O'REILLY®

Второе издание

Конкурентность в C#

асинхронное, параллельное и многопоточное программирование

Стивен Клири

SECOND EDITION

Concurrency in C# Cookbook

Asynchronous, Parallel, and Multithreaded Programming

Stephen Cleary

Конкурентность в C#

асинхронное, параллельное и многопоточное программирование

Стивен Клири

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2020

ББК 32.973.2-018.1
УДК 004.43
К49

Клири Стивен

К49 Конкурентность в C#. Асинхронное, параллельное и многопоточное программирование. 2-е межд. изд. — СПб.: Питер, 2020. — 272 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1572-3

Если вы побаиваетесь конкурентного и многопоточного программирования, эта книга написана для вас. Стивен Клири предоставляет в ваше распоряжение 85 рецептов работы с .NET и C# 8.0, необходимых для параллельной обработки и асинхронного программирования. Конкурентность уже стала общепринятым методом разработки хорошо масштабируемых приложений, но параллельное программирование остается непростой задачей. Подробные примеры и комментарии к коду позволят разобраться в том, как современные инструменты повышают уровень абстракции и упрощают конкурентное программирование. Вы научитесь использовать async и await для асинхронных операций, расширять возможности кода за счет использования асинхронных потоков, исследовать потенциал параллельного программирования с библиотекой TPL Dataflow, создавать конвейеры потоков данных с библиотекой TPL Dataflow, задействовать функциональность System.Reactive на базе LINQ, использовать потоково-безопасные и неизменяемые коллекции, проводить модульное тестирование конкурентного кода, брать под контроль пул потоков, реализовывать корректную кооперативную отмену, анализировать сценарии на предмет объединения конкурентных методов, пользоваться всеми возможностями асинхронно-совместимого объектно-ориентированного программирования, распознавать и создавать адаптеры для кода, в котором используются старые стили асинхронного программирования.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1492054504 англ.

Authorized Russian translation of the English edition of Concurrency in C# Cookbook, 2nd Edition. ISBN 9781492054504 © 2019 Stephen Cleary This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

978-5-4461-1572-3

Оглавление

Следующим значительным феноменом в области компьютерных технологий станет доступность массового параллелизма для простых смертных. Сейчас разработчики могут предоставить в наше распоряжение большую вычислительную мощь, чем когда-либо, однако выражение конкурентных вычислений для многих все еще остается проблемой. Стивен уделяет внимание этой проблеме, помогая лучше разобраться в конкурентности, многопоточности, модели реактивного программирования, параллелизме и многих других темах в этом доступном, но подробном руководстве.

Скотт Ханзельман (Scott Hanselman),
главный администратор проекта, ASP.NET
и Azure Web Tools, Microsoft

Разнообразие описанных методов и формат сборника рецептов делают эту книгу идеальным руководством по современной конкурентности на платформе .NET.

Джон Скит (Jon Skeet),
старший инженер-разработчик в Google

Стивен Клири завоевал репутацию ведущего эксперта по асинхронности и параллелизму в C#. В этой книге четко и доступно представлены важнейшие положения и принципы, которые разработчик должен понимать для того, чтобы начать пользоваться этими технологиями и добиться с ними успеха.

Стивен Тауб (Stephen Toub),
главный архитектор, Microsoft

Предисловие

Животное на обложке — мусанг, или малайская пальмовая куница, — пожалуй, отлично подходит для представления темы этой книги. Пока я не увидел обложку, я ничего не знал о нем и поэтому решил поискать информацию. Мусанги считаются вредителями, потому что засоряют своим пометом чердаки и шумят. Их анальные железы выделяют секрет с противным запахом. Мусанг относится к исчезающим видам из категории «Вызывающие наименьшее опасение», что по сути является политкорректным аналогом утверждения «Убивайте сколько угодно; никому не жалко». Мусанги поедают спелые плоды кофейного дерева (кофейные вишни), которые проходят через их желудочно-кишечный тракт. Копи-лувак, один из самых дорогих видов кофе в мире, делается из кофейных зерен, извлеченных из испражнений мусанга. По утверждениям Американской ассоциации спешиалти кофе, «он просто имеет неприятный вкус».

Все это делает мусанга идеальным символом для конкурентной и многопоточной разработки. Для непосвященного конкурентность и многопоточность нежелательны. Из-за них добропорядочный код начинает вести себя совершенно непостижимым образом. Состояния гонки и т. д. приводят к катастрофическим сбоям (которые, похоже, всегда происходят в продакшен или во время демонстрации). Некоторые разработчики заходят настолько далеко, что заявляют: «Потоки — зло», и полностью избегают конкурентности. Немногочисленная группа разработчиков вошла во вкус и использует конкурентность без опасений; но многие в прошлом уже обжигались на ней, и от прошлого опыта у них остались неприятные воспоминания.

Тем не менее для современных приложений конкурентность становится практически обязательным требованием. В наши дни пользователь хочет видеть интерфейс, быстро реагирующий на происходящее, а сер-

верным приложениям приходится масштабироваться до беспрецедентных уровней. Конкурентность способствует решению проблем в обоих направлениях.

К счастью, существует множество современных библиотек, которые *значительно* упрощают конкурентность! Параллельная обработка и асинхронное программирование перестали быть уделом избранных. Эти библиотеки повышают уровень абстракции, вследствие чего разработка масштабируемых приложений с хорошей скоростью реакции становятся вполне реальным делом для любого разработчика. Если в прошлом, когда конкурентное программирование было исключительно сложным делом и вы на нем обожглись, рекомендую сделать новую попытку, вооружившись современным инструментарием. Мы, наверное, никогда не сможем назвать конкурентность простым делом, но сейчас она уже не так сложна, как раньше!

Для кого написана эта книга

Эта книга написана для разработчиков, которые хотят освоить современные подходы к конкурентному программированию. Предполагается, что читатель уже обладает опытом программирования .NET, включая понимание обобщенных коллекций, перечисляемых объектов и LINQ. Знание многопоточного или асинхронного программирования не потребуется. Если у вас имеется некоторый опыт в этих областях, книга все равно может вам пригодиться, потому что в ней представлены новые библиотеки — более безопасные и простые в использовании.

Конкурентность полезна в приложениях любого типа. Неважно, работаете ли вы над настольными, мобильными или серверными приложениями; в наши дни конкурентность стала практически обязательным требованием во всех ситуациях. Рецепты, приведенные в книге, помогут вам сделать пользовательские интерфейсы более отзывчивыми, а серверы — лучше масштабируемыми. Мы уже достигли точки, в которой конкурентность получила повсеместное распространение, и понимание этих приемов и их применений стало одним из важнейших навыков профессионального разработчика.

Почему я написал эту книгу

На заре своей карьеры я изучал многопоточное программирование методом проб и ошибок. Затем я изучал асинхронное программирование методом проб и ошибок. Хотя и то и другое принесло полезный опыт, я бы предпочел иметь тогда некоторые инструменты и ресурсы, которые доступны сейчас. В частности, поддержка `async` и `await` в современных языках .NET — настоящее сокровище.

Однако если обратиться к сегодняшним книгам и другим ресурсам для изучения конкурентности, почти все они начинаются с изложения большинства низкоуровневых концепций. Обычно приводится превосходное описание потоков и примитивов синхронизации, а высокоуровневые методы откладываются на потом, если вообще рассматриваются. Полагаю, это происходит по двум причинам. Во-первых, многие разработчики конкурентных программ (включая меня) начинали с изучения низкоуровневых концепций и подолгу корпели над описаниями старых методов. Во-вторых, многие книги были написаны давно и содержат устаревшие сведения; с появлением более новых средств книги обновлялись, но, к сожалению, новая информация размещалась в конце.

Считаю, что такой подход неверен. В этой книге рассматриваются *только* современные подходы к реализации конкурентности. Это вовсе не значит, что понимание всех низкоуровневых концепций не принесет вам пользы. Когда я изучал программирование в колледже, на одном из курсов мне пришлось строить виртуальный процессор из набора элементарных логических вентилей, а на другом — изучать программирование на языке ассемблера. За всю профессиональную карьеру я не спроектировал ни одного процессора и написал всего пару десятков строк ассемблерного кода, но понимание основ приносит пользу ежедневно. И все же лучше начинать с высокоуровневых абстракций; на моем первом курсе программирования рассматривался вовсе не язык ассемблера.

Эта книга занимает определенную нишу: она содержит введение (и справочник) по конкурентности с использованием современных методов. В ней рассматриваются различные виды конкурентности, включая параллельную обработку, асинхронное и реактивное программирование. Тем не менее в ней не рассматриваются устаревшие методы, которые описаны в других книгах и сетевых ресурсах.

Типографские соглашения

В этой книге приняты следующие типографские соглашения:

Курсив

Используется для обозначения новых терминов.

`Моноширинный шрифт`

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

Так выделяются советы и предложения.

Так обозначаются предупреждения и предостережения.

Структура книги

Книга имеет следующую структуру:

- В главе 1 содержится введение в различные виды конкурентности, описанные в книге: параллелизм, асинхронное и реактивное программирование, потоки данных.
- В главах 2–6 представлено более подробное введение в разновидности конкурентности.
- В каждой из оставшихся глав рассматривается конкретный аспект конкурентности; они также могут рассматриваться как сборник решений типичных проблем.

Рекомендую прочитать (или по крайней мере просмотреть) первую главу, даже если вы уже знакомы с некоторыми разновидностями конкурентности.

На момент отправки книги в печать .NET Core 3.0 еще находится на стадии бета-тестирования, поэтому некоторые нюансы асинхронных потоков могут измениться.

Благодарности

Эта книга просто не смогла бы появиться без помощи множества людей!

Прежде всего хочу выразить благодарность своему Господу и Спасителю Иисусу Христу. Принятие христианства стало самым важным решением в моей жизни! Если вам понадобится дополнительная информация по этой теме, вы можете связаться со мной на моем сайте http://stephencleary.com/.

Также я благодарен своей семье, простившей мне все то время, которое я мог бы провести с ней. Когда я начал работать над книгой, некоторые друзья говорили мне: «Попрощайся с семьей на следующий год!» Я думал, что они шутят. Моя жена Мэнди и наши дети Эс-Ди и Эмма проявляли понимание, когда я проводил долгие дни за работой, а потом писал по вечерам и выходным. Спасибо вам всем. Я люблю вас!

Конечно, эта книга была бы далеко не такой качественной, если бы не мои редакторы и научные редакторы: Стивен Тауб (Stephen Toub), Петр Ондерка (Petr Onderka) («svick»), Ник Палдино (Nick Paldino) («casperOne»), Ли Кэмпбелл (Lee Campbell) и Педро Феликс (Pedro Felix). И если в книгу прокрались какие-либо неточности — это целиком их вина... Шучу! Их мнение оказало неоценимую помощь в формировании (и исправлении) материала, а за все оставшиеся ошибки, конечно, отвечаю только я. Выражаю особую благодарность Стивену Таубу (Stephen Toub), который научил меня «трюку с логическим аргументом» (рецепт 14.5) и рассказал о бесчисленных нюансах, связанных с `async`, и Ли Кэмпбеллу (Lee Campbell), который помог мне освоить System.Reactive и сделать мой наблюдаемый код более идиоматическим.

Наконец, я хочу поблагодарить некоторых людей, от которых я узнал об этих методах: Стивена Тауба (Stephen Toub), Люциана Вищика

(Lucian Wischik), Томаса Левеска (Thomas Levesque), Ли Кэмпбелла (Lee Campbell), сообщество Stack Overflow и форумов MSDN, а также участников конференций по программированию в моем родном штате Мичиган. Мне нравится быть частью сообщества разработчиков ПО, и если эта книга кому-то поможет, то стоит поблагодарить многих других, показавших правильный путь. Спасибо всем!

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Обратите внимание: по всей книге автор дает перекрестные ссылки на рецепты. Нумерованные подзаголовки в главах и являются этими рецептами.

ГЛАВА 1

Конкурентность: общие сведения

Конкурентность (concurrency) является ключевым аспектом построения красивых программ. В течение десятилетий конкурентность была возможна, но реализовывалась с изрядными трудностями. Конкурентные программы создавали трудности с написанием, отладкой и сопровождением. В итоге многие разработчики выбирали более простой путь и избегали конкурентности. Благодаря библиотекам и языковым средствам, доступным для современных программ .NET, в наши дни конкурентность достигается гораздо проще. Компания Microsoft стала лидером движения к существенному понижению планки сложности конкурентности. Когда-то конкурентное программирование было уделом экспертов; но в наши дни каждый разработчик может (и должен) владеть средствами конкурентности.

Знакомство с конкурентностью

Прежде чем продолжать, стоит разобраться с терминами, которые будут использоваться в книге. Это мои собственные определения, которые я постоянно использую для того, чтобы различать разные методы программирования. Начнем с *конкурентности*.

Конкурентность

> Выполнение сразу нескольких действий в одно и то же время.

Надеюсь, полезность конкурентности сомнений не вызывает. Приложения для конечного пользователя используют конкурентность, чтобы реагировать на ввод данных пользователем *во время* записи в базу данных. Серверные приложения используют конкурентность для реакции на второй запрос в ходе завершения первого запроса. Конкурентность пригодится в любой ситуации, когда приложение должно делать что-то одно во время

работы над чем-то другим. Практически любое программное приложение может выиграть от применения конкурентности.

Многие разработчики, слыша термин «конкурентность», немедленно думают о «многопоточности». Тем не менее эти два понятия следует различать.

Многопоточность

> Форма конкурентности, использующая несколько программных потоков выполнения.

Многопоточность относится к буквальному использованию нескольких потоков. Как было продемонстрировано во многих рецептах книги, многопоточность является разновидностью конкурентности, но, безусловно, это не *единственная* форма. Непосредственное использование низкоуровневых видов многопоточности в современных приложениях практически не имеет смысла; высокоуровневые абстракции превосходят многопоточные средства старой школы как по мощи, так и по эффективности. По этой причине рассмотрение устаревших средств будет сведено к минимуму. Ни в одном из многопоточных рецептов этой книги не используются типы `Thread` или `BackgroundWorker`; они были заменены более качественными альтернативами.

Как только вы вводите команду `new Thread()`, все кончено: ваш проект уже содержит устаревший код.

Только не подумайте, что многопоточность мертва! Многопоточность продолжает жить в *пулах потоков* — полезном месте для постановки рабочих операций в очередь, которое автоматически регулируется в зависимости от нагрузки. В свою очередь, с пулом потоков становится возможной одна важная форма конкурентности: параллельная обработка.

Параллельная обработка

> Выполнение большого объема работы за счет распределения ее между несколькими потоками, выполняемыми одновременно.

Параллельная обработка (или параллельное программирование) использует многопоточность для максимально эффективного использования многоядерных процессоров. Современные процессоры часто имеют несколько ядер, и при большом объеме выполняемой работы было бы неразумно

поручать всю работу одному ядру, в то время как остальные простаивают. Параллельная обработка распределяет работу между несколькими потоками, каждый из которых может выполняться независимо на отдельном ядре.

Параллельная обработка является одной из разновидностей многопоточности, а многопоточность является одной из разновидностей конкурентности. Также существует другая разновидность конкурентности, которая важна в современных приложениях, но не так хорошо знакома многим разработчикам: *асинхронное программирование*.

Асинхронное программирование

> Разновидность конкурентности, использующая обещания или обратные вызовы для предотвращения создания лишних потоков.

Обещание (future/promise), или *преднамеченный тип* — тип, представляющий некоторую операцию, которая завершится в будущем. Примеры современных типов обещаний в .NET — `Task` и `Task<TResult>`. Более старые асинхронные API используют обратные вызовы или события вместо обещаний. В асинхронном программировании центральное место занимает идея *асинхронной операции* — некоторой запущенной операции, которая завершится через некоторое время. Хотя операция продолжается, она не блокирует исходный поток; поток, который запустил операцию, свободен для выполнения другой работы. Когда операция завершится, она уведомляет свое обещание или активизирует обратный вызов или событие, чтобы приложение узнало о завершении.

Асинхронное программирование — мощная разновидность конкурентности, оно до недавнего времени требовало чрезвычайно сложного кода. Благодаря поддержке `async` и `await` в современных языках асинхронное программирование становится почти таким же простым, как и синхронное (неконкурентности) программирование.

Другая форма конкурентности — *реактивное программирование* (reactive programming). Асинхронное программирование подразумевает, что приложение запускает операцию, которая завершится в будущем. Реактивное программирование тесно связано с асинхронным программированием, но в его основе лежат асинхронные события вместо асинхронных операций. Асинхронные события могут не иметь фактического «начала», могут происходить в любое время и могут инициироваться многократно. Один из примеров такого рода — ввод данных пользователем.

Реактивное программирование

> Декларативный стиль программирования, при котором приложение реагирует на события.

Если рассматривать приложение как огромный конечный автомат, поведение приложения может быть описано как реакция на серию событий с обновлением своего состояния на каждое событие. Это не настолько абстрактное или теоретическое определение, как может показаться: с современными фреймворками этот метод весьма полезен в реальных приложениях. Реактивное программирование не обязательно конкурентно, но оно тесно связано с конкурентностью, поэтому в книге будут изложены его основы.

Обычно при написании конкурентной программы применяется комбинация разных методов. В большинстве приложений используется как минимум многопоточность (через пул потоков) и асинхронное программирование. Вы можете свободно смешивать разные формы конкурентности, используя подходящий инструмент для каждой части приложения.

Введение в асинхронное программирование

Асинхронное программирование обладает двумя главными преимуществами. Первое характерно для программ с графическим интерфейсом (GUI), предназначенных для пользователя: асинхронное программирование обеспечивает быстрый отклик. Каждому из нас попадались программы, которые вдруг зависают во время работы; асинхронная программа сможет быстро реагировать на действия пользователя во время работы. Второе преимущество характерно для программ, работающих на стороне сервера: асинхронное программирование обеспечивает масштабируемость. Серверное приложение может в некоторой степени масштабироваться за счет использования пула потоков, но асинхронное серверное приложение обычно обладает на порядок лучшими возможностями масштабирования.

Оба преимущества асинхронного программирования обусловлены одним и тем же аспектом: асинхронное программирование освобождает потоки. Для GUI-программ асинхронное программирование освобождает UI-поток; это позволяет графическому приложению сохранить высокую скорость отклика на ввод пользователя. Для серверных приложений асинхронное

программирование освобождает потоки запросов и позволяет серверу использовать свои потоки для обслуживания большего количества запросов.

В современных асинхронных приложениях .NET используются два ключевых слова: `async` и `await`. Ключевое слово `async` добавляется в объявление метода и имеет двойное назначение: оно разрешает использование ключевого слова `await` внутри этого метода и приказывает компилятору сгенерировать для этого метода конечный автомат по аналогии с тем, как работает `yield return`. Метод с ключевым словом `async` может вернуть `Task<TResult>`, если он возвращает значение; `Task` — если он не возвращает значения; или любой другой «сходный» тип — такой, как `ValueTask`. Кроме того, `async`-метод может вернуть `IAsyncEnumerable<T>` или `IAsyncEnumerator<T>`, если он возвращает несколько значений в перечислении. «Сходные» типы представляют обещания; они могут уведомлять вызывающий код о завершении `async`-метода.

Избегайте `async void`! Возможно создать async-метод, который возвращает void, но это следует делать только при написании async-обработчика событий. Обычный async-метод без возвращаемого значения должен возвращать `Task`, а не `void`.

С учетом всего сказанного рассмотрим короткий пример:

```
async Task DoSomethingAsync()
{
  int value = 13;

  // Асинхронно ожидать 1 секунду.
  await Task.Delay(TimeSpan.FromSeconds(1));

  value *= 2;

  // Асинхронно ожидать 1 секунду.
  await Task.Delay(TimeSpan.FromSeconds(1));

  Trace.WriteLine(value);
}
```

`async`-метод начинает выполняться синхронно, как и любой другой метод. Внутри `async`-метода команда `await` выполняет асинхронное ожидание по своему аргументу. Сначала она проверяет, завершилась ли операция: если

да, то метод продолжает выполняться (синхронно). В противном случае `await` приостанавливает `async`-метод и возвращает незавершенную задачу. Когда операция завершится позднее, метод `async` продолжает выполнение.

`async`-метод может рассматриваться как состоящий из нескольких синхронных частей, разделенных командами `await`. Первая синхронная часть выполняется в потоке, который вызвал метод, но где выполняются другие синхронные части? Ответ на этот вопрос не прост.

При выполнении `await` для задачи (самый распространенный сценарий) в момент, когда `await` решает приостановить метод, сохраняется *контекст*. Это текущий объект `SynchronizationContext`, если только он не равен `null` (в этом случае контекстом является текущий объект `TaskScheduler`). Метод возобновляет выполнение в этом сохраненном контексте. Обычно контекстом является UI-контекст (для UI-потока) или контекст пула потоков (в большинстве других ситуаций). Если вы пишете приложение ASP.NET Classic (до Core), то контекстом также может быть контекст запроса ASP.NET. В ASP.NET Core используется контекст пула потоков вместо специального контекста запроса.

Таким образом, в приведенном коде все синхронные части пытаются возобновить продолжение в исходном контексте. Если вызвать метод `DoSomethingAsync` из UI-потока, каждая из его синхронных частей будет выполняться в этом UI-потоке, но если вызвать его из потока из пула потоков, то каждая из синхронных частей будет выполняться в любом потоке из пула потоков.

Чтобы обойти это поведение по умолчанию, можно выполнить `await` по результату метода расширения `ConfigureAwait` с передачей `false` в параметре `continueOnCapturedContext`. Следующий код начинает выполнение в вызывающем потоке, а после приостановки `await` он возобновляет выполнение в потоке из пула потоков:

```
async Task DoSomethingAsync()
{
  int value = 13;

  // Асинхронно ожидать 1 секунду.
  await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);

  value *= 2;

  // Асинхронно ожидать 1 секунду.
```

```
  await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);

  Trace.WriteLine(value);
}
```

Хорошей практикой программирования считается вызывать ConfigureAwait в базовых «библиотечных» методах и возобновлять контекст только тогда, когда потребуется — в ваших внешних методах «пользовательского интерфейса».

Ключевое слово `await` не ограничивается работой с задачами, оно может работать с любым объектом, *допускающим ожидание* (awaitable), построенным по определенной схеме. Например, библиотека Base Class Library включает тип `ValueTask<T>`, который сокращает затраты памяти, если результат в основном является синхронным; например, если результат может быть прочитан из кэша в памяти. Тип `ValueTask<T>` не преобразуется в `Task<T>` напрямую, но строится по схеме, допускающей ожидание, поэтому может использоваться с `await`. Также существуют другие примеры, и вы можете строить свои собственные, но в большинстве случаев `await` получает `Task` или `Task<TResult>`.

Существует два основных способа создания экземпляров `Task`. Некоторые задачи представляют реальный код, который должен выполняться процессором; такие вычислительные задачи должны создаваться вызовом `Task.Run` (или `TaskFactory.StartNew`, если они должны выполняться по определенному расписанию). Другие задачи представляют *уведомления*; такие задачи, основанные на событиях, создаются `TaskCompletionSource<TResult>` (или одной из сокращенных форм). Большинство задач ввода/вывода использует `TaskCompletionSource<TResult>`.

Обработка ошибок с `async` и `await` выглядит логично. В следующем фрагменте кода `PossibleExceptionAsync` может выдать исключение `NotSupportedException`, но `TrySomethingAsync` может перехватить исключение естественным образом. Трассировка стека перехваченного исключения сохраняется без искусственной упаковки в `TargetInvocationException` или `AggregateException`:

```
async Task TrySomethingAsync()
{
  try
```

```
  {
    await PossibleExceptionAsync();
  }
  catch (NotSupportedException ex)
  {
    LogException(ex);
    throw;
  }
}
```

Когда `async`-метод выдает (или распространяет) исключение, оно помещается в возвращаемый объект `Task`, и задача `Task` завершается. При выполнении `await` для этого объекта `Task` оператор `await` получает это исключение и (заново) выдает его так, что исходная трассировка стека сохраняется. Такой код, как в примере ниже, будет работать так, как ожидается, если `PossibleExceptionAsync` является `async`-методом:

```
async Task TrySomethingAsync()
{
  // Исключение попадает в Task, а не выдается напрямую.
  Task task = PossibleExceptionAsync();

  try
  {
    // Исключение из Task exception будет выдано здесь, в точке await.
    await task;
  }
  catch (NotSupportedException ex)
  {
    LogException(ex);
    throw;
  }
}
```

Относительно `async`-методов существует одна важная рекомендация: при использовании ключевого слова `async` лучше позволить ему распространяться в вашем коде. Если вы вызываете `async`-метод, следует (в конечном итоге) выполнить `await` для возвращаемой им задачи. Боритесь с искушением вызвать `Task.Wait`, `Task<TResult>.Result` или `GetAwaiter().GetResult()`: это приведет к взаимоблокировке (deadlock). Рассмотрим следующий метод:

```
async Task WaitAsync()
{
  // await сохранит текущий контекст ...
  await Task.Delay(TimeSpan.FromSeconds(1));
  // ... и попытается возобновить метод в этой точке с этим контекстом.
}

void Deadlock()
{
  // Начать задержку.
  Task task = WaitAsync();
  // Синхронное блокирование с ожиданием завершения async-метода.
  task.Wait();
}
```

Код в этом примере создаст взаимоблокировку при вызове из UI-контекста или контекста ASP.NET Classic, потому что оба эти контекста допускают выполнение только одного потока. `Deadlock` вызовет `WaitAsync`, что приводит к началу задержки. Затем `Deadlock` (синхронно) ожидает завершения этого метода с блокированием контекстного потока. Когда задержка завершится, `await` пытается возобновить `WaitAsync` в сохраненном контексте, но не сможет, так как в контексте уже есть заблокированный поток, а контекст допускает только один поток в любой момент времени. Взаимоблокировку можно предотвратить двумя способами: использовать `ConfigureAwait(false)` в `WaitAsync` (что заставляет `await` игнорировать его контекст) или же использовать `await` с вызовом `WaitAsync` (что превращает `Deadlock` в `async`-метод).

Используйте `async` по полной программе.

Если вы хотите более подробно изучить `async`, компания Microsoft предоставляет великолепную документацию по этой теме. Рекомендую прочитать по крайней мере обзор «Asynchronous Programming» и «Task-based Asynchronous Pattern (TAP)». Кроме этого, также имеется документация «Async in Depth».

Асинхронные потоки берут основу `async` и `await` и расширяют ее для работы с множественными значениями. Асинхронные потоки строятся

на основе концепции асинхронных перечисляемых объектов, которые похожи на обычные перечисляемые объекты (enumerables), за исключением того, что позволяют выполнить асинхронную работу при получении следующего элемента последовательности. Это исключительно мощная концепция, которая более подробно рассматривается в главе 3. Асинхронные потоки особенно полезны тогда, когда имеется последовательность данных, поступающих либо поодиночке, либо блоками. Например, если приложение обрабатывает ответ API, в котором используется разбиение на страницы с параметрами `limit` и `offset`, асинхронные потоки могут стать идеальной абстракцией. На момент написания книги асинхронные потоки были доступны только на новейших платформах .NET.

Введение в параллельное программирование

Параллельное программирование следует использовать в любой ситуации, в которой серьезный объем вычислительной работы может быть разделен на независимые блоки. Параллельное программирование временно повышает загрузку процессора для улучшения пропускной способности системы; это может быть полезно в клиентских системах, в которых процессор часто простаивает, но в серверных системах обычно неуместно. У большинства серверов присутствуют некоторые встроенные средства параллелизма; например, ASP.NET обрабатывает несколько запросов параллельно. Написание параллельного кода на сервере может приносить пользу в некоторых ситуациях (если вам *известно*, что количество одновременно обслуживаемых пользователей всегда будет низким), но, как правило, параллельное программирование на сервере будет конфликтовать со встроенными параллельными средствами и не принесет никакой реальной пользы.

Есть две формы параллельного программирования: *параллелизм данных* и *параллелизм задач*. Параллелизм данных возникает тогда, когда имеется набор элементов данных, ожидающих обработки, и обработка каждого фрагмента данных в основном не зависит от других фрагментов. Под параллелизмом задач понимается такая ситуация, в которой имеется некоторый пул работы, где каждый фрагмент работы в основном не зависит от остальных. Параллелизм задач может быть динамическим — если один фрагмент работы порождает несколько дополнительных фрагментов работы, они могут быть добавлены в пул работы.

Известно несколько разных подходов к реализации параллелизма данных. Метод `Parallel.ForEach` является аналогом цикла `foreach` и должен использоваться там, где это возможно. `Parallel.ForEach` рассматривается в рецепте 4.1. Класс `Parallel` также поддерживает `Parallel.For` — аналог цикла `for` и может использоваться, если обработка данных зависит от индекса. Код, использующий `Parallel.ForEach`, выглядит примерно так:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees)
{
  Parallel.ForEach(matrices, matrix => matrix.Rotate(degrees));
}
```

Другой вариант — PLINQ (Parallel LINQ), предоставляющий метод расширения `AsParallel` для запросов LINQ. `Parallel` более эффективно расходует ресурсы, чем PLINQ; `Parallel` лучше сосуществует с другими процессами в системе, тогда как PLINQ (по умолчанию) будет пытаться распространиться по всем процессорам. К недостаткам `Parallel` следует отнести то, что он требует более явной реализации; PLINQ во многих случаях позволяет писать более элегантный код. PLINQ рассматривается в рецепте 4.5 и выглядит примерно так:

```
IEnumerable<bool> PrimalityTest(IEnumerable<int> values)
{
  return values.AsParallel().Select(value => IsPrime(value));
}
```

Что бы вы ни выбрали, есть одна рекомендация, которая справедлива при выполнении параллельной обработки.

Блоки работы должны быть независимы друг от друга настолько, насколько это возможно.

Независимость блока работы от всех остальных блоков обеспечивает максимизацию параллелизма. Как только вы начнете использовать совместный доступ к состоянию в разных потоках, придется синхронизировать доступ к общему состоянию, и ваше приложение становится менее параллельным. Синхронизация более подробно рассматривается в главе 12.

Результаты параллельной обработки могут обрабатываться разными способами. Выход можно поместить в некоторую разновидность конкурент-

ной коллекции или же провести агрегирование результатов для получения сводного показателя. Агрегирование часто применяется в параллельной обработке; такая разновидность функциональности «отображение/свертка» также поддерживается перегруженными версиями методов класса `Parallel`. Агрегирование более подробно рассматривается в рецепте 4.2.

Давайте рассмотрим параллелизм задач. Параллелизм данных ориентирован на обработку данных, а параллелизм задач — на выполнение работы. На высоком уровне между параллелизмом данных и параллелизмом задач есть много общего; «обработка данных» может рассматриваться как разновидность «работы». Многие задачи параллелизма могут решаться любым из этих способов; используйте тот API, который покажется вам более естественным для текущей задачи.

`Parallel.Invoke` — одна из разновидностей метода `Parallel`, которая реализует разновидность параллелизма задач типа «ветвление/объединение». Этот метод рассматривается в рецепте 4.3; вы просто передаете делегатов, которые должны выполняться параллельно:

```
void ProcessArray(double[] array)
{
  Parallel.Invoke(
      () => ProcessPartialArray(array, 0, array.Length / 2),
      () => ProcessPartialArray(array, array.Length / 2, array.Length)
  );
}

void ProcessPartialArray(double[] array, int begin, int end)
{
  // Действия, интенсивно использующие процессор...
}
```

Тип `Task` изначально был разработан для параллелизма задач, хотя он также использовался для асинхронного программирования. Экземпляр `Task` — в том виде, в котором используется в параллелизме задач, — представляет некоторую работу. Метод `Wait` может использоваться для ожидания завершения задачи, а свойства `Result` и `Exception` — для получения результатов этой работы. Код, использующий `Task` напрямую, сложнее кода, в котором используется `Parallel`, но и он может быть полезным, если структура параллелизма неизвестна до стадии выполнения. С этой разновидностью динамического параллелизма количество необходимых

фрагментов работы неизвестно до начала обработки; это выясняется во время выполнения. В общем случае динамический фрагмент работы должен запускать все дочерние задачи, необходимые ему, а затем ожидать их завершения. У типа `Task` имеется специальный флаг `TaskCreationOptions.AttachedToParent`, который может использоваться для этой цели. Динамический параллелизм рассматривается в рецепте 4.4.

Параллелизм задач должен стремиться к независимости составляющих, как и параллелизм данных. Чем более независимы ваши делегаты, тем эффективнее программа. Кроме того, если делегаты зависимы друг от друга, их придется синхронизировать, а синхронизация усложняет написание правильного кода. При параллелизме задач следует особенно внимательно следить за переменными, сохраненными в *замыканиях* (closures). Помните, что в замыканиях сохраняются ссылки (а не значения), и это может привести к неочевидным ситуациям с совместным использованием данных.

Обработка ошибок при всех типах параллелизма организуется аналогично. Так как операции выполняются параллельно, в программе могут возникнуть множественные исключения, поэтому они упаковываются в исключение `AggregateException`, запускаемое в ваш код. Это поведение последовательно реализуется для `Parallel.ForEach`, `Parallel.Invoke`, `Task.Wait` и т. д. Тип `AggregateException` содержит полезные методы `Flatten` и `Handle`, упрощающие код обработки ошибок:

```
try
{
  Parallel.Invoke(() => { throw new Exception(); },
      () => { throw new Exception(); });
}
catch (AggregateException ex)
{
  ex.Handle(exception =>
  {
    Trace.WriteLine(exception);
    return true; // "обработано"
  });
}
```

Обычно не приходится беспокоиться о том, как пул потоков организует выполнение работы. Параллелизм данных и задач используют динамически регулируемые *распределители* (partitioners) для распре-

деления работы между рабочими потоками. Пул потоков увеличивает количество потоков по мере необходимости. Он имеет одну рабочую очередь, и каждый поток из пула потоков использует собственную рабочую очередь. Когда поток из пула ставит в очередь дополнительную работу, то сначала отправляет ее в свою очередь, так как работа обычно связывается с текущим рабочим элементом (work item); такое поведение заставляет потоки заниматься своей собственной частью работы и максимизирует процент попаданий в кэш. Если у другого потока нет работы, он забирает работу из очереди другого потока. Компания Microsoft потратила много сил на то, чтобы пул потоков по возможности работал эффективно; существует множество настроек, которые можно изменять для достижения максимального быстродействия. Если ваши задачи не слишком малы, они должны хорошо работать с настройками по умолчанию.

Задачи должны быть ни слишком короткими, ни слишком длинными.

Если задачи получаются слишком короткими, то затраты ресурсов на разбиение данных на задачи и планирование этих задач в пуле потоков начинают играть значительную роль. Если задачи слишком длинные, то пул потоков не может динамически регулировать равномерное распределение работы. Трудно заранее определить, какую задачу следует считать «слишком короткой» или «слишком длинной»; это зависит от решаемой задачи и приблизительных возможностей оборудования. Как правило, я стараюсь делать свои задачи как можно более короткими без создания проблем быстродействия (если быстродействие внезапно падает, значит задачи слишком короткие). Еще лучше не работать с задачами напрямую, а воспользоваться типом `Parallel` или PLINQ. Эти высокоуровневые формы параллелизма содержат встроенные средства распределения работы, которые решают эту задачу за вас (и вносят необходимые корректировки во время выполнения).

Если вы хотите глубже изучить тему параллельного программирования, то лучшая книга по этой теме — «Parallel Programming with Microsoft .NET» Колина Кэмпбелла и др. (Colin Campbell et al., Microsoft Press).

Введение в реактивное программирование (Rx)

Изучение реактивного программирования занимает больше времени, чем другие формы конкурентности, а сопровождение кода создает больше проблем, если только вы не эксперт в области реактивного программирования. Но если вы не пожалеете времени и сил, реактивное программирование открывает исключительно мощные возможности. Реактивное программирование позволяет рассматривать поток событий как поток данных. Как правило, если событию передаются какие-либо аргументы, то в коде лучше использовать `System.Reactive` вместо обычного обработчика событий.

Ранее пакет System.Reactive назывался Reactive Extensions; это название часто сокращалось до «Rx.» Все три термина относятся к одной технологии.

Реактивное программирование основано на концепции *наблюдаемых потоков* (observable streams). Подписавшись на наблюдаемый поток, вы будете получать любое количество элементов данных (`OnNext`); поток может завершиться одной ошибкой (`OnError`) или уведомлением «конец потока» (`OnCompleted`). Некоторые наблюдаемые потоки никогда не завершаются. Реальные интерфейсы выглядят так:

```
interface IObserver<in T>
{
  void OnNext(T item);
  void OnCompleted();
  void OnError(Exception error);
}

interface IObservable<out T>
{
  IDisposable Subscribe(IObserver<TResult> observer);
}
```

Однако вам никогда не придется реализовать эти интерфейсы. Библиотека System.Reactive (Rx) компании Microsoft содержит все реализации, которые могут понадобиться. Код Reactive в конечном итоге очень похож на LINQ; его можно рассматривать как своего рода «LINQ to Events». System.Reactive содержит все возможности LINQ, а также добавляет большое

количество собственных операторов — особенно предназначенных для работы со временем. Следующий код начинается с незнакомых операторов (`Interval` и `Timestamp`) и завершается `Subscribe`, но в середине находятся операторы `Where` и `Select`, которые должны быть знакомы вам по LINQ:

```
Observable.Interval(TimeSpan.FromSeconds(1))
    .Timestamp()
    .Where(x => x.Value % 2 == 0)
    .Select(x => x.Timestamp)
    .Subscribe(x => Trace.WriteLine(x));
```

Пример кода начинается с запуска счетчика по периодическому таймеру (`Interval`) и добавления временной метки для каждого события (`Timestamp`). Затем события фильтруются так, чтобы включались только четные значения счетчика (`Where`), выбираются значения временной метки (`Timestamp`), после чего каждое поступившее значение временной метки записывается в отладчик (`Subscribe`). Не беспокойтесь, если новые операторы (например, `Interval`) покажутся непонятными; мы рассмотрим их позже. Пока просто помните, что это запрос LINQ, очень похожий на уже знакомые вам. Главное отличие заключается в том, что LINQ to Objects и LINQ to Entities используют *модель вытягивания* (pull model), при которой перечисление запроса LINQ «вытягивает» данные из запроса, тогда как LINQ to Events (System.Reactive) использует *модель проталкивания* (push model), при которой события поступают и перемещаются по запросу сами по себе.

Определение наблюдаемого потока не зависит от его подписок. Последний пример эквивалентен следующему коду:

```
IObservable<DateTimeOffset> timestamps =
    Observable.Interval(TimeSpan.FromSeconds(1))
        .Timestamp()
        .Where(x => x.Value % 2 == 0)
        .Select(x => x.Timestamp);
timestamps.Subscribe(x => Trace.WriteLine(x));
```

Для типа нормально определять наблюдаемые потоки и делать их доступными в виде ресурса `IObservable<TResult>`. Затем другие типы могут подписываться на эти потоки или объединять их с другими операторами для создания другого наблюдаемого потока.

Подписка System.Reactive также является ресурсом. Операторы `Subscribe` возвращают реализацию `IDisposable`, которая представляет подписку.

Когда ваш код завершит прослушивание наблюдаемого потока, он должен прекратить свою подписку.

Подписки ведут себя по-разному с холодными и горячими наблюдаемыми объектами. *Горячий* (hot) наблюдаемый объект представляет собой поток событий, который всегда находится в движении, и, если при появлении события нет ни одного подписчика, оно теряется. Например, перемещение мыши является горячим наблюдаемым событием. У *холодного* (cold) наблюдаемого объекта события не поступают постоянно. Холодный наблюдаемый объект реагирует на подписку, начиная последовательность событий. Например, загрузка HTTP является холодным наблюдаемым объектом; подписка инициирует отправку запроса HTTP.

Оператор `Subscribe` также всегда должен получать параметр обработки ошибок. В предыдущих примерах этого параметра нет; ниже приведен более правильный пример, который будет правильно реагировать, если наблюдаемый поток завершается с ошибкой:

```
Observable.Interval(TimeSpan.FromSeconds(1))
    .Timestamp()
    .Where(x => x.Value % 2 == 0)
    .Select(x => x.Timestamp)
    .Subscribe(x => Trace.WriteLine(x),
        ex => Trace.WriteLine(ex));
```

`Subject<TResult>` — один из типов, который может пригодиться при экспериментах с System.Reactive. Он напоминает ручную реализацию наблюдаемого потока. Ваш код может вызывать `OnNext`, `OnError` и `OnCompleted`, а объект будет передавать эти вызовы своим подписчикам. `Subject<TResult>` хорошо подходит для экспериментов, но в реально эксплуатируемом коде лучше использовать операторы вроде тех, которые показаны в главе 6.

Существует множество полезных операторов System.Reactive, и в этой книге рассматриваются лишь отдельные примеры. За дополнительной информацией о System.Reactive рекомендую обращаться к превосходной электронной книге «Introduction to Rx» (http://introtorx.com/).

Введение в Dataflow

Библиотека TPL Dataflow — интересное сочетание асинхронных и параллельных технологий. Эта библиотека может быть полезной для последо-

вательности процессов, которые должны применяться к вашим данным. Представьте, что нужно загрузить данные по URL-адресу, разобрать их, а затем обработать параллельно с другими данными. TPL Dataflow обычно используется в качестве простого конвейера: данные входят с одного конца и перемещаются, пока не выйдут с другого конца. Однако возможности TPL Dataflow этим далеко не ограничиваются; библиотека способна справиться с сетчатыми (mesh) структурами любого типа. Вы можете определять в сетях ветвления, объединения и циклы — TPL Dataflow обработает их так, как нужно. Но в большинстве случаев сети TPL Dataflow используются как конвейеры.

Базовым структурным элементом сети потока данных (dataflow mesh) является *блок потока данных* (dataflow block). Блок может быть блоком-приемником (получение данных), блоком-источником (производство данных) или их сочетанием. Блоки-источники могут связываться с блоками-приемниками для формирования сети; связывание рассматривается в рецепте 5.1. Блоки являются полунезависимыми; они обрабатывают данные по мере поступления и передают результат дальше. В обычном способе использования TPL Dataflow вы создаете все блоки, устанавливаете связи между ними, а затем начинаете подавать данные с одного конца. Данные после этого выходят с другого конца сами по себе. Еще раз уточню, что возможности потоков данных этим не ограничиваются; можно создавать связи и добавлять их в сеть в то время, когда по ним перемещаются данные, но это весьма нетривиальный сценарий.

Блоки-приемники содержат буферы для получаемых данных. Наличие буфера позволяет им получать новые элементы данных даже в том случае, если они еще не готовы к их обработке; это позволяет данным перемещаться по сети. Такая буферизация может создать проблемы в сценариях с ветвлением, в которых один блок-источник связывается с двумя блоками-приемниками. Если у блока-источника имеются данные для отправки по направлению потока, он начинает предлагать их своим связанным блокам по одному. По умолчанию первый блок-приемник просто получает данные и буферизует их, а второй блок-приемник эти данные никогда не получит. Проблема решается ограничением буферов блоков-приемников; эта тема рассматривается в рецепте 5.4.

Если что-то пойдет не так, происходит *отказ* блока — например, если обрабатывающий делегат выдает исключение при обработке элемента данных. Когда в блоке происходит отказ, он перестает получать данные.

По умолчанию это не приводит к нарушению работоспособности всей сети, а позволяет перестроить эту часть сети или перенаправить данные. Тем не менее это нетривиальный сценарий; в большинстве случаев обычно нужно, чтобы отказы распространялись по связям к целевым блокам. Поток данных тоже поддерживает этот вариант; единственный неочевидный аспект — исключение, распространяемое по связям, упаковывается в `AggregateException`. Следовательно, при длинном конвейере могут появляться исключения с большой глубиной вложенности; проблему можно обойти с помощью метода `AggregateException.Flatten`:

```
try
{
  var multiplyBlock = new TransformBlock<int, int>(item =>
  {
    if (item == 1)
      throw new InvalidOperationException("Blech.");
    return item * 2;
  });
  var subtractBlock = new TransformBlock<int, int>(item => item - 2);
  multiplyBlock.LinkTo(subtractBlock,
      new DataflowLinkOptions { PropagateCompletion = true });

  multiplyBlock.Post(1);
  subtractBlock.Completion.Wait();
}
catch (AggregateException exception)
{
  AggregateException ex = exception.Flatten();
  Trace.WriteLine(ex.InnerException);
}
```

Обработка ошибок в потоках данных более подробно рассматривается в рецепте 5.2.

На первый взгляд сети потоков данных очень похожи на наблюдаемые потоки. Как у сетей, так и у потоков существует концепция элементов данных, которые в них перемещаются. Кроме того, у сетей и у потоков есть концепции нормального завершения (уведомление о том, что данные перестали поступать) и завершения с отказом (уведомление о том, что в ходе обработки данных произошла некоторая ошибка). Но System. Reactive (Rx) и TPL Dataflow обладают разными возможностями. Наблюдаемые объекты Rx в общем случае лучше блоков потока данных

при выполнении любых операций, связанных с хронометражом. Блоки потоков данных в общем случае лучше наблюдаемых объектов Rx при выполнении параллельной обработки. На концептуальном уровне работа Rx напоминает настройку обратных вызовов: каждый шаг наблюдаемого объекта напрямую вызывает следующий шаг. С другой стороны, каждый блок в сети потока данных практически независим от всех остальных блоков. Как Rx, так и TPL Dataflow имеют собственные области применения, которые отчасти перекрываются. Они также хорошо работают вместе; в рецепте 8.8 рассматриваются возможности взаимодействия между Rx и TPL Dataflow.

Если вы знакомы с акторскими фреймворками, то увидите, что TPL Dataflow на первый взгляд имеет ряд общих черт с ними. Каждый блок потока данных независим от других — он запускает задачи для выполнения работы по мере необходимости (например, выполнения преобразующего делегата или передачи вывода следующему блоку). Можно также настроить каждый блок для параллельного выполнения, чтобы он запускал несколько задач для обработки дополнительного ввода. Из-за этого поведения каждый блок отчасти напоминает актора в акторских фреймворках. Но TPL Dataflow не является полноценным акторским фреймворком; в частности, отсутствует встроенная поддержка корректного восстановления после ошибок или повторных попыток. TPL Dataflow — библиотека с функциональностью, сходной с функциональностью акторов, но не являющаяся полноценным акторским фреймворком.

Самые распространенные типы блоков — `TransformBlock<TInput, TOutput>` (аналог LINQ `Select`), `TransformManyBlock<TInput, TOutput>` (аналог LINQ `SelectMany`) и `ActionBlock<TResult>`, выполняющий делегата для каждого элемента данных. За дополнительной информацией о TPL Dataflow я рекомендую обращаться к документации MSDN (https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/dataflow-task-parallel-library?redirectedfrom=MSDN) и руководству «Guide to Implementing Custom TPL Dataflow Blocks» (https://blogs.msdn.microsoft.com/b/pfxteam/archive/2011/12/05/10244302.aspx).

Введение в многопоточное программирование

Поток является независимым исполнителем (executor). Каждый процесс состоит из нескольких потоков, и все эти потоки могут выполнять разные

операции одновременно. Каждый поток имеет собственный независимый стек, но он совместно использует память со всеми остальными потоками процесса. В некоторых приложениях существует один специальный поток. Например, приложения с пользовательским интерфейсом имеют один специальный UI-поток, а у консольных приложений существует один специальный главный поток.

У каждого приложения .NET имеется пул потоков. Пул потоков содержит набор рабочих потоков, готовых к выполнению любой работы, которая им будет назначена. Пул потоков отвечает за определение количества потоков, находящихся в пуле потоков в любой момент времени. Есть десятки настроек конфигурации, с которыми можно экспериментировать для изменения этого поведения, но я не рекомендую это делать; пул потоков был тщательно оптимизирован для большинства реальных ситуаций.

Создавать новые потоки самостоятельно вам не потребуется. Единственная ситуация, в которой может возникнуть необходимость в создании экземпляров `Thread`, — создание потоков STA для COM-взаимодействий.

Поток относится к низкоуровневым абстракциям. Пул потоков находится на чуть более высоком уровне абстракции; когда код ставит работу в очередь пула потоков, то сам пул потоков в случае необходимости позаботится о создании потока. Абстракции, рассмотренные в книге, находятся на еще более высоком уровне: параллельная обработка и потоки данных ставят работу в очередь пула потоков по мере необходимости. Код, использующий эти высокоуровневые абстракции, пишется проще, чем код, работающий с низкоуровневыми абстракциями.

По этой причине типы `Thread` и `BackgroundWorker` в книге не рассматриваются вообще. Их время прошло.

Коллекции для конкурентных приложений

Существует пара разновидностей коллекций, которые могут принести пользу при конкурентном программировании: *конкурентные коллекции* и *неизменяемые коллекции*. Обе категории коллекций рассматриваются в главе 9. Конкурентные коллекции позволяют нескольким потокам обновлять их одновременно с обеспечением безопасности. Многие конкурентные коллекции используют *снимки* (snapshots) текущего состояния, чтобы

один поток мог перечислять значения, пока другой может добавлять или удалять значения. Конкурентные коллекции обычно работают эффективнее простой защиты обычной коллекции с помощью блокировок (lock).

С неизменяемыми коллекциями дело обстоит иначе. Неизменяемая коллекция действительно не может изменяться; вместо этого для модификации неизменяемой коллекции создается новая коллекция, представляющая измененную коллекцию. Это может показаться ужасно неэффективным, но неизменяемые коллекции разделяют максимально возможный объем памяти между экземплярами коллекций, поэтому все не так плохо. Одно из достоинств неизменяемых коллекций заключается в том, что все их операции являются чистыми, поэтому они очень хорошо работают в сочетании с функциональным кодом.

Современная разработка

У многих современных технологий есть одно сходство: они функциональны по своей природе. В данном случае речь идет не о *функциональности* в том смысле, что «они делают то, что положено», а в смысле стиля программирования, основанного на композиции функций. И если вы возьмете на вооружение функциональный менталитет, ваши конкурентные архитектуры будут менее запутанными.

Одним из принципов функционального программирования является *чистота* (т. е. отсутствие побочных эффектов). Каждый компонент решения получает некоторое значение(-я) на входе и выдает некоторое значение(-я) на выходе. Эти компоненты должны настолько, насколько это возможно, избегать зависимости этих компонентов от глобальных (или общих) переменных или обновления глобальных (или общих) структур данных. Это справедливо для любых компонентов: `async`-методов, параллельных задач, операций System.Reactive или блоков потоков данных. Конечно, рано или поздно ваши вычисления должны на что-то повлиять, но код будет более элегантным, если вы сможете провести *обработку* в чистых блоках, а затем проводить обновления с *результатами*.

Другой принцип функционального программирования — *неизменяемость*. Неизменяемость означает, что блок данных не может изменяться. Почему неизменяемость полезна в конкурентных программах? Одна из причин заключается в том, что для неизменяемых данных не нужна синхрони-

зация; если данные не могут измениться, то синхронизация становится излишней. Кроме того, неизменяемые данные помогают предотвратить побочные эффекты. Разработчики все чаще используют неизменяемые типы, и в книге представлено несколько рецептов, в которых рассматриваются неизменяемые структуры данных.

О ключевых технологиях кратко

Фреймворк .NET в некоторой степени поддерживает асинхронное программирование с самых первых версий. Тем не менее асинхронное программирование было достаточно трудным делом до 2012 года, когда в .NET 4.5 (вместе с C# 5.0 и VB 2012) появились ключевые слова `async` и `await`. В этой книге во всех асинхронных рецептах используется современный подход с `async`/`await` и есть рецепты, демонстрирующие взаимодействие `async` и более старых паттернов асинхронного программирования. Если вам понадобится поддержка старых платформ, обращайтесь к приложению A.

Библиотека Task Parallel Library была представлена в .NET 4.0 с полной поддержкой как параллелизма данных, так и параллелизма задач. В наши дни она доступна даже на платформах с меньшими ресурсами, включая мобильные телефоны. Библиотека TPL построена на базе .NET.

Команда разработчиков System.Reactive приложила немало усилий для поддержки максимального количества платформ. System.Reactive, как и `async` с `await`, предоставляет полезные возможности для любых типов приложений — как клиентских, так и серверных. Поддержка System.Reactive доступна в пакете `System.Reactive`.

Библиотека TPL Dataflow официально распространяется в составе пакета NuGet для `System.Threading.Tasks.Dataflow`.

Многие конкурентные коллекции встроены в .NET; также существуют другие конкурентные коллекции, содержащиеся в пакете `System.Threading.Channels`. Неизменяемые коллекции доступны в пакете `System.Collections.Immutable`.

ГЛАВА 2

Основы async

В этой главе будут представлены основы использования синтаксиса `async` и `await` для асинхронных операций. Мы рассмотрим только естественные асинхронные операции: запросы HTTP, команды баз данных и вызовы веб-служб.

Если имеется операция, создающая интенсивную нагрузку на процессор, которую вы хотели бы рассматривать как асинхронную (например, чтобы она не блокировала UI-поток), обращайтесь к главе 4 и рецепту 8.4. Кроме того, в этой главе рассматриваются только операции, которые один раз начинаются и один раз завершаются; если нужно обрабатывать потоки событий, обращайтесь к главам 3 и 6.

2.1. Приостановка на заданный период времени

Задача

Требуется (асинхронно) приостановить выполнение программы на некоторый период времени. Такая ситуация часто встречается при модульном тестировании или реализации задержки для повторного использования. Она также возникает при программировании простых тайм-аутов.

Решение

Тип `Task` содержит статический метод `Delay`, который возвращает задачу, завершающуюся после истечения заданного времени.

В следующем примере определяется асинхронно завершаемая задача. При имитации асинхронной операции важно проверить синхронный успех

и асинхронный успех, а также асинхронную неудачу. Следующий пример возвращает задачу, используемую для случая асинхронного успеха:

