3 Асинхронное программирование на основе задач

Многие задачи программирования предусматривают запуск и реагирование на асинхронную работу. Мы создаем распределенные программы, функционирующие на более чем одной физической или виртуальной машине. Многие приложения охватывают несколько потоков, процессов, контейнеров, виртуальных или физических машин. Однако асинхронное программирование не является синонимом многопоточного программирования. Современное программирование означает управление асинхронной работой. Такая работа может включать ожидание очередного сетевого пакета или пользовательского ввода.

Язык C# вместе с рядом классов в инфраструктуре .NET Framework предоставляет инструменты, которые облегчают асинхронное программирование. Асинхронное программирование может быть трудным, но когда вы запомните несколько важных практик, оно станет проще, чем когда-либо.

Совет 27. Используйте асинхронные методы для выполнения асинхронной работы

Асинхронные методы предлагают более удобный способ построения асинхронных алгоритмов. Вы пишете основную логику для асинхронного метода так, как если бы он был синхронным методом. Тем не менее, путь выполнения не будет аналогичным синхронному методу. В случае синхронного метода вы пишете последовательность инструкций и ожидаете, что они выполнятся в порядке их написания. В случае асинхронного метода это не обязательно так. Асинхронный метод может возвращать управление до того, как выполнится вся написанная вами логика. Затем в более позднее время в ответ на завершение задачи метод возобновит выполнение там, где он его оставил, пока ваша программа продолжает работу в своем обычном потоке. Если вы *не* понимаете представленный процесс, тогда он может выглядеть похожим на магию. При *не очень хорошем* понимании процесс может показаться сильно запутанным и породить больше вопросов, чем ответов. Читайте дальше, чтобы обрести *полное* понимание того, как компилятор трансформирует ваш код в асинхронные методы. Вы научитесь анализировать асинхронный код, оценивая алгоритмы, которые он описывает, и понимать, как такой код выполняется по мере его следования через инструкции и задачи.

Начнем с простейшего примера: асинхронного метода, который на самом деле выполняется синхронно:

```
static async Task SomeMethodAsync()
{
  Console.WriteLine("Вход в SomeMethodAsync()");
  Task awaitable = SomeMethodReturningTask();

  Console.WriteLine("Внутри SomeMethodAsync(), перед await");
  var result = await awaitable;
  Console.WriteLine("Внутри SomeMethodAsync(), после await");
}
```

В ряде случаев асинхронная работа может завершиться до ожидания первой задачи. Возможно, проектировщик библиотеки предусмотрел кеш, из которого вы можете извлечь уже загруженное значение. Когда вы ожидаете начальную задачу, она завершается и выполнение продолжается синхронно со следующей инструкции. Остаток метода выполняется до конца, а результат упаковывается в объект `Task` и возвращается. Все происходит синхронным образом. Когда управление покидает метод, он возвращает завершенный объект `Task` и вызывающий код также продолжает работу синхронно наряду с ожиданием завершения этой задачи. До сих пор процесс должен быть знакомым любому разработчику.

Но что случится в том же самом методе, если результат недоступен, когда задача ожидается? Поток управления становится более сложным. До появления в языке ключевых слов `async` и `await` приходилось конфигурировать обратный вызов для обработки возврата из асинхронной задачи. Он мог бы иметь форму либо обработчика событий, либо делегата какого-то вида. Теперь все гораздо проще. Чтобы исследовать асинхронную обработку, давайте сначала рассмотрим происходящее с концептуальной точки зрения, не касаясь того, как язык реализует такое поведение.

По достижении инструкции `await` управление покидает метод. Метод возвращает объект `Task`, который указывает, что асинхронная работа пока еще не завершена. Именно здесь происходит вся магия: когда ожидаемая задача завершается, этот метод продолжает выполнение с инструкции, следующей сразу за `await`. Он продолжает делать свою работу, а после ее завершения обновляет возвращенный ранее объект `Task` окончательным результатом. Данная задача уведомляет любой ожидающий его код о том, что он завершен. Затем такие сегменты кода могут продолжить выполнение с точки, где они были прерваны ожиданием задачи.

Лучший способ проанализировать поток управления во время описанного процесса — прогнать ряд примеров в отладчике. Выполните пошагово код с выражениями `await` и посмотрите, как продолжается поток выполнения.

Вы также можете счесть полезной аналогию с реальными асинхронными задачами. Возьмем задачи, связанные с приготовлением домашней пиццы. Вы начинаете с синхронного приготовления теста. Затем вы можете запустить асинхронный процесс, чтобы дать возможность тесту подняться. Пока задача находится на стадии продвижения, вы можете продолжить, занявшись подготовкой соуса. После того, как соус сделан, вы можете ожидать завершения задачи поднятия теста. Потом вы можете запустить асинхронную задачу по разогреву духовки. Пока она выполняется,

вы можете начать формировать пиццу. Наконец, дождавшись подходящей температуры в духовке, вы помещаете пиццу в духовку, чтобы испечь.

Теперь давайте уберем магию, объяснив, как реализован весь процесс. Когда компилятор обрабатывает асинхронный метод, он строит механизмы для запуска асинхронной работы и выполнения дальнейших инструкций после ее завершения. Интересные изменения происходят в выражении `await`. Компилятор строит структуры данных и делегирует работу так, что выполнение может продолжаться с инструкции, следующей непосредственно за выражением `await`. Структуры данных гарантируют сохранение значений всех локальных переменных. Компилятор конфигурирует продолжение на основе ожидаемой задачи, так что когда задача завершена, продолжение переходит обратно в то же самое место внутри метода. По существу компилятор генерирует делегат для кода, который находится после выражения `await`. Компилятор записывает информацию о состоянии, чтобы обеспечить вызов этого делегата, когда ожидаемая задача завершается.

После завершения ожидаемой задачи инициируется событие, которое указывает на ее завершение. Производится повторный вход в метод и состояние восстанавливается. Кажется, что управление возвратилось в код, который ранее оставило, т.е. состояние восстановлено и выполнение переходит в соответствующее место. Это похоже на то, что происходит, когда выполнение продолжается после синхронного вызова: состояние для метода установлено и выполнение переходит в точку, следующую после вызова метода. При выполнении остатка метода его работа завершается, ранее возвращенный объект `Task` обновляется, и события, сигнализирующие о завершении, генерируются.

Когда задача завершается, механизм уведомлений вызывает асинхронный метод, который продолжает свое выполнение. За реализацию такого поведения несет ответственность класс контекста синхронизации `SynchronizationContext`. Этот класс гарантирует, что как только асинхронный метод возобновляет работу после завершения ожидаемой задачи, среда и контекст совместимы с состоянием, существующим на момент приостановки ожидаемой задачи. Фактически контекст “возвращает вас туда, где вы были”.

Компилятор генерирует код, применяющий `SynchronizationContext` для возвращения в желаемое состояние. Перед началом асинхронного метода компилятор кеширует текущий объект `SynchronizationContext`, используя статическое свойство `Current`. При возобновлении ожидаемой задачи компилятор отправляет оставшийся код в виде делегата тому же самому объекту `SynchronizationContext`. Объект `SynchronizationContext` планирует работу с применением средств, подходящих для среды.

В приложении с графическим пользовательским интерфейсом объект `SynchronizationContext` использует для планирования работы класс `Dispatcher` (см. совет 37). В контексте веб-приложения `SynchronizationContext` применяет пул потоков и метод `QueueUserWorkItem()` (см. совет 35). В консольном приложении, где `SynchronizationContext` отсутствует, работа продолжается в текущем потоке. Обратите внимание, что некоторые контексты имеют множество потоков, в то время как другие располагают единственным потоком и планируют работу кооперативно.

Если ожидаемая асинхронная задача отказала, тогда исключение, которое вызвало отказ, генерируется в коде, отправляемом объекту `SynchronizationContext`. Исключение генерируется, когда выполняется продолжение. Как следствие, для задач, которые не ожидаются, исключения в случае их отказа наблюдаться не будут. Их продолжения не планируются, а исключения перехватываются, но никогда повторно не генерируются в `SynchronizationContext`. По указанной причине всегда важно ожидать любые задачи, которые вы запускаете: это лучший способ наблюдать за исключениями, которые генерируются при выполнении асинхронной работы.

Та же самая стратегия дополнительно расширяется, когда методы имеют множество выражений `await`. Каждое выражение `await` может привести к тому, что асинхронный метод возвратится в вызывающий код с задачей, которая все еще не закончена. Внутреннее состояние обновляется, так что когда процедура повторяется снова, выполнение начинается в корректной точке. Как и при наличии лишь одного выражения `await` контекст синхронизации устанавливает способ планирования оставшейся работы: либо в единственном потоке внутри контекста, либо в другом потоке.

Компилятор записывает код того же вида, который бы вы писали для регистрации уведомлений о том, что асинхронная работа завершена. Он делает это в стандартной манере, которая облегчает чтение кода, будто бы он является синхронным.

