



Рикардо Террелл

Конкурентность и параллелизм на платформе .NET

Паттерны эффективного
проектирования



Concurrency in .NET

Modern patterns of concurrent and parallel programming

RICCARDO TERRELL



MANNING
SHELTER ISLAND

Рикардо Террелл

**Конкурентность
и параллелизм
на платформе
.NET**

Паттерны
эффективного
проектирования



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

2019

ББК 32.973.2-018-02

УДК 004.4

Т35

Террелл Рикардо

Т35 Конкурентность и параллелизм на платформе .NET. Паттерны эффективного проектирования. — СПб.: Питер, 2019. — 624 с.: ил. — (Серия «Для профессионалов»). ISBN 978-5-4461-1072-8

Рикардо Террелл научит вас писать идеальный код, с которым любые приложения будут просто лепить. Книга содержит примеры на языках C# и F#, описывает паттерны проектирования конкурентных и параллельных программ как в теории, так и на практике.

Вы начнете с теоретических основ параллелизма, после чего перейдете к примерам и проверенным решениям, помогающим создавать и оптимизировать код для современных многопроцессорных систем.

В этой книге автор раскрыл важнейшие конкурентные абстракции, реализацию потоковой обработки событий в реальном времени и наилучшие конкурентные паттерны и практики, применимые на любых платформах.

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

ББК 32.973.2-018-02

УДК 004.4

Права на издание получены по соглашению с Apress. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1617292996 англ.

ISBN 978-5-4461-1072-8

© 2018 by Manning Publications Co. All rights reserved

© Перевод на русский язык ООО Издательство «Питер», 2019

© Издание на русском языке, оформление ООО Издательство «Питер», 2019

© Серия «Для профессионалов», 2019

Краткое содержание

Предисловие.....	18
Благодарности	22
Об этой книге	24
Об авторе	28
Об иллюстрации на обложке	29

Часть I. Преимущества функционального программирования в применении к конкурентным программам

Глава 1. Основы функциональной конкурентности	33
Глава 2. Технологии функционального программирования для конкурентных систем...	66
Глава 3. Функциональные структуры данных и неизменяемость	99

Часть II. Конкурентная программа: разные части, разные подходы

Глава 4. Основы обработки больших данных: распараллеливание данных, часть 1 ...	145
Глава 5. PLINQ и MapReduce: распараллеливание данных, часть 2	169
Глава 6. Потоки событий реального времени: функциональное реактивное программирование	204
Глава 7. Функциональный параллелизм на основе задач.....	243

6 Краткое содержание

Глава 8. Асинхронность задач — путь к победе.....	279
Глава 9. Асинхронное функциональное программирование на F#	318
Глава 10. Функциональные комбинаторы для быстрого конкурентного программирования	350
Глава 11. Реактивное программирование с использованием агентов.....	411
Глава 12. Параллельный рабочий процесс и агентное программирование с помощью TPL Dataflow	453

Часть III. Современные шаблоны конкурентного программирования

Глава 13. Рецепты и шаблоны для успешного конкурентного программирования.....	489
Глава 14. Построение масштабируемого мобильного приложения методом конкурентного функционального программирования	547

Приложения

Приложение А. Функциональное программирование	588
Приложение Б. Обзор F#	603
Приложение В. Совместимость асинхронного рабочего процесса F# и задач .NET ...	620

Оглавление

Предисловие.....	18
Благодарности.....	22
Об этой книге	24
Кому следует прочесть эту книгу.....	24
Структура издания: дорожная карта.....	25
О коде	27
От издательства	27
Об авторе.....	28
Об иллюстрации на обложке	29

Часть I. Преимущества функционального программирования в применении к конкурентным программам

Глава 1. Основы функциональной конкурентности	33
1.1. Что вы узнаете из этой книги	35
1.2. Начнем с терминологии.....	36
1.2.1. При последовательном программировании задачи выполняются одна за другой.....	37
1.2.2. При конкурентном программировании выполняется несколько задач в одно и то же время	38
1.2.3. При параллельном программировании выполняется несколько задач одновременно	39
1.2.4. При многозадачности выполняется несколько задач одновременно.....	41
1.2.5. Многопоточность как средство повышения производительности.....	42
1.3. Зачем нужна конкурентность	44

8 Оглавление

1.4.	Ловушки параллельного программирования.....	47
1.4.1.	Риски конкурентности.....	48
1.4.2.	Эволюция разделяемых состояний.....	51
1.4.3.	Простой пример из практики: параллельная быстрая сортировка	52
1.4.4.	Бенчмаркинг в F#	56
1.5.	Почему для конкурентности выбирают функциональное программирование	57
1.6.	Область применения функциональной парадигмы.....	61
1.7.	Зачем использовать F# и C# для функционального конкурентного программирования	62
	Резюме	65
	Глава 2. Технологии функционального программирования для конкурентных систем...	66
2.1.	Использование компоновки функций для решения сложных задач.....	67
2.1.1.	Функциональная компоновка в C#	68
2.1.2.	Функциональная компоновка в F#	70
2.2.	Использование замыканий для упрощения функционального мышления.....	71
2.2.1.	Захваченные переменные в замыканиях с лямбда-выражениями	72
2.2.2.	Замыкания в многопоточной среде	75
2.3.	Технология мемоизации и кэширования для ускорения программы	77
2.4.	Применение мемоизации для создания быстрого веб-робота	81
2.5.	«Ленивая» мемоизация для повышения производительности.....	85
2.6.	Эффективная конкурентная упреждающая обработка для уменьшения издержек на затратные вычисления	87
2.6.1.	Предварительные вычисления с естественной поддержкой функционального программирования	89
2.6.2.	Пускай победит быстрейший!	91
2.7.	Лень — это хорошо	92
2.7.1.	Использование строгих языков программирования для лучшего понимания конкурентного поведения	92
2.7.2.	«Ленивое» кэширование и потокобезопасный шаблон «Одиночка»	94
2.7.3.	Поддержка «ленивых» вычислений в F#.....	96
2.7.4.	Мощное сочетание Lazy и Task.....	96
	Резюме	98
	Глава 3. Функциональные структуры данных и неизменяемость.....	99
3.1.	Практический пример: охота на потоконебезопасный объект	100
3.1.1.	Неизменяемые коллекции .NET: надежное решение.....	104
3.1.2.	Конкурентные коллекции .NET: более быстрое решение	109
3.1.3.	Шаблон агента передачи сообщений: более быстрое и верное решение.....	112

3.2.	Безопасное совместное использование потоками функциональных структур данных.....	114
3.3.	Наконец-то неизменяемость!.....	115
3.3.1.	Функциональная структура данных для обеспечения их параллелизма.....	119
3.3.2.	Влияние неизменяемости на производительность	119
3.3.3.	Неизменяемость в C#	120
3.3.4.	Неизменяемость в F#.....	123
3.3.5.	Функциональные списки: объединение ячеек в цепочки	124
3.3.6.	Построение персистентной структуры данных: неизменяемое бинарное дерево	131
3.4.	Рекурсивные функции: естественный способ итерирования	134
3.4.1.	Хвост правильной рекурсивной функции: оптимизация хвостового вызова	135
3.4.2.	Оптимизация рекурсивной функции в стиле передачи продолжений	137
	Резюме	142

Часть II. Конкурентная программа: разные части, разные подходы

Глава 4.	Основы обработки больших данных: распараллеливание данных, часть 1	145
4.1.	Что такое распараллеливание данных.....	146
4.1.1.	Распараллеливание данных и задач.....	147
4.1.2.	Концепция «естественной параллельности»	148
4.1.3.	Поддержка распараллеливания данных в .NET	149
4.2.	Шаблон Fork/Join: параллельный алгоритм Мандельброта	150
4.2.1.	Когда узким местом является сборка мусора: структуры и объекты класса	156
4.2.2.	Оборотная сторона параллельных циклов	159
4.3.	Измерение производительности	160
4.3.1.	Определение предела повышения эффективности по закону Амдала.....	161
4.3.2.	Закон Густафсона: еще один шаг вперед в измерении повышения производительности	162
4.3.3.	Ограничения параллельных циклов: сумма простых чисел	162
4.3.4.	Что может пойти не так в простом цикле.....	164
4.3.5.	Модель декларативного параллельного программирования	166
	Резюме	168

Глава 5. PLINQ и MapReduce: распараллеливание данных, часть 2	169
5.1. Краткое введение в PLINQ.....	170
5.1.1. Почему язык PLINQ более функциональный.....	171
5.1.2. PLINQ и чистые функции: параллельный счетчик слов	172
5.1.3. Исключение побочных эффектов посредством чистых функций.....	174
5.1.4. Изоляция и контроль побочных эффектов: рефакторинг параллельного счетчика слов	176
5.2. Агрегирование и сокращение данных в параллельных программах	177
5.2.1. Усечение: одно из многих преимуществ свертки.....	180
5.2.2. Свертки в PLINQ: функции агрегирования	182
5.2.3. Реализация параллельной функции Reduce на PLINQ	188
5.2.4. Параллельное списковое включение в F#: PSeq	191
5.2.5. Параллельные массивы в F#	192
5.3. Параллельный шаблон MapReduce	193
5.3.1. Функции Map и Reduce.....	195
5.3.2. Использование MapReduce совместно с галереей пакетов NuGet	196
Резюме	203
Глава 6. Потоки событий реального времени: функциональное реактивное программирование	204
6.1. Реактивное программирование: обработка больших событий.....	206
6.2. Инструментарий .NET для реактивного программирования.....	209
6.2.1. Улучшенное решение: комбинаторы событий	210
6.2.2. Совместимость .NET с комбинаторами F#	211
6.3. Реактивное программирование в .NET: реактивные расширения (Rx)	214
6.3.1. От LINQ/PLINQ к Rx	217
6.3.2. IObservable и IEnumerable: дуальные интерфейсы	218
6.3.3. Реактивные расширения в действии	219
6.3.4. Потоковая передача в реальном времени с помощью Rx	220
6.3.5. От событий к наблюдаемым объектам F#	221
6.4. Укрощение потока событий: анализ эмоций в Twitter посредством Rx-программирования	222
6.5. Шаблон «издатель — подписчик» в Rx	232
6.5.1. Использование типа Subject для создания мощного концентратора в шаблоне «издатель — подписчик»	233
6.5.2. Rx и конкурентность	234
6.5.3. Реализация на Rx многоразового шаблона «издатель — подписчик»	235
6.5.4. Анализ эмоций твитов с помощью Rx-класса Pub-Sub.....	237
6.5.5. Наблюдатели в действии	240
6.5.6. Удобное выражение объектов в F#	240
Резюме	242