```
async Task<T> DelayResult<T>(T result, TimeSpan delay)
{
  await Task.Delay(delay);
  return result;

}
```

Экспоненциальная задержка — стратегия увеличения задержек между повторными попытками. Используйте ее при работе с веб-службами, чтобы не перегружать сервер повторными попытками. Ниже приведен пример простой реализации экспоненциальной задержки:

```
async Task<string> DownloadStringWithRetries(HttpClient client, string uri)
{
  // Повторить попытку через 1 секунду, потом через 2 и через 4 секунды.
  TimeSpan nextDelay = TimeSpan.FromSeconds(1);
  for (int i = 0; i != 3; ++i)
  {
    try
    {
      return await client.GetStringAsync(uri);
    }
    catch
    {
    }

    await Task.Delay(nextDelay);
    nextDelay = nextDelay + nextDelay;
  }

  // Попробовать в последний раз и разрешить распространение ошибки.
  return await client.GetStringAsync(uri);
}
```

В реальном коде я бы рекомендовал применить более качественное решение (например, использующее библиотеку Polly NuGet); код, приведенный здесь, является всего лишь примером использования `Task.Delay`.

`Task.Delay` также можно использовать для организации простого тайм-аута. Обычно для реализации тайм-аута используется тип `CancellationTokenSource` (рецепт 10.3). Его можно упаковать в `Task.Delay` с неограниченной задержкой, чтобы предоставить задачу, которая отменяется по истечении заданного времени. Наконец, используйте задачу с таймером в сочетании с `Task.WhenAny` (рецепт 2.5) для реализации «мягкого» тайм-аута. Следующий пример возвращает `null`, если служба не вернет ответ в течение 3 секунд:

```
async Task<string> DownloadStringWithTimeout(HttpClient client, string uri)
{
  using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(3));
  Task<string> downloadTask = client.GetStringAsync(uri);
  Task timeoutTask = Task.Delay(Timeout.InfiniteTimeSpan, cts.Token);

  Task completedTask = await Task.WhenAny(downloadTask, timeoutTask);
  if (completedTask == timeoutTask)
    return null;
  return await downloadTask;
}
```

И хотя `Task.Delay` можно использовать для реализации «мягкого» тайм-аута, у такого подхода есть свои ограничения. Если в операции происходит тайм-аут, она не отменяется; в предыдущем примере задача загрузки продолжит прием данных и загрузит весь ответ перед тем, как потерять его. Рекомендуемое решение основано на использовании *маркера отмены* (cancellation token) в качестве тайм-аута и передаче его операции напрямую (`GetStringAsync` в последнем примере). При этом операция может оказаться неотменяемой; в этом случае `Task.Delay` может использоваться другим кодом для имитации действий, выполняемых по тайм-ауту.

Пояснение

`Task.Delay` неплохо подходит для модульного тестирования асинхронного кода или реализации логики повторных попыток. Но если нужно реализовать тайм-аут, лучшим кандидатом будет `CancellationToken`.

Дополнительная информация

В рецепте 2.5 рассматривается использование `Task.WhenAny` для определения того, какая задача завершится первой.

В рецепте 10.3 рассматривается использование `CancellationToken` в качестве тайм-аута.

2.2. Возвращение завершенных задач

Задача

Требуется реализовать синхронный метод с асинхронной сигнатурой. Например, такая ситуация может возникнуть, если вы наследуете от асинхронного интерфейса или базового класса, но хотите реализовать его синхронно. Этот прием особенно полезен при модульном тестировании асинхронного кода, когда нужна простая заглушка или имитированная реализация для асинхронного интерфейса.

Решение

Можно использовать `Task.FromResult` для создания и возвращения нового объекта `Task<T>`, уже завершенного с заданным значением:

```
interface IMyAsyncInterface
{
  Task<int> GetValueAsync();
}

class MySynchronousImplementation : IMyAsyncInterface
{
  public Task<int> GetValueAsync()
  {
    return Task.FromResult(13);
  }
}
```

Для методов, не имеющих возвращаемого значения, можно использовать `Task.CompletedTask` — кэшированный объект успешно завершенной задачи `Task`:

```
interface IMyAsyncInterface
{
  Task DoSomethingAsync();
```

```
}

class MySynchronousImplementation : IMyAsyncInterface
{
  public Task DoSomethingAsync()
  {
    return Task.CompletedTask;
  }
}
```

`Task.FromResult` предоставляет завершенные задачи только для успешных результатов. Если потребуется задача с другим типом результата (например, задача, завершенная с `NotImplementedException`), вы можете использовать `Task.FromException`:

```
Task<T> NotImplementedAsync<T>()
{
  return Task.FromException<T>(new NotImplementedException());
}
```

Аналогично существует метод `Task.FromCanceled` для создания задач, уже отмененных из заданного маркера `CancellationToken`:

```
Task<int> GetValueAsync(CancellationToken cancellationToken)
{
  if (cancellationToken.IsCancellationRequested)
    return Task.FromCanceled<int>(cancellationToken);
  return Task.FromResult(13);
}
```

Если в синхронной реализации может произойти отказ, перехватывайте исключения и используйте `Task.FromException` для их возвращения:

```
interface IMyAsyncInterface
{
  Task DoSomethingAsync();
}

class MySynchronousImplementation : IMyAsyncInterface
{
  public Task DoSomethingAsync()
  {
    try
```

```
    {
      DoSomethingSynchronously();
      return Task.CompletedTask;
    }
    catch (Exception ex)
    {
      return Task.FromException(ex);
    }
  }
}
```

Пояснение

Если вы реализуете асинхронный интерфейс синхронным кодом, избегайте любых форм блокировки. Избегайте блокирования с последующим возвращением завершенной задачи в асинхронном методе, если метод может быть реализован асинхронно. В качестве контрпримера рассмотрим средства чтения текста из `Console` в .NET BCL. `Console.In.ReadLineAsync` блокирует вызывающий поток, пока не будет прочитана строка, после чего возвращает завершенную задачу. Такое поведение не интуитивно, оно преподносило сюрпризы многим разработчикам. Если асинхронный метод блокируется, он не позволяет вызывающему потоку запускать другие задачи, что противоречит идее конкурентности и может привести к взаимоблокировке.

Если вы регулярно используете `Task.FromResult` с одним значением, подумайте о кэшировании задачи. Например, если вы один раз создали `Task<int>` с нулевым результатом, избегайте создания других экземпляров, которые должны будут уничтожаться в ходе уборки мусора:

```
private static readonly Task<int> zeroTask = Task.FromResult(0);
Task<int> GetValueAsync()
{
  return zeroTask;
}
```

На логическом уровне `Task.FromResult`, `Task.FromException` и `Task.FromCanceled` являются вспомогательными методами и сокращенными формами обобщенного типа `TaskCompletionSource<T>`. `TaskCompletionSource<T>` представляет собой низкоуровневый тип, полезный для взаимодействия с другими формами асинхронного кода. В общем случае следует применять

сокращенную форму `Task.FromResult` и родственные формы, если хотите вернуть уже завершенную задачу. Используйте `TaskCompletionSource<T>` для возвращения задачи, которая завершается в некоторый момент будущего.

Дополнительная информация

В рецепте 7.1 рассматривается модульное тестирование асинхронных методов.

В рецепте 11.1 рассматривается наследование `async`-методов.

В рецепте 8.3 показано, как использовать `TaskCompletionSource<T>` для обобщенного взаимодействия с другим асинхронным кодом.

2.3. Передача информации о ходе выполнения операции

Задача

Требуется отреагировать на прогресс выполнения операции.

Решение

Используйте типы `IProgress<T>` и `Progress<T>`. Ваш `async`-метод должен получать аргумент `IProgress<T>`; здесь `T` — тип прогресса, о котором вы хотите сообщать:

```
async Task MyMethodAsync(IProgress<double> progress = null)
{
  bool done = false;
  double percentComplete = 0;
  while (!done)
  {
    ...
    progress?.Report(percentComplete);
  }
}
```

Пример использования в вызывающем коде:

```
async Task CallMyMethodAsync()
{
  var progress = new Progress<double>();
  progress.ProgressChanged += (sender, args) =>
  {
    ...
  };
  await MyMethodAsync(progress);
}
```

Пояснение

По действующим соглашениям параметр `IProgress<T>` может быть равен `null`, если вызывающей стороне не нужны уведомления о прогрессе; включите соответствующую проверку в свой `async`-метод.

Помните, что метод `IProgress<T>.Report` обычно является асинхронным. Это означает, что `MyMethodAsync` может продолжить выполнение перед сообщением о прогрессе.

По этой причине лучше определить `T` как *неизменяемый тип* (или по крайней мере тип-значение). Если `T` является изменяемым ссылочным типом, то вам придется самостоятельно создавать отдельную копию при каждом вызове `IProgress<T>.Report`.

`Progress<T>` сохраняет текущий контекст при создании и активизирует свой обратный вызов в этом контексте. Это означает, что если `Progress<T>` конструируется в UI-потоке, то вы сможете обновить пользовательский интерфейс из его обратного вызова, даже если асинхронный метод вызывает `Report` из фонового потока.

Если метод поддерживает уведомления о прогрессе, он также должен приложить максимальные усилия для поддержки отмены.

`IProgress<T>` не ограничивается одним асинхронным кодом; как прогресс, так и отмена также могут (и должны) использоваться в долгосрочном синхронном коде.

Дополнительная информация

В рецепте 10.4 рассматривается поддержка отмены в асинхронных методах.

2.4. Ожидание завершения группы задач

Задача

У вас есть несколько задач, и нужно подождать, пока они все закончатся.

Решение

Фреймворк предоставляет для этой цели метод `Task.WhenAll`. Метод получает несколько задач и возвращает задачу, которая завершается при завершении всех указанных задач.

```
Task task1 = Task.Delay(TimeSpan.FromSeconds(1));
Task task2 = Task.Delay(TimeSpan.FromSeconds(2));
Task task3 = Task.Delay(TimeSpan.FromSeconds(1));

await Task.WhenAll(task1, task2, task3);
```

Если все задачи имеют одинаковый тип результата и все завершаются успешно, то задача `Task.WhenAll` возвращает массив, содержащий результаты всех задач:

```
Task<int> task1 = Task.FromResult(3);
Task<int> task2 = Task.FromResult(5);
Task<int> task3 = Task.FromResult(7);

int[] results = await Task.WhenAll(task1, task2, task3);

// "results" содержит { 3, 5, 7 }
```

Есть перегруженная версия `Task.WhenAll`, которая получает `IEnumerable` с задачами; тем не менее я не рекомендую ее использовать. Каждый раз, когда я смешиваю асинхронный код с LINQ, на мой взгляд, код получается более понятным, когда я явно «материализую» последовательность (т. е. обрабатываю последовательность с созданием коллекции):

```
async Task<string> DownloadAllAsync(HttpClient client,
    IEnumerable<string> urls)
{
  // Определить действие, выполняемое для каждого URL.
  var downloads = urls.Select(url => client.GetStringAsync(url));
```

```
  // Обратите внимание: задачи еще не запущены,
  //  потому что последовательность не была обработана.

  // Запустить загрузку для всех URL одновременно.
  Task<string>[] downloadTasks = downloads.ToArray();
  // Все задачи запущены.

  // Асинхронно ожидать завершения всех загрузок.
  string[] htmlPages = await Task.WhenAll(downloadTasks);

  return string.Concat(htmlPages);
}
```

Пояснение

Если какие-либо задачи выдают исключения, то `Task.WhenAll` сообщает об отказе своей возвращенной задачи с этим исключением. Если сразу несколько задач выдают исключение, то все эти исключения помещаются в задачу `Task`, возвращаемую `Task.WhenAll`. Тем не менее при ожидании этой задачи будет выдано только одно из них. Если нужно каждое конкретное исключение, проверьте свойство `Exception` задачи `Task`, возвращаемой `Task.WhenAll`:

```
async Task ThrowNotImplementedExceptionAsync()
{
  throw new NotImplementedException();
}

async Task ThrowInvalidOperationExceptionAsync()
{
  throw new InvalidOperationException();
}

async Task ObserveOneExceptionAsync()
{
  var task1 = ThrowNotImplementedExceptionAsync();
  var task2 = ThrowInvalidOperationExceptionAsync();

  try
  {
    await Task.WhenAll(task1, task2);
  }
```

```
  catch (Exception ex)
  {
   // "ex" - либо NotImplementedException, либо InvalidOperationException.
    ...
  }
}

async Task ObserveAllExceptionsAsync()
{
  var task1 = ThrowNotImplementedExceptionAsync();
  var task2 = ThrowInvalidOperationExceptionAsync();

  Task allTasks = Task.WhenAll(task1, task2);
  try
  {
    await allTasks;
  }
  catch
  {
    AggregateException allExceptions = allTasks.Exception;
    ...
  }
}
```

Как правило, я не отслеживаю все исключения при использовании `Task.WhenAll`. Обычно достаточно отреагировать только на первую выданную ошибку, а не на все.

Обратите внимание: в предыдущем примере методы `ThrowNotImplementedExceptionAsync` и `ThrowInvalidOperationExceptionAsync` не выдают свои исключения напрямую; они используют ключевое слово `async`, поэтому исключения перехватываются и помещаются в задачу, которая возвращается нормальным образом. Это нормальное поведение методов, которые возвращают типы, допускающие ожидание.

Дополнительная информация

В рецепте 2.5 рассматривается ожидание завершения любой задачи из группы задач.

В рецепте 2.6 рассматривается ожидание завершения коллекции задач с выполнением действий при завершении каждой задачи.

В рецепте 2.8 рассматривается обработка исключений для методов `async Task`.

2.5. Ожидание завершения любой задачи

Задача

Есть несколько задач и требуется отреагировать на завершение любой задачи из группы. Задача чаще всего встречается при выполнении нескольких независимых попыток выполнения операции в структуре «первому достается все». Например, можно запросить биржевые котировки у нескольких веб-служб одновременно, но интересует вас только первый ответ.

Решение

Используйте метод `Task.WhenAny`. Метод `Task.WhenAny` получает последовательность задач и возвращает задачу, которая завершается при завершении любой из задач последовательности. Результатом возвращенной задачи является завершенная задача. Не огорчайтесь, если это прозвучало непонятно; некоторые вещи трудно объяснить, но легко понять на примере кода:

```
// Возвращает длину данных первого ответившего URL-адреса.
async Task<int> FirstRespondingUrlAsync(HttpClient client,
    string urlA, string urlB)
{
  // Запустить обе загрузки параллельно.
  Task<byte[]> downloadTaskA = client.GetByteArrayAsync(urlA);
  Task<byte[]> downloadTaskB = client.GetByteArrayAsync(urlB);

  // Ожидать завершения любой из этих задач.
  Task<byte[]> completedTask =
      await Task.WhenAny(downloadTaskA, downloadTaskB);

  // Вернуть длину данных, загруженных по этому URL-адресу.
  byte[] data = await completedTask;
  return data.Length;
}
```

Пояснение

Задача, возвращенная `Task.WhenAny`, никогда не завершается в состоянии отказа или отмены. Эта «внешняя» задача всегда завершается успешно, а ее результирующее значение представляет собой первую завершенную задачу `Task` («внутреннюю»). Если внутренняя задача завершилась с исключением, то это исключение не распространяется на внешнюю задачу (возвращенную `Task.WhenAny`). Обычно ваш код ожидает внутреннюю задачу посредством `await`, чтобы обеспечить отслеживание всех исключений.

Когда первая задача завершается, подумайте, не отменить ли остальные задачи. Если другие задачи не отменяются, но к ним не применяется `await`, они просто теряются. Потерянные задачи отрабатывают до завершения, но их результаты игнорируются. Все исключения от потерянных задач также будут проигнорированы. Если эти задачи не будут отменены, они продолжат работать и неэффективно расходовать ресурсы (подключения HTTP, подключения к базе данных, таймеры и т. д.).

Вы можете использовать `Task.WhenAny` для реализации тайм-аута (например, при использовании `Task.Delay` как одной из задач), но так поступать не рекомендуется. Более естественно выражать тайм-ауты отменой, и у отмены есть дополнительное преимущество: она позволяет действительно *отменить* операцию(-и) в случае тайм-аута.

Другой антипаттерн `Task.WhenAny` — обработка задач по мере их завершения. Сначала может показаться разумным вести список задач и удалять каждую задачу из списка при завершении. Проблема в том, что такое решение выполняется за время $O(N^2)$, хотя существует алгоритм со временем O(N). Правильный алгоритм O(N) рассматривается в рецепте 2.6.

Дополнительная информация

В рецепте 2.4 рассматривается асинхронное ожидание завершения всех задач из набора.

В рецепте 2.6 рассматривается ожидание завершения группы задач с выполнением действий при завершении каждой задачи.

В рецепте 10.3 рассматривается использование маркера отмены для реализации тайм-аута.

2.6. Обработка задач при завершении

Задача

Имеется коллекция задач, которые будут использоваться с `await`; требуется организовать обработку каждой задачи после ее завершения. При этом обработка каждой задачи должна происходить сразу же после завершения, без ожидания других задач.

Следующий пример запускает три отложенные задачи, а затем ожидает каждую из них:

```
async Task<int> DelayAndReturnAsync(int value)
{
  await Task.Delay(TimeSpan.FromSeconds(value));
  return value;
}

// В текущей версии метод выводит "2", "3" и "1".
// При этом метод должен выводить "1", "2" и "3".
async Task ProcessTasksAsync()
{
  // Создать последовательность задач.
  Task<int> taskA = DelayAndReturnAsync(2);
  Task<int> taskB = DelayAndReturnAsync(3);
  Task<int> taskC = DelayAndReturnAsync(1);
  Task<int>[] tasks = new[] { taskA, taskB, taskC };

  // Ожидать каждую задачу по порядку.
  foreach (Task<int> task in tasks)
  {
    var result = await task;
    Trace.WriteLine(result);
  }
}
```

В этой версии код ожидает каждую задачу в порядке последовательности, хотя третья задача в последовательности завершается первой. Код должен выполнять обработку (например, `Trace.WriteLine`) при завершении каждой задачи, не дожидаясь завершения других.

Решение

Есть несколько разных подходов к решению этой задачи. Метод, который описан первым в этом рецепте, является рекомендуемым; другой описан в разделе «Пояснение».

Простейшее решение заключается в рефакторинге кода и введении высокоуровневого `async`-метода, который обеспечивает ожидание задачи и обработку ее результата. Вынесение обработки в отдельный метод существенно упрощает код:

```
async Task<int> DelayAndReturnAsync(int value)
{
  await Task.Delay(TimeSpan.FromSeconds(value));
  return value;
}

async Task AwaitAndProcessAsync(Task<int> task)
{
  int result = await task;
  Trace.WriteLine(result);
}

// Этот метод теперь выводит "1", "2" и "3".
async Task ProcessTasksAsync()
{
  // Создать последовательность задач.
  Task<int> taskA = DelayAndReturnAsync(2);
  Task<int> taskB = DelayAndReturnAsync(3);
  Task<int> taskC = DelayAndReturnAsync(1);
  Task<int>[] tasks = new[] { taskA, taskB, taskC };

  IEnumerable<Task> taskQuery =
      from t in tasks select AwaitAndProcessAsync(t);
  Task[] processingTasks = taskQuery.ToArray();

  // Ожидать завершения всей обработки
  await Task.WhenAll(processingTasks);
}
```

С другой стороны, этот код можно записать в следующем виде:

```
async Task<int> DelayAndReturnAsync(int value)
{
  await Task.Delay(TimeSpan.FromSeconds(value));
  return value;
}

// Этот метод теперь выводит "1", "2" и "3".
async Task ProcessTasksAsync()
{
  // Создать последовательность задач.
  Task<int> taskA = DelayAndReturnAsync(2);
  Task<int> taskB = DelayAndReturnAsync(3);
  Task<int> taskC = DelayAndReturnAsync(1);
  Task<int>[] tasks = new[] { taskA, taskB, taskC };
  Task[] processingTasks = tasks.Select(async t =>
  {
    var result = await t;
    Trace.WriteLine(result);
  }).ToArray();

  // Ожидать завершения всей обработки.
  await Task.WhenAll(processingTasks);
}
```

Показанный рефакторинг — самое элегантное и портируемое решение проблемы. Обратите внимание: оно несколько отличается от исходного кода. В этом решении обработка задач выполняется конкурентно, тогда как в исходном коде задачи будут обрабатываться по одной. Обычно это не создает затруднений, но если такой способ обработки в вашей ситуации недопустим, рассмотрите возможность использования блокировок (рецепт 12.2) или следующего альтернативного решения.

Пояснение

Если рефакторинг не дает приемлемого решения, есть альтернатива. Стивен Тауб (Stephen Toub) и Джон Скит (Jon Skeet) разработали методы расширения, возвращающие массив задач, которые завершаются по порядку. Решение Стивена Тауба (Stephen Toub) доступно в блоге Parallel Programming with .NET (https://devblogs.microsoft.com/pfxteam/processing-

tasks-as-they-complete/), а решение Джона Скита (Jon Skeet) — в его блоге, посвященном программированию (https://codeblog.jonskeet.uk/2012/01/16/eduasync-part-19-ordering-by-completion-ahead-of-time/).

Метод расширения `OrderByCompletion` также доступен в библиотеке с открытым кодом `AsyncEx` (NuGet-пакет `Nito.AsyncEx`).

С таким методом расширения, как `OrderByCompletion`, изменения в исходной версии кода сводятся до минимума:

```
async Task<int> DelayAndReturnAsync(int value)
{
  await Task.Delay(TimeSpan.FromSeconds(value));
  return value;
}

// Этот метод теперь выводит "1", "2" и "3".
async Task UseOrderByCompletionAsync()
{
  // Создать последовательность задач.
  Task<int> taskA = DelayAndReturnAsync(2);
  Task<int> taskB = DelayAndReturnAsync(3);
  Task<int> taskC = DelayAndReturnAsync(1);
  Task<int>[] tasks = new[] { taskA, taskB, taskC };

  // Ожидать каждой задачи по мере выполнения.
  foreach (Task<int> task in tasks.OrderByCompletion())
  {
    int result = await task;
    Trace.WriteLine(result);
  }
}
```

Дополнительная информация

В рецепте 2.4 рассматривается асинхронное ожидание завершения последовательности задач.

2.7. Обход контекста при продолжении

Задача

Когда `async`-метод возобновляет работу после `await`, по умолчанию он продолжает выполнение в том же контексте. Это может создать проблемы с быстродействием, если контекстом был UI-контекст, а в UI-контексте возобновляет работу большое количество `async`-методов.

Решение

Чтобы избежать возобновления в контексте, используйте `await` для результата `ConfigureAwait` и передайте `false` в параметре `continueOnCapturedContext`:

```
async Task ResumeOnContextAsync()
{
  await Task.Delay(TimeSpan.FromSeconds(1));

  // Этот метод возобновляется в том же контексте.
}

async Task ResumeWithoutContextAsync()
{
  await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);

  // Этот метод теряет свой контекст при возобновлении.
}
```

Пояснение

Наличие слишком большого количества продолжений, выполняемых в UI-потоке, может создать проблемы с быстродействием. Такие проблемы трудно диагностировать, потому что это не единственный метод, замедляющий систему. Скорее, с увеличением сложности приложения пользовательский интерфейс начинает «кровоточить из-за тысяч мелких порезов».

Остается понять: сколько продолжений в UI-потоке превышает допустимый порог? Простого и однозначного ответа на этот вопрос нет, но Люциан Вищик из Microsoft огласил рекомендацию, которая использо-

валась командой Universal Windows: около сотни в секунду — нормально, но около тысячи в секунду — уже слишком много.

Лучше обойти эту проблему с самого начала. Для каждого написанного вами `async`-метода, если он не должен возобновляться в исходном контексте, используйте `ConfigureAwait`. Никаких неудобств это не создаст.

Также стоит учитывать контекст при написании `async`-кода. Обычно `async`-метод должен либо требовать определенного контекста (работа с UI-элементами или запросами/ответами ASP.NET), либо быть свободным от контекста (выполняя фоновые операции). Если у вас имеется `async`-метод с частями, требующими контекста, и частями, свободными от контекста, рассмотрите возможность его разбиения на два (или более) `async`-метода. Такой подход помогает организовать код по уровням.

Дополнительная информация

В главе 1 рассматривается введение в асинхронное программирование.

2.8. Обработка исключений из методов async Task

Задача

Обработка исключений является важнейшей частью любой программной архитектуры. Спроектировать код для успешного результата несложно, но структура кода не может считаться правильной, если в ней не обрабатываются потенциальные ошибки. К счастью, обработка исключений из методов `async Task` реализуется прямолинейно.

Решение

Исключения можно перехватывать простой конструкцией `try/catch`, как вы бы сделали для синхронного кода:

```
async Task ThrowExceptionAsync()
{
  await Task.Delay(TimeSpan.FromSeconds(1));
  throw new InvalidOperationException("Test");
}
```

```
async Task TestAsync()
{
  try
  {
    await ThrowExceptionAsync();
  }
  catch (InvalidOperationException)
  {
  }
}
```

Исключения, выданные из методов `async Task`, помещаются в возвращаемый объект `Task`. Они выдаются только при использовании `await` с возвращаемым объектом `Task`:

```
async Task ThrowExceptionAsync()
{
  await Task.Delay(TimeSpan.FromSeconds(1));
  throw new InvalidOperationException("Test");
}

async Task TestAsync()
{
  // Исключение выдается методом и помещается в задачу.
  Task task = ThrowExceptionAsync();
  try
  {
    // Здесь исключение будет выдано повторно.
    await task;
  }
  catch (InvalidOperationException)
  {
    // Здесь исключение правильно перехватывается.
  }
}
```

Пояснение

Когда в методе `async Task` выдается исключение, это исключение сохраняется и включается в возвращаемый объект `Task`. Так как методы

`async void` не имеют объекта `Task` для размещения исключения, для них используется другое поведение; перехват исключений из методов `async void` рассматривается в рецепте 2.9.

При использовании `await` с задачей `Task`, в которой произошел отказ, первое исключение этой задачи выдается повторно. Если вы знакомы с проблемами повторной выдачи исключений, могут возникнуть вопросы о трассировках стека. Не сомневайтесь: при повторной выдаче исключения исходная трассировка стека будет правильно сохранена.

Такая конфигурация может показаться излишне запутанной, но вся эта сложность работает на то, чтобы в простом сценарии использовался простой код. В большинстве случаев код должен распространять исключения из вызываемых асинхронных методов; все, что нужно сделать, — использовать `await` с задачей, возвращенной из асинхронного метода, и это исключение будет распространяться естественным образом.

Возможны ситуации (например, с `Task.WhenAll`), в которых `Task` может содержать несколько исключений, а `await` повторно выдает только первое из них. За примером обработки всех исключений обращайтесь к рецепту 2.4.

Дополнительная информация

В рецепте 2.4 рассматривается ожидание для нескольких задач.

В рецепте 2.9 рассматриваются методы перехвата исключений из методов `async void`.

В рецепте 7.2 рассматриваются исключения модульного тестирования, выданные из методов `async Task`.

2.9. Обработка исключений из методов async void

Задача

Имеется метод `async void`. Требуется обработать исключения, распространенные из этого метода.

Решение

Хорошего решения не существует. Если возможно, измените метод так, чтобы он возвращал `Task` вместо `void`. В некоторых ситуациях это невозможно; например, представьте, что нужно провести модульное тестирование реализации `ICommand` (которая должна возвращать `void`). В этом случае необходимо предоставить перегруженную версию вашего метода `Execute`, которая возвращает `Task`:

```
sealed class MyAsyncCommand : ICommand
{
  async void ICommand.Execute(object parameter)
  {
    await Execute(parameter);
  }

  public async Task Execute(object parameter)
  {
    ... // Здесь размещается асинхронная реализация команды.
  }

  ... // Другие составляющие (CanExecute и т. д.)
}
```

Лучше избегать распространения исключений из методов `async void`. Если же вы должны использовать метод `async void`, рассмотрите возможность упаковки всего кода в блок `try` и прямой обработки исключений.

Существует и другой возможный способ обработки исключений из методов `async void`. Когда метод `async void` распространяет исключение, это исключение выдается в контексте `SynchronizationContext`, активном на момент начала выполнения метода `async void`. Если среда выполнения предоставляет `SynchronizationContext`, то обычно она предоставляет механизм обработки этих высокоуровневых исключений на глобальном уровне. Например, WPF предоставляет `Application.DispatcherUnhandledException`, Universal Windows — `Application.UnhandledException`, а ASP.NET — `UseExceptionHandler`.

Также возможно обрабатывать исключения из методов `async void` посредством управления `SynchronizationContext`. Написать собственный вариант `SynchronizationContext` непросто, но можно воспользоваться типом `AsyncContext` из бесплатной вспомогательной NuGet-библиотеки

Nito.AsyncEx. Тип AsyncContext особенно полезен для приложений, не имеющих встроенного объекта SynchronizationContext (например, консольных приложений и служб Win32). В следующем примере AsyncContext используется для запуска и обработки исключений из метода async void:

```
static class Program
{
  static void Main(string[] args)
  {
    try
    {
      AsyncContext.Run(() => MainAsync(args));
    }
    catch (Exception ex)
    {
      Console.Error.WriteLine(ex);
    }
  }

  // ПЛОХОЙ КОД!!!
  // В реальных приложениях не используйте метод async void
  // без крайней необходимости.
  static async void MainAsync(string[] args)
  {
    ...
  }
}
```

Пояснение

Одна из причин, по которым стоит отдать предпочтение async Task перед методами async void, заключается в том, что методы, возвращающие Task, проще тестировать. Как минимум перегрузка методов, возвращающих void, методами, возвращающими Task, предоставит поверхность API, удобную для тестирования.

Если нужно предоставить ваш собственный тип SynchronizationContext (например, AsyncContext), будьте внимательны и не устанавливайте этот контекст SynchronizationContext в потоках, которые вам не принадлежат. Как правило, этот тип не должен размещаться в потоках, в которых он уже есть (например, UI-потоках или классических потоках запросов ASP.

NET); также не стоит размещать SynchronizationContext в потоках из пула потоков. Главный поток консольного приложения принадлежит вам, как и все потоки, которые вы самостоятельно создаете вручную.

Тип AsyncContext находится в пакете Nito.AsyncEx.

Дополнительная информация

В рецепте 2.8 рассматривается обработка исключений с методами async Task.

В рецепте 7.3 рассматривается модульное тестирование методов async void.

2.10. Создание ValueTask

Задача

Требуется создать метод, возвращающий ValueTask<T>.

Решение

ValueTask<T> используется как возвращаемый тип в ситуациях, в которых обычно может быть возвращен синхронный результат, а асинхронное поведение встречается реже. В общем случае в коде приложения следует использовать в качестве возвращаемого типа Task<T>, а не ValueTask<T>. Рассматривать использование ValueTask<T> в качестве возвращаемого типа следует только после профилирования, которое показывает, что это приведет к повышению быстродействия. Впрочем, возможны ситуации, в которых требуется реализовать метод, возвращающий ValueTask<T>. Одна из таких ситуаций встречается при использовании интерфейса IAsyncDisposable, метод DisposeAsync которого возвращает ValueTask. За более подробным пояснением асинхронного освобождения ресурсов обращайтесь к рецепту 11.6.

Простейший способ реализации метода, возвращающего `ValueTask<T>`, основан на использовании `async` и `await`, как и обычный `async`-метод:

```
public async ValueTask<int> MethodAsync()
{
  await Task.Delay(100); // Асинхронная работа.
  return 13;
}
```

Нередко метод, возвращающий `ValueTask<T>`, способен немедленно вернуть значение; в таких случаях можно применить оптимизацию для этого сценария с использованием конструктора `ValueTask<T>`, а затем передавать управление медленному асинхронному методу только при необходимости:

```
public ValueTask<int> MethodAsync()
{
  if (CanBehaveSynchronously)
    return new ValueTask<int>(13);
  return new ValueTask<int>(SlowMethodAsync());
}

private Task<int> SlowMethodAsync();
```

Аналогичный подход возможен для `ValueTask` без параметризации. Здесь конструктор по умолчанию `ValueTask` используется для возвращения успешно завершенного объекта `ValueTask`. В следующем примере показана реализация `IAsyncDisposable`, которая выполняет свою логику асинхронного освобождения однократно; при будущих вызовах метод `DisposeAsync` завершается успешно и синхронно:

```
private Func<Task> _disposeLogic;

public ValueTask DisposeAsync()
{
  if (_disposeLogic == null)
    return default;

  // Примечание: этот простой пример не является потокобезопасным;
  //  если сразу несколько потоков вызовут DisposeAsync,
  //  логика может быть выполнена более одного раза.
```

```
  Func<Task> logic = _disposeLogic;
  _disposeLogic = null;
  return new ValueTask(logic());
}
```

Пояснение

Большинство методов должно возвращать `Task<T>`, поскольку при потреблении `Task<T>` возникает меньше скрытых ловушек, чем при потреблении `ValueTask<T>`. Подробности см. в рецепте 2.11.

Чаще при реализации интерфейсов, использующих `ValueTask` или `ValueTask<T>`, можно просто применять `async` и `await`. Более сложные реализации нужны тогда, когда вы собираетесь использовать `ValueTask<T>` самостоятельно.

Решения, рассмотренные в этом рецепте, соответствуют более простым и распространенным подходам к созданию экземпляров `ValueTask<T>` и `ValueTask`. Есть другой способ, более подходящий для сложных сценариев, в которых выделение ресурсов должно быть сведено к абсолютному минимуму. В более сложном решении вы кэшируете или помещаете в пул реализацию `IValueTaskSource<T>` и повторно используете ее для многих вызовов асинхронных методов. За вводным описанием сложного сценария обращайтесь к документации Microsoft по типу `ManualResetValueTaskSourceCore<T>`.

Дополнительная информация

В рецепте 2.11 рассматриваются ограничения при потреблении типов `ValueTask<T>` и `ValueTask`.

В рецепте 11.6 рассматривается асинхронное освобождение.

2.11. Потребление ValueTask

Задача

Требуется организовать потребление `ValueTask<T>`.

Решение

Самый простой и прямолинейный способ потребления `ValueTask<T>` или `ValueTask` основан на `await`. В большинстве случаев это все, что вам необходимо сделать:

```
ValueTask<int> MethodAsync();
async Task ConsumingMethodAsync()
{
  int value = await MethodAsync();
}
```

Также можно выполнить `await` после выполнения конкурентной операции, как в случае с `Task<T>`:

```
ValueTask<int> MethodAsync();

async Task ConsumingMethodAsync()
{
  ValueTask<int> valueTask = MethodAsync();
  ... // Другая параллельная работа.
  int value = await valueTask;
}
```

Оба варианта подходят, потому что `ValueTask` ожидается только один раз. Это одно из ограничений `ValueTask`.

`ValueTask` или `ValueTask<T>` может ожидаться только один раз.

Чтобы сделать что-то более сложное, преобразуйте `ValueTask<T>` в `Task<T>` вызовом `AsTask`:

```
ValueTask<int> MethodAsync();
async Task ConsumingMethodAsync()
{
  Task<int> task = MethodAsync().AsTask();
  ... // Другая параллельная работа.
```

```
  int value = await task;
  int anotherValue = await task;
}
```

Многократное ожидание `Task<T>` абсолютно безопасно. Также возможны другие операции — например, асинхронное ожидание завершения нескольких операций (см. рецепт 2.4):

```
ValueTask<int> MethodAsync();

async Task ConsumingMethodAsync()
{
  Task<int> task1 = MethodAsync().AsTask();
  Task<int> task2 = MethodAsync().AsTask();
  int[] results = await Task.WhenAll(task1, task2);
}
```

Тем не менее для каждого `ValueTask<T>` можно вызвать `AsTask` только один раз. Самое распространенное решение — немедленно преобразовать его в `Task<T>`, а в дальнейшем игнорировать `ValueTask<T>`. Также замечу, что вы не можете одновременно использовать `await` и вызвать `AsTask` для одного `ValueTask<T>`.

В большинстве программ следует либо немедленно выполнить `await` для `ValueTask<T>`, либо преобразовать значение в `Task<T>`.

Пояснение

Другие свойства `ValueTask<T>` предназначены для нетривиального использования. Обычно они работают не так, как другие известные свойства; в частности, для `ValueTask<T>.Result` действуют более жесткие ограничения, чем для `Task<T>.Result`. Код, который синхронно получает результат от `ValueTask<T>`, может вызвать `ValueTask<T>.Result` или `ValueTask<T>`, `GetAwaiter().GetResult()`, но эти компоненты не должны вызываться до завершения `ValueTask<T>`. Синхронная загрузка результата из `Task<T>` блокирует вызывающий поток до завершения задачи; `ValueTask<T>` таких гарантий не дает.

Синхронное получение результатов от `ValueTask` или `ValueTask<T>` может быть выполнено только один раз, после завершения `ValueTask`, и это значение `ValueTask` уже не может использоваться для ожидания или преобразования в задачу.

Рискуя повториться, все же скажу: когда ваш код вызывает метод, возвращающий `ValueTask` или `ValueTask<T>`, он должен либо немедленно выполнить `await` для этого `ValueTask`, либо немедленно вызвать `AsTask` для преобразования в `Task`. Возможно, эта простая рекомендация не исчерпывает все нетривиальные сценарии, но большинству приложений этого будет вполне достаточно.

Дополнительная информация

В рецепте 2.10 рассматривается возвращение значений `ValueTask<T>` и `ValueTask` из ваших методов.

В рецептах 2.4 и 2.5 рассматривается одновременное ожидание нескольких задач.

ГЛАВА 3

Асинхронные потоки

Асинхронные потоки — механизм асинхронного получения нескольких элементов данных. Они строятся на основе асинхронных перечисляемых объектов (`IAsyncEnumerable<T>`). *Асинхронный перечисляемый объект* представляет собой асинхронную версию перечисляемого объекта (enumerable); т. е. он может производить элементы по требованию для потребителя, и каждый элемент может быть произведен асинхронно.

На мой взгляд, полезно сравнить асинхронные потоки с другими, более знакомыми типами и проанализировать различия. Это помогает лучше запомнить, в каких случаях следует использовать асинхронные потоки, а в каких случаях более уместными будут другие типы.

Асинхронные потоки и Task<T>

Стандартного асинхронного механизма с `Task<T>` достаточно только для асинхронной обработки одного значения данных. После того как `Task<T>` завершится, все кончено; одна задача `Task<T>` не может предоставить своим потребителям более одного значения `T`. Даже если `T` представляет собой коллекцию, значение может быть предоставлено только один раз. За информацией об использовании `async` с `Task<T>` обращайтесь к разделу «Введение в асинхронное программирование» главы 1, а в главе 2 более подробно рассматривается использование `async` с `Task<T>`. При сравнении `Task<T>` с асинхронными потоками мы видим, что асинхронные потоки более похожи на перечисляемые объекты. А именно: `IAsyncEnumerator<T>` может предоставить любое количество значений `T`, по одному за раз. Как и `IEnumerator<T>`, `IAsyncEnumerator<T>` может иметь неограниченную длину.

Асинхронные потоки и IEnumerable<T>

`IAsyncEnumerable<T>`, как следует из названия, является аналогом `IEnumerable<T>`. Пожалуй, это неудивительно; оба позволяют потребителям получать элементы по одному за раз. В имени же скрыто принципиальное различие: один интерфейс асинхронен, а другой нет.

Когда ваш код перебирает `IEnumerable<T>`, то блокирует каждый элемент из перечисляемого объекта. Если `IEnumerable<T>` представляет некоторую операцию, связанную с вводом/выводом (такую, как запрос к базе данных или вызов API), то код-потребитель в конечном итоге блокируется по вводу/выводу — ситуация отнюдь не идеальная. `IAsyncEnumerable<T>` работает точно так же, как и `IEnumerable<T>`, не считая того, что он асинхронно получает каждый следующий элемент.

Асинхронные потоки и Task<IEnumerable<T>>

Ничто не мешает асинхронно вернуть коллекцию, которая содержит более одного элемента; типичный пример — `Task<List<T>>`. Тем не менее `async`-методы, которые возвращают `List<T>`, могут выполнить только одну команду `return`; коллекция должна быть заполнена до возвращения. Даже методы, возвращающие `Task<IEnumerable<T>>`, могут асинхронно вернуть перечисляемый объект, но тогда этот перечисляемый объект обрабатывается синхронно. Представьте, что LINQ-to-Entities содержит метод LINQ `ToListAsync`, возвращающий `Task<List<T>>`. Когда этот метод выполняется провайдером LINQ, он должен взаимодействовать с базой данных и получить все подходящие ответы до того, как он завершит заполнение списка и вернет его.

Принципиальное ограничение типа `Task<IEnumerable<T>>` заключается в том, что он не может возвращать элементы по мере получения; если возвращается коллекция, он должен загрузить все свои элементы в память, заполнить коллекцию, а затем вернуть всю коллекцию сразу. Даже если возвращается запрос LINQ, он может асинхронно построить этот запрос, но после возвращения запроса получение элементов из него будет происходить синхронно. `IAsyncEnumerable<T>` тоже возвращает несколько элементов асинхронно, но отличие в том, что `IAsyncEnumerable<T>` может асинхронно действовать с каждым возвращаемым элементом. Это настоящий асинхронный поток элементов.

Асинхронные потоки и IObservable<T>

Наблюдаемые объекты являются истинным воплощением асинхронных потоков; они генерируют свои уведомления по одному с полноценной поддержкой асинхронного генерирования (без блокирования). Но паттерн потребления объектов для `IObservable<T>` полностью отличен от `IAsyncEnumerable<T>`. Подробности о `IObservable<T>` см. в главе 6.

Чтобы потреблять `IObservable<T>`, код должен определить LINQ-подобный запрос, через который будут проходить наблюдаемые уведомления, после чего подписаться на наблюдаемый объект для запуска потока. При работе с наблюдаемыми объектами код сначала определяет, как будет реагировать на входящие уведомления, а затем включает их (отсюда и «реактивность» в названии). С другой стороны, потребление `IAsyncEnumerable<T>` очень похоже на потребление `IEnumerable<T>`, кроме асинхронности.

Также возникает проблема обратного давления: все уведомления в `System.Reactive` синхронны, поэтому сразу же после того, как уведомление одного элемента отправляется подписчикам, наблюдаемый объект продолжает выполнение и получает следующий элемент для публикации, возможно — с повторным вызовом API. Если потребляющий код использует поток асинхронно (т. е. с выполнением некоторого асинхронного действия для каждого уведомления при его поступлении), то наблюдаемый объект опередит потребляющий код.

Удобно считать, что `IObservable<T>` работает по принципу проталкивания (push), а `IAsyncEnumerable<T>` — по принципу вытягивания (pull). Наблюдаемый поток проталкивает уведомления коду, но асинхронный поток пассивно позволяет коду (асинхронно) вытягивать данные. Только когда потребляющий код запросит следующий элемент, наблюдаемый поток возобновит выполнение.

Итоги

Пожалуй, стоит рассмотреть теоретический пример. Многие API получают параметры `offset` и `limit` для создания страничной организации результатов. Допустим, вы хотите определить метод, который получает результаты от API с поддержкой страничной организации, и метод должен

обрабатывать страницы, чтобы этим не приходилось заниматься нашим высокоуровневым методам.

Если метод возвращает `Task<T>`, вы ограничиваетесь возвращением только одного `T`. Это нормально для одного вызова API, результатом которого является `T`, но он не будет плохо работать в качестве возвращаемого типа, если вы хотите, чтобы метод вызывал API несколько раз.

Если метод возвращает `IEnumerable<T>`, можно создать цикл, который перебирает результаты API по страницам, вызывая его несколько раз. Каждый раз, когда метод обращается с вызовом к API, он использует `yield return` с результатами этой страницы. Дальнейшие вызовы API необходимы только в том случае, если перечисление продолжается. К сожалению, методы, возвращающие `IEnumerable<T>`, не могут быть асинхронными, так что все вызовы API вынуждены быть синхронными.

Если метод возвращает `Task<List<T>>`, можно создать цикл, который по страницам перебирает результаты API и вызывает API асинхронно. Тем не менее код не может возвращать каждый элемент при получении ответа; ему придется построить все результаты и вернуть их одновременно.

Если ваш метод возвращает `IObservable<T>`, вы сможете использовать `System.Reactive` для реализации наблюдаемого потока, который начинает запросы при подписке и публикует каждый элемент при получении. Абстракция работает по принципу выталкивания; для потребляющего кода все выглядит так, словно результаты API проталкиваются им, что несколько затрудняет обработку. `IObservable<T>` будет лучше подходить для таких сценариев, как получение и реакция на сообщения WebSocket/SignalR.

Если ваш метод возвращает `IAsyncEnumerable<T>`, можно создать естественный цикл, использующий как `await`, так и `yield return` для создания настоящего асинхронного потока на базе вытягивания. `IAsyncEnumerable<T>` отлично подходит для таких сценариев.

В табл. 3.1 приведена сводка различных ролей распространенных типов.

На момент сдачи книги в печать .NET Core 3.0 все еще находится в фазе бета-тестирования, поэтому подробности асинхронных потоков могут измениться.

Таблица 3.1. Классификация типов

Тип	Одно или несколько значений	Асинхронно или синхронно	Вытягивание или проталкивание
`T`	Одно	Синхронно	–
`IEnumerable<T>`	Несколько	Синхронно	–
`Task<T>`	Одно	Асинхронно	Вытягивание
`IAsyncEnumerable<T>`	Несколько	Асинхронно	Вытягивание
`IObservable<T>`	Одно или несколько	Асинхронно	Проталкивание

3.1. Создание асинхронных потоков

Задача

Нужно вернуть несколько значений, при этом каждое значение может потребовать некоторой асинхронной работы. Задачу можно решить одним из двух путей:

- Есть несколько значений, которые требуется вернуть (например, `IEnumerable<T>`), а затем выполнить некоторую асинхронную работу.
- Есть одно асинхронное возвращение (как `Task<T>`), после которого добавляются другие возвращаемые значения.

Решение

Возвращение нескольких значений из метода может осуществляться командой `yield return`, а асинхронные методы используют `async` и `await`. С асинхронными потоками можно объединить эти два подхода; просто используйте возвращаемый тип `IAsyncEnumerable<T>`:

```
async IAsyncEnumerable<int> GetValuesAsync()
{
  await Task.Delay(1000); // Асинхронная работа
  yield return 10;
  await Task.Delay(1000); // Другая асинхронная работа
  yield return 13;
}
```

Этот простой пример показывает, как `await` может использоваться в сочетании с `yield return` для создания асинхронного потока.

В другом, более реалистичном примере асинхронно перебираются результаты API, использующего параметры для страничной организации результатов:

```
async IAsyncEnumerable<string> GetValuesAsync(HttpClient client)
{
  int offset = 0;
  const int limit = 10;
  while (true)
  {
    // Получить текущую страницу результатов и разобрать их.
    string result = await client.GetStringAsync(
        $"https://example.com/api/values?offset={offset}&limit={limit}");
    string[] valuesOnThisPage = result.Split('\n');

    // Произвести результаты для этой страницы.
    foreach (string value in valuesOnThisPage)
      yield return value;

    // Если это последняя страница, работа закончена.
    if (valuesOnThisPage.Length != limit)
      break;

    // В противном случае перейти к следующей странице.
    offset += limit;
  }
}
```

Когда метод `GetValuesAsync` начинает работу, он выдает асинхронный запрос первой страницы данных, после чего производит первый элемент. Когда будет запрошен второй элемент, `GetValuesAsync` выдает его немедленно, потому что он содержится на той же первой странице данных. Следующий элемент также находится на этой странице... и т. д. до 10 элементов. Затем при запросе 11-го элемента были произведены все значения в `valuesOnThisPage`, и на первой странице элементов уже не осталось. `GetValuesAsync` продолжит выполнение своего цикла `while`, перейдет к следующей странице, выполнит асинхронный запрос второй страницы данных, получит обратно новую группу значений, после чего произведет 11-й элемент.

Пояснение

С самого момента появления `async` и `await` пользователи задавались вопросом, как использовать их с `yield return`. В течение многих лет это было невозможно, но асинхронные потоки ввели эту возможность в C# и современные версии .NET. В более реалистичном примере стоит обратить внимание на одну особенность: асинхронная работа нужна не для всех результатов. В приведенном примере с длиной страницы 10 только приблизительно одному из каждых 10 элементов потребуется асинхронная работа. Если размер страницы равен 20, то асинхронная работа потребуется только одному из каждых 20 элементов.

Это обычный паттерн с асинхронными потоками. Для многих потоков большинство операций асинхронного перебора на самом деле синхронно; асинхронные потоки только позволяют асинхронно получить любой следующий элемент. Асинхронные потоки проектировались с учетом как асинхронного, так и синхронного кода; вот почему асинхронные потоки строятся на основе `ValueTask<T>`. Используя `ValueTask<T>` во внутренней реализации, асинхронные потоки максимизируют свою эффективность как при синхронном, так и при асинхронном получении элементов. За дополнительной информацией о типе `ValueTask<T>` и о том, в каких ситуациях его уместно использовать, рассказано в рецепте 2.10.

Когда вы реализуете асинхронные потоки, подумайте о поддержке отмены. В рецепте 3.4 подробно обсуждается отмена с асинхронными потоками. Некоторые сценарии не требуют реальной отмены; потребляющий код всегда может отказаться от получения следующего элемента. Это абсолютно нормальный подход при отсутствии внешнего источника для отмены. Если вы хотите, чтобы асинхронный поток можно было отменить даже в процессе получения следующего элемента, то следует обеспечить поддержку отмены с использованием `CancellationToken`.

Дополнительная информация

В рецепте 3.2 рассматривается потребление асинхронных потоков.

В рецепте 3.4 рассматривается обработка отмены для асинхронных потоков.

В рецепте 2.10 приведена более подробная информация о `ValueTask<T>` и о том, в каких ситуациях его уместно использовать.

3.2. Потребление асинхронных потоков

Задача

Требуется обработать результаты асинхронного потока, также называемого асинхронным перечисляемым объектом.

Решение

Потребление асинхронной операции осуществляется ключевым словом `await`, а для потребления перечисляемого объекта обычно используется `foreach`. Потребление асинхронного перечисляемого объекта основано на объединении этих двух конструкций в `await foreach`. Например, для асинхронного перечисляемого объекта, который выдает ответы API по страницам, можно организовать потребление и вывести каждый элемент на консоль:

```
IAsyncEnumerable<string> GetValuesAsync(HttpClient client);

public async Task ProcessValueAsync(HttpClient client)
{
  await foreach (string value in GetValuesAsync(client))
  {
    Console.WriteLine(value);
  }
}
```

На концептуальном уровне вызывается метод `GetValuesAsync`, который возвращает `IAsyncEnumerable<T>`. Цикл `foreach` затем создает асинхронный перечислитель на базе асинхронного перечисляемого объекта. Асинхронные перечислители на логическом уровне похожи на обычные перечислители, не считая того, что операция «получить следующий элемент» может быть асинхронной. Таким образом, `await foreach` будет ожидать поступления следующего элемента или завершения асинхронного перечислителя. Если элемент поступил, то `await foreach` выполнит свое тело цикла; если асинхронный перечислитель завершен, происходит выход из цикла.

Также можно выполнить асинхронную обработку каждого элемента:

```
IAsyncEnumerable<string> GetValuesAsync(HttpClient client);

public async Task ProcessValueAsync(HttpClient client)
{
  await foreach (string value in GetValuesAsync(client))
  {
    await Task.Delay(100); // асинхронная работа
    Console.WriteLine(value);
  }
}
```

В этом случае `await foreach` не переходит к следующему элементу до завершения тела цикла. Таким образом, `await foreach` асинхронно получит первый элемент, после чего асинхронно выполняет тело цикла для первого элемента, затем асинхронно получает первый элемент, асинхронно выполняет тело цикла для следующего элемента и т. д.

В `await foreach` скрыта команда `await`: к операции «получить следующий элемент» применяется `await`. С обычной командой `await` можно обойти неявно сохраненный контекст с помощью `ConfigureAwait(false)`, как описано в разделе 2.7. Асинхронные потоки также поддерживают `ConfigureAwait(false)`, которые передаются скрытым командам `await`:

```
IAsyncEnumerable<string> GetValuesAsync(HttpClient client);

public async Task ProcessValueAsync(HttpClient client)
{
  await foreach (string value in
     GetValuesAsync(client).ConfigureAwait(false))
  {
    await Task.Delay(100).ConfigureAwait(false); // асинхронная работа
    Console.WriteLine(value);
  }
}
```

Пояснение

`await foreach` — самый логичный способ потребления асинхронных потоков. Язык поддерживает `ConfigureAwait(false)` для предотвращения контекста в `await foreach`.

Также возможен вариант с передачей маркеров отмены; этот вариант чуть сложнее из-за сложности асинхронных потоков (этот вариант рассматривается в рецепте 3.4). И хотя возможно и естественно использовать `await foreach` для потребления асинхронных потоков, есть обширная библиотека асинхронных операторов LINQ; наиболее популярные из них рассматриваются в рецепте 3.3.

Тело `await foreach` может быть как синхронным, так и асинхронным. Для асинхронного случая правильно реализовать его намного сложнее, чем с другими потоковыми абстракциями (например, `IObservable<T>`). Это объясняется тем, что наблюдаемые подписки должны быть синхронными, но `await foreach` допускает естественную асинхронную обработку.

Конструкция `await foreach` генерирует команду `await`, используемую для операции «получить следующий элемент»; она также генерирует команду `await`, используемую для асинхронного освобождения перечисляемого объекта.

Дополнительная информация

В рецепте 3.1 рассматривается создание асинхронных потоков.

В рецепте 3.4 рассматривается реализация отмены для асинхронных потоков.

В рецепте 3.3 рассматриваются основные методы LINQ для асинхронных потоков.

В рецепте 11.6 рассматривается асинхронное освобождение ресурсов.

3.3. Использование LINQ с асинхронными потоками

Задача

Требуется обработать асинхронный поток с использованием четко определенных и хорошо протестированных операторов.

Решение

`IEnumerable<T>` поддерживает LINQ to Objects, а `IObservable<T>` поддерживает LINQ to Events. Оба типа поддерживают библиотеки методов расширения, которые определяют операторы, используемые для построения запросов. `IAsyncEnumerable<T>` также включает поддержку LINQ, предоставляемую сообществом .NET в NuGet-пакете `System.Linq.Async`.

Например, один из самых распространенных вопросов о LINQ заключается в том, как использовать оператор `Where`, если предикат `Where` является асинхронным. Вы хотите отфильтровать последовательность на основании некоторого асинхронного условия — например, необходимо провести поиск каждого элемента в базе данных или API, чтобы узнать, должен ли он быть включен в итоговую последовательность. `Where` не работает с асинхронными условиями, потому что оператор `Where` требует, чтобы его делегат возвращал немедленный синхронный ответ.

У асинхронных потоков имеется вспомогательная библиотека, определяющая много полезных операций. В следующем примере `WhereAwait` является правильным решением:

```
IAsyncEnumerable<int> values = SlowRange().WhereAwait(
    async value =>
    {
      // Выполнить некоторую асинхронную работу для определения
      //  того, должен ли элемент быть включен в результат.
      await Task.Delay(10);
      return value % 2 == 0;
    });

await foreach (int result in values)
{
  Console.WriteLine(result);
}

// Производит последовательность, которая замедляется
// в процессе выполнения операции.
async IAsyncEnumerable<int> SlowRange()
{
  for (int i = 0; i != 10; ++i)
  {
```

```
    await Task.Delay(i * 100);
    yield return i;
  }
}
```

Операторы LINQ для асинхронных потоков также включают синхронные версии; есть смысл применить синхронную операцию `Where` (`Select` и т. д.) для асинхронного потока. Результат все равно представляет собой асинхронный поток:

```
IAsyncEnumerable<int> values = SlowRange().Where(
    value => value % 2 == 0);

await foreach (int result in values)
{
  Console.WriteLine(result);
}
```

Здесь присутствуют все знакомые операторы LINQ: `Where`, `Select`, `SelectMany` и даже `Join`. Многие операторы LINQ теперь могут получать асинхронных делегатов (как в приведенном примере с `WhereAwait`).

Пояснение

Асинхронные потоки работают по принципу вытягивания, поэтому здесь нет операторов, связанных со временем (как для наблюдаемых объектов). `Throttle` и `Sample` здесь не имеют смысла, так как элементы вытягиваются из асинхронного потока по требованию.

Методы LINQ для асинхронных потоков также могут принести пользу для обычных перечисляемых объектов. Оказавшись в этой ситуации, можно вызвать `ToAsyncEnumerable()` для любого `IEnumerable<T>`; тогда вы получите интерфейс асинхронного потока, который можно использовать с `WhereAwait`, `SelectAwait` и другими операторами, которые поддерживают асинхронных делегатов.

Прежде чем погружаться в подробности, необходимо сказать пару слов об именах. Пример в этом рецепте использует `WhereAwait` как асинхронный эквивалент `Where`. При изучении операторов LINQ для асинхронных потоков вы увидите, что одни из них оканчиваются суффиксом `Async`, а другие — суффиксом `Await`. Операторы, заканчивающиеся суффиксом `Async`,

возвращают объект, допускающий ожидание; они представляют обычное значение, а не асинхронную последовательность. Операторы с суффиксом `Await` получают асинхронного делегата; `Await` в имени подразумевает, что они фактически выполняют `await` с переданным им делегатом.

Мы уже рассматривали пример суффикса `Await` в случае с `Where` и `WhereAwait`. Суффикс `Async` применяется только к *операторам терминации* (termination operators) — операторам, которые извлекают некоторое значение или выполняют некоторые вычисления и возвращают асинхронное скалярное значение вместо асинхронной последовательности. Пример такого оператора — `CountAsync`, версия `Count` для асинхронного потока, которая может подсчитать количество элементов, соответствующая некоторому предикату:

```
int count = await SlowRange().CountAsync(
    value => value % 2 == 0);
```

Предикат может *также* быть асинхронным; в этом случае используется оператор `CountAwaitAsync`, поскольку он получает асинхронного делегата (который будет использоваться с `await`) и производит одно терминальное значение:

```
int count = await SlowRange().CountAwaitAsync(
    async value =>
    {
      await Task.Delay(10);
      return value % 2 == 0;
    });
```

Операторы, которые могут получать делегатов, существуют в двух именах: с суффиксом `Await` и без него. Кроме того, операторы, возвращающие терминальное значение вместо асинхронного потока, завершаются суффиксом `Async`. Если оператор получает асинхронного делегата и возвращает терминальное значение, то имеет оба суффикса.

Операторы LINQ для асинхронных потоков находятся в NuGet-пакете для `System.Linq.Async`. Дополнительные операторы LINQ для асинхронных потоков находятся в NuGet-пакете для `System.Interactive.Async`.

Дополнительная информация

В рецепте 3.1 рассматривается производство асинхронных потоков.

В рецепте 3.2 рассматривается потребление асинхронных потоков.

3.4. Асинхронные потоки и отмена

Задача

Требуется механизм отмены асинхронных потоков.

Решение

Не всем асинхронным потокам необходима отмена. Перечисление может быть просто остановлено при достижении условия. Если это единственная разновидность «отмены», реально необходимая в программе, то полноценная отмена не нужна, как показывает следующий пример:

```
await foreach (int result in SlowRange())
{
  Console.WriteLine(result);
  if (result >= 8)
    break;
}

// Производит последовательность, которая замедляется
// в процессе выполнения операции.

async IAsyncEnumerable<int> SlowRange()
{
  for (int i = 0; i != 10; ++i)
  {
    await Task.Delay(i * 100);
    yield return i;
  }
}
```

Часто бывает полезно отменять асинхронные потоки, так как некоторые операторы передают маркеры отмены своим потокам-источникам. В этом сценарии следует использовать `CancellationToken` для остановки `await foreach` из внешнего кода.

`async`-метод, возвращающий `IAsyncEnumerable<T>`, может получать маркер отмены, для чего определяется параметр, помеченный атрибутом `EnumeratorCancellation`. После этого маркер можно использовать естественным образом, для чего он обычно передается другим API, получающим маркеры отмены:

```
using var cts = new CancellationTokenSource(500);
CancellationToken token = cts.Token;
await foreach (int result in SlowRange(token))
{
  Console.WriteLine(result);
}

// Производит последовательность, которая замедляется
// в процессе выполнения операции.
async IAsyncEnumerable<int> SlowRange(
    [EnumeratorCancellation] CancellationToken token = default)
{
  for (int i = 0; i != 10; ++i)
  {
    await Task.Delay(i * 100, token);
    yield return i;
  }
}
```

Пояснение

В этом примере `CancellationToken` передается непосредственно методу, возвращающему асинхронный перечислитель. Это самый распространенный вариант использования.

Возможны и другие сценарии, в которых код получает асинхронный перечислитель и хочет применить `CancellationToken` к перечислителям, которые он использует. Маркеры отмены используются при запуске нового перечисления для перечисляемого объекта, поэтому есть смысл использовать `CancellationToken` именно таким образом. Сам перечис-

ляемый объект *определяется* методом `SlowRange`, но он не запускается до момента потребления. Бывают даже ситуации, в которых разные маркеры отмены должны передаваться разным перечислениям перечисляемого объекта.

Вкратце отменяться может не перечисляемый объект, но перечислитель, созданный этим перечисляемым объектом. Это нетипичный, но важный сценарий использования; именно по этой причине асинхронные потоки поддерживают метод расширения `WithCancellation`, который может использоваться для присоединения маркера `CancellationToken` к конкретной итерации асинхронного потока:

```
async Task ConsumeSequence(IAsyncEnumerable<int> items)
{
  using var cts = new CancellationTokenSource(500);
  CancellationToken token = cts.Token;
  await foreach (int result in items.WithCancellation(token))
  {
    Console.WriteLine(result);
  }
}

// Производит последовательность, которая замедляется
// в процессе выполнения операции.
async IAsyncEnumerable<int> SlowRange(
    [EnumeratorCancellation] CancellationToken token = default)
{
  for (int i = 0; i != 10; ++i)
  {
    await Task.Delay(i * 100, token);
    yield return i;
  }
}

await ConsumeSequence(SlowRange());
```

При наличии атрибута `EnumeratorCancellation` компилятор обеспечит передачу маркера из `WithCancellation` параметру `token`, помеченному `EnumeratorCancellation`, и запрос отмены теперь заставит `await foreach` выдать исключение `OperationCanceledException` после того, как он обработает несколько первых элементов.

Метод расширения `WithCancellation` не препятствует `ConfigureAwait(false)`. Оба метода расширения могут быть объединены в цепочку:

```
async Task ConsumeSequence(IAsyncEnumerable<int> items)
{
  using var cts = new CancellationTokenSource(500);
  CancellationToken token = cts.Token;
  await foreach (int result in items
      .WithCancellation(token).ConfigureAwait(false))
  {
    Console.WriteLine(result);
  }
}
```

Дополнительная информация

В рецепте 3.1 рассматривается создание асинхронных потоков.

В рецепте 3.2 рассматриваются асинхронные потоки.

В главе 10 рассматривается кооперативная отмена в разных технологиях.

ГЛАВА 4

Основы параллельного программирования

В этой главе представлены паттерны параллельного программирования. Параллельное программирование используется для разбиения блоков работы, ограниченных по вычислениям, и распределения их между несколькими потоками. Рецепты параллельной обработки ориентируются только на счетную работу. Если у вас имеются операции, асинхронные по своей природе (например, работа, связанная с вводом/выводом), которые должны выполняться параллельно, обращайтесь к главе 2 и рецепту 2.4.

Абстракции параллельной обработки, рассмотренные в этой главе, являются частью библиотеки TPL (Task Parallel Library). Библиотека TPL является частью фреймворка .NET.

4.1. Параллельная обработка данных

Задача

Имеется коллекция данных. Требуется выполнить одну и ту же операцию с каждым элементом данных. Эта операция является ограниченной по вычислениям и может занять некоторое время.

Решение

Тип `Parallel` содержит метод `ForEach`, разработанный специально для этой задачи. Следующий пример получает коллекцию матриц и поворачивает эти матрицы:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees)
{
  Parallel.ForEach(matrices, matrix => matrix.Rotate(degrees));
}
```

Возможны ситуации, в которых преждевременно требуется прервать цикл (например, при обнаружении недействительного значения). Следующий пример обращает каждую матрицу, но при обнаружении недействительной матрицы цикл будет прерван:

```
void InvertMatrices(IEnumerable<Matrix> matrices)
{
  Parallel.ForEach(matrices, (matrix, state) =>
  {
    if (!matrix.IsInvertible)
      state.Stop();
    else
      matrix.Invert();
  });
}
```

Этот код использует `ParallelLoopState.Stop` для остановки цикла и предотвращения любых дальнейших вызовов тела цикла. Учтите, что цикл является параллельным, поэтому могут уже выполняться другие вызовы тела цикла, включая вызовы для элементов, следующих после текущего. В приведенном примере кода если третья матрица не является обратимой, то цикл прерывается и новые матрицы обрабатываться не будут, но может оказаться, что уже обрабатываются другие матрицы (например, четвертая и пятая).

Более распространенная ситуация встречается тогда, когда требуется отменить параллельный цикл. Это не то же, что остановка цикла; цикл *останавливается* изнутри и *отменяется* за своими пределами. Например, кнопка отмены может отменить `CancellationTokenSource`, отменяя параллельный цикл, как в следующем примере:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees,
    CancellationToken token)
{
  Parallel.ForEach(matrices,
      new ParallelOptions { CancellationToken = token },
      matrix => matrix.Rotate(degrees));
}
```

Следует иметь в виду, что каждая параллельная задача может выполняться в другом потоке, поэтому любое совместное состояние должно быть защищено. Следующий пример обращает каждую матрицу и подсчитывает количество матриц, которые обратить не удалось:

```
// Примечание: это не самая эффективная реализация.
// Это всего лишь пример использования блокировки
// для защиты совместного состояния.
int InvertMatrices(IEnumerable<Matrix> matrices)
{
  object mutex = new object();
  int nonInvertibleCount = 0;
  Parallel.ForEach(matrices, matrix =>
  {
    if (matrix.IsInvertible)
    {
      matrix.Invert();
    }
    else
    {
      lock (mutex)
      {
        ++nonInvertibleCount;
      }
    }
  });
  return nonInvertibleCount;
}
```

Пояснение

Метод `Parallel.ForEach` предоставляет возможность параллельной обработки для последовательности значений. Аналогичное решение Parallel LINQ (PLINQ) предоставляет практически те же возможности в LINQ-подобном синтаксисе. Одно из различий между `Parallel` и PLINQ заключается в том, что PLINQ предполагает, что может использовать все ядра на компьютере, тогда как `Parallel` может динамически реагировать на изменения условий процессора.

`Parallel.ForEach` реализует параллельный цикл `foreach`. Если вам потребуется выполнить параллельный цикл `for`, то класс `Parallel` также

поддерживает метод `Parallel.For`. Метод `Parallel.For` особенно полезен при работе с несколькими массивами данных, которые получают один индекс.

Дополнительная информация

В рецепте 4.2 рассматривается параллельное агрегирование серий значений, включая суммирование и вычисление средних значений.

В рецепте 4.5 рассматриваются основы PLINQ.

В главе 10 рассматривается отмена.

4.2. Параллельное агрегирование

Задача

Требуется агрегировать результаты при завершении параллельной операции (примеры агрегирования — суммирование значений или вычисление среднего).

Решение

Для поддержки агрегирования класс `Parallel` использует концепцию *локальных значений* — переменных, существующих локально внутри параллельного цикла. Это означает, что тело цикла может просто обратиться к значению напрямую, без необходимости синхронизации. Когда цикл готов к агрегированию всех своих локальных результатов, он делает это с помощью делегата `localFinally`. Следует отметить, что делегату `localFinally` не нужно синхронизировать доступ к переменной для хранения результата. Пример параллельного суммирования:

```
// Примечание: это не самая эффективная реализация.
// Это всего лишь пример использования блокировки
// для защиты совместного состояния.
int ParallelSum(IEnumerable<int> values)
{
  object mutex = new object();
  int result = 0;
```

```
  Parallel.ForEach(source: values,
      localInit: () => 0,
      body: (item, state, localValue) => localValue + item,
      localFinally: localValue =>
      {
        lock (mutex)
          result += localValue;
      });
  return result;
}
```

В Parallel LINQ реализована более понятная поддержка агрегирования, чем в классе `Parallel`:

```
int ParallelSum(IEnumerable<int> values)
{
  return values.AsParallel().Sum();
}
```

О'кей, это был дешевый трюк, потому что в PLINQ реализована встроенная поддержка многих распространенных операторов (например, `Sum`). В PLINQ также предусмотрена обобщенная поддержка агрегирования с оператором `Aggregate`:

```
int ParallelSum(IEnumerable<int> values)
{
  return values.AsParallel().Aggregate(
      seed: 0,
      func: (sum, item) => sum + item
  );
}
```

Пояснение

Если вы уже используете класс `Parallel`, следует использовать его поддержку агрегирования. В остальных случаях поддержка PLINQ, как правило, более выразительна, а код получается короче.

Дополнительная информация

В рецепте 4.5 изложены основы PLINQ.

4.3. Параллельный вызов

Задача

Имеется набор методов, которые должны вызываться параллельно. Эти методы (в основном) независимы друг от друга.

Решение

Класс `Parallel` содержит простой метод `Invoke`, спроектированный для таких сценариев. В следующем примере массив разбивается надвое, и две половины обрабатываются независимо:

```
void ProcessArray(double[] array)
{
  Parallel.Invoke(
      () => ProcessPartialArray(array, 0, array.Length / 2),
      () => ProcessPartialArray(array, array.Length / 2, array.Length)
  );
}

void ProcessPartialArray(double[] array, int begin, int end)
{
  // Обработка, интенсивно использующая процессор...
}
```

Методу `Parallel.Invoke` также можно передать массив делегатов, если количество вызовов неизвестно до момента выполнения:

```
void DoAction20Times(Action action)
{
  Action[] actions = Enumerable.Repeat(action, 20).ToArray();
  Parallel.Invoke(actions);
}
```

`Parallel.Invoke` поддерживает отмену, как и другие методы класса `Parallel`:

```
void DoAction20Times(Action action, CancellationToken token)
{
  Action[] actions = Enumerable.Repeat(action, 20).ToArray();
```

```
  Parallel.Invoke(new ParallelOptions { CancellationToken = token },
     actions);
}
```

Пояснение

Метод `Parallel.Invoke` — отличное решение для простого параллельного вызова. Отмечу, что он уже не так хорошо подходит для ситуаций, в которых требуется активизировать действие для каждого элемента входных данных (для этого лучше использовать `Parallel.ForEach`), или если каждое действие производит некоторый вывод (вместо этого следует использовать Parallel LINQ).

Дополнительная информация

В рецепте 4.1 рассматривается метод `Parallel.ForEach`, который выполняет действие для каждого элемента данных.

В рецепте 4.5 рассматривается Parallel LINQ.

4.4. Динамический параллелизм

Задача

Требуется реализовать более сложную параллельную ситуацию: структура и количество параллельных задач зависит от информации, которая становится известной только во время выполнения.

Решение

В библиотеке TPL (Task Parallel Library) центральное место занимает тип `Task`. Класс `Parallel` и Parallel LINQ — всего лишь удобные обертки для мощного типа `Task`. Если потребуется реализовать динамический параллелизм, проще использовать тип `Task` напрямую.

В приведенном ниже примере для каждого узла бинарного дерева необходимо выполнить некоторую затратную обработку. Структура дерева неизвестна до стадии выполнения, поэтому этот сценарий хорошо по-

дойдет для динамического параллелизма. Метод `Traverse` обрабатывает текущий узел, а затем создает две дочерние задачи, по одной для каждой ветви под узлом (в данном примере предполагается, что родительские узлы должны быть обработаны до перехода к дочерним узлам). Метод `ProcessTree` начинает обработку, создавая родительскую задачу верхнего уровня и ожидая ее завершения:

```
void Traverse(Node current)
{
  DoExpensiveActionOnNode(current);
  if (current.Left != null)
  {
    Task.Factory.StartNew(
        () => Traverse(current.Left),
        CancellationToken.None,
        TaskCreationOptions.AttachedToParent,
        TaskScheduler.Default);
  }
  if (current.Right != null)
  {
    Task.Factory.StartNew(
        () => Traverse(current.Right),
        CancellationToken.None,
        TaskCreationOptions.AttachedToParent,
        TaskScheduler.Default);
  }
}

void ProcessTree(Node root)
{
  Task task = Task.Factory.StartNew(
      () => Traverse(root),
      CancellationToken.None,
      TaskCreationOptions.None,
      TaskScheduler.Default);
  task.Wait();
}
```

Флаг `AttachedToParent` гарантирует, что задача `Task` для каждой ветви связывается с задачей `Task` своего родительского узла. Таким образом создаются отношения «родитель/потомок» между экземплярами `Task`, мо-

делирующими отношения «родитель/потомок» в узлах дерева. Родительские задачи выполняют своего делегата, после чего ожидают завершения своих дочерних задач. Исключения от дочерних задач распространяются от дочерних задач к своей родительской задаче. Таким образом, `ProcessTree` может ожидать задач для всего дерева, для чего достаточно вызвать `Wait` для одной задачи `Task` в корне дерева.

Если ваша ситуация не относится к категории «родитель/потомок», вы можете запланировать запуск любой задачи после другой задачи, используя *продолжение*. Продолжение (continuation) представляет собой отдельную задачу, которая выполняется после завершения исходной:

```
Task task = Task.Factory.StartNew(
    () => Thread.Sleep(TimeSpan.FromSeconds(2)),
    CancellationToken.None,
    TaskCreationOptions.None,
    TaskScheduler.Default);
Task continuation = task.ContinueWith(
    t => Trace.WriteLine("Task is done"),
    CancellationToken.None,
    TaskContinuationOptions.None,
    TaskScheduler.Default);
// Аргумент "t" для продолжения - то же, что "task".
```

Пояснение

`CancellationToken.None` и `TaskScheduler.Default` используются в предыдущем примере кода. Маркеры отмены рассматриваются в рецепте 10.2, а планировщики задач — в рецепте 13.3. Всегда лучше явно задать планировщик `TaskScheduler`, используемый `StartNew` и `ContinueWith`.

Такая структура родительских и дочерних задач типична для динамического параллелизма, хотя и не обязательна. С таким же успехом можно сохранить каждую новую задачу в потокобезопасной коллекции, а затем ожидать завершения их всех с использованием `Task.WaitAll`.

Использование `Task` для параллельной обработки принципиально отличается от использования `Task` для асинхронной обработки.

Тип `Task` в параллельном программировании служит двум целям: он может представлять параллельную или асинхронную задачу. Параллельные задачи могут использовать блокирующие методы, такие как `Task.Wait`, `Task.Result`, `Task.WaitAll` и `Task.WaitAny`. Параллельные задачи также обычно используют `AttachedToParent` для создания отношений «родитель/потомок» между задачами. Параллельные задачи следует создавать методами `Task.Run` или `Task.Factory.StartNew`.

С другой стороны, асинхронным задачам следует избегать блокирующих методов в пользу `await`, `Task.WhenAll` и `Task.WhenAny`. Асинхронные задачи не должны использовать `AttachedToParent`, но они могут формировать неявные отношения «родитель/потомок», используя ожидание других задач.

Дополнительная информация

В рецепте 4.3 рассматривается параллельный вызов последовательности методов в том случае, если все методы известны на момент начала параллельной работы.

4.5. Parallel LINQ

Задача

Требуется выполнить параллельную обработку последовательности данных, чтобы сгенерировать другую их последовательность или обобщение этих данных.

Решение

Многие разработчики знакомы с технологией LINQ, позволяющей программировать вычисления с последовательностями, работающей по принципу вытягивания. Parallel LINQ (PLINQ) расширяет эту поддержку LINQ параллельной обработкой.

PLINQ хорошо работает в потоковых сценариях, которые получают последовательность входных значений и производят последовательность выходных значений. Следующий простой пример просто умножает каждый

элемент последовательности на 2 (в реальных сценариях выполняются гораздо более серьезные вычисления, чем простое умножение):

```
IEnumerable<int> MultiplyBy2(IEnumerable<int> values)
{
  return values.AsParallel().Select(value => value * 2);
}
```

Пример может генерировать свои выходные значения в любом порядке; это поведение используется по умолчанию в Parallel LINQ. Также можно потребовать, чтобы сохранялся исходный порядок. В следующем примере обработка ведется параллельно, но с сохранением исходного порядка:

```
IEnumerable<int> MultiplyBy2(IEnumerable<int> values)
{
  return values.AsParallel().AsOrdered().Select(value => value * 2);
}
```

Другое логичное применение Parallel LINQ — агрегирование или обобщение данных в параллельном режиме. В следующем примере выполняется параллельное суммирование:

```
int ParallelSum(IEnumerable<int> values)
{
  return values.AsParallel().Sum();
}
```

Пояснение

Класс `Parallel` хорошо подходит для многих сценариев, но код PLINQ получается более простым при агрегировании или преобразовании одной последовательности в другую. Следует помнить, что класс `Parallel` ведет себя более корректно с другими процессами в системе, чем PLINQ; этот фактор становится особенно существенным при выполнении параллельной обработки на серверной машине.

PLINQ предоставляет параллельные версии многих операторов, включая фильтрацию (`Where`), проекцию (`Select`) и разные виды агрегирования, такие как `Sum`, `Average` и более общую форму `Aggregate`. В общем случае все, что можно сделать с обычным LINQ, также можно сделать в параллельном режиме с PLINQ. В результате PLINQ становится отличным кандидатом

для переработки существующего кода LINQ, который выиграл бы от выполнения в параллельном режиме.

Дополнительная информация

В рецепте 4.1 рассматривается использование класса `Parallel` для выполнения кода для каждого элемента в последовательности.

В рецепте 10.5 рассматривается отмена запросов PLINQ.

ГЛАВА 5

Основы Dataflow

TPL Dataflow — мощная библиотека, позволяющая создать сеть или конвейер, а затем (асинхронно) отправить по ним свои данные. Dataflow использует декларативный стиль программирования; т. е. сначала вы полностью определяете сеть, а затем начинаете обрабатывать данные. Сеть описывает структуру, по которой перемещаются данные. Для этого придется взглянуть на свое приложение под несколько иным углом, но после того как вы сделаете этот шаг, поток данных (dataflow) станет само собой разумеющимся кандидатом для многих сценариев.

Каждая сеть состоит из различных блоков, связанных друг с другом. Отдельные блоки просты, они отвечают только за один этап обработки данных. Когда блок завершает работу над своими данными, он передает свой результат всем связанным блокам.

Чтобы использовать TPL Dataflow, установите в своем приложении NuGet-пакет из `System.Threading.Tasks.Dataflow`.

5.1. Связывание блоков

Задача

Требуется связать блоки Dataflow для создания сети.

Решение

Блоки, предоставляемые библиотекой TPL Dataflow, определяют только самые базовые составляющие. Многие полезные методы TPL Dataflow в действительности являются методами расширения. Метод расширения `LinkTo` предоставляет простой механизм связывания блоков потока данных:

```
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);
// После связывания значения, выходящие из multiplyBlock,
// будут входить в subtractBlock.
multiplyBlock.LinkTo(subtractBlock);
```

По умолчанию связанные блоки только распространяют данные; они не распространяют завершение (или ошибки). Если ваш поток данных линеен (например, в конвейере), то, скорее всего, вы захотите распространять завершение. Чтобы распространять завершение (и ошибки), установите параметр `PropagateCompletion` для связи:

```
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);

var options = new DataflowLinkOptions { PropagateCompletion = true };
multiplyBlock.LinkTo(subtractBlock, options);

...

// Завершение первого блока автоматически распространяется во второй блок.
multiplyBlock.Complete();
await subtractBlock.Completion;
```

Пояснение

После связывания данные будут автоматически перемещаться от блока-источника к блоку-приемнику. Параметр `PropagateCompletion` перемещает не только данные, но и завершение; тем не менее на каждом этапе конвейера сбойный блок будет распространять в следующий блок свое исключение, упакованное в `AggregateException`. Таким образом, если имеется длинный конвейер, распространяющий завершения, исходная ошибка может быть вложена в несколько экземпляров `AggregateException`. `AggregateException` содержит несколько методов (например, `Flatten`), упрощающих обработку ошибок в подобных ситуациях.

Блоки потока данных могут связываться друг с другом многими разными способами; сеть может содержать ветвления, соединения и даже циклы. Для большинства сценариев обычно хватает простого линейного конвейера. Мы будем работать в основном с конвейерами (и рассмотрим ветвления); более сложные сценарии выходят за рамки книги.

Тип `DataflowLinkOptions` предоставляет ряд параметров, которые могут устанавливаться для связей (как, например, параметр `PropagateCompletion`, использованный в этом решении), а перегруженная версия `LinkTo` также может получать предикат, использующийся для фильтрации данных, передаваемых через связь. Данные, прошедшие через фильтр, перемещаются по связи; данные, не прошедшие через фильтр, не теряются, а пытаются пройти по альтернативной связи (и остаются в блоке, если другой связи не существует). Если элемент данных «застревает» в блоке, то этот блок не производит других элементов данных; весь блок приостанавливается до извлечения элемента данных.

Дополнительная информация

В рецепте 5.2 рассматривается распространение ошибок по связям.

В рецепте 5.3 рассматриваются связи между блоками.

В рецепте 8.8 рассматривается связывание блоков потока данных с наблюдаемыми потоками System.Reactive.

5.2. Распространение ошибок

Задача

Найти способ реагировать на ошибки, которые могут происходить в сети потока данных.

Решение

Если делегат, переданный блоку потока данных, выдает исключение, то этот блок входит в состояние отказа. Блок в состоянии отказа теряет все свои данные (и перестает принимать новые). В следующем коде блок не производит никаких выходных данных; первое значение выдает исключение, а второе просто теряется:

```
var block = new TransformBlock<int, int>(item =>
{
  if (item == 1)
    throw new InvalidOperationException("Blech.");
```

```
  return item * 2;
});
block.Post(1);
block.Post(2);
```

Чтобы перехватывать исключения от блока потока данных, необходимо ожидать его свойства `Completion`. Свойство `Completion` возвращает объект `Task`, который завершается при завершении блока, а если в блоке происходит отказ, то и в задаче `Completion` тоже происходит отказ:

```
try
{
  var block = new TransformBlock<int, int>(item =>
  {
    if (item == 1)
      throw new InvalidOperationException("Blech.");
    return item * 2;
  });
  block.Post(1);
  await block.Completion;
}
catch (InvalidOperationException)
{
  // Здесь перехватывается исключение.
}
```

Когда вы распространяете завершение с помощью параметра связи `PropagateCompletion`, ошибки тоже распространяются. Однако исключение передается следующему блоку, упакованному в `AggregateException`. Следующий пример перехватывает исключение в конце конвейера, поэтому он перехватит `AggregateException`, если исключение было распространено из предыдущих блоков:

```
try
{
  var multiplyBlock = new TransformBlock<int, int>(item =>
  {
    if (item == 1)
      throw new InvalidOperationException("Blech.");
    return item * 2;
  });
  var subtractBlock = new TransformBlock<int, int>(item => item - 2);
```

```
  multiplyBlock.LinkTo(subtractBlock,
      new DataflowLinkOptions { PropagateCompletion = true });
  multiplyBlock.Post(1);
  await subtractBlock.Completion;
}
catch (AggregateException)
{
  // Здесь перехватывается исключение.
}
```

Каждый блок упаковывает входящие ошибки в `AggregateException`, даже если входящая ошибка уже представляет собой `AggregateException`. Если ошибка происходит на ранней стадии конвейера и успевает переместиться на несколько связей перед тем, как будет обнаружена, исходная ошибка будет упакована в `AggregateException` на нескольких уровнях. Метод `AggregateException.Flatten` упрощает обработку ошибок в этом сценарии.

Пояснение

Занимаясь построением сети (или конвейера), подумайте над тем, как будут обрабатываться ошибки. В более простых ситуациях может быть лучше распространить ошибки и перехватывать их все сразу в конце. В более сложных сетях вам, возможно, придется проверить каждый блок при завершении потока данных.

Также возможен другой вариант: если вы хотите, чтобы блоки сохраняли работоспособность перед лицом исключений, можно рассматривать исключения как другую разновидность данных и дать им проходить через сеть с правильно обрабатываемыми элементами данных. Использование этого паттерна позволит сохранить работоспособность сети потока данных, так как сами блоки не будут переходить в состояние отказа и продолжат обработку следующего элемента данных. За дополнительной информацией обращайтесь к рецепту 14.6.

Дополнительная информация

В рецепте 5.1 рассматриваются связи между блоками.

В рецепте 5.3 рассматривается разрыв связей между блоками.

В рецепте 14.6 рассматривается перемещение исключений вместе с данными в сети потока данных.

5.3. Удаление связей между блоками

Задача

В процессе обработки необходимо динамически изменить структуру потока данных. Это нетипичный сценарий, с которым вы вряд ли когда-либо столкнетесь в жизни.

Решение

Связи между блоками потока данных можно создавать или удалять в любой момент; данные могут свободно проходить по сети, и это не помешает безопасно создавать или удалять связи. Как создание, так и удаление связей являются полностью потокобезопасными.

При создании связи между блоками потока данных сохраните объект `IDisposable`, возвращенный методом `LinkTo`, и уничтожьте его, когда потребуется разорвать связь между блоками:

```
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);
IDisposable link = multiplyBlock.LinkTo(subtractBlock);
multiplyBlock.Post(1);
multiplyBlock.Post(2);
// Удаление связей между блоками.
// Данные, отправленные выше, могут быть уже переданы
// или не переданы по связи. В реальном коде стоит рассмотреть
// возможность блока using вместо простого вызова Dispose.
link.Dispose();
```

Пояснение

Если нет гарантии, что связь бездействует, при ее удалении может возникнуть состояние гонки (race). Но состояние гонки обычно не создает проблем; данные либо проходят по связи, перед тем как она будет унич-

тожена, либо не проходят. Здесь нет условий гонки, которые могли бы привести к дублированию или потере данных.

Сценарий с разрывом связи нетипичен, но и он может пригодиться в некоторых ситуациях. Например, невозможно изменить фильтр для связи — придется удалить старую связь и создать новую с новым фильтром (возможно, с присваиванием `DataflowLinkOptions.Append` значения `false`). Также удаление связи в стратегической точке может использоваться для приостановки сети потока данных.

Дополнительная информация

В рецепте 5.1 рассматривается создание связей между блоками.

5.4. Регулирование блоков

Задача

Имеется сеть потока данных с ветвлением. Требуется организовать передачу данных с распределением нагрузки.

Решение

По умолчанию блок, генерирующий выходные данные, проверяет все свои связи (в порядке их создания) и пытается последовательно передавать данные по каждому каналу. Кроме того, по умолчанию каждый блок поддерживает входной буфер и принимает произвольное количество данных до того, как он будет готов их обработать.

Это создает проблему в сценарии с ветвлением, в котором один блок-источник связан с двумя блоками-приемниками: в этом случае второй блок будет простаивать. Первый блок-приемник всегда будет принимать данные и буферизовать их, так что блок-источник никогда не будет пытаться передавать данные второму блоку-приемнику. Проблему можно решить *регулировкой* (throttling) блоков-приемников с использованием параметра блока `BoundedCapacity`. По умолчанию `BoundedCapacity` присваивается значение `DataflowBlockOptions.Unbounded`, при котором пер-

вый блок-приемник буферизует все данные, даже если еще не готов к их обработке.

`BoundedCapacity` можно присвоить любое значение больше нуля (или, конечно, `DataflowBlockOptions.Unbounded`). Если блоки-приемники успевают обрабатывать данные, поступающие от блоков-источников, простого значения 1 будет достаточно:

```
var sourceBlock = new BufferBlock<int>();
var options = new DataflowBlockOptions { BoundedCapacity = 1 };
var targetBlockA = new BufferBlock<int>(options);
var targetBlockB = new BufferBlock<int>(options);

sourceBlock.LinkTo(targetBlockA);
sourceBlock.LinkTo(targetBlockB);
```

Пояснение

Регулировка полезна для распределения нагрузки в конфигурациях с ветвлением, но она также может применяться везде, где возникнет необходимость в регулировании. Например, если сеть потока данных заполняется данными от операции ввода/вывода, можно применить `BoundedCapacity` к блокам своей сети. В этом случае вы не прочитаете слишком много данных ввода/вывода до того, как сеть будет к этому готова, а все входные данные не будут буферизованы сетью до того, как она сможет его обработать.

Дополнительная информация

В рецепте 5.1 рассматривается связывание блоков.

5.5. Параллельная обработка с блоками потока данных

Задача

Требуется выполнить параллельную обработку в сети потока данных.

Решение

По умолчанию каждый блок потока данных не зависит от других блоков. Когда вы связываете два блока, они будут выполнять обработку независимо друг от друга. Таким образом, в каждую сеть потока данных встроена некоторая степень естественного параллелизма.

Если требуется выйти за эти рамки, — например, если один конкретный блок выполняет интенсивные вычисления на процессоре, — вы можете дать команду этому блоку работать параллельно с входными данными, устанавливая параметр `MaxDegreeOfParallelism`. По умолчанию этому параметру тоже присваивается значение 1, поэтому каждый блок потока данных обрабатывает только один фрагмент данных за раз.

`BoundedCapacity` можно присвоить `DataflowBlockOptions.Unbounded` или любое значение, большее 0. Следующий пример позволяет любому количеству задач умножать данные одновременно:

```
var multiplyBlock = new TransformBlock<int, int>(
    item => item * 2,
    new ExecutionDataflowBlockOptions
    {
      MaxDegreeOfParallelism = DataflowBlockOptions.Unbounded
    });
var subtractBlock = new TransformBlock<int, int>(item => item - 2);
multiplyBlock.LinkTo(subtractBlock);
```

Пояснение

Параметр `MaxDegreeOfParallelism` упрощает организацию параллельной обработки в блоке. Сложнее определить, каким блокам это потребуется. Один из способов заключается в том, чтобы приостановить выполнение потока данных в отладчике и проанализировать количество элементов данных в очереди (т. е. элементов, которые еще не были обработаны блоком). Неожиданно высокое количество элементов данных может указывать на то, что реорганизация или параллелизация могли бы принести пользу.

`MaxDegreeOfParallelism` также работает и в том случае, если блок потока данных выполняет асинхронную обработку. В этом случае параметр `MaxDegreeOfParallelism` задает *уровень параллелизма* — определенное количество *слотов*. Каждый элемент данных занимает слот, когда блок

приступает к его обработке, и покидает этот слот только при полном завершении асинхронной обработки.

Дополнительная информация

В рецепте 5.1 рассматривается связывание блоков.

5.6. Создание собственных блоков

Задача

Имеется некоторая логика, которую требуется разместить в нестандартном блоке потока данных. Это позволит создавать большие блоки, содержащие сложную логику.

Решение

Можно выделить любую часть сети потока данных, содержащую один входной и один выходной блок, с помощью метода `Encapsulate`. `Encapsulate` формирует блок по двум конечным точкам. Распространение данных и завершение между этими конечными точками остаются на вашей ответственности. Следующий код создает из двух блоков нестандартный блок потока данных с распространением данных и завершения:

```
IPropagatorBlock<int, int> CreateMyCustomBlock()
{
  var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
  var addBlock = new TransformBlock<int, int>(item => item + 2);
  var divideBlock = new TransformBlock<int, int>(item => item / 2);

 var flowCompletion = new DataflowLinkOptions { PropagateCompletion = true };
  multiplyBlock.LinkTo(addBlock, flowCompletion);
  addBlock.LinkTo(divideBlock, flowCompletion);

  return DataflowBlock.Encapsulate(multiplyBlock, divideBlock);
}
```

Пояснение

Инкапсулируя сеть в нестандартный блок, подумайте о том, какие параметры вы хотите предоставить пользователям. Подумайте, как каждый параметр блока должен (или не должен) передаваться вашей внутренней сети; во многих случаях некоторые параметры блоков неприменимы или не имеют смысла. По этой причине нестандартные блоки обычно определяют собственные нестандартные параметры вместо того, чтобы получать параметр `DataflowBlockOptions`.

`DataflowBlock.Encapsulate` инкапсулирует только сеть с одним входным и одним выходным блоками. Если у вас имеется сеть с несколькими входными и/или выходными блоками, предназначенная для повторного использования, вам следует инкапсулировать ее в специальном объекте и предоставить доступ к входам и выходам как к свойствам типа `ITargetBlock<T>` (для входов) и `IReceivableSourceBlock<T>` (для выходов).

Все эти примеры используют `Encapsulate` для создания нестандартного блока. Также возможно реализовать интерфейсы потока данных самостоятельно, но это намного сложнее. Компания Microsoft опубликовала статью (https://devblogs.microsoft.com/pfxteam/paper-guide-to-implementing-custom-tpl-dataflow-blocks/) с описанием нетривиальных приемов создания нестандартных блоков потока данных.

Дополнительная информация

В рецепте 5.1 рассматривается связывание блоков.

В рецепте 5.2 рассматривается распространение ошибок по связям между каналами.

ГЛАВА 6

Основы System.Reactive

LINQ — набор языковых средств, которые могут использоваться разработчиками для выдачи запросов к последовательностям. Два самых популярных провайдера LINQ — LINQ to Objects (на базе `IEnumerable<T>`) и LINQ to Entities (на базе `IQueryable<T>`). Есть множество других провайдеров, имеющих сходную общую структуру. Запросы обрабатываются в отложенном режиме (lazily), а последовательности генерируют значения по мере необходимости. На концептуальном уровне используется модель *с вытягиванием*; при обработке элементы-значения извлекаются из очереди по одному.

System.Reactive (Rx) интерпретирует события как последовательности данных, поступающих с течением времени. Соответственно Rx можно рассматривать как LINQ to Events (на базе `IObservable<T>`). Главное различие между наблюдаемыми объектами и другими провайдерами LINQ заключается в том, что Rx использует модель проталкивания, т. е. запрос определяет, как программа реагирует при поступлении событий. Rx строится на базе LINQ и добавляет новые мощные операторы как методы расширения.

В этой главе рассматриваются более типичные операции Rx. Помните, что все операторы LINQ тоже доступны, так что простые операции — фильтрация (`Where`), проекция (`Select`) и т. д. — на концептуальном уровне работают так же, как и с любым другим провайдером LINQ. Эти распространенные операции LINQ здесь не рассматриваются; мы сосредоточимся на новых возможностях, которые Rx добавляет к LINQ, особенно предназначенным для работы со *временем*.

Чтобы использовать System.Reactive, установите NuGet-пакет для System.Reactive в своем приложении.

6.1. Преобразование событий .NET

Задача

Имеется событие, которое должно интерпретироваться как входной поток System.Reactive, генерирующий данные через `OnNext` при каждом инициировании события.

Решение

Класс `Observable` определяет несколько преобразователей событий. Большинство событий фреймворка .NET совместимо с `FromEventPattern`, но, если ваши события не соответствуют общей схеме, используйте `FromEvent`.

`FromEventPattern` лучше всего работает с типом делегата события `EventHandler<T>`. Этот тип делегата события используется во многих более новых фреймворках. Например, тип `Progress<T>` определяет событие `ProgressChanged` с типом `EventHandler<T>`, что позволяет легко упаковать его в `FromEventPattern`:

```
var progress = new Progress<int>();
IObservable<EventPattern<int>> progressReports =
    Observable.FromEventPattern<int>(
        handler => progress.ProgressChanged += handler,
        handler => progress.ProgressChanged -= handler);
progressReports.Subscribe(data => Trace.WriteLine("OnNext:
   " + data.EventArgs));
```

Отмечу, что `data.EventArgs` сильно типизован с типом `int`. Аргумент-тип `FromEventPattern` (`int` в приведенном примере) совпадает с типом `T` в `EventHandler<T>`. Два лямбда-аргумента `FromEventPattern` позволяют System.Reactive подписываться и отменять подписку на событие.

Более новые фреймворки пользовательского интерфейса используют `EventHandler<T>`, что позволяет легко использовать их из `FromEventPattern`, но более старые типы часто определяют уникальный тип делегата для каждого события. Они также могут использоваться с `FromEventPattern`, но это потребует несколько большей работы. Например, тип `System.Timers.Timer` определяет событие `Elapsed`, относящееся к типу `ElapsedEventHandler`. Подобные старые события можно упаковать в `FromEventPattern`:

```
var timer = new System.Timers.Timer(interval: 1000) { Enabled = true };
IObservable<EventPattern<ElapsedEventArgs>> ticks =
    Observable.FromEventPattern<ElapsedEventHandler, ElapsedEventArgs>(
        handler => (s, a) => handler(s, a),
        handler => timer.Elapsed += handler,
        handler => timer.Elapsed -= handler);
ticks.Subscribe(data => Trace.WriteLine("OnNext:
   " + data.EventArgs.SignalTime));
```

Обратите внимание: в этом примере `data.EventArgs` также имеет сильную типизацию. Аргументы-типы `FromEventPattern` теперь содержат уникальный тип обработчика и производный тип `EventArgs`. Первый лямбда-аргумент `FromEventPattern` содержит преобразователь `EventHandler<ElapsedEventArgs>` в `ElapsedEventHandler`; преобразователь не делает ничего, кроме простой передачи события.

Синтаксис определенно становится неудобным. Ниже приведен другой вариант, использующий *отражение* (reflection):

```
var timer = new System.Timers.Timer(interval: 1000) { Enabled = true };
IObservable<EventPattern<object>> ticks =
    Observable.FromEventPattern(timer, nameof(Timer.Elapsed));
ticks.Subscribe(data => Trace.WriteLine("OnNext: "
    + ((ElapsedEventArgs)data.EventArgs).SignalTime));
```

При таком подходе вызов `FromEventPattern` выглядит намного проще. При этом у него есть один недостаток: потребитель не получает данные с сильной типизацией. Так как `data.EventArgs` относится к типу `object`, вам придется преобразовать его в `ElapsedEventArgs` самостоятельно.

Пояснение

События — распространенный источник данных для потоков System.Reactive. В этом рецепте рассматривается упаковка любых событий, соответствующих стандартной схеме события (в первом аргументе содержится отправитель, во втором — тип аргументов события). Даже если вы используете необычные типы событий, вы можете использовать перегруженные версии метода `Observable.FromEvent`, чтобы упаковать их в наблюдаемый объект.

Когда события упаковываются в наблюдаемый объект, `OnNext` вызывается при каждом инициировании события. Когда вы имеете дело

с AsyncCompletedEventArgs, это может привести к неожиданному поведению, потому что любое исключение передается как данные (OnNext), а не как ошибка (OnError). Например, рассмотрим следующую обертку для WebClient.DownloadStringCompleted:

```
var client = new WebClient();
IObservable<EventPattern<object>> downloadedStrings =
    Observable.
    FromEventPattern(client, nameof(WebClient.DownloadStringCompleted));
downloadedStrings.Subscribe(
    data =>
    {
      var eventArgs = (DownloadStringCompletedEventArgs)data.EventArgs;
      if (eventArgs.Error != null)
        Trace.WriteLine("OnNext: (Error) " + eventArgs.Error);
      else
        Trace.WriteLine("OnNext: " + eventArgs.Result);
    },
    ex => Trace.WriteLine("OnError: " + ex.ToString()),
    () => Trace.WriteLine("OnCompleted"));
client.DownloadStringAsync(new Uri("http://invalid.example.com/"));
```

Когда WebClient.DownloadStringAsync завершается с ошибкой, инициируется событие с исключением в AsyncCompletedEventArgs.Error. К сожалению, System.Reactive воспринимает его как событие данных, так что при выполнении приведенного кода будет выведено сообщение OnNext: (Error) вместо OnError:.

Некоторые подписки и отмены подписки на события должны выполняться из определенного контекста. Например, подписка на события многих UI-элементов должна выполняться из UI-потока. System.Reactive предоставляет оператор для управления контекстом создания и отмены подписки: SubscribeOn. В большинстве случаев без оператора SubscribeOn можно обойтись, потому что обычно подписки, относящиеся к UI, создаются из UI-потока.

SubscribeOn управляет контекстом кода, в котором добавляются и удаляются обработчики событий. Не путайте с оператором ObserveOn, управляющим контекстом для уведомлений наблюдаемого объекта (делегатов, передаваемых Subscribe).

Дополнительная информация

В рецепте 6.2 рассматривается изменение контекста, в котором инициируются события.

В рецепте 6.4 рассматривается регулировка событий для предотвращения перегрузки.

6.2. Отправка уведомлений контексту

Задача

System.Reactive старается действовать по возможности потоково-нейтрально. Таким образом, уведомления (например, `OnNext`) будут выдаваться в том потоке, который окажется текущим. Все уведомления `OnNext` происходят последовательно, но небязательно в одном потоке.

Часто бывает нужно, чтобы эти уведомления выдавались в конкретный контекст. Например, все манипуляции с UI-элементами должны осуществляться в UI-потоке, которому принадлежат эти элементы, поэтому если вы обновляете пользовательский интерфейс в ответ на уведомление, поступившее в поток из пула потоков, необходимо перейти к UI-потоку.

Решение

System.Reactive предоставляет оператор `ObserveOn` для перемещения уведомлений к другому планировщику.

В следующем примере оператор `Interval` используется для создания уведомлений `OnNext` один раз в секунду:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  Trace.WriteLine($"UI thread is {Environment.CurrentManagedThreadId}");
  Observable.Interval(TimeSpan.FromSeconds(1))
       .Subscribe(x => Trace.WriteLine(
          $"Interval {x} on thread {Environment.CurrentManagedThreadId}"));
}
```

На моей машине вывод выглядит примерно так:

```
UI thread is 9
Interval 0 on thread 10
Interval 1 on thread 10
Interval 2 on thread 11
Interval 3 on thread 11
Interval 4 on thread 10
Interval 5 on thread 11
Interval 6 on thread 11
```

Так как `Interval` работает по таймеру (без привязки к конкретному потоку), уведомления будут выдаваться в потоке из пула потоков, а не в UI-потоке. Если вам потребуется обновить UI-элемент, вы можете направить эти уведомления через `ObserveOn` и передать контекст синхронизации, представляющий UI-поток:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  SynchronizationContext uiContext = SynchronizationContext.Current;
  Trace.WriteLine($"UI thread is {Environment.CurrentManagedThreadId}");
  Observable.Interval(TimeSpan.FromSeconds(1))
      .ObserveOn(uiContext)
      .Subscribe(x => Trace.WriteLine(
         $"Interval {x} on thread {Environment.CurrentManagedThreadId}"));
}
```

Также `ObserveOn` часто используется для выхода из UI-потока в случае необходимости. Допустим, нужно выполнять некоторые вычисления, создающие высокую нагрузку на процессор, при каждом перемещении мыши. По умолчанию все события перемещения мыши инициируются в UI-потоке, и с помощью `ObserveOn` вы можете переместить эти уведомления в поток из пула потоков, выполнить вычисления, а затем переместить уведомления о результатах обратно в UI-поток:

```
SynchronizationContext uiContext = SynchronizationContext.Current;
Trace.WriteLine($"UI thread is {Environment.CurrentManagedThreadId}");
Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
        handler => (s, a) => handler(s, a),
        handler => MouseMove += handler,
```

```
    handler => MouseMove -= handler)
.Select(evt => evt.EventArgs.GetPosition(this))
.ObserveOn(Scheduler.Default)
.Select(position =>
{
  // Сложные вычисления
  Thread.Sleep(100);
  var result = position.X + position.Y;
  var thread = Environment.CurrentManagedThreadId;
  Trace.WriteLine($"Calculated result {result} on thread {thread}");
  return result;
})
.ObserveOn(uiContext)
.Subscribe(x => Trace.WriteLine(
    $"Result {x} on thread {Environment.CurrentManagedThreadId}"));
```

Выполнив этот пример, вы увидите, что вычисления выполняются в потоке из пула потоков, а результаты выводятся в UI-потоке. Вы также заметите, что вычисления и результаты отстают от ввода; они помещаются в очередь, потому что местоположение мыши обновляется чаще, чем каждые 100 мс. В System.Reactive предусмотрено несколько возможных решений задачи; одно из них — регулировка ввода — рассматривается в рецепте 6.4.

Пояснение

В действительности `ObserveOn` перемещает уведомления *планировщику* System.Reactive. В этом рецепте рассматриваются планировщик по умолчанию (для пула потоков) и один способ создания UI-планировщика. Оператор `ObserveOn` чаще всего используется для перемещения в UI-поток и из него, но планировщики также могут пригодиться во многих других сценариях. Более сложный сценарий, в котором планировщики могут принести пользу, связан с имитацией прохождения времени при модульном тестировании (рассматривается в рецепте 7.6).

Оператор `ObserveOn` управляет контекстом для наблюдаемых уведомлений. Не путайте его с оператором `SubscribeOn`, который управляет контекстом для кода, добавляющего и удаляющего обработчики событий.

Дополнительная информация

В рецепте 6.1 рассматривается создание последовательностей на базе событий и использование `SubscribeOn`.

В рецепте 6.4 рассматривается регулировка потоков событий.

В рецепте 7.6 рассматривается специальный планировщик, используемый при тестировании кода System.Reactive.

6.3. Группировка данных событий с использованием Window и Buffer

Задача

Имеется последовательность событий, требуется группировать входящие события по мере их поступления. Другой пример: вы хотите реагировать на пары событий или на весь ввод в пределах двухсекундного окна.

Решение

System.Reactive предоставляет пару операторов для группировки входных последовательностей: `Buffer` и `Window`. `Buffer` сохраняет входные события до завершения группы, после чего передает их все сразу как коллекцию событий. `Window` логически группирует входные события, но передает их по мере поступления. Возвращаемым типом `Buffer` является `IObservable<IList<T>>` (поток событий коллекций), а возвращаемым типом `Window` — `IObservable<IObservable<T>>` (поток событий потоков событий).

В следующем примере оператор `Interval` используется для создания ежесекундных уведомлений `OnNext`, после чего буферизует их по два:

```
Observable.Interval(TimeSpan.FromSeconds(1))
    .Buffer(2)
    .Subscribe(x => Trace.WriteLine(
        $"{DateTime.Now.Second}: Got {x[0]} and {x[1]}"));
```

На моей машине этот код генерирует парный вывод каждые две секунды:

```
13: Got 0 and 1
15: Got 2 and 3
17: Got 4 and 5
19: Got 6 and 7
21: Got 8 and 9
```

В следующем примере `Window` используется для создания групп из двух событий:

```
Observable.Interval(TimeSpan.FromSeconds(1))
    .Window(2)
    .Subscribe(group =>
    {
      Trace.WriteLine($"{DateTime.Now.Second}: Starting new group");
      group.Subscribe(
          x => Trace.WriteLine($"{DateTime.Now.Second}: Saw {x}"),
         () => Trace.WriteLine($"{DateTime.Now.Second}: Ending group"));
    });
```

На моем компьютере этот пример с `Window` выдает следующий результат:

```
17: Starting new group
18: Saw 0
19: Saw 1
19: Ending group
19: Starting new group
20: Saw 2
21: Saw 3
21: Ending group
21: Starting new group
22: Saw 4
23: Saw 5
23: Ending group
23: Starting new group
```

Эти примеры показывают различия между `Buffer` и `Window`. `Buffer` ожидает всех событий в своей группе, а затем публикует одну коллекцию. `Window` группирует события аналогичным образом, но публикует события по

мере поступления; `Window` немедленно публикует наблюдаемый объект, который публикует события для этого окна.

Как `Buffer`, так и `Window` работают с временными интервалами. В следующем примере все события перемещения мыши собираются в окнах продолжительностью в 1 секунду:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
          handler => (s, a) => handler(s, a),
          handler => MouseMove += handler,
          handler => MouseMove -= handler)
      .Buffer(TimeSpan.FromSeconds(1))
      .Subscribe(x => Trace.WriteLine(
          $"{DateTime.Now.Second}: Saw {x.Count} items."));
}
```

В зависимости от того, как перемещается мышь, на экране должен появиться вывод следующего вида:

```
49: Saw 93 items.
50: Saw 98 items.
51: Saw 39 items.
52: Saw 0 items.
53: Saw 4 items.
54: Saw 0 items.
55: Saw 58 items.
```

Пояснение

`Buffer` и `Window` входят в число инструментов для подготовки ввода и придания ему нужной формы. Другой полезный прием — регулировка — рассматривается в рецепте 6.4.