Описанная до сих пор линия поведения предполагает, что асинхронная работа завершается успешно. Иногда генерируются исключения. Асинхронные методы обязаны также справляться с такими условиями. Такое требование усложняет поток управления, потому что асинхронный метод может возвратить управление вызывающему коду до завершения всей своей работы. Любые исключения должны каким-то образом внедряться в стек вызовов. Внутри асинхронного метода компилятор генерирует блок `try/catch`, перехватывающий все исключения. Исключения сохраняются в объекте `AggregateException`, который является членом объекта `Task`, и объект `Task` устанавливается в состояние ошибки. Когда заканчивается ожидание объекта `Task` в состоянии ошибки, выражение `await` генерирует первое исключение из объекта `AggregateException`.

В наиболее распространенном случае имеется только одно исключение, которое сгенерировалось в контексте вызывающего кода. Если есть несколько исключений, тогда вызывающий код обязан распаковать объект `AggregateException` и проанализировать каждое из них (см. совет 34).

Такой асинхронный механизм можно переопределять с использованием известных API-интерфейсов `Task`. Если вы на самом деле должны ожидать завершения объекта `Task`, то можете вызвать метод `Task.Wait()` или проверить свойство `Task<T>.Result`. Оба действия будут блокироваться до тех пор, пока вся асинхронная работа не завершится, что может быть полезно для метода `Main()` в консольном приложении. В совете 35 показано, как эти API-интерфейсы могут стать причиной взаимоблокировок и почему их следует избегать.

Когда вы создаете асинхронные методы с применением ключевых слов `async` и `await`, компилятор совершенно не прибегает к какой-либо магии. Взамен он вы-

полняет много работы по генерации большого объема кода для обработки продолжений, сообщения об ошибках и возобновления выполнения методов. Преимущество всех этих манипуляций компилятора заключается в том, что выполнение кажется приостановленным, пока асинхронная работа не завершена. Выполнение возобновляется, когда асинхронная работа готова. Такая пауза может перемещаться вверх по стеку вызовов настолько далеко, насколько это необходимо, при условии организации ожидания объектов `Task`. Механизм работает хорошо, если только вы его не переопределите.

Совет 28. Никогда не создавайте методы `async void`

В названии настоящего совета сделано строгое утверждение, и есть лишь небольшое число исключений из приведенной в нем рекомендации (как вы увидите в данном совете). Однако данная рекомендация сформулирована настолько убедительно по причине своей важности. Когда вы пишете метод `async void`, то тем самым нарушаете протокол, который дает возможность перехватывать исключения, сгенерированные асинхронными методами, в методах, запустивших асинхронную работу. Асинхронные методы сообщают об исключениях через объект `Task`. В результате генерации исключения объект `Task` входит в состояние ошибки. В случае ожидания задачи с ошибкой выражение `await` генерирует исключение. При ожидании задачи, которая инициирует ошибку позже, исключение генерируется, когда метод запланирован для возобновления работы.

В противоположность этому ожидание методов `async void` невозможно. Код, который вызывает метод `async void`, не в состоянии ни перехватывать, ни распространять исключение, сгенерированное асинхронным методом. Не пишите методы `async void`, поскольку ошибки будут скрываться от вызывающего кода.

Код в методе `async void` может сгенерировать исключение, с которым что-то должно произойти. Создаваемый для метода `async void` код генерирует любые исключения прямо на объекте `SynchronizationContext` (см. совет 27), который был активным, когда запустился метод `async void`. Это значительно затрудняет обработку таких исключений разработчиками, использующими вашу библиотеку. Вам придется применять обработчик события `AppDomain.UnhandledException` или похожий обработчик, перехватывающий все события. Обратите внимание, что обработчик для `AppDomain.UnhandledException` не позволяет восстанавливаться после исключения. Вы можете зарегистрировать ошибку в журнале и, скорее всего, сохранить данные, но не предотвратить прекращение работы приложения из-за неперехваченного исключения.

Взгляните на следующий метод:

```
private static async void FireAndForget()
{
  var task = DoAsyncThings();
  await task;
  var task2 = ContinueWork();
  await task2;
}
```

Если вы хотите регистрировать ошибки перед вызовом `FireAndForget()`, тогда должны настроить обработчик неперехваченных исключений. В данном примере информация об исключении выводится на консоль с использованием цвета `Cyan`:

```
AppDomain.CurrentDomain.UnhandledException += (sender, e) =>
{
  Console.ForegroundColor = ConsoleColor.Cyan;
  Console.WriteLine(e.ExceptionObject.ToString());
};
```

Установка в свойстве `ForegroundColor` консоли первоначального цвета не имеет особого смысла, т.к. приложение все равно прекратит работу.

Принуждение разработчиков к применению механизма обработки ошибок, который полностью отличается от того, который они используют во всем остальном коде, говорит о неудачно спроектированном API-интерфейсе. Конечно, многие разработчики не будут делать такую дополнительную работу. Еще хуже не предложить разработчикам ни одного способа восстановления после любых ошибок. Если разработчики не проделают дополнительную работу, то не будет сообщено об исключениях, которые могли сгенерироваться в методах `async void`. Исполняющая среда по-прежнему будет прерывать поток, выполняющийся в контексте синхронизации, но разработчики, применяющие ваш код, не получат уведомлений, не запустятся перехватывающие обработчики и не произойдет регистрация каких-либо исключений. По существу поток просто тихо исчезнет.

Помимо поведения исключений методы `async void` привносят и другие проблемы. Во многих асинхронных методах вы захотите запустить асинхронную работу, организовать ее ожидание и затем выполнять дополнительную работу после того, как ожидаемая задача завершится. В таких случаях создавать асинхронные задачи легко. Тем не менее, как отмечалось ранее, методы `async void` не поддаются ожиданию. Следовательно, разработчики, использующие ваши асинхронные методы, не смогут без труда установить, когда какой-то метод `async void` закончил всю свою работу. Это означает, что простая композиция больше невозможна. По сути, метод `async void` функционирует в стиле “запустил и забыл”: после запуска асинхронной работы разработчики не могут легко узнать, когда она завершится.

Те же самые проблемы усложняют процесс тестирования асинхронных методов. Автоматизированные тесты не могут выяснить, когда завершился метод `async void`. В итоге не удастся написать автоматизированный тест для проверки любых воздействий, оказываемых выполняющимся методом `async void`, вплоть до его завершения. Рассмотрим процесс написания автоматизированного модульного теста для показанного ниже метода:

```
public async void SetSessionState()
{
  var config = await ReadConfigFromNetwork();
  this.CurrentUser = config.User;
}
```

При написании теста вы можете обдумывать код следующего вида:

```
var t = new SessionManager();
t.SetSessionState();
// Немного подождать
await Task.Delay(1000);
Assert.Equal(t.User, "Пользователь TestLibrary");
```

Здесь задействована неудачная практика, которая на самом деле может даже не всегда работать. Ключом является вызов `Task.Delay()`. Вы не можете безопасно реализовать данный тест, потому что не знаете, когда асинхронная работа заканчивается. Может быть, одной секунды достаточно, а возможно и нет. Хуже того, одной секунды может оказаться достаточно в большинстве случаев, но в редких ситуациях — нет. В сценарии такого рода ваши тесты не пройдут, и они будут предоставлять ложный отклик.

К настоящему моменту должно быть ясно, что методы `async void` нежелательны. Всякий раз, когда возможно, вы обязаны создавать асинхронные методы, которые возвращают объекты `Task` или другие объекты, допускающие ожидание (см. совет 34). И все же методы `async void` разрешены, т.к. без них не удалось бы создавать асинхронные обработчики событий.

Протокол для обработчиков событий, представляющий их как методы, которые возвращают `void`, был установлен до появления в языке C# поддержки ключевых слов `async` и `await`. Несмотря на изменения, вам все равно понадобятся методы `async void` для присоединения асинхронных обработчиков событий к событиям, которые были определены в более ранних версиях. Вдобавок автор библиотеки может не знать, требует ли обработчик событий асинхронного доступа. С учетом всех перечисленных аспектов в языке C# поддерживаются асинхронные методы, возвращающие `void`. К тому же код, вызывающий обработчики событий, обычно не является пользовательским кодом. Если вызывающему коду не известно, что делать с возвращенным объектом `Task`, тогда зачем требовать его возвращения?

Хотя в названии текущего совета указано, что вы никогда не должны писать методы `async void`, в один прекрасный день вы наверняка обнаружите, что вам нужно написать обработчик событий `async void`. В таком случае вы обязаны реализовать асинхронный обработчик событий максимально безопасно.

Начните с осознания того, что методы `async void` не допускают ожидания. Код, который сгенерировал событие, не будет знать, когда ваш обработчик событий завершит выполнение. Обработчики событий, как правило, не возвращают данные в вызывающий код, поэтому вызывающий код может генерировать события в стиле “запустить и забыть”.

Безопасная обработка потенциальных исключений требует бо́льших усилий. Если внутри вашего метода `async void` сгенерируются любые исключения, то контекст синхронизации будет уничтожен. Вы должны написать свой обработчик событий `async void` так, чтобы никакие исключения в нем не генерировались. Это может идти вразрез с другими рекомендациями, но представляет собой идиому, когда вы обычно хотите перехватывать все исключения. Вот как выглядит типичный обработчик событий `async void`:

```
private async void OnCommand(object sender, RoutedEventArgs e)
{
  var viewModel = (DataContext as SampleViewModel);
  try
  {
    await viewModel.Update();
  }
  catch (Exception ex)
  {
    viewModel.Messages.Add(ex.ToString());
  }
}
```

В коде предполагается, что вы чувствуете себя в безопасности, просто регистрируя любые исключения и продолжая нормальное выполнение. На самом деле поведение такого вида может быть безопасным во многих сценариях. Если это справедливо для вашего сценария, тогда все готово.