Глава 7. Функциональный параллелизм на основе задач.....	243
7.1. Краткое введение в параллелизм	244
7.1.1. Зачем нужны параллелизм задач и функциональное программирование.....	245
7.1.2. Поддержка параллелизма задач в .NET.....	246
7.2. Библиотека параллельных задач в .NET	248
7.2.1. Выполнение параллельных операций с помощью Parallel.Invoke из библиотеки TPL	250
7.3. Проблема void в C#	253
7.4. Стиль продолжений: функциональный поток управления.....	256
7.4.1. Зачем нужен CPS	256
7.4.2. Ожидание завершения задачи: модель продолжения.....	258
7.5. Стратегии компоновки операций для выполнения задач	263
7.5.1. Использование математических шаблонов для оптимальной компоновки.....	265
7.5.2. Рекомендации по использованию задач	271
7.6. Параллельный функциональный шаблон конвейера.....	271
Резюме	278
Глава 8. Асинхронность задач — путь к победе.....	279
8.1. Модель асинхронного программирования (APM)	280
8.1.1. В чем ценность асинхронного программирования	281
8.1.2. Асинхронное программирование и масштабируемость	284
8.1.3. Операции с ограничениями процессора и ввода-вывода	285
8.2. Неограниченный параллелизм при асинхронном программировании.....	286
8.3. Поддержка асинхронности в .NET.....	287
8.3.1. Асинхронное программирование нарушает структуру кода	290
8.3.2. Асинхронное программирование на основе событий	291
8.4. Асинхронное программирование на основе задач в C#	291
8.4.1. Анонимные асинхронные лямбда-функции.....	295
8.4.2. Монадический контейнер Task<T>.....	296
8.5. Асинхронное программирование на основе задач: практический пример	299
8.5.1. Отмена асинхронных операций.....	304
8.5.2. Асинхронная компоновка на основе задач с монадическим оператором Bind	307
8.5.3. Отсрочка асинхронного вычисления, обеспечивающая компоновку	309
8.5.4. Повторная попытка в случае, если что-то пошло не так	310
8.5.5. Обработка ошибок в асинхронных операциях	312
8.5.6. Асинхронная параллельная обработка изменений фондового рынка	314
8.5.7. Асинхронная параллельная обработка данных фондового рынка после завершения выполнения задач	315
Резюме	317

Глава 9. Асинхронное функциональное программирование на F#	318
9.1. Аспекты асинхронного функционального программирования	319
9.2. Что такое асинхронный рабочий процесс F#	319
9.2.1. Стиль прохождения продолжений в вычислительных выражениях.....	320
9.2.2. Асинхронный рабочий процесс в действии: параллельные операции сохранения данных из Azure Blob	322
9.3. Асинхронные вычислительные выражения	328
9.3.1. Различия между вычислительными выражениями и монадами.....	330
9.3.2. AsyncRetry: построение собственных вычислительных выражений.....	331
9.3.3. Расширение асинхронного рабочего процесса	334
9.3.4. Отображение асинхронных операций: функтор Async.map.....	335
9.3.5. Распараллеливание асинхронных рабочих процессов: Async.Parallel	337
9.3.6. Поддержка отмены асинхронного рабочего процесса	343
9.3.7. Управление параллельными асинхронными операциями.....	345
Резюме	349
Глава 10. Функциональные комбинаторы для быстрого конкурентного программирования	350
10.1. Поток выполнения не всегда проходит по «счастливому пути»: обработка ошибок	351
10.2. Комбинаторы ошибок: Retry, Otherwise и Task.Catch в C#	355
10.2.1. Обработка ошибок в функциональном программировании: исключения при управлении потоком выполнения.....	358
10.2.2. Обработка ошибок с помощью Task<Option<T>> в C#	360
10.2.3. Тип AsyncOption в F#: сочетание Async и Option	361
10.2.4. Функциональная асинхронная обработка ошибок, свойственная F#	362
10.2.5. Сохранение семантики исключений с помощью типа Result	364
10.3. Обработка исключений при асинхронных операциях.....	368
10.3.1. Моделирование обработки ошибок в F# с помощью Async и Result	373
10.3.2. Расширение типа F#AsyncResult посредством операторов монадического связывания	374
10.4. Абстрагирование операций с функциональными комбинаторами.....	379
10.5. Коротко о функциональных комбинаторах	380
10.5.1. Встроенные асинхронные комбинаторы TPL	381
10.5.2. Использование комбинатора Task.WhenAny для избыточности и чередования	382
10.5.3. Использование комбинатора Task.WhenAll в асинхронном цикле for-each	384
10.5.4. Обзор математических шаблонов: что мы уже знаем?	385

10.6.	Окончательный вариант параллельной компоновки applicативного функтора.....	389
10.6.1.	Расширение асинхронного рабочего процесса F# с помощью операторов applicативных функторов	396
10.6.2.	Семантика applicативных функторов и инфиксных операторов в F#	398
10.6.3.	Использование applicативных функторов в гетерогенных параллельных вычислениях	399
10.6.4.	Компоновка и выполнение гетерогенных параллельных вычислений	401
10.6.5.	Управление потоком с помощью условных асинхронных комбинаторов	404
10.6.6.	Как работают асинхронные комбинаторы	408
	Резюме	410
Глава 11.	Реактивное программирование с использованием агентов.....	411
11.1.	Что такое реактивное программирование и чем оно полезно	413
11.2.	Программная модель асинхронной передачи сообщений.....	415
11.2.1.	Передача сообщений и неизменяемость	417
11.2.2.	Естественная изоляция	417
11.3.	Что такое агент	418
11.3.1.	Компоненты агента	419
11.3.2.	Что может делать агент	420
11.3.3.	Подход без разделения ресурсов для конкурентного программирования без блокировок	420
11.3.4.	Как функционирует агентное программирование	422
11.3.5.	Агент является объектно-ориентированным	422
11.4.	Агенты в F#: MailboxProcessor	423
11.5.	Как избежать узких мест при обращении к базе данных с помощью F#-типа MailboxProcessor	426
11.5.1.	Тип сообщения MailboxProcessor: размеченные объединения	429
11.5.2.	Двусторонний обмен данными посредством MailboxProcessor	430
11.5.3.	Использование AgentSQL из C#	431
11.5.4.	Распараллеливание рабочего процесса и координация групп агентов	433
11.5.5.	Как обрабатывать ошибки на F# с помощью MailboxProcessor	435
11.5.6.	Остановка агентов MailboxProcessor — CancellationToken	436
11.5.7.	Распределение работы с помощью MailboxProcessor	437
11.5.8.	Операции кэширования в агентном программировании	439
11.5.9.	Получение результатов от MailboxProcessor	443
11.5.10.	Использование пула потоков для сообщения о событиях MailboxProcessor	446
11.6.	F# MailboxProcessor: 10 000 агентов для Game of Life	446
	Резюме	452

Глава 12. Параллельный рабочий процесс и агентное программирование с помощью TPL Dataflow	453
12.1. Преимущества TPL Dataflow.....	454
12.2. Блоки TPL Dataflow: созданы для компоновки	455
12.2.1. Использование BufferBlock<TInput> в качестве буфера FIFO.....	457
12.2.2. Преобразование данных с помощью TransformBlock<TInput, TOutput>	458
12.2.3. Законченный пример с ActionBlock<TInput>	459
12.2.4. Связывание блоков в потоке данных	461
12.3. Реализация сложных шаблонов «поставщик — потребитель» с помощью TDF	461
12.3.1. Реализация шаблона «несколько поставщиков — один потребитель» с помощью TDF.....	461
12.3.2. Шаблон «один поставщик — несколько потребителей»	463
12.4. Использование агентной модели в C# с помощью TPL Dataflow.....	464
12.4.1. Свертка состояний и сообщений агента: Aggregate	468
12.4.2. Агентное взаимодействие: параллельный счетчик слов	468
12.5. Параллельный рабочий процесс для сжатия и шифрования больших потоков.....	474
12.5.1. Контекст: проблема обработки большого потока данных	474
12.5.2. Сохранение порядка в потоке сообщений	480
12.5.3. Связывание, распространение и завершение	481
12.5.4. Правила построения рабочего процесса TDF	483
12.5.5. Реактивные расширения генерации сетки (Rx) и TDF.....	484
Резюме	486

Часть III. Современные шаблоны конкурентного программирования

Глава 13. Рецепты и шаблоны для успешного конкурентного программирования.....	489
13.1. Освобождение памяти, занимаемой объектами, для сокращения потребления памяти.....	490
13.2. Нестандартный параллельный оператор Fork/Join	494
13.3. Распараллеливание задач с зависимостями: разработка кода для оптимизации производительности	497
13.4. Шлюз для координации конкурентных операций ввода-вывода с разделяемыми ресурсами: одна операция записи, несколько операций чтения	502
13.5. Потокобезопасный генератор случайных чисел.....	508
13.6. Полиморфный агрегатор событий	510

13.7. Нестандартный Rx-планировщик для управления степенью параллелизма.....	513
13.8. Конкурентное реактивное масштабируемое приложение «клиент-сервер».....	516
13.9. Многоразовый специальный высокопроизводительный параллельный оператор фильтрации-отображения	527
13.10. Неблокирующая синхронная модель передачи сообщений	532
13.11. Координирование конкурентных заданий с использованием агентной модели программирования	537
13.12. Компоновка монадических функций	542
Резюме	546

Глава 14. Построение масштабируемого мобильного приложения методом
конкурентного функционального программирования 547

14.1. Практическое применение функционального программирования для серверной части приложения	548
14.2. Как разработать высокопроизводительное приложение	550
14.2.1. Ноу-хау: ACD	551
14.2.2. Другой асинхронный шаблон: постановка в очередь для отложенного выполнения	552
14.3. Правильный выбор конкурентной модели программирования	553
14.4. Торговля акциями в реальном времени: архитектура высокого уровня для фондового рынка	557
14.5. Главные элементы приложения для фондового рынка	562
14.6. Пишем код приложения для торговли на фондовом рынке.....	563
Резюме	586

