Параллелизм задач

Введение

Параллелизм задач – это подход самого низкого уровня к распараллеливанию с применением инфраструктуры PFX. Классы для работы на таком уровне определены в пространстве имен *System.Threading.Tasks* и включают перечисленные ниже.

Табл. Классы для работы с задачами

Класс	Назначение
Task	Для управления единицей работы
Task<TResult>	Для управления единицей работы с возвращаемым значением
TaskFactory	Для создания задач
TaskFactory<TResult>	Для создания задач и продолжений с тем же самым возвращаемым типом
TaskScheduler	Для управления планированием задач
TaskCompletionSource	Для ручного управления рабочим потоком действий задачи

Рассмотрим расширенные возможности задач, которые ориентированы на параллельное программирование. В частности, будут обсуждаться следующие темы:

- тонкая настройка планирования задачи;
- установка отношения “родительская/дочерняя”, когда одна задача запускается из другой;
- расширенное использование продолжений;
- класс *TaskFactory*.

❖ Библиотека параллельных задач (TPL) позволяет создавать сотни (или даже тысячи) задач с минимальными накладными расходами. Но если

необходимо создавать миллионы задач, то для поддержания эффективности эти задачи понадобится организовывать в более крупные единицы работы. Класс *Parallel* и PLINQ делают автоматически.

- В Visual Studio предусмотрено окно для мониторинга задач (Debug=>Window=>*Parallel Tasks* (Отладка=>Окно=>Параллельные задачи)). Окно *Parallel Tasks* (Параллельные задачи) эквивалентно окну *Threads* (Потоки), но предназначено для задач. Окно *Parallel Stacks* (Параллельные стеки) также поддерживает специальный режим для задач.

Создание и запуск задач

Метод *Task.Run* создает и запускает объект *Task* или *Task<TResult>*. На самом деле *Task.Run* является сокращением для вызова метода *Task.Factory.StartNew*, который предлагает более высокую гибкость через дополнительные перегруженные версии.

Указание объекта состояния

Метод *Task.Factory.StartNew* позволяет указывать объект состояния, который передается целевому методу. Сигнатура целевого метода должна в данном случае содержать одиночный параметр типа *object*:

```
static void Main()
{
  var task = Task.Factory.StartNew(Greet, "Hello");
  task.Wait(); // Ожидать, пока задача завершится.
}
static void Greet(object state)
{ Console.Write(state); } // Hello
```

Такой прием дает возможность избежать затрат на замыкание, требуемое для выполнения лямбда-выражения, которое вызывает метод *Greet*. Это является микро-оптимизацией и редко необходимо на практике, так что мы можем оставить объект состояния для более полезного сценария – назначение задаче значащего имени. Затем для запрашивания имени можно применять свойство *AsyncState*:

```
static void Main()
{
  var task = Task.Factory.StartNew(state => Greet("Hello"),
"Greeting");
  Console.WriteLine(task.AsyncState);    // Greeting
  task.Wait();
}
static void Greet(string message) { Console.WriteLine(message); }
```

- Среда Visual Studio отображает значение свойства *AsyncState* каждой задачи в окне *Parallel Tasks*, так что значащее имя задачи может основательно упростить отладку.

TaskCreationOptions

Настроить выполнение задачи можно за счет указания перечисления *TaskCreationOptions* при вызове *StartNew* (или создании объекта *Task*). *TaskCreationOptions* – перечисление флагов со следующими (комбинируемыми) значениями:

LongRunning, *PreferFairness*, *AttachedToParent*

Значение *LongRunning* предлагает планировщику выделить для задачи поток; это полезно для задач с интенсивным вводом-выводом, а также для длительно выполняющихся задач, которые иначе могут заставить кратко выполняющиеся задачи ожидать чрезмерно долгое время перед тем, как они будут запланированы.

Значение *PreferFairness* сообщает планировщику о необходимости попытаться обеспечить планирование задач в том порядке, в каком они были запущены. Обычно планировщик может поступать иначе, потому что он внутренне оптимизирует планирование задач с использованием локальных очередей захвата работ – оптимизация, позволяющая создавать дочерние задачи без накладных расходов на состязания, которые в противном случае возникли бы при доступе к единственной очереди работ. Дочерняя задача создается путем указания значения *AttachedToParent*.