Но что, если какие-то исключения, которые могут сгенерироваться в обработчике событий, связаны с катастрофическими условиями, не поддающимися обработке? Может быть, они способны вызвать серьезную порчу данных. В таком случае вы пожелаете немедленно прекратить работу программы, а не блаженно продолжать дальше разрушать данные. Чтобы добиться подобного результата, вы захотите сгенерировать исключение и заставить систему прервать поток в текущем контексте синхронизации.

В рамках этого процесса вам наверняка понадобится зарегистрировать в журнале необходимые сведения и сгенерировать исключение в методе `async void`. В предыдущий код необходимо внести небольшое изменение:

```
private async void OnCommand(object sender, RoutedEventArgs e)
{
  var viewModel = (DataContext as SampleViewModel);
  try
  {
    await viewModel.Update();
  }
  catch (Exception ex) when (logMessage(viewModel, ex))
  {
  }
}
private bool logMessage(SampleViewModel viewModel, Exception ex)
{
  viewModel.Messages.Add(ex.ToString());
  return false;
}
```

Этот метод фиксирует в журнале каждое исключение за счет применения фильтра исключений (см. совет 50 в книге *Эффективное программирование на C#, 3-е издание*) для регистрации сведений об исключении. Затем он повторно генерирует исключение, чтобы заставить контекст синхронизации остановить выполнение, возможно остановив также и программу.

Оба метода можно обобщить с использованием аргумента `Func` для представления асинхронной работы, выполняющейся в каждом методе. Далее вы можете многократно применять общие элементы из этих двух идиом.

```
public static class Utilities
{
  public static async void FireAndForget(this Task,
    Action<Exception> onErrors)
  {
    try
    {
      await task;
    }
    catch (Exception ex)
    {
      onErrors(ex);
    }
  }
  public static async void FireAndForget(this Task task,
    Func<Exception, bool> onError)
  {
    try
    {
      await task;
    }
    catch (Exception ex) when (onError(ex))
    {
    }
  }
}
```

В реальном мире лучшее решение не всегда может быть настолько простым, чтобы сводиться к перехвату всех исключений либо к их повторной генерации. Во многих реальных приложениях вы можете быть в состоянии восстановиться после одних исключений, но не после других. Например, вы можете располагать возможностью восстановления после `FileNotFoundException`, но не после каких-то других исключений. Такое поведение легко сделать более общим и многократно используемым, заменив специфический тип исключения обобщенным типом:

```
public static async void FireAndForget<TException>
  (this Task task,
   Action<TException> recovery,
   Func<Exception, bool> onError)
   where TException : Exception
{
  try
  {
    await task;
  }
  // Полагается на метод регистрации onError(),
  // всегда возвращающий false:
```

```
    catch (Exception ex) when (onError(ex))
    {
    }
    catch (TException ex2)
    {
      recovery(ex2);
    }
  }
```

При желании вы можете расширить тот же самый подход на большее число типов исключений.

Описанные приемы помогают сделать методы `async void` чуть более надежными в терминах восстановления после ошибок. Они не содействуют повышению пригодности к тестированию и объединению. Фактически для решения этих проблем нет хороших способов. Именно потому вы должны ограничивать применение методов `async void` местами, где они должны быть написаны: в обработчиках событий. Никогда не пишите методы `async void` в других ситуациях.

Совет 29. Избегайте объединения синхронных и асинхронных методов

Объявление метода с модификатором `async` сигнализирует о том, что данный метод может возвращать управление до завершения всей своей работы. Возвращаемый объект представляет состояние этой работы: завершена, содержит ошибки или все еще не закончена. Использование асинхронного метода дополнительно указывает, что любая незаконченная работа способна занять достаточно много времени, чтобы вызывающему коду было рекомендовано ожидать результата, пока выполняется другая полезная работа.

Объявление синхронного метода заявляет, что когда он завершится, все его постусловия будут удовлетворены. Безотносительно ко времени, которое необходимо методу для выполнения, он делает всю свою работу с применением тех же самых ресурсов, что и вызывающий код. Вызывающий код блокируется до тех пор, пока метод не завершится.

Смешивание этих ясных операторов приводит к плохому проектному решению API-интерфейса и может порождать ошибки, в том числе взаимоблокировку. Возможность таких исходов позволяет сформулировать два важных правила. Во-первых, не создавайте синхронные методы, которые блокируются в ожидании завершения асинхронной работы. Во-вторых, избегайте использования асинхронных методов для выполнения длительной работы с интенсивными вычислениями.

Исследуем первое правило более детально. Есть три причины, по которым создание синхронного кода вдобавок к асинхронному коду вызывает проблемы: разная семантика исключений, возможные взаимоблокировки и потребление ресурсов.

Асинхронные задачи могут приводить к возникновению множества исключений, поэтому класс `Task` содержит список исключений, которые были сгенерированы. Если в данном списке присутствуют какие-то исключения, тогда при ожидании задачи генерируется первое исключение из списка. Однако когда вы вызываете `Task.Wait()` или обращаетесь к `Task.Result`, для задачи в состоянии ошибки генерируется содержащееся исключение `AggregateException`. Затем вы должны

перехватить AggregateException и развернуть исключение, которое было сгенерировано. Сравните следующие две конструкции try/catch:

```
public static async Task<int> ComputeUsageAsync()
{
  try
  {
    var operand = await GetLeftOperandForIndex(19);
    var operand2 = await GetRightOperandForIndex(23);
    return operand + operand2;
  }
  catch (KeyNotFoundException e)
  {
    return 0;
  }
}

public static int ComputeUsage()
{
  try
  {
    var operand = GetLeftOperandForIndex(19).Result;
    var operand2 = GetRightOperandForIndex(23).Result;
    return operand + operand2;
  }
  catch (AggregateException e)
  when (e.InnerExceptions.FirstOrDefault().GetType()
       == typeof(KeyNotFoundException))
  {
    return 0;
  }
}
```

Обратите внимание на несходство в семантике обработки исключений. Версия, в которой ожидается задача, гораздо более читабельна, чем версия, где используется блокирующий вызов. Версия с ожиданием перехватывает исключения специфического типа, в то время как версия с блокированием перехватывает агрегированное исключение и должна применять фильтр исключений, чтобы гарантировать перехват исключения, только когда первое исключение, содержащееся в AggregateException, соответствует искомым типам. Идиомы, необходимые для блокирующих API-интерфейсов, оказываются более запутанными и сложными для понимания другими разработчиками.

А теперь рассмотрим второе правило — в частности, каким образом создание синхронных методов поверх асинхронного кода способно привести к взаимоблокировкам. Взгляните на приведенный ниже код:

```
private static async Task SimulatedWorkAsync()
{
  await Task.Delay(1000);
}

// Этот метод может стать причиной взаимоблокировки в контексте приложения
// ASP.NET или приложения с графическим пользовательским интерфейсом.
```

```
public static void SyncOverAsyncDeadlock()
{
  // Запустить задачу.
  var delayTask = SimulatedWorkAsync();

  // Синхронно ожидать окончания задержки.
  delayTask.Wait();
}
```

Вызов метода `SyncOverAsyncDeadlock()` работает корректно в консольном приложении, но приведет к взаимоблокировке в контексте приложения с графическим пользовательским интерфейсом или веб-приложения. Причина в том, что эти отличающиеся типы приложений задействуют разные виды контекстов синхронизации (см. совет 31). Контекст синхронизации для консольного приложения работает с множеством потоков из пула потоков, а контексты синхронизации для приложений с графическим пользовательским интерфейсом и ASP.NET имеют дело с единственным потоком. Ожидаемая задача, запущенная методом `SimulatedWorkAsync()`, не может продолжаться, поскольку один доступный поток заблокирован в ожидании завершения этой задачи. В идеале ваши API-интерфейсы должны быть пригодными в как можно большем числе типов приложений. Создание синхронного кода вдобавок к асинхронным API-интерфейсам делает такую цель недействительной. Вместо ожидания в синхронной манере завершения асинхронной работы выполняйте другую работу, пока ожидаете окончания задачи.

Обратите внимание, что в предыдущем примере для передачи управления и эмуляции длительно выполняющейся задачи использовался метод `Task.Delay()`, а не `Thread.Sleep()`. Указанный подход предпочтительнее, т.к. `Thread.Sleep()` заставляет приложение расходовать ресурсы данного потока в течение всего времени его простоя. Вы создали этот поток, потому займите его полезной работой. Метод `Task.Delay()` является асинхронным и позволит вызывающему коду составлять асинхронную работу, в то время как ваша задача эмулирует длительно выполняющиеся операции. Такое поведение может быть удобно в модульных тестах для обеспечения асинхронного завершения ваших задач (см. совет 27).

