С учетом того, что основной вкусностью новой версии C# будет метод директива async, которая как раз и предназначена более всего для работы с классом Task, то будет уместно рассказать о том, как ловить и обрабатывать исключения, которые могут возникнуть при работе вашего метода в асинхронном режиме.

Далее будет рассмотрена работа с исключениями не только в теле метода, который был явным образом указан в качестве жертвы асинхронной работы, но и как получать исключения из вложенных асинхронных методов. Можете думать об этом как о сне во сне, и вспоминать фильм «Начало».

Все последующие примеры не открывают Америки и все прекрасно описано в книги J.Albahari “C# in the Nutshell 4.0”, но пока не увидишь, как не надо делать, сложно бывает запомнить как надо.

Подготовка

Демонстрировать работу с исключениями буду на примере тестового WinForms приложения, по причине простоты, а так же на примере некоторых тестов. Постараюсь привести как можно более близкие к жизни примеры но, не усложняя сверх необходимости.

Итак, рассмотрим типовое приложение, которое имеет слой отображения, в нашем случае это UI форма. Далее идет слой модели – у нас это будет простой класс **Model**, который не будет отличаться большим размером и ответственностью. Последним классом будет **Service,** в котором собственно и будет происходить работа, точнее разнообразные ошибки.

Интерфейсом пусть будет простейшая форма с одной кнопкой и одной строкой для вывода некоторой информации. По нажатию на кнопку вызываем метод модели на запуск сервиса. Код приводить не буду, так как там ничего интересного нет.

Модель на самом деле тоже не может похвастаться многими строками кода.

```
public class Model {
    private readonly Service service;
    public string result;

    public Model() {
        service = new Service();
    }

    public void Act() {
        service.EndTask += EndTask;
        service.DoTaskAsync();
    }

    private void EndTask(Task obj) {
         //  some actions goes here
    }
}
```

Для удобства и демонстрации некоторых особенностей будем выводить на форму значение переменной result. Все остальное не представляет особенной сложности.

Как видно из текста, данный метод собирает сообщения из всех возникших исключений и возвращает их единой строкой.

В реализации сервиса применена, самая простая инфраструктура использования класса Task: публичный метод с инфраструктурной обвязкой, и приватный метод с логикой.

```
public class Service {
    public event Action<Task> EndTask;

    public Task<int> DoTaskAsync() {
        var task = new Task<int>(DoTask);

        if (EndTask != null) {
            task.ContinueWith(EndTask);
        }

        task.Start();

        return task;
    }

    private int DoTask() {
         //  variation goes here
    }
}
```

Как вы можете догадаться, в основном речь пойдет об особенностях реализации метода **DoTask()**.

Ситуация #1

Так как фантазия у меня сегодня страдает, то подходы буду называть порядковыми номерами от простых к извращенным.

Начнем с самого простого. Пусть в методе DoTask будет вызываться исключение. Т.е. не будем мудрить с каким-либо кодом, а явно бросим исключение.

```
private int DoTask() {
    throw new InvalidProgramException("DoTask Exception");
}
```

Хотя механизм исключений для класса Task и описан четко, но многие все равно предполагают, что это исключение будет прокинуто до самого верхнего уровня приложения и вызовет какое-либо общее сообщение об ошибке на интерфейсе пользователя. Попробуем проверить на практике, что у нас получится. Итак, сильно не морочатся с тестами на отлов исключений в данной ситуации, просто запустим без дебага UI, и смотрим на результат. Никаких исключений поймано не было. С точки зрения модели, все чудесно и жизнь прекрасна. Но ошибка ведь есть!

