Задачи

Введение

Поток – это низкоуровневый инструмент для организации параллельной обработки и, будучи таковым, он обладает описанными ниже ограничениями.

- Несмотря на простоту передачи данных запускаемому потоку, не существует простого способа получить “возвращаемое значение” обратно из потока, для которого выполняется метод *Join*. Потребуется предусмотреть какое-то разделяемое поле. И если операция сгенерирует исключение, то его перехват и распространение будут сопряжены с аналогичными трудностями.
- После завершения потоку нельзя сообщить о том, что необходимо запустить что-нибудь еще; взамен к нему придется присоединяться с помощью метода *Join* (блокируя собственный поток в процессе).

Указанные ограничения препятствуют реализации мелкомодульного параллелизма; другими словами, они затрудняют формирование более крупных параллельных операций за счет комбинирования мелких операций. В свою очередь возникает более высокая зависимость от ручной синхронизации (блокировки, выдачи сигналов и т.д.) и проблем, которые ее сопровождают.

Прямое применение потоков также оказывает влияние на производительность. И если требуется запустить сотни или тысячи параллельных операций с интенсивным вводом-выводом, то подход на основе потоков повлечет за собой затраты сотен или тысяч мегабайтов памяти исключительно на накладные расходы, связанные с потоками.

Класс *Task*, реализующий задачу, помогает решить все упомянутые проблемы. В сравнении с потоком тип *Task* – абстракция более высокого уровня, т.к. он представляет параллельную операцию, которая может быть или не быть подкреплена потоком. Задачи поддерживают возможность композиции (их можно соединять вместе с использованием продолжения). Они могут работать с пулом потоков в целях снижения задержки во время запуска, а с помощью класса *TaskCompletionSource* задачи позволяют задействовать подход с обратными вызовами, при котором потоки вообще не будут ожидать завершения операций с интенсивным вводом-выводом.

Типы *Task* появились в версии .NET Framework 4.0 как часть библиотеки параллельного программирования. Однако с тех пор они были усовершенствованы (за счет применения *объектов ожидания (awaiter)*), чтобы функционировать столь же эффективно в более универсальных сценариях реализации параллелизма, и имеют поддерживающие типы для асинхронных функций C#.

Запуск задачи

Простейший способ запуска задачи, подкрепленной потоком, предусматривает вызов статического метода *Task.Run* (класс *Task* находится в пространстве имен *System.Threading.Tasks*). Упомянутому методу нужно просто передать делегат *Action*:

```
Task.Run (() => Console.WriteLine ("Foo"));
```

❖ По умолчанию задачи используют потоки из пула, которые являются фоновыми потоками. Это означает, что когда главный поток завершается, то завершаются и любые созданные вами задачи. Следовательно, чтобы запускать приводимые здесь примеры из консольного приложения, потребуется блокировать главный поток после старта задачи (скажем, ожидая завершения задачи или вызывая метод *Console.ReadLine*):

```
Task.Run(() => Console.WriteLine("Foo”));
Console.ReadLine();
```

В примерах для LINQPad вызов *Console .ReadLine* опущен, т.к. процесс LINQPad удерживает фоновые потоки в активном состоянии.

Вызов метода *Task.Run* в подобной манере похож на запуск потока следующим образом (за исключением влияния пула потоков, о котором речь пойдет чуть позже):

```
new Thread (() => Console.WriteLine ("Foo")).Start();
```

Метод *Task.Run* возвращает объект *Task*, который можно применять для мониторинга хода работ, почти как в случае объекта *Thread*. (Тем не менее, обратите внимание, что мы не вызываем метод *Start* после вызова *Task.Run*, т.к. метод *Run* создает “горячие” задачи; взамен можно воспользоваться конструктором класса *Task* и создавать “холодные” задачи, хотя на практике так поступают редко.)

Отслеживать состояние выполнения задачи можно с помощью ее свойства *Status*.