Дочерние задачи

Когда одна задача запускает другую, можно дополнительно установить отношение “родительская/дочерняя”:

```
Task parent = Task.Factory.StartNew(() =>
{
  Console.WriteLine("I am a parent");

  Task.Factory.StartNew(() =>  //  Отсоединенная задача
  {
    Console.WriteLine("I am detached");
  });

  Task.Factory.StartNew(() =>  // Дочерняя задача
  {
    Console.WriteLine("I am a child");
  }, TaskCreationOptions.AttachedToParent);
});
```

Дочерняя задача специфична тем, что при ожидании завершения родительской задачи ожидаются также и любые ее дочерние задачи. До этой точки поднимаются любые дочерние исключения:

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
var parent = Task.Factory.StartNew(() =>
{
  Task.Factory.StartNew(() => // Дочерняя
  {
    Task.Factory.StartNew(() => { throw null; }, atp); //
Внучатая
  }, atp);
});
// Следующий вызов генерирует исключение NullReferenceException
// (помещенное в оболочку AggregateExceptions):
parent.Wait();
```

Как вскоре будет показано, прием может быть особенно полезным, когда дочерняя задача является продолжением.

Ожидание на множестве задач

Организовать ожидание на одиночной задаче можно либо вызовом ее метода *Wait*, либо обращением к ее свойству *Result* (в

случае *Task<TResult>*). Можно также реализовать ожидание сразу на множестве задач – с помощью статических методов *Task.WaitAll* (ожидание завершения всех указанных задач) и *Task.WaitAny* (ожидание завершения какой-то одной задачи).

Метод *WaitAll* похож на ожидание каждой задачи по очереди, но более эффективен тем, что требует (максимум) одного переключения контекста. Кроме того, если одна или более задач генерируют необработанное исключение, то *WaitAll* по-прежнему ожидает каждую задачу и затем генерирует исключение *AggregateException*, в котором накоплены исключения из всех отказавших задач (ситуация, когда класс *AggregateException* по-настоящему полезен). Ниже показан эквивалентный код:

```
// Предполагается, что t1, t2 и t3 - задачи:
var exceptions = new List<Exception>();
try { t1.Wait(); }
catch (AggregateException ex) { exceptions.Add(ex); }
try { t2.Wait(); }
catch (AggregateException ex) { exceptions.Add(ex); }
try { t3.Wait(); }
catch (AggregateException ex) { exceptions.Add(ex); }
if (exceptions.Count > 0) throw new
AggregateException(exceptions);
```

Вызов *WaitAny* эквивалентен ожиданию события *ManualResetEventSlim*, которое сигнализируется каждой задачей, как только она завершена.

Помимо времени тайм-аута методам *Wait* можно также передавать признак отмены: это позволяет отменить ожидание, но не саму задачу.

Отмена задач

При запуске задачи можно дополнительно передавать признак отмены. Если позже через данный признак произойдет отмена, то задача войдет в состояние *Canceled* (отменена):

```
var cts = new CancellationTokenSource();
CancellationToken token = cts.Token;
cts.CancelAfter (500);

Task task = Task.Factory.StartNew (() =>
{
  Thread.Sleep (1000);
  token.ThrowIfCancellationRequested(); // Проверить запрос отмены
}, token);

try { task.Wait(); }
catch (AggregateException ex)
{
  Console.WriteLine(ex.InnerException
           is TaskCanceledException); // True
  Console.WriteLine(task.IsCanceled); // True
  Console.WriteLine(task.Status);     // Canceled
}
```

TaskCanceledException – подкласс класса *OperationCanceledException*. Если нужно явно сгенерировать исключение *OperationCanceledException* (вместо вызова *token.ThrowIfCancellationRequested*), тогда потребуется передать признак отмены конструктору *OperationCanceledException*. Если это не сделано, то задача не войдет в состояние *TaskStatus.Canceled* и не будет инициировать продолжение *OnlyOnCanceled*.

Если задача отменяется еще до своего запуска, тогда она не будет запланирована – в таком случае исключение *OperationCanceledException* сгенерируется немедленно.

Поскольку признаки отмены распознаются другими API-интерфейсами, их можно передавать другим конструкциям и отмена будет распространяться гладким образом:

```
var cancelSource = new CancellationTokenSource();
CancellationToken token = cancelSource.Token;
Task task = Task.Factory.StartNew(() =>
{
  // Передать признак отмены в запрос PLINQ:
  var query =
someSequence.AsParallel().WithCancellation(token)...
  ...перечислить результаты запроса...
});
```

Вызов *Cancel* на *cancelSource* в рассмотренном примере приведет к отмене запроса PLINQ с генерацией исключения *OperationCanceledException* в теле задачи, которое затем отменит задачу.

- ❖ Признаки отмены, которые можно передавать в методы, подобные *Wait* и *CancelAndWait*, позволяют отменить операцию ожидания, а не саму задачу

Продолжение

Метод *ContinueWith* выполняет делегат сразу после завершения задачи:

```
Task task1 = Task.Factory
     .StartNew(() => Console.Write("antecedant.."));
Task task2 = task1
     .ContinueWith(ant => Console.Write("..continuation"));
```

Как только задача *task1* (предшественник) завершается, отказывает или отменяется, запускается задача *task2* (продолжение). (Если задача *task1* была завершена до того, как выполнилась вторая строка кода, то задача *task2* будет запланирована для выполнения немедленно.) Аргумент *ant*, переданный лямбда-выражению продолжения, представляет собой ссылку на предшествующую задачу. Сам метод *ContinueWith* возвращает задачу, облегчая добавление дополнительных продолжений.

По умолчанию предшествующая задача и задача продолжения могут выполняться в разных потоках. Указав *TaskContinuationOptions.ExecuteSynchronously* при вызове *ContinueWith*, можно заставить их выполняться в одном и том же потоке: это позволяет улучшить производительность при мелкомодульных продолжениях за счет уменьшения косвенности.

Продолжение и *Task<TResult>*

Подобно обычным задачам продолжения могут иметь тип *Task<TResult>* и возвращать данные. В следующем примере мы вычисляем *Math.Sqrt(8*2)* с применением последовательности соединенных в цепочку задач и выводим результат:

```
Task.Factory.StartNew<int>(() => 8)
  .ContinueWith(ant => ant.Result * 2)
  .ContinueWith(ant => Math.Sqrt(ant.Result))
  .ContinueWith(ant => Console.WriteLine(ant.Result)); // 4
```

Из-за стремления к простоте этот пример получился несколько неестественным; в реальности такие лямбда-выражения могли бы вызывать функции с интенсивными вычислениями.

Продолжение и исключения

Продолжение может узнать, отказал ли предшественник, запросив свойство *Exception* предшествующей задачи или просто вызвав метод *Result/Wait* и перехватив результирующее исключение *AggregateException*. Если предшественник отказал, и то же самое сделало продолжение, тогда исключение считается необнаруженным; в таком случае при последующей обработке задачи сборщиком мусора будет инициировано статическое событие *TaskScheduler.UnobservedTaskException*.

Безопасный шаблон предполагает повторную генерацию исключений предшественника. До тех пор, пока ожидается продолжение, исключение будет распространяться и повторно генерироваться для ожидающей задачи:

```
Task continuation = Task.Factory.StartNew(() => { throw null; })
          .ContinueWith(ant =>
  {
    ant.Wait();
    // Продолжить обработку ...
  });
continuation.Wait();     // Исключение теперь передается обратно
                         // вызывающей задаче.
```

Другой способ иметь дело с исключениями предусматривает указание разных продолжений для исходов с и без исключений, что делается с помощью перечисления *TaskContinuationOptions*:

```
Task task1 = Task.Factory.StartNew(() => { throw null; });
Task error = task1.ContinueWith(ant =>
               Console.Write(ant.Exception),
               TaskContinuationOptions.OnlyOnFaulted);
Task ok = task1.ContinueWith(ant =>
               Console.Write("Success!"),
               TaskContinuationOptions.NotOnFaulted);
```

Как вскоре будет показано, такой шаблон особенно удобен в сочетании с дочерними задачами.

Следующий расширяющий метод “поглощает” необработанные исключения задачи:

```
public static void IgnoreExceptions(this Task task)
{
  task.ContinueWith(t => { var ignore = t.Exception; },
    TaskContinuationOptions.OnlyOnFaulted);
}
```

(Метод может быть улучшен добавлением кода для регистрации исключения.) Вот как его можно использовать:

```
Task.Factory.StartNew (() => { throw null; }).IgnoreExceptions();
```

Продолжение и дочерние задачи

Мощная особенность продолжений связана с тем, что они запускаются, только когда завершены все дочерние задачи (рис. 1). В этой точке любые исключения, сгенерированные дочерними задачами, маршализируются в продолжение.

В следующем примере мы начинаем три дочерние задачи, каждая из которых генерирует исключение *NullReferenceException*.

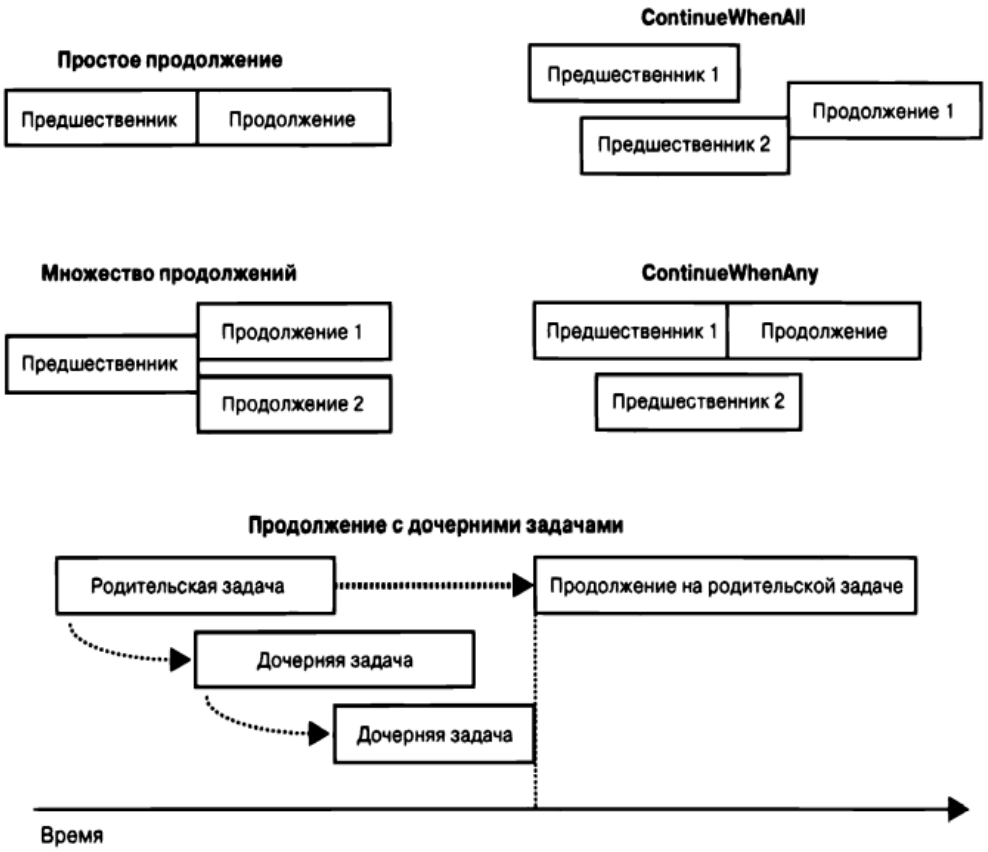

Рис. 1. Продолжения

Затем мы перехватываем все исключения сразу через продолжение на родительской задаче:

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
Task.Factory.StartNew(() =>
{
  Task.Factory.StartNew(() => { throw null; }, atp);
  Task.Factory.StartNew(() => { throw null; }, atp);
  Task.Factory.StartNew(() => { throw null; }, atp);
})
.ContinueWith(p =>
    Console.WriteLine(p.Exception),
    TaskContinuationOptions.OnlyOnFaulted);
```