Подход #1.1

Посмотрим, что можно вытащить из объекта Task после его выполнения. Модифицируем метод EndTask() сервиса так, чтобы можно было проверить состояние булевой переменной **IsFaulted**.

```
private void EndTask(Task obj) {
     var task = ((Task<int>) obj);
     result = "isFaulted:" + task.IsFaulted;
}
```

И напишем такой тест:

```
[Test]
public void ShouldBeFaulted() {
    // Arrange
    var model = new Model();

    // Act
    model.Act();
    Thread.Sleep(100);

    // Assert
    model.result.Should().Be("isFaulted:True");
}
```

Запустив тест, видим, что на самом деле ошибка была, задача помечена как проваленная, но сама ошибка не прокидывается выше. Текст ошибки мы можем узнать, если явно опросим свойство **Exception** объекта типа Task. Данное свойство является экземпляром типа AggregatedException и для работы с ним я сделал небольшой метод расширения **ExploreMessages()**.

```
public static class AggregateExceptionExtensions {
    public static string ExploreMessages(this AggregateException e) {
        var builder = new StringBuilder();

        foreach (var i in e.Flatten().InnerExceptions) {
            builder.AppendLine("Type:" + i.GetType() + "   Message:" + i.Message);
        }
        builder.AppendLine();
        return builder.ToString();
    }
}
```

Итак, модифицируем метод EndTask() до следующего вида:

```
private void EndTask(Task obj) {
     var task = ((Task<int>) obj);
     result = "isFaulted:" + task.IsFaulted;
     if (task.IsFaulted)
        exceptionText = obj.Exception.ExploreMessages();
}
```

А так же напишем еще один тест:

```
[Test]
public void ShouldBeGetExceptionMessage() {
    // Arrange
    var model = new Model();

    // Act
    model.Act();
    Thread.Sleep(100);

    // Assert
    model.exceptionText.Should().Be("Type:System.InvalidProgramException   Message:DoTask
Exception\r\n\r\n");
}
```

Тест проходит, и мы получаем информацию о том, какие ошибки возникли в результате выполнения асинхронного метода. Дальше мы уже должны сами решить, что делать в результате такого исключения. Если запустить приложение без дебага, то по нажатию на кнопку, мы не увидим никакого сообщения об ошибке, только информация выведется, что метод содержал ошибки.

Вывод 1: Исключения не прокидываются автоматически выше. Надо опрашивать результат самостоятельно.

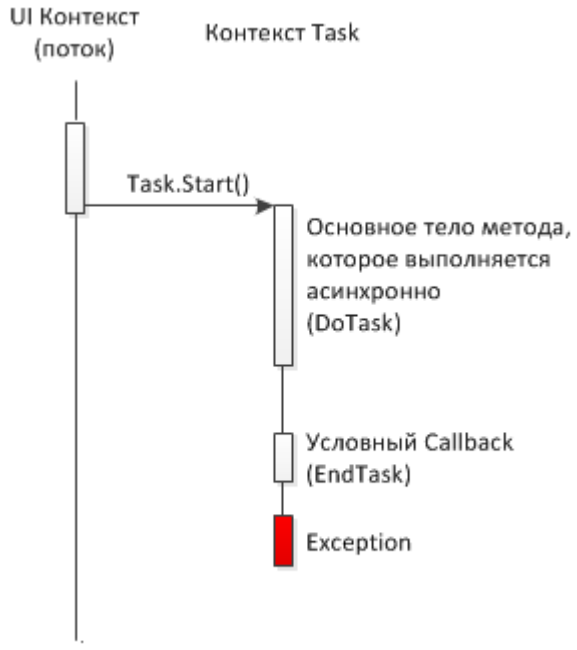
Подход #1.2

Сейчас мы сделаем одно маленькое изменение и получим исключения без дополнительных действий. Самые внимательные наверно уже озадачились вопросом, почему у нас объект класса Task имеет шаблонный вид. Сейчас нам это пригодится. Итак, нам требуется только обратиться к свойству **Result**, для того, чтобы получить сообщение об ошибке. Дописываем единственную строку:

```
private void EndTask(Task obj) {
     var task = ((Task<int>) obj);
     var s = task.Result;
     result = "isFaulted:" + task.IsFaulted;
     if (task.IsFaulted)
        exceptionText = obj.Exception.ExploreMessages();
}
```

После этого пробуем запустить наши тесты. Оба теста упали, так как возникло необработанное исключение типа AggregatedException. Можно ради интереса запустить приложение и посмотреть что будет.