У `Buffer` и `Window` существуют другие перегруженные версии, которые могут использоваться в более сложных сценариях. Перегруженные версии с параметрами `skip` и `timeShift` позволяют создавать группы, перекрывающиеся с другими группами, или пропускать элементы между группами. Также есть перегруженные версии, получающие делегатов, что позволяет динамически определять границы групп.

Дополнительная информация

В рецепте 6.1 рассматривается создание последовательностей из событий.

В рецепте 6.4 рассматривается регулировка потоков событий.

6.4. Контроль потоков событий посредством регулировки и выборки

Задача

Типичная проблема при написании реактивного кода — слишком быстрое поступление событий. Слишком быстрый поток событий может превысить возможности вашей программы.

Решение

System.Reactive предоставляет операторы, предназначенные специально для предотвращения «затопления» данными событий. Операторы `Throttle` и `Sample` предоставляют два разных способа контроля над быстро поступающими событиями ввода.

Оператор `Throttle` устанавливает скользящее окно тайм-аута. При поступлении входного события окно тайм-аута сбрасывается. По истечении окна тайм-аута публикуется значение последнего события, поступившего в границах окна.

Следующий пример отслеживает движения мыши и использует `Throttle`, чтобы сообщения об обновлениях выдавались только в том случае, если мышь оставалась неподвижной в течение секунды:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
          handler => (s, a) => handler(s, a),
          handler => MouseMove += handler,
          handler => MouseMove -= handler)
      .Select(x => x.EventArgs.GetPosition(this))
      .Throttle(TimeSpan.FromSeconds(1))
```

```
      .Subscribe(x => Trace.WriteLine(
          $"{DateTime.Now.Second}: Saw {x.X + x.Y}"));
}
```

Вывод серьезно изменяется в зависимости от перемещений мыши, но один из результатов запуска на моем компьютере выглядел так:

```
47: Saw 139
49: Saw 137
51: Saw 424
56: Saw 226
```

Например, `Throttle` часто используется при автозаполнении: пользователь вводит текст в текстовом поле, и поиск должен начаться только после того, как пользователь завершит ввод.

`Sample` использует другой подход к контролю быстрых последовательностей. `Sample` устанавливает регулярный тайм-аут и публикует последнее значение в этом окне при каждом истечении тайм-аута. Если в период выборки не было получено ни одного значения, то за этот период результаты не публикуются.

Следующий пример отслеживает перемещения мыши и осуществляет их выборку с секундными интервалами. В отличие от примера с `Throttle`, в примере с `Sample` для просмотра данных не нужно держать мышь неподвижной в течение секунды:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
          handler => (s, a) => handler(s, a),
          handler => MouseMove += handler,
          handler => MouseMove -= handler)
      .Select(x => x.EventArgs.GetPosition(this))
      .Sample(TimeSpan.FromSeconds(1))
      .Subscribe(x => Trace.WriteLine(
          $"{DateTime.Now.Second}: Saw {x.X + x.Y}"));
}
```

Вот как выглядел результат на моей машине, когда я сначала оставил мышь неподвижной на несколько секунд, а затем непрерывно двигал ее:

```
12: Saw 311
17: Saw 254
```

```
18: Saw 269
19: Saw 342
20: Saw 224
21: Saw 277
```

Пояснение

Регулировка и выборка — основные инструменты контроля чрезмерного потока ввода. Не забывайте, что вы можете легко выполнить фильтрацию стандартным оператором LINQ `Where`. Можно считать, что операторы `Throttle` и `Sample` похожи на `Where`, только они выполняют фильтрацию по временным окнам, а не по данным событий. Все три оператора помогают взять под контроль слишком быстрые входные потоки, но делают это разными способами.

Дополнительная информация

В рецепте 6.1 рассматривается создание последовательностей из событий.

В рецепте 6.2 рассматривается изменение контекста для выдачи событий.

6.5. Тайм-ауты

Задача

Ожидается, что событие поступит в течение определенного времени. Требуется обеспечить своевременную реакцию программы даже в том случае, если событие не поступит. Чаще всего подобные ожидаемые события являются одиночными асинхронными операциями (например, ожиданием ответа на запрос веб-службы).

Решение

Оператор `Timeout` устанавливает скользящее окно тайм-аута в своем входном потоке. При каждом поступлении нового события окно тайм-аута сбрасывается. Если тайм-аут истекает без получения события в окне, оператор `Timeout` завершает поток с уведомлением `OnError`, содержащим исключение `TimeoutException`.

Следующий пример выдает веб-запрос к условному домену и устанавливает тайм-аут продолжительностью в 1 секунду. Чтобы запустить веб-запрос, в коде используется `ToObservable` для преобразования `Task<T>` в `IObservable<T>` (см. рецепт 8.6):

```
void GetWithTimeout(HttpClient client)
{
  client.GetStringAsync("http://www.example.com/").ToObservable()
      .Timeout(TimeSpan.FromSeconds(1))
      .Subscribe(
         x => Trace.WriteLine($"{DateTime.Now.Second}: Saw {x.Length}"),
          ex => Trace.WriteLine(ex));
}
```

`Timeout` идеально подходит для асинхронных операций (таких, как веб-запросы), но может применяться к любому потоку событий. В следующем примере `Timeout` применяется к перемещениям мыши, с которыми проще экспериментировать:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
         handler => (s, a) => handler(s, a),
         handler => MouseMove += handler,
         handler => MouseMove -= handler)
      .Select(x => x.EventArgs.GetPosition(this))
      .Timeout(TimeSpan.FromSeconds(1))
      .Subscribe(
         x => Trace.WriteLine($"{DateTime.Now.Second}: Saw {x.X + x.Y}"),
         ex => Trace.WriteLine(ex));
}
```

На своей машине я переместил мышь, а потом держал ее неподвижной в течение секунды, получив следующие результаты:

```
16: Saw 180
16: Saw 178
16: Saw 177
16: Saw 176
System.TimeoutException: The operation has timed out.
```

Учтите, что при отправке `OnError` исключения `TimeoutException` поток завершается и перемещения мыши перестают проходить. Возможно, такое

поведение окажется нежелательным, поэтому у оператора `Timeout` существуют перегруженные версии, которые подставляют другой поток при возникновении тайм-аута вместо того, чтобы завершать поток с исключением.

Код следующего примера отслеживает перемещения мыши до возникновения тайм-аута. После тайм-аута код начинает отслеживать щелчки мышью:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  IObservable<Point> clicks =
      Observable.FromEventPattern<MouseButtonEventHandler,
        MouseButtonEventArgs>(
          handler => (s, a) => handler(s, a),
          handler => MouseDown += handler,
          handler => MouseDown -= handler)
      .Select(x => x.EventArgs.GetPosition(this));
  Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
          handler => (s, a) => handler(s, a),
          handler => MouseMove += handler,
          handler => MouseMove -= handler)
      .Select(x => x.EventArgs.GetPosition(this))
      .Timeout(TimeSpan.FromSeconds(1), clicks)
      .Subscribe(
          x => Trace.WriteLine($"{DateTime.Now.Second}: Saw
             {x.X},{x.Y}"),
          ex => Trace.WriteLine(ex));
}
```

На своей машине я переместил мышь, потом держал ее неподвижной в течение секунды, а затем щелкнул в паре разных точек. Из следующих результатов видно, что до наступления тайм-аута мышь быстро перемещалась, после чего последовали два щелчка:

```
49: Saw 95,39
49: Saw 94,39
49: Saw 94,38
49: Saw 94,37
53: Saw 130,141
55: Saw 469,4
```

Пояснение

Оператор `Timeout` играет важную роль в нетривиальных приложениях, потому что в идеале ваша программа всегда должна реагировать на происходящее, что бы ни происходило вокруг. Он особенно полезен при выполнении асинхронных операций, но также может быть применен к любому потоку событий. Обратите внимание: используемая операция не отменяется; в случае тайм-аута она продолжит выполнение до тех пор, пока не завершится успехом или неудачей.

Дополнительная информация

В рецепте 6.1 рассматривается создание последовательностей на базе событий.

В рецепте 8.6 рассматривается упаковка асинхронного кода как наблюдаемого потока событий.

В рецепте 10.6 рассматривается отмена подписки на последовательности в результате `CancellationToken`.

В рецепте 10.3 рассматривается использование `CancellationToken` для тайм-аута.

ГЛАВА 7

Тестирование

Тестирование — важнейший фактор качества программного обеспечения. В последние годы модульное тестирование получило повсеместное распространение; такое впечатление, что о нем говорят и пишут буквально повсюду. Многие разработчики высказываются в пользу *разработки через тестирование* — стиля программирования, который гарантирует наличие исчерпывающих тестов при завершении приложения. Преимущества модульного тестирования для качества кода и общего времени разработки хорошо известны, и все же многие программисты еще не пишут модульные тесты.

Рекомендую писать хотя бы несколько модульных тестов. Начните с кода, в котором вы уверены меньше всего. По моему опыту, модульные тесты обладают двумя основными преимуществами:

- **Лучшее понимание кода.** Вы знаете, что часть приложения работает, но понятия не имеете, как? Эта мысль остается где-то на заднем плане, но потом приходит крайне странное сообщение об ошибке. Написание модульных тестов для тех частей кода, которые вам кажутся сложными, — прекрасный способ разобраться в том, как они работают. После написания модульных тестов, описывающих его поведение, код перестает быть загадочным; у вас появляется набор модульных тестов, описывающих его поведение и зависимости от других частей кода.
- **Уверенность при внесении изменений.** Рано или поздно вы получите запрос на реализацию новой функции, для которого придется изменить пугающий код, и уже не удастся сделать вид, что его не существует (я знаю, каково это; со мной такое тоже случалось). Лучше действовать на опережение: напишите модульные тесты для неприятного кода, пока не пришел запрос на новую функциональность. После того как

тесты будут готовы, у вас появится система раннего предупреждения, которая немедленно оповестит вас об изменениях, нарушающих существующее поведение. А если вы собираетесь включить новый код в проект, модульные тесты придадут больше уверенности в том, что изменения в коде не нарушат существующего поведения.

Оба преимущества относятся не только к вашему коду, но и к коду других разработчиков. Уверен, что есть и другие преимущества. Сокращает ли модульное тестирование частоту ошибок? Скорее всего. Сокращает ли модульное тестирование общее время работы над проектом? Возможно. Но преимущества, описанные мною, являются бесспорными; я наблюдаю их каждый раз, когда пишу модульные тесты. В общем, так я пропагандирую модульное тестирование.

Все рецепты, содержащиеся в этой главе, относятся к тестированию. Многие разработчики (даже те, которые обычно пишут модульные тесты) шарахаются от тестирования конкурентного кода, так как считают, что это слишком сложно. Но как покажут эти рецепты, модульное тестирование конкурентного кода далеко не так сложно, как они полагают. Разработчики современных инструментов и библиотек (таких, как `async` и System.Reactive) уделили значительное внимание тестированию, и это проявляется на практике. Рекомендую использовать эти рецепты для написания модульных тестов, особенно если вы недавно имеете дело с конкурентностью (т. е. новый конкурентный код кажется слишком сложным или пугающим).

7.1. Модульное тестирование async-методов

Задача

Имеется `async`-метод, для которого необходимо провести модульное тестирование.

Решение

Многие современные фреймворки модульного тестирования — включая MSTest, NUnit и xUnit — поддерживают методы модульного тестирования `async Task`. В MSTest поддержка этих тестов появилась в Visual Studio 2012.

Если вы используете другой фреймворк модульного тестирования, возможно, вам придется перейти на последнюю версию.

Пример `async`-модульного теста в MSTest:

```
[TestMethod]
public async Task MyMethodAsync_ReturnsFalse()
{
  var objectUnderTest = ...;
  bool result = await objectUnderTest.MyMethodAsync();
  Assert.IsFalse(result);
}
```

Фреймворк модульного тестирования замечает, что метод имеет возвращаемый тип `Task`, и ожидает завершения задачи перед тем, как сделать отметку о прохождении или отказе теста.

Если ваш фреймворк модульного тестирования не поддерживает модульные тесты `async Task`, то ему придется помочь с ожиданием тестируемой асинхронной операции. Один из вариантов — использовать `GetAwaiter().GetResult()` для синхронного блокирования по задаче; если после этого использовать `GetAwaiter().GetResult()` вместо `Wait()`, это позволит избежать обертки `AggregateException`, когда в задаче произойдет исключение. Однако я предпочитаю использовать тип `AsyncContext` из NuGet-пакета `Nito.AsyncEx`:

```
[TestMethod]
public void MyMethodAsync_ReturnsFalse()
{
  AsyncContext.Run(async () =>
  {
    var objectUnderTest = ...;
    bool result = await objectUnderTest.MyMethodAsync();
    Assert.IsFalse(result);
  });
}
```

`AsyncContext.Run` ожидает завершения всех асинхронных методов.

Пояснение

Имитация (mocking) асинхронных зависимостей на первый взгляд кажется немного неуклюжей. Всегда желательно хотя бы проверить, как ваши

методы реагируют на синхронный успех (имитация с `Task.FromResult`), синхронные ошибки (имитация с `Task.FromException`) и асинхронный успех (имитация с `Task.Yield` и возвращаемым значением). `Task.FromResult` и `Task.FromException` рассматриваются в рецепте 2.2. `Task.Yield` может использоваться для принудительного применения асинхронного поведения и задействуется прежде всего при модульном тестировании:

```
interface IMyInterface
{
  Task<int> SomethingAsync();
}

class SynchronousSuccess : IMyInterface
{
  public Task<int> SomethingAsync()
  {
    return Task.FromResult(13);
  }
}

class SynchronousError : IMyInterface
{
  public Task<int> SomethingAsync()
  {
    return Task.FromException<int>(new InvalidOperationException());
  }
}

class AsynchronousSuccess : IMyInterface
{
  public async Task<int> SomethingAsync()
  {
    await Task.Yield(); // Принудительно включить асинхронное поведение.
    return 13;
  }
}
```

При тестировании асинхронного кода взаимоблокировки и состояния гонки могут проявляться чаще, чем при тестировании синхронного кода. Я считаю полезным назначение тайм-аута на уровне тестов; в Visual Studio можно добавить в решение файл тестовых настроек, в котором можно

задавать тайм-ауты для отдельных тестов. Значение по умолчанию достаточно велико; обычно я использую двухсекундный тайм-аут уровня тестов.

Тип `AsyncContext` находится в пакете `Nito.AsyncEx`.

Дополнительная информация

В рецепте 7.2 рассматриваются асинхронные методы модульного тестирования, которые должны провалиться.

7.2. Асинхронные методы модульного тестирования, которые не должны проходить

Задача

Требуется написать модульный тест, который проверяет конкретный отказ метода `async Task`.

Решение

Если вы занимаетесь разработкой для настольных компьютеров или серверов, MSTest поддерживает тестирование на отказ с помощью обычного класса `ExpectedExceptionAttribute`:

```
// Использовать это решение не рекомендуется; см. ниже.
[TestMethod]
[ExpectedException(typeof(DivideByZeroException))]
public async Task Divide_WhenDenominatorIsZero_ThrowsDivideByZero()
{
  await MyClass.DivideAsync(4, 0);
}
```

Тем не менее это не лучшее решение: `ExpectedException` обычно является признаком плохого дизайна. Ожидаемое исключение может быть выдано любым из методов, вызванных вашим методом модульного тестирования.

Более качественный код проверяет, что исключение было выдано конкретным фрагментом кода, а не модульным тестом в целом.

Многие современные фреймворки модульного тестирования включают `Assert.ThrowsAsync<TException>` в той или иной форме. Например, `ThrowsAsync` из xUnit можно использовать так:

```
[Fact]
public async Task Divide_WhenDenominatorIsZero_ThrowsDivideByZero()
{
  await Assert.ThrowsAsync<DivideByZeroException>(async () =>
  {
    await MyClass.DivideAsync(4, 0);
  });
}
```

Не забудьте использовать `await` с задачей, возвращенной `ThrowsAsync`! Ключевое слово await распространяет все обнаруженные нарушения тестовых условий. Если вы забудете `await` и проигнорируете предупреждение компилятора, ваш модульный тест всегда будет проходить успешно независимо от поведения метода.

К сожалению, некоторые фреймворки модульного тестирования не содержат эквивалентных `async`-совместимых конструкций `ThrowsAsync`. Если вы окажетесь в такой ситуации, создайте собственный аналог:

```
/// <summary>
/// Гарантирует, что асинхронный делегат выдает исключение.
/// </summary>
/// <typeparam name="TException">
/// Тип ожидаемого исключения.
/// </typeparam>
/// <param name="action">Асинхронный делегат для тестирования.</param>
/// <param name="allowDerivedTypes">
/// Должны ли приниматься производные типы.
/// </param>
public static async Task<TException> ThrowsAsync<TException>(Func<Task>
   action,
    bool allowDerivedTypes = true)
    where TException : Exception
{
```

```
  try
  {
    await action();
    var name = typeof(Exception).Name;
    Assert.Fail($"Delegate did not throw expected exception {name}.");
    return null;
  }
  catch (Exception ex)
  {
    if (allowDerivedTypes && !(ex is TException))
      Assert.Fail($"Delegate threw exception of type
        {ex.GetType().Name}" +
          $", but {typeof(TException).Name} or a derived type was
             expected.");
    if (!allowDerivedTypes && ex.GetType() != typeof(TException))
      Assert.Fail($"Delegate threw exception of type
        {ex.GetType().Name}" +
          $", but {typeof(TException).Name} was expected.");
    return (TException)ex;
  }
}
```

Метод можно использовать точно так же, как и любой другой метод `Assert.ThrowsAsync<TException>`. Не забудьте использовать `await` с возвращаемым значением!

Пояснение

Тестирование обработки ошибок не менее важно, чем тестирование успешных сценариев. Пожалуй, оно даже важнее, потому что успешные сценарии всегда проверяются перед выпуском программного продукта. Если приложение ведет себя странно, это происходит из-за непредвиденной ошибочной ситуации.

Но я рекомендую разработчикам воздерживаться от `ExpectedException`. Лучше протестировать выдачу исключения в конкретной точке вместо того, чтобы тестировать исключение в любой момент во время теста. Вместо `ExpectedException` используйте либо `ThrowsAsync` (или его аналог в вашем фреймворке модульного тестирования), либо его реализацию, как в последнем примере кода.

Дополнительная информация

В рецепте 7.1 рассматриваются основы модульного тестирования асинхронных методов.

7.3. Модульное тестирование методов async void

Задача

Имеется метод `async void`, для которого необходимо написать модульные тесты.

Решение

Стоп.

Такой ситуации нужно избегать всеми силами. Если метод `async void` можно преобразовать в метод `async Task` — сделайте это.

Если ваш метод *обязан* быть методом `asyn void` (например, для соответствия сигнатуре метода интерфейса), рассмотрите возможность написания двух методов: метода `async Task`, содержащего всю логику, и обертки `async void`, которая просто вызывает метод `async Task` и ожидает результата. Метод `async void` удовлетворяет требованиям архитектуры, тогда как метод `async Task` (со всей логикой) пригоден для тестирования.

Если изменить метод невозможно и вы *вынуждены* заниматься модульным тестированием метода `async void`, это тоже возможно. Используйте класс `AsyncContext` из библиотеки `Nito.AsyncEx`:

```
// Не рекомендуется; см. далее в этом разделе.
[TestMethod]
public void MyMethodAsync_DoesNotThrow()
{
  AsyncContext.Run(() =>
  {
    var objectUnderTest = new Sut(); // ...;
    objectUnderTest.MyVoidMethodAsync();
  });
}
```

Тип `AsyncContext` ожидает завершения всех асинхронных операций (включая методы `async void`) и распространяет выданные ими исключения.

Тип `AsyncContext` находится в пакете `Nito.AsyncEx`.

Пояснение

Одно из важнейших правил `async`-кода — по возможности избегать `async void`. Настоятельно рекомендую провести рефакторинг кода, а не использовать `AsyncContext` для модульного тестирования методов `async void`.

Дополнительная информация

В рецепте 7.1 рассматривается модульное тестирование методов `async Task`.

7.4. Модульное тестирование сетей потоков данных

Задача

В приложении существует сеть потока данных. Требуется убедиться в правильности ее работы.

Решение

Сети потоков данных независимы; они имеют собственный срок жизни и асинхронны по своей природе. Таким образом, самый простой подход к их тестированию — асинхронные модульные тесты. Следующий модульный тест проверяет нестандартный блок потока данных из рецепта 5.6:

```
[TestMethod]
public async Task MyCustomBlock_AddsOneToDataItems()
{
  var myCustomBlock = CreateMyCustomBlock();
  myCustomBlock.Post(3);
  myCustomBlock.Post(13);
```

```
  myCustomBlock.Complete();
  Assert.AreEqual(4, myCustomBlock.Receive());
  Assert.AreEqual(14, myCustomBlock.Receive());
  await myCustomBlock.Completion;
}
```

К сожалению, модульное тестирование отказов не столь прямолинейно. Дело в том, что исключения в сетях потоков данных упаковываются в новую обертку `AggregateException` каждый раз, когда они распространяются в следующий блок. В следующем примере используется вспомогательный метод для проверки того, что исключение отбрасывает данные и распространяется через нестандартный блок:

```
[TestMethod]
public async Task MyCustomBlock_Fault_DiscardsDataAndFaults()
{
  var myCustomBlock = CreateMyCustomBlock();
  myCustomBlock.Post(3);
  myCustomBlock.Post(13);
  (myCustomBlock as IDataflowBlock).Fault(new
     InvalidOperationException());
  try
  {
    await myCustomBlock.Completion;
  }
  catch (AggregateException ex)
  {
    AssertExceptionIs<InvalidOperationException>(
        ex.Flatten().InnerException, false);
  }
}

public static void AssertExceptionIs<TException>(Exception ex,
    bool allowDerivedTypes = true)
{
  if (allowDerivedTypes && !(ex is TException))
    Assert.Fail($"Exception is of type {ex.GetType().Name}, but " +
        $"{typeof(TException).Name} or a derived type was expected.");
  if (!allowDerivedTypes && ex.GetType() != typeof(TException))
    Assert.Fail($"Exception is of type {ex.GetType().Name}, but " +
        $"{typeof(TException).Name} was expected.");
}
```

Пояснение

Прямое модульное тестирование сетей потоков данных возможно, но несколько неудобно. Если ваша сеть является частью большего компонента, возможно, будет проще организовать модульное тестирование для большего компонента (тогда сеть будет тестироваться неявно). Но если вы разрабатываете нестандартный блок или сеть для повторного использования, используйте модульные тесты вроде описанных в этом рецепте.

Дополнительная информация

В рецепте 7.1 рассматривается модульное тестирование `async`-методов.

7.5. Модульное тестирование наблюдаемых объектов System.Reactive

Задача

Часть вашей программы использует `IObservable<T>`. Требуется организовать модульное тестирование этой части.

Решение

System.Reactive содержит операторы, генерирующие последовательности (например, `Return`), а также другие операторы, которые могут преобразовать реактивную последовательность в обычную коллекцию или элемент (например, `SingleAsync`). Такие операторы, как `Return`, могут использоваться для создания заглушек для наблюдаемых зависимостей, а такие операторы, как `SingleAsync`, — для тестирования вывода.

Следующий код получает службу HTTP в качестве зависимости и применяет тайм-аут к вызову HTTP:

```
public interface IHttpService
{
  IObservable<string> GetString(string url);
}
```

```
public class MyTimeoutClass
{
  private readonly IHttpService _httpService;

  public MyTimeoutClass(IHttpService httpService)
  {
    _httpService = httpService;
  }

  public IObservable<string> GetStringWithTimeout(string url)
  {
    return _httpService.GetString(url)
        .Timeout(TimeSpan.FromSeconds(1));
  }
}
```

В данном случае тестируется класс `MyTimeoutClass`, который потребляет зависимость наблюдаемого объекта и генерирует наблюдаемый объект на выходе.

Оператор `Return` возвращает холодную последовательность (cold sequence), состоящую из одного элемента; он может использоваться для построения простых заглушек. Оператор `SingleAsync` возвращает объект `Task<T>`, завершаемый при поступлении следующего события. `SingleAsync` может использоваться в простых модульных тестах следующего вида:

```
class SuccessHttpServiceStub : IHttpService
{
  public IObservable<string> GetString(string url)
  {
    return Observable.Return("stub");
  }
}

[TestMethod]
public async Task MyTimeoutClass_SuccessfulGet_ReturnsResult()
{
  var stub = new SuccessHttpServiceStub();
  var my = new MyTimeoutClass(stub);
  var result = await my.GetStringWithTimeout("http://www.example.com/")
      .SingleAsync();
  Assert.AreEqual("stub", result);
}
```

В коде заглушек играет важную роль оператор `Throw`, который возвращает наблюдаемый объект, заканчивающийся с ошибкой. С помощью этого оператора можно писать модульные тесты и для ошибочных случаев. В следующем примере используется вспомогательный метод `ThrowsAsync` из рецепта 7.2:

```
private class FailureHttpServiceStub : IHttpService
{
  public IObservable<string> GetString(string url)
  {
    return Observable.Throw<string>(new HttpRequestException());
  }
}
[TestMethod]
public async Task MyTimeoutClass_FailedGet_PropagatesFailure()
{
  var stub = new FailureHttpServiceStub();
  var my = new MyTimeoutClass(stub);

  await ThrowsAsync<HttpRequestException>(async () =>
  {
    await my.GetStringWithTimeout("http://www.example.com/")
        .SingleAsync();
  });
}
```

Пояснение

`Return` и `Throw` хорошо подходят для создания наблюдаемых заглушек, а `SingleAsync` предоставляет простые средства тестирования наблюдаемых объектов с асинхронными модульными тестами. Эта комбинация неплохо справляется с простыми наблюдаемыми объектами, но когда вы начинаете работать со *временем*, их возможностей оказывается недостаточно. Например, если вы хотите протестировать функциональность тайм-аута `MyTimeoutClass`, модульным тестам придется ожидать нужный промежуток времени. Однако такое решение нельзя назвать качественным: оно снижает надежность модульных тестов за счет введения состояния гонки и плохо масштабируется с добавлением новых модульных тестов. В рецепте 7.6 рассматривается специальный механизм, с помощью которого System.Reactive позволяет создавать заглушки для самого времени.

Дополнительная информация

В разделе 7.1 рассматривается модульное тестирование `async`-методов, которое имеет много общего с модульными тестами, ожидающими `SingleAsync`.

В рецепте 7.6 рассматривается модульное тестирование наблюдаемых последовательностей, зависящих от прохождения времени.

7.6. Модульное тестирование наблюдаемых объектов System.Reactive с использованием имитации планирования

Задача

Имеется наблюдаемый объект, зависящий от времени. Требуется написать модульный тест, который не зависел бы от времени. К числу наблюдаемых объектов, зависящих от времени, относятся те, которые используют тайм-аут, окна/буферизацию и регулировку/выборку. Нужно провести модульное тестирование таких объектов, но так, чтобы модульные тесты выполнялись за приемлемое время.

Решение

Конечно, в модульные тесты можно включить задержку, но у такого решения есть два недостатка: 1) выполнение модульных тестов занимает больше времени; 2) возникает состояние гонки, потому что все модульные тесты выполняются одновременно, а последовательность выполнения становится непредсказуемой.

Библиотека System.Reactive (Rx) проектировалась с учетом потребностей тестирования; более того, сама библиотека Rx прошла тщательное модульное тестирование. Чтобы сделать возможным тщательное модульное тестирование, в Rx была введена концепция *планировщика*, и каждый оператор Rx, работающий со временем, реализуется с использованием этого абстрактного планировщика.

Чтобы ваши наблюдаемые объекты можно было тестировать, необходимо дать возможность вызывающей стороне задать планировщика. Например, можно взять класс `MyTimeoutClass` из рецепта 7.5 и добавить планировщика:

```
public interface IHttpService
{
  IObservable<string> GetString(string url);
}

public class MyTimeoutClass
{
  private readonly IHttpService _httpService;

  public MyTimeoutClass(IHttpService httpService)
  {
    _httpService = httpService;
  }

  public IObservable<string> GetStringWithTimeout(string url,
      IScheduler scheduler = null)
  {
    return _httpService.GetString(url)
        .Timeout(TimeSpan.FromSeconds(1), scheduler ??
           Scheduler.Default);
  }
}
```

Затем вы можете изменить заглушку службы HTTP, чтобы она также поддерживала планирование, и ввести переменную задержку:

```
private class SuccessHttpServiceStub : IHttpService
{
  public IScheduler Scheduler { get; set; }
  public TimeSpan Delay { get; set; }

  public IObservable<string> GetString(string url)
  {
    return Observable.Return("stub")
        .Delay(Delay, Scheduler);
  }
}
```

Теперь можно двигаться вперед и использовать `TestScheduler` — тип, включенный в библиотеку System.Reactive. `TestScheduler` предоставляет мощные средства управления (виртуальным) временем.

`TestScheduler` находится в пакете отдельно от остальных частей System.Reactive; необходимо установить пакет `NuGet Microsoft.Reactive.Testing`.

```
[TestMethod]
public void MyTimeoutClass_SuccessfulGetShortDelay_ReturnsResult()
{
  var scheduler = new TestScheduler();
  var stub = new SuccessHttpServiceStub
  {
    Scheduler = scheduler,
    Delay = TimeSpan.FromSeconds(0.5),
  };
  var my = new MyTimeoutClass(stub);
  string result = null;
  my.GetStringWithTimeout("http://www.example.com/", scheduler)
      .Subscribe(r => { result = r; });

  scheduler.Start();

  Assert.AreEqual("stub", result);
}
```

Этот код моделирует сетевую задержку продолжительностью 0,5 секунды. Важно отметить, что выполнение этого модульного теста не занимает 0,5 секунды; на моей машине он выполняется приблизительно за 70 миллисекунд. Полусекундная задержка существует только в виртуальном времени. Другое заметное отличие заключается в том, что этот модульный тест не является асинхронным; так как вы используете `TestScheduler`, все тесты могут завершаться немедленно.

После того как в коде будут использоваться тестовые планировщики, протестировать ситуации тайм-аута будет несложно:

```
[TestMethod]
public void MyTimeoutClass_SuccessfulGetLongDelay_
   ThrowsTimeoutException()
{
  var scheduler = new TestScheduler();
  var stub = new SuccessHttpServiceStub
  {
    Scheduler = scheduler,
    Delay = TimeSpan.FromSeconds(1.5),
  };
  var my = new MyTimeoutClass(stub);
  Exception result = null;

  my.GetStringWithTimeout("http://www.example.com/", scheduler)
      .Subscribe(_ => Assert.Fail("Received value"), ex => { result
          = ex; });

  scheduler.Start();

  Assert.IsInstanceOfType(result, typeof(TimeoutException));
}
```

И снова выполнение теста не занимает 1 секунду (или 1,5 секунды); тест выполняется немедленно с использованием виртуального времени.

Пояснение

В этом рецепте вы едва соприкоснулись с планировщиками System.Reactive и виртуальным временем. Рекомендую приступать к написанию модульных тестов тогда, когда вы начнете писать код System.Reactive; со временем ваш код будет становиться все более сложным, но вы будете уверены в том, что `Microsoft.Reactive.Testing` с ним справится.

`TestScheduler` также содержит методы `AdvanceTo` и `AdvanceBy` для частичных перемещений в виртуальном времени. Эти методы могут быть полезны в некоторых ситуациях, но вы должны стремиться к тому, чтобы каждый модульный тест проверял что-то одно. Вообще, для тестирования тайм-аута можно написать один модульный тест, который частично продвигает вперед `TestScheduler` и убеждается в том, что тайм-аут не произошел слишком рано, а потом выводит `TestScheduler` за время тайм-аута и убеждается в том, что на этот раз тайм-аут произошел. Тем не менее я предпочитаю

разделять модульные тесты настолько, насколько это возможно; например, один модульный тест проверяет, что тайм-аут не произошел слишком рано, а другой модульный тест — что он произошел позднее.

Дополнительная информация

В рецепте 7.5 рассматриваются основы модульного тестирования наблюдаемых последовательностей.

ГЛАВА 8

Взаимодействие

Асинхронное, параллельное, реактивное программирование — каждое из них на своем месте, но как они работают в сочетании друг с другом?

В этой главе рассматриваются различные сценарии взаимодействий, которые научат вас объединять разные подходы. Вы узнаете, как они взаимно дополняют друг друга, вместо того чтобы конкурировать; на границах соприкосновения двух подходов почти нет каких-либо неровностей.

8.1. Асинхронные обертки для «Async»-методов с «Completed»-событиями

Задача

Существует старый асинхронный паттерн, в котором используются методы с именами вида *Операция*`Async` и события с именами вида *Операция*`Completed`. Требуется выполнить операцию с использованием старого асинхронного паттерна и использовать `await` с результатом.

Паттерн *Операция*`Async`/*Операция*`Completed` называется асинхронным паттерном на основе событий (EAP, Event-based Asynchronous Pattern). Они будут упакованы в метод, возвращающий Task и реализующий асинхронный паттерн на основе Task (Task-based Asynchronous Pattern).

Решение

С помощью типа `TaskCompletionSource<TResult>` можно создавать обертки для асинхронных операций. Тип`TaskCompletionSource<TResult>` управляет `Task<TResult>` и позволяет завершить задачу в нужный момент.

В следующем примере определяется метод расширения для `WebClient`, который загружает строку. Тип `WebClient` определяет методы `DownloadStringAsync` и `DownloadStringCompleted`. С их помощью можно определить метод `DownloadStringTaskAsync`:

```
public static Task<string> DownloadStringTaskAsync(this WebClient client,
    Uri address)
{
  var tcs = new TaskCompletionSource<string>();

  // Обработка события завершит задачу и отменит свою регистрацию.
  DownloadStringCompletedEventHandler handler = null;
  handler = (_, e) =>
  {
    client.DownloadStringCompleted -= handler;
    if (e.Cancelled)
      tcs.TrySetCanceled();
    else if (e.Error != null)
      tcs.TrySetException(e.Error);
    else
      tcs.TrySetResult(e.Result);
  };

  // Зарегистрировать событие и *затем* начать операцию.
  client.DownloadStringCompleted += handler;
  client.DownloadStringAsync(address);
  return tcs.Task;
}
```

Пояснение

Этот конкретный пример не слишком полезен, потому что `WebClient` уже определяет `DownloadStringTaskAsync`, а вы можете использовать более удобную для `async` версию `HttpClient`. Тем не менее этот прием также мо-

жет использоваться для взаимодействия со старым асинхронным кодом, который еще не был обновлен для использования `Task`.

В новом коде всегда следует использовать `HttpClient`. Используйте `WebClient` только в том случае, если вы работаете с унаследованным кодом.

В обычной ситуации TAP-метод для загрузки строк будет называться *Операция*`Async` (например, `DownloadStringAsync`); тем не менее эта схема формирования имен в данном случае не работает, потому что EAP уже определяет метод с таким именем. В таком случае TAP-методу присваивается имя *Операция*`TaskAsync` (например, `DownloadStringTaskAsync`).

При создании оберток для EAP-методов существует вероятность того, что «стартовый» метод может выдать исключение; в предыдущем примере это может произойти в `DownloadStringAsync`. В этом случае необходимо решить, разрешить ли исключению распространяться или перехватить исключение и вызвать `TrySetException`. В большинстве случае исключения, выданные в этой точке, происходят от ошибок использования, поэтому неважно, какой из вариантов вы выберете. Если вы не уверены в том, являются ли исключения ошибками использования, рекомендую перехватить исключение и вызвать `TrySetException`.

Дополнительная информация

В рецепте 8.2 рассматриваются TAP-обертки для методов APM (`BeginOperation` и `EndOperation`).

В рецепте 8.3 рассматриваются TAP-обертки для любых типов уведомлений.

8.2. Асинхронные обертки для методов «Begin/End»

Задача

В старом асинхронном паттерне используются пары методов с именами `Begin`*Операция* и `End`*Операция*, а также объектом `IAsyncResult`, представляющим асинхронную операцию. Имеется операция, реализо-

ванная на базе старого асинхронного паттерна; требуется организовать ее потребление с ключевым словом `await`.

Паттерн «`BeginОперация/EndОперация`» называется асинхронной моделью программирования (APM, Asynchronous Programming Model). Он будет упакован в метод, возвращающий `Task` и реализующий асинхронный паттерн на основе `Task` (Task-based Asynchronous Pattern).

Решение

Лучший способ упаковки APM — использование одного из методов `FromAsync` с типом `TaskFactory`. `FromAsync` использует `TaskCompletionSource<TResult>` во внутренней реализации, но при создании обертки для APM `FromAsync` намного проще использовать.

Этот пример определяет метод расширения для `WebRequest`, который отправляет запрос HTTP и получает ответ. Тип `WebRequest` определяет `BeginGetResponse` и `EndGetResponse`; вы можете определить метод `GetResponseAsync` в следующем виде:

```
public static Task<WebResponse> GetResponseAsync(this WebRequest client)
{
  return Task<WebResponse>.Factory.FromAsync(client.BeginGetResponse,
      client.EndGetResponse, null);
}
```

Пояснение

У `FromAsync` есть множество перегруженных версий, от которых голова идет кругом!

Как правило, лучше всего вызывать `FromAsync` так, как это сделано в нашем примере. Сначала передайте метод `Begin`*Операция* (не вызывая его), затем передайте метод `End`*Операция* (тоже без вызова). Затем передайте все аргументы, которые получает `Begin`*Операция*, кроме последних аргументов `AsyncCallback` и `object`. Наконец, передайте `null`.

Не вызывайте метод `Begin`*Операция* перед вызовом `FromAsync`. Можно вызвать `FromAsync` с передачей объекта `IAsyncOperation`, полученного от

`Begin`*Операция*, но при таком вызове `FromAsync` придется использовать менее эффективную реализацию.

Возможно, вас интересует, почему в рекомендованном паттерне в конце всегда передается `null`. Метод `FromAsync` появился вместе с типом `Task` в .NET 4.0, перед появлением `async`. В то время в асинхронных обратных вызовах было принято использовать объекты `state`, а тип `Task` поддерживает эту возможность с помощью своего метода `AsyncState`. В новом паттерне `async` объекты `state` уже не нужны, поэтому в параметре `state` можно всегда передавать `null`. В те дни параметр `state` использовался только для предотвращения создания экземпляра замыкания (closure) при оптимизации использования памяти.

Дополнительная информация

В рецепте 8.3 рассматривается написание TAP-оберток для любых типов уведомлений.

8.3. Асинхронные обертки для чего угодно

Задача

Есть нетипичная или нестандартная асинхронная операция или событие. Требуется обеспечить их потребление с ключевым словом `await`.

Решение

Тип `TaskCompletionSource<T>` может использоваться для построения объектов `Task<T>` в любых сценариях. С помощью `TaskCompletionSource<T>` можно завершить задачу тремя разными способами: с успешным результатом, с отказом или с отменой.

До появления `async` компания Microsoft рекомендовала использовать два других асинхронных паттерна: APM (рецепт 8.2) и EAP (рецепт 8.1). Однако и APM, и EAP были достаточно неудобными, а в некоторых ситуациях их было трудно реализовать. По этой причине появилась неофициальная схема, основанная на обратных вызовах, с методами следующего вида:

```
public interface IMyAsyncHttpService
{
  void DownloadString(Uri address, Action<string, Exception> callback);
}
```

Такие методы следуют соглашению, согласно которому `DownloadString` запускает (асинхронную) загрузку, а при завершении `callback` активизируется либо с результатом, либо с исключением. Обычно `callback` активизируется в фоновом потоке.

Нестандартные асинхронные методы (вроде приведенного в предыдущем примере) могут быть упакованы в `TaskCompletionSource<T>`, чтобы они естественным образом работали с `await`, как в следующем примере:

```
public static Task<string> DownloadStringAsync(
    this IMyAsyncHttpService httpService, Uri address)
{
  var tcs = new TaskCompletionSource<string>();
  httpService.DownloadString(address, (result, exception) =>
  {
    if (exception != null)
      tcs.TrySetException(exception);
    else
      tcs.TrySetResult(result);
  });
  return tcs.Task;
}
```

Пояснение

Тот же паттерн `TaskCompletionSource<T>` может использоваться для упаковки *любых* асинхронных методов, какими бы нестандартными они ни были. Сначала создайте экземпляр `TaskCompletionSource<T>`. Затем организуйте обратный вызов, чтобы `TaskCompletionSource<T>` завершал свою задачу соответствующим образом. Запустите асинхронную операцию и наконец верните объект `Task<T>`, связанный с этим `TaskCompletionSource<T>`.

В этом паттерне важно быть *уверенным* в том, чтобы объект `TaskCompletionSource<T>` всегда завершался. Тщательно продумайте обработку ошибок и убедитесь в том, чтобы `TaskCompletionSource<T>` завершался соответствующим образом. В последнем примере исключения явно передаются

обратному вызову, так что блок `catch` не понадобится; но некоторые нестандартные паттерны могут потребовать перехвата исключений в обратных вызовах и включения их в `TaskCompletionSource<T>`.

Дополнительная информация

В рецепте 8.1 рассматриваются TAP-обертки для методов EAP (*Операция*`Async`, *Операция*`Completed`).

В рецепте 8.2 рассматриваются TAP-обертки для методов APM (`Begin`*Операция*, `End`*Операция*).

8.4. Асинхронные обертки для параллельного кода

Задача

Существуют параллельные вычисления (создающие нагрузку на процессор), которые вы хотите потреблять с помощью `await`. Обычно бывает нужно, чтобы UI-поток не блокировался в ожидании завершения параллельных вычислений.

Решение

Тип `Parallel` и Parallel LINQ используют пул потоков для выполнения параллельной обработки. При этом вызывающий поток также включается в число потоков параллельной обработки, так что при вызове параллельного метода из UI-потока пользовательский интерфейс перестанет реагировать на действия пользователя до завершения обработки.

Чтобы пользовательский интерфейс не блокировался, упакуйте параллельную обработку в `Task.Run` и примените `await` к результату:

```
await Task.Run(() => Parallel.ForEach(...));
```

Ключевой аспект этого рецепта заключается в том, что параллельный код *включает вызывающий поток* в свой пул потоков, используемых для параллельной обработки. Это относится как к Parallel LINQ, так и к классу `Parallel`.

Пояснение

Это простой рецепт, но его часто упускают из виду. Используя `Task.Run`, вы перемещаете всю параллельную обработку в пул потоков. `Task.Run` возвращает объект `Task`, представляющий параллельную работу, а UI-поток может (асинхронно) ожидать его завершения.

Этот рецепт относится только к UI-коду. На стороне сервера (например, ASP.NET) параллельная обработка выполняется редко, потому что параллелизм уже обеспечивается серверным хостом. По этой причине код на стороне сервера не должен ни выполнять параллельную обработку, ни передавать ее пулу потоков.

Дополнительная информация

В главе 4 рассматриваются основные принципы параллельного кода.

В главе 2 рассматриваются основные принципы асинхронного кода.

8.5. Асинхронные обертки для наблюдаемых объектов System.Reactive

Задача

Имеется наблюдаемый поток, который требуется потреблять с использованием `await`.

Решение

Сначала необходимо решить, *какие* события наблюдаемого объекта в потоке событий вас интересуют. Самые распространенные ситуации:

- Последнее событие перед завершением потока.
- Следующее событие.
- Все события.

Чтобы получить *последнее* событие в потоке, можно применить `await` либо к результату `LastAsync`, либо к наблюдаемому объекту напрямую:

```
IObservable<int> observable = ...;
int lastElement = await observable.LastAsync();
// или:  int lastElement = await observable;
```

Когда вы используете `await` с наблюдаемым объектом или `LastAsync`, код (асинхронно) ожидает завершения потока, после чего возвращает последний элемент. Во внутренней реализации `await` подписывается на поток.

Чтобы получить *следующее* событие в потоке, используйте `FirstAsync`. В следующем коде `await` подписывается на поток, после чего завершается (и отменяет подписку) сразу же с поступлением первого события:

```
IObservable<int> observable = ...;
int nextElement = await observable.FirstAsync();
```

Для отслеживания всех событий в потоке можно воспользоваться `ToList`:

```
IObservable<int> observable = ...;
IList<int> allElements = await observable.ToList();
```

Пояснение

Библиотека System.Reactive предоставляет все инструменты, необходимые для потребления потоков с использованием `await`. Единственный нюанс заключается в том, что придется думать о том, будет ли объект, допускающий ожидание, ожидать до завершения потока. Из примеров данного рецепта `LastAsync`, `ToList` и собственно `await` будут ожидать завершения потока; `FirstAsync` будет ожидать только следующего события.

Если этих примеров вам недостаточно, вспомните, что в вашем распоряжении вся мощь LINQ, а также манипуляторы System.Reactive. Такие операции, как `Take` и `Buffer`, помогут в асинхронном ожидании необходимых элементов без ожидания завершения всего потока.

Некоторые операторы для использования с `await` — такие, как `FirstAsync` и `LastAsync`, — не возвращают `Task<T>`. Если вы планируете использовать `Task.WhenAll` или `Task.WhenAny`, то вам понадобится объект `Task<T>`, который можно получить вызовом `ToTask` для любого наблюдаемого объекта. `ToTask` вернет объект `Task<T>`, который завершается с последним значением в потоке.

Дополнительная информация

В разделе 8.6 рассматривается использование асинхронного кода с наблюдаемым потоком.

В разделе 8.8 рассматривается использование наблюдаемых потоков в качестве входных данных для блока потока данных (который может выполнять асинхронную работу).

В разделе 6.3 рассматривается использование окон и буферизации для наблюдаемых потоков.

8.6. Наблюдаемые обертки для асинхронного кода в System.Reactive

Задача

Имеется асинхронная операция, которую требуется объединить с функциональностью наблюдаемых объектов.

Решение

Любая асинхронная операция может интерпретироваться как наблюдаемый поток, который делает одно из двух:

- производит один элемент, после чего завершается;
- выдает отказ без генерирования каких-либо элементов.

Для реализации этой трансформации в библиотеке System.Reactive существует простое преобразование `Task<T>` в `IObservable<T>`. Следующий код запускает асинхронную загрузку веб-страниц, интерпретируя ее как наблюдаемую последовательность:

```
IObservable<HttpResponseMessage> GetPage(HttpClient client)
{
  Task<HttpResponseMessage> task =
      client.GetAsync("http://www.example.com/");
  return task.ToObservable();
}
```

Решение с `ToObservable` предполагает, что вы уже вызвали `async`-метод и у вас имеется объект `Task` для преобразования.

Другой подход основан на вызове `StartAsync`. `StartAsync` также вызывает `async`-метод немедленно, но с поддержкой отмены: если подписка будет отменена, то `async`-метод отменяется:

```
IObservable<HttpResponseMessage> GetPage(HttpClient client)
{
  return Observable.StartAsync(
      token => client.GetAsync("http://www.example.com/", token));
}
```

Как `ToObservable`, так и `StartAsync` немедленно запускают асинхронную операцию без ожидания подписки; наблюдаемый объект является «горячим». Чтобы создать «холодный» наблюдаемый объект, который запускает операцию только при подписке, используйте метод `FromAsync` (который так же поддерживает отмену, как и `StartAsync`):

```
IObservable<HttpResponseMessage> GetPage(HttpClient client)
{
  return Observable.FromAsync(
      token => client.GetAsync("http://www.example.com/", token));
}
```

`FromAsync` существенно отличается от `ToObservable` и `StartAsync`, которые возвращают наблюдаемый объект для уже запущенной `async`-операции. `FromAsync` запускает новую независимую `async`-операцию каждый раз, когда создается подписка.

Наконец, можно использовать специальные перегруженные версии `SelectMany` для запуска асинхронных операций для каждого события в исходном потоке по мере их поступления. `SelectMany` также поддерживает отмену.

Следующий пример получает существующий поток событий с URL, после чего инициирует запрос при получении каждого URL:

```
IObservable<HttpResponseMessage> GetPages(
    IObservable<string> urls, HttpClient client)
{
  return urls.SelectMany(
      (url, token) => client.GetAsync(url, token));
}
```

Пояснение

Библиотека System.Reactive существовала до появления `async`, но эти (и другие) операторы были добавлены для нормального взаимодействия с кодом `async`. Рекомендую использовать описанные в этом рецепте операторы даже в том случае, если ту же функциональность можно построить с использованием других операторов System.Reactive.

Дополнительная информация

В рецепте 8.5 рассматривается потребление наблюдаемых потоков в асинхронном коде.

В рецепте 8.8 рассматривается использование блоков потоков данных (которые могут содержать асинхронный код) как источников наблюдаемых потоков.

8.7. Асинхронные потоки и сети потоков данных

Задача

В одной части вашего решения используются асинхронные потоки, а в другой — сети потоков данных. Требуется организовать передачу данных между ними.

Решение

Потребление `Channel` как асинхронного потока встроено непосредственно в тип `Channel`; подробности см. в рецепте 9.8. Потребление блока TPL Dataflow как асинхронного потока происходит немного сложнее, но безусловно реально. На мой взгляд, проще сначала определить метод расширения для блоков потоков данных, которые делают их API более сходным с `Channel`, а затем использовать их как асинхронный поток с помощью этого метода расширения:

```
public static class DataflowExtensions
{
```

```
  public static bool TryReceiveItem<T>(this ISourceBlock<T> block,
     out T value)
  {
    if (block is IReceivableSourceBlock<T> receivableSourceBlock)
      return receivableSourceBlock.TryReceive(out value);

    try
    {
      value = block.Receive(TimeSpan.Zero);
      return true;
    }
    catch (TimeoutException)
    {
      // На данный момент доступного элемента нет.
      value = default;
      return false;
    }
    catch (InvalidOperationException)
    {
      // Блок завершен, элементов больше нет.
      value = default;
      return false;
    }
  }

  public static async IAsyncEnumerable<T> ReceiveAllAsync<T>(
      this ISourceBlock<T> block,
      [EnumeratorCancellation] CancellationToken cancellationToken =
         default)
  {
    while (await block
        .OutputAvailableAsync(cancellationToken).ConfigureAwait(false))
    {
      while (block.TryReceiveItem(out var value))
      {
        yield return value;
      }
    }
  }
}
```

Подробную информацию об атрибуте `EnumeratorCancellation` см. в рецепте 3.4.