Приложения

Приложение А. Функциональное программирование	588
Что такое функциональное программирование	588
Преимущества функционального программирования.....	589
Основные принципы функционального программирования	590
Противостояние программных парадигм: от императивного к объектно-ориентированному и функциональному программированию	590
Применение функций высшего порядка для увеличения абстракции.....	592
Применение функций высшего порядка и лямбда-выражений для создания многократно используемого кода	593
Лямбда-выражения и анонимные функции.....	593
Каррирование	595
Частично примененные функции.....	599
Преимущества частичного применения и каррирования функций в C#	601

Приложение Б. Обзор F#	603
let-привязки	603
Сигнатуры функций в F#	604
Создание изменяемых типов: mutable и ref	604
Функции как типы первого класса	605
Компоновка: операторы конвейера и компоновки	605
Делегаты	606
Комментарии	606
Оператор open	606
Основные типы данных	607
Специальное определение строки	607
Кортежи	607
Записи	608
Размеченные объединения	609
Сопоставление с образцом	610
Активные образцы	611
Коллекции	612
Массивы	612
Последовательности (seq)	613
Списки	613
Множества	614
Словари	614
Циклы	614
Классы и наследование	615
Абстрактные классы и наследование	615
Интерфейсы	616
Объектные выражения	617
Приведение типов	617
Единицы измерения	618
Краткая справка по API модуля событий	618
Приложение В. Совместимость асинхронного рабочего процесса F# и задач .NET ...	620

Я посвящаю эту книгу своей жене Бриони — замечательной, всегда готовой прийти на помощь. Твоя поддержка, любовь, забота и постоянное воодушевление, пока я писал книгу, — только они позволили мне претворить настоящий проект в жизнь. Я люблю тебя и преклоняюсь пред тобой за все твои усилия и терпение, пока я был занят данной работой. Спасибо, что всегда верила в меня.

Я также посвящаю эту книгу Буггине и Стеллине, двум моим верным монсам, которые были постоянно со мной при ее написании, беззаветно и с энтузиазмом поддерживая меня. Вы лучшие друзья человека и самые горячие мои натуралистические фанаты автора.

Предисловие

Эту книгу, «Конкурентность в .NET», вы читаете, возможно, потому, что хотите научиться создавать невероятно быстрые приложения или узнать, как резко повысить производительность уже существующего приложения. Вас заботит производительность, потому что вы посвятили себя созданию более быстрых программ и потому что вас восхищает, когда всего несколько изменений в коде делают приложение более быстрым и отзывчивым. Параллельное программирование предоставляет бесконечные возможности разработчикам, которые влюблены в свое дело и хотят внедрять новые технологии. В отношении производительности преимущества применения параллелизма в программировании невозможно переоценить. Но при использовании императивного и объектно-ориентированного стиля программирования (ООП) написание конкурентного кода может оказаться запутанным и сложным делом. Именно поэтому конкурентное программирование не стало распространенной практикой и крупным ведущим программистам приходилось искать другие варианты.

В колледже я прослушал курс функционального программирования. В то время я изучал Haskell и, несмотря на большие учебные нагрузки, наслаждался каждым занятием. Помню, как, увидев первые примеры, был поражен элегантностью и простотой решений. Пятнадцать лет спустя, когда я стал искать варианты улучшения своих программ с использованием конкурентности, я снова вспомнил те уроки. На этот раз я в полной мере оценил, насколько мощным и полезным было бы применение функционального программирования в моих текущих задачах. У функционального стиля программирования есть несколько преимуществ; каждое из них мы обсудим в данной книге.

Знания, полученные во время учебы, пригодились мне в профессиональной деятельности, когда пришлось создавать программную систему для сферы здравоохранения. Этот проект требовал разработки приложения для анализа медицинских

рентгеновских снимков. Обработка подобных изображений состояла из нескольких этапов, таких как уменьшение шума, обработка по алгоритму Гаусса, интерполяция и фильтрация, в результате чего черно-белые изображения становились цветными. Приложение было разработано на Java и сначала функционировало так, как и ожидалось. Но однажды, как это часто бывает, нагрузку увеличили, и начались проблемы. В программе не было никаких неполадок или ошибок, но с увеличением количества анализируемых изображений она стала работать медленнее.

Естественно, первым предложением для решения данной проблемы было приобрести более мощный сервер. На тот момент это было правильное решение, но сегодня, купив новую машину с целью получить более высокую вычислительную скорость процессора, вы будете разочарованы. Дело в том, что у современного процессора несколько ядер и скорость каждого из них не выше, чем у ядра, приобретенного в 2007 г. Лучшим и более надежным вариантом (вместо того чтобы покупать новый сервер или компьютер) было использовать параллелизм, чтобы реализовать преимущества многоядерного оборудования, задействовав все его ресурсы, что в итоге ускорило бы обработку изображений.

Теоретически это была простая задача, но на практике не все было так тривиально. Мне пришлось научиться применять потоки и устанавливать блокировки, и, к сожалению, я на собственном опыте узнал, что такое взаимная блокировка.

Эта взаимная блокировка побудила меня внести существенные изменения в код приложения. Их было так много, что я допустил ошибки, даже не связанные с первоначальной целью изменений. Я был расстроен: код стал неподдерживаемым и нестабильным и в процессе постоянно возникали новые ошибки. Мне пришлось временно отложить решение исходной проблемы и посмотреть на задачу с другой точки зрения. Должен был существовать способ получить.

Инструменты, которые мы используем, оказывают глубокое (и коварное!) влияние на наши мыслительные привычки и, следовательно, на наши интеллектуальные способности.

Эдсгер Дейкстра (Edsger Dijkstra)

Проведя несколько дней в поисках выхода из этого многопоточного кошмара, я нашел ответ. Все, что я исследовал и читал, указывало на функциональную парадигму. Принципы, изученные мною в колледже много лет назад, стали тем механизмом, который позволил мне продвинуться вперед. Я переписал ядро приложения для обработки изображений с учетом параллельной работы, используя функциональный язык программирования. Сначала переход от императивного к функциональному стилю вызывал трудности. Я забыл почти все, что выучил в колледже, и не горжусь тем, что написал тогда на функциональном языке код, выглядевший очень объектно-ориентированным; но в целом это было успешное решение. После компиляции новая программа заработала без ошибок, резко повысилась производительность, аппаратные ресурсы были использованы полностью. Но самым неожиданным было то, что функциональное программирование привело

к впечатляющему сокращению числа строк кода: их стало почти на 50 % меньше, чем в исходной реализации на объектно-ориентированном языке.

Этот опыт заставил меня пересмотреть отношение к ООП как к решению всех проблем программирования. Я понял, что перспективы данной модели программирования и ее подхода к решению проблем ограничены. Мое путешествие в мир функционального программирования началось с потребности в хорошей модели конкурентного программирования.

С тех пор я проявляю большой интерес к применению функционального программирования для многопоточности и конкурентности. Там, где другие усматривали сложную проблему и источник трудностей, я видел решение в функциональном программировании как в мощном инструменте, позволяющем ускорить работу, используя имеющееся оборудование. Я пришел к пониманию того, как данная дисциплина приводит к последовательной, удобной и красивой форме написания конкурентных программ.

Впервые идея создания этой книги возникла у меня в июле 2010 г., после того как Microsoft представила F# как часть Visual Studio 2010. Уже в то время было ясно, что все больше основных языков программирования — включая C#, C++, Java и Python — поддерживают функциональную парадигму. В 2007 г. в C# 3.0 появились функции первого класса, а также новые конструкции, такие как лямбда-выражения и вывод типов, что позволило программистам использовать концепции функционального программирования. Вскоре после этого появился Language Integrated Query (LINQ), допускающий декларативное программирование.

Платформа .NET особенно сильно погрузилась в мир функционального программирования. После F# у Microsoft появились полнофункциональные языки, поддерживающие как объектно-ориентированную, так и функциональную парадигму. Кроме того, объектно-ориентированные языки, такие как C#, становятся все более гибридными и устраниют разрыв между различными парадигмами, что позволяет применять оба стиля программирования.

Более того, мы вступаем в эпоху многоядерности, когда мощность процессоров измеряется не числом тактов в секунду, а количеством доступных ядер. При такой тенденции однопоточные приложения не позволят повысить скорость на многоядерных системах, если не интегрировать в них параллелизм и не использовать алгоритмы для распределения работы между несколькими ядрами.

Мне стало ясно, что многопоточность востребована, и я загорелся идеей поделиться с вами таким подходом к программированию. В этой книге показано, как, применяя возможности параллельного программирования и функциональной парадигмы, писать легко читаемый, лучше разбитый на модули, поддерживаемый код на языках C# и F#. Благодаря данным технологиям ваш код будет работать с максимальной скоростью при меньшем количестве строк, что приведет к повышению производительности и отказоустойчивости программ.

Настали великолепные времена — можно начинать разработку многопоточного кода. Сегодня компании — разработчики программного обеспечения создают больше, чем когда-либо, инструментов и средств, позволяющих выбирать стиль программирования без каких-либо компромиссов. Первоначальные проблемы, связанные

с освоением параллельного программирования, быстро уйдут, а награда за вашу настойчивость будет неизмеримой. Независимо от области знаний, будь вы разработчик серверной или клиентской части приложений, создатель облачных продуктов или видеоигр, вам все равно придется применять параллелизм для повышения производительности и создания масштабируемых приложений.

Эта книга основана на моем опыте функционального программирования для написания конкурентных программ в .NET с использованием C# и F#. Я уверен, что функциональное программирование фактически становится способом написания конкурентного кода для координации асинхронных и параллельных программ в .NET. Я также считаю, что данная книга даст вам все необходимое для освоения этого захватывающего мира многоядерных компьютеров.

Благодарности

Написать книгу — всегда подвиг. Сделать это на неродном языке бесконечно сложнее и страшнее. Что касается меня, то мне бы такое и не приснилось, не будь у меня группы поддержки размером с небольшой город. Я хотел бы поблагодарить всех, кто поддерживал меня и участвовал в создании данной книги.

Мои приключения с F# начались в 2013 г., когда я посетил FastTrack по F# в Нью-Йорке. Там я встретил Томаса Петричека (Tomas Petricek), который вдохновил меня нырнуть с головой в мир F#. Томас пригласил меня в сообщество и с тех пор является моим наставником и поверенным моих тайн.

Я неизмеримо благодарен фантастическому персоналу Manning Publications. Пятнадцать месяцев назад я начал тяжелый труд над данной книгой с редактором — консультантом по аудитории Дэном Магарри (Dan Maharry), затем продолжил его с Марииной Майклз (Marina Michaels). Оба они были моими терпеливыми и мудрыми руководителями в этой удивительной работе.

Спасибо многим рецензентам, особенно научному редактору-консультанту Майклу Лунду (Michael Lund) и техническому корректору Вайорель Моисей (Viorel Moisei). Ваш критический анализ позволил мне убедиться: я правильно передал на бумаге все, что было у меня в голове, хотя большая часть из этого рисковала потеряться при «переводе». Спасибо также тем, кто участвовал в программе Manning MEAP и оказал поддержку в качестве рецензентов: Энди Киршу (Andy Kirsch), Антону Херцогу (Anton Herzog), Крису Бойярду (Chris Bolyard), Крейгу Фуллертону (Craig Fullerton), Гарету ван дер Бергу (Gareth van der Berg), Джереми Ланге (Jeremy Lange), Джиму Вельху (Jim Velch), Джоэлу Котарски (Joel Kotarski), Кевину Орру (Kevin Orr), Люку Бирлу (Luke Bearl), Павлу Климчику (Pawel Klimczyk), Рикардо Пересу (Ricardo Peres), Рохиту Шарме (Rohit Sharma), Стефано Дриосси (Stefano Driussi) и Субхазису Гошу (Subhasis Ghosh).

Я получил огромную поддержку от членов сообщества F#, сплотившихся вокруг меня на этом пути, особенно от Сергея Тихона (Sergey Tihon), который оказался прекрасным слушателем.

И спасибо моей семье и друзьям, которые подбадривали меня и терпеливо ждали, когда я снова вернусь в мир общих уик-эндов, обедов и отдыха.

Прежде всего я бы хотел выразить признательность моей жене, которая поддерживала все мои усилия и никогда не позволяла мне уходить от проблем.

Я также признателен моим преданным и верным мопсам, Бутгине и Стеллине, которые всегда были на моей стороне — или на моих коленях, — когда я писал эту книгу глубокой ночью. А во время наших долгих вечерних прогулок я имел возможность освежать голову и находить лучшие идеи для книги.

Об этой книге

Книга «Конкурентность и параллелизм на платформе .NET» дает представление о рекомендуемых методах создания конкурентных и масштабируемых программ в .NET, освещая преимущества функциональной парадигмы и предоставляя соответствующие инструменты и принципы, позволяющие легко и правильно поддерживать конкурентность. В итоге, вооружившись новыми навыками, вы получите знания, необходимые для того, чтобы стать экспертом в предоставлении успешных высокопроизводительных решений.

Кому следует прочесть эту книгу

Если вы пишете многопоточный код на .NET, то эта книга может вам помочь. Если вы заинтересованы в использовании функциональной парадигмы для упрощения конкурентного программирования и максимального повышения производительности приложений, то данная книга станет для вас важным руководством. Она принесет пользу любым разработчикам на .NET, желающим писать конкурентные, реактивные и асинхронные приложения, которые масштабируются и автоматически адаптируются к имеющимся аппаратным ресурсам везде, где бы ни работали такие программы.

Данная книга также подойдет тем разработчикам, которых интересует применение функционального программирования для реализации конкурентных методов. Предварительные знания или опыт работы с функциональной парадигмой для этого не требуются, а основные понятия функциональной парадигмы рассматриваются в приложении А.

В примерах кода используются языки C# и F#. Читатели, знакомые с C#, сразу почувствуют себя комфортно. Знакомство с языком F# не является обязательным,

его основы рассмотрены в приложении Б. Опыт и знания в области функционального программирования не требуются: в книгу включены все необходимые понятия.

Предполагается хорошее знание .NET. Вы должны иметь некоторый опыт работы с коллекциями .NET и обладать знаниями .NET Framework как минимум версии .NET 3.5 (LINQ-делегаты `Action<>` и `Func<>`). Наконец, эта книга подходит для любой платформы, поддерживающей .NET (включая .NET Core).

Структура издания: дорожная карта

Четырнадцать глав этой книги разделены на три части. В части I представлены функциональные концепции конкурентного программирования и описаны навыки, необходимые для понимания функциональных аспектов написания многопоточных программ.

- ❑ В главе 1 описаны основные понятия и цели конкурентного программирования, а также причины применения функционального программирования для написания многопоточных приложений.
- ❑ В главе 2 исследуется ряд технологий функционального программирования для повышения производительности многопоточных приложений. Цель этой главы — предоставить читателю концепции, используемые в остальной книге, и познакомить с мощными идеями, возникающими из функциональной парадигмы.
- ❑ В главе 3 дается обзор функциональной концепции неизменяемости. Здесь объясняется, как неизменяемость применяется для написания предсказуемых и корректных конкурентных программ и для реализации функциональных структур данных, которые являются потокобезопасными по своей сути.

В части II углубленно рассматриваются различные модели конкурентного программирования в функциональной парадигме. Мы изучим такие темы, как библиотека Task Parallel Library (TPL), и реализуем параллельные шаблоны, такие как Fork/Join, «разделяй и властвуй» и MapReduce. В этой части также обсуждаются декларативная компоновка, высокоуровневые абстракции в асинхронных операциях, агентное программирование и семантика передачи сообщений.

- ❑ В главе 4 изложены основы параллельной обработки большого количества данных, включая такие шаблоны, как Fork/Join.
- ❑ В главе 5 представлены более сложные методы параллельной обработки больших объемов информации, такие как параллельное агрегирование, сокращение данных и реализация параллельного шаблона MapReduce.
- ❑ В главе 6 представлена подробная информация о функциональных методах обработки потоков событий (данных) в реальном времени с применением функциональных операторов высокого порядка в .NET Reactive Extensions для формирования асинхронных комбинаторов событий. Изученные методы затем будут использованы для реализации рассчитанного на конкурентность реактивного шаблона «издатель — подписчик».

- ❑ В главе 7 дается объяснение модели программирования на основе задач в применении к функциональному программированию для реализации конкурентных операций с использованием шаблона Monadic. Затем этот метод применяется для построения конкурентного конвейера на основе функциональной парадигмы программирования.
- ❑ Глава 8 посвящена реализации неограниченных параллельных вычислений с помощью модели асинхронного программирования на C#. В этой главе также рассматриваются методы обработки ошибок и методы построения асинхронных операций.
- ❑ В главе 9 описывается асинхронный рабочий процесс на F#. В ней показано, как отсроченная и явная оценка в этой модели позволяет получить более высокую композиционную семантику. Затем мы научимся реализовывать пользовательские вычислительные выражения для повышения уровня абстракции до декларативного программирования.
- ❑ В главе 10 показано, как на основе знаний, полученных в предыдущих главах, можно реализовать комбинаторы и шаблоны, такие как Functor, Monad и Applicative, для составления и запуска нескольких асинхронных операций и обработки ошибок без побочных эффектов.
- ❑ В главе 11 анализируется реактивное программирование с использованием программной модели передачи сообщений. В ней раскрывается концепция естественной изоляции как технологии, дополняющей неизменяемость и позволяющей создавать конкурентные программы. В этой главе основное внимание уделяется классу MailboxProcessor, используемому в F# для распределения параллельной работы с применением агентного программирования и подхода без разделения ресурсов.
- ❑ В главе 12 описывается агентное программирование с использованием библиотеки TPL Dataflow из .NET с примерами на C#. Здесь показано, как реализовать на C# агенты без сохранения состояния и с сохранением состояния, а также как параллельно выполнить несколько вычислений, которые обмениваются данными между собой, применяя (передавая) сообщения в стиле конвейера.

В части III показано, как реализовать на практике все функциональные методы конкурентного программирования, изученные в предыдущих главах.

- ❑ В главе 13 представлен набор полезных рецептов для решения сложных проблем конкурентности, взятых из реальной практики. В этих рецептах используются все функциональные шаблоны, описанные в данной книге.
- ❑ В главе 14 описано полноценное приложение, разработанное и внедренное с применением функциональных конкурентных шаблонов и методов, изученных в этой книге. Вы создадите хорошо масштабируемое, отзывчивое серверное приложение и реактивную клиентскую программу. В книге представлены две версии: одна для iOS (iPad), созданная с помощью Xamarin Visual Studio, и вторая — созданная с помощью Windows Presentation Foundation (WPF). Для обеспечения максимальной масштабируемости в серверном приложении использована комбинация различных моделей программирования, таких как асинхронная, агентская и реактивная.

В книге также содержится три приложения.

- ❑ В приложении А кратко описаны основные понятия функционального программирования, а также представлена базовая теория функциональных методов, использованных в данной книге.
- ❑ В приложении Б раскрываются основные понятия языка F#. Это базовый обзор F#, который позволит вам ближе познакомиться с данным языком и комфортно чувствовать себя в процессе чтения книги.
- ❑ В приложении В наглядно демонстрируется несколько методов, упрощающих взаимодействие между асинхронным рабочим процессом на F# и задачей .NET на C#.

О коде

В этой книге содержится много примеров исходного кода: в виде многочисленных листингов и прямо в тексте. В обоих случаях исходный код выделен **вот таким моноширинным шрифтом**, что позволяет отличать его от обычного текста. Иногда код также выделен **жирным шрифтом**, чтобы подчеркнуть обсуждаемую тему.

В некоторых местах первоначальный исходный код переформатирован; мы добавили разрывы строк и изменили отступы так, чтобы наилучшим образом разместить код на странице. Но иногда даже этого было недостаточно, и тогда в листингах были вставлены маркеры переноса строки (**►**). Кроме того, комментарии в исходном коде часто удалены из листингов, так как описание кода содержится в тексте главы. Многие листинги снабжены аннотациями к коду, в которых описываются важные понятия.

Исходный код листингов, представленных в этой книге, можно загрузить с сайта издательства (www.manning.com/books/concurrency-in-dotnet) и с GitHub (<https://github.com/rikace/fConcBook>). Большая часть кода представлена в двух версиях: на C# и F#. Инструкции по использованию кода содержатся в файле README, который находится в корневом каталоге репозитория.

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.

Об авторе



Рикардо Террелл (Riccardo Terrell) – опытный инженер по программному обеспечению, обладатель статуса Microsoft MVP, увлеченный идеей функционального программирования. Имеет более чем 20-летний опыт разработки экономически эффективных технологических решений в конкурентной экономической среде.

В 1998 г. Рикардо основал в Италии собственный бизнес по разработке программного обеспечения. Он специализировался на индивидуальной разработке медицинского программного обеспечения по требованиям

клиентов. В 2007 г. Рикардо переехал в Соединенные Штаты и с тех пор является старшим разработчиком программного обеспечения .NET и старшим архитектором программного обеспечения по предоставлению экономически эффективных технологических решений для бизнеса. Рикардо специализируется на интеграции передовых технологических инструментов для повышения внутренней эффективности, повышения производительности и сокращения издержек.

Он хорошо известен как активный участник сообщества функционального программирования, включая встречи .NET и международные конференции. Рикардо верит в мультипарадигматическое программирование как механизм повышения эффективности кода. Вы можете присоединиться к Рикардо в его исследованиях кода, которые он ведет в своем блоге www.rickyterrell.com.

Об иллюстрации на обложке

На обложке этой книги — человек из деревни в Абиссинии, стране, которую сегодня называют Эфиопией. Иллюстрация взята из испанского альбома с изображениями костюмов разных народов, впервые опубликованного в Мадриде в 1799 г., с гравюрами Мануэля Альбуерне (Manuel Albuerne) (1764–1815). На титульной странице книги написано:

Coleccion general de los Trages que usan actualmente todas las Nacionas del Mundo desubierto, dibujados y grabados con la mayor exactitud por R.M.V.A.R. Obra muy util y en especial para los que tienen la del viajero universal.

Если перевести это как можно точнее, то получится следующее:

Общая коллекция костюмов, используемых в настоящее время в странах известного мира, смоделированная и напечатанная с большой точностью R.M.V.A.R. Эта работа особенно полезна для тех, кто считает себя универсальным путешественником.

О граверах, дизайнерах и рабочих, которые вручную раскрасили эту иллюстрацию, известно очень мало. Но точность исполнения данного рисунка очевидна. Абиссинец — всего лишь одна из многих фигур, представленных в этой красочной коллекции. Многообразие костюмов ярко свидетельствует об уникальности и индивидуальности городов и стран мира всего 200 лет назад. То было время, когда по одежде можно было однозначно определить принадлежность человека к тому или иному региону, даже если расстояние между этими регионами не превышало нескольких десятков миль. Данный альбом говорит об изолированности и дистанцированности между людьми в то время и в любой другой исторический период, кроме того, в котором живем мы, нашего гиперактивного настоящего.

С тех пор традиции одежды изменились, а ее различия в зависимости от страны, некогда столь значительные,стерлись. Сегодня зачастую трудно отличить жителя одного континента от обитателя другого. Возможно, с оптимистической точки зрения мы отдали культурное и визуальное многообразие в обмен на более богатую личную жизнь — или более разностороннюю и интересную интеллектуальную и техническую жизнь.

Издательство Manning приветствует изобретательность, инициативу и получение удовольствия от компьютерного бизнеса, поэтому снабжает книги обложками с иллюстрациями из данного альбома, свидетельствующими о широком многообразии жизни в разных странах два столетия назад.

Часть I

*Преимущества
функционального
программирования
в применении
к конкурентным
программам*

Функциональное программирование — это парадигма программирования, которая фокусируется на абстракции и компоновке. В первых трех главах книги вы научитесь рассматривать вычисления как определение значения выражений и избегать изменения данных. Функциональная парадигма позволяет усовершенствовать конкурентное программирование, предоставляя инструменты и методы для написания детерминированных программ, в которых выходные данные зависят только от входных, а не от состояния программы во время выполнения. Благодаря функциональной парадигме вам будет проще писать код с меньшим количеством ошибок, так как она позволяет разделять проблемы между чисто функциональными аспектами, изолировать побочные эффекты и контролировать нежелательное поведение.

В этой части книги представлены основные понятия и преимущества функционального программирования в применении к конкурентным программам. Здесь будут рассмотрены такие понятия, как программирование с использованием чистых функций, неизменяемость, «ленивое» выполнение и компоновка.

1

Основы функциональной конкурентности

В этой главе:

- зачем нужна конкурентность;
- различия между конкурентностью, параллелизмом и многопоточностью;
- как избежать типичных ошибок при написании конкурентных приложений;
- обмен переменными между потоками;
- использование функциональной парадигмы для разработки конкурентных программ.

В прошлом разработчики программного обеспечения были уверены, что со временем их программы станут работать быстрее, чем когда-либо. На протяжении многих лет это подтверждалось за счет улучшения аппаратного обеспечения, которое позволяло повышать скорость работы программ с каждым новым поколением аппаратных средств.

В течение последних 50 лет индустрия компьютерного оборудования претерпевала непрерывные улучшения. До 2005 г. эволюция процессоров исправно поставляла все более быстрые одноядерные CPU, пока наконец не был достигнут предел скорости процессора, предсказанный Гордоном Муром (Gordon Moore). Мур, учений в области компьютерных технологий, еще в 1965 г. предсказал, что плотность и скорость транзисторов будут удваиваться каждые 18 месяцев, пока не достигнут максимальной скорости, за пределы которой технология не способна выйти. Первоначальное предсказание увеличения скорости процессоров предполагало тенденцию к удвоению скорости в течение десяти лет. Прогноз Мура, известный как закон

Мура, оказался верным — за исключением того, что прогресс продолжался почти 50 лет (несколько десятилетий после того, как был сформулирован).

Сегодня одноядерный процессор почти достиг скорости света и при этом выделяет огромное количество тепла из-за рассеивания энергии; данное тепло является ограничивающим фактором для дальнейших улучшений.

CPU почти достиг скорости света

Скорость света является абсолютным физическим пределом для передачи электрической энергии, в том числе и для передачи электрических сигналов в процессоре. Никакие данные не могут передаваться быстрее, чем свет. Соответственно, сигналы не могут распространяться по поверхности микросхемы настолько быстро, чтобы обеспечить еще более высокую скорость. Базовая частота цикла у современных чипов составляет примерно 3,5 ГГц. Это означает один цикл за каждую $1/3500000000$ с, то есть за 2,85 нс. Скорость света равна примерно 3 \times 10 8 м/с, а это означает, что данные способны распространяться примерно на 0,30 см/нс. Но чем больше чип, тем больше времени требуется для прохождения через него данных.

Существует фундаментальная взаимосвязь между длиной электрической цепи (физическими размером процессора) и скоростью обработки данных: времяя, необходимое для выполнения операции, представляет собой отношение длины цепи к скорости света. Поскольку скорость света постоянна, единственной переменной является размер процессора; другими словами, чтобы увеличить скорость, нужен небольшой процессор, ведь чем короче цепь, тем меньше для нее требуется переключателей. Чем меньше процессор, тем быстрее передача данных. В сущности, именно создание все более мелких чипов было основным методом получения более быстрых процессоров с более высокими тактовыми частотами. И это было сделано столь эффективно, что мы почти достигли физического предела для повышения скорости процессора.

Например, если повысить тактовую частоту до 100 ГГц, то длительность цикла составит 0,01 нс и за это время сигналы будут распространяться только на 3 мм. Следовательно, ядро процессора в идеале должно иметь размер около 0,3 мм. Такой путь ведет к ограничению физических размеров. Кроме того, такая высокая частота при столь небольшом размере процессора создает тепловые проблемы. Мощность переключения транзистора примерно равна частоте в квадрате, поэтому при переходе от 4 к 6 ГГц выделение энергии увеличивается на 225 % (что приводит к нагреву). Помимо размера чипа, возникает проблема его теплового разрушения: от повышения температуры появляются изменения в структуре кристалла.

Предсказание Мура о скорости переключения транзисторов осуществилось (транзисторы не могут работать быстрее), но не устарело (в современных процессорах повышается плотность транзисторов, что обеспечивает возможность параллелизма в пределах этой максимальной скорости). Благодаря сочетанию многоядерной архитектуры и моделей параллельного программирования закон Мура продолжает действовать! Производительность одноядерных процессоров больше не повышается, но разработчики приспосабливаются к этому, создавая многоуровневую архитектуру и программное обеспечение, которое поддерживает и интегрирует конкурентность.

Началась революция процессоров. Новая тенденция построения многоядерных процессоров привела к тому, что основным направлением программирования стало параллельное программирование. Архитектура многоядерных процессоров позволяет строить более эффективные вычисления, но вся эта мощь требует дополнительных усилий от разработчиков. Если программистам нужен более производительный код, то им придется осваивать новые шаблоны проектирования, чтобы максимально эффективно использовать аппаратные средства за счет параллелизма и конкурентности.

В этой главе мы поговорим о том, что такое конкурентность, изучим некоторые ее преимущества, а также проблемы, возникающие при написании типичных конкурентных программ. Затем мы познакомимся с концепциями функциональной парадигмы, которые позволяют преодолеть традиционные ограничения с применением простого и поддерживаемого кода. В конце этой главы вы поймете, почему конкурентность является ценной моделью программирования и почему функциональная парадигма — правильный инструмент для написания корректных конкурентных программ.

1.1. ЧТО ВЫ УЗНАЕТЕ ИЗ ЭТОЙ КНИГИ

В этой книге мы рассмотрим особенности и проблемы написания многопоточных конкурентных приложений в традиционной парадигме программирования. Мы научимся успешно решать данные проблемы и узнаем, как избежать ошибок конкурентности, используя функциональную парадигму. Затем мы познакомимся с преимуществами применения абстракций в функциональном программировании для создания декларативных, простых в реализации и высокопроизводительных конкурентных программ. На протяжении этой книги мы исследуем сложные проблемы конкурентного программирования, которые позволят нам познакомиться с рекомендуемыми методами, необходимыми для построения конкурентных и масштабируемых программ в среде .NET с использованием функциональной парадигмы. Вы узнаете, как функциональное программирование помогает разработчикам поддерживать конкурентность, поощряя применение неизменяемых структур данных, которые можно передавать между потоками, не беспокоясь о разделенном состоянии и избегая побочных эффектов. Заканчивая читать эту книгу, вы научитесь писать более модульный, легко читаемый и поддерживаемый код на языках C# и F#. Вы станете продуктивнее и компетентнее в написании программ, работающих с максимальной производительностью и при этом состоящих из меньшего количества строк кода. В итоге, вооружившись новыми навыками, вы получите знания, необходимые для того, чтобы стать экспертом в разработке успешных высокопроизводительных решений.

Из этой книги вы узнаете следующее.

- ❑ Как объединять асинхронные операции с библиотекой параллельных задач (Task Parallel Library).
- ❑ Как избежать типичных проблем и устранять неполадки в многопоточных и асинхронных приложениях.

- ❑ Какие конкурентные модели программирования используют функциональную парадигму (функциональная, асинхронная, управляемая событиями, а также передача сообщений с агентами и акторами).
- ❑ Как создавать высокопроизводительные конкурентные системы с применением функциональной парадигмы.
- ❑ Как составлять выражения и выполнять асинхронные вычисления в декларативном стиле.
- ❑ Как легко ускорить «чистые» последовательные программы, используя программирование с распараллеливанием данных.
- ❑ Как декларативно реализовать реактивные и ситуационные программы с потоками событий в Rx-стиле.
- ❑ Как использовать функциональные конкурентные коллекции для создания неблокирующихся многопоточных программ.
- ❑ Как писать масштабируемые, производительные и надежные серверные приложения.
- ❑ Как решать проблемы с применением конкурентных шаблонов программирования, таких как Fork/Join, параллельное агрегирование и технология «разделяй и властвуй».
- ❑ Как обрабатывать большие наборы данных с помощью параллельных потоков и параллельных реализаций Map/Reduce.

Предполагается, что читатели этой книги знакомы с основами программирования в целом, но не функционального программирования в частности. Для того чтобы применять функциональную конкурентность при написании кода, необходимо лишь усвоить подмножество понятий из области функционального программирования, и я объясню, что вам нужно знать на этом пути. Таким образом, вы получите многие преимущества функциональной конкурентности за более короткий срок обучения, сосредоточившись на том, что вы сможете сразу использовать в своей ежедневной работе по программированию.

1.2. Начнем с терминологии

В этом разделе будут определены термины, связанные с темой данной книги, чтобы дальше мы говорили на одном языке. В компьютерном программировании некоторые термины (такие как *конкурентность*, *параллелизм* и *многопоточность*) используются в одном и том же контексте, но имеют разные значения. Из-за сходства данные термины часто рассматривают как одно и то же, но это неверно. Когда требуется обсудить поведение программы, крайне важно провести различия между терминами компьютерного программирования. Например, конкурентность по определению является многопоточной, но многопоточность не обязательно конкурентна. Можно легко написать для многоядерного процессора функцию, использующую его как одноядерный процессор, но не наоборот.

Цель данного раздела — согласовать употребление определений и терминов, связанных с темой этой книги. В конце раздела вы узнаете значение следующих терминов:

- последовательное программирование;
- конкурентное программирование;
- параллельное программирование;
- многозадачность;
- многопоточность.

1.2.1. При последовательном программировании задачи выполняются одна за другой

Последовательное программирование — это пошаговый способ выполнения задачи. Рассмотрим простой пример: нужно получить чашку капучино в маленьком кафе. Сначала вы стоите в очереди, чтобы сделать заказ у единственного в заведении бариста. Бариста отвечает за выполнение заказа и доставку напитка; более того, он может приготовить только один напиток за один раз, поэтому вам придется терпеливо — или не очень — ожидать своей очереди сделать заказ. Чтобы приготовить капучино, нужно смолоть зерна, заварить кофе, вскипятить молоко, сделать пенку и смешать кофе с молоком. Поэтому вам требуется еще больше времени, чтобы получить свой капучино. Весь процесс показан на рис. 1.1.

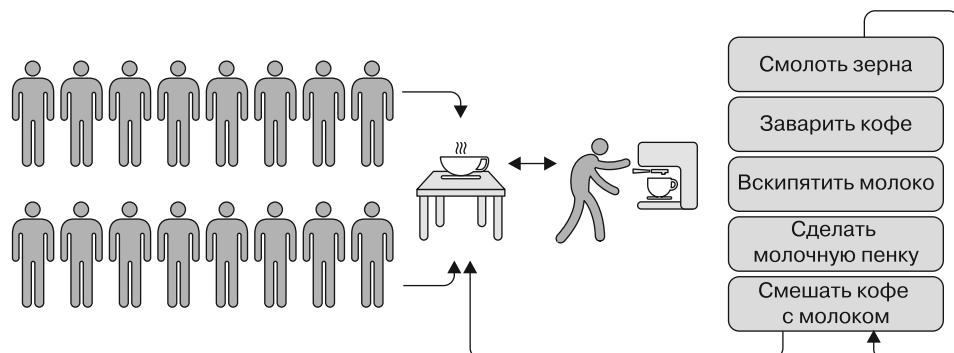


Рис. 1.1. Для каждого человека в очереди бариста последовательно повторяет один и тот же набор инструкций (мелет зерна, варит кофе, кипятит молоко, делает молочную пенку и смешивает кофе с молоком, чтобы приготовить капучино)

На рис. 1.1 показан пример последовательной работы, когда предыдущая задача должна быть завершена, прежде чем можно будет перейти к следующей. Это удобный подход с понятным набором систематических (пошаговых) инструкций о том, что и когда должно быть сделано. В данном примере бариста, скорее всего, не запускается и не допустит ошибки при приготовлении капучино, поскольку все операции

ясны и упорядочены. Недостатком пошагового приготовления капучино является то, что бариста должен ждать завершения отдельных этапов процесса. Пока мелется кофе или кипятится молоко, бариста, в сущности, неактивен (заблокирован). Такая же концепция применима к последовательной и конкурентной моделям программирования. Как показано на рис. 1.2, последовательное программирование требует поэтапного выполнения процессов в порядке возрастания номеров операций, линейно, по одной инструкции в единицу времени.

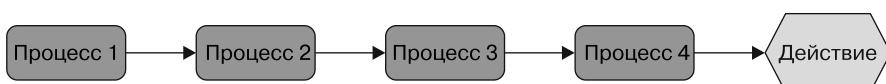


Рис. 1.2. Типичное последовательное кодирование, включающее в себя поэтапное выполнение процессов в порядке возрастания их номеров

В императивном и объектно-ориентированном программировании (ООП) мы склонны писать код, который ведет себя последовательно, при этом все внимание и ресурсы сосредоточены на текущей задаче. Мы моделируем и реализуем программу, выполняя упорядоченный набор операций, одну за другой.

1.2.2. При конкурентном программировании выполняется несколько задач в одно и то же время

А что, если бариста предпочитает начать несколько дел одновременно и выполнять их конкурентно?

Тогда очередь клиентов станет двигаться гораздо быстрее (а вместе с ней ускорится и сбор чаевых). Например, после того как кофе смолот, бариста может начать заваривать эспрессо. Пока кофе заваривается, бариста может принять новый заказ или поставить кипятить молоко, или начать делать пенку. На первый взгляд, бариста выполняет несколько операций одновременно (многозадачность), но это только иллюзия.

Подробнее многозадачность будет описана в подразделе 1.2.4. На самом деле, поскольку у бариста есть только одна машина для приготовления эспрессо, ему придется приостановить задачу, чтобы начать или продолжить другую. Это означает, что бариста выполняет только одну задачу за раз, как показано на рис. 1.3. Для современных многоядерных компьютеров это напрасная трата ценных ресурсов.

Конкурентность означает возможность одновременного запуска нескольких программ или нескольких частей одной программы. В компьютерном программировании применение конкурентности в приложении обеспечивает реальную многозадачность с разделением приложения на несколько независимых процессов, которые выполняются одновременно (конкурентно) в разных потоках. Это может происходить либо в одном ядре процессора, либо параллельно, если доступно несколько ядер. Пропускная способность (скорость, с которой процессор выполняет

вычисления) и отзывчивость программы могут быть повышенены посредством асинхронного или параллельного выполнения задач. Например, приложение, которое передает видео, является конкурентным, поскольку оно одновременно получает цифровые данные из сети, распаковывает их и обновляет изображение на экране.

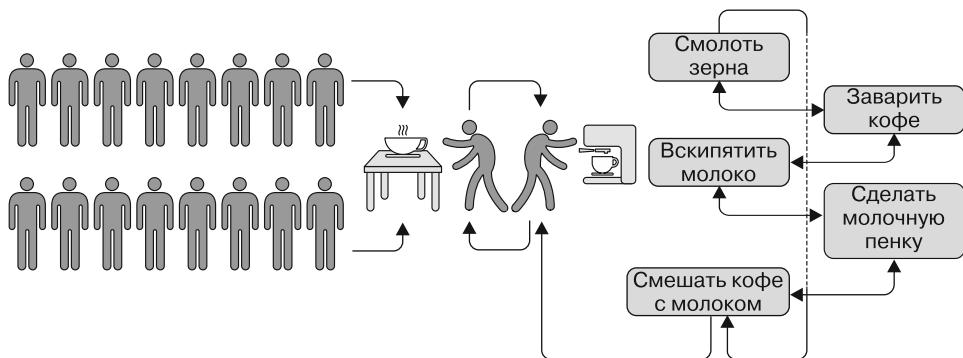


Рис. 1.3. Бариста переключается между операциями (многозадачность) приготовления кофе (мелет и заваривает) и готовит молоко (кипятит и делает пенку). В результате он чередует выполнение задач, создавая иллюзию многозадачности. Но из-за совместного использования общих ресурсов выполняется только одна операция за раз

Конкурентность создает впечатление, что эти потоки работают параллельно и что разные части программы могут выполняться одновременно. Но в одноядерной среде вместо этого выполнение одного потока временно приостанавливается и программа переключается на другой поток, как в случае с бариста на рис. 1.3. Если бариста захочет ускорить приготовление кофе за счет одновременного выполнения нескольких задач, то ему следует увеличить доступные ресурсы. В компьютерном программировании этот процесс называется параллелизмом.

1.2.3. При параллельном программировании выполняется несколько задач одновременно

С точки зрения разработчика мы думаем о параллелизме, когда рассматриваем такие вопросы: «Как может моя программа выполнять несколько операций одновременно?» или «Как моя программа может решить одну проблему быстрее?». *Параллелизм* — это концепция выполнения нескольких задач одновременно, конкурентно, буквально в одно и то же время на разных ядрах, что позволяет повысить скорость выполнения приложения. Все параллельные программы являются конкурентными, но, как мы уже видели, не всегда конкурентность параллельна. Это связано с тем, что параллелизм зависит от реальной среды выполнения и требует аппаратной поддержки (несколько ядер). Параллелизм достижим только на многоядерных устройствах (рис. 1.4) и является средством повышения производительности и пропускной способности программы.

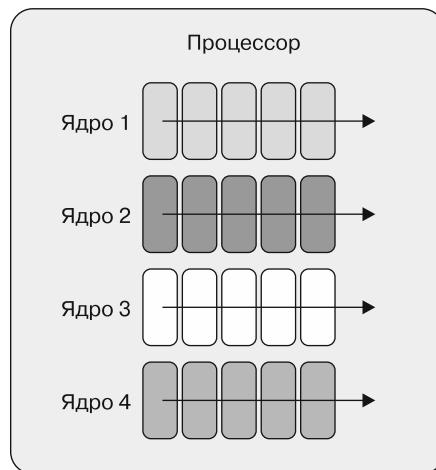


Рис. 1.4. Только многоядерные машины обеспечивают параллелизм для одновременного решения разных задач. На этом рисунке каждое ядро выполняет самостоятельную задачу

Возвращаясь к примеру с кафе, представим, что вы менеджер и хотите сократить время ожидания клиентов, ускорив приготовление напитков. Интуитивное решение — нанять второго бариста и поставить вторую кофемашину. Когда два бариста будут работать одновременно, очереди клиентов смогут обрабатываться независимо и параллельно и процесс приготовления капучино ускорится (рис. 1.5).

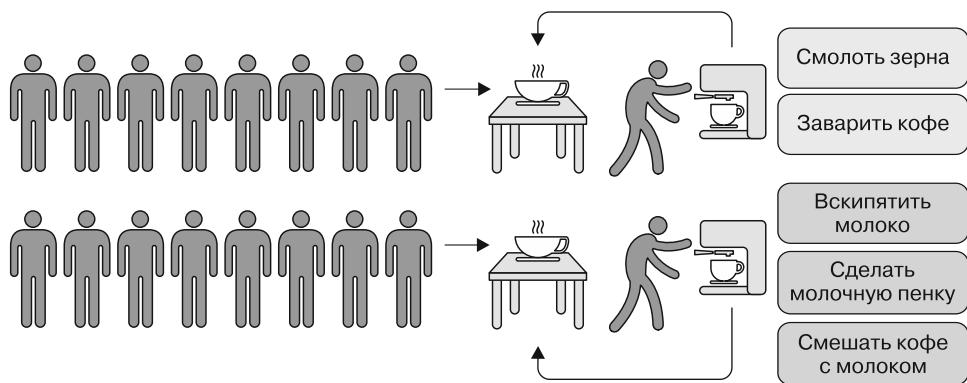


Рис. 1.5. Приготовление капучино идет быстрее, потому что два бариста работают параллельно на двух кофемашинах

Отсутствие перерывов в работе дает выигрыш в производительности. Целью параллелизма является максимальное использование всех доступных вычислительных ресурсов; в данном случае два бариста работают параллельно на отдельных кофемашинках (многоядерная обработка).

Параллелизм может быть достигнут, если одна задача разбивается на несколько независимых подзадач, которые затем запускаются на всех доступных ядрах. На рис. 1.5 многоядерная машина (две кофемашины) допускает параллелизм для одновременного выполнения разных задач (двух занятых бариста) без перерыва.

Концепция синхронизации является фундаментальной для одновременного выполнения параллельных операций. В такой программе операции будут *конкурентными*, если они могут выполняться параллельно, и *параллельными*, если их выполнение перекрываеться во времени (рис. 1.6).

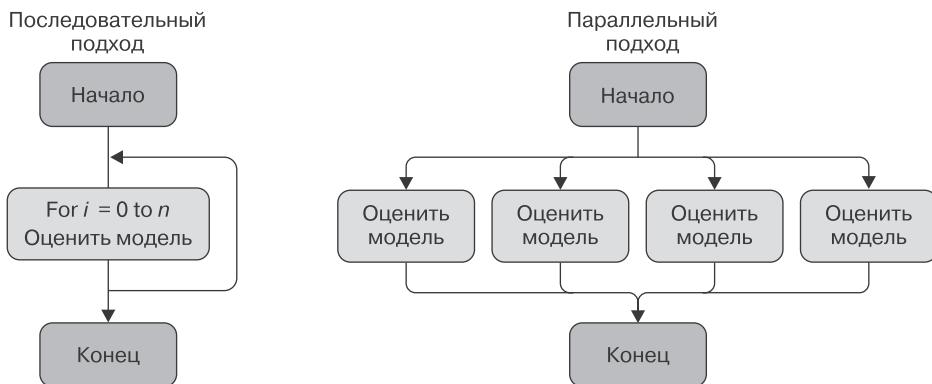


Рис. 1.6. Параллельные вычисления — это тип вычислений, в котором одновременно выполняется множество расчетов, работающих по следующему принципу: большие проблемы часто можно разделить на более мелкие, которые затем решаются одновременно

Параллелизм и конкурентность — связанные модели программирования. Любая параллельная программа одновременно конкурентна, но не всякая конкурентная программа параллельна. При этом параллельное программирование является подмножеством конкурентного. Конкурентность описывает строение системы, а параллелизм — выполнение. Конкурентная и параллельная модели программирования напрямую связаны с аппаратной средой, в которой они выполняются.

1.2.4. При многозадачности выполняется несколько задач одновременно

Многозадачность — это концепция одновременного выполнения нескольких задач в течение определенного периода времени. Нам такая идея хорошо знакома: ведь в повседневной жизни мы действуем именно многозадачно. Например, ожидая, пока бариста приготовит капучино, мы читаем почту или новости на смартфоне. Мы делаем две вещи одновременно: ожидаем кофе и используем смартфон.

Компьютерная многозадачность была разработана еще в те времена, когда у компьютеров был только один процессор. Тогда она предназначалась для конкурентного выполнения нескольких задач, применяющих одни и те же вычислительные ресурсы.

Изначально только одна задача выполнялась в единицу времени, в режиме квантования времени процессора. (*Квантованием времени* называется сложная логика планирования, которая координирует выполнение задач между несколькими потоками.) Количество времени, выделенного так, чтобы поток успевал выполниться до запуска следующего потока, называется *квантом потока*. Процессорное время разбивается на интервалы таким образом, чтобы каждый поток успевал выполнить одну операцию до того, как контекст выполнения переключится на другой поток. Переключение контекста — это процедура, выполняемая операционной системой для оптимизации производительности посредством многозадачности (рис. 1.7). Но в одноядерном компьютере многозадачность может снизить производительность программы, так как вносит дополнительные издержки на переключение контекста между потоками.

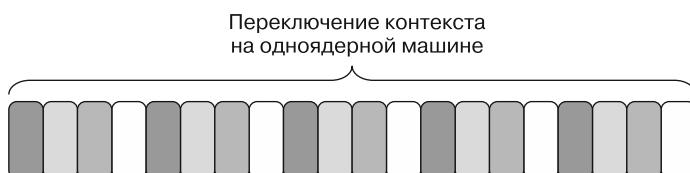


Рис. 1.7. На этом рисунке каждая задача имеет свой цвет. Как видим, переключение контекста в одноядерном компьютере создает иллюзию параллельного выполнения нескольких задач, но на самом деле в каждую единицу времени обрабатывается только одна задача

Существует два вида многозадачных операционных систем:

- ❑ *совместные многозадачные системы*, в которых планировщик позволяет каждой задаче выполняться до тех пор, пока она не завершится или явно не возвратит управление выполнением планировщику;
- ❑ *упреждающие многозадачные системы* (такие как Microsoft Windows), где планировщик назначает каждой задаче свой приоритет, а базовая система, учитывая его, переключает последовательность выполнения задач после завершения каждого выделенного интервала времени, передавая управление другим задачам.

Большинство операционных систем, разработанных за последний десяток лет, дают возможность появления превентивной многозадачности. Многозадачность полезна для обеспечения отзывчивости пользовательского интерфейса, так как позволяет избежать его зависаний во время длительных операций.

1.2.5. Многопоточность как средство повышения производительности

Многопоточность — это расширение концепции многозадачности, целью которой является повышение производительности программы путем максимального использования и оптимизации компьютерных ресурсов. Многопоточность — это форма конкурентности, которая применяет несколько потоков выполнения задач.

Многопоточность подразумевает конкурентность, но конкурентность не обязательно подразумевает многопоточность. Последняя позволяет приложению явно разделять определенные задачи на отдельные потоки, которые запускаются параллельно в одном и том же процессе.

ПРИМЕЧАНИЕ

Процесс — это экземпляр программы, работающей в компьютерной системе. Каждый процесс имеет один или несколько потоков выполнения, и ни один поток не может существовать вне процесса.

Поток — это единица компьютерных вычислений (независимый набор программных инструкций, предназначенных для достижения определенного результата), независимо выполняемый и управляемый планировщиком операционной системы. Многопоточность отличается от многозадачности тем, что, в отличие от многозадачности, при многопоточности потоки совместно применяют ресурсы. Но этот тип совместного использования ресурсов создает больше проблем при программировании, чем многозадачность. Подробнее проблема совместного применения переменных разными потоками обсуждается позже в этой главе, в подразделе 1.4.1.

Концепции параллельного и многопоточного программирования тесно связаны. Но, в отличие от параллелизма, многопоточность является аппаратно независимой: другими словами, она может быть реализована независимо от количества ядер. Параллельное программирование — это надмножество многопоточности. Вы в силах использовать многопоточность для параллелизации программы, например, путем совместного применения ресурсов в одном процессе, но вы также можете распараллелить программу, организовав вычисления в нескольких процессах или даже на нескольких компьютерах. Отношения между этими терминами показаны на рис. 1.8.