А ничего не будет! Программа молчаливо прекратит действия в момент возникновения ошибки при обращении к свойству Result. А так как контекст выполнения метода EndTask отличен от того, где отрисовывается UI, то мы совсем-совсем ничего не увидим.



Для исправления данной ситуации необходимо либо заранее опрашивать о наличии ошибок при выполнении задачи, либо оборачивать обращение в try()...catch() и отлавливать исключение AggregatedException. Т.е. чтобы исправить ситуацию, нам надо сделать примерно следующее:

```
private void EndTask(Task obj) {
    try {
        var task = ((Task<int>) obj);
        result = "isFaulted:" + task.IsFaulted;
        if (task.IsFaulted)
            exceptionText = obj.Exception.ExploreMessages();

        var s = task.Result;
    }
    catch (AggregateException e) {
         exceptionText = obj.Exception.ExploreMessages();
    }
}
```

Теперь тесты снова проходят и при запуске приложения мы видим сообщение об ошибке.

Вывод 2: Исключения автоматически возбуждаются при обращении к свойству Result класса Task, и обрабатывать их требуется соответствующе.

Ситуация #2

Усложним задачу, так как в реальной жизни захочется больше параллельности, больше задач, больше всего! По крайней мере, если не вам, то вашему начальству, которое скажет делать: больше параллельности, больше задач, больше всего!

Для эмуляции такой ситуации представим, что в методе DoTask() запускается еще несколько задач асинхронно. Допустим таким образом:

```
private int DoTask() {
    for (var i = 0; i < 4; i++) {
        var task = new Task(() => {
                                    throw new InvalidProgramException("for");
                                }
            );
        task.Start();
    }

    return 5;
}
```

Хотя результат предсказуем, но тем не менее. Запускаем тесты, приложение. Уже по тестам видим, что никаких исключений не возникло. Я думаю понятно, по какой причине: так как новые задачи из цикла выполняются в своем контексте, который никак не связан с отслеживаемыми. Плюс ко всему в реальной жизни, из метода DoTask() мы выйдем с большой вероятностью раньше, чем что-то там произойдет в дочерних задачах.

Для исправления сложившейся ситуации можно использовать несколько подходов.

Подход #2.1

Самым простым решением будет использование перегруженного конструктора класса Task, для указания параметров создания новой задачи. В нашем случае необходимо будет использовать опцию **AttachedToParent**, которая говорит сама за себя.

```
private int DoTask() {
         for (var i = 0; i < 4; i++) {
           var task = new Task(() => {
                                    throw new InvalidProgramException("for");
                                }, TaskCreationOptions.AttachedToParent
               );
           task.Start();
       }
       return 5;
   }
```

После такой простой модификации кода, мы получим работающий тест, говорящий о том, что ошибки были, и сломанный тест, где мы проверяли текст ошибки – что логично. При запуске приложения мы получим сообщение с перечислением четырех ошибок.

В данном месте стоит припомнить о разновидности задач «Fire and forget», результат которых можно отследить только по косвенным признакам изменения системы. Но в таком случае и ошибки, возникающие в них, нас не интересуют, так что можно их исключить из рассмотрения, а для всех остальных «подзадач» использовать перегруженный конструктор с опцией **AttachedToParent**.

Вывод 3: Дочерние задачи создавать с опцией AttachedToParent.

Подход #2.2

Впрочем, существует другой вариант решения задачи, который стоит упомянуть, так как используется еще один механизм получения исключений.

Опять будем менять только реализацию метода **DoTask()**, на этот раз с помощью статического метода **WaitAll()** класса Task мы будет ожидать завершения всех задач.

```
    private int DoTask() {
        var tasks = new List<Task>();
        for (var i = 0; i < 4; i++) {
            var task = new Task(() => {
                                    throw new InvalidProgramException("for");
                                }
                );
            task.Start();
            tasks.Add(task);
        }

        Task.WaitAll(tasks.ToArray());
        return 5;
    }
```

Результат, по сути, ничем не будет отличаться от того, что был в подходе 2.1. Вызов статических методов Wait() или WaitAll() инициирует повторный выброс исключений, если они не были обработаны ранее.