Существует одно общее исключение из правила о том, что вы не должны создавать синхронные методы поверх асинхронных: метод `Main()` в консольном приложении. Если бы метод `Main()` был асинхронным, тогда он мог бы возвратить управление до завершения всей работы, прекращая программу. Таким образом, этот метод является тем местом, где синхронным методам должно отдаваться предпочтение перед асинхронными. В других отношениях он почти асинхронный. Было внесено предложение разрешить методу `Main()` быть асинхронным и справиться с данной ситуацией. Кроме того, доступен пакет NuGet под названием `AsyncEx`, который поддерживает асинхронный главный контекст.

Возможно, в своих библиотеках вы имеете синхронные API-интерфейсы, которые можно обновить, чтобы сделать их асинхронными. Устранение синхронного API-интерфейса было бы серьезным изменением. Я просто не рекомендую преобразовывать API-интерфейс в асинхронный и вместо блокирующего синхронного метода вызывать асинхронный метод. Но это вовсе не означает, что вам остается поддерживать только синхронный API-интерфейс, никак не развивая библиотеку

в дальнейшем. Вы можете создать асинхронный API-интерфейс, который зеркально отражает синхронный код и поддерживать обе версии. Пользователи, готовые к асинхронной работе, будут применять асинхронный метод, другие же смогут продолжить пользоваться синхронным методом. В будущем вы можете объявить синхронный метод не рекомендуемым. Фактически некоторые разработчики уже начинают выдвигать предположения относительно библиотек, поддерживающих синхронную и асинхронную версии того же самого метода: они считают синхронный метод унаследованным, тогда как асинхронный — рекомендуемым.

Такое наблюдение подсказывает, почему асинхронный метод, который является оболочкой для синхронной операции с интенсивными вычислениями, также будет неудачной идеей. Если разработчики полагают, что в случае доступности синхронной и асинхронной версий для того же самого метода его асинхронная версия предпочтительнее, то они будут естественным образом склоняться к асинхронной версии. Когда асинхронный метод — просто оболочка, вы их запутаете. Рассмотрим следующие два метода:

```
public double ComputeValue()
{
  // Выполнить много работы.
  double finalAnswer = 0;
  for (int i = 0; i < 10_000_000; i++)
    finalAnswer += InterimCalculation(i);
  return finalAnswer;
}
public Task<double> ComputeValueAsync()
{
  return Task.Run(() => ComputeValue());
}
```

Синхронная версия позволяет вызывающему коду решать, нужно ли выполнять такую работу с интенсивными вычислениями синхронно или асинхронно в другом потоке. Приведенный выше асинхронный метод не оставляет выбора. Вызывающий код вынужден организовать новый поток либо извлечь его из пула и затем запустить операцию в этом новом потоке. Если работа с интенсивными вычислениями является частью более крупной операции, тогда она уже может выполняться в отдельном потоке. В качестве альтернативы метод может быть вызван из консольного приложения и в этом случае выполняться в фоновом потоке, просто связывая больше ресурсов.

Это не означает, что работу с интенсивными вычислениями нельзя выполнять в отдельных потоках. Скорее дело в том, что работа с интенсивными вычислениями должна быть как можно более крупной. Код, запускающий фоновую задачу, обязан находиться в точке входа приложения. Подумайте о методе `Main()` в консольном приложении, обработчиках ответов в веб-приложении или обработчиках действий пользователей в приложении с графическим пользовательским интерфейсом: они являются точками в приложении, где работа с интенсивными вычислениями должна направляться другим потокам. Создание асинхронных методов для синхронной работы с интенсивными вычислениями в других местах просто вводит в заблуждение сторонних разработчиков.

Эффекты выгрузки работы с применением асинхронных методов начинают пронизывать ваше приложение по мере того, как вы создаете больше асинхронных методов вдобавок к прочим асинхронным API-интерфейсам. Это в точности то, что должно быть. Вы продолжите добавлять еще больше асинхронных методов в вертикальных срезах выше в стеке вызовов. Если вы преобразуете или расширяете существующую библиотеку, тогда обдумайте поддержку асинхронной и синхронной версий вашего API-интерфейса бок о бок. Но избирайте такой курс, только когда работа является асинхронной, и вы выгружаете работу в другой ресурс. Если вы добавляете асинхронные версии методов с интенсивными вычислениями, то просто запутываете своих пользователей.

Совет 30. Используйте асинхронные методы, чтобы избежать размещения потоков и переключения контекста

Слишком легко начать считать, что все асинхронные задачи представляют работу, которая выполняется в других потоках. В конце концов, это одно из применений асинхронной работы. На самом деле во многих случаях асинхронная работа вовсе не запускает новый поток. Файловый ввод-вывод является асинхронным, но использует порты завершения ввода-вывода, а не потоки. Веб-запросы также асинхронны, но применяют вместо потоков сетевые прерывания. В таких ситуациях использование асинхронных задач освобождает поток для выполнения полезной работы.

Выгружая работу в другой поток, вы освобождаете один поток ценой создания и запуска другого. Такое проектное решение разумно, только если освобождаемый поток входит в число дефицитных ресурсов. Поток пользовательского интерфейса в приложении с графическим пользовательским интерфейсом относится к дефицитным ресурсам: только один поток взаимодействует с любыми визуальными элементами, которые видит пользователь. Тем не менее, потоки из пула не будут ни уникальными, ни дефицитными (хотя их количество ограничено), т.е. один поток оказывается таким же, как любой другой поток из того же самого пула. По этой причине в приложениях, отличных от приложений с графическим пользовательским интерфейсом, вы должны избегать асинхронных задач с интенсивными вычислениями.

Для дальнейшего изучения вопроса давайте сначала займемся приложениями с графическим пользовательским интерфейсом. Когда пользователь инициирует действие в пользовательском интерфейсе, он ожидает, что пользовательский интерфейс останется отзывчивым. Так не произойдет, если поток пользовательского интерфейса тратит на выполнение последнего действия секунду и более. Решение проблемы предусматривает выгрузку данной работы в другой ресурс, чтобы пользовательский интерфейс мог оставаться отзывчивым для других действий со стороны пользователя. Как было показано в совете 29, обработчики событий пользовательского интерфейса являются теми местами, где создание асинхронного кода поверх синхронного имеет смысл.

Теперь перейдем к консольным приложениям. Консольные приложения, которые выполняют только одну длительную задачу с интенсивными вычислениями, не получат выигрыша от ее выполнения в отдельном потоке. Главный поток будет син-

хронно ожидать, а рабочий поток будет занятым. В таком случае два потока связаны для выполнения работы одного потока.

Однако если консольное приложение выполняет *несколько* длительных операций с интенсивными вычислениями, тогда их запуск в разных потоках может иметь смысл. В совете 35 обсуждаются варианты выполнения работы с интенсивными вычислениями во множестве потоков.

Это подводит нас к серверным приложениям ASP.NET, которые, похоже, вызывают немалую путаницу у разработчиков. В идеале вы хотите удерживать потоки свободными, чтобы ваше приложение могло обрабатывать большее число входящих запросов. Результатом оказывается проектное решение, в котором работа с интенсивными вычислениями выгружается в другие потоки внутри обработчиков ASP.NET:

```
public async Task<IActionResult> Compose()
{
  var model = await LongRunningCPUTask();
  return View(model);
}
```

Давайте подробно рассмотрим, что происходит в данной ситуации. Запуская еще один поток для задачи, вы выделяете второй поток из пула потоков. Первый поток сидит без дела и может быть применен для выполнения добавочной работы, но это требует бо́льших накладных расходов. Чтобы "вернуть вас туда, где вы находились", контекст синхронизации должен отслеживать все состояние для входящего запроса и восстанавливать его, когда ожидаемая работа с интенсивными вычислениями завершится. Только затем обработчик может ответить клиенту.

При таком подходе вы не освободили ресурсы, но добавили два переключения контекста во время обработки запроса.

Если в ответ на входящие запросы необходимо выполнять длительную работу с интенсивными вычислениями, тогда вам понадобится выгрузить эту работу в другой процесс или на другую машину, чтобы освободить ресурс потока и повысить способность веб-приложения к обслуживанию запросов. Например, вы можете располагать вторым заданием, которое получает работы с интенсивными вычислениями и выполняет их по очереди. В качестве альтернативы для работ с интенсивными вычислениями вы можете выделить вторую машину.

Какой вариант окажется быстрее зависит от характеристик приложения: объема трафика, времени, необходимого для выполнения работы с интенсивными вычислениями, и сетевой задержки. Чтобы принять обоснованное решение, вы обязаны измерить упомянутые показатели. Одна из конфигураций, которую вы должны оценить, предусматривает выполнение всей работы в веб-приложении синхронным образом. Скорее всего, такой подход окажется быстрее, чем выгрузка работы в другой поток из того же самого пула и процесса.

Асинхронная работа выглядит сродни магии: выгрузив работу в другой ресурс, вы возобновляете свою обработку после того, как работа завершится. Для обеспечения эффективности этого подхода вы должны удостовериться, что при выгрузке работы действительно освобождаете ресурсы, а не просто переключаете контекст между похожими ресурсами.