Wait

Вызов метода *Wait* на объекте задачи приводит к блокированию до тех пор, пока она не будет завершена, и эквивалентен вызову метода *Join* на объекте потока:

```
Task task = Task.Run(() =>
{
  Thread.Sleep(2000);
  Console.WriteLine("Foo");
});
Console.WriteLine(task.IsCompleted); // False
task.Wait(); // Блокируется вплоть до завершения задачи
```

Метод *Wait* позволяет дополнительно указывать тайм-аут и признак отмены для раннего завершения ожидания.

Длительно выполняющиеся задачи

По умолчанию среда CLR запускает задачи в потоках из пула, что идеально в случае кратковременных задач с интенсивными

вычислениями. Для длительно выполняющихся и блокирующих операций (как в предыдущем примере) использованию потоков из пула можно воспрепятствовать, как показано ниже:

```
Task task = Task.Factory.StartNew(() => ...,
                                  TaskCreationOptions.LongRunning);
```

❖ Запуск *одной* длительно выполняющейся задачи в потоке из пула не приведет к проблеме; производительность может пострадать, когда параллельно запускается несколько длительно выполняющихся задач (особенно таких, которые производят блокирование). И в этом случае обычно существуют более эффективные решения, нежели указание *TaskCreationOptions.LongRunning*:

- если задачи являются интенсивными в плане ввода-вывода, то вместо потоков следует применять класс *TaskCompletionSource* и асинхронные функции, которые позволяют реализовать параллельное выполнение с обратными вызовами (продолжениями);
- если задачи являются интенсивными в плане вычислений, то отрегулировать параллелизм для таких задач позволит очередь производителей/потребителей, избегая ограничения других потоков и процессов.

Возвращение значений

Класс *Task* имеет обобщенный подкласс по имени *Task<TResult>*, который позволяет задаче выдавать возвращаемое значение. Для получения объекта *Task<TResult>* можно вызвать метод *Task.Run* с делегатом *Func<TResult>* (или совместимым лямбда-выражением) вместо делегата *Action*:

```
Task<int> task = Task.Run(() =>
{ Console.WriteLine("Foo"); return 3; });
```

Позже можно получить результат, запросив свойство *Result*. Если задача еще не закончилась, то доступ к этому свойству заблокирует текущий поток до тех пор, пока задача не завершится:

```
int result = task.Result; // Блокирует поток, если задача еще не
завершена
Console.WriteLine(result); // 3
```

В следующем примере создается задача, которая использует LINQ для подсчета количества простых чисел в первых трех миллионах (начиная с 2) целочисленных значений:

```
Task<int> primeNumberTask = Task.Run(() =>
Enumerable.Range(2, 3000000).Count(n =>
Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)));
Console.WriteLine("Task running...");
Console.WriteLine("The answer is " + primeNumberTask.Result);
```

Код выводит строку “Task running...” (Задача выполняется...) и спустя несколько секунд выдает ответ 216815.

❖ Класс *Task<Result>* можно воспринимать как “будущее”, поскольку он инкапсулирует свойство *Result*, которое станет доступным позже во времени.

Исключения

В отличие от потоков задачи без труда распространяют исключения. Таким образом, если код внутри задачи генерирует необработанное исключение (другими словами, если задача *отказывает*), то это исключение автоматически повторно сгенерируется при вызове метода *Wait* или доступе к свойству *Result* класса *Task<TResult>*:

```
// Запустить задачу, которая генерирует исключение NullReferenceException:
Task task = Task.Run(() => { throw null; });
try
{
  task.Wait();
}
catch (AggregateException aex)
{
  if (aex.InnerException is NullReferenceException)
    Console.WriteLine("Null!");
  else
    throw;
}
```

(Среда CLR помещает исключение в оболочку *AggregateException* для нормальной работы в сценариях параллельного программирования.)

Проверить, отказала ли задача, можно без повторной генерации исключения посредством свойств *IsFaulted* и *IsCanceled* класса *Task*. Если оба свойства возвращают *false*, то ошибки не возникали; если *IsCanceled* равно *true*, то для задачи было сгенерировано исключение *OperationCanceledException*; если *IsFaulted* равно *true*, то было сгенерировано исключение другого типа и на ошибку укажет свойство *Exception*.