Вывод 4: Исключения автоматически возбуждаются при обращении к статическим методам Wait() или WaitAll() класса Task, и обрабатывать их требуется соответствующе.

Ситуация #3 (проверочная)

Постепенно мы подходим к все более изощренным методам. Погружаемся все глубже и глубже в пучину снов, т.е. взаимной генерации задач. Давайте попробуем создать еще одну ступень/уровень зависимых задач и на этом уровне возбудить исключительную ситуацию. Может быть в таком явном виде у вас такой ситуации не возникнет, но я вероятность такой реализации сохраняется.

```
    private int DoTask() {
         var tasks = new List<Task>();
        for (var i = 0; i < 4; i++) {
            var task = new Task(() => {
                  var t = new Task(
                            () => { throw new InvalidOperationException("operation"); },
TaskCreationOptions.AttachedToParent);
                  t.Start();
                  t.Wait();
                  throw new InvalidProgramException("for"); }
                  ,TaskCreationOptions.AttachedToParent
                );
            task.Start();
            tasks.Add(task);
        }

        Task.WaitAll(tasks.ToArray());
        return 5;
    }
```

Прежде чем запускать, попробуем определить, что получим на выходе. Следуя выводам из предыдущих примеров, ищем самый «глубокий» метод. Он вызывается с опцией AttachedToParent, значит, мы получим данные об исключениях. Уровень выше так же вызывается с этой опцией. Так что хоть что-то, но до верхнего уровня дойдет.

Далее видим, что самая «глубокая» задача отправляется на выполнение и тут же ожидается ее результат. Хм… Данное исключение прерывает работу кода, а значит **throw new InvalidProgramException("for");** будет недоступен и не будет выполнен. Можно запустить и проверить.

Если же удалить строчку **t.Wait()**, то получим сообщение со всеми возникшими исключениями. Нюанс в целом не относящийся напрямую к классу Task, но тем не менее.

Вывод 5: Не забывайте о базовых свойствах исключений.

Обработка исключений

Я думаю, что так же стоит сказать пару слов об обработке исключений, которые приходят в обертке AggregatedExceptions. Может случиться так, что часть исключений не критические и программа может самостоятельно или же с помощью пользователя разрешить возникшую ситуацию в лучшем виде, но критические исключения необходимо выкидывать уровнями выше,

тем слоям кода, которые возможно смогут разрешить такие ситуации. В общем, вы поняли. Итак, мы поймали исключение:

```
try {
     . . .
}
catch (AggregateException ae) {
    ae.Handle(x => {
        if (x is InvalidOperationException) { // допустим знаем как его обработать.
                    Console.WriteLine("Bla-bla-bla");
                    return true;
         }
         return false; // пусть что-нибудь еще остановит приложение.
     });
}
```

С помощью метода **Handle()** можно выборочно обработать исключения, которые мы знаем, как обработать, остальное отправиться вверх по стеку.

Итого

Надеюсь, что теперь стало еще чуточку яснее как обрабатывать исключения при работе с классом Task. Хотя без практики все равно тяжело запомнить. Главное, на мой взгляд запомнить основные «выводы». Еще раз их выпишу:

Вывод 1: Исключения не прокидываются автоматически выше. Надо опрашивать результат самостоятельно.

Вывод 2: Исключения автоматически возбуждаются при обращении к свойству **Result** класса Task, и обрабатывать их требуется соответствующе.

Вывод 3: Дочерние задачи создавать с опцией **AttachedToParent**.

Вывод 4: Исключения автоматически возбуждаются при обращении к статическим методам **Wait()** или **WaitAll()** класса Task, и обрабатывать их требуется соответствующе.

Вывод 5: Не забывайте о базовых свойствах исключений.

Hard’n’heavy!

Violet Tape