С методом расширения из приведенного примера любой выходной блок потока данных может использоваться как асинхронный поток:

```
var multiplyBlock = new TransformBlock<int, int>(value => value * 2);

multiplyBlock.Post(5);
multiplyBlock.Post(2);
multiplyBlock.Complete();

await foreach (int item in multiplyBlock.ReceiveAllAsync())
{
  Console.WriteLine(item);
}
```

Также возможно использовать асинхронный поток в качестве источника элементов для блока потока данных. Все, что для этого понадобится, — это цикл, который извлекает элементы и помещает их в блок. В приведенном ниже коде действует пара допущений, которые могут быть неприемлемы в некоторых сценариях. Во-первых, предполагается, что блок должен завершаться при завершении потока. Во-вторых, он начинает выполняться в своем вызывающем потоке; в некоторых сценариях бывает нужно, чтобы весь цикл выполнялся в потоке из пула потоков:

```
public static async Task WriteToBlockAsync<T>(
    this IAsyncEnumerable<T> enumerable,
    ITargetBlock<T> block, CancellationToken token = default)
{
  try
  {
    await foreach (var item in enumerable
        .WithCancellation(token).ConfigureAwait(false))
    {
      await block.SendAsync(item, token).ConfigureAwait(false);
    }

    block.Complete();
  }
  catch (Exception ex)
  {
    block.Fault(ex);
  }
}
```

Пояснение

Предполагается, что методы расширения в этом рецепте станут отправной точкой для дальнейшей работы. В частности, метод расширения `WriteToBlockAsync` основан на некоторых допущениях; прежде чем пользоваться этими методами, обязательно продумайте их поведение и убедитесь в том, что оно соответствует вашей ситуации.

Дополнительная информация

В рецепте 9.8 рассматривается потребление `Channel` как асинхронного потока.

В рецепте 3.4 рассматривается отмена асинхронных потоков.

В главе 5 представлены рецепты для TPL Dataflow.

В главе 3 представлены рецепты для асинхронных потоков.

8.8. Наблюдаемые объекты System.Reactive Observables и сети потока данных

Задача

В одной части решения используются наблюдаемые объекты System.Reactive, в другой — сети потоков данных. Требуется организовать их взаимодействие.

И у наблюдаемых объектов System.Reactive, и у сетей потоков данных существуют свои области применения, которые перекрываются на концептуальном уровне; этот рецепт показывает, как легко организовать их совместную работу, чтобы вы могли выбрать наилучший инструмент для каждой части работы.

Решение

Для начала рассмотрим использование блока потока данных как входа для наблюдаемого потока. Следующий пример создает буферный блок

(который не выполняет никакой обработки), а также наблюдаемый интерфейс на базе этого блока вызовом `AsObservable`:

```
var buffer = new BufferBlock<int>();
IObservable<int> integers = buffer.AsObservable();
integers.Subscribe(data => Trace.WriteLine(data),
    ex => Trace.WriteLine(ex),
    () => Trace.WriteLine("Done"));
buffer.Post(13);
```

Как буферные блоки, так и наблюдаемые потоки могут завершаться нормально или с ошибкой, и метод `AsObservable` преобразует завершение блока (или отказ) в завершение наблюдаемого потока. Однако если в блоке происходит отказ с выдачей исключения, при передаче наблюдаемому потоку это исключение будет упаковано в объект `AggregateException`. Происходящее напоминает механизм распространения отказов связанными блоками.

Ненамного сложнее взять сеть и работать с ней как с приемником для наблюдаемого потока. Следующий код вызывает `AsObserver`, чтобы блок мог подписаться на наблюдаемый поток:

```
IObservable<DateTimeOffset> ticks =
    Observable.Interval(TimeSpan.FromSeconds(1))
        .Timestamp()
        .Select(x => x.Timestamp)
        .Take(5);
var display = new ActionBlock<DateTimeOffset>(x => Trace.WriteLine(x));
ticks.Subscribe(display.AsObserver());
try
{
  display.Completion.Wait();
  Trace.WriteLine("Done.");
}
catch (Exception ex)
{
  Trace.WriteLine(ex);
}
```

Как и прежде, завершение наблюдаемого потока преобразуется в завершение блока, а любые ошибки из наблюдаемого потока преобразуются в отказ блока.

Пояснение

У блоков потоков данных и наблюдаемых потоков на концептуальном уровне есть много общего: через них проходят данные, в обоих случаях поддерживаются завершение и отказы. Они проектировались для разных сценариев; библиотека TPL Dataflow предназначалась для комбинаций асинхронного и параллельного программирования, а System.Reactive — для реактивного программирования. Впрочем, это концептуальное перекрытие совместимо в достаточной мере, чтобы они очень хорошо и естественно работали друг с другом.

Дополнительная информация

В рецепте 8.5 рассматривается потребление наблюдаемых потоков в асинхронном коде.

В рецепте 8.6 рассматривается использование асинхронного кода в наблюдаемых потоках.

8.9. Преобразование наблюдаемых объектов System.Reactive в асинхронные потоки

Задача

В отдельной части вашего решения используются наблюдаемые объекты System.Reactive. Требуется потреблять их как асинхронные потоки.

Решение

Наблюдаемые объекты System.Reactive работают по принципу проталкивания, а асинхронные потоки — по принципу вытягивания. А значит, необходимо прежде всего понять, что существует концептуальное несоответствие. Требуется обеспечить отклик для наблюдаемого потока и сохранить его уведомления до того, как они будут запрошены потребляющим кодом.

Самое прямолинейное решение уже включено в библиотеку `System.Linq.Async`:

```
IObservable<long> observable =
    Observable.Interval(TimeSpan.FromSeconds(1));

// ПРЕДУПРЕЖДЕНИЕ: может потреблять неограниченную память; см. обсуждение!
IAsyncEnumerable<long> enumerable =
    observable.ToAsyncEnumerable();
```

Метод расширения `ToAsyncEnumerable` находится в NuGet-пакете `System.Linq.Async`.

Важно понять, что этот простой метод расширения `ToAsyncEnumerable` во внутренней реализации использует неограниченную очередь «производитель/потребитель». По сути он не отличается от метода расширения, который вы можете написать самостоятельно с использованием `Channel` как неограниченной очереди «производитель/потребитель»:

```
// ПРЕДУПРЕЖДЕНИЕ: может потреблять неограниченную память; см. обсуждение!
public static async IAsyncEnumerable<T> ToAsyncEnumerable<T>(
    this IObservable<T> observable)
{
  Channel<T> buffer = Channel.CreateUnbounded<T>();
  using (observable.Subscribe(
      value => buffer.Writer.TryWrite(value),
      error => buffer.Writer.Complete(error),
      () => buffer.Writer.Complete()))
  {
    await foreach (T item in buffer.Reader.ReadAllAsync())
      yield return item;
  }
}
```

Эти решения просты, но используют неограниченные очереди, поэтому пользоваться ими следует только в том случае, если вы уверены, что потребитель сможет (со временем) не отстать от наблюдаемых событий.

Если потребитель в течение какого-то времени отстает от производителя, ничего страшного; в это время наблюдаемые события сохраняются в буфере. Если потребитель со временем догонит его, приведенные решения будут работать нормально. Но если производитель всегда работает быстрее потребителя, наблюдаемые события продолжат поступать, буфер будет расширяться и со временем займет всю память процесса.

Чтобы избежать проблем с памятью, можно воспользоваться ограниченной очередью. Правда, вам придется решить, что делать с лишними элементами, если наблюдаемые события переполнят очередь. Один из возможных вариантов — отбросить лишние элементы; в следующем примере ограниченный канал используется для уничтожения самого старого наблюдаемого уведомления при переполнении буфера:

```
// ПРЕДУПРЕЖДЕНИЕ: возможна потеря элементов; см. обсуждение!
public static async IAsyncEnumerable<T> ToAsyncEnumerable<T>(
    this IObservable<T> observable, int bufferSize)
{
  var bufferOptions = new BoundedChannelOptions(bufferSize)
  {
    FullMode = BoundedChannelFullMode.DropOldest,
  };
  Channel<T> buffer = Channel.CreateBounded<T>(bufferOptions);
  using (observable.Subscribe(
      value => buffer.Writer.TryWrite(value),
      error => buffer.Writer.Complete(error),
      () => buffer.Writer.Complete()))
  {
    await foreach (T item in buffer.Reader.ReadAllAsync())
      yield return item;
  }
}
```

Пояснение

Если ваш производитель работает быстрее, чем потребитель, придется либо буферизовать элементы производителя (в предположении, что потребитель со временем его догонит), либо ограничить их количество. Второе решение в этом рецепте ограничивает элементы производителя за счет потери элементов, не помещающихся в буфере. Также можно ограничить элементы производителя с помощью операторов наблюдаемых объектов,

предназначенных для этой цели, — таких, как `Throttle` или `Sample`; за подробностями обращайтесь к рецепту 6.4. В зависимости от ваших потребностей может быть лучше применить `Throttle` или `Sample` к входному наблюдаемому объекту перед его преобразованием в `IAsyncEnumerable<T>` одним из способов, описанных в этом рецепте.

Кроме ограниченных и неограниченных очередей существует третий вариант, который здесь не описан: использование обратного давления (backpressure) для уведомления наблюдаемого потока о том, что он должен перестать производить уведомления, пока буфер не будет готов принять их. К сожалению, в System.Reactive паттерн обратного давления еще не стандартизирован, поэтому на момент написания книги этот вариант еще не был работоспособным. Обратное давление — сложный и неоднозначный паттерн, а в реактивных библиотеках для других языков были реализованы разные паттерны для обратного давления. Пока неясно, примет ли System.Reactive один из этих паттернов, изобретет ли собственный паттерн обратного давления или просто оставит проблему обратного давления неразрешенной.

Дополнительная информация

В рецепте 6.4 рассматриваются операторы System.Reactive, предназначенные для регулировки ввода.

В рецепте 9.8 рассматривается использование `Channel` как неограниченной очереди «производитель/потребитель».

В рецепте 9.10 рассматривается использование `Channel` как очереди выборки с потерей элементов при переполнении.

ГЛАВА 9

Коллекции

Правильный выбор коллекций чрезвычайно важен для конкурентных приложений. Речь не о стандартных коллекциях вроде `List<T>`; полагаю, вы о них уже знаете. В этой главе я хочу представить новые коллекции, предназначенные специально для конкурентного или асинхронного использования.

Неизменяемые коллекции представляют собой экземпляры коллекций, которые не могут изменяться ни при каких условиях. На первый взгляд может показаться, что они абсолютно бесполезны; но неизменяемые коллекции чрезвычайно полезны даже в однопоточных, неконкурентных приложениях. Операции, доступные только для чтения (например, перечисление) выполняются с неизменяемым экземпляром напрямую. Операции записи (например, добавление элемента) возвращают новый неизменяемый экземпляр вместо изменения существующего экземпляра. Это не настолько неэффективно, как может показаться, потому что в большинстве случаев неизменяемые коллекции совместно используют бóльшую часть своей памяти. Кроме того, неизменяемые коллекции обладают таким преимуществом, как неявная безопасность обращения из разных потоков; если коллекции не могут изменяться, то они являются потокобезопасными.

Неизменяемые коллекции находятся в NuGet-пакете `System.Collections.Immutable`.

Неизменяемые коллекции — относительно новое явление, и стоит применять их в новых разработках, только если вам *не нужен* изменяемый экземпляр. Если вы не знакомы с неизменяемыми коллекциями, рекомендую начать с рецепта 9.1; даже если вам не нужен именно стек или

очередь — в этом рецепте рассматриваются некоторые общие паттерны, задействованные для всех неизменяемых коллекций.

Есть специальные способы более эффективного конструирования неизменяемых коллекций с большим количеством существующих элементов; в примерах из этих рецептов элементы добавляются по одному. В документации MSDN содержится подробная информация об эффективном создании неизменяемых коллекций, если вам понадобится ускорить процесс инициализации.

- *Потокобезопасные коллекции*. Эти изменяемые экземпляры коллекций могут изменяться несколькими потоками одновременно. Потокобезопасные коллекции используют сочетание детализированных блокировок и приемов, не использующих блокировки, которое гарантирует, что потоки будут блокироваться на минимальное время (а обычно не блокируются вовсе). В случае потокобезопасных коллекций при перечислении коллекции создается ее снимок, после чего перечисление выполняется с этим снимком. Ключевое преимущество потокобезопасных коллекций — возможность безопасного обращения к ним из нескольких потоков при том, что операции будут блокировать ваш код на минимальное время (или не блокировать вообще).
- *Коллекции «производитель/потребитель»*. Эти экземпляры изменяемых коллекций проектировались с конкретной целью: разрешить (возможно, нескольким) производителям заносить элементы в коллекцию, в то время как потребители (которых тоже может быть несколько) извлекают элементы из коллекции. Таким образом, коллекции играют роль моста между кодом производителя и кодом потребителя с дополнительной возможностью ограничить количество элементов в коллекции. Коллекции «производитель/потребитель» также могут иметь блокирующий или асинхронный API. Например, если коллекция пуста, блокирующая коллекция «производитель/потребитель» блокирует вызывающий поток-потребитель до того момента, когда будет добавлен новый элемент; с другой стороны, асинхронная коллекция «производитель/потребитель» позволяет вызывающему потоку асинхронно ожидать добавления нового элемента.

В рецептах этой главы используются разные коллекции «производитель/потребитель», обладающие разными преимуществами. Таблица 9.1 поможет определиться, какую коллекцию стоит использовать в конкретном случае.

Таблица 9.1. Коллекции «производитель/потребитель»

Признак	Channels	BlockingCollection<T>	BufferBlock<T>	AsyncProducerConsumerQueue<T>	AsyncCollection<T>
Семантика очереди	✓	✓	✓	✓	✓
Семантика стека/мультимножества	×	✓	×	×	✓
Синхронный API	✓	✓	✓	✓	✓
Асинхронный API	✓	×	✓	✓	✓
Потеря элементов при переполнении	✓	×	×	×	×
Протестирован Microsoft	✓	✓	✓	×	×

Библиотека Channels находится в пакете `System.Threading.Channels`, `BufferBlock<T>` — в пакете `System.Threading.Tasks.Dataflow`, а `AsyncProducerConsumerQueue<T>` и `AsyncCollection<T>` — в пакете `Nito.AsyncEx`.

9.1. Неизменяемые стеки и очереди

Задача

Вам нужна коллекция — стек или очередь, которая изменяется не очень часто и к которой можно безопасно обращаться из нескольких потоков.

Например, очередь может использоваться для представления последовательности выполняемых операций, а стек — для представления последовательности операций отмены.

Решение

Простейшие неизменяемые коллекции — неизменяемые стеки и очереди. По своему поведению они очень близки к стандартным коллекциям `Stack<T>` и `Queue<T>`. В отношении быстродействия неизменяемые стеки и очереди обладают практически такой же временной сложностью, что

и стандартные стеки и очереди; впрочем, в простых сценариях с частым обновлением коллекций стандартные стеки и очереди работают быстрее.

Стеки относятся к категории LIFO («Last In, First Out», т. е. «последним зашел, первым вышел»). Следующий пример создает пустой неизменяемый стек, заносит в него два элемента, перебирает элементы, после чего извлекает элемент из стека:

```
ImmutableStack<int> stack = ImmutableStack<int>.Empty;
stack = stack.Push(13);
stack = stack.Push(7);

// Выводит "7", затем "13".
foreach (int item in stack)
  Trace.WriteLine(item);

int lastItem;
stack = stack.Pop(out lastItem);
// lastItem == 7
```

Обратите внимание: в этом примере многократно перезаписывается локальная переменная `stack`. Неизменяемые коллекции строятся на основе паттерна, в соответствии с которым они возвращают обновленную коллекцию; ссылка на исходную коллекцию остается без изменений. Это означает, что если имеется ссылка на конкретный экземпляр неизменяемой коллекции, она никогда не изменится. Рассмотрим следующий пример:

```
ImmutableStack<int> stack = ImmutableStack<int>.Empty;
stack = stack.Push(13);
ImmutableStack<int> biggerStack = stack.Push(7);

// Выводит "7", затем "13".
foreach (int item in biggerStack)
  Trace.WriteLine(item);

// Выводит только "13".
foreach (int item in stack)
  Trace.WriteLine(item);
```

Во внутренней реализации два стека совместно используют память, выделенную для хранения элемента 13. Такая реализация весьма эффективна, к тому же она позволяет легко создавать снимки текущего состояния. Каждый экземпляр неизменяемой коллекции потокобезопасен по сво-

ей природе, но неизменяемые коллекции также могут использоваться в однопоточных приложениях. По моему опыту, неизменяемые коллекции особенно удобны при использовании функционального кода, а также при необходимости хранить большое количество снимков коллекции, которые должны по возможности совместно использовать одну память.

Очереди похожи на стеки, но они относятся к категории структур FIFO («First In, First Out», т. е. «первым зашел, первым вышел»). Следующий пример создает пустую неизменяемую очередь, помещает в очередь два элемента, перебирает элементы, а затем извлекает элемент из очереди:

```
ImmutableQueue<int> queue = ImmutableQueue<int>.Empty;
queue = queue.Enqueue(13);
queue = queue.Enqueue(7);

// Выводит "13", затем "7".
foreach (int item in queue)
  Trace.WriteLine(item);
int nextItem;
queue = queue.Dequeue(out nextItem);

// Выводит "13".
Trace.WriteLine(nextItem);
```

Пояснение

В этом рецепте представлены две простейшие неизменяемые коллекции — стек и очередь. В нем также изложены некоторые важные принципы проектирования, справедливые для всех неизменяемых коллекций:

- Экземпляр неизменяемой коллекции никогда не изменяется.
- Так как экземпляр никогда не изменяется, он потокобезопасен по своей природе.
- При вызове изменяющего метода для неизменяемой коллекции возвращается новая измененная коллекция.

Неизменяемые коллекции являются потокобезопасными, но ссылки на них потокобезопасными не являются. Переменная, ссылающаяся на неизменяемую коллекцию, нуждается в такой же синхронизационной защите, как и любая другая переменная (см. главу 12).

Неизменяемые коллекции идеально подходят для хранения общего состояния. С другой стороны, в качестве коммуникационного канала они работают не так хорошо. В частности, неизменяемые очереди не следует использовать для передачи данных между потоками; очереди «производитель/потребитель» подходят для этой цели намного лучше.

`ImmutableStack<T>` и `ImmutableQueue<T>` находятся в пакете `System.Collections.Immutable`.

Дополнительная информация

В рецепте 9.6 рассматриваются потокобезопасные (блокирующие) изменяемые очереди.

В рецепте 9.7 рассматриваются потокобезопасные (блокирующие) изменяемые стеки.

В рецепте 9.8 рассматриваются `async`-совместимые изменяемые очереди.

В рецепте 9.11 рассматриваются `async`-совместимые изменяемые стеки.

В рецепте 9.12 рассматриваются блокирующие/асинхронные изменяемые очереди.

9.2. Неизменяемые списки

Задача

Нужна структура данных с возможностью индексирования, которая изменяется не слишком часто и допускает безопасные обращения из нескольких потоков.

Решение

Список — структура данных общего назначения, которая может использоваться для хранения разнообразных данных состояния приложения. Неизменяемые списки поддерживают индексирование, однако вы должны

учитывать их характеристики быстродействия. Они не должны рассматриваться как тривиальная замена для `List<T>`.

`ImmutableList<T>` поддерживает примерно те же методы, что и `List<T>`, как показывают следующие примеры:

```
ImmutableList<int> list = ImmutableList<int>.Empty;
list = list.Insert(0, 13);
list = list.Insert(0, 7);

// Выводит "7", затем "13".
foreach (int item in list)
  Trace.WriteLine(item);

list = list.RemoveAt(1);
```

Во внутренней реализации неизменяемого списка используется двоичное дерево, чтобы экземпляры неизменяемого списка могли максимизировать объем памяти, используемый совместно с другими экземплярами. В результате для некоторых распространенных операций существуют различия в быстродействии между `ImmutableList<T>` и `List<T>` (табл. 9.2).

Таблица 9.2. Различия в быстродействии для неизменяемых списков

Операция	List<T>	ImmutableList<T>
Add	Амортизированная O(1)	O(log N)
Insert	O(N)	O(log N)
RemoveAt	O(N)	O(log N)
Item[индекс]	O(1)	O(log N)

Стоит отметить, что операция индексирования для `ImmutableList<T>` обладает сложностью O(log N), а не O(1), как можно было бы ожидать. Если вы заменяете `List<T>` на `ImmutableList<T>` в существующем коде, следует учесть, как ваши алгоритмы обращаются к элементам коллекции.

Это означает, что следует использовать `foreach` вместо `for` там, где это возможно. Цикл `foreach` по `ImmutableList<T>` выполняется за время O(N), тогда как цикл `for` по той же коллекции выполняется за время O(N * log N):

```
// Лучший способ перебора ImmutableList<T>.
foreach (var item in list)
```

```
  Trace.WriteLine(item);
// Тоже будет работать, но намного медленнее.

for (int i = 0; i != list.Count; ++i)
  Trace.WriteLine(list[i]);
```

Пояснение

`ImmutableList<T>` — хорошая структура данных общего назначения, но из-за различий в быстродействии вы не сможете бездумно заменить ей все `List<T>`. `List<T>` часто используется по умолчанию — именно эту структуру данных следует использовать, если только у вас нет веских причин для выбора другой коллекции. Коллекция `ImmutableList<T>` не настолько распространена; следует тщательно проанализировать другие неизменяемые коллекции и выбрать ту, которая лучше всего подходит для вашей ситуации.

`ImmutableList<T>` находится в пакете `System.Collections.Immutable`.

Дополнительная информация

В рецепте 9.1 рассматриваются неизменяемые стеки и очереди — структуры данных, сходные со списками, но ограничивающие доступ некоторым элементам.

В документации `ImmutableList<T>.Builder` в MSDN (https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablelist-1.builder?redirectedfrom=MSDN&view=netcore-3.0) рассматривается эффективный способ заполнения неизменяемых списков.

9.3. Неизменяемые множества

Задача

Нужна структура данных, не рассчитанная на хранение дубликатов, которая не слишком часто изменяется и допускает безопасные обращения из нескольких потоков.

Например, индекс слов из файла может быть хорошим кандидатом для применения множества.

Решение

Существует два типа неизменяемых множеств: `ImmutableHashSet<T>` — коллекция уникальных элементов и `ImmutableSortedSet<T>` — отсортированная коллекция уникальных элементов. Типы обладают похожим интерфейсом:

```
ImmutableHashSet<int> hashSet = ImmutableHashSet<int>.Empty;
hashSet = hashSet.Add(13);
hashSet = hashSet.Add(7);

// Выводит "7" и "13" в непредсказуемом порядке.
foreach (int item in hashSet)
  Trace.WriteLine(item);

hashSet = hashSet.Remove(7);
```

Только отсортированное множество допускает индексирование по аналогии со списком:

```
ImmutableSortedSet<int> sortedSet = ImmutableSortedSet<int>.Empty;
sortedSet = sortedSet.Add(13);
sortedSet = sortedSet.Add(7);

// Выводит "7", затем "13".
foreach (int item in sortedSet)
  Trace.WriteLine(item);
int smallestItem = sortedSet[0];
// smallestItem == 7
sortedSet = sortedSet.Remove(7);
```

Несортированные и отсортированные множества обладают сходным быстродействием (табл. 9.3).

Рекомендую использовать несортированное множество, если только вы не уверены в том, что оно должно быть отсортированным. Многие типы поддерживают только базовое равенство, но не полное сравнение, так что несортированное множество может использоваться для большего количества типов, чем отсортированное множество.

Таблица 9.3. Быстродействие неизменяемых множеств

Операция	ImmutableHashSet<T>	ImmutableSortedSet<T>
Add	O(log N)	O(log N)
Remove	O(log N)	O(log N)
Item[индекс]	–	O(log N)

Одно важное примечание по поводу отсортированных множеств: индексирование для них выполняется за время O(log N), а не O(1), как у `ImmutableList<T>` (см. рецепт 9.2). Это означает, что в данной ситуации действует та же рекомендация: используйте `foreach` вместо `for` там, где это возможно, с `ImmutableSortedSet<T>`.

Пояснение

Неизменяемые множества полезны, но заполнение большого неизменяемого множества может быть медленной операцией. У многих неизменяемых коллекций имеются специальные построители, которые могут использоваться для быстрого их построения в изменяемом виде с последующим преобразованием в неизменяемую коллекцию. Это относится ко многим неизменяемым коллекциям, но, на мой взгляд, они особенно полезны для неизменяемых множеств.

`ImmutableHashSet<T>` и `ImmutableSortedSet<T>` находятся в пакете `System.Collections.Immutable`.

Дополнительная информация

В рецепте 9.7 рассматриваются потокобезопасные изменяемые мультимножества, сходные с множествами.

В рецепте 9.11 рассматриваются `async`-совместимые изменяемые мультимножества.

В документации `ImmutableHashSet<T>.Builder` в MSDN (https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablehashset-1.builder?redirectedfrom=MSDN&view=netcore-3.0) рассматривается эффективный способ заполнения неизменяемых хешированных множеств.

В документации `ImmutableSortedSet<T>.Builder` в MSDN (https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablesortedset-1.builder?redirectedfrom=MSDN&view=netcore-3.0) рассматривается эффективный способ заполнения неизменяемых отсортированных множеств.

9.4. Неизменяемые словари

Задача

Нужна коллекция «ключ/значение», которая не слишком часто изменяется и допускает безопасные обращения из нескольких потоков. Например, в этой коллекции могут храниться данные ссылок в подстановочной таблице; данные ссылок редко изменяются, но они должны быть доступны для разных потоков.

Решение

Есть два типа неизменяемых множеств: `ImmutableDictionary<TKey, TValue>` и `ImmutableSortedDictionary<TKey, TValue>`. Как нетрудно догадаться по именам, если элементы `ImmutableDictionary` следуют в непредсказуемом порядке, `ImmutableSortedDictionary` гарантирует, что его элементы следуют в порядке сортировки.

Эти типы коллекций имеют очень похожие составляющие:

```
ImmutableDictionary<int, string> dictionary =
    ImmutableDictionary<int, string>.Empty;
dictionary = dictionary.Add(10, "Ten");
dictionary = dictionary.Add(21, "Twenty-One");
dictionary = dictionary.SetItem(10, "Diez");

// Выводит "10Diez" и "21Twenty-One" в непредсказуемом порядке.
foreach (KeyValuePair<int, string> item in dictionary)
  Trace.WriteLine(item.Key + item.Value);

string ten = dictionary[10];
// ten == "Diez"

dictionary = dictionary.Remove(21);
```

Обратите внимание на использование `SetItem`. В изменяемом словаре можно было бы попытаться использовать конструкцию вида *словарь*`[`*ключ*`] = = `*элемент*, но неизменяемые словари должны возвращать обновленный неизменяемый словарь, поэтому вместо этого они должны использовать метод `SetItem`:

```
ImmutableSortedDictionary<int, string> sortedDictionary =
    ImmutableSortedDictionary<int, string>.Empty;
sortedDictionary = sortedDictionary.Add(10, "Ten");
sortedDictionary = sortedDictionary.Add(21, "Twenty-One");
sortedDictionary = sortedDictionary.SetItem(10, "Diez");

// Выводит "10Diez", затем "21Twenty-One".
foreach (KeyValuePair<int, string> item in sortedDictionary)
  Trace.WriteLine(item.Key + item.Value);

string ten = sortedDictionary[10];
// ten == "Diez"
sortedDictionary = sortedDictionary.Remove(21);
```

Несортированные и отсортированные словари обладают сходным быстродействием, но я рекомендую использовать неупорядоченные словари, если только не требуется, чтобы элементы были отсортированы (табл. 9.4). Несортированные словари могут работать в целом немного быстрее. Кроме того, несортированные словари могут использоваться с любыми типами ключей, тогда как отсортированные словари требуют полной совместимости типов их ключей.

Таблица 9.4. Быстродействие неизменяемых словарей

Операция	ImmutableDictionary<TK,TV>	ImmutableSortedDictionary<TK,TV>
`Add`	O(log N)	O(log N)
`SetItem`	O(log N)	O(log N)
`Item[key]`	O(log N)	O(log N)
`Remove`	O(log N)	O(log N)

Пояснение

По опыту скажу, что словари являются полезным и общепринятым инструментом при работе с состоянием приложения. Они могут ис-

пользоваться в любых сценариях, связанных с ключами/значениями или подстановками.

Неизменяемые словари, как и другие неизменяемые коллекции, поддерживают механизм для эффективного построения словарей, содержащих большое количество элементов. Например, если исходные ссылочные данные загружаются в начале работы программы, вы сможете воспользоваться механизмом построителей для конструирования исходного неизменяемого словаря. С другой стороны, если ссылочные данные строятся постепенно во время выполнения, вероятно, можно будет воспользоваться обычным методом `Add` неизменяемых словарей.

`ImmutableDictionary<TK, TV>` и `ImmutableSortedDictionary<TK, TV>` находятся в пакете System.Collections.Immutable.

Дополнительная информация

В рецепте 9.5 рассматриваются потокобезопасные изменяемые словари.

В документации `ImmutableDictionary<TK,TV>.Builder` в MSDN (https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutabledictionary-2.builder?redirectedfrom=MSDN&view=netcore-3.0) рассматривается эффективный способ заполнения неизменяемых словарей.

В документации `ImmutableSortedDictionary<TK,TV>.Builder` в MSDN (https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablesorteddictionary-2.builder?redirectedfrom=MSDN&view=netcore-3.0) рассматривается эффективный способ заполнения неизменяемых отсортированных словарей.

9.5. Потокобезопасные словари

Задача

Имеется коллекция «ключ/значение» (например, кэш в памяти), которая должна поддерживаться в синхронизированном состоянии, даже если несколько потоков выполняют с ней операции чтения и записи.

Решение

Тип `ConcurrentDictionary<TKey, TValue>` в фреймворке .NET — настоящее сокровище среди структур данных. Он является потокобезопасным и использует сочетание детализированных блокировок и приемов, не использующих блокировки, которое гарантирует быстрый доступ в подавляющем большинстве сценариев.

Вероятно, понадобится какое-то время, чтобы привыкнуть к этому API. Он сильно отличается от стандартного типа `Dictionary<TKey, TValue>`, поскольку должен иметь дело с конкурентным доступом из многих потоков. Но после того как вы ознакомитесь с основами из этого рецепта, поймете, что `ConcurrentDictionary<TKey, TValue>` — один из самых полезных типов коллекций.

Для начала посмотрим, как записать значение в коллекцию. Чтобы задать значение для ключа, используйте метод `AddOrUpdate`:

```
var dictionary = new ConcurrentDictionary<int, string>();
string newValue = dictionary.AddOrUpdate(0,
    key => "Zero",
    (key, oldValue) => "Zero");
```

Метод `AddOrUpdate` выглядит сложно, так как должен делать несколько вещей в зависимости от текущего содержимого конкурентного словаря. В первом аргументе метода передается ключ. Во втором аргументе передается делегат, преобразующий ключ (в данном случае `0`) в значение, которое будет добавлено в словарь (в данном случае `"Zero"`). Этот делегат вызывается только в том случае, если ключ не существует в словаре. В третьем аргументе передается еще один делегат, преобразующий ключ (`0`) и старое значение в обновленное значение, которое должно быть сохранено в словаре (`"Zero"`). Этот делегат вызывается в том случае, если ключ уже существует в словаре. `AddOrUpdate` возвращает новое значение для этого ключа (то же значение, которое было возвращено одним из делегатов).

А теперь начинается то, от чего действительно голова может пойти кругом: чтобы конкурентный словарь работал правильно, *может* оказаться, что метод `AddOrUpdate` должен вызвать одного (или обоих) делегатов несколько раз. Такое бывает очень редко, но возможно. А значит, ваши делегаты должны быть простыми и быстрыми и не должны иметь побочных эффектов. Следовательно, делегаты должны только создавать

значение, не изменяя никакие другие переменные в вашем приложении. Этот принцип распространяется на всех делегатов, передаваемых методом `ConcurrentDictionary<TKey,TValue>`.

Есть несколько других способов добавления значений в словари. Один из упрощенных вариантов просто использует синтаксис индексирования:

```
// Используется тот же словарь "dictionary".
// Добавляет (или обновляет) ключ 0, связывая с ним значение "Zero".
dictionary[0] = "Zero";
```

Синтаксис индексирования обладает меньшими возможностями; он не предоставляет возможности обновления значений на основании существующего значения. Впрочем, этот синтаксис проще и он нормально работает, если вам уже известно значение, которое требуется сохранить в словаре.

Теперь посмотрим, как выполняется чтение значений. Это легко делается методом `TryGetValue`:

```
// Используется тот же словарь "dictionary".
bool keyExists = dictionary.TryGetValue(0, out string currentValue);
```

`TryGetValue` вернет `true` и задаст значение, если ключ был найден в словаре. Если ключ не найден, `TryGetValue` вернет `false`. Синтаксис индексирования также может использоваться для чтения значений, но я считаю, что в данной ситуации он уже не столь полезен, потому что при отсутствии ключа будет выдано исключение. Помните, что в конкурентном словаре несколько потоков могут заниматься чтением, обновлением, добавлением и удалением значений; во многих ситуациях бывает трудно проверить, существует ключ или нет, до того как вы попытаетесь прочитать его.

Удаление значений выполняется так же просто, как и их чтение:

```
// Используется тот же словарь "dictionary".
bool keyExisted = dictionary.TryRemove(0, out string removedValue);
```

Метод `TryRemove` почти идентичен `TryGetValue`, не считая того, что он удаляет пару «ключ/значение», если ключ был обнаружен в словаре.

Пояснение

Хотя тип `ConcurrentDictionary<TKey, TValue>` является потокобезопасным, это не означает атомарности его операций. Если несколько потоков вызы-

вают AddOrUpdate конкурентно, может оказаться, что два потока обнаружат отсутствие ключа, а затем оба одновременно выполнят своего делегата, создающего новое значение.

Я считаю, что ConcurrentDictionary<TKey, TValue> — замечательный тип прежде всего из-за невероятно мощного метода AddOrUpdate. Тем не менее он подходит не для каждой ситуации. ConcurrentDictionary<TKey, TValue> хорошо работает при чтении и записи со стороны нескольких потоков в общую коллекцию. Если обновления не выполняются постоянно (т. е. эта операция относительно редка), то, возможно, ImmutableDictionary<TKey, TValue> будет более подходящим кандидатом.

Тип ConcurrentDictionary<TKey, TValue> лучше подходит для ситуаций с общими данными, когда несколько потоков совместно используют одну коллекцию. Если некоторые потоки только добавляют элементы, а другие только удаляют их, возможно, вам лучше подойдет коллекция «производитель/потребитель».

ConcurrentDictionary<TKey, TValue> — не единственная потокобезопасная коллекция. BCL также предоставляет типы ConcurrentStack<T>, ConcurrentQueue<T> и ConcurrentBag<T>.

Потокобезопасные коллекции часто используются в качестве коллекций «производитель/потребитель», которые будут рассмотрены далее в этой главе.

Дополнительная информация

В рецепте 9.4 рассматриваются неизменяемые словари, идеально подходящие для ситуаций, в которых содержимое словаря изменяется очень редко.

9.6. Блокирующие очереди

Задача

Необходимо создать коммуникационный канал для передачи сообщений или данных между потоками. Например, один поток может загружать данные, которые отправляются по каналу по мере загрузки; другие потоки на стороне получения получают эти данные и обрабатывают их.

Решение

Тип .NET `BlockingCollection<T>` проектировался для создания таких коммуникационных каналов. По умолчанию `BlockingCollection<T>` работает в режиме блокирующей очереди и предоставляет поведение «первым зашел, первым вышел».

Блокирующая очередь должна совместно использоваться несколькими потоками, и обычно определяется как приватное поле, доступное только для чтения:

```
private readonly BlockingCollection<int> _blockingQueue =
    new BlockingCollection<int>();
```

Обычно поток делает что-то одно: либо добавляет элементы в коллекцию, либо удаляет элементы. Потоки, добавляющие элементы, называются *потоками-производителями*, а потоки, удаляющие элементы, называются *потоками-потребителями*.

Потоки-производители могут добавлять элементы вызовами `Add`, а когда поток-производитель завершится (когда будут добавлены все элементы), он может завершить коллекцию вызовом `CompleteAdding`. Тем самым он уведомляет коллекцию о том, что элементы далее добавляться не будут, а коллекция может сообщить своим потребителям, что элементов больше не будет.

В следующем простом примере производитель добавляет два элемента, а потом помечает коллекцию как завершенную:

```
_blockingQueue.Add(7);
_blockingQueue.Add(13);
_blockingQueue.CompleteAdding();
```

Потоки-потребители обычно выполняются в цикле, ожидая следующего элемента и выполняя его последующую обработку. Если выделить код производителя в отдельный поток (например, вызовом `Task.Run`), то эти элементы можно будет потреблять следующим образом:

```
// Выводит "7", затем "13".
foreach (int item in _blockingQueue.GetConsumingEnumerable())
  Trace.WriteLine(item);
```

Если потребителей должно быть несколько, `GetConsumingEnumerable` может вызываться из нескольких потоков одновременно. Тем не менее каждый элемент передается только одному из этих потоков. При завершении коллекции завершается и перечисляемый объект.

Пояснение

Во всех приведенных примерах `GetConsumingEnumerable` используется для потоков-потребителей; это самая распространенная ситуация. Но существует и метод `Take`, который позволяет потребителю получить только один элемент (вместо потребления всех элементов в цикле).

При использовании таких коммуникационных каналов необходимо подумать о том, что произойдет, если производители работают быстрее потребителей. Если элементы производятся быстрее, чем потребляются, возможно, придется применить регулировку очереди.

Блокирующие очереди хорошо работают при наличии отдельного потока (например, из пула потоков), действующего как производитель или потребитель. Они не настолько хороши, если вы хотите обращаться к коммуникационному каналу асинхронно — например, если UI-поток должен действовать в режиме потребителя. Асинхронные очереди рассматриваются в рецепте 9.8.

Если вы вводите в свое приложение подобный коммуникационный канал, подумайте о переходе на библиотеку TPL Dataflow. Во многих случаях решение с использованием TPL Dataflow проще самостоятельного построения коммуникационных каналов и фоновых потоков.

Тип `BufferBlock<T>` из TPL Dataflow может работать как блокирующая очередь, к тому же TPL Dataflow позволяет построить конвейер или сеть для обработки. Впрочем, во многих простых случаях обычные блокирующие очереди (например, `BlockingCollection<T>`) станут более подходящим вариантом при проектировании.

Также можно воспользоваться типом `AsyncProducerConsumerQueue<T>` библиотеки `AsyncEx`, который может работать как блокирующая очередь.

Дополнительная информация

В рецепте 9.7 рассматриваются блокирующие стеки и мультимножества на случай, если вам потребуются сходные коммуникационные каналы без семантики «первым зашел, первым вышел».

В рецепте 9.8 рассматриваются очереди, имеющие асинхронный API вместо блокирующего.

В рецепте 9.12 рассматриваются очереди, имеющие как асинхронный, так и блокирующий API.

В рецепте 9.9 рассматриваются очереди с регулировкой количества элементов.

9.7. Блокирующие стеки и мультимножества

Задача

Требуется коммуникационный канал для передачи сообщений или данных из одного потока в другой, но вы не хотите, чтобы этот канал использовал семантику FIFO.

Решение

Тип .NET `BlockingCollection<T>` по умолчанию работает как блокирующая очередь, но он также может работать как любая другая коллекция «производитель/потребитель». По сути это обертка для потокобезопасной коллекции, реализующей `IProducerConsumerCollection<T>`.

Таким образом, вы можете создать `BlockingCollection<T>` с семантикой LIFO или семантикой неупорядоченного мультимножества:

```
BlockingCollection<int> _blockingStack = new BlockingCollection<int>(
    new ConcurrentStack<int>());
BlockingCollection<int> _blockingBag = new BlockingCollection<int>(
    new ConcurrentBag<int>());
```

Важно учитывать, что с упорядочением элементов связаны некоторые условия гонки. Если вы позволите тому же коду-производителю отработать ранее любой код-потребитель, а затем выполните код-потребитель после кода-производителя, порядок элементов будет в точности таким же, как у стека:

```
// Код-производитель
_blockingStack.Add(7);
_blockingStack.Add(13);
_blockingStack.CompleteAdding();

// Код-потребитель
// Выводит "13", затем "7".
foreach (int item in _blockingStack.GetConsumingEnumerable())
  Trace.WriteLine(item);
```

Если код-производитель и код-потребитель выполняются в разных потоках (как это обычно бывает), потребитель всегда получает следующим тот элемент, который был добавлен последним. Например, производитель добавляет 7, потребитель получает 7, затем производитель добавляет 13, потребитель получает 13. Потребитель *не* ожидает вызова `CompleteAdding` перед тем, как вернуть первый элемент.

Пояснение

Все, чтобы было сказано о регулировке применительно к блокирующим очередям, также применимо к блокирующим стекам или мультимножествам. Если ваши производители работают быстрее потребителей и вы хотите ограничить использование памяти блокирующим стеком/очередью, используйте регулировку так, как показано в рецепте 9.9.

В этом рецепте для кода-потребителя используется `GetConsumingEnumerable` — самый распространенный сценарий. Также существует метод `Take`, который позволяет потребителю получить только один элемент (вместо потребления всех элементов).

Если вы хотите обращаться к совместно используемым стекам или мультимножествам асинхронно (например, если UI-поток должен действовать в режиме потребителя), обращайтесь к рецепту 9.11.

Дополнительная информация

В рецепте 9.6 рассматриваются блокирующие очереди, которые используются намного чаще блокирующих стеков или мультимножеств.

В рецепте 9.11 рассматриваются асинхронные стеки и мультимножества.

9.8. Асинхронные очереди

Задача

Требуется коммуникационный канал для передачи сообщений или данных из одной части кода в другую по принципу FIFO без блокирования потоков.

Например, один фрагмент кода может загружать данные, которые отправляются по каналу по мере загрузки; при этом UI-поток получает данные и выводит их.

Решение

Требуется очередь с асинхронным API. В базовом фреймворке .NET такого типа нет, но NuGet предоставляет пару возможных решений.

Во-первых, вы можете использовать Channels. Channels — современная библиотека для асинхронных коллекций «производитель/потребитель», уделяющая особое внимание высокому быстродействию в крупномасштабных сценариях. Производители обычно записывают элементы в канал вызовом `WriteAsync`, а когда они завершат производство элементов, один из них вызывает `Complete` для уведомления канала о том, что в дальнейшем элементов больше не будет:

```
Channel<int> queue = Channel.CreateUnbounded<int>();

// Код-производитель
ChannelWriter<int> writer = queue.Writer;
await writer.WriteAsync(7);
await writer.WriteAsync(13);
writer.Complete();
```

```
// Код-потребитель
// Выводит "7", затем "13".
ChannelReader<int> reader = queue.Reader;
await foreach (int value in reader.ReadAllAsync())
  Trace.WriteLine(value);
```

Более простой код-потребитель использует асинхронные потоки; дополнительную информацию см. в главе 3. На момент написания книги асинхронные потоки были доступны только на самых новых платформах .NET; старые платформы могут использовать следующий паттерн:

```
// Код-потребитель (старые платформы).
// Выводит "7", затем "13".
ChannelReader<int> reader = queue.Reader;
while (await reader.WaitToReadAsync())
  while (reader.TryRead(out int value))
    Trace.WriteLine(value);
```

Обратите внимание на двойной цикл `while` в коде-потребителе для старых платформ, это нормально. Метод `WaitToReadAsync` будет асинхронно ожидать до того, как элемент станет доступным, или канал будет помечен как завершенный; при наличии элемента, доступного для чтения, возвращается `true`. Метод `TryRead` пытается прочитать элемент (немедленно и синхронно), возвращая `true`, если элемент был прочитан. Если `TryRead` вернет `false`, это может объясняться тем, что прямо сейчас доступного элемента нет, или же тем, что канал был помечен как завершенный, и элементов больше вообще не будет. Таким образом, когда `TryRead` возвращает `false`, происходит выход из внутреннего цикла `while`, а потребитель снова вызывает метод `WaitToReadAsync`, который вернет `false`, если канал был помечен как завершенный.

Другой вариант организации очереди «производитель/потребитель» — использование `BufferBlock<T>` из библиотеки TPL Dataflow. Тип `BufferBlock<T>` имеет много общего с каналом. Следующий пример показывает, как объявить `BufferBlock<T>`, как выглядит код-производитель и как выглядит код-потребитель:

```
var _asyncQueue = new BufferBlock<int>();

// Код-производитель.
await _asyncQueue.SendAsync(7);
await _asyncQueue.SendAsync(13);
```

```
_asyncQueue.Complete();

// Код-потребитель.
// Выводит "7", затем "13".
while (await _asyncQueue.OutputAvailableAsync())
  Trace.WriteLine(await _asyncQueue.ReceiveAsync());
```

Код-потребитель использует метод `OutputAvailableAsync`, который на самом деле полезен только с одним потребителем. Если потребителей несколько, может случиться, что `OutputAvailableAsync` вернет `true` для нескольких потребителей, хотя элемент только один. Если очередь завершена, то `ReceiveAsync` выдаст исключение `InvalidOperationException`. Таким образом, с несколькими потребителями код будет выглядеть так:

```
while (true)
{
  int item;
  try
  {
    item = await _asyncQueue.ReceiveAsync();
  }
  catch (InvalidOperationException)
  {
    break;
  }
  Trace.WriteLine(item);
}
```

Также можно воспользоваться типом `AsyncProducerConsumerQueue<T>` из NuGet-библиотеки `Nito.AsyncEx`. Его API похож на API `BufferBlock<T>`, но не совпадает с ним полностью:

```
var _asyncQueue = new AsyncProducerConsumerQueue<int>();

// Код-производитель
await _asyncQueue.EnqueueAsync(7);
await _asyncQueue.EnqueueAsync(13);
_asyncQueue.CompleteAdding();
// Код-потребитель
// Выводит "7", затем "13".
while (await _asyncQueue.OutputAvailableAsync())
  Trace.WriteLine(await _asyncQueue.DequeueAsync());
```

В этом коде также используется метод `OutputAvailableAsync`, который обладает теми же недостатками, что и `BufferBlock<T>`. С несколькими потребителями код обычно выглядит примерно так:

```
while (true)
{
  int item;
  try
  {
    item = await _asyncQueue.DequeueAsync();
  }
  catch (InvalidOperationException)
  {
    break;
  }
  Trace.WriteLine(item);
}
```

Пояснение

Рекомендую использовать библиотеку Channels для асинхронных очередей «производитель/потребитель» там, где это возможно. Помимо регулировки поддерживаются несколько режимов выборки, а код тщательно оптимизирован. Однако, если логика вашего приложения может быть выражена в виде «конвейера», через который проходят данные, TPL Dataflow может быть более логичным кандидатом. Последний вариант — `AsyncProducerConsumerQueue<T>` — имеет смысл в том случае, если в вашем приложении уже используются другие типы из `AsyncEx`.

Библиотека Channels находится в пакете `System.Threading.Channels`, `BufferBlock<T>` — в пакете `System.Threading.Tasks.Dataflow`, а тип `AsyncProducerConsumerQueue<T>` — в пакете `Nito.AsyncEx`.

Дополнительная информация

В рецепте 9.6 рассматриваются очереди «производитель/потребитель» с блокирующей семантикой вместо асинхронной.

В рецепте 9.12 рассматриваются очереди «производитель/потребитель» как с блокирующей, так и с асинхронной семантикой.

В рецепте 9.7 рассматриваются асинхронные стеки и мультимножества, если вам нужен аналогичный коммуникационный канал с семантикой FIFO.

9.9. Регулировка очередей

Задача

Имеется очередь «производитель/потребитель», но производители могут работать быстрее потребителей, что может привести к неэффективному использованию памяти. Также вам хотелось бы сохранить все элементы в очереди, а следовательно, понадобится механизм регулировки производителей.

Решение

При использовании очередей «производитель/потребитель» необходимо учитывать, что произойдет, если производители работают быстрее потребителей (если только вы твердо не уверены в том, что потребители всегда работают быстрее). Если вы производите элементы быстрее, чем можете потреблять их, очередь придется отрегулировать. Для этого можно задать максимальное количество элементов. Когда очередь будет «заполнена», она применяет обратное давление к производителям, блокируя их до того, как в очереди не появится свободное место.

Регулировка может выполняться посредством создания ограниченного канала (вместо неограниченного). Так как каналы асинхронны, производители будут регулироваться асинхронно:

```
Channel<int> queue = Channel.CreateBounded<int>(1);
ChannelWriter<int> writer = queue.Writer;

// Эта запись завершается немедленно.
await writer.WriteAsync(7);
```

```
// Эта запись (асинхронно) ожидает удаления 7
// перед тем как ставить в очередь 13.
await writer.WriteAsync(13);

writer.Complete();
```

Тип `BufferBlock<T>` имеет встроенную поддержку регулировки, которая более подробно рассмотрена в рецепте 5.4. С блоками потоков данных следует задать параметр `BoundedCapacity`:

```
var queue = new BufferBlock<int>(
    new DataflowBlockOptions { BoundedCapacity = 1 });

// Эта отправка завершается немедленно.
await queue.SendAsync(7);

// Эта отправка (асинхронно) ожидает удаления 7
// перед тем как ставить в очередь 13.
await queue.SendAsync(13);

queue.Complete();
```

Производитель в этом фрагменте кода использует асинхронный API `SendAsync`; тот же подход работает и для синхронного API `Post`.

Тип `AsyncProducerConsumerQueue<T>` из `AsyncEx` содержит поддержку регулировки. Просто сконструируйте очередь с подходящим значением:

```
var queue = new AsyncProducerConsumerQueue<int>(maxCount: 1);

// Эта операция постановки в очередь завершается немедленно.
await queue.EnqueueAsync(7);

// Эта операция постановки в очередь (асинхронно) ожидает удаления 7
// перед тем как ставить в очередь 13.
await queue.EnqueueAsync(13);

queue.CompleteAdding();
```

Блокирующие очереди «производитель/потребитель» также поддерживают регулировку. Вы можете использовать тип `BlockingCollection<T>` для регулировки количества элементов, для чего при создании передается соответствующее значение:

```
var queue = new BlockingCollection<int>(boundedCapacity: 1);

// Это добавление завершается немедленно.
queue.Add(7);

// Это добавление ожидает удаления 7 перед тем, как добавлять 13.
queue.Add(13);

queue.CompleteAdding();
```

Пояснение

Регулировка необходима в том случае, если производители работают быстрее потребителей. Один из сценариев, которые необходимо рассмотреть: могут ли производители работать быстрее потребителей, если ваше приложение будет работать на другом оборудовании? Обычно некоторая регулировка потребуется для того, чтобы гарантировать нормальную работу на будущем оборудовании и/или облачных платформах, которые нередко более ограничены в ресурсах, чем машины разработчиков.

Регулировка создает обратное давление на производителей, замедляя их для того, чтобы потребители гарантированно могли обработать все элементы без создания излишних затрат памяти. Если обрабатывать каждый элемент не обязательно, можно использовать выборку вместо регулировки. Выборка из очередей «производитель/потребитель» рассматривается в рецепте 9.10.

Библиотека Channels находится в пакете `System.Threading.Channels`, тип `BufferBlock<T>` — в пакете `System.Threading.Tasks.Dataflow`, а тип `AsyncProducerConsumerQueue<T>` — в пакете `Nito.AsyncEx`.

Дополнительная информация

В рецепте 9.8 рассматривается базовое использование асинхронных очередей «производитель/потребитель».

В рецепте 9.6 рассматривается базовое использование синхронных очередей «производитель/потребитель».

В рецепте 9.10 рассматривается выборка в очередях «производитель/потребитель» как альтернатива регулировке.

9.10. Выборка в очередях

Задача

Есть очередь «производитель/потребитель», но производители могут работать быстрее потребителей, что может привести к неэффективному использованию памяти. Сохранять все элементы из очереди не обязательно; необходимо отфильтровать элементы очереди так, чтобы более медленные потребители могли ограничиться обработкой самых важных элементов.

Решение

Библиотека Channels предоставляет самые простые средства применения выборки к элементам ввода. Типичный пример — всегда брать последние n элементов с потерей самых старых элементов при заполнении очереди:

```
Channel<int> queue = Channel.CreateBounded<int>(
    new BoundedChannelOptions(1)
    {
      FullMode = BoundedChannelFullMode.DropOldest,
    });
ChannelWriter<int> writer = queue.Writer;

// Операция записи завершается немедленно.
await writer.WriteAsync(7);

// Операция записи тоже завершается немедленно.
// Элемент 7 теряется, если только он не был
// немедленно извлечен потребителем.
await writer.WriteAsync(13);
```

Это самый простой механизм контроля входных потоков и предотвращения «затопления» потребителей.

Есть и другие режимы `BoundedChannelFullMode`. Например, если вы хотите, чтобы самые старые элементы сохранялись, можно при заполнении канала терять новые элементы:

```
Channel<int> queue = Channel.CreateBounded<int>(
    new BoundedChannelOptions(1)
    {
      FullMode = BoundedChannelFullMode.DropWrite,
    });
ChannelWriter<int> writer = queue.Writer;

// Операция записи завершается немедленно.
await writer.WriteAsync(7);

// Операция записи тоже завершается немедленно.
// Элемент 13 теряется, если только элемент 7 не был
// немедленно извлечен потребителем.
await writer.WriteAsync(13);
```

Пояснение

Библиотека Channels отлично подходит для простой выборки. Во многих ситуациях чрезвычайно полезен режим `BoundedChannelFullMode.DropOldest`. Более сложная выборка должна выполняться самими потребителями.