Исключения и автономные задачи

В автономных задачах, работающих по принципу “установить и забыть” (для которых не требуется взаимодействие через метод *Wait* или свойство *Result* либо продолжение, делающее то же самое), общепринятой практикой является явное написание кода обработки исключений во избежание молчаливого отказа (в точности, как с потоком).

❖ Игнорировать исключения нормально в ситуации, когда исключение только указывает на неудачу при получении результата, который больше не интересует. Например, если пользователь отменяет запрос на загрузку веб-страницы, то мы не должны переживать, если выяснится, что веб-страница не существует.
Игнорировать исключения проблематично, когда исключение указывает на ошибку в программе, по двум причинам:

- ошибка может оставить программу в недопустимом состоянии;
- в результате ошибки позже могут возникнуть другие исключения, и отказ от регистрации первоначальной ошибки может затруднить диагностику.

Подписаться на необнаруженные исключения на глобальном уровне можно через статическое событие *TaskScheduler.UnobservedTaskException*; обработка этого события и регистрация ошибки нередко имеют смысл.

Есть пара интересных нюансов, касающихся того, какое исключение считать необнаруженным.

- Задачи, ожидающие с указанием тайм-аута, будут генерировать необнаруженное исключение, если ошибки возникают после истечения интервала тайм-аута.
- Действие по проверке свойства *Exception* задачи после ее отказа помечает исключение как обнаруженное.

Продолжение

Продолжение сообщает задаче о том, что после завершения она должна продолжиться и делать что-то другое. Продолжение обычно реализуется посредством обратного вызова, который выполняется один раз после завершения операции. Существуют два способа присоединения признака продолжения к задаче. Первый из них особенно важен, поскольку применяется асинхронными функциями C#, что вскоре будет показано. Мы можем продемонстрировать его на примере с подсчетом простых чисел:

```
Task<int> primeNumberTask = Task.Run(() =>
    Enumerable.Range(2, 3000000).Count(n =>
        Enumerable.Range(2, (int)Math.Sqrt(n) - 1)
                  .All(i => n % i > 0)));
var awaiter = primeNumberTask.GetAwaiter();
awaiter.OnCompleted(() =>
{
  int result = awaiter.GetResult();
  Console.WriteLine(result); // Выводит значение result
}
```

Вызов метода *GetAwaiter* на объекте задачи возвращает *объект ожидания*, метод *OnCompleted* которого сообщает *предыдущей* задаче (*primeNumberTask*) о необходимости выполнить делегат, когда она завершится (или откажет). Признак продолжения допускается присоединять к уже завершенным задачам; в таком случае продолжение будет запланировано для немедленного выполнения.

❖ Объект ожидания (*awaiter*) – это любой объект, открывающий доступ к двум методам, которые мы только что видели (*OnCompleted* и *GetResult*), и к булевскому свойству по имени *IsCompleted*. Никакого интерфейса или базового класса для унификации указанных членов не предусмотрено (хотя метод *OnCompleted* является частью интерфейса *INotifyCompletion*).

Если предшествующая задача терпит отказ, то исключение генерируется повторно, когда код продолжения вызывает метод *awaiter.GetResult*. Вместо вызова *GetResult* мы могли бы просто обратиться к свойству *Result* предшествующей задачи. Преимущество вызова *GetResult* связано с тем, что в случае отказа предшествующей задачи исключение генерируется напрямую без помещения в оболочку *AggregateException*, позволяя писать более простые и чистые блоки *catch*.

Для необобщенных задач метод *GetResult* не имеет возвращаемого значения. Его польза состоит единственно в повторной генерации исключений .

Если присутствует контекст синхронизации, тогда метод *OnCompleted* его автоматически захватывает и отправляет ему признак продолжения. Это очень удобно в обогащенных клиентских приложениях, т.к. признак продолжения возвращается обратно потоку пользовательского интерфейса. Тем не менее, в случае библиотек подобное обычно нежелательно, потому что относительно затратный возврат в поток пользовательского интерфейса должен происходить только раз при покидании библиотеки, а не между вызовами методов. Следовательно, его можно аннулировать с помощью метода *ConfigureAwait*:

```
var awaiter = primeNumberTask.ConfigureAwait (false).GetAwaiter();
```

Когда контекст синхронизации отсутствует (или применяется *ConfigureAwait* (*false*)), продолжение будет (в общем случае) выполняться в том же самом потоке, избегая ненужных накладных расходов.