Условные продолжения

По умолчанию продолжение планируется безусловным образом – независимо от того, завершена предшествующая задача, сгенерировано исключение или задача была отменена. Такое поведение можно изменить с помощью набора (комбинируемых)

флагов, определенных в перечислении *TaskContinuationOptions*. Вот три основных флага, которые управляют условным продолжением:

```
NotOnRanToCompletion = 0x10000,
NotOnFaulted = 0x20000,
NotOnCanceled = 0x40000,
```

Флаги являются субтрактивными в том смысле, что чем больше их применяется, тем менее вероятно выполнение продолжения. Для удобства также предоставляются следующие заранее скомбинированные значения:

```
OnlyOnRanToCompletion = NotOnFaulted | NotOnCanceled,
OnlyOnFaulted = NotOnRanToCompletion | NotOnCanceled,
OnlyOnCanceled = NotOnRanToCompletion | NotOnFaulted
```

(Объединение всех флагов *Not*(NotOnRanToCompletion*, *NotOnFaulted*, *NotOnCanceled*) бессмысленно, т.к. в результате продолжение будет всегда отменяться.)

Наличие "*RanToCompletion*" в имени означает успешное завершение предшествующей задачи – без отмены или необработанных исключений.

Наличие "*Faulted*" в имени означает, что в предшествующей задаче было сгенерировано необработанное исключение.

Наличие "*Canceled*" в имени означает одну из следующих двух ситуаций.

- Предшествующая задача была отменена через ее признак отмены. Другими словами, в предшествующей задаче было сгенерировано исключение *OperationCanceledException*, свойство *CancellationToken* которого соответствует тому, что было передано предшествующей задаче во время запуска.
- Предшествующая задача была неявно отменена, поскольку она не удовлетворила предикат условного продолжения.

Важно понимать, что когда продолжение не выполнилось из-за упомянутых флагов, оно не забыто и не отброшено – это продолжение отменено. Другими словами, любые продолжения на самом отмененном продолжении затем запустятся – если только вы не указали в условии флаг *NotOnCanceled*. Например, взгляните на приведенный далее код:

```
Task t1 = Task.Factory.StartNew(...);
Task fault = t1.ContinueWith(ant => Console.WriteLine("fault"),
                        TaskContinuationOptions.OnlyOnFaulted);
Task t3 = fault.ContinueWith(ant => Console.WriteLine("t3"));
```

Несложно заметить, что задача *t3* всегда будет запланирована – даже если *t1* не генерирует исключение (рис. 2). Причина в том, что если задача *t1* завершена успешно, тогда задача *fault* будет отменена, и с учетом отсутствия ограничений продолжения задача *t3* будет запущена безусловным образом.

Если нужно, чтобы задача *t3* выполнялась, только если действительно была запущена задача *fault*, то потребуется поступить так:

```
Task t3 = fault.ContinueWith(ant => Console.WriteLine("t3"),
                    TaskContinuationOptions.NotOnCanceled);
```

(В качестве альтернативы мы могли бы указать *OnlyOnRanToCompletion*; разница в том, что тогда задача *t3* не запустилась бы в случае генерации исключения внутри задачи *fault*.)

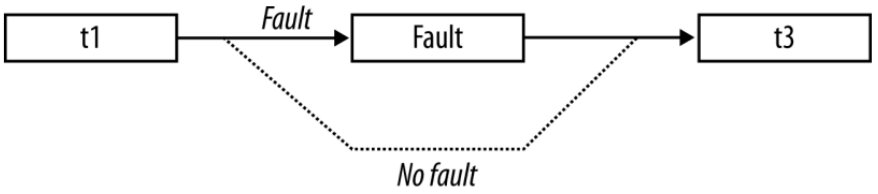

Рис. 2. Условные продолжения

Продолжение на основе множества предшествующих задач

С помощью методов *ContinueWhenAll* и *ContinueWhenAny* класса *TaskFactory* выполнение продолжения можно планировать на основе завершения множества предшествующих задач. Однако эти методы стали избыточными после появления комбинаторов задач. В частности, при наличии следующих задач:

```
var task1 = Task.Run (() => Console.Write ("X"));
var task2 = Task.Run(() => Console.Write("Y"));
```