Если выборка должна выполняться по времени (например, «только 10 элементов в секунду»), используйте System.Reactive. В System.Reactive предусмотрены естественные операторы для работы со временем.

Библиотека Channels находится в пакете `System.Threading.Channels`.

Дополнительная информация

В рецепте 9.9 рассматривается регулировка каналов, которая ограничивает количество элементов в канале посредством блокировки производителей вместо потери элементов.

В рецепте 9.8 рассматривается базовое использование каналов, включая код производителя и потребителя.

В рецепте 6.4 рассматриваются регулировка и выборка в библиотеке System.Reactive, которая поддерживает выборку по времени.

9.11. Асинхронные стеки и мультимножества

Задача

Требуется коммуникационный канал для передачи сообщений или данных из одной части кода в другую, но вы не хотите, чтобы этот канал использовал семантику FIFO.

Решение

Библиотека `Nito.AsyncEx` предоставляет тип `AsyncCollection<T>`, который по умолчанию работает как асинхронная очередь, но он также может работать как любая разновидность коллекций «производитель/потребитель». Обертка для `IProducerConsumerCollection<T>` — `AsyncCollection<T>` — также является `async`-эквивалентом типа .NET `BlockingCollection<T>`, который рассматривается в рецепте 9.7.

Тип `AsyncCollection<T>` поддерживает семантику LIFO (стек) или неупорядоченности (мультимножество) в зависимости от того, какая коллекция передается его конструктору:

```
var _asyncStack = new AsyncCollection<int>(
    new ConcurrentStack<int>());
var _asyncBag = new AsyncCollection<int>(
    new ConcurrentBag<int>());
```

Обратите внимание на состояние гонки в отношении упорядочения элементов в стеке. Если все производители завершатся до того, как начнут работать потребители, то порядок элементов будет соответствовать обычному стеку:

```
// Код-производитель
await _asyncStack.AddAsync(7);
await _asyncStack.AddAsync(13);
_asyncStack.CompleteAdding();
```

```
// Код-потребитель
// Выводит "13", затем "7".
while (await _asyncStack.OutputAvailableAsync())
  Trace.WriteLine(await _asyncStack.TakeAsync());
```

Если производители и потребители выполняются конкурентно (как это обычно бывает), потребитель всегда получает элемент, который был добавлен последним. Это приводит к тому, что поведение коллекции в целом не всегда соответствует поведению стека. Конечно, у мультимножества упорядочение вообще отсутствует.

В `AsyncCollection<T>` предусмотрена поддержка регулировки, которая необходима в тех случаях, когда производители могут добавлять данные в коллекцию быстрее, чем потребители их извлекают. Просто сконструируйте коллекцию с нужным значением:

```
var _asyncStack = new AsyncCollection<int>(
    new ConcurrentStack<int>(), maxCount: 1);
```

Теперь тот же код-производитель будет асинхронно ожидать по мере необходимости:

```
// Это добавление завершается немедленно.
await _asyncStack.AddAsync(7);

// Это добавление (асинхронно) ожидает удаления 7
// перед тем как помещать в очередь 13.
await _asyncStack.AddAsync(13);

_asyncStack.CompleteAdding();
```

Код-потребитель использует тип `OutputAvailableAsync`, на который распространяются ограничения, описанные в рецепте 9.8. При наличии нескольких потребителей код-потребитель обычно выглядит примерно так:

```
while (true)
{
  int item;
  try
  {
    item = await _asyncStack.TakeAsync();
  }
  catch (InvalidOperationException)
```

```
  {
    break;
  }
  Trace.WriteLine(item);
}
```

Пояснение

`AsyncCollection<T>` представляет собой асинхронный эквивалент `BlockingCollection<T>` с несколько отличающимся API.

Тип `AsyncCollection<T>` находится в пакете `Nito.AsyncEx`.

Дополнительная информация

В рецепте 9.8 рассматриваются асинхронные очереди, намного более распространенные, чем асинхронные стеки или мультимножества.

В рецепте 9.7 рассматриваются синхронные (блокирующие) стеки и мультимножества.

9.12. Блокирующие/асинхронные очереди

Задача

Требуется коммуникационный канал для передачи сообщений или данных из одной части кода в другую по принципу FIFO, но при этом необходима гибкость для того, чтобы сторона производителя или сторона потребителя могла рассматриваться как синхронная или асинхронная.

Например, фоновый поток может загружать данные и заносить их в коммуникационный канал, и вы хотите, чтобы фоновый поток синхронно блокировался при заполнении канала. В это время UI-поток получает данные из коммуникационного канала, и вы хотите, чтобы UI-поток асинхронно извлекал данные из канала, чтобы он продолжал реагировать на действия пользователя.

Решение

После знакомства с блокирующими очередями в рецепте 9.6 и асинхронными очередями в рецепте 9.8 мы изучим несколько типов очередей, поддерживающих как блокирующие, так и асинхронные API.

Начнем с типов `BufferBlock<T>` и `ActionBlock<T>` из NuGet-библиотеки TPL Dataflow. Тип `BufferBlock<T>` может легко использоваться как асинхронная очередь «производитель/потребитель» (за подробностями обращайтесь к рецепту 9.8):

```
var queue = new BufferBlock<int>();

// Код-производитель
await queue.SendAsync(7);
await queue.SendAsync(13);
queue.Complete();

// Для одного потребителя
while (await queue.OutputAvailableAsync())
  Trace.WriteLine(await queue.ReceiveAsync());

// Для нескольких потребителей
while (true)
{
  int item;
  try
  {
    item = await queue.ReceiveAsync();
  }
  catch (InvalidOperationException)
  {
    break;
  }

  Trace.WriteLine(item);
}
```

Как показано в следующем примере, `BufferBlock<T>` также поддерживает синхронный API для производителей и потребителей:

```
var queue = new BufferBlock<int>();

// Код-производитель
```

```
queue.Post(7);
queue.Post(13);
queue.Complete();

// Код-потребитель
while (true)
{
  int item;
  try
  {
    item = queue.Receive();
  }
  catch (InvalidOperationException)
  {
    break;
  }

  Trace.WriteLine(item);
}
```

Код-потребитель, использующий `BufferBlock<T>`, получается довольно неудобным, так как не соответствует стилю программирования потоков данных. Библиотека TPL Dataflow включает ряд блокировок, которые могут объединяться в цепочки для определения реактивной сети. В данном случае очередь «производитель/потребитель», завершающая конкретное действие, может определяться с помощью `ActionBlock<T>`:

```
// Код-потребитель передается конструктору очереди
ActionBlock<int> queue = new ActionBlock<int>(item =>
   Trace.WriteLine(item));
// Асинхронный код-производитель
await queue.SendAsync(7);
await queue.SendAsync(13);

// Синхронный код-производитель
queue.Post(7);
queue.Post(13);
queue.Complete();
```

Если библиотека TPL Dataflow недоступна на нужной вам платформе(-ах), в `Nito.AsyncEx` существует тип `AsyncProducerConsumerQueue<T>`, который также поддерживает как синхронные, так и асинхронные методы:

```
var queue = new AsyncProducerConsumerQueue<int>();

// Асинхронный код-производитель
await queue.EnqueueAsync(7);
await queue.EnqueueAsync(13);

// Синхронный код-производитель
queue.Enqueue(7);
queue.Enqueue(13);

queue.CompleteAdding();

// Для одного асинхронного потребителя
while (await queue.OutputAvailableAsync())
  Trace.WriteLine(await queue.DequeueAsync());

// Для нескольких асинхронных потребителей
while (true)
{
  int item;
  try
  {
    item = await queue.DequeueAsync();
  }
  catch (InvalidOperationException)
  {
    break;
  }
  Trace.WriteLine(item);
}

// Синхронный код-потребитель
foreach (int item in queue.GetConsumingEnumerable())
  Trace.WriteLine(item);
```

Пояснение

Рекомендую по возможности использовать `BufferBlock<T>` или `ActionBlock<T>`, потому что библиотека TPL Dataflow была протестирована более тщательно, чем библиотека `Nito.AsyncEx`. Однако тип `AsyncProducer-`

`ConsumerQueue<T>` тоже может пригодиться, если приложение уже использует другие типы из библиотеки `AsyncEx`.

`System.Threading.Channels` также можно использовать синхронно, но только косвенно. Их естественный API является асинхронным, но поскольку эти коллекции относятся к числу потокобезопасных, вы можете заставить их работать синхронно, упаковав код производства или потребления в `Task.Run` с последующим блокированием по задаче, возвращенной `Task.Run`:

```
Channel<int> queue = Channel.CreateBounded<int>(10);

// Код-производитель
ChannelWriter<int> writer = queue.Writer;
Task.Run(async () =>
{
  await writer.WriteAsync(7);
  await writer.WriteAsync(13);
  writer.Complete();
}).GetAwaiter().GetResult();

// Код-потребитель
ChannelReader<int> reader = queue.Reader;
Task.Run(async () =>
{
  while (await reader.WaitToReadAsync())
    while (reader.TryRead(out int value))
      Trace.WriteLine(value);
}).GetAwaiter().GetResult();
```

Блоки TPL Dataflow, `AsyncProducerConsumerQueue<T>` и Channels поддерживают регулировку, для чего при конструировании необходимо задать соответствующие параметры. Регулировка необходима в ситуациях, в которых производители заносят элементы в коллекцию быстрее, чем потребители могут потреблять их, в результате чего приложение будет расходовать слишком много памяти.

Типы `BufferBlock<T>` и `ActionBlock<T>` находятся в пакете `System.Threading.Tasks.Dataflow`. Тип `AsyncProducerConsumerQueue<T>` находится в пакете `Nito.AsyncEx`. Библиотека Channels находится в пакете `System.Threading.Channels`.

Дополнительная информация

В рецепте 9.6 рассматриваются блокирующие очереди «производитель/потребитель».

В рецепте 9.8 рассматриваются асинхронные очереди «производитель/потребитель».

В рецепте 5.4 рассматривается регулировка блоков потоков данных.

ГЛАВА 10

Отмена

В фреймворке .NET 4.0 появилась обширная и хорошо спроектированная поддержка отмены. Эта поддержка является кооперативной; это означает, что отмена может запрашиваться, но не осуществляется принудительно в коде. Вследствие кооперативной природы отмены невозможно отменить код, если он написан без учета поддержки отмены. По этой причине рекомендую включать поддержку отмены как можно в большей части вашего кода.

Отмена является разновидностью сигнала, в котором участвуют две стороны: *источник*, инициирующий отмену, и получатель, реагирующий на отмену. В .NET источником является объект `CancellationTokenSource`, а получателем — `CancellationToken`. В рецептах этой главы рассматриваются как источники, так и получатели отмены при нормальном использовании, а также описывается использование поддержки отмены для взаимодействия с ее нестандартными формами.

Отмена рассматривается как специальная разновидность ошибки. По действующим правилам, отмененный код инициирует исключение типа `OperationCanceledException` (или производного типа — например, `TaskCanceledException`). В этом случае вызывающий код знает, что отмена была замечена.

Чтобы сообщить вызывающему коду, что ваш метод поддерживает отмену, необходимо получить `CancellationToken` в параметре. Обычно такой параметр стоит на последнем месте, если только ваш метод не выдает уведомления о прогрессе (рецепт 2.3). Также можно рассмотреть возможность использования перегруженной версии или значения параметра по умолчанию для потребителей, не требующих отмены:

```
public void CancelableMethodWithOverload(CancellationToken
   cancellationToken)
{
  // Здесь размещается код.
}

public void CancelableMethodWithOverload()
{
  CancelableMethodWithOverload(CancellationToken.None);
}

public void CancelableMethodWithDefault(
    CancellationToken cancellationToken = default)
{
  // Здесь размещается код.
}
```

Значение `CancellationToken.None` представляет маркер отмены, который никогда не будет отменяться; это специальное значение, эквивалентное `default(CancellationToken)`. Потребители передают это значение, если необходимость отмены операции не возникнет никогда.

У асинхронных потоков реализован похожий, но более сложный способ отмены. Отмена асинхронных потоков подробно рассматривается в рецепте 3.4.

10.1. Выдача запросов на отмену

Задача

Из вашего кода вызывается другой код, который может отменяться (для чего используется `CancellationToken`). Требуется отменить вызванный код.

Решение

Тип `CancellationTokenSource` является источником для `CancellationToken`. Он позволяет коду реагировать на запросы отмены; компоненты `CancellationTokenSource` позволяют коду выдать запрос на отмену.

Каждый объект `CancellationTokenSource` существует независимо от всех остальных (если только вы не свяжете их так, как сделано в рецепте 10.8).

Свойство `Token` возвращает `CancellationToken` для этого источника, а метод `Cancel` выдает непосредственный запрос на отмену.

Следующий пример демонстрирует создание `CancellationTokenSource`, а также использование `Token` и `Cancel`. В коде используется `async`-метод, потому что его проще продемонстрировать в коротком примере; одна пара `Token`/`Cancel` используется для отмены *всех* видов кода:

```
void IssueCancelRequest()
{
  using var cts = new CancellationTokenSource();
  var task = CancelableMethodAsync(cts.Token);

  // В этой точке операция была запущена.

  // Выдать запрос на отмену.
  cts.Cancel();
}
```

В приведенном примере переменная `task` игнорируется после запуска; вероятно, в реальном приложении эта задача будет где-то сохранена и использована с `await`, чтобы пользователь знал о результате.

При отмене кода практически всегда возникает состояние гонки. Возможно, отменяемый код как раз собирался завершиться при выдаче запроса на отмену, и если перед завершением он не проверит свой маркер отмены, то код фактически завершится успешно. При отмене кода возможны три варианта: он может ответить на запрос на отмену (с выдачей исключения `OperationCanceledException`), он может завершиться успешно или же завершиться с ошибкой, не имеющей отношения к отмене (с выдачей другого исключения).

Следующий пример очень похож на предыдущий, не считая того, что он использует с задачей ключевое слово `await`, с демонстрацией всех трех возможных результатов:

```
async Task IssueCancelRequestAsync()
{
  using var cts = new CancellationTokenSource();
  var task = CancelableMethodAsync(cts.Token);

  // В этой точке операция выполняется.
```

```
  // Выдать запрос на отмену.
  cts.Cancel();
  // (Асинхронно) ожидать завершения операции.
  try
  {
    await task;
    // Если управление окажется в этой точке, значит, операция
    // была успешно завершена перед тем, как вступил в силу
    // запрос на отмену.
  }
  catch (OperationCanceledException)
  {
    // Если управление окажется в этой точке, значит, операция
    // была отменена до ее завершения.
  }
  catch (Exception)
  {
    // Если управление окажется в этой точке, значит, операция
    // завершилась с ошибкой перед тем как вступил в силу
    // запрос на отмену.
    throw;
  }
}
```

Обычно создание `CancellationTokenSource` и выдача запроса на отмену выполняются в разных методах. После того как вы отмените экземпляр `CancellationTokenSource`, он отменяется безвозвратно. Если понадобится другой источник, необходимо создать другой экземпляр. Ниже приведен более реалистичный пример приложения с графическим интерфейсом, в котором одна кнопка используется для запуска асинхронной операции, а другая кнопка — для ее отмены. Приложение также снимает и устанавливает блокировку кнопок `StartButton` и `CancelButton`, чтобы в любой момент времени выполнялась только одна операция:

```
private CancellationTokenSource _cts;

private async void StartButton_Click(object sender, RoutedEventArgs e)
{
  StartButton.IsEnabled = false;
  CancelButton.IsEnabled = true;
  try
```

```
  {
    _cts = new CancellationTokenSource();
    CancellationToken token = _cts.Token;
    await Task.Delay(TimeSpan.FromSeconds(5), token);
    MessageBox.Show("Delay completed successfully.");
  }
  catch (OperationCanceledException)
  {
    MessageBox.Show("Delay was canceled.");
  }
  catch (Exception)
  {
    MessageBox.Show("Delay completed with error.");
    throw;
  }
  finally
  {
    StartButton.IsEnabled = true;
    CancelButton.IsEnabled = false;
  }
}

private void CancelButton_Click(object sender, RoutedEventArgs e)
{
  _cts.Cancel();
  CancelButton.IsEnabled = false;
}
```

Пояснение

В самом реалистичном примере этого рецепта используется GUI-интерфейс, но не стоит думать, будто отмена предназначена исключительно для пользовательских интерфейсов. Отмена также находит применение на сервере; например, ASP.NET предоставляет маркер отмены, представляющий тайм-аут запроса или отсоединение клиента. Правда, источники маркера запроса на стороне сервера встречаются реже, однако ничего не мешает их использованию; они могут пригодиться, если вам понадобилось выполнить отмену по некоторой причине, на которую не распространяется отмена ASP.NET, — например, дополнительного тайм-аута для части обработки запроса.

Дополнительная информация

В рецепте 10.4 рассматривается передача маркеров `async`-коду.

В рецепте 10.5 рассматривается передача маркеров параллельному коду.

В рецепте 10.6 рассматривается использование маркеров в реактивном коде.

В рецепте 10.7 рассматривается передача маркеров сетям потоков данных.

10.2. Реагирование на запросы на отмену посредством периодического опроса

Задача

В коде имеется цикл, который должен поддерживать отмену.

Решение

Если в коде присутствует цикл обработки, то в нем нет низкоуровневых функций API, которым можно было передать `CancellationToken`. В этом случае необходимо периодически проверять, не был ли отменен маркер. Следующий пример периодически проверяет маркер в ходе выполнения цикла, создающего вычислительную нагрузку на процессор:

```
public int CancelableMethod(CancellationToken cancellationToken)
{
  for (int i = 0; i != 100; ++i)
  {
    Thread.Sleep(1000); // Некоторые вычисления.
    cancellationToken.ThrowIfCancellationRequested();
  }
  return 42;
}
```

Если тело цикла выполняется очень быстро, то, возможно, стоит ограничить частоту проверки маркера отмены. Как обычно, измерьте быстродействие до и после таких изменений, чтобы решить, какой из вариан-

тов является лучшим. Следующий пример похож на предыдущий, но выполняет больше итераций более быстрого цикла, поэтому я добавил ограничение на частоту проверки маркера:

```
public int CancelableMethod(CancellationToken cancellationToken)
{
  for (int i = 0; i != 100000; ++i)
  {
    Thread.Sleep(1); // Некоторые вычисления.
    if (i % 1000 == 0)
      cancellationToken.ThrowIfCancellationRequested();
  }
  return 42;
}
```

Предельное значение зависит исключительно от того, какой объем работы выполняется и насколько быстрой должна быть реакция на отмену.

Пояснение

В большинстве случаев ваш код должен просто передать `CancellationToken` на следующий уровень. Примеры такого рода встречаются в рецептах 10.4–10.7. Метод периодического опроса (polling), использованный в этом рецепте, следует применять только в том случае, если у вас имеется вычислительный цикл, который должен поддерживать отмену.

У типа `CancellationToken` имеется другой метод `IsCancellationRequested`, который начинает возвращать `true` при отмене маркера. Некоторые разработчики используют его для реакции на отмену, обычно возвращая значение по умолчанию или `null`. Я не рекомендую использовать этот метод в большей части кода. В стандартном паттерне отмены выдается исключение `OperationCanceledException`, для чего вызывается метод `ThrowIfCancellationRequested`. Если код, находящийся выше в стеке, захочет перехватить исключение и действовать так, словно результат равен `null`, это нормально, но любой код, получающий `CancellationToken`, должен следовать стандартному паттерну отмены. Если вы решите не соблюдать паттерн отмены, по крайней мере четко документируйте свои намерения.

Работа метода `ThrowIfCancellationRequested` основана на периодическом опросе маркера отмены; ваш код должен вызывать его с регулярными

интервалами. Также существует способ регистрации обратного вызова, который вызывается при запросе на отмену. Решение с обратным вызовом в большей степени ориентировано на взаимодействие с другими системами отмены; в рецепте 10.9 рассматривается использование обратных вызовов с отменой.

Дополнительная информация

В рецепте 10.4 рассматривается передача маркеров `async`-коду.

В рецепте 10.5 рассматривается передача маркеров параллельному коду.

В рецепте 10.6 рассматривается использование маркеров с реактивным кодом.

В рецепте 10.7 рассматривается передача маркеров сетям потоков данных.

В рецепте 10.9 рассматривается использование обратных вызовов вместо периодического опроса для реакции на запросы на отмену.

В рецепте 10.1 рассматривается выдача запросов на отмену.

10.3. Отмена по тайм-ауту

Задача

Имеется код, который должен остановить выполнение после тайм-аута.

Решение

Отмена — очевидное решение для ситуаций с тайм-аутом. Тайм-аут — всего лишь одна из разновидностей запроса на отмену. Код, который необходимо отменить, просто отслеживает маркер отмены, как и при любой другой отмене; ему не нужно знать, что источником отмены является таймер.

Также у источников маркеров отмены существуют вспомогательные методы, которые автоматически выдают запрос на отмену по таймеру. Тайм-аут, кроме того, можно передать конструктору:

```
async Task IssueTimeoutAsync()
{
  using var cts = new CancellationTokenSource
     (TimeSpan.FromSeconds(5));
  CancellationToken token = cts.Token;
  await Task.Delay(TimeSpan.FromSeconds(10), token);
}
```

Если у вас уже имеется экземпляр `CancellationTokenSource`, можно запустить тайм-аут для этого экземпляра:

```
async Task IssueTimeoutAsync()
{
  using var cts = new CancellationTokenSource();
  CancellationToken token = cts.Token;
  cts.CancelAfter(TimeSpan.FromSeconds(5));
  await Task.Delay(TimeSpan.FromSeconds(10), token);
}
```

Пояснение

Чтобы выполнить код с тайм-аутом, используйте `CancellationTokenSource` и `CancelAfter` (или конструктор). Той же цели можно добиться другими способами, но использование существующей системы отмены — самый простой и эффективный вариант.

Помните, что отменяемый код должен отслеживать состояние маркера отмены. Вам не удастся легко отменить код, для которого отмена не предусмотрена.

Дополнительная информация

В рецепте 10.4 рассматривается передача маркеров `async`-коду.

В рецепте 10.5 рассматривается передача маркеров параллельному коду.

В рецепте 10.6 рассматривается использование маркеров с реактивным кодом.

В рецепте 10.7 рассматривается передача маркеров сетям потоков данных.

10.4. Отмена async-кода

Задача

Вы используете `async`-код, для которого нужно обеспечить возможность отмены.

Решение

Простейший способ поддержки отмены в асинхронном коде — просто передать `CancellationToken` на следующий уровень. Следующий пример выполняет асинхронную задержку, после чего возвращает значение; для поддержки отмены маркер передается `Task.Delay`:

```
public async Task<int> CancelableMethodAsync(CancellationToken
   cancellationToken)
{
  await Task.Delay(TimeSpan.FromSeconds(2), cancellationToken);
  return 42;
}
```

Многие асинхронные API поддерживают `CancellationToken`, поэтому обеспечение отмены обычно сводится к простой передаче маркера. Как правило, если ваш метод вызывает функции API, получающие `CancellationToken`, то ваш метод также должен получать `CancellationToken` и передавать его всем функциям API, которые его поддерживают.

Пояснение

К сожалению, некоторые методы не поддерживают отмену. Если вы оказались в такой ситуации, простого решения не существует. Невозможно безопасно остановить произвольный код, если только он не упакован в отдельный исполняемый модуль. Если ваш код вызывает код, не поддерживающий отмену и вы не хотите упаковывать этот код в отдельный исполняемый модуль, всегда можно имитировать отмену, просто игнорируя результат.

Отмена должна предоставляться как вариант там, где это возможно. Дело в том, что правильно реализованная отмена на высоком уровне

зависит от правильно реализованной отмены на нижнем уровне. Таким образом, когда вы пишете собственные `async`-методы, постарайтесь как можно тщательнее обеспечить поддержку отмены. Никогда неизвестно заранее, какие высокоуровневые методы будут вызывать ваш код, и им тоже может понадобиться отмена.

Дополнительная информация

В рецепте 10.1 рассматривается выдача запроса на отмену.

В рецепте 10.3 рассматривается использование отмены в качестве тайм-аута.

10.5. Отмена параллельного кода

Задача

Вы используете параллельный код, для которого нужно обеспечить возможность отмены.

Решение

Простейший способ поддержки отмены — передача `CancellationToken` параллельному коду. Параллельные методы поддерживают эту возможность посредством получения экземпляра `ParallelOptions`. Установка `CancellationToken` для экземпляра `ParallelOptions` выполняется так:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees,
    CancellationToken token)
{
  Parallel.ForEach(matrices,
      new ParallelOptions { CancellationToken = token },
      matrix => matrix.Rotate(degrees));
}
```

Также возможно отслеживать `CancellationToken` непосредственно в теле цикла:

```
void RotateMatrices2(IEnumerable<Matrix> matrices, float degrees,
    CancellationToken token)
{
  // Предупреждение: так поступать не рекомендуется; см. ниже.
  Parallel.ForEach(matrices, matrix =>
  {
    matrix.Rotate(degrees);
    token.ThrowIfCancellationRequested();
  });
}
```

Альтернативное решение требует большего объема работы и не так хорошо интегрируется, потому что параллельный цикл упаковывает `OperationCanceledException` в `AggregateException`. Кроме того, если `CancellationToken` передается в составе экземпляра `ParallelOptions`, класс `Parallel` сможет принять более разумные решения относительно частоты проверки маркера. По этим причинам маркер лучше передавать в параметре. В этом случае маркер *также* можно передать в тело цикла, но не следует *только* передавать маркер в тело цикла.

В Parallel LINQ (PLINQ) также предусмотрена встроенная поддержка отмены с оператором `WithCancellation`:

```
IEnumerable<int> MultiplyBy2(IEnumerable<int> values,
    CancellationToken cancellationToken)
{
  return values.AsParallel()
      .WithCancellation(cancellationToken)
      .Select(item => item * 2);
}
```

Пояснение

Поддержка отмены для параллельной работы — важный критерий хорошего пользовательского интерфейса. Если ваше приложение выполняет параллельную работу, оно создает серьезную нагрузку на процессор в течение хотя бы короткого времени. Высокий уровень использования процессора обычно заметен для пользователей, даже если не мешает работе других приложений на той же машине. Таким образом, я рекомендую поддерживать отмену при любых параллельных вычислениях (или любой другой работе, связанной с интенсивной нагрузкой на процессор), даже если общее время высокой нагрузки на процессор относительно невелико.

Дополнительная информация

В рецепте 10.1 рассматривается выдача запроса на отмену.

10.6. Отмена кода System.Reactive

Задача

Имеется реактивный код, для которого нужно обеспечить возможность отмены.

Решение

В библиотеке System.Reactive предусмотрена концепция подписки на наблюдаемый поток. Ваш код может освободить подписку на поток. Во многих случаях достаточно логического освобождения потока. Например, следующий код подписывается на щелчки кнопкой мыши при нажатии одной кнопки и освобождает подписку при нажатии другой кнопки:

```
private IDisposable _mouseMovesSubscription;

private void StartButton_Click(object sender, RoutedEventArgs e)
{
  IObservable<Point> mouseMoves = Observable
      .FromEventPattern<MouseEventHandler, MouseEventArgs>(
          handler => (s, a) => handler(s, a),
          handler => MouseMove += handler,
          handler => MouseMove -= handler)
      .Select(x => x.EventArgs.GetPosition(this));
  _mouseMovesSubscription = mouseMoves.Subscribe(value =>
  {
    MousePositionLabel.Content = "(" + value.X + ", " + value.Y + ")";
  });
}

private void CancelButton_Click(object sender, RoutedEventArgs e)
{
  if (_mouseMovesSubscription != null)
```

```
    _mouseMovesSubscription.Dispose();
}
```

System.Reactive довольно удобно использовать вместе с системой `CancellationTokenSource`/`CancellationToken`, повсеместно применяемой для отмены. В оставшейся части этого рецепта рассматриваются возможности взаимодействия наблюдаемых объектов System.Reactive с `CancellationToken`.

Первый сценарий — наблюдаемый код, упакованный в асинхронный код. Базовое решение было рассмотрено в рецепте 8.5, и теперь вы хотите добавить поддержку `CancellationToken`. В общем случае проще всего выполнить все операции с использованием реактивных операторов, а затем вызвать `ToTask` для преобразования последнего полученного элемента в задачу, допускающую ожидание. Следующий пример показывает, как асинхронно получить последний элемент в последовательности:

```
CancellationToken cancellationToken = ...
IObservable<int> observable = ...
int lastElement = await
   observable.TakeLast(1).ToTask(cancellationToken);
// или: int lastElement = await observable.ToTask(cancellationToken);
```

Получение первого элемента выглядит очень похоже; просто измените наблюдаемый объект перед вызовом `ToTask`:

```
CancellationToken cancellationToken = ...
IObservable<int> observable = ...
int firstElement = await
   observable.Take(1).ToTask(cancellationToken);
```

Асинхронное преобразование всей наблюдаемой последовательности в задачу тоже происходит аналогично:

```
CancellationToken cancellationToken = ...
IObservable<int> observable = ...
IList<int> allElements = await
   observable.ToList().ToTask(cancellationToken);
```

Наконец, рассмотрим обратную ситуацию. Мы рассмотрели несколько решений для ситуаций, в которых код System.Reactive реагирует на `CancellationToken`, т. е. где запрос отмены `CancellationTokenSource` преобразуется в освобождение подписки. Также можно пойти в другом направлении: выдать запрос на отмену как реакцию на освобождение подписки.

Операторы `FromAsync`, `StartAsync` и `SelectMany` поддерживают отмену, как показано в рецепте 8.6. Этих операторов достаточно для большинства практических ситуаций. Rx также предоставляет тип `CancellationDisposable`, который отменяет `CancellationToken` при освобождении. Вы можете использовать `CancellationDisposable` напрямую:

```
using (var cancellation = new CancellationDisposable())
{
  CancellationToken token = cancellation.Token;
  // Маркер передается методам, которые на него реагируют.
}
// В этой точке маркер отменяется.
```

Пояснение

В System.Reactive (Rx) есть собственная концепция отмены: освобождение подписок. В этом рецепте рассматриваются различные способы интегрировать Rx в универсальную структуру отмены, появившуюся в .NET 4.0. Пока вы находитесь в той части вашего кода, которая относится к миру Rx, используйте систему подписки/освобождения Rx; чтобы решение работало как следует, вводите поддержку `CancellationToken` только на границах.

Дополнительная информация

В рецепте 8.5 рассматриваются асинхронные обертки для кода Rx (без поддержки отмены).

В рецепте 8.6 рассматриваются обертки Rx для асинхронного кода (с поддержкой отмены).

В рецепте 10.1 рассматривается выдача запросов на отмену.

10.7. Отмена сетей потоков данных

Задача

Вы используете сети потоков данных. Требуется обеспечить поддержку отмены.

Решение

Лучшим способом поддержки отмены в вашем коде будет сквозная передача `CancellationToken` функциям API, поддерживающим отмену. У каждого блока в сети потока данных поддержка отмены является частью `DataflowBlockOptions`. Если вы захотите дополнить нестандартный блок данных поддержкой отмены, задайте свойство `CancellationToken` в параметрах блока:

```
IPropagatorBlock<int, int> CreateMyCustomBlock(
    CancellationToken cancellationToken)
{
  var blockOptions = new ExecutionDataflowBlockOptions
  {
    CancellationToken = cancellationToken
  };
  var multiplyBlock = new TransformBlock<int, int>(item => item * 2,
      blockOptions);
  var addBlock = new TransformBlock<int, int>(item => item + 2,
      blockOptions);
  var divideBlock = new TransformBlock<int, int>(item => item / 2,
      blockOptions);

  var flowCompletion = new DataflowLinkOptions
  {
    PropagateCompletion = true
  };
  multiplyBlock.LinkTo(addBlock, flowCompletion);
  addBlock.LinkTo(divideBlock, flowCompletion);

  return DataflowBlock.Encapsulate(multiplyBlock, divideBlock);
}
```

В этом примере я применил `CancellationToken` к каждому блоку в сети, хотя это не является строго необходимым. Так как завершение тоже распространяется по связям, я мог применить его к первому блоку и разрешить ему распространяться. Отмены считаются особой формой ошибок, поэтому блоки, находящиеся далее в конвейере, будут завершаться с ошибкой по мере распространения ошибки. С другой стороны, если я отменяю сеть потоков данных, то предпочту отменить все блоки одновременно и в этом случае обычно задаю параметр `CancellationToken` для каждого блока.

Пояснение

В сетях потоков данных отмена *не является* разновидностью сброса (flush). Когда блок отменяется, он теряет все свои входные данные и отказывается принимать новые элементы. Таким образом, если вы отменяете блок во время выполнения, это *приведет* к потере данных.

Дополнительная информация

В рецепте 10.1 рассматривается выдача запросов на отмену.

10.8. Внедрение запросов на отмену

Задача

В коде присутствует уровень, который должен реагировать на запросы на отмену, а также выдавать собственные запросы на отмену на следующий уровень.

Решение

В системе отмены .NET 4.0 предусмотрена встроенная поддержка этого сценария в виде *связанных маркеров отмены*. Источник маркера отмены может быть создан связанным с одним (или несколькими) существующими маркерами. Когда вы создаете источник связанного маркера отмены, полученный маркер будет отменяться при отмене любых из существующих маркеров или при явной отмене связанного источника.

Следующий пример выполняет асинхронный запрос HTTP. Маркер, переданный методу `GetWithTimeoutAsync`, представляет отмену, запрошенную конечным пользователем, а метод `GetWithTimeoutAsync` также применяет тайм-аут к запросу:

```
async Task<HttpResponseMessage> GetWithTimeoutAsync(HttpClient client,
    string url, CancellationToken cancellationToken)
{
  using CancellationTokenSource cts = CancellationTokenSource
      .CreateLinkedTokenSource(cancellationToken);
  cts.CancelAfter(TimeSpan.FromSeconds(2));
```

```
  CancellationToken combinedToken = cts.Token;

  return await client.GetAsync(url, combinedToken);
}
```

Полученный маркер `combinedToken` отменяется либо когда пользователь отменяет существующий маркер `cancellationToken`, либо при отмене связанного источника вызовом `CancelAfter`.

Пояснение

Хотя в предыдущем примере используется только один источник `CancellationToken`, метод `CreateLinkedTokenSource` может получать любое количество маркеров отмены в своих параметрах. Это позволяет вам создать один объединенный маркер, на базе которого можно реализовать собственную логическую отмену. Например, ASP.NET предоставляет маркер отмены, представляющий отключение пользователя (`HttpContext.RequestAborted`); код обработчика может создать связанный маркер, который реагирует либо на отключение пользователя, либо на свои причины отмены (например, тайм-аут).

Помните о сроке существования источника связанного маркера отмены. Предыдущий пример является наиболее типичным: один или несколько маркеров отмены передаются методу, который связывает их и передает как комбинированный маркер. Также обратите внимание на то, что в примере используется команда `using`, которая гарантирует, что источник связанного маркера отмены будет освобожден, когда операция будет завершена (а комбинированный маркер перестанет использоваться). Подумайте, что произойдет, если код не освободит источник связанного маркера отмены: может оказаться, что метод `GetWithTimeoutAsync` будет вызван несколько раз с одним (долгосрочным) существующим маркером; в этом случае код будет связывать новый источник маркера при каждом вызове метода. Даже после того, как запросы HTTP завершатся (и ничто не будет использовать комбинированный маркер), этот связанный источник все еще остается присоединенным к существующему маркеру. Чтобы предотвратить подобные утечки памяти, освободите источник связанного маркера отмены, когда комбинированный маркер перестанет быть нужным.

Дополнительная информация

В рецепте 10.1 рассматривается общий механизм выдачи запросов на отмену.

В рецепте 10.3 рассматривается использование отмены по тайм-ауту.

10.9. Взаимодействие с другими системами отмены

Задача

Имеется внешний или унаследованный код с собственными концепциями отмены. Требуется управлять им с использованием стандартного объекта `CancellationToken`.

Решение

У типа `CancellationToken` существует два основных способа реакции на запрос на отмену: периодический опрос (рассматривается в рецепте 10.2) и обратные вызовы (тема этого рецепта). Периодический опрос обычно используется для кода, интенсивно использующего процессор, — например, циклов обработки данных; обратные вызовы обычно используются во всех остальных ситуациях. Регистрация обратного вызова для маркера осуществляется методом `CancellationToken.Register`.

Допустим, вы пишете обертку для `System.Net.NetworkInformation.Pingtype` и хотите предусмотреть возможность отмены тестового опроса. Класс `Ping` уже имеет API на базе `Task`, но не поддерживает `CancellationToken`. Вместо этого тип `Ping` содержит собственный метод `SendAsyncCancel`, который может использоваться для отмены. Для этого зарегистрируйте обратный вызов, который активизирует этот метод:

```
async Task<PingReply> PingAsync(string hostNameOrAddress,
    CancellationToken cancellationToken)
{
  using var ping = new Ping();
  Task<PingReply> task = ping.SendPingAsync(hostNameOrAddress);
```

```
  using CancellationTokenRegistration _ = cancellationToken
      .Register(() => ping.SendAsyncCancel());
  return await task;
}
```

Теперь при запросе на отмену `CancellationToken` вызовет метод `SendAsyncCancel` за вас, отменяя метод `SendPingAsync`.

Пояснение

Метод `CancellationToken.Register` может использоваться для взаимодействия с любой альтернативной системой отмены. Но следует помнить, что если метод получает `CancellationToken`, запрос отмены должен отменять только эту одну операцию. Некоторые альтернативные системы отмены реализуют отмену закрытием некоторого ресурса, что может привести к отмене нескольких операций; эта разновидность системы отмены плохо соответствует `CancellationToken`. Если вы решите инкапсулировать такую разновидность отмены в `CancellationToken`, следует документировать ее необычную семантику отмены.

Помните о сроке существования регистрации обратных вызовов. Метод `Register` возвращает отменяемый объект, который должен быть освобожден, когда обратный вызов перестанет быть нужным. Предыдущий пример использует команду `using` для выполнения завершающих действий при завершении асинхронной операции. Если в коде отсутствует команда `using`, то при каждом вызове кода с тем же (долгосрочным) маркером `CancellationToken` он будет добавлять новый обратный вызов (который, в свою очередь, будет поддерживать существование объекта `Ping`). Чтобы избежать утечки памяти и ресурсов, отмените регистрацию обратного вызова, когда он перестанет быть нужным.

Дополнительная информация

В рецепте 10.2 рассматривается реакция на маркер отмены посредством периодического опроса (вместо обратных вызовов).

В рецепте 10.1 рассматривается общий механизм выдачи запросов на отмену.

ГЛАВА 11

ООП, хорошо сочетающееся с функциональным программированием

Современные программы требуют асинхронного программирования; в наши дни серверы должны масштабироваться лучше, чем когда-либо, а приложения для конечного пользователя должны реагировать на действия пользователя так быстро, как никогда. Разработчикам приходится изучать асинхронное программирование, и в реальной работе они часто обнаруживают, что этот стиль программирования часто вступает в конфликт с традиционным объектно-ориентированным программированием, к которому они привыкли.

Главная причина заключается в том, что асинхронное программирование является функциональным. Я не имею в виду, что «оно работает»; речь идет о функциональном стиле программирования, в отличие от процедурного стиля. Многие разработчики изучали основы функционального программирования в вузе и с тех пор практически не взаимодействовали с ним. Если от кода вида `(car(cdr '(3 5 7)))` вы поеживаетесь, то, возможно, вы принадлежите к этой категории. Не бойтесь! Современное асинхронное программирование не так уж сложно, стоит к нему привыкнуть.

Главное преимущество `async` заключается в том, что вы можете мыслить процедурными категориями, но программировать асинхронно. Это существенно упрощает написание и понимание асинхронных методов. Но во внутренней реализации асинхронный код остается функциональным по своей природе, и это создает некоторые проблемы, когда разработчики пытаются втиснуть `async`-методы в классические объектно-ориентированные архитектуры. В рецептах этой главы рассматриваются проблемные

области, в которых асинхронный код конфликтует с объектно-ориентированным программированием.

Эти проблемные области становятся особенно заметны при преобразовании существующей ООП-кодовой базы в кодовую базу, хорошо сочетающуюся с `async`.

11.1. Асинхронные интерфейсы и наследование

Задача

В интерфейсе или базовом классе метода имеется метод, который требуется сделать асинхронным.

Решение

Ключом к пониманию этой задачи и ее решения станет понимание того, что `async` относится к подробностям реализации. Ключевое слово `async` может применяться только к методам с реализациями; невозможно применить его к абстрактным методам или методам интерфейсов (если они не имеют реализаций по умолчанию). Тем не менее вы можете определить метод с такой же сигнатурой, как у `async`-метода, но без ключевого слова `async`.

Вспомните, что ожидание допускают *типы*, а не *методы*. Вы можете использовать `await` с объектом `Task`, возвращенным методом, независимо от того, был метод реализован с ключевым словом `async` или нет. Таким образом, интерфейс или абстрактный метод может просто вернуть `Task` (или `Task<T>`), а возвращаемое значение этого метода может допускать ожидание.

Следующий пример определяет интерфейс с асинхронным методом (без ключевого слова `async`), реализацию этого интерфейса (с `async`), и независимый метод, который потребляет метод этого интерфейса (посредством `await`):

```
interface IMyAsyncInterface
{
  Task<int> CountBytesAsync(HttpClient client, string url);
}
```

```
class MyAsyncClass : IMyAsyncInterface
{
  public async Task<int> CountBytesAsync(HttpClient client, string url)
  {
    var bytes = await client.GetByteArrayAsync(url);
    return bytes.Length;
  }
}

async Task UseMyInterfaceAsync(HttpClient client,
   IMyAsyncInterface service)
{
  var result = await service.CountBytesAsync(client,
     "http://www.example.com");
  Trace.WriteLine(result);
}
```

Этот паттерн работает и с абстрактными методами базовых классов.

Асинхронная сигнатура метода означает лишь то, что реализация *может* быть асинхронной. Фактическая реализация может быть синхронной, если нет реальной асинхронной работы, которую нужно было бы выполнять. Например, тестовая заглушка может реализовать тот же интерфейс (без `async`), используя нечто вроде `FromResult`:

```
class MyAsyncClassStub : IMyAsyncInterface
{
  public Task<int> CountBytesAsync(HttpClient client, string url)
  {
    return Task.FromResult(13);
  }
}
```

Пояснение

На момент написания книги `async` и `await` еще только набирали обороты. По мере того как асинхронные методы становятся все более распространенными, асинхронные методы интерфейсов и базовых классов встречаются все чаще. Работать с ними не так уж сложно, если помнить, что ожидание должно применяться к возвращаемому типу (а не к методу), а определение асинхронного метода может быть реализовано асинхронно или синхронно.

Дополнительная информация

В рецепте 2.2 рассматривается возвращение завершенной задачи, реализующей асинхронную сигнатуру метода с синхронным кодом.

11.2. Асинхронное конструирование: фабрики

Задача

Вы программируете тип, который требует выполнения некоторой асинхронной работы в конструкторе.

Решение

Конструкторы не могут объявляться с `async`; кроме того, они не могут содержать ключевое слово `await`. Конечно, использование `await` в конструкторе могло бы быть полезным, но это привело бы к существенному изменению языка C#.

Одна из возможностей — использовать конструктор в паре с инициализирующим `async`-методом, чтобы тип использовался следующим образом:

```
var instance = new MyAsyncClass();
await instance.InitializeAsync();
```

У такого подхода есть недостатки. Разработчик может забыть вызвать метод `InitializeAsync`, а экземпляр не может использоваться сразу же после выполнения конструктора.

Вот более качественное решение, которое основано на применении паттерна асинхронного фабричного метода:

```
class MyAsyncClass
{
  private MyAsyncClass()
  {
  }

  private async Task<MyAsyncClass> InitializeAsync()
  {
```

```
    await Task.Delay(TimeSpan.FromSeconds(1));
    return this;
  }

  public static Task<MyAsyncClass> CreateAsync()
  {
    var result = new MyAsyncClass();
    return result.InitializeAsync();
  }
}
```

Конструктор и метод `InitializeAsync` объявлены приватными, чтобы они не могли использоваться в другом коде; экземпляры могут создаваться только одним способом — статическим фабричным методом `CreateAsync`. Вызывающий код не может обратиться к экземпляру до того, как инициализация будет завершена.

В другом коде экземпляр может создаваться следующим образом:

```
MyAsyncClass instance = await MyAsyncClass.CreateAsync();
```

Пояснение

Главное преимущество этого паттерна заключается в том, что другой код никак не сможет получить неинициализированный экземпляр `MyAsyncClass`. Именно по этой причине я предпочитаю этот паттерн и использую его везде, где только возможно.

К сожалению, в некоторых сценариях этот способ не работает — в частности, когда в коде используется провайдер внедрения зависимостей. Ни одна заметная библиотека внедрения зависимостей или инверсии управления не работает с `async`-кодом. Если вы окажетесь в одной из таких ситуаций, существует пара альтернатив, которые также стоит рассмотреть.

Если создаваемый экземпляр в действительности является общим ресурсом, можно использовать асинхронную отложенную инициализацию, рассмотренную в рецепте 14.1. В противном случае можно воспользоваться паттерном асинхронной инициализации, описанным в рецепте 11.3.

Пример того, как поступать *не* следует:

```
class MyAsyncClass
{
```

```
  public MyAsyncClass()
  {
    InitializeAsync();
  }

  // ПЛОХОЙ КОД!!
  private async void InitializeAsync()
  {
    await Task.Delay(TimeSpan.FromSeconds(1));
  }
}
```

На первый взгляд решение может показаться разумным: вы получаете обычный конструктор, который запускает асинхронную операцию; при этом у него есть ряд недостатков, обусловленных использованием `async void`. Первая проблема заключается в том, что при завершении конструктора экземпляр все еще продолжает асинхронно инициализироваться, и не существует очевидного способа определить, когда завершится асинхронная инициализация. Вторая проблема связана с обработкой ошибок: любые исключения, выданные из `InitializeAsync`, не могут быть перехвачены секциями `catch`, окружающими конструирование объекта.

Дополнительная информация

В рецепте 11.3 рассматривается паттерн асинхронной инициализации — способ выполнения асинхронного конструирования, который работает с контейнерами внедрения зависимостей/инверсии управления.

В рецепте 14.1 рассматривается асинхронная отложенная инициализация; это решение приемлемо, если экземпляр на концептуальном уровне представляет общий ресурс или сервис.

11.3. Асинхронное конструирование: паттерн асинхронной инициализации

Задача

Вы программируете тип, требующий выполнения некоторой асинхронной работы в его конструкторе, но не можете воспользоваться паттерном

асинхронной фабрики (рецепт 11.2), так как экземпляр создается с применением отражения (например, библиотеки внедрения зависимостей/инверсии управления, связывания данных, `Activator.CreateInstance` и т. д.).

Решение

Если вы столкнулись с таким сценарием, вам приходится возвращать неинициализированный экземпляр, хотя ситуацию можно частично сгладить применением распространенного *паттерна асинхронной инициализации*. Каждый тип, требующий асинхронной инициализации, должен определять свойство следующего вида:

```
Task Initialization { get; }
```

Обычно я определяю его в интерфейсе-маркере для типов, требующих асинхронной инициализации:

```
/// <summary>
/// Помечает тип как требующий асинхронной инициализации
/// и предоставляет результат этой инициализации.
/// </summary>
public interface IAsyncInitialization
{
  /// <summary>
  /// Результат асинхронной инициализации этого экземпляра.
  /// </summary>
  Task Initialization { get; }
}
```

При реализации этого паттерна следует начать инициализацию (и задать значение свойства `Initialization`) в конструкторе. Доступ к результатам асинхронной инициализации (вместе с любыми исключениями) предоставляется через свойство `Initialization`. Пример реализации простого типа, использующего асинхронную инициализацию:

```
class MyFundamentalType : IMyFundamentalType, IAsyncInitialization
{
  public MyFundamentalType()
  {
    Initialization = InitializeAsync();
  }
```

```
  public Task Initialization { get; private set; }

  private async Task InitializeAsync()
  {
    // Провести асинхронную инициализацию этого экземпляра.
    await Task.Delay(TimeSpan.FromSeconds(1));
  }
}
```

Если вы используете библиотеку внедрения зависимостей/инверсии управления, то экземпляр этого типа может быть создан и инициализирован кодом следующего вида:

```
IMyFundamentalType instance =
   UltimateDIFactory.Create<IMyFundamentalType>();
var instanceAsyncInit = instance as IAsyncInitialization;
if (instanceAsyncInit != null)
  await instanceAsyncInit.Initialization;
```

Этот паттерн можно расширить так, чтобы он допускал композицию типов с асинхронной инициализацией. В следующем примере определяется другой тип, зависящий от `IMyFundamentalType`:

```
class MyComposedType : IMyComposedType, IAsyncInitialization
{
  private readonly IMyFundamentalType _fundamental;
  public MyComposedType(IMyFundamentalType fundamental)
  {
    _fundamental = fundamental;
    Initialization = InitializeAsync();
  }

  public Task Initialization { get; private set; }

  private async Task InitializeAsync()
  {
    // Асинхронно ожидать инициализации фундаментального экземпляра
    //  при необходимости.
    var fundamentalAsyncInit = _fundamental as IAsyncInitialization;
    if (fundamentalAsyncInit != null)
      await fundamentalAsyncInit.Initialization;
```

```
    // Выполнить собственную инициализацию (синхронно или асинхронно).
    ...
  }
}
```

Составной тип ожидает инициализации всех своих компонентов перед тем, как переходить к собственной инициализации. При этом следует руководствоваться таким правилом: каждый компонент должен быть инициализирован к концу `InitializeAsync`. Это гарантирует, что все зависимые типы будут инициализированы как часть составной инициализации. Любые исключения, возникающие в ходе составной инициализации, распространяются в инициализацию составного типа.

Пояснение

По возможности я рекомендую применять асинхронные фабрики (рецепт 11.2) или асинхронную отложенную инициализацию (рецепт 14.1) вместо этого решения. Эти решения предпочтительны, потому что в них исключается доступ к неинициализированному экземпляру. Если ваши экземпляры создаются библиотеками внедрения зависимостей/инверсии управления, связывания данных и т. д., вы будете вынуждены открыть доступ к неинициализированному экземпляру; в этом случае рекомендую использовать паттерн асинхронной инициализации из этого рецепта.

Как было сказано при описании асинхронных интерфейсов (рецепт 11.1), асинхронная сигнатура метода означает лишь то, что метод *может* быть асинхронным. Код `MyComposedType.InitializeAsync` является хорошим примером: если экземпляр `IMyFundamentalType` не реализует `IAsyncInitialization`, а `MyComposedType` не имеет собственной асинхронной инициализации, то его метод `InitializeAsync` завершается синхронно.

Код, который проверяет, реализует ли экземпляр `IAsyncInitialization` и инициализирует его, получается немного громоздким — и ситуация только усугубляется для составных типов, зависящих от большого количества компонентов. Вы можете легко создать вспомогательный метод для упрощения кода:

```
public static class AsyncInitialization
{
  public static Task WhenAllInitializedAsync(params object[] instances)
  {
```

```
    return Task.WhenAll(instances
        .OfType<IAsyncInitialization>()
        .Select(x => x.Initialization));
  }
}
```

Вызовите `InitializeAllAsync` и передайте любые экземпляры, которые требуется инициализировать; метод проигнорирует экземпляры, не реализующие `IAsyncInitialization`. Код инициализации для составного типа, зависящего от трех внедренных экземпляров, будет выглядеть примерно так:

```
private async Task InitializeAsync()
{
 // Асинхронно ожидать инициализации всех 3 экземпляров, если потребуется.
 await AsyncInitialization.WhenAllInitializedAsync(_fundamental,
     _anotherType, _yetAnother);

 // Выполнить собственную инициализацию (синхронно или асинхронно).
 ...
}
```

Дополнительная информация

В рецепте 11.2 рассматриваются асинхронные фабрики как механизм выполнения асинхронного конструирования без предоставления доступа к неинициализированным экземплярам.

В рецепте 14.1 рассматривается асинхронная отложенная инициализация, которая может использоваться для экземпляров, которые представляют общие ресурсы или сервисы.

В рецепте 11.1 рассматриваются асинхронные интерфейсы.

11.4. Асинхронные свойства

Задача

Имеется свойство, которое вам хотелось бы объявить как асинхронное. Свойство не задействовано в связывании данных.

Решение

Эта проблема часто встречается при преобразовании существующего кода для использования `async`; в таких ситуациях создается свойство, get-метод которого вызывает асинхронный метод. Вообще, такого понятия, как «асинхронное свойство», не существует. Ключевое слово `async` не может использоваться со свойством, и это хорошо. Get-методы свойств должны возвращать текущие значения; они не должны запускать фоновые операции:

```
// Чего хотелось бы (не компилируется).
public int Data
{
  async get
  {
    await Task.Delay(TimeSpan.FromSeconds(1));
    return 13;
  }
}
```

Когда вы обнаруживаете, что вашему коду нужно «асинхронное свойство», вашему коду в действительности *требуется* нечто иное. Решение зависит от того, должно ли значение свойства вычисляться один раз или несколько; у вас есть выбор между следующими семантиками:

- Значение асинхронно вычисляется каждый раз при чтении.
- Значение асинхронно вычисляется один раз и кэшируется для будущих обращений.

Если ваше «асинхронное свойство» должно запускать новое (асинхронное) вычисление каждый раз, когда оно читается, то по сути это не *свойство*, а замаскированный *метод*. Если вы сталкиваетесь с этой ситуацией при преобразовании синхронного кода в асинхронный, то пора признать, что исходная архитектура в действительности была неверной; свойство должно было с самого начала быть реализовано в виде метода:

```
// В виде асинхронного метода.
public async Task<int> GetDataAsync()
{
  await Task.Delay(TimeSpan.FromSeconds(1));
  return 13;
}
```

Вы *можете* получить `Task<int>` непосредственно от свойства, как показано в следующем коде:

```
// Это "асинхронное свойство" является асинхронным методом.
public Task<int> Data
{
  get { return GetDataAsync(); }
}

private async Task<int> GetDataAsync()
{
  await Task.Delay(TimeSpan.FromSeconds(1));
  return 13;
}
```

И все же я не рекомендую применять этот подход. Если при каждом обращении к свойству будет запускаться новая асинхронная операция, это «свойство» в действительности должно быть оформлено в виде метода. Тот факт, что код оформлен в виде асинхронного метода, более ясно показывает, что каждый раз запускается новая асинхронная операция, поэтому API не вводит пользователя в заблуждение. В рецептах 11.3 и 11.6 используются свойства, возвращающие задачи, но эти свойства относятся к экземпляру в целом; они не запускают новую асинхронную операцию при каждом чтении.

Иногда значение свойства должно вычисляться при каждом чтении. В других случаях свойство должно инициировать только одно (асинхронное) вычисление и кэшировать полученное значение для использования в будущем. В этом случае можно использовать асинхронную отложенную инициализацию. Это решение подробно рассматривается в рецепте 14.1, а пока приведем пример того, как может выглядеть этот код:

```
// Как кэшированное значение
public AsyncLazy<int> Data
{
  get { return _data; }
}

private readonly AsyncLazy<int> _data =
    new AsyncLazy<int>(async () =>
    {
      await Task.Delay(TimeSpan.FromSeconds(1));
```

```
    return 13;
  });
```

Код будет выполнять асинхронное вычисление только один раз, а затем полученное значение будет возвращаться всем остальным вызывающим сторонам. Код вызова выглядит примерно так:

```
int value = await instance.Data;
```

В данном случае синтаксис свойства уместен, потому что вычисление происходит только один раз.

Пояснение

Один из самых важных вопросов, которые следует задать себе: должно ли чтение свойства запускать новую асинхронную операцию. Если ответ будет положительным, используйте асинхронный метод вместо свойства. Если свойство должно работать как кэш с отложенным вычислением, используйте асинхронную инициализацию (см. рецепт 14.1). В этом рецепте я не рассматриваю свойства, используемые в связывании данных; они будут рассматриваться в рецепте 14.3.

Если вы преобразуете синхронное свойство в «асинхронное свойство», следующий пример показывает, как это делать *не следует*:

```
private async Task<int> GetDataAsync()
{
  await Task.Delay(TimeSpan.FromSeconds(1));
  return 13;
}

public int Data
{
  // ПЛОХОЙ КОД!!
  get { return GetDataAsync().Result; }
}
```

Раз уж речь зашла о свойствах в `async`-коде, стоит задуматься над тем, как состояние соотносится с асинхронным кодом. Это особенно актуально при преобразовании синхронной кодовой базы в асинхронную. Возьмем любое состояние, доступ к которому осуществляется через API (напри-

мер, через свойства): для каждой составляющей состояния спросите себя: что считать текущим состоянием объекта с незавершенной асинхронной операцией? Правильного ответа не существует, но важно продумать то, какие семантики вам нужны и как их документировать.

Для примера возьмем объект `Stream.Position`, представляющий текущее смещение указателя в потоке. С синхронным API при вызове `Stream.Read` или `Stream.Write` чтение/запись завершается, а `Stream.Position` обновляется новой позицией перед возвращением управления методом `Read` или `Write`. Для синхронного кода семантика ясна.

Теперь возьмем `Stream.ReadAsync` и `Stream.WriteAsync`: когда должно обновляться значение `Stream.Position`? При завершении операции чтения/записи или до того, как это фактически произойдет? Если оно обновляется перед завершением операции, то будет ли оно обновлено синхронно к моменту возвращения управления `ReadAsync`/`WriteAsync` или же вскоре после этого?

Это отличный пример того, как свойство, предоставляющее доступ к состоянию, обладает абсолютно ясной семантиков для синхронного кода, но не имеет очевидно правильной семантики для асинхронного кода. Конечно, ничего ужасного в этом нет — просто нужно продумать весь API при реализации поддержки `async` для ваших типов и документировать выбранную вами семантику.

Дополнительная информация

В рецепте 14.1 подробно рассматривается асинхронная отложенная инициализация.

В рецепте 14.3 рассматриваются «асинхронные свойства», которые должны поддерживать связывание данных.

11.5. async-события

Задача

Событие должно использоваться с обработчиками, которые могут быть асинхронными; требуется проверить, завершились ли обработчики со-

бытий. Следует учитывать, что при работе с событиями такая ситуация встречается довольно редко; обычно при выдаче события вас не интересует, когда завершатся обработчики.

Решение

Определить, когда обработчики `async void` вернули управление, невозможно, поэтому вам потребуется механизм обнаружения факта завершения асинхронных обработчиков. На платформе Universal Windows появилась концепция так называемых *объектов отложенного выполнения* (deferrals), которые могут использоваться для отслеживания асинхронных обработчиков. Асинхронный обработчик создает объект отложенного выполнения перед первым ключевым словом `await` и позднее уведомляет объект отложенного выполнения при завершении. Синхронным обработчикам использовать объекты отложенного выполнения не нужно.

Библиотека `Nito.AsyncEx` включает тип `DeferralManager`, который используется компонентом, выдающим событие. Объект `DeferralManager` затем разрешает обработчикам событий создавать объекты отложенного выполнения и отслеживает завершение всех таких объектов.

Для каждого из ваших событий, для которых необходимо дождаться завершения обработчиков, начните с расширения типа аргументов события:

```
public class MyEventArgs : EventArgs, IDeferralSource
{
  private readonly DeferralManager _deferrals = new DeferralManager();

  ... // Ваши конструкторы и свойства

  public IDisposable GetDeferral()
  {
    return _deferrals.DeferralSource.GetDeferral();
  }

  internal Task WaitForDeferralsAsync()
  {
    return _deferrals.WaitForDeferralsAsync();
  }
}
```

С асинхронными обработчиками событий лучше сделать тип аргументов события потокобезопасным. Проще всего для этого объявить его неизменяемым (т. е. потребовать, чтобы все его свойства были доступны только для чтения).

Затем каждый раз при выдаче события вы можете (асинхронно) ожидать завершения всех асинхронных обработчиков событий. Следующий пример вернет завершенную задачу при отсутствии обработчиков; в противном случае он создаст новый экземпляр типа аргументов события, передаст его обработчикам и будет ожидать завершения всех асинхронных обработчиков:

```
public event EventHandler<MyEventArgs> MyEvent;

private async Task RaiseMyEventAsync()
{
  EventHandler<MyEventArgs> handler = MyEvent;
  if (handler == null)
    return;
  var args = new MyEventArgs(...);
  handler(this, args);
  await args.WaitForDeferralsAsync();
}
```

После этого асинхронные обработчики событий смогут использовать объект отложенного выполнения в блоке `using`. Объект отложенного выполнения уведомит `DeferralManager` о своем освобождении:

```
async void AsyncHandler(object sender, MyEventArgs args)
{
  using IDisposable deferral = args.GetDeferral();
  await Task.Delay(TimeSpan.FromSeconds(2));
}
```

Происходящее несколько отличается от того, как работают объекты отложенного выполнения в Universal Windows. В Universal Windows API каждое событие, которому понадобятся объекты отложенного выполнения, определяет собственный тип объекта отложенного выполнения, и этот тип содержит явно определенный метод `Complete` (вместо того чтобы реализовать `IDisposable`).

Пояснение

На логическом уровне в .NET используются две разновидности событий с очень сильно различающейся семантикой. Я называю их *событиями уведомлений* и *командными событиями*; это не официальная терминология, а просто некие термины, которые я выбрал для ясности. События уведомлений используются для оповещения других компонентов о некоторой ситуации. Уведомление является чисто односторонним; отправителя события не интересует, есть ли у этого события получатели. С уведомлениями отправитель и получатель могут быть полностью изолированы друг от друга. Большинство событий относится к категории событий уведомления; одним из примеров такого рода является щелчок на кнопке.

С другой стороны, командное событие инициируется для реализации некоторой функциональности по поручению компонента-отправителя. Командные события не являются «событиями» в подлинном смысле этого термина, хотя они часто реализуются в виде событий .NET. Отправитель команды должен дождаться, пока она будет обработана получателем, прежде чем двигаться дальше. Если события используются для реализации паттерна «Посетитель», то это командные события. События жизненного цикла тоже являются командными событиями; к этой категории также относятся события жизненного цикла страниц ASP.NET и многие события UI-фреймворков (например, событие Xamarin `Application.PageAppearing`). Любое событие UI-фреймворка, который в действительности является реализацией, также является командным событием (например, `BackgroundWorker.DoWork`).

События уведомлений не требуют специального кода, чтобы сделать возможным использование асинхронных обработчиков; обработчики событий могут быть объявлены `async void` и при этом будут нормально работать. Когда обработчик события выдает событие, асинхронные обработчики событий не завершаются немедленно, но это неважно, потому что они являются событиями уведомлений. Таким образом, если ваше событие является событием уведомления, то для поддержки асинхронных обработчиков, по сути, не придется делать ничего.

Командные события — совсем другое дело. При использования командного события необходимо каким-то образом определить, когда обработчики были завершены. Приведенное решение с объектами отложенного выполнения должно использоваться только для командных событий.

Тип `DeferralManager` находится в пакете `Nito.AsyncEx`.

Дополнительная информация

В главе 2 рассматриваются основы асинхронного программирования.

11.6. Асинхронное освобождение

Задача

Имеется тип с асинхронными операциями, который должен обеспечить освобождение своих ресурсов.

Решение

Есть два распространенных варианта действий в отношении существующих операций при освобождении экземпляра: освобождение можно либо рассматривать как запрос на отмену, применяемый ко всем существующим операциям, либо реализовать настоящее асинхронное освобождение.

Интерпретация освобождения как отмены уже встречалась в Windows; такие типы, как файловые потоки и сокеты, отменяют все существующие операции чтения и записи при закрытии. Определив собственный тип `CancellationTokenSource` и передавая этот маркер внутренним операциям, можно сделать нечто очень похожее в .NET. В следующем коде `Dispose` отменит операции, но не ожидает завершения этих операций:

```
class MyClass : IDisposable
{
  private readonly CancellationTokenSource _disposeCts =
      new CancellationTokenSource();

  public async Task<int> CalculateValueAsync()
  {
    await Task.Delay(TimeSpan.FromSeconds(2), _disposeCts.Token);
```

```
    return 13;
  }

  public void Dispose()
  {
    _disposeCts.Cancel();
  }
}
```

Этот пример демонстрирует основной паттерн, относящийся к `Dispose`. В реальном приложении следовало бы включить проверку того, что объект еще не был освобожден, а также предоставить пользователю возможность передать собственный маркер `CancellationToken` (с использованием приема из рецепта 10.8):

```
public async Task<int> CalculateValueAsync(CancellationToken
   cancellationToken)
{
  using CancellationTokenSource combinedCts = CancellationTokenSource
      .CreateLinkedTokenSource(cancellationToken, _disposeCts.Token);
  await Task.Delay(TimeSpan.FromSeconds(2), combinedCts.Token);
  return 13;
}
```

При вызове `Dispose` будут отменены все существующие операции в вызывающем коде:

```
async Task UseMyClassAsync()
{
  Task<int> task;
  using (var resource = new MyClass())
  {
    task = resource.CalculateValueAsync(default);
  }

  // Выдает OperationCanceledException.
  var result = await task;
}
```

Для некоторых типов реализация `Dispose` как запроса на отмену работает вполне нормально (например, `HttpClient` обладает такой семантикой). Однако другим типам необходимо знать, когда будут завершены все

операции. Для таких типов необходима некоторая разновидность асинхронного освобождения.

Асинхронное освобождение впервые появилось в C# 8.0 и .NET Core 3.0. В BCL появился новый интерфейс `IAsyncDisposable`, который является асинхронным аналогом `IDisposable`. В языке одновременно была введена команда `await using` — асинхронный аналог `using`. Таким образом, типы, которые собирались выполнить асинхронную работу при освобождении, теперь получили такую возможность:

```
class MyClass : IAsyncDisposable
{
  public async ValueTask DisposeAsync()
  {
    await Task.Delay(TimeSpan.FromSeconds(2));
  }
}
```

Возвращаемым типом `DisposeAsync` является `ValueTask`, а не `Task`, но стандартные ключевые слова `async` и `await` работают с `ValueTask` ничуть не хуже, чем с `Task`.

Типы, реализующие `IAsyncDisposable`, обычно потребляются с `await`:

```
await using (var myClass = new MyClass())
{
  ...
} // Здесь вызывается DisposeAsync (с ожиданием)
```

Если нужно обойти контекст с использованием `ConfigureAwait(false)`, это возможно, но решение получается более громоздким, потому что переменная должна быть объявлена за пределами команды `await using`:

```
var myClass = new MyClass();
await using (myClass.ConfigureAwait(false))
{
  ...
} // Здесь вызывается DisposeAsync (с ожиданием)
   с ConfigureAwait(false).
```

Пояснение

Асинхронное освобождение определенно проще, чем реализация `Dispose` в виде запроса на отмену, а более сложный подход должен использоваться

только в том случае, если это действительно необходимо. Собственно, в большинстве случаев можно обойтись вообще без освобождения — и конечно, это самое простое решение, потому что вам вообще ничего не придется делать.

В этом рецепте представлены два паттерна для реализации освобождения; также при желании вы можете использовать их одновременно. Одновременное использование обоих паттернов наделит ваш тип семантикой четкого завершения, если в клиентском коде используется `await using`, и семантикой отмены, если клиентский код использует `Dispose`. Я бы не рекомендовал так поступать, но знайте, что такой вариант существует.

Дополнительная информация

В рецепте 10.8 рассматриваются связанные маркеры отмены.

В рецепте 11.1 рассматриваются асинхронные интерфейсы.

В рецепте 2.10 рассматривается реализация методов, возвращающих `ValueTask`.

В рецепте 2.7 рассматривается обход контекста с использованием `ConfigureAwait(false)`.

ГЛАВА 12

Синхронизация

Если в вашем приложении используется конкурентность (как практически во всех приложениях .NET), следует остерегаться ситуаций, в которых один блок кода должен обновлять данные, пока другой код обращается к тем же данным. Столкнувшись с подобной ситуацией, необходимо синхронизировать доступ к данным. В рецептах этой главы рассматриваются наиболее распространенные типы, используемые для синхронизации доступа. Впрочем, если вы будете правильно пользоваться другими рецептами из книги, то увидите, что большая часть синхронизации в типичных ситуациях уже реализуется за вас соответствующими библиотеками. Но прежде чем углубляться в рецепты, посвященные синхронизации, будут описаны некоторые типичные ситуации, в которых может появиться такая необходимость.

Описания синхронизации в этом разделе несколько упрощены, но все заключения правильны.

Существует две основные разновидности синхронизации: *передачи данных* и *защиты данных*. Синхронизация при передаче данных используется в тех случаях, когда один блок кода должен оповестить другой блок кода о некотором условии (например, о поступлении нового сообщения). Синхронизация при передаче данных будет более подробно рассмотрена в рецептах этой главы; оставшаяся часть введения будет посвящена защите данных.

Синхронизация должна использоваться для защиты общих данных при выполнении *всех трех* условий из следующего списка:

- Несколько частей кода выполняются одновременно.
- Эти части кода обращаются (читают или записывают) одни и те же данные.
- По крайней мере одна часть кода обновляет (или записывает) данные.

Причина для первого условия должна быть очевидна; если весь ваш код выполняется от начала к концу и ничего не происходит одновременно, то вообще не нужно беспокоиться о синхронизации. Некоторые простые консольные приложения могут работать по этому принципу, но в большинстве приложений .NET используется та или иная форма конкурентности. Второе условие означает, что если каждый блок кода работает с собственными локальными данными, к которым не могут обращаться другие, то необходимость в синхронизации отсутствует; другие части кода никогда не обращаются к локальным данным. Также необходимость в синхронизации отсутствует и в том случае, если общие данные присутствуют, но никогда не изменяются: например, если данные определяются с использованием неизменяемых типов. Под третье условие подпадают такие сценарии, как данные конфигурации и т. п.; они задаются в начале работы приложения, а затем никогда не изменяются. Если общие данные используются только для чтения, то они не нуждаются в синхронизации; она необходима только для данных, которые одновременно находятся *в общем доступе* и *изменяются*.

Цель защиты данных заключается в том, чтобы каждая часть кода имела целостное представление данных. Если одна часть кода обновляет данные, вы можете воспользоваться синхронизацией, чтобы обновления воспринимались как атомарные для остальных компонентов системы.

Вероятно, вы не сразу научитесь понимать, когда нужна синхронизация, поэтому перед тем как переходить к рецептам этой главы, рассмотрим несколько примеров. Начнем со следующего кода:

```
async Task MyMethodAsync()
{
  int value = 10;
  await Task.Delay(TimeSpan.FromSeconds(1));
  value = value + 1;
  await Task.Delay(TimeSpan.FromSeconds(1));
  value = value - 1;
  await Task.Delay(TimeSpan.FromSeconds(1));
  Trace.WriteLine(value);
}
```

Если метод `MyMethodAsync` вызывается в потоке из пула потоков (например, из `Task.Run`), то строки кода, обращающиеся к значению, могут выполняться в разных потоках. Но понадобится ли синхронизация в этом

случае? Нет, потому что они не могут выполняться одновременно. Это асинхронный метод, но он также является последовательным (т. е. сначала выполняется одна часть, потом другая и т. д.).

Немного усложним пример. На этот раз будет выполняться конкурентный асинхронный код:

```
private int value;

async Task ModifyValueAsync()
{
  await Task.Delay(TimeSpan.FromSeconds(1));
  value = value + 1;
}

// ВНИМАНИЕ: может требовать синхронизации; см. ниже.
async Task<int> ModifyValueConcurrentlyAsync()
{
  // Start three concurrent modifications.
  Task task1 = ModifyValueAsync();
  Task task2 = ModifyValueAsync();
  Task task3 = ModifyValueAsync();
  await Task.WhenAll(task1, task2, task3);
  return value;
}
```

Этот код запускает три изменения, выполняемых конкурентно. Понадобится ли синхронизация? Зависит от обстоятельств. Если вы знаете, что метод вызывается из GUI-контекста или контекста ASP.NET (или любого контекста, который позволяет выполняться только одному фрагменту кода в любой момент времени), синхронизация будет излишней, потому что при выполнении кода изменения `data` он будет выполняться в разное время с двумя другими изменениями. Например, если приведенный код выполняется в GUI-контексте, то существует только один UI-поток, который будет выполнять каждое изменение `data`, поэтому он *должен* выполнять их по одному. Итак, если вы знаете, что контекст является последовательным (one-at-a-time), то синхронизация *не нужна*. Но если тот же метод вызывается в потоке из пула потоков (например, из `Task.Run`), то синхронизация *будет необходима*. В данном случае три изменения данных могут выполняться в разных потоках из пула и обновлять `data.Value` одновременно, поэтому доступ к `data.Value` требуется синхронизировать.

А теперь еще одна вариация:

```
private int value;
async Task ModifyValueAsync()
{
  int originalValue = value;
  await Task.Delay(TimeSpan.FromSeconds(1));
  value = originalValue + 1;
}
```

Посмотрим, что произойдет, если `ModifyValueAsync` вызывается несколько раз конкурентно. Даже при вызове из последовательного контекста поле данных используется совместно всеми вызовами `ModifyValueAsync`, а значение может измениться в любое время, когда в методе выполняется `await`. Иногда синхронизация применяется даже в последовательных контекстах для предотвращения общего доступа такого рода. Иначе говоря, если вы хотите добиться того, чтобы каждый вызов `ModifyValueAsync` ожидал завершения всех предыдущих вызовов, следует добавить синхронизацию. Это справедливо даже в том случае, если контекст гарантирует, что для всего кода используется только один поток (т. е. UI-поток). Синхронизация в этом сценарии является разновидностью *регулировки* для асинхронных методов (см. раздел 12.2).

Рассмотрим еще один пример с `async`. `Task.Run` можно использовать для того, что я называю простым параллелизмом — простейшей разновидностью параллельной обработки, которая не обладает такой эффективностью и возможностями настройки, как истинный параллелизм Parallel/PLINQ. Следующий код обновляет общее значение с использованием простого параллелизма:

```
// ПЛОХОЙ КОД!!
async Task<int> SimpleParallelismAsync()
{
  int value = 0;
  Task task1 = Task.Run(() => { value = value + 1; });
  Task task2 = Task.Run(() => { value = value + 1; });
  Task task3 = Task.Run(() => { value = value + 1; });
  await Task.WhenAll(task1, task2, task3);
  return value;
}
```

В этом коде три отдельные задачи выполняются в пуле потоков (через `Task.Run`), причем все они изменяют одно значение `value`. Следовательно,

условия выполняются, и синхронизация определенно необходима. Обратите внимание: синхронизация нужна даже несмотря на то, что переменная `value` является локальной; она все равно *совместно используется* разными потоками, хотя и является локальной для одного метода.

Переходя к настоящему параллельному коду, рассмотрим пример, в котором используется тип `Parallel`:

```
void IndependentParallelism(IEnumerable<int> values)
{
  Parallel.ForEach(values, item => Trace.WriteLine(item));
}
```

Так как в коде используется `Parallel`, необходимо предполагать, что тело параллельного цикла (`item =>Trace.WriteLine(item)`) может выполняться в нескольких потоках. Однако тело цикла читает только собственные данные; совместного использования данных между потоками здесь не будет. Класс `Parallel` разделяет данные между потоками, чтобы им не приходилось совместно использовать свои данные. Каждый поток, выполняющий тело цикла, не зависит от всех остальных потоков, выполняющих то же тело цикла. Таким образом, синхронизация в приведенном коде не нужна.

Рассмотрим пример агрегирования, похожий на описанный в рецепте 4.2:

```
// ПЛОХОЙ КОД!!
int ParallelSum(IEnumerable<int> values)
{
  int result = 0;
  Parallel.ForEach(source: values,
      localInit: () => 0,
      body: (item, state, localValue) => localValue + item,
      localFinally: localValue => { result += localValue; });
  return result;
}
```

В этом примере код снова использует несколько потоков; на этот раз каждый поток в начале своей работы инициализирует свое локальное значение 0(`() => 0`), и для каждого входного значения, обработанного потоком, входное значение прибавляется к локальному (`((item, state, localValue) =>localValue + item`). Наконец, все локальные значения прибавляются к возвращаемому значению (`localValue => { result +=`

`localValue; }`). Первые два шага не создают проблем, потому что потоки не имеют общих данных; локальное и входное значение каждого потока существует независимо от локальных и входных значений всех остальных потоков, но на последнем шаге возникают проблемы; когда локальное значение каждого потока прибавляется к возвращаемому значению, возникает ситуация, в которой сразу несколько потоков обращаются к общей переменной (`result`) и обновляют ее. Таким образом, на последнем шаге необходимо применить синхронизацию (см. рецепт 12.1).

PLINQ, TPL Dataflow и реактивные библиотеки очень похожи на примеры `Parallel`: пока ваш код имеет дело только со своим собственным вводом, ему не приходится беспокоиться о синхронизации. По опыту могу сказать, что при правильном использовании этих библиотек добавлять синхронизацию в свой код мне почти никогда не приходится.

И наконец, поговорим о коллекциях. Вспомните три условия, требующие синхронизации: *наличие нескольких частей кода*, *общие данные* и *обновления данных*.

Неизменяемые типы потокобезопасны по своей природе, потому что *не могут* изменяться; невозможно обновить неизменяемую коллекцию, поэтому синхронизация не нужна. Например, следующий код не требует синхронизации, так как когда каждый отдельный поток из пула потоков заносит значение в стек, он создает новый неизменяемый стек с этим значением, а исходный стек остается без изменений.

```
async Task<bool> PlayWithStackAsync()
{
  ImmutableStack<int> stack = ImmutableStack<int>.Empty;
  Task task1 = Task.Run(() => Trace.WriteLine(stack.Push(3).Peek()));
  Task task2 = Task.Run(() => Trace.WriteLine(stack.Push(5).Peek()));
  Task task3 = Task.Run(() => Trace.WriteLine(stack.Push(7).Peek()));
  await Task.WhenAll(task1, task2, task3);

  return stack.IsEmpty; // Всегда возвращает true.
}
```

Когда в вашем коде используются неизменяемые коллекции, обычно в нем также присутствует общая «корневая» переменная, которая сама по себе неизменяемой не является. В этом случае *приходится* применять синхро-

низацию. В следующем примере каждый поток заносит значение в стек (создавая новый неизменяемый стек), после чего обновляет общую корневую переменную; для обновления переменной `stack` синхронизация *необходима*:

```
// ПЛОХОЙ КОД!!
async Task<bool> PlayWithStackAsync()
{
  ImmutableStack<int> stack = ImmutableStack<int>.Empty;
  Task task1 = Task.Run(() => { stack = stack.Push(3); });
  Task task2 = Task.Run(() => { stack = stack.Push(5); });
  Task task3 = Task.Run(() => { stack = stack.Push(7); });
  await Task.WhenAll(task1, task2, task3);
  return stack.IsEmpty;
}
```

С потокобезопасными коллекциями (например, `ConcurrentDictionary`) дело обстоит иначе. В отличие от неизменяемых коллекций, потокобезопасные коллекции могут обновляться. Но вся необходимая синхронизация уже встроена в них, так что вам не придется беспокоиться о синхронизации изменений в коллекции. Если бы в следующем коде обновлялась коллекция `Dictionary` вместо `ConcurrentDictionary`, то синхронизация была бы необходима; но поскольку обновляется `ConcurrentDictionary`, она становится излишней:

```
async Task<int> ThreadsafeCollectionsAsync()
{
  var dictionary = new ConcurrentDictionary<int, int>();
  Task task1 = Task.Run(() => { dictionary.TryAdd(2, 3); });
  Task task2 = Task.Run(() => { dictionary.TryAdd(3, 5); });
  Task task3 = Task.Run(() => { dictionary.TryAdd(5, 7); });
  await Task.WhenAll(task1, task2, task3);
  return dictionary.Count; // Всегда возвращает 3.
}
```

12.1. Блокировки и команда lock

Задача

Имеются общие данные. Требуется обеспечить безопасное чтение и запись этих данных из нескольких потоков.

Решение

Лучшее решение в такой ситуации — использование команды блокировки `lock`. Если поток входит в блок `lock`, то все остальные потоки не смогут войти в этот блок, пока блокировка не будет снята:

```
class MyClass
{
  // Блокировка защищает поле _value.
  private readonly object _mutex = new object();

  private int _value;

  public void Increment()
  {
    lock (_mutex)
    {
      _value = _value + 1;
    }
  }
}
```

Пояснение

В фреймворке .NET существует несколько механизмов блокировки: `Monitor`, `Spin`, `Lock` и `ReaderWriterLockSlim`. В большинстве приложений эти типы блокировок практически никогда не должны использоваться напрямую. В частности, для разработчиков проще переключиться на `ReaderWriterLockSlim`, когда такая сложность не является необходимой. Базовая команда `lock` нормально справляется с 99 % случаев.

При использовании блокировок следует руководствоваться четырьмя важными рекомендациями:

- Ограничьте видимость блокировки.
- Документируйте, что именно защищает блокировка.
- Сократите до минимума объем кода, защищенного блокировкой.
- Никогда не выполняйте произвольный код при удержании блокировки.

Во-первых, *стремитесь к ограничению видимости блокировки*. Объект, используемый в команде `lock`, должен быть приватным полем, которое никогда не должно быть доступным для любых методов за пределами класса. Обычно есть не более одного поля блокировки на тип; если у вас их несколько, рассмотрите возможность рефакторинга этого типа на несколько типов. Блокировка *может* устанавливаться по любому ссылочному типу, но я предпочитаю создавать отдельное поле специально для команды `lock`, как в последнем примере. Если вы устанавливаете блокировку по другому экземпляру, убедитесь в том, что он является приватным для вашего класса; он не должен передаваться в конструкторе или возвращаться из get-метода свойства. Никогда не используйте `lock(this)` или `lock` с любым экземпляром `Type` или `string`; это может привести к взаимоблокировкам, доступным из другого кода.

Во-вторых, *документируйте, что именно защищает блокировка*. Об этом шаге легко забыть во время первоначального написания кода, но он становится более важным по мере роста сложности кода.

В-третьих, *сократите до минимума объем кода, защищенного блокировкой*. Один из аспектов, на которые следует обращать внимание, — блокирующие вызовы при удержании блокировок. В идеале их быть вообще не должно.

Наконец, никогда не вызывайте произвольный код при удержании блокировки. Произвольный код может включать выдачу событий, вызов виртуальных методов или вызов делегатов. Если вам потребуется выполнить произвольный код, сделайте это после снятия блокировки.

Дополнительная информация

В рецепте 12.2 рассматриваются async-совместимые блокировки. Команда `lock` несовместима с `await`.

В рецепте 12.3 рассматривается передача сигналов между потоками. Команда `lock` предназначена для защиты общих данных, а не для отправки сигналов между потоками.

В рецептах 12.5 рассматривается регулировка, которая представляет собой обобщенную форму блокировки. Блокировка может рассматриваться как регулировка до уровня 1.

12.2. Блокировки с async

Задача

Имеются общие данные. Требуется обеспечить безопасное чтение и запись этих данных из разных программных блоков, которые могут использовать `await`.

Решение

Тип `SemaphoreSlim` из фреймворка .NET был обновлен в .NET 4.5 для обеспечения совместимости с `async`. Пример использования:

```
class MyClass
{
  // Блокировка защищает поле _value.
  private readonly SemaphoreSlim _mutex = new SemaphoreSlim(1);

  private int _value;

  public async Task DelayAndIncrementAsync()
  {
    await _mutex.WaitAsync();
    try
    {
      int oldValue = _value;
      await Task.Delay(TimeSpan.FromSeconds(oldValue));
      _value = oldValue + 1;
    }
    finally
    {
      _mutex.Release();
    }
  }
}
```

Также можно воспользоваться типом `AsyncLock` из библиотеки `Nito.AsyncEx`, который обладает чуть более элегантным API:

```
class MyClass
{
  // Блокировка защищает поле _value.
  private readonly AsyncLock _mutex = new AsyncLock();

  private int _value;

  public async Task DelayAndIncrementAsync()
  {
    using (await _mutex.LockAsync())
    {
      int oldValue = _value;
      await Task.Delay(TimeSpan.FromSeconds(oldValue));
      _value = oldValue + 1;
    }
  }
}
```

Пояснение

В этой ситуации действуют рекомендации из рецепта 12.1:

- Ограничьте видимость блокировки.
- Документируйте, что именно защищает блокировка.
- Сократите до минимума объем кода, защищенного блокировкой.
- Никогда не выполняйте произвольный код при удержании блокировки.

Экземпляры блокировок должны быть приватными; они не должны быть доступными за пределами класса. Обязательно четко документируйте (и тщательно продумывайте), что именно защищает экземпляр блокировки. Сведите к минимуму объем кода, выполняемого при удержании блокировки. В частности, не вызывайте произвольный код, включая выдачу событий, вызов виртуальных методов и вызов делегатов.

Тип `AsyncLock` находится в пакете `Nito.AsyncEx`.

Дополнительная информация

В рецепте 12.4 рассматриваются `async`-совместимые сигналы. Блокировки предназначены для защиты общих данных, а не для передачи сигналов.

В рецепте 12.5 рассматривается регулировка, которая представляет собой обобщенную форму. Блокировка может рассматриваться как регулировка до уровня 1.

12.3. Блокирующие сигналы

Задача

Требуется отправить уведомление от одного потока другому.

Решение

Самый распространенный и универсальный межпотоковый сигнал — событие с ручным сбросом `ManualResetEventSlim`. Событие с ручным сбросом может находиться в одном из двух состояний: установленном или сброшенном. Любой поток может перевести поток в установленное состояние или провести его сброс. Поток также может ожидать перехода события в установленное состояние.

Следующие два метода вызываются разными потоками; один поток ожидает сигнала от другого:

```
class MyClass
{
  private readonly ManualResetEventSlim _initialized =
      new ManualResetEventSlim();

  private int _value;

  public int WaitForInitialization()
  {
```

```
    _initialized.Wait();
    return _value;
  }

  public void InitializeFromAnotherThread()
  {
    _value = 13;
    _initialized.Set();
  }
}
```

Пояснение

`ManualResetEventSlim` — отличный универсальный сигнал, передаваемый одним потоком другому, однако использовать его следует только тогда, когда это действительно уместно. Если «сигнал» представляет собой сообщение, отправляющее некоторые данные между потоками, рассмотрите возможность использования очереди «производитель/потребитель». С другой стороны, если сигналы используются только для координации доступа к общим данным, лучше использовать `lock`.

В фреймворке .NET существуют и другие разновидности сигналов синхронизации потоков, которые используются реже. Если `ManualResetEventSlim` не подходит для ваших потребностей, подумайте об использовании `AutoResetEvent`, `CountdownEvent` или `Barrier`.

`ManualResetEventSlim` является синхронным сигналом, поэтому `WaitForInitialization` блокирует вызывающий поток до отправки сигнала. Если вы хотите ожидать сигнала без приостановки потока, используйте асинхронный сигнал так, как описано в рецепте 12.4.

Дополнительная информация

В рецепте 9.6 рассматриваются блокирующие очереди «производитель/потребитель».

В рецепте 12.1 рассматривается команда `lock`.

В рецепте 12.4 рассматриваются `async`-совместимые сигналы.

12.4. Асинхронные сигналы

Задача

Требуется отправить уведомление от одного потока другому, при этом получатель оповещения должен ожидать его асинхронно.

Решение

Используйте `TaskCompletionSource<T>` для того, чтобы отправить уведомление асинхронно, если уведомление должно быть отправлено только один раз. Код-отправитель вызывает `TrySetResult`, а код-получатель ожидает его свойство `Task`:

```
class MyClass
{
  private readonly TaskCompletionSource<object> _initialized =
      new TaskCompletionSource<object>();

  private int _value1;
  private int _value2;

  public async Task<int> WaitForInitializationAsync()
  {
    await _initialized.Task;
    return _value1 + _value2;
  }

  public void Initialize()
  {
    _value1 = 13;
    _value2 = 17;
    _initialized.TrySetResult(null);
  }
}
```

Тип `TaskCompletionSource<T>` может использоваться для асинхронного ожидания любой ситуации — в данном случае уведомления от другой части кода. Этот способ хорошо работает, если сигнал отправляется

только один раз, но совершенно не работает, если сигнал нужно не только включать, но и отключать.

Библиотека `Nito.AsyncEx` содержит тип `AsyncManualResetEvent` — приблизительный аналог `ManualResetEvent` для асинхронного кода. Следующий пример является искусственным, но показывает, как правильно использовать тип `AsyncManualResetEvent`:

```
class MyClass
{
  private readonly AsyncManualResetEvent _connected =
      new AsyncManualResetEvent();

  public async Task WaitForConnectedAsync()
  {
    await _connected.WaitAsync();
  }

  public void ConnectedChanged(bool connected)
  {
    if (connected)
      _connected.Set();
    else
      _connected.Reset();
  }
}
```

Пояснение

Сигналы представляют собой механизм уведомлений общего назначения. Но если этот «сигнал» представляет собой *сообщение*, используемое для отправки данных от одной части кода в другую, рассмотрите возможность использования очереди «производитель/потребитель». Не стоит использовать и сигналы общего назначения для простой координации доступа к общим данным; в таких ситуациях следует применять асинхронную блокировку.

Тип `AsyncManualResetEvent` находится в пакете `Nito.AsyncEx`.

Дополнительная информация

В рецепте 9.8 рассматриваются асинхронные очереди «производитель/потребитель».

В рецепте 12.2 рассматриваются асинхронные блокировки.

В рецепте 12.3 рассматриваются блокирующие сигналы, которые могут использоваться для передачи уведомлений между потоками.

12.5. Регулировка

Задача

Имеется код с высокой степенью конкурентности — даже слишком высокой. Требуется найти способ скорректировать конкурентность.

Конкурентность оказывается чрезмерной, если части приложения не успевают друг за другом, элементы данных накапливаются и занимают память. В этом сценарии регулировка частей кода может предотвратить проблемы с памятью.

Решение

Решение зависит от типа конкурентности, используемой в вашем коде. Все представленные решения ограничивают конкурентность конкретным значением. В Reactive Extensions предусмотрены более разнообразные возможности — например, скользящие временные окна; регулировка по наблюдаемым объектам System.Reactive более подробно рассматривается в рецепте 6.4.

В Dataflow и в параллельном коде существуют встроенные параметры для регулировки степени конкурентности:

```
IPropagatorBlock<int, int> DataflowMultiplyBy2()
{
  var options = new ExecutionDataflowBlockOptions
  {
    MaxDegreeOfParallelism = 10
  };
```

```
  return new TransformBlock<int, int>(data => data * 2, options);
}

// Использование Parallel LINQ (PLINQ)
IEnumerable<int> ParallelMultiplyBy2(IEnumerable<int> values)
{
  return values.AsParallel()
      .WithDegreeOfParallelism(10)
      .Select(item => item * 2);
}

// Использование класса Parallel
void ParallelRotateMatrices(IEnumerable<Matrix> matrices, float degrees)
{
  var options = new ParallelOptions
  {
    MaxDegreeOfParallelism = 10
  };
  Parallel.ForEach(matrices, options, matrix => matrix.Rotate(degrees));
}
```

Конкурентный асинхронный код может регулироваться с помощью `SemaphoreSlim`:

```
async Task<string[]> DownloadUrlsAsync(HttpClient client,
    IEnumerable<string> urls)
{
  using var semaphore = new SemaphoreSlim(10);
  Task<string>[] tasks = urls.Select(async url =>
  {
    await semaphore.WaitAsync();
    try
    {
      return await client.GetStringAsync(url);
    }
    finally
    {
      semaphore.Release();
    }
  }).ToArray();
  return await Task.WhenAll(tasks);
}
```

Пояснение

Регулировка может быть необходимой тогда, когда вы обнаруживаете, что код задействует слишком много ресурсов (например, процессорного времени или сетевых подключений). Учтите, что конечные пользователи обычно работают на машинах, менее мощных, чем у разработчиков, поэтому чрезмерная регулировка обычно лучше недостаточной.

Дополнительная информация

В рецепте 6.4 рассматривается регулировка реактивного кода.

ГЛАВА 13

Планирование

Каждая часть должна выполняться в каком-то потоке. *Планировщик* (scheduler) — объект, который решает, где должен выполняться тот или иной код. В фреймворке .NET существует несколько разных типов планировщиков, которые по-разному используются параллельным кодом и кодом потоков данных.

Рекомендую при возможности *не* задавать планировщика; обычно настройки по умолчанию работают правильно. Например, оператор `await` в асинхронном коде автоматически возобновит выполнение метода в том же контексте, если только вы не переопределите значение по умолчанию, как описано в рецепте 2.7. У реактивного кода тоже имеются разумные контексты по умолчанию для выдачи событий, хотя их можно переопределить с помощью `ObserveOn`, как описано в рецепте 6.2.

Если другой код должен выполняться в конкретном контексте (например, в контексте UI-потока или в контексте запроса ASP.NET), то рецепты этой главы помогут в планировании кода.

13.1. Планирование работы в пуле потоков

Задача

Имеется фрагмент кода, который должен выполняться в потоке из пула потоков.

Решение

В большинстве случаев следует использовать `Task.Run`; это достаточно просто. Следующий пример блокирует поток из пула потоков на 2 секунды:

```
Task task = Task.Run(() =>
{
  Thread.Sleep(TimeSpan.FromSeconds(2));
});
```

`Task.Run` также поддерживает возвращаемые значения и асинхронные лямбда-выражения. Задача, возвращаемая `Task.Run` в следующем коде, завершится через 2 секунды с результатом 13:

```
Task<int> task = Task.Run(async () =>
{
  await Task.Delay(TimeSpan.FromSeconds(2));
  return 13;
});
```

`Task.Run` возвращает объект `Task` (или `Task<T>`), который может естественным образом потребляться асинхронным или реактивным кодом.

Пояснение

`Task.Run` идеально подходит для UI-приложений с продолжительной работой, которая не должна выполняться в UI-потоке. Например, в рецепте 8.4 `Task.Run` используется для вынесения параллельной обработки в поток из пула потоков. Тем не менее не используйте `Task.Run` в ASP.NET, если только вы не уверены в том, что делаете. В ASP.NET код обработки запросов уже выполняется в потоке из пула потоков, так что перенесение его в другой поток из пула потоков обычно нерационально.

`Task.Run` является фактической заменой для `BackgroundWorker`, `Delegate.BeginInvoke` и `ThreadPool.QueueUserWorkItem`. Ни один из этих старых API не следует использовать в новом коде; код с `Task.Run` намного проще пишется и сопровождается со временем. Более того, `Task.Run` справляется с большинством задач, для которых используется `Thread`, так что в большинстве случаев `Thread` может заменяться `Task.Run` (за редким исключением потоков из модели однопоточного подразделения).

Параллельный код и код потоков данных выполняется в пуле потоков по умолчанию, поэтому обычно `Task.Run` не нужно использовать с кодом, выполняемым `Parallel`, библиотекой TPL Dataflow или Parallel LINQ.

Если вы применяете динамический параллелизм, используйте `Task.Factory.StartNew` вместо `Task.Run`. Это необходимо из-за того, что у объекта `Task`, возвращаемого `Task.Run`, параметры по умолчанию настроены для асинхронного использования (т. е. для потребления в асинхронном или реактивном коде). Кроме того, он не поддерживает такие расширенные возможности, как задачи «родитель/потомок», типичные для динамического параллельного кода.

Дополнительная информация

В рецепте 8.6 рассматривается потребление асинхронного кода (например, задачи, возвращенной `Task.Run`) из реактивного кода.

В рецепте 8.4 рассматривается асинхронное ожидание параллельного кода, проще всего реализуемое с использованием `Task.Run`.

В рецепте 4.4 рассматривается динамический параллелизм — сценарий, в котором следует использовать `Task.Factory.StartNew` вместо `Task.Run`.

13.2. Выполнение кода с помощью планировщика задач

Задача

Есть несколько частей кода, которые требуется выполнить определенным способом. Например, все части кода должны выполняться в UI-потоке или же в любой момент времени должно выполняться только определенное количество частей.

В этом рецепте показано, как определить и сконструировать планировщик для этих частей кода. Применению планировщика будут посвящены следующие два рецепта.

Решение

В .NET есть немало типов, предназначенных для работы с планированием; в этом рецепте мы сосредоточимся на типе `TaskScheduler`, потому что он портируем и относительно прост в использовании.

Простейшая разновидность `TaskScheduler` — `TaskScheduler.Default` — ставит работу в очередь пула потоков. Вам редко придется использовать `TaskScheduler.Default` в своем коде, но важно о нем помнить, потому что этот планировщик используется по умолчанию во многих сценариях планирования. `TaskScheduler.Default` используется `Task.Run` в параллельном коде и в коде потоков данных.

Вы можете сохранить конкретный контекст и позднее спланировать работу в этом контексте с помощью `TaskScheduler.FromCurrentSynchronizationContext`:

```
TaskScheduler scheduler =
   TaskScheduler.FromCurrentSynchronizationContext();
```

Этот код создает объект `TaskScheduler`, чтобы сохранить текущий объект `SynchronizationContext` и спланировать выполнение кода в этом контексте. Тип `SynchronizationContext` представляет контекст планирования общего назначения. В фреймворке .NET предусмотрено несколько разных контекстов; многие UI-фреймворки предоставляют контекст `SynchronizationContext`, представляющий UI-поток, а в ASP.NET до Core предоставлялся контекст `SynchronizationContext`, представляющий контекст запроса HTTP.

`ConcurrentExclusiveSchedulerPair` — еще один высокоэффективный тип, появившийся в .NET 4.5; в действительности это два планировщика, связанных друг с другом. Компонент `ConcurrentScheduler` содержит планировщик, позволяющий нескольким задачам выполняться одновременно — при условии, что ни одна задача не выполняется в `ExclusiveScheduler`. `ExclusiveScheduler` выполняет только по одной задаче за раз и только в том случае, если в настоящее время никакие задачи не выполняются в `ConcurrentScheduler`:

```
var schedulerPair = new ConcurrentExclusiveSchedulerPair();
TaskScheduler concurrent = schedulerPair.ConcurrentScheduler;
TaskScheduler exclusive = schedulerPair.ExclusiveScheduler;
```

Одно из частых применений `ConcurrentExclusiveSchedulerPair` — простое использование `ExclusiveScheduler`, гарантирующее, что в любой момент времени будет выполняться только одна задача. Код, выполняемый в `ExclusiveScheduler`, будет выполняться в пуле потоков, но будет ограничен монопольным выполнением без всего остального кода с использованием экземпляра `ExclusiveScheduler`.

Также ConcurrentExclusiveSchedulerPair может выполняться в качестве регулирующего планировщика. Вы можете создать объект ConcurrentExclusiveSchedulerPair, который будет ограничивать собственный уровень параллелизма. В этом сценарии ExclusiveScheduler обычно не используется:

```
var schedulerPair = new ConcurrentExclusiveSchedulerPair(
    TaskScheduler.Default, maxConcurrencyLevel: 8);
TaskScheduler scheduler = schedulerPair.ConcurrentScheduler;
```

Учтите, что такая регулировка влияет на код только во время его выполнения; она сильно отличается от логической регулировки, рассмотренной в рецепте 12.5. В частности, асинхронный код не считается выполняемым во время ожидания операции. ConcurrentScheduler регулирует выполняющийся код; другие виды регулировки (такие, как SemaphoreSlim) осуществляют регулировку на более высоком уровне (т. е. всего async-метода.)

Пояснение

Возможно, вы заметили, что в последнем примере конструктору ConcurrentExclusiveSchedulerPair передается объект TaskScheduler.Default. Это объясняется тем, что ConcurrentExclusiveSchedulerPair применяет свою конкурентную/монопольную логику к существующему TaskScheduler.

В этом рецепте представлен метод TaskScheduler.FromCurrentSynchronizationContext, используемый для выполнения кода в сохраненном контексте. Также возможно напрямую использовать SynchronizationContext для выполнения кода в этом контексте; тем не менее я не рекомендую применять этот подход. Там, где это возможно, используйте оператор await для возобновления в неявно сохраненном контексте или обертку TaskScheduler.

Никогда не используйте платформенно-зависимые типы для выполнения кода в UI-потоке. WPF, Silverlight, iOS и Android предоставляют тип Dispatcher, Universal Windows использует тип CoreDispatcher, а в Windows Forms существует интерфейс ISynchronizeInvoke (т. е. Control.Invoke). Не используйте эти типы в новом коде; просто считайте, что их вообще нет. Эти типы только без всякой необходимости привязывают код к конкретной платформе. SynchronizationContext — абстракция общего назначения на базе этих типов.

В System.Reactive (Rx) появилась еще более универсальная абстракция планировщика: IScheduler. Планировщик Rx способен инкапсулировать

любую разновидность планировщика; `TaskPoolScheduler` инкапсулирует любой объект `TaskFactory` (который содержит `TaskScheduler`). Команда Rx также определила реализацию `IScheduler`, которой можно управлять вручную в целях тестирования. Если вам потребовалось использовать абстракцию планировщика, я рекомендую `IScheduler` из Rx; она хорошо спроектирована, четко определена и удобна для тестирования. В большинстве случаев абстракция планировщика не нужна, а более ранние библиотеки — такие, как Task Parallel Library (TPL) и TPL Dataflow, — «понимают» только тип `TaskScheduler`.

Дополнительная информация

В рецепте 13.3 рассматривается применение `TaskScheduler` в параллельном коде.

В рецепте 13.4 рассматривается применение `TaskScheduler` в коде потоков данных.

В рецепте 12.5 рассматривается высокоуровневая логическая регулировка.

В рецепте 6.2 рассматриваются планировщики System.Reactive для потоков событий.

В рецепте 7.6 рассматривается тестовый планировщик System.Reactive.

13.3. Планирование параллельного кода

Задача

Требуется управлять выполнением отдельных фрагментов в параллельном коде.

Решение

После того как вы создадите нужный экземпляр `TaskScheduler` (см. рецепт 13.2), можете включить его в набор параметров, передаваемых методу `Parallel`. Следующий код получает последовательность последовательностей матриц. Он запускает несколько параллельных циклов и огра-

ничивает общий параллелизм всех циклов одновременно независимо от количества матриц в каждой последовательности:

```
void RotateMatrices(IEnumerable<IEnumerable<Matrix>> collections,
   float degrees)
{
  var schedulerPair = new ConcurrentExclusiveSchedulerPair(
      TaskScheduler.Default, maxConcurrencyLevel: 8);
  TaskScheduler scheduler = schedulerPair.ConcurrentScheduler;
  ParallelOptions options = new ParallelOptions { TaskScheduler =
     scheduler };
  Parallel.ForEach(collections, options,
      matrices => Parallel.ForEach(matrices, options,
          matrix => matrix.Rotate(degrees)));
}
```

Пояснение

`Parallel.Invoke` также получает экземпляр `ParallelOptions`, поэтому вы можете передать `TaskScheduler` при вызове `Parallel.Invoke` так же, как и для `Parallel.ForEach`. При выполнении динамического параллельного кода можно передать `TaskScheduler` непосредственно `TaskFactory.StartNew` или `Task.ContinueWith`.

Передать `TaskScheduler` коду Parallel LINQ (PLINQ) невозможно.

Дополнительная информация

В рецепте 13.2 рассматриваются основные планировщики задач и рекомендации по выбору между ними.

13.4. Синхронизация потоков данных с помощью планировщиков

Задача

Требуется управлять выполнением отдельных фрагментов в коде потоков данных.

Решение

После того как вы создадите нужный экземпляр `TaskScheduler` (см. рецепт 13.2), вы можете включить его в набор параметров, передаваемых блоку потока данных. При вызове из UI-потока следующий код создает сеть потока данных, которая умножает все свои входные значения на 2 (с использованием пула потоков), а затем присоединяет полученные значения к списку (в UI-потоке):

```
var options = new ExecutionDataflowBlockOptions
{
  TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext(),
};
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var displayBlock = new ActionBlock<int>(
    result => ListBox.Items.Add(result), options);
multiplyBlock.LinkTo(displayBlock);
```

Пояснение

Назначение `TaskScheduler` особенно полезно при координации действий блоков в разных частях вашей сети потока данных. Например, можно воспользоваться `ConcurrentExclusiveSchedulerPair.ExclusiveScheduler`, чтобы блоки A и C никогда не выполнялись одновременно, а блок B мог выполняться тогда, когда пожелает.

Учтите, что синхронизация `TaskScheduler` действует только во время *выполнения* кода. Например, если имеется блок действия, который выполняет асинхронный код и применяет монопольный планировщик, код не считается выполняющимся во время ожидания.

`TaskScheduler` можно задать для любой разновидности блоков потока данных. Даже при том, что блок может не выполнять ваш код (например, `BufferBlock<T>`), у него все равно имеются служебные задачи, которые необходимо выполнять, и блок будет использовать предоставленный объект `TaskScheduler` для всей своей внутренней работы.

Дополнительная информация

В рецепте 13.2 рассматриваются основные планировщики задач и рекомендации по выбору между ними.

ГЛАВА 14

Сценарии

В этой главе будут рассмотрены различные типы и методы для типичных сценариев, встречающихся при написании параллельных программ. О таких сценариях можно было бы написать отдельную книгу, поэтому я выбрал лишь несколько примеров, которые показались мне наиболее полезными.

14.1. Инициализация совместных ресурсов

Задача

Имеется ресурс, совместно используемый несколькими частями кода. Требуется инициализировать этот ресурс при первом обращении к нему.

Решение

В фреймворк .NET включен тип, специально предназначенный для этой цели, — `Lazy<T>`. Экземпляр типа `Lazy<T>` конструируется фабричным делегатом, который используется для инициализации экземпляра. Затем экземпляр становится доступным через свойство `Value`. Следующий пример показывает использование типа `Laz<T>`:

```
static int _simpleValue;
static readonly Lazy<int> MySharedInteger = new Lazy<int>(() =>
   _simpleValue++);
void UseSharedInteger()
{
  int sharedValue = MySharedInteger.Value;
}
```

Сколько бы потоков ни вызывало `UseSharedInteger` одновременно, фабричный делегат выполняется только один раз, и все потоки ожидают одного экземпляра. После того как экземпляр будет создан, он кэшируется, и все будущие обращения к свойству `Value` возвращают тот же экземпляр (в приведенном примере `MySharedInteger.Value` всегда будет содержать `0`).

Очень похожее решение может использоваться в том случае, если инициализация требует асинхронной работы; используйте `Lazy<Task<T>>`:

```
static int _simpleValue;
static readonly Lazy<Task<int>> MySharedAsyncInteger =
    new Lazy<Task<int>>(async () =>
    {
      await Task.Delay(TimeSpan.FromSeconds(2)).ConfigureAwait(false);
      return _simpleValue++;
    });

async Task GetSharedIntegerAsync()
{
  int sharedValue = await MySharedAsyncInteger.Value;
}
```

В этом примере делегат возвращает `Task<int>`, т. е. целое значение, определяемое асинхронно. Сколько бы частей кода ни вызывало `Value` одновременно, `Task<int>` создается только один раз и возвращается всем вызывающим сторонам. Каждая вызывающая сторона получает возможность (асинхронно) ожидать завершения задачи, для чего задача передается `await`.

Этот паттерн может использоваться на практике, но необходимо учесть ряд дополнительных аспектов. Во-первых, асинхронный делегат может быть выполнен в любом потоке, который вызывает `Value`, и делегат будет выполняться в этом контексте. Если существуют разные типы потоков, которые могут вызывать `Value` (например, UI-поток и поток из пула потоков или потоки двух разных запросов ASP.NET), возможно, будет лучше, если асинхронный делегат будет всегда выполняться в потоке из пула. Это легко сделать, заключив фабричного делегата в вызов `Task.Run`:

```
static int _simpleValue;
static readonly Lazy<Task<int>> MySharedAsyncInteger =
  new Lazy<Task<int>>(() => Task.Run(async () =>
```

```
  {
    await Task.Delay(TimeSpan.FromSeconds(2));
    return _simpleValue++;
  }));

async Task GetSharedIntegerAsync()
{
  int sharedValue = await MySharedAsyncInteger.Value;
}
```

Другой аспект заключается в том, что экземпляр `Task<T>` создается только один раз. Если асинхронный делегат выдаст исключение, то `Lazy<Task<T>>` будет кэшировать эту задачу с ошибкой. Такая ситуация нежелательна; в большинстве случаев лучше снова выполнить делегата при следующем запросе отложенного значения вместо того, чтобы кэшировать исключение. Механизма «сброса» `Lazy<T>` не существует, но можно создать новый класс, который обеспечивает повторное создание экземпляра `Lazy<T>`:

```
public sealed class AsyncLazy<T>
{
  private readonly object _mutex;
  private readonly Func<Task<T>> _factory;
  private Lazy<Task<T>> _instance;

  public AsyncLazy(Func<Task<T>> factory)
  {
    _mutex = new object();
    _factory = RetryOnFailure(factory);
    _instance = new Lazy<Task<T>>(_factory);
  }

  private Func<Task<T>> RetryOnFailure(Func<Task<T>> factory)
  {
    return async () =>
    {
      try
      {
        return await factory().ConfigureAwait(false);
      }
      catch
      {
        lock (_mutex)
```

```
        {
          _instance = new Lazy<Task<T>>(_factory);
        }
        throw;
      }
    };
  }

  public Task<T> Task
  {
    get
    {
      lock (_mutex)
        return _instance.Value;
    }
  }
}

static int _simpleValue;
static readonly AsyncLazy<int> MySharedAsyncInteger =
  new AsyncLazy<int>(() => Task.Run(async () =>
  {
    await Task.Delay(TimeSpan.FromSeconds(2));
    return _simpleValue++;
  }));

async Task GetSharedIntegerAsync()
{
  int sharedValue = await MySharedAsyncInteger.Task;
}
```

Пояснение

Последний пример кода в этом рецепте представляет общий паттерн асинхронной отложенной инициализации. Выглядит он несколько неуклюже. Библиотека `AsyncEx` включает тип `AsyncLazy<T>`, который работает, как тип `Lazy<Task<T>>`, выполняющий своего фабричного делегата в пуле потоков с возможностью повторения попытки при неудаче. Возможно и прямое ожидание `await`, так что код объявления и использования выглядит примерно так:

```
static int _simpleValue;
private static readonly AsyncLazy<int> MySharedAsyncInteger =
  new AsyncLazy<int>(async () =>
  {
    await Task.Delay(TimeSpan.FromSeconds(2));
    return _simpleValue++;
  },
  AsyncLazyFlags.RetryOnFailure);

public async Task UseSharedIntegerAsync()
{
  int sharedValue = await MySharedAsyncInteger;
}
```

Тип `AsyncLazy<T>` находится в пакете `Nito.AsyncEx`.

Дополнительная информация

В главе 1 рассматриваются основы программирования `async`/`await`.

В рецепте 13.1 рассматривается планирование работы в пуле потоков.

14.2. Отложенное вычисление в System.Reactive

Задача

Требуется создать новый наблюдаемый объект каждый раз, когда кто-то на него подписывается. Например, каждая подписка может представлять отдельный запрос к веб-службе.

Решение

В библиотеке System.Reactive существует оператор `Observable.Defer`, который выполняет делегата при каждой подписке на наблюдаемый объект. Делегат работает как фабрика, создающая наблюдаемый объект. В следующем примере `Defer` используется для вызова асинхронного метода каждый раз, когда кто-то подписывается на наблюдаемый объект:

```
void SubscribeWithDefer()
{
  var invokeServerObservable = Observable.Defer(
      () => GetValueAsync().ToObservable());
  invokeServerObservable.Subscribe(_ => { });
  invokeServerObservable.Subscribe(_ => { });

  Console.ReadKey();
}

async Task<int> GetValueAsync()
{
  Console.WriteLine("Calling server...");
  await Task.Delay(TimeSpan.FromSeconds(2));
  Console.WriteLine("Returning result...");
  return 13;
}
```

При выполнении этого кода будет получен следующий результат:

```
Calling server...
Calling server...
Returning result...
Returning result...
```

Пояснение

Ваш собственный код обычно не подписывается на наблюдаемый объект более одного раза, но некоторые операторы System.Reactive поступают так в своей реализации. Например, оператор `Observable.While` заново подписывается на исходную последовательность, пока его условие остается истинным. `Defer` позволяет определить наблюдаемый объект, который заново вычисляется каждый раз, когда поступает новая подписка. Это может быть полезно, если потребуется обновить данные для этого наблюдаемого объекта.

Дополнительная информация

В рецепте 8.6 рассматривается инкапсуляция асинхронных методов в наблюдаемых объектах.

14.3. Асинхронное связывание данных

Задача

Данные загружаются асинхронно. Требуется осуществить связывание данных с результатами (например, в компоненте модели представления (ViewModel) в архитектуре «модель—представление—модель представления»).

Решение

Если свойство используется в связывании данных, оно должно немедленно и синхронно вернуть некое подобие результата. Если фактическое значение должно определяться асинхронно, вы можете вернуть результат по умолчанию и позднее обновить свойство правильным значением.

Помните, что асинхронные операции могут завершаться не только успехом, но и неудачей. Так как вы пишете модель представления, связывание данных может использоваться для обновления пользовательского интерфейса и для ситуации ошибки.

В библиотеке `Nito.Mvvm.Async` имеется тип `NotifyTask`, который может использоваться для этой цели:

```
class MyViewModel
{
  public MyViewModel()
  {
    MyValue = NotifyTask.Create(CalculateMyValueAsync());
  }

  public NotifyTask<int> MyValue { get; private set; }

  private async Task<int> CalculateMyValueAsync()
  {
    await Task.Delay(TimeSpan.FromSeconds(10));
    return 13;
  }
}
```

Как показывает следующий пример, связывание данных может применяться к различным свойствам по свойству `NotifyTask<T>`:

```
<Grid>
  <Label Content="Loading..."
      Visibility="{Binding MyValue.IsNotCompleted,
          Converter={StaticResource BooleanToVisibilityConverter}}"/>
  <Label Content="{Binding MyValue.Result}"
      Visibility="{Binding MyValue.IsSuccessfullyCompleted,
          Converter={StaticResource BooleanToVisibilityConverter}}"/>
  <Label Content="An error occurred" Foreground="Red"
      Visibility="{Binding MyValue.IsFaulted,
          Converter={StaticResource BooleanToVisibilityConverter}}"/>
</Grid>
```

В библиотеку `MvvmCross` входит тип `MvxNotifyTask`, очень похожий на `NotifyTask<T>`.

Пояснение

Можно написать собственную обертку связывания данных (вместо использования обертки из библиотек). Следующий код дает примерное представление о том, как это делается:

```
class BindableTask<T> : INotifyPropertyChanged
{
  private readonly Task<T> _task;

  public BindableTask(Task<T> task)
  {
    _task = task;
    var _ = WatchTaskAsync();
  }

  private async Task WatchTaskAsync()
  {
    try
    {
      await _task;
    }
    catch
    {
    }
```

```
    OnPropertyChanged("IsNotCompleted");
    OnPropertyChanged("IsSuccessfullyCompleted");
    OnPropertyChanged("IsFaulted");
    OnPropertyChanged("Result");
  }

  public bool IsNotCompleted { get { return !_task.IsCompleted; } }
  public bool IsSuccessfullyCompleted
  {
    get { return _task.Status == TaskStatus.RanToCompletion; }
  }
  public bool IsFaulted { get { return _task.IsFaulted; } }
  public T Result
  {
    get { return IsSuccessfullyCompleted ? _task.Result : default; }
  }

  public event PropertyChangedEventHandler PropertyChanged;

  protected virtual void OnPropertyChanged(string propertyName)
  {
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs
       (propertyName));
  }
}
```

Обратите внимание: пустое условие `catch` использовано намеренно — мы хотим перехватывать все исключения и обрабатывать их через механизм связывания данных. Также в коде не должен использоваться вызов `ConfigureAwait(false)`, потому что событие `PropertyChanged` должно выдаваться в UI-потоке.

Тип `NotifyTask` находится в NuGet-пакете `Nito.Mvvm.Async`. Тип `MvxNotifyTask` находится в NuGet-пакете `MvvmCross`.

Дополнительная информация

В главе 1 рассматриваются основы программирования `async`/`await`.

В рецепте 2.7 рассматривается использование `ConfigureAwait`.

14.4. Неявное состояние

Задача

Имеются переменные состояния, которые должны быть доступны в разных точках стека вызовов. Например, идентификатор текущей операции должен использоваться для вывода информации в журнал, но вы не хотите добавлять его в виде параметра в каждый метод.

Решение

Лучшее решение — добавить параметры к методам, сохранить данные в полях класса или воспользоваться внедрением зависимостей для предоставления данных разным частям вашего кода. Но в некоторых ситуациях это приведет к чрезмерному усложнению кода.

Тип `AsyncLocal<T>` позволяет связать с состоянием объект, в котором оно сможет существовать в логическом «контексте». Следующий код демонстрирует использование `AsyncLocal<T>` для назначения идентификатора операции, который позднее читается методом журнального вывода:

```
private static AsyncLocal<Guid> _operationId = new AsyncLocal<Guid>();

async Task DoLongOperationAsync()
{
  _operationId.Value = Guid.NewGuid();

  await DoSomeStepOfOperationAsync();
}

async Task DoSomeStepOfOperationAsync()
{
  await Task.Delay(100); // Некоторая асинхронная работа

  // Вывод в журнал.
  Trace.WriteLine("In operation: " + _operationId.Value);
}
```

Во многих случаях бывает полезно создать более сложную структуру данных (например, стек) в одном экземпляре `AsyncLocal<T>`. Это возможно с одной оговоркой: в `AsyncLocal<T>` следует хранить только неизменяемые данные. Каждый раз, когда возникнет необходимость в обновлении данных, вы должны перезаписать существующее значение. Часто бывает полезно скрыть `AsyncLocal<T>` внутри вспомогательного типа, который гарантирует неизменяемость хранимых данных и их корректное обновление:

```
internal sealed class AsyncLocalGuidStack
{
  private readonly AsyncLocal<ImmutableStack<Guid>> _operationIds =
      new AsyncLocal<ImmutableStack<Guid>>();

  private ImmutableStack<Guid> Current =>
      _operationIds.Value ?? ImmutableStack<Guid>.Empty;

  public IDisposable Push(Guid value)
  {
    _operationIds.Value = Current.Push(value);
    return new PopWhenDisposed(this);
  }

  private void Pop()
  {
    ImmutableStack<Guid> newValue = Current.Pop();
    if (newValue.IsEmpty)
      newValue = null;
    _operationIds.Value = newValue;
  }

  public IEnumerable<Guid> Values => Current;

  private sealed class PopWhenDisposed : IDisposable
  {
    private AsyncLocalGuidStack _stack;

    public PopWhenDisposed(AsyncLocalGuidStack stack) =>
        _stack = stack;

    public void Dispose()
    {
```

```
      _stack?.Pop();
      _stack = null;
    }
  }
}

private static AsyncLocalGuidStack _operationIds = new
   AsyncLocalGuidStack();

async Task DoLongOperationAsync()
{
  using (_operationIds.Push(Guid.NewGuid()))
    await DoSomeStepOfOperationAsync();
}

async Task DoSomeStepOfOperationAsync()
{
  await Task.Delay(100); // Некоторая асинхронная работа

  // Вывод в журнал.
  Trace.WriteLine("In operation: " +
      string.Join(":", _operationIds.Values));
}
```

Тип-обертка гарантирует, что используемые данные неизменяемы и что новые значения будут заноситься в стек. Он также предоставляет удобный способ извлечения значений из стека через `IDisposable`.

Пояснение

В старом коде можно использовать атрибут `ThreadStatic` для контекстного состояния, используемого синхронным кодом. При преобразовании старого кода в асинхронный `AsyncLocal<T>` является основным кандидатом для замены `ThreadStaticAttribute`. `AsyncLocal<T>` работает как для синхронного, так и для асинхронного кода, и этот способ должен использоваться по умолчанию для неявного состояния в современных приложениях.

Дополнительная информация

В главе 1 рассматриваются основы программирования `async`/`await`.

В главе 9 рассматривается применение неизменяемых коллекций для сохранения сложных данных в форме неявного состояния.

14.5. Идентичный синхронный и асинхронный код

Задача

Имеется код, к которому нужно предоставить доступ через синхронный и асинхронный API без дублирования логики. Такая ситуация часто встречается при преобразовании существующего кода в асинхронный, если существующие синхронные потребители изменять (пока) нельзя.

Решение

По возможности постарайтесь структурировать свой код в соответствии с современными паттернами проектирования (например, гексагональной архитектуры или архитектуры портов и адаптеров), отделяющими бизнес-логику от побочных эффектов (таких, как ввод/вывод). Если вы сможете добиться этого, то необходимости предоставлять как синхронный, так и асинхронный API для каких-либо целей не будет; бизнес-логика всегда будет синхронной, а ввод/вывод всегда будет асинхронным.

Это весьма амбициозная цель. В Реальном Мире старый код бывает весьма неряшливым, и у нас редко находится время для того, чтобы привести его в идеальное состояние, прежде чем преобразовывать к асинхронному виду. Существующие API часто приходится поддерживать для обеспечения обратной совместимости, даже если они были плохо спроектированы.

Здесь нет идеального решения. Многие разработчики пытаются сделать так, чтобы их синхронный код обращался с вызовами к асинхронному коду или асинхронный код обращался с вызовами к синхронному, но оба подхода представляют собой антипаттерны. В такой ситуации я предпочитаю использовать «трюк с логическим аргументом», который позволяет хранить всю логику в одном методе, предоставляя как синхронный, так и асинхронный API.

Основная идея «трюка с логическим аргументом» заключается в создании приватного базового метода, содержащего логику. Базовый метод

имеет асинхронную сигнатуру и получает логический аргумент, который определяет, должен ли базовый метод быть асинхронным или нет. Если логический аргумент указывает, что базовый метод должен быть синхронным, то он должен вернуть уже завершенную задачу. Тогда вы можете написать методы как асинхронного, так и синхронного API, передающие управление базовому методу:

```
private async Task<int> DelayAndReturnCore(bool sync)
{
  int value = 100;

  // Выполнение некоторой работы.
  if (sync)
    Thread.Sleep(value); // Вызвать синхронный API.
  else
    await Task.Delay(value); // Вызвать асинхронный API.

  return value;
}

// Асинхронный API
public Task<int> DelayAndReturnAsync() =>
    DelayAndReturnCore(sync: false);

// Синхронный API
public int DelayAndReturn() =>
    DelayAndReturnCore(sync: true).GetAwaiter().GetResult();
```

Метод асинхронного API `DelayAndReturnAsync` вызывает `DelayAndReturnCore` с логическим параметром `sync`, равным `false`; это означает, что метод `DelayAndReturnCore` может работать асинхронно и он использует `await` в используемом методе API «асинхронной задержки» `Task.Delay`. Задача, возвращаемая `DelayAndReturnCore`, возвращается напрямую на сторону вызова `DelayAndReturnAsync`.

Метод синхронного API `DelayAndReturn` вызывает `DelayAndReturnCore` с логическим параметром `sync`, равным `true`; это означает, что `DelayAndReturnCore` может работать синхронно и он использует метод API «синхронной задержки» `Thread.Sleep`. Задача, возвращаемая `DelayAndReturnCore`, уже должна быть завершена, что позволяет безопасно получить результат. `DelayAndReturn` использует `GetAwaiter().GetResult()` для получения резуль-

тата от задачи; это позволяет обойтись без обертки `AggregateException`, которая могла бы потребоваться при использовании свойства `Task<T>.Result`.

Пояснение

Такое решение не идеально, но оно может помочь в построении реальных приложений.

Впрочем, необходимо учитывать ряд нюансов. Катастрофические проблемы возникнут в том случае, если метод `Core` неправильно обрабатывает свой параметр `sync`. Если метод `Core` когда-либо вернет незавершенную задачу при условии, что `sync` содержит `true`, то синхронный API может легко создать взаимную блокировку; единственная причина, по которой синхронный API может блокироваться по этой задаче, — если он знает, что задача уже завершена. Аналогично, если метод `Core` блокирует поток при переменной `sync`, равной `false`, то приложение работает не настолько эффективно, насколько могло бы.

Одним из возможных усовершенствований этого решения могло бы стать добавление в синхронном API проверки, которая убеждалась бы в том, что возвращенная задача действительно завершена. Если задача окажется незавершенной, то это указывает на серьезную ошибку программирования.

Дополнительная информация

В главе 1 рассматриваются основы программирования `async`/`await`, включая обсуждение взаимных блокировок, которые могут возникнуть при блокировании по асинхронному коду вообще.

14.6. «Рельсовое» программирование с сетями потоков данных

Задача

Имеется настроенная сеть потоков данных, но при обработке некоторых элементов данных происходит ошибка. Требуется реагировать на эти ошибки так, чтобы сеть потоков данных оставалась работоспособной.

Решение

Если при обработке элемента данных возникает исключение, по умолчанию в этом блоке происходит отказ, что не позволяет ему обрабатывать другие элементы данных. Основная идея решения заключается в том, чтобы рассматривать исключения как другую разновидность данных. Если сеть потока данных работает с типами, которые могут представлять как исключения, так и данные, то она останется работоспособной даже при возникновении исключений и продолжит обрабатывать другие элементы данных.

Иногда такое программирование называется рельсовым (railway), потому что элементы сети могут рассматриваться как двигающиеся по одному из двух путей. Существует нормальный «путь данных»: если все идет нормально, то элемент остается на «пути данных» и перемещается по сети, к нему применяются преобразования и различные операции, пока он не достигнет конца сети. Второй путь — «путь ошибок»; в любом блоке при возникновении исключения при обработке элемента это исключение переходит на «путь ошибок» и перемещается по сети. Элементы исключений не обрабатываются; они всего лишь передаются от блока к блоку, чтобы также достичь конца сети. Терминальные (завершающие) блоки сети в итоге получают последовательность элементов, каждый из которых может быть элементом данных или элементом исключения; элемент данных представляет данные, успешно прошедшие всю сеть, а элемент исключения представляет ошибку обработки в некоторой позиции сети.

Чтобы создать подобную структуру «рельсового» программирования, необходимо сначала определить тип, представляющий элемент данных или исключение. Если вы захотите воспользоваться готовым типом, есть несколько вариантов. Такие типы получили распространение в сообществе функционального программирования, где они обычно называются `Try`, `Error` или `Exceptional`, и являются особым случаем монады `Either`. Я определил собственный тип `Try<T>`, который можно использовать для примера; он находится в пакете `Nito.Try`, а исходный код хранится на GitHub (https://github.com/StephenCleary/Try).

Если имеется тип `Try<T>` или его разновидность, создание сети становится немного монотонным, но не сложным делом. Тип каждого блока потока данных следует заменить с `T` на `Try<T>`, а любая обработка в этом блоке должна осуществляться отображением одного значения `Try<T>` на другое. С моим типом `Try<T>` это делается вызовом `Try<T>.Map`. На мой взгляд, удобно

определить небольшие фабричные методы для «рельсовых» блоков потоков данных вместо того, чтобы включать этот дополнительный код во встроенном виде. Ниже приведен пример вспомогательного метода, который строит блок `TransformBlock`, работающий со значениями `Try<T>` вызовом `Try<T>.Map`:

```
private static TransformBlock<Try<TInput>, Try<TOutput>>
    RailwayTransform<TInput, TOutput>(Func<TInput, TOutput> func)
{
  return new TransformBlock<Try<TInput>, Try<TOutput>>(t =>
     t.Map(func));
}
```

С такими вспомогательными методами код создания сети потоков данных получается более прямолинейным:

```
var subtractBlock = RailwayTransform<int, int>(value => value - 2);
var divideBlock = RailwayTransform<int, int>(value => 60 / value);
var multiplyBlock = RailwayTransform<int, int>(value => value * 2);
var options = new DataflowLinkOptions { PropagateCompletion = true };
subtractBlock.LinkTo(divideBlock, options);
divideBlock.LinkTo(multiplyBlock, options);

// Вставить элементы данных в первый блок.
subtractBlock.Post(Try.FromValue(5));
subtractBlock.Post(Try.FromValue(2));
subtractBlock.Post(Try.FromValue(4));
subtractBlock.Complete();

// Получить элементы данных/исключений из последнего блока.
while (await multiplyBlock.OutputAvailableAsync())
{
  Try<int> item = await multiplyBlock.ReceiveAsync();
  if (item.IsValue)
    Console.WriteLine(item.Value);
  else
    Console.WriteLine(item.Exception.Message);
}
```

Пояснение

«Рельсовое программирование» — отличный способ предотвращения перехода блоков потока данных в состояние отказа. Так как «рельсовое программирование» представляет собой конструкцию функционального

программирования, основанную на монадах, результат его перевода на платформу .NET получается немного громоздким, но работоспособным. Если имеется сеть потока данных, которая должна быть отказоустойчивой, то «рельсовое программирование», безусловно, заслуживает вашего внимания.

Дополнительная информация

В рецепте 5.2 рассматривается обычный процесс того, как исключения переводят блоки в состояние отказа и могут распространяться по сети, если «рельсовое программирование» не используется.

14.7. Регулировка обновлений о ходе выполнения операции

Задача

Имеется продолжительная операция, которая выдает сообщения о прогрессе операции; обновления отображаются в пользовательском интерфейсе. Однако обновления поступают слишком быстро и тормозят работу пользовательского интерфейса.

Решение

В следующем примере выдаются слишком частые уведомления о прогрессе операции:

```
private string Solve(IProgress<int> progress)
{
  // Вести максимально быстрый отсчет в течение 3 секунд.
  var endTime = DateTime.UtcNow.AddSeconds(3);
  int value = 0;
  while (DateTime.UtcNow < endTime)
  {
    value++;
    progress?.Report(value);
  }
  return value.ToString();
}
```

Чтобы выполнить этот код из GUI-приложения, упакуйте его в `Task.Run` и передайте `IProgress<T>`. Следующий пример предназначен для WPF, но используемые концепции действуют независимо от платформы GUI (WPF, Xamarin или Windows Forms):

```
// Для простоты код обновляет надпись напрямую.
// В реальном MVVM-приложении эти присваивания
//  осуществлялись бы обновлением свойства ViewModel,
//  связанного с пользовательским интерфейсом.
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
  MyLabel.Content = "Starting...";
  var progress = new Progress<int>(value => MyLabel.Content = value);
  var result = await Task.Run(() => Solve(progress));
  MyLabel.Content = $"Done! Result: {result}";
}
```

Этот код на некоторое время парализует пользовательский интерфейс (около 20 секунд на моей машине). Затем интерфейс снова начинает работать, и в нем выводится только сообщение `"Done! Result:"`. Вы не увидите промежуточные уведомления о прогрессе. Здесь фоновый код отправляет отчеты о прогрессе UI-потоку слишком быстро — настолько быстро, что после выполнения в течение всего 3 секунд UI-потоку требуется еще 17 секунд или около того для обработки всех этих уведомлений, а текст надписи обновляется снова и снова. Затем UI-поток обновляет надпись в последний раз со значением `"Done! Result:"` и *наконец* получает возможность перерисовать экран с выводом обновленного текста надписи.

Прежде всего следует понять, что сообщения о прогрессе необходимо регулировать. Только так можно гарантировать, что у пользовательского интерфейса будет достаточно времени для перерисовки между обновлениями. Затем необходимо осознать, что регулировка должна осуществляться по *времени*, а не по *количеству* отчетов. Идея регулировки, основанной на отправке одного сообщения из сотни или около того, выглядит заманчиво, но она не идеальна по причинам, изложенным в разделе «Пояснение».

Тот факт, что мы должны иметь дело со *временем*, наводит на мысль, что нам стоит рассмотреть возможность использования System.Reactive. Собственно, в System.Reactive имеются операторы, предназначенные специально для регулировки по времени. Похоже, System.Reactive сможет сыграть положительную роль в этом решении.

Для начала можно определить реализацию `IProgress<T>`, которая выдает событие для каждого отчета о прогрессе, а затем создать наблюдаемый объект, получающий эти отчеты:

```
public static class ObservableProgress
{
  private sealed class EventProgress<T> : IProgress<T>
  {
    void IProgress<T>.Report(T value) => OnReport?.Invoke(value);
    public event Action<T> OnReport;
  }

  public static (IObservable<T>, IProgress<T>) Create<T>()
  {
    var progress = new EventProgress<T>();
    var observable = Observable.FromEvent<T>(
        handler => progress.OnReport += handler,
        handler => progress.OnReport -= handler);
    return (observable, progress);
  }
}
```

Метод `ObservableProgress.Create<T>` создает пару объектов `IObservable<T>` и `IProgress<T>`, при этом все отчеты о прогрессе, отправленные `IProgress<T>`, будут отправляться подписчикам `IObservable<T>`. Теперь мы имеем наблюдаемый поток для отчетов о прогрессе; следующим шагом должна стать его регулировка.

Пользовательский интерфейс должен обновляться медленно, чтобы реагировать на происходящее, и при этом достаточно быстро, чтобы пользователи видели обновления. Человек воспринимает информацию намного медленнее, чем происходит перерисовка экрана, поэтому существует широкий диапазон допустимых значений частоты обновления. Если вы хотите добиться истинного удобства восприятия информации, регулировки до одного обновления в секунду или около того может быть достаточно. Если вы предпочитаете обратную связь, приближенную к реальному времени, используйте одно обновление каждые 100 или 200 миллисекунд (мс). Оно будет достаточно быстрым, чтобы пользователь понимал суть происходящего и имел общее представление о прогрессе, но при этом достаточно медленным, чтобы пользовательский интерфейс успевал реагировать на происходящее.

Следует помнить и о том, что отчеты о прогрессе могут выдаваться из других потоков — в данном случае они выдаются из фонового потока. Регулировка должна происходить как можно ближе к источнику, поэтому желательно вынести ее в фоновый поток. Однако код, обновляющий пользовательский интерфейс, должен выполняться в UI-потоке. С учетом этого факта можно определить метод `CreateForUi`, который обеспечивает как регулировку, так и переход в UI-поток:

```
public static class ObservableProgress
{
  // Примечание: должен вызываться из UI-потока.
  public static (IObservable<T>, IProgress<T>) CreateForUi<T>(
      TimeSpan? sampleInterval = null)
  {
    var (observable, progress) = Create<T>();
    observable = observable
        .Sample(sampleInterval ?? TimeSpan.FromMilliseconds(100))
        .ObserveOn(SynchronizationContext.Current);
    return (observable, progress);
  }
}
```

Теперь у вас имеется вспомогательный метод, который будет регулировать обновления прогресса до того, как они доберутся до пользовательского интерфейса. В приведенном примере вспомогательный метод может использоваться в обработчике щелчка на кнопке:

```
// Для простоты код обновляет надпись напрямую.
// В реальном MVVM-приложении эти присваивания
//  осуществлялись бы обновлением свойства ViewModel,
//  связанного с пользовательским интерфейсом.
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
  MyLabel.Content = "Starting...";
  var (observable, progress) = ObservableProgress.CreateForUi<int>();
  string result;
  using (observable.Subscribe(value => MyLabel.Content = value))
    result = await Task.Run(() => Solve(progress));
  MyLabel.Content = $"Done! Result: {result}";
}
```

Новый код вызывает наш вспомогательный метод `ObservableProgress.CreateForUi`, который создает пару `IObservable<T>` и `IProgress<T>`. Код подписывается на обновления о прогрессе и продолжает выполнение до тех пор, пока `Solve` не завершится. Наконец, `IProgress<T>` передается методу `Solve` с длительным выполнением. Когда `Solve` вызывает `IProgress<T>.Report`, сначала производится выборка этих отчетов в 100-миллисекундном окне; одно обновление за каждые 100 миллисекунд передается UI-потоку и используется для обновления текста надписи. Теперь пользовательский интерфейс сохраняет высокую скорость отклика!

Пояснение

Этот рецепт представляет собой интересную комбинацию других рецептов из книги. В нем не представлены никакие новые приемы; чтобы получить искомое решение, мы просто объединили ряд предшествующих рецептов.

У задачи есть и другое альтернативное решение, которое часто встречается на практике, — «решение с делением». Суть в том, что метод `Solve` сам регулирует свои обновления прогресса; например, если код хочет обрабатывать только одно обновление на каждые 100 фактических обновлений, то в коде можно использовать проверку с вычислением остатка вида

```
if (value % 100 == 0) progress?.Report(value);
```

У этого решения есть пара недостатков. Во-первых, «правильного» делителя не существует; обычно разработчик перебирает разные значения, пока не найдет то, которое хорошо работает на его машине. Однако тот же код может не лучшим образом работать на гигантском сервере клиента или на недостаточно мощной виртуальной машине. Кроме того, в разных платформах и средах по-разному организуется кэширование, в результате чего код может работать намного быстрее (или медленнее), чем ожидалось. И конечно, мощь «новейшего» компьютерного оборудования тоже изменяется со временем. Таким образом, значение делителя выбирается в какой-то степени случайно; оно не будет правильным везде и всегда.

Другой недостаток этого решения заключается в том, что оно пытается исправить проблему в неправильной части кода. Эта проблема относится исключительно к пользовательскому интерфейсу, и ее решение должен предоставлять пользовательский интерфейс. В примере этого рецепта метод `Solve` представляет некоторую фоновую бизнес-логику обработки;

он не должен беспокоиться о проблемах, присущих пользовательскому интерфейсу. Возможно, в консольном приложении будет использоваться совсем не такой делитель, как в приложении WPF.

С другой стороны, решение с делением правильно в том, что обновления лучше регулировать перед отправкой обновлений UI-потоку. Решение в этом рецепте также действует по этому принципу: оно регулирует обновления немедленно и синхронно в фоновом потоке перед отправкой UI-потоку. Внедряя собственную реализацию `IProgress<T>`, пользовательский интерфейс может выполнить собственную регулировку, не требуя никаких изменений в самом методе `Solve`.

Дополнительная информация

В рецепте 2.3 рассматривается использование `IProgress<T>` для уведомлений о прогрессе продолжительных операций.

В рецепте 13.1 рассматривается использование `Task.Run` для синхронного кода в потоке из пула потоков.

В рецепте 6.1 рассматривается использование `FromEvent` для упаковки событий .NET в наблюдаемых объектах.

В рецепте 6.4 рассматривается использование `Sample` для регулировки наблюдаемых объектов по времени.

В рецепте 6.2 рассматривается использование `ObserveOn` для перемещения наблюдаемых уведомлений в другой контекст.

ПРИЛОЖЕНИЕ А

Поддержка унаследованных платформ

Многие технологии, описанные в книге, также в той или иной степени поддерживаются на старых платформах. Если вы оказались в незавидной ситуации и вам приходится поддерживать эти платформы, то это приложение поможет определить, какие технологии вам доступны. Использование этих технологий на старых платформах не идеально; но даже если вы добьетесь своего и код заработает, учтите, что единственным долгосрочным решением является обновление целевой платформы для кода. Предполагается, что это приложение представляет собой историческую справку, а не набор рекомендаций; возможно, оно будет полезно программистам, занимающимся сопровождением старого кода.

В табл. А.1 приведена сводка поддержки старых платформ для разных технологий.

Таблица А.1. Поддержка старых платформ

Платформа	async	Parallel	Reactive	Dataflow	Параллельные коллекции	Неизменяемые коллекции
.NET 4.5	✓	✓	NuGet	NuGet	✓	NuGet
.NET 4.0	NuGet	✓	NuGet		✓	X
Windows Phone Apps 8.1	✓	✓	NuGet	NuGet	✓	NuGet
Windows Phone SL 8.0	✓	X	NuGet	NuGet	X	NuGet
Windows Phone SL 7.1	NuGet	X	NuGet	X	X	X
Silverlight 5	NuGet	X	NuGet	X	X	X

Поддержка async на старых платформах

Если вам нужна поддержка `async` на старых платформах, установите NuGet-пакет для `Microsoft.Bcl.Async` (табл. A.2).

Не используйте `Microsoft.Bcl.Async` для включения поддержки `async` на платформе ASP.NET в .NET 4.0! Конвейер ASP.NET был обновлен в .NET 4.5 — в него была включена поддержка `async`, и вы должны использовать .NET 4.5 или более новую версию для async-проектов ASP.NET. Пакет `Microsoft.Bcl.Async` не предназначен для приложений ASP.NET.

Таблица A.2. Поддержка async на старых платформах

Платформа	Поддержка async
.NET 4.5	✓
.NET 4.0	NuGet: `Microsoft.Bcl.Async`
Windows Phone Apps 8.1	✓
Windows Phone SL 8.0	✓
Windows Phone SL 7.1	NuGet: `Microsoft.Bcl.Async`
Silverlight 5	NuGet: `Microsoft.Bcl.Async`

При использовании `Microsoft.Bcl.Async` многие составляющие современного типа `Task` присутствуют в типе `TaskEx`, включая `Delay`, `FromResult`, `WhenAll` и `WhenAny`.

Поддержка Dataflow на старых платформах

Чтобы использовать TPL Dataflow, установите NuGet-пакет `System.Threading.Tasks.Dataflow` в своем приложении. Библиотека TPL Dataflow обладает ограниченной поддержкой старых платформ (табл. A.3).

Не используйте старый пакет `Microsoft.Tpl.Dataflow`. Он больше не поддерживается.

Таблица A.3. Поддержка TPL Dataflow на старых платформах

Платформа	Поддержка TPL Dataflow
.NET 4.5	NuGet: `System.Threading.Tasks.Dataflow`
.NET 4.0	X
Windows Phone Apps 8.1	NuGet: `System.Threading.Tasks.Dataflow`
Windows Phone SL 8.0	NuGet: `System.Threading.Tasks.Dataflow`
Windows Phone SL 7.1	X
Silverlight 5	X

Поддержка System.Reactive на старых платформах

Чтобы использовать System.Reactive, установите NuGet-пакет `System.Reactive` в своем приложении. Библиотека System.Reactive исторически обладает широкой поддержкой разных платформ (табл. A.4); тем не менее многие старые платформы в настоящее время не поддерживаются.

Таблица A.4. Поддержка System.Reactive на старых платформах

Платформа	Поддержка Reactive
.NET 4.7.2	NuGet: `System.Reactive`
.NET 4.5	NuGet: `System.Reactive v3.x`
.NET 4.0	NuGet: `Rx.Main`
Windows Phone Apps 8.1	NuGet: `System.Reactive v3.x`
Windows Phone SL 8.0	NuGet: `System.Reactive v3.x`
Windows Phone SL 7.1	NuGet: `Rx.Main`
Silverlight 5	NuGet: `Rx.Main`

Старый пакет `Rx.Main` больше не поддерживается.

ПРИЛОЖЕНИЕ Б

Распознавание и интерпретация асинхронных паттернов

Преимущества асинхронного кода были хорошо понятны за много лет до изобретения .NET. На заре существования .NET было разработано несколько разных стилей асинхронного кода. Они использовались тут и там, но в итоге от них отказались. Нельзя сказать, что все идеи были плохими; многие из них проложили путь к современному подходу с `async/await`. Но есть еще много старого кода, в котором используются более старые асинхронные паттерны. В этом приложении рассматриваются наиболее распространенные паттерны; я объясню, как они работают и интегрируются с современным кодом.

Иногда тип обновляется с годами и обрастает все большим количеством полей и методов при поддержке новых асинхронных паттернов. Пожалуй, лучшим примером такого рода служит класс `Socket`. Ниже приведены некоторые составляющие класса `Socket` для базовой операции `Send`:

```
class Socket
{
  // Синхронный
  public int Send(byte[] buffer, int offset, int size, SocketFlags flags);

  // APM
  public IAsyncResult BeginSend(byte[] buffer, int offset, int size,
      SocketFlags flags, AsyncCallback callback, object state);
  public int EndSend(IAsyncResult result);

  // Специализированный, очень близок к APM
  public IAsyncResult BeginSend(byte[] buffer, int offset, int size,
      SocketFlags flags, out SocketError error,
```

```
        AsyncCallback callback, object state);
    public int EndSend(IAsyncResult result, out SocketError error);

    // Специализированный
    public bool SendAsync(SocketAsyncEventArgs e);

    // TAP (как метод расширения)
    public Task<int> SendAsync(ArraySegment<byte> buffer,
        SocketFlags socketFlags);

    // TAP (как метод расширения) с использованием более эффективных типов
    public ValueTask<int> SendAsync(ReadOnlyMemory<byte> buffer,
        SocketFlags socketFlags, CancellationToken cancellationToken =
            default);
}
```

К сожалению, большая часть документации упорядочена по алфавиту, а с множеством перегруженных версий разобраться в таком типе, как `Socket`, будет достаточно сложно. Надеюсь, приведенные в этом разделе рекомендации вам помогут.

Асинхронный паттерн на основе Task (TAP)

Асинхронный паттерн на основе Task (TAP) — современный паттерн асинхронных API, готовый для использования с `await`. Каждая асинхронная операция представляется одним методом, который возвращает ожидаемый тип. В данном случае «ожидаемым» является любой тип, который может потребляться `await`; обычно это `Task` или `Task<T>`, но им также может быть `ValueTask`, `ValueTask<T>`, тип, определенный фреймворком (например, `IAsyncAction` или `IAsyncOperation<T>`, используемый приложениями Universal Windows), или даже специализированный тип, определяемый библиотекой.

Методы TAP обычно снабжаются суффиксом `Async`. Впрочем, это всего лишь условное соглашение; не все методы TAP имеют суффикс `Async`. Он может отсутствовать, если разработчик API считает, что асинхронный контекст и так выражен достаточно ясно; например, у методов `Task.WhenAll` и `Task.WhenAny` нет суффикса `Async`. Кроме того, следует помнить, что суффикс `Async` может присутствовать у методов, к TAP не относящих-

ся (например, `WebClient.DownloadStringAsync` не является методом TAP). Обычно в таких случаях метод TAP имеет суффикс `TaskAsync` (например, `WebClient.DownloadStringTaskAsync` является методом TAP).

Методы, возвращающие асинхронные потоки, также следуют похожему паттерну с использованием суффикса `Async`. И хотя они не возвращают ожидаемые объекты, они возвращают ожидаемые потоки — типы, которые могут потребляться конструкцией `await foreach`.

Паттерн TAP можно узнать по следующим характеристикам:

1. Операция представляется одним методом.
2. Операция возвращает ожидаемый объект или ожидаемый поток.
3. Имя метода обычно завершается суффиксом `Async`.

Пример типа с TAP API:

```
class ExampleHttpClient
{
  public Task<string> GetStringAsync(Uri requestUri);
  // Синхронный эквивалент для сравнения
  public string GetString(Uri requestUri);
}
```

Потребление паттерна TAP осуществляется ключевым словом `await`; этой теме посвящены значительные части книги. Если вы как-то добрались до приложения, так и не научившись пользоваться `await`, вряд ли я смогу помочь вам на этой стадии. Попробуйте перечитать главы 1 и 2; возможно, они помогут освежить память.

Модель асинхронного программирования (APM)

Вероятно, следующим по популярности после TAP является паттерн модели асинхронного программирования, или APM (Asynchronous Programming Model). Это был первый паттерн, в котором асинхронные операции получили полноценные объектные представления. Характерный признак этого паттерна — объекты `IAsyncResult` в сочетании с парой методов, управляющих операцией; имя одного начинается с `Begin`, а имя другого — с `End`.

На разработку `IAsyncResult` сильно повлиял платформенный ввод/вывод с перекрытием. Паттерн APM позволяет потреблять код с синхронным или асинхронным поведением. Потребляющий код может выбирать из следующих вариантов:

- Блокироваться до завершения операции. Это делается вызовом метода `End`.
- Периодически опрашивать завершение операции, занимаясь чем-то другим.
- Предоставить делегата обратного вызова, который должен вызываться при завершении операции.

Во всех случаях потребляющий код должен в итоге вызвать метод `End`, чтобы получить результаты асинхронной операции. Если операция не завершена при вызове `End`, вызывающий поток блокируется до завершения операции.

Метод `Begin` получает в двух последних позициях параметр `AsyncCallback` и параметр `object` (обычно с именем `state`). Они используются потребляющим кодом для передачи делегата обратного вызова, который должен вызываться при завершении операции. Параметр `object` может содержать что угодно; это пережиток самых первых дней существования .NET, еще до появления лямбда-методов и даже анонимных методов. Он просто используется для предоставления контекста для параметра `AsyncCallback`.

Паттерн APM широко распространен в библиотеках Microsoft, но в экосистеме .NET встречается нечасто. Это объясняется тем, что реализации `IAsyncResult` для повторного использования были недоступны, а правильно реализовать этот интерфейс достаточно сложно. Кроме того, системы на базе APM трудно включать в композицию. Я видел несколько нестандартных реализаций `IAsyncResult`; все они были разновидностями реализации `IAsyncResult` общего назначения, разработанной Джеффри Рихтером (Jeffrey Richter) и опубликованной в его статье «Concurrent Affairs: Implementing the CLR Asynchronous Programming Model» из MSDN Magazine в марте 2007 года.

Паттерн APM можно узнать по следующим характеристикам:

1. Операция представляется парой методов; имя одного начинается с `Begin`, а имя другого — с `End`.

2. Метод `Begin` возвращает `IAsyncResult` и получает все обычные входные параметры наряду с дополнительным параметром `AsyncCallback` и дополнительным параметром `object`.
3. Метод `End` получает только `IAsyncResult` и возвращает результирующее значение, если оно есть.

Пример типа с APM API:

```
class MyHttpClient
{
  public IAsyncResult BeginGetString(Uri requestUri,
      AsyncCallback callback, object state);
  public string EndGetString(IAsyncResult asyncResult);

  // Синхронный эквивалент для сравнения
  public string GetString(Uri requestUri);
}
```

Потребление паттерна APM осуществляется преобразованием в TAP с помощью `Task.Factory.FromAsync`; см. рецепт 8.2 и документацию Microsoft (https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/interop-with-other-asynchronous-patterns-and-types#ApmToTap).

Возможны ситуации, в которых код *почти* следует паттерну APM; например, старые клиентские библиотеки `Microsoft.TeamFoundation` не включали параметр `object` в свои методы `Begin`. В таких случаях `Task.Factory.FromAsync` работать не будет, и у вас появляется выбор из двух вариантов. Менее эффективный вариант — вызов метода `Begin` и передача `IAsyncResult` методу `FromAsync`. Менее элегантный вариант — использование более гибкого типа `TaskCompletionSource<T>`; см. рецепт 8.3.

Асинхронный паттерн на основе событий (EAP)

Асинхронный паттерн на основе событий (EAP) определяет пару «метод/событие». Имя метода обычно завершается суффиксом `Async`, и он в конечном итоге приводит к выдаче события, имя которого завершается суффиксом `Completed`.

При работе с EAP существует ряд нюансов, из-за которых ваша задача оказывается сложнее, чем кажется на первый взгляд. Во-первых, вы должны помнить о добавлении обработчика в событие перед вызовом метода;

в противном случае возникнет ситуация гонки, когда событие может произойти до подписки, и тогда вы никогда не увидите его завершения. Во-вторых, компоненты, написанные на основе паттерна EAP, обычно в какой-то момент сохраняют текущий контекст `SynchronizationContext`, а затем выдают свое событие в этом контексте. Некоторые компоненты сохраняют `SynchronizationContext` в конструкторе, другие делают это в момент вызова метода и начала асинхронной операции.

Паттерн EAP можно узнать по следующим характеристикам:

1. Операция представляется событием и методом.
2. Имя события завершается суффиксом `Completed`.
3. Тип аргументов для события `Completed` должен быть производным от `AsyncCompletedEventArgs`.
4. Имя метода обычно завершается суффиксом `Async`.
5. Метод возвращает `void`.

Методы EAP, имена которых завершаются суффиксом `Async`, можно отличить от имен методов TAP с суффиксом `Async`, потому что методы EAP возвращают `void`, тогда как методы TAP возвращают ожидаемый тип.

Пример типа с EAP API:

```
class GetStringCompletedEventArgs : AsyncCompletedEventArgs
{
  public string Result { get; }
}

class MyHttpClient
{
  public void GetStringAsync(Uri requestUri);
  public event Action<object, GetStringCompletedEventArgs>
     GetStringCompleted;

  // Синхронный эквивалент для сравнения
  public string GetString(Uri requestUri);
}
```

Потребление паттерна EAP осуществляется преобразованием в TAP с помощью `TaskCompletionSource<T>`; см. рецепт 8.3 и документацию Microsoft

(https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/interop-with-other-asynchronous-patterns-and-types#EAP).

Стиль передачи продолжений (CPS)

Этот паттерн намного чаще встречается в других языках, особенно в JavaScript и TypeScript при использовании разработчиками Node.js. В этом паттерне каждая асинхронная операция получает делегата обратного вызова, который вызывается при завершении операции (успешном или с ошибкой). Разновидность этого паттерна использует двух делегатов обратного вызова, для успеха и для ошибки. Такая разновидность обратного вызова называется продолжением; продолжение передается в параметре, отсюда и название «стиль с передачей продолжений» (CPS, Continuation Passing Style»). Этот паттерн никогда не был широко распространен в мире .NET, но применялся в некоторых старых библиотеках с открытым кодом.

Паттерн EAP можно узнать по следующим характеристикам:

1. Операция представляется одним методом.
2. Метод получает дополнительный параметр с делегатом обратного вызова; делегат обратного вызова получает два аргумента, для ошибок и для результатов.
3. Также метод операции может получать два дополнительных параметра, содержащих делегатов обратного вызова; один делегат обратного вызова предназначен только для ошибок, а другой — только для результатов.
4. Делегатам обратного вызова обычно присваиваются имена `done` или `next`.

Вот пример типа API-интерфейса в стиле передачи продолжения:

```
class MyHttpClient
{
  public void GetString(Uri requestUri, Action<Exception, string> done);
  // Синхронный эквивалент для сравнения
  public string GetString(Uri requestUri);
}
```

Потребление паттерна CPS осуществляется преобразованием в TAP с помощью `TaskCompletionSource<T>` с передачей делегатов обратного вызова, которые просто завершают `TaskCompletionSource<T>`; см. рецепт 8.3.

Нестандартные асинхронные паттерны

Сильно специализированные типы иногда определяют собственные асинхронные паттерны. Самый известный пример такого рода — тип `Socket`, который определяет паттерн с передачей экземпляров `SocketAsyncEventArgs`, представляющих операцию. Причина для введения этого паттерна состояла в том, что `SocketAsyncEventArgs` может использоваться повторно, сокращая нарастающие затраты памяти в приложениях с интенсивными сетевыми операциями. Современные приложения используют `ValueTask<T>` с `ManualResetValueTaskSourceCore<T>` для того, чтобы добиться сходного выигрыша по быстродействию.

Нестандартные паттерны не обладают общими характеристиками, поэтому распознать их труднее всего. К счастью, нестандартные асинхронные паттерны встречаются редко.

Пример типа с нестандартным асинхронным API:

```
class MyHttpClient
{
  public void GetString(Uri requestUri,
      MyHttpClientAsynchronousOperation operation);

  // Синхронный эквивалент для сравнения
  public string GetString(Uri requestUri);
}
```

`TaskCompletionSource<T>` — единственный способ потребления нестандартных асинхронных паттернов; см. рецепт 8.3.

ISynchronizeInvoke

Все предыдущие паттерны предназначены для асинхронных операций, которые запускаются, после чего завершаются однократно. Некоторые компоненты следуют паттерну подписки: они представляют поток событий вместо одной операции, которая один раз запускается и один раз заверша-

ется. Хорошим примером модели подписки служит тип `FileSystemWatcher`. Чтобы отслеживать изменения в файловой системе, код-потребитель сначала подписывается на несколько событий, после чего задает свойству `EnableRaisingEvents` значение `true`. Если `EnableRaisingEvents` содержит `true`, могут инициироваться множественные события изменений в файловой системе.

Некоторые компоненты используют для своих событий паттерн `ISynchronizeInvoke`. Они предоставляют одно свойство `ISynchronizeInvoke`, а потребители задают этому свойству реализацию, которая позволяет компоненту планировать работу. Чаще всего оно используется для планирования работы в UI-потоке, чтобы события компонента выдавались в UI-потоке. По соглашениям, если `ISynchronizeInvoke` содержит `null`, то синхронизация событий не осуществляется и события могут выдаваться в фоновых потоках.

Паттерн `ISynchronizeInvoke` можно узнать по следующим характеристикам:

1. Свойство типа `ISynchronizeInvoke`.
2. Свойству обычно присваивается имя `SynchronizingObject`.

Пример типа, использующего паттерн `ISynchronizeInvoke`:

```
class MyHttpClient
{
  public ISynchronizeInvoke SynchronizingObject { get; set; }
  public void StartListening();
  public event Action<string> StringArrived;
}
```

Так как паттерн `ISynchronizeInvoke` подразумевает существование множественных событий в модели подписки, правильный способ потребления этих компонентов заключается в преобразовании событий в наблюдаемый поток — с использованием либо `FromEvent` (см. рецепт 6.1), либо `Observable.Create`.

Об авторе

Стивен Клири — опытный разработчик, прошел путь от ARM до Azure. Участвовал в написании открытого исходного кода библиотеки Boost C ++ и выпустил несколько собственных библиотек и утилит.

Об обложке

Животное на обложке книги — мусанг, или малайская пальмовая куница. Вид млекопитающих из семейства виверровых, обитающий в Южной и Юго-Восточной Азии. Мусанги — одиночки, ищут себе подобных лишь в брачный период и при выращивании потомства. Они ведут уединенный ночной образ жизни, но иногда обитают рядом с человеком и пробираются в жилища, поэтому считаются вредителями. Были недавно завезены в Японию и на Малые Зондские острова.

Длина тела мусанга составляет от 48 до 59 см, длина хвоста — от 44 до 54 см. Вес животного от 2,5 до 4 кг. Они имеют заостренные морды, мех может быть белым, серым и бурым, часто с темными пятнами, полосами или другими отметинами. На морде иногда бывают узоры, которые напоминают маску енота.

В отличие от других видов виверровых, на их хвостах нет колец. Анальные железы мусангов выделяют секрет с противным запахом при угрозе нападения. Этот секрет также используется в брачный период для привлечения партнера.

Пальмовые куницы играют важную роль в поддержании биоразнообразия тропических лесов, распространяя семена фруктов, которые они потребляют. Они также лакомятся соком пальмовых цветов, который после брожения становится сладким ликером; эта привычка принесла им прозвище «страннохвост».

Мусанги поедают спелые плоды кофейного дерева (кофейные вишни), которые проходят через их желудочно-кишечный тракт. Копи-лувак, один из самых дорогих видов кофе в мире, делается из кофейных зерен, извлеченных из испражнений мусанга. По утверждениям Американской ассоциации спешиалти кофе, «он просто имеет неприятный вкус». Компания, представляющая на Западе копи-лувак, теперь выступает против него из-за жестокого обращения с животными.

Многие из животных на обложках O'Reilly находятся под угрозой исчезновения; все они важны для мира. Изображение на обложке нарисовала Карен Монтгомери. За основу взята черно-белая гравюра из «Королевской естественной истории» (1893) Ричарда Лидеккера.

Стивен Клири

Конкурентность в C#. Асинхронное, параллельное и многопоточное программирование 2-е межд. изд.

Перевел на русский Е. Матвеев

Руководитель проекта	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Руденко*
Корректоры	*Н. Сидорова, Н. Сулейманова*
Обложка	*В. Мостипан*
Верстка	*Е. Неволайнен*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 03.2020. Наименование: книжная продукция.
Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020,
РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 05.03.20. Формат 70x100/16. Бумага офсетная. Усл. п. л. 21,930. Тираж 1000. Заказ