Другой способ присоединить продолжение предполагает вызов метода *ContinueWith* задачи:

```
primeNumberTask.ContinueWith(antecedent =>
{
  int result = antecedent.Result;
  Console.WriteLine(result); // Выводит 123
});
```

Сам метод *ContinueWith* возвращает объект *Task*, который полезен, если планируется присоединение дальнейших признаков продолжения. Однако если задача отказывает, то в приложениях с пользовательским интерфейсом придется иметь дело напрямую с исключением *AggregateException* и предусмотреть дополнительный код для маршализации продолжения. В контекстах, не связанных с пользовательским интерфейсом, потребуется указывать *TaskContinuationOptions.ExecuteSynchronously*, если продолжение должно выполняться в том же потоке, иначе произойдет возврат в пул потоков. Метод *ContinueWith* особенно удобен в сценариях параллельного программирования.

TaskCompletionSource

Ранее уже было указано, что метод *Task.Run* создает задачу, которая запускает делегат в потоке из пула (или не из пула). Еще один способ создания задачи заключается в использовании класса *TaskCompletionSource*.

Класс *TaskCompletionSource* позволяет создавать задачу из любой операции, которая начинается и через некоторое время заканчивается. Он работает путем предоставления “подчиненной ” задачи, которой вы управляете вручную, указывая, когда операция завершилась или отказала. Это идеально для работы с интенсивным вводом-выводом: вы получаете все преимущества задач (с их возможностями передачи возвращаемых значений, исключений и признаков продолжения), не блокируя поток на период выполнения операции.

Для применения класса *TaskCompletionSource* нужно просто создать его экземпляр. Данный класс открывает доступ к свойству *Task*, возвращающему объект задачи, для которой можно организовать ожидание и присоединить признак продолжения – как делается с любой другой задачей . Тем не менее, такая задача полностью управляется объектом *TaskCompletionSource* с помощью следующих методов:

```
public class TaskCompletionSource<TResult>
{
  public void SetResult(TResult result);
  public void SetException(Exception exception);
  public void SetCanceled();
  public bool TrySetResult(TResult result);
  public bool TrySetException(Exception exception);
  public bool TrySetCanceled();
  public bool TrysetCanceled(CancellationToken cancellationToken);
    ...
}
```

Вызов одного из перечисленных методов передает сигнал задаче, помещая ее в состояние завершения, отказа или отмены. Предполагается, что вы будете вызывать любой из этих методов в точности один раз: в случае повторного вызова методы *SetResult*, *SetException* и *SetCanceled* сгенерируют исключение, а методы *Try** возвратят *false*.

В следующем примере после пятисекундного ожидания выводится число 42:

```
var tcs = new TaskCompletionSource<int>();
new Thread(() => { Thread.Sleep(5000); tcs.SetResult(42); })
{ IsBackground = true }
    .Start();
Task<int> task = tcs.Task;// "Подчиненная" задача
Console.WriteLine(task.Result);// 42
```

Можно реализовать собственный метод *Run* с использованием класса *TaskCompletionSource*:

```
Task<TResult> Run<TResult>(Func<TResult> function)
{
  var tcs = new TaskCompletionSource<TResult>();
  new Thread(() =>
  {
    try { tcs.SetResult(function()); }
    catch (Exception ex) { tcs.SetException(ex); }
  }).Start();
  return tcs.Task;
}
...
Task<int> task = Run(() => { Thread.Sleep(5000); return 42; });
```

Вызов данного метода эквивалентен вызову *Task.Factory.StartNew* с параметром *TaskCreationOptions.LongRunning* для запроса потока не из пула.