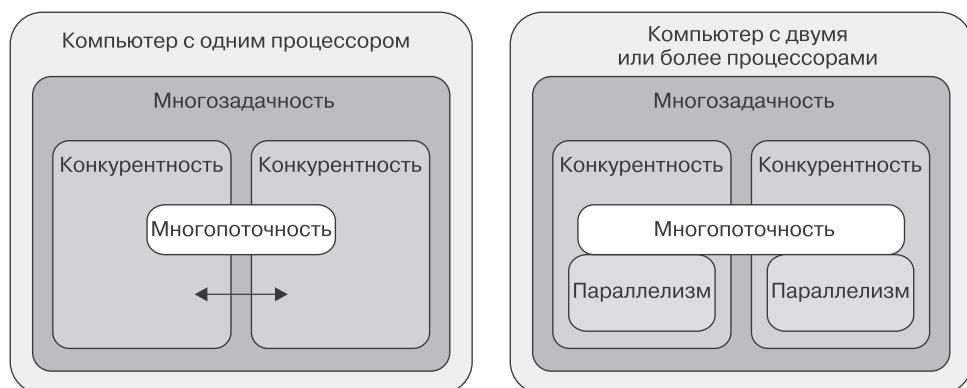


Рис. 1.8. Отношения между конкурентностью, параллелизмом, многопоточностью и многозадачностью в одно- и многоядерном устройстве

Подведем итоги.

- ❑ *Последовательное программирование* представляет собой набор упорядоченных инструкций, выполняемых одна за другой на одном процессоре.
- ❑ При *конкурентном программировании* обрабатывается несколько операций за один раз; это не требует аппаратной поддержки (может использоваться одно или несколько ядер).
- ❑ При *параллельном программировании* выполняется несколько операций одновременно на нескольких процессорах. Все параллельные программы конкурентны, так как выполняются одновременно, но не любая конкурентность является параллельной. Причина в том, что параллелизм достижим только на многоядерных устройствах.
- ❑ При *многозадачности* одновременно выполняется несколько потоков из разных процессов. Многозадачность не обязательно означает параллельное выполнение, которое достигается только при использовании нескольких процессоров.
- ❑ *Многопоточность* расширяет идею многозадачности; это форма конкурентности, которая применяет несколько независимых потоков выполнения, принадлежащих одному и тому же процессу. Каждый поток может реализоваться конкурентно или параллельно, в зависимости от аппаратной поддержки.

1.3. Зачем нужна конкурентность

Конкурентность — естественная часть жизни. Мы, люди, привыкли к многозадачности. Мы можем читать почту и одновременно пить кофе или набирать текст, слушая любимую песню. Основной причиной использования конкурентности в приложениях являются повышение производительности и отзывчивости и достижение низкой латентности. Общеизвестно, что если человек выполняет две задачи одну за другой, то это занимает больше времени, чем если бы два человека выполняли те же две задачи одновременно.

То же самое касается и приложений. Проблема в том, что большинство их не рассчитано на равномерное распределение задач между доступными процессорами. Компьютеры применяются во многих областях, таких как аналитика, финансы, наука и здравоохранение. Количество анализируемых данных увеличивается с каждым годом, два ярких примера этого — Google и Pixar.

В 2012 г. поисковик Google обрабатывал более 2 млн запросов в минуту; в 2014 г. данное число увеличилось более чем вдвое. В 1995 г. компания Pixar выпустила первый полностью компьютерный мультфильм «История игрушек». В компьютерной анимации для каждого изображения генерируется бесчисленное множество деталей и информации, таких как затенение и освещение. Вся эта информация изменяется со скоростью 24 кадра в секунду. В трехмерном фильме скорость изменения информации возрастает экспоненциально.

Создатели «Истории игрушек» использовали для своего фильма 100 соединенных двухпроцессорных машин, и параллельные вычисления были незаменимы.

К моменту появления «Истории игрушек – 2» инструменты Pixar усовершенствовались; для редактирования цифрового фильма компания задействовала 1400 компьютерных процессоров, что значительно повысило качество видео и сократило время редактирования. К началу 2000 г. мощность компьютеров Pixar выросла до 3500 процессоров. Шестнадцать лет спустя мощность компьютеров, используемых для обработки полностью анимационного фильма, достигла безумной цифры 24 000 ядер. Как видим, потребность в параллельных вычислениях продолжает экспоненциально возрастать.

Рассмотрим процессор с N работающих ядер (где N – любое целое число). В однопоточном приложении задействовано только одно ядро. То же самое приложение, выполняющее несколько потоков, будет работать быстрее, и по мере роста требований к производительности требования к числу N также будут возрастать, что делает параллельные программы стандартной моделью программирования будущего.

Запуская на многоядерной машине приложение, которое не было спроектировано с учетом конкурентности, вы зря тратите вычислительные возможности компьютера, потому что приложение, последовательно выполняющее процессы, будет использовать лишь часть доступной мощности компьютера. В этом случае, если вы откроете диспетчер задач или любой другой счетчик производительности процессора, вы заметите, что только одно ядро хорошо нагружено, возможно даже на 100 %, тогда как остальные ядра недогружены или вообще простоявают. Выполнение неконкурентных программ на машине с восемью ядрами означает, что общая эффективность применения ресурсов может составлять всего 15 % (рис. 1.9).

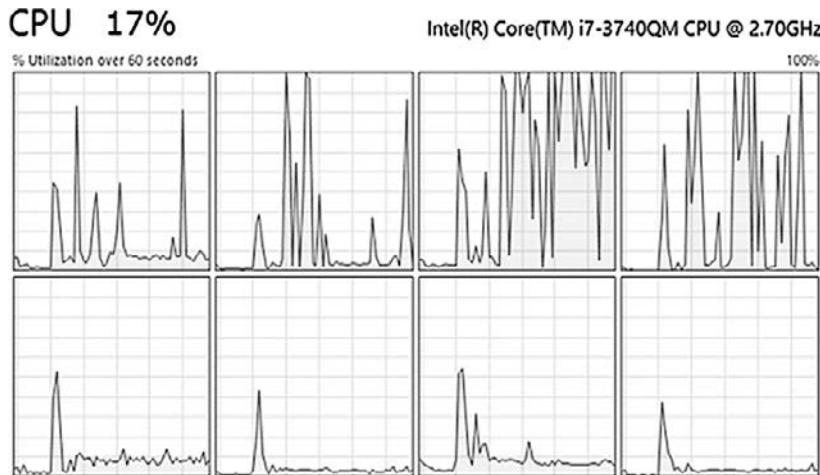


Рис. 1.9. Диспетчер задач Windows показывает, что программа плохо использует ресурсы процессора

Такое нерациональное применение вычислительной мощности недвусмысленно показывает, что последовательный код не является правильной моделью программирования для многоядерных процессоров. Чтобы максимально использовать

доступные вычислительные ресурсы, платформа Microsoft .NET предоставляет возможность параллельного выполнения кода посредством многопоточности. Применяя параллелизм, программа может в полной мере задействовать имеющиеся ресурсы, о чем свидетельствует счетчик производительности процессора, показанный на рис. 1.10. Здесь мы видим, что все ядра процессора высоко нагружены, возможно на 100 %. Современные тенденции развития оборудования направлены в сторону увеличения количества ядер, а не повышения тактовых частот; поэтому разработчикам не остается иного выбора, кроме как принять такую тенденцию и писать параллельные программы.

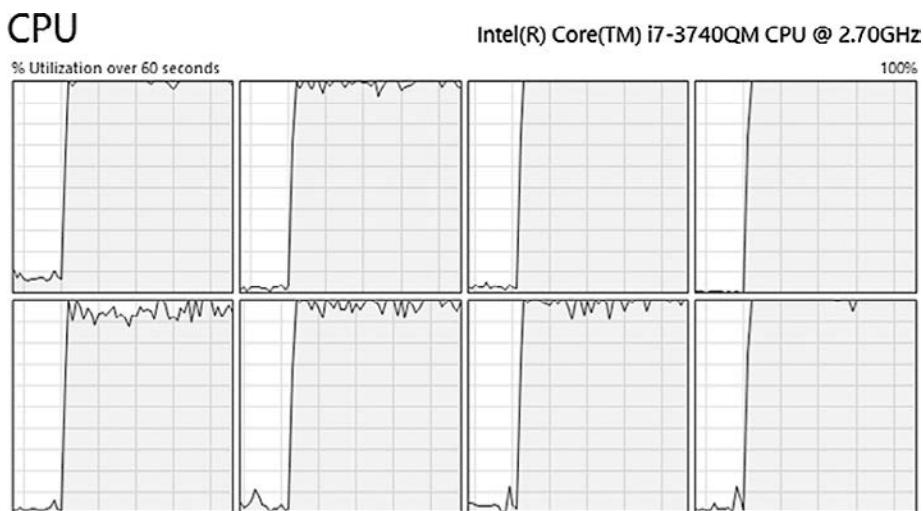


Рис. 1.10. Программа, написанная с учетом конкурентности, позволяет максимально использовать ресурсы процессора — возможно, на 100 %

Настоящее и будущее конкурентного программирования. Освоение параллелизма для создания масштабируемых программ сегодня стало необходимым навыком. Компании заинтересованыанимать и инвестировать в инженеров, имеющих глубокие знания в области написания конкурентного кода. Фактически правильное написание параллельных вычислений экономит время и деньги. Дешевле создавать масштабируемые программы, эффективно применяющие вычислительные ресурсы при меньшем количестве серверов, чем постоянно покупать и устанавливать дорогостоящее оборудование, которое к тому же нерационально используется, чтобы получить тот же уровень производительности. Кроме того, чем больше оборудования, тем больше требуется обслуживания и электроэнергии.

Сейчас самое время, чтобы научиться писать многопоточный код, а наградой за использование функционального программирования (ФП) будет повышение производительности ваших программ. Функциональное программирование — это стиль программирования, при котором компьютерные вычисления рассматриваются как определение значения выражений; такой стиль избегает изменяемых состояний

и изменяемых данных. Поскольку неизменяемость является стандартным требованием, то после добавления компоновки и декларативного программирования с их фантастическими возможностями использование ФП упрощает написание