Совет 31. Избегайте ненужной маршализации контекста

Мы называем код, который может выполняться в любом контексте синхронизации, “контекстно-свободным кодом”. В противоположность ему код, который обязан запускаться в специфическом контексте, представляет собой “контекстно-зависимый код”. Большинство кода, который вы пишете, является контекстно-свободным. Контекстно-зависимый код включает код внутри приложения с графическим пользовательским интерфейсом, взаимодействующий с элементами управления, и код внутри веб-приложения, который взаимодействует с `HTTPContext` или связанными классами. Когда контекстно-зависимый код выполняется после завершения ожидаемой задачи, он должен находиться в надлежащем контексте (см. совет 27). Тем не менее, весь остальной код может выполняться в стандартном контексте.

При столь малом количестве мест, где код зависит от контекста, вас может интересовать, почему стандартное поведение предусматривает выполнение продолжений в захваченном контексте. На самом деле последствия переключения контекста без надобности гораздо менее обременительны, чем последствия не переключения контекста, когда оно необходимо. Если вы выполните контекстно-свободный код в захваченном контексте, то вряд ли что-то будет развиваться радикально плохо. С другой стороны, если вы запустите контекстно-зависимый код в ошибочном контексте, тогда ваше приложение, скорее всего, потерпит отказ. По указанной причине стандартное поведение предполагает выполнение продолжения в захваченном контексте, нужно это или нет.

Хотя возобновление выполнения в захваченном контексте не всегда приводит к возникновению крупных проблем, оно все равно вызывает проблемы, которые с течением времени могут усугубиться. Запуская продолжения в захваченном контексте, вы никогда не воспользуетесь возможностью выгрузки некоторых продолжений в другие потоки. В приложениях с графическим пользовательским интерфейсом это может сделать пользовательский интерфейс неотзывчивым. В веб-приложениях это может ограничить число запросов в минуту, которые приложение способно обработать. Показатели производительности со временем будет ухудшаться. В приложениях с графическим пользовательским интерфейсом увеличивается вероятность взаимоблокировки (см. совет 39). В веб-приложениях не полностью утилизируется пул потоков.

Способ обхода таких нежелательных исходов предусматривает применение метода `ConfigureAwait()` для указания о том, что продолжения не нуждаются в запуске внутри захваченного контекста. В коде библиотеки, где продолжение является контекстно-свободным кодом, вы могли бы использовать его следующим образом:

```
public static async Task<XElement> ReadPacket(string Url)
{
  var result = await DownloadAsync(Url)
    .ConfigureAwait(continueOnCapturedContext: false);
  return XElement.Parse(result);
}
```

В простых случаях это легко. Вы добавляете вызов `ConfigureAwait()` и продолжение будет запускаться в стандартном контексте. Взгляните на представленный ниже метод:

```
public static async Task<Config> ReadConfig(string Url)
{
  var result = await DownloadAsync(Url)
    .ConfigureAwait(continueOnCapturedContext: false);
  var items = XElement.Parse(result);
  var userConfig = from node in items.Descendants()
                   where node.Name == "Config"
                   select node.Value;
  var configUrl = userConfig.SingleOrDefault();
  if (configUrl != null)
  {
    result = await DownloadAsync(configUrl)
      .ConfigureAwait(continueOnCapturedContext: false);
    var config = await ParseConfig(result)
      .ConfigureAwait(continueOnCapturedContext: false);
    return config;
  }
  else
    return new Config();
}
```

Вы можете посчитать, что после достижения первого выражения `await` продолжение запустится в стандартном контексте, а потому `ConfigureAwait()` не нуждается в каких-либо последующих асинхронных вызовах. Такое предположение может оказаться ошибочным. Что, если первая задача завершается синхронно? Вспомните из совета 27, что это будет означать синхронное продолжение работы в захваченном контексте. Затем выполнение доберется до очередного выражения `await`, по-прежнему находясь внутри первоначального контекста. Последующие вызовы не будут продолжаться в стандартном контексте, так что вся работа продолжится в захваченном контексте.

По этой причине всякий раз, когда вызывается асинхронный метод, возвращающий `Task`, и продолжение является контекстно-свободным кодом, вы должны применять `ConfigureAwait(false)` для продолжения в стандартном контексте. Ваша цель — изолировать контекстно-зависимый код кодом, который обязан манипулировать пользовательским интерфейсом. Чтобы выяснить, как это работает, рассмотрим следующий метод:

```
private async void OnCommand(object sender, RoutedEventArgs e)
{
  var viewModel = (DataContext as SampleViewModel);
  try
  {
    var userInput = viewModel.webSite;
    var result = await DownloadAsync(userInput);
    var items = XElement.Parse(result);
    var userConfig = from node in items.Descendants()
                     where node.Name == "Config"
                     select node.Value;
```

```
    var configUrl = userConfig.SingleOrDefault();
    if (configUrl != null)
    {
      result = await DownloadAsync(configUrl);
      var config = await ParseConfig(result);
      await viewModel.Update(config);
    }
    else
      await viewModel.Update(new Config());
  }
  catch (Exception ex) when (logMessage(viewModel, ex))
  {
  }
}
```

Показанный метод структурирован так, что изолировать контекстно-зависимый код трудно. Внутри данного метода вызывается несколько асинхронных методов, большинство из которых контекстно-свободны. Однако код в конце метода обновляет элементы управления пользовательского интерфейса и зависит от контекста. Вы должны трактовать весь допускающий ожидание код как контекстно-свободный, если только он не обновляет элементы управления пользовательского интерфейса. Еще раз: от контекста зависит лишь код, который обновляет пользовательский интерфейс.

В приведенном виде пример метода обязан запускать все свои продолжения в захваченном контексте. После того как любое продолжение стартует в стандартном контексте, легкого пути назад не существует. Первый шаг в исправлении положения предполагает реструктуризацию кода, чтобы переместить весь контекстно-свободный код в новый метод. Когда реструктуризация проведена, вы можете добавить к каждому методу вызовы `ConfigureAwait(false)` для запуска асинхронных продолжений в стандартном контексте:

```
private async void OnCommand(object sender, RoutedEventArgs e)
{
  var viewModel = (DataContext as SampleViewModel);
  try
  {
    Config config = await ReadConfigAsync(viewModel);
    await viewModel.Update(config);
  }
  catch (Exception ex) when (logMessage(viewModel, ex))
  {
  }
}

private async Task<Config> ReadConfigAsync(SampleViewModel
  viewModel)
{
  var userInput = viewModel.webSite;
  var result = await DownloadAsync(userInput)
    .ConfigureAwait(continueOnCapturedContext: false);
  var items = XElement.Parse(result);
```

```
  var userConfig = from node in items.Descendants()
                   where node.Name == "Config"
                   select node.Value;
  var configUrl = userConfig.SingleOrDefault();
  var config = default(Config);
  if (configUrl != null)
  {
    result = await DownloadAsync(configUrl)
      .ConfigureAwait(continueOnCapturedContext: false);
    config = await ParseConfig(result)
      .ConfigureAwait(continueOnCapturedContext: false);
  }
  else
    config = new Config();
  return config;
}
```

Ситуация могла сложиться проще, если бы по умолчанию продолжение выполнялось в стандартном контексте. Тем не менее, такой подход означал бы, что когда что-то пойдет не так, приложение потерпит неудачу. Согласно текущей стратегии, если все продолжения используют захваченный контекст, то приложение будет работать, но менее эффективно. Ваши пользователи заслуживают большего. Структурируйте свой код для изоляции кода, который должен выполняться в захваченном контексте. Применяйте вызов `ConfigureAwait(false)`, чтобы продолжать в стандартном контексте всякий раз, когда это возможно.

Совет 32. Формируйте асинхронную работу с использованием объектов задач

Задачи являются абстракцией для работы, выгруженной в другой ресурс. Тип `Task` и связанные классы и структуры имеют развитые API-интерфейсы для манипулирования задачами и выгруженной работой. Задачи также представляют собой объекты, которыми можно манипулировать с применением их методов и свойств. Задачи можно агрегировать для образования более крупных задач. Задачи можно упорядочивать или запускать параллельно. Чтобы навязать упорядочение, вы используете выражения `await`: код, находящийся после выражения `await`, не будет выполнен до тех пор, пока не завершится ожидаемая задача. Вы можете указать, что задача должна запускаться только в ответ на завершение другой задачи. В целом богатый набор API-интерфейсов, применяемых задачами, дает возможность писать элегантные алгоритмы, которые имеют дело с такими объектами и представляемой ими работой. Чем больше вы узнаете о том, как использовать задачи в виде объектов, тем более элегантным будет ваш асинхронный код.