вот как можно запланировать выполнение продолжения, когда обе они завершатся:

```
var continuation = Task.Factory.ContinueWhenAll(
  new[] { task1, task2 }, tasks => Console.WriteLine("Done"));
```

Тот же результат легко получить с помощью комбинатора задач *WhenAll*:

```
var continuation = Task.WhenAll(task1, task2)
.ContinueWith(ant => Console.WriteLine("Done”));
```

Множество продолжений на единственной предшествующей задаче

Вызов *ContinueWith* более одного раза на той же самой задаче создает множество продолжений на единственном предшественнике. Когда предшественник завершается, все продолжения запускаются вместе (если только не было указано значение *TaskContinuationOptions.ExecuteSynchronously*, из-за чего продолжения будут выполняться последовательно). Следующий код ожидает одну секунду, а затем выводит на консоль либо XY, либо YX:

```
var t = Task.Factory.StartNew(() => Thread.Sleep(1000));
t.ContinueWith(ant => Console.Write("X"));
t.ContinueWith(ant => Console.Write("Y"));
```

Планировщики задач

Планировщик задач распределяет задачи по потокам и представлен абстрактным классом *TaskScheduler*. В .NET Framework предлагаются две конкретные реализации: стандартный планировщик, который работает в тандеме с пулом потоков CLR, и планировщик контекста синхронизации. Последний предназначен (главным образом) для содействия в работе с потоковой моделью WPF и Windows Forms, которая требует, чтобы доступ к элементам управления пользовательского интерфейса осуществлялся только из создавшего их потока. Захватив такой планировщик, мы можем сообщить задаче или продолжению о выполнении в этом контексте:

```
// Предположим, что мы находимся в потоке пользовательского
// интерфейса внутри приложения Windows Forms или WPF:
_uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
```

Предполагая, что *Foo* – метод с интенсивными вычислениями, возвращающий строку, *lblResult* – метка WPF или Windows Forms, вот как можно было бы безопасно обновить метку после завершения операции:

```
Task.Run(() => Foo())
  .ContinueWith(ant =>
  lblResult.Content = ant.Result, _uiScheduler);
```

Разумеется, для действий подобного рода чаще будут использоваться асинхронные функции C#.

Возможно также написание собственного планировщика задач (путем создания подкласса *TaskScheduler*), хотя это делается только

в очень специализированных сценариях. Для специального планирования чаще всего будет применяться класс *TaskCompletionSource*.

TaskFactory

Когда вызывается *Task.Factory*, происходит обращение к статическому свойству класса *Task*, которое возвращает стандартный объект фабрики задач, т.е. *TaskFactory*. Назначение фабрики задач заключается в создании задач – в частности, трех их видов:

- “обычных” задач (через метод *StartNew*);
- продолжений с множеством предшественников (через методы *ContinueWhenAll* и *ContinueWhenAny*);
- задач, которые являются оболочками для методов, следующих устаревшему шаблону APM.

Еще один способ создания задач предусматривает создание экземпляра *Task* и вызов метода *Start*. Тем не менее подобным образом можно создавать только “обычные” задачи, но не продолжения.

Создание собственных фабрик задач

Класс *TaskFactory* – не абстрактная фабрика: на самом деле вы можете создавать объекты данного класса, что удобно, когда нужно многократно создавать задачи с использованием тех же самых (нестандартных) значений для *TaskCreationOptions*, *TaskContinuationOptions* или *TaskScheduler*. Например, если требуется многократно создавать длительно выполняющиеся родительские задачи, тогда специальную фабрику можно построить следующим образом:

```
var factory = new TaskFactory(
  TaskCreationOptions.LongRunning |
  TaskCreationOptions.AttachedToParent,
  TaskContinuationOptions.None);
```

Затем создание задач сводится просто к вызову метода *StartNew* на фабрике:

```
Task task1 = factory.StartNew(Method1);
Task task2 = factory.StartNew(Method2);
```

Специальные опции продолжения применяются в случае вызова методов *ContinueWhenAll* и *ContinueWhenAny*.