Реальная мощь класса *TaskCompletionSource* состоит в возможности создания задач, не связывающих потоки. Например, рассмотрим задачу, которая ожидает пять секунд и затем возвращает число 42. Мы можем реализовать ее без потока с применением класса *Timer*, который с помощью CLR (и в свою очередь ОС) инициирует событие каждые x миллисекунд:

```
Task<int> GetAnswerToLife()
{
  var tcs = new TaskCompletionSource<int>();
  // Создать таймер, который инициирует событие раз в 5000 мс:
  var timer = new System.Timers.Timer(5000) { AutoReset = false };
   timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult(42);
};
  timer.Start();
  return tcs.Task;
}
```

Таким образом, наш метод возвращает объект задачи, которая завершается спустя пять секунд с результатом 42. Присоединив к задаче продолжение, мы можем вывести ее результат, не блокируя *ни одного потока*:

```
var awaiter = GetAnswerToLife().GetAwaiter();
awaiter.OnCompleted(() => Console.WriteLine(awaiter.GetResult()));
```

Мы могли бы сделать код более полезным и превратить его в универсальный метод *Delay*, параметризировав время задержки и избавившись от возвращаемого значения. Это означало бы возвращение объекта *Task* вместо *Task<int>*. Тем не менее, необобщенной версии *TaskCompletionSource* не существует, а потому мы не можем напрямую создавать необобщенный объект *Task*. Обойти ограничение довольно просто: поскольку класс *Task<TResult>* является производным от *Task*, мы создаем *TaskCompletionSource<какой-то-тип>* и затем неявно преобразуем получаемый экземпляр *Task<какой-то-тип>* в *Task*, примерно так:

```
var tcs = new TaskCompletionSource<object>();
Task task = tcs.Task;
```

Теперь можно реализовать универсальный метод *Delay*:

```
Task Delay(int milliseconds)
{
  var tcs = new TaskCompletionSource<object>();
  var timer = new System.Timers.Timer(milliseconds)
                    { AutoReset = false };
  timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult(null);
};
  timer.Start();
  return tcs.Task;
}
```

❖ В версии .NET 5 появился необобщенный класс *TaskCompletionSource*, так что если вы нацелите проект на .NET 5 или последующую версию, то сможете вместо *TaskCompletionSource<object>* указать *TaskCompletionSource*.

Ниже показано, как использовать данный метод для вывода числа 42 после пятисекундной паузы:

```
Delay(5000).GetAwaiter().OnCompleted(() => Console.WriteLine(42));
```

Такое применение класса *TaskCompletionSource* без потока означает, что поток будет занят, только когда запускается продолжение, т.е. спустя пять секунд. Мы можем продемонстрировать это, запустив 10000 таких операций

одновременно и не столкнувшись с ошибкой или чрезмерным потреблением ресурсов:

```
for (int i = 0; i < 10000; i++)
  Delay(5000).GetAwaiter()
  .OnCompleted(() => Console.WriteLine(42));
```

❖ Таймеры инициируют свои обратные вызовы на потоках из пула, так что через пять секунд пул потоков получит 10000 запросов вызова *SetResult(null)* на *TaskCompletionSource*. Если запросы поступают быстрее, чем они могут быть обработаны, тогда пул потоков отреагирует постановкой их в очередь и последующей обработкой на оптимальном уровне параллелизма для ЦП. Это идеально в ситуации, когда привязанные к потокам задания являются кратковременными, что в данном случае справедливо: привязанное к потоку задание просто вызывает метод *SetResult* и либо осуществляет отправку признака продолжения контексту синхронизации (в приложении с пользовательским интерфейсом), либо выполняет само продолжение (*Console.WriteLine(42)*).

Task.Delay

Только что реализованный метод *Delay* достаточно полезен своей доступностью в качестве статического метода класса *Task*:

```
Task.Delay(5000).GetAwaiter()
  .OnCompleted(() => Console.WriteLine (42));
```

или:

```
Task.Delay(5000).ContinueWith(ant => Console.WriteLine (42));
```

Метод *Task.Delay* является асинхронным эквивалентом метода *Thread.Sleep*.