Давайте начнем с асинхронного метода, который запускает несколько задач и ожидает завершения каждой из них. Очень наивная реализация выглядела бы так:

```
public static async Task<IEnumerable<StockResult>>
  ReadStockTicker(IEnumerable<string> symbols)
{
  var results = new List<StockResult>();
```

```
  foreach (var symbol in symbols)
  {
    var result = await ReadSymbol(symbol);
    results.Add(result);
  }
  return results;
}
```

Поскольку задачи независимые, нет никаких причин ожидать завершения каждой задачи перед запуском следующей. Можно было бы внести в код одно изменение: запустить все задачи и затем ожидать завершения их всех, прежде чем выполнять продолжения:

```
public static async Task<IEnumerable<StockResult>>
  ReadStockTicker(IEnumerable<string> symbols)
{
  var resultTasks = new List<Task<StockResult>>();
  foreach (var symbol in symbols)
  {
    resultTasks.Add(ReadSymbol(symbol));
  }
  var results = await Task.WhenAll(resultTasks);
  return results.OrderBy(s => s.Price);
}
```

Такая реализация будет корректной, если для фактического продолжения требуются результаты всех задач. С применением метода `WhenAll()` создается новая задача, которая завершится, когда все наблюдаемые задачи окажутся завершенными. Результатом `Task.WhenAll()` является массив всех завершенных (или ошибочных) задач.

В других случаях вы можете запускать несколько задач, которые все генерируют тот же самый результат. В такой ситуации вы преследуете цель опробовать разные источники и продолжить работу с первой завершившейся задачей. Метод `Task.WhenAny()` создает новую задачу, которая завершается, как только завершится любая из ожидаемых задач.

Предположим, что вы хотите читать одиночный символ акций из множества онлайновых источников и возвращать первый готовый результат. Вы могли бы воспользоваться методом `WhenAny()` для выяснения, какая из запущенных задач завершается первой:

```
public static async Task<StockResult>
  ReadStockTicker(string symbol, IEnumerable<string> sources)
{
  var resultTasks = new List<Task<StockResult>>();
  foreach (var source in sources)
  {
    resultTasks.Add(ReadSymbol(symbol, source));
  }
  return await Task.WhenAny(resultTasks);
}
```

Иногда вам может понадобиться запускать продолжение по мере завершения каждой задачи. Ниже представлена наивная реализация:

```
public static async Task<IEnumerable<StockResult>>
  ReadStockTicker(IEnumerable<string> symbols)
{
  var resultTasks = new List<Task<StockResult>>();
  var results = new List<StockResult>();
  foreach (var symbol in symbols)
  {
    resultTasks.Add(ReadSymbol(symbol));
  }
  foreach(var task in resultTasks)
  {
    var result = await task;
    results.Add(result);
  }
  return results;
}
```

Нет никаких гарантий, что задачи будут завершаться в том же порядке, в каком они запускались. Такой алгоритм крайне неэффективен: множество задач могут ожидать обработки просто из-за того, что находятся в очереди после задачи, выполнение которой занимает более долгое время.

Вы могли бы попытаться улучшить положение с применением метода `Task.WhenAny()`:

```
public static async Task<IEnumerable<StockResult>>
  ReadStockTicker(IEnumerable<string> symbols)
{
  var resultTasks = new List<Task<StockResult>>();
  var results = new List<StockResult>();
  foreach (var symbol in symbols)
  {
    resultTasks.Add(ReadSymbol(symbol));
  }
  while (resultTasks.Any())
  {
    // При каждом прохождении цикла этот код создает
    // новую задачу, что может быть затратным.
    Task<StockResult> finishedTask = await
      Task.WhenAny(resultTasks);
    var result = await finishedTask;
    resultTasks.Remove(finishedTask);
    results.Add(result);
  }
  var first = await Task.WhenAny(resultTasks);
  return await first;
}
```

Как отмечено в комментарии, такая стратегия не обеспечивает хороший способ реализации желаемого поведения. Каждый раз, когда вызывается метод `Task.WhenAny()`, создается новая задача. С ростом количества задач, которыми

нужно управлять, данный алгоритм выполняет все больше и больше размещений, становясь все менее и менее эффективным.

В качестве альтернативы можно использовать класс `TaskCompletionSource`, делающий возможным возращение объекта `Task`, которым вы манипулируете для порождения результата в более позднее время. Фактически вы можете получать результат из любого метода асинхронно. Самым распространенным применением этой стратегии является предоставление средства передачи между исходным объектом (или объектами) `Task` и целевым объектом (или объектами) `Task`. Вы пишете код, который должен выполняться при завершении исходной задачи. Затем ваш код организует ожидание исходной задачи и обновляет целевую задачу с использованием класса `TaskCompletionSource`.

В показанном далее примере предполагается, что у вас есть массив исходных задач. Вы создадите массив целевых объектов `TaskCompletionSource`. По мере завершения каждой задачи вы будете обновлять одну из целевых задач с применением ее объекта `TaskCompletionSource`. Код показан ниже:

```
public static Task<T>[] OrderByCompletion<T>(
  this IEnumerable<Task<T>> tasks)
{
  // Копировать в объект List, потому что
  // он будет перечисляться много раз.
  var sourceTasks = tasks.ToList();

  // Разместить исходные и целевые задачи.
  // Каждой целевой задаче соответствует
  // завершенная исходная задача.
  var completionSources =
    new TaskCompletionSource<T>[sourceTasks.Count];
  var outputTasks = new Task<T>[completionSources.Length];
  for (int i = 0; i < completionSources.Length; i++)
  {
    completionSources[i] = new TaskCompletionSource<T>();
    outputTasks[i] = completionSources[i].Task;
  }

  // Магия, часть 1:
  // Каждая задача имеет продолжение, которое помещает
  // ее результат в следующий свободный элемент
  // массива completionSources.
  int nextTaskIndex = -1;
  Action<Task<T>> continuation = completed =>
  {
    var bucket = completionSources
      [Interlocked.Increment(ref nextTaskIndex)];
    if (completed.IsCompleted)
      bucket.TrySetResult(completed.Result);
    else if (completed.IsFaulted)
      bucket.TrySetException(completed.Exception);
  };

  // Магия, часть 2:
  // Для каждой исходной задачи настроить продолжение,
  // чтобы установить целевую задачу.
```

```
    // После завершения каждая задача использует
    // следующий элемент в списке.
    foreach (var inputTask in sourceTasks)
    {
      inputTask.ContinueWith(continuation,
     CancellationToken.None,
        TaskContinuationOptions.ExecuteSynchronously,
        TaskScheduler.Default);
    }
    return outputTasks;
}
```

В коде происходит довольно многое, поэтому мы рассмотрим его раздел за разделом. Сначала метод размещает массив объектов `TaskCompletionSource`. Затем он определяет код продолжения, который будет запускаться при завершении каждой исходной задачи. Этот код продолжения устанавливает следующий элемент в массиве целевых объектов `TaskCompletionSource`, подлежащих завершению. Он применяет метод `InterlockedIncrement()`, чтобы обновлять следующий свободный элемент в безопасной к потокам манере. Наконец, для каждого объекта `Task` устанавливается продолжение, чтобы выполнить данный код. В итоге метод возвращает последовательность задач из массива объектов `TaskCompletionSource`.

Теперь вызывающий код перечисляет список задач, который будет упорядочен по времени их завершения. Давайте разберем один типовой случай выполнения, при котором запускается 10 задач. Пусть задачи завершаются в следующем порядке: 3, 7, 2, 0, 4, 9, 1, 6, 5, 8. Когда завершится задача 3, запустится ее продолжение и поместит результат в элемент 0 целевого массива. Далее заканчивается задача 7 с помещением результата в элемент 1. Задача 2 помещает свой результат в элемент 2. Процесс продолжается до тех пор, пока не завершится задача 8, поместив свой результат в элемент 9 (рис. 3.1).

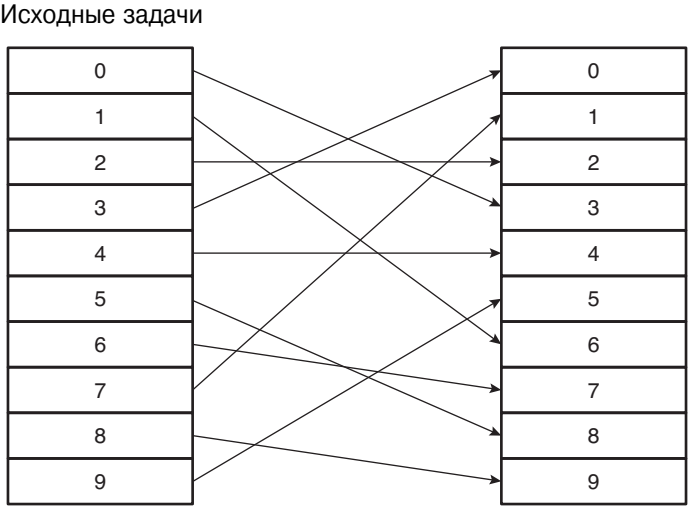

Рис. 3.1. *Упорядочение задач на основе времени их завершения*

Расширим код, чтобы он обрабатывал задачи, которые оказываются в состоянии ошибки. Необходимо внести в код продолжения единственное изменение:

```
// Магия, часть 1:
// Каждая задача имеет продолжение, которое помещает
// ее результат в следующий свободный элемент
// массива completionSources.
int nextTaskIndex = -1;
Action<Task<T>> continuation = completed =>
{
  var bucket = completionSources
    [Interlocked.Increment(ref nextTaskIndex)];
  if (completed.IsCompleted)
    bucket.TrySetResult(completed.Result);
  else if (completed.IsFaulted)
    bucket.TrySetException(completed.Exception);
};
```

Доступно несколько методов и API-интерфейсов, которые поддерживают программирование с использованием задач и позволяют выполнять действия, когда задачи завершаются или отказывают. Применение таких методов облегчает построение элегантных алгоритмов, обрабатывающих результаты асинхронного кода, когда они будут готовы. Эти расширения, находящиеся в библиотеке работы с задачами, указывают действия, которые предпринимаются при завершении задач. С другой стороны, легко читаемый код манипулирует задачами по мере их завершения крайне неэффективно.

Совет 33. Обдумайте реализацию протокола отмены задачи

Модель асинхронного программирования на основе задач включает стандартные API-интерфейсы для отмены и сообщения о продвижении. Они являются необязательными, но должны быть реализованы корректно, когда асинхронная работа способна сообщать о продвижении или быть отмененной.

Не каждая асинхронная работа может отменяться, поскольку лежащий в основе механизм не всегда поддерживает протокол отмены. В таких случаях ваш асинхронный API-интерфейс не должен поддерживать любые перегруженные версии, которые указывают на возможность отмены. Вы не хотите, чтобы в вызывающем коде проводились дополнительные работы по реализации протокола отмены, когда он не имеет никакого эффекта.

То же самое справедливо в отношении сообщения о продвижении. Модель программирования поддерживает сообщение о продвижении, но API-интерфейсы реализуют этот протокол, только когда они действительно сообщают о продвижении. Не реализуйте перегруженную версию метода для сообщения о продвижении, если вы не в состоянии точно указать, сколько асинхронной работы было сделано. Для примера возьмем веб-запрос. Вы не получите от сетевого стека промежуточные сведения о доставке запроса, обработке запроса или выполнении любого другого действия до поступления ответа. После получения ответа задача завершена. Никакой дополнительной ценности сообщение о продвижении не предлагает.

Сравните это с задачей, которая последовательно делает пять веб-запросов к разным службам для выполнения сложной операции. Предположим, что вы написали API-интерфейс для обработки платежной ведомости. Он может включать следующие шаги.

1. Обращение к первой веб-службе для извлечения списка сотрудников и отработанных ими часов.
2. Обращение ко второй веб-службе для подсчета налогов и сообщения их размеров.
3. Обращение к третьей веб-службе для генерации счетов на оплату и их отправки по электронной почте сотрудникам.
4. Обращение к четвертой веб-службе для банковского перевода заработной платы.
5. Закрытие платежного периода.

Разумно предположить, что каждый из перечисленных шагов представляет 20% работы. Вы можете реализовать перегруженную версию для сообщения о продвижении после завершения каждого из пяти шагов. Кроме того, вы можете реализовать API-интерфейс отмены. Операцию разрешено отменять, пока не начался четвертый шаг. Однако после того как оплата произведена, отмена не допускается.

Рассмотрим перегруженные версии, которые вы должны поддерживать в настоящем примере. Давайте обратимся к простейшей из них — обработка платежной ведомости без поддержки отмены и сообщения о продвижении:

```
public async Task RunPayroll(DateTime payrollPeriod)
{
  // Шаг 1: подсчет отработанных часов и размера оплаты
  var payrollData = await RetrieveEmployeePayrollDataFor(
    payrollPeriod);

  // Шаг 2: подсчет налогов и сообщение их размеров
  var taxReporting = new Dictionary<EmployeePayrollData,
    TaxWithholding>();
  foreach(var employee in payrollData)
  {
    var taxWithholding = await RetrieveTaxData(employee);
    taxReporting.Add(employee, taxWithholding);
  }

  // Шаг 3: генерация и отправка по электронной почте счетов на оплату
  var paystubs = new List<Task>();
  foreach(var payrollItem in taxReporting)
  {
    var payrollTask = GeneratePayrollDocument(
      payrollItem.Key, payrollItem.Value);
    var emailTask = payrollTask.ContinueWith(
      paystub => EmailPaystub(
        payrollItem.Key.Email, paystub.Result));
    paystubs.Add(emailTask);
  }
  await Task.WhenAll(paystubs);
```

```
    // Шаг 4: банковский перевод оплаты
    var depositTasks = new List<Task>();
    foreach(var payrollItem in taxReporting)
    {
      depositTasks.Add(MakeDeposit(payrollItem.Key,
        payrollItem.Value));
    }
    await Task.WhenAll(depositTasks);

    // Шаг 5: закрытие платежного периода
    await ClosePayrollPeriod(payrollPeriod);
  }
```

Далее добавим перегруженную версию, которая поддерживает сообщение о продвижении. Вот как она может выглядеть:

```
  public async Task RunPayroll2(DateTime payrollPeriod,
    IProgress<(int, string)> progress)
  {
    progress?.Report((0, "Начало обработки платежной ведомости"));
    // Шаг 1: подсчет отработанных часов и размера оплаты
    var payrollData = await RetrieveEmployeePayrollDataFor(payrollPeriod);

    progress?.Report((20, "Извлечение сведений о сотрудниках и отработанных
часах"));

    // Шаг 2: подсчет налогов и сообщение их размеров
    var taxReporting = new Dictionary<EmployeePayrollData,
      TaxWithholding>();
    foreach (var employee in payrollData)
    {
      var taxWithholding = await RetrieveTaxData(employee);
      taxReporting.Add(employee, taxWithholding);
    }
    progress?.Report((40, "Подсчет удержаний"));

    // Шаг 3: генерация и отправка по электронной почте счетов на оплату
    var paystubs = new List<Task>();
    foreach (var payrollItem in taxReporting)
    {
      var payrollTask = GeneratePayrollDocument(
        payrollItem.Key, payrollItem.Value);
      var emailTask = payrollTask.ContinueWith(
        paystub => EmailPaystub(payrollItem.Key.Email,
        paystub.Result));
      paystubs.Add(emailTask);
    }
    await Task.WhenAll(paystubs);
    progress?.Report((60, "Отправка по электронной почте счетов на оплату"));

    // Шаг 4: банковский перевод оплаты
    var depositTasks = new List<Task>();
    foreach (var payrollItem in taxReporting)
    {
      depositTasks.Add(MakeDeposit(payrollItem.Key,
        payrollItem.Value));
    }
```

```
  await Task.WhenAll(depositTasks);
  progress?.Report((80, "Банковский перевод оплаты"));

  // Шаг 5: закрытие платежного периода
  await ClosePayrollPeriod(payrollPeriod);
  progress?.Report((100, "Обработка платежной ведомости завершена"));
}
```

Вызывающий код использовал бы эту идиому следующим образом:

```
public class ProgressReporter :
  IProgress<(int percent, string message)>
{
  public void Report((int percent, string message) value)
  {
    WriteLine(
   $"{value.percent}% выполнено: {value.message}");
  }
}

await generator.RunPayroll(DateTime.Now, new ProgressReporter());
```

Добавив сообщение о продвижении, можно заняться обеспечением отмены. Ниже приведена реализация, которая поддерживает отмену, но не сообщение о продвижении:

```
public async Task RunPayroll(DateTime payrollPeriod,
  CancellationToken cancellationToken)
{
  // Шаг 1: подсчет отработанных часов и размера оплаты
  var payrollData = await RetrieveEmployeePayrollDataFor(
    payrollPeriod);
  cancellationToken.ThrowIfCancellationRequested();

  // Шаг 2: подсчет налогов и сообщение их размеров
  var taxReporting = new Dictionary<EmployeePayrollData,
    TaxWithholding>();
  foreach (var employee in payrollData)
  {
    var taxWithholding = await RetrieveTaxData(employee);
    taxReporting.Add(employee, taxWithholding);
  }
  cancellationToken.ThrowIfCancellationRequested();

  // Шаг 3: генерация и отправка по электронной почте счетов на оплату
  var paystubs = new List<Task>();
  foreach (var payrollItem in taxReporting)
  {
    var payrollTask = GeneratePayrollDocument(
      payrollItem.Key, payrollItem.Value);
    var emailTask = payrollTask.ContinueWith(
      paystub => EmailPaystub(payrollItem.Key.Email,
      paystub.Result));
    paystubs.Add(emailTask);
  }
  await Task.WhenAll(paystubs);
  cancellationToken.ThrowIfCancellationRequested();
```

```
    // Шаг 4: банковский перевод оплаты
    var depositTasks = new List<Task>();
    foreach (var payrollItem in taxReporting)
    {
      depositTasks.Add(MakeDeposit(payrollItem.Key,
        payrollItem.Value));
    }
    await Task.WhenAll(depositTasks);

    // Шаг 5: закрытие платежного периода
    await ClosePayrollPeriod(payrollPeriod);
  }
```

Обращаться к данному методу в вызывающем коде можно было бы так:

```
var cts = new CancellationTokenSource();
generator.RunPayroll(DateTime.Now, cts.Token);
// Отменить:
cts.Cancel();
```

Вызывающий код запрашивает отмену с применением объекта `CancellationTokenSource`. Подобно `TaskCompletionSource` из совета 32 этот класс выступает посредником между кодом, который затребовал отмену, и кодом, поддерживающим отмену.

Обратите внимание, что идиома сообщает об отмене, генерируя исключение `TaskCancelledException` для указания на то, что работа не была завершена. Отмененные задачи не являются задачами в состоянии ошибки. Отсюда следует, что вы никогда не должны поддерживать отмену для методов `async void` (см. совет 28). Если вы попытаетесь сделать это, тогда отмененная задача будет вызывать обработчик событий `UnhandledException`.

Наконец, давайте объединим все в общую реализацию:

```
  public Task RunPayroll(DateTime payrollPeriod) =>
    RunPayroll(payrollPeriod, new CancellationToken(), null);

  public Task RunPayroll(DateTime payrollPeriod,
    CancellationToken cancellationToken) =>
    RunPayroll(payrollPeriod, cancellationToken, null);

  public Task RunPayroll(DateTime payrollPeriod,
    IProgress<(int, string)> progress) =>
    RunPayroll(payrollPeriod, new CancellationToken(), progress);

  public async Task RunPayroll(DateTime payrollPeriod,
    CancellationToken cancellationToken,
    IProgress<(int, string)> progress)
  {
    progress?.Report((0, "Начало обработки платежной ведомости"));
    // Шаг 1: подсчет отработанных часов и размера оплаты
    var payrollData = await RetrieveEmployeePayrollDataFor(payrollPeriod);
    cancellationToken.ThrowIfCancellationRequested();
    progress?.Report((20, "Извлечение сведений о сотрудниках и отработанных
часах"));

    // Шаг 2: подсчет налогов и сообщение их размеров
    var taxReporting = new Dictionary<EmployeePayrollData, TaxWithholding>();
```

```
  foreach (var employee in payrollData)
  {
    var taxWithholding = await RetrieveTaxData(employee);
    taxReporting.Add(employee, taxWithholding);
  }
  cancellationToken.ThrowIfCancellationRequested();
  progress?.Report((40, "Подсчет удержаний"));

  // Шаг 3: генерация и отправка по электронной почте счетов на оплату
  var paystubs = new List<Task>();
  foreach (var payrollItem in taxReporting)
  {
    var payrollTask = GeneratePayrollDocument(
      payrollItem.Key, payrollItem.Value);
    var emailTask = payrollTask.ContinueWith(
      paystub => EmailPaystub(payrollItem.Key.Email,
        paystub.Result));
    paystubs.Add(emailTask);
  }
  await Task.WhenAll(paystubs);
  cancellationToken.ThrowIfCancellationRequested();
  progress?.Report((60, "Отправка по электронной почте счетов на оплату"));

  // Шаг 4: банковский перевод оплаты
  var depositTasks = new List<Task>();
  foreach (var payrollItem in taxReporting)
  {
    depositTasks.Add(MakeDeposit(payrollItem.Key,
      payrollItem.Value));
  }
  await Task.WhenAll(depositTasks);
  progress?.Report((80, "Банковский перевод оплаты"));

  // Шаг 5: закрытие платежного периода
  await ClosePayrollPeriod(payrollPeriod);
  cancellationToken.ThrowIfCancellationRequested();
  progress?.Report((100, "Обработка платежной ведомости завершена"));
}
```

Обратите внимание, что весь общий код включен в один метод. О продвижении сообщается, только когда это было запрошено. Маркер отмены создается для всех перегруженных версий, которые не поддерживают отмену, но такие перегруженные версии никогда не будут запрашивать отмену.

Как видите, модель асинхронного программирования на основе задач поддерживает развитые средства для запуска, отмены и отслеживания асинхронных операций. Предлагаемые протоколы позволяют проектировать асинхронный API-интерфейс, который представляет возможности лежащей в основе асинхронной работы. Поддерживайте один или оба дополнительных протокола, если можете обеспечить их эффективность. Когда это невозможно, не реализуйте их, чтобы не вводить в заблуждение пользователей вашего кода.

Совет 34. Кешируйте обобщенные возвращаемые типы в асинхронном коде

Каждый предмет, обсуждавшийся в модели асинхронного программирования на основе задач, использовал `Task` или `Task<T>` в качестве возвращаемого типа асинхронного кода. Они представляют собой самые распространенные типы, которые вы будете применять как возвращаемые типы для асинхронной работы. Тем не менее, временами типы `Task` привносят узкие места в плане производительности. Если вы делаете асинхронные вызовы в плотном цикле или в "горячих" путях кода, тогда класс `Task` может оказаться затратным для размещения и использования в асинхронных методах. Язык C# 7 не навязывает применение `Task` или `Task<T>` для возвращаемых типов в асинхронных методах, но взамен требует, чтобы метод с модификатором `async` возвращал тип, который соответствует паттерну "Объект ожидания" (Awaiter). Он обязан иметь доступный метод `GetAwaiter()`, возвращающий объект класса, который реализует интерфейсы `INotifyCompletion` и `ICriticalNotifyCompletion`. Такой доступный метод `GetAwaiter()` может быть предоставлен посредством расширяющего метода.

Последний выпуск инфраструктуры .NET Framework включает новый тип `ValueTask<T>`, который может оказаться эффективнее в использовании. Он является типом значения и потому не требует дополнительного размещения — фактор, сокращающий нагрузку на сборщик мусора. Тип `ValueTask<T>` лучше всего подходит для идиом, в которых асинхронный метод может извлекать кешированные результаты.

В качестве примера рассмотрим метод, проверяющий данные о погоде:

```
public async Task<IEnumerable<WeatherData>>
  RetrieveHistoricalData(DateTime start, DateTime end)
{
  var observationDate = this.startDate;
  var results = new List<WeatherData>();
  while (observationDate < this.endDate)
  {
    var observation = await RetrieveObservationData(observationDate);
    results.Add(observation);
    observationDate += TimeSpan.FromDays(1);
  }
  return results;
}
```

В текущем виде метод `RetrieveHistoricalData()` обращается к сети при каждом своем вызове. Если он является частью виджета приложения смартфона, который ежеминутно отображает краткое состояние, то такая операция приложения будет крайне неэффективной — данные о погоде не меняются настолько часто. Вы принимаете решение кешировать результаты в течение 5 минут. Реализация с применением `Task` может выглядеть следующим образом:

```
private List<WeatherData> recentObservations =
  new List<WeatherData>();
private DateTime lastReading;
```

```
public async Task<IEnumerable<WeatherData>>
  RetrieveHistoricalData()
{
  if (DateTime.Now - lastReading > TimeSpan.FromMinutes(5))
  {
    recentObservations = new List<WeatherData>();
    var observationDate = this.startDate;
    while (observationDate < this.endDate)
    {
      var observation = await RetrieveObservationData(observationDate);
      recentObservations.Add(observation);
      observationDate += TimeSpan.FromDays(1);
    }
    lastReading = DateTime.Now;
  }
  return recentObservations;
}
```

Во многих случаях такого изменения, скорее всего, будет достаточно для улучшения производительности. Наиболее значительным узким местом в показанном коде является сетевая задержка.

Но теперь предположим, что виджет запускается в среде с очень жесткими ограничениями на расходуемую память. В такой ситуации вы хотите избежать размещения объекта каждый раз, когда метод вызывается. Именно тогда имеет смысл переключиться на использование типа `ValueTask`. Вот на что похожа реализация:

```
public ValueTask<IEnumerable<WeatherData>>
  RetrieveHistoricalData()
{
  if (DateTime.Now - lastReading > TimeSpan.FromMinutes(5))
  {
    return new ValueTask<IEnumerable<WeatherData>>
      (recentObservations);
  }
  else
  {
    async Task<IEnumerable<WeatherData>> loadCache()
    {
      recentObservations = new List<WeatherData>();
      var observationDate = this.startDate;
      while (observationDate < this.endDate)
      {
        var observation = await
          RetrieveObservationData(observationDate);
        recentObservations.Add(observation);
        observationDate += TimeSpan.FromDays(1);
      }
      lastReading = DateTime.Now;
      return recentObservations;
    }
    return new ValueTask<IEnumerable<WeatherData>>(loadCache());
  }
}
```

Приведенный метод содержит несколько важных идиом, которые вы должны применять при работе с `ValueTask`. Во-первых, метод определен не как асинхронный, а как возвращающий тип `ValueTask`. Вложенная функция, выполняющая асинхронную работу, снабжена модификатором `async`. Это указывает на то, что программа не занимается дополнительным управлением состоянием и размещением, если кеш допустим. Во-вторых, обратите внимание на наличие у `ValueTask` конструктора, который принимает в своем аргументе объект `Task`. Он будет внутренне выполнять ожидающую работу.

Тип `ValueTask` позволяет внедрить оптимизацию, когда измерения производительности показывают, что выделение памяти для объектов `Task` создает узкие места в коде. Вероятно, в большинстве своих асинхронных методов вы по-прежнему будете использовать типы `Task`. На самом деле типы `Task` и `Task<T>` рекомендуется применять для всех асинхронных методов, если только вы не провели измерения и не обнаружили, что выделения памяти являются узким местом. Перейти к типу значения нетрудно, и так следует поступать, когда выясняется, что это изменение устранит проблемы, связанные с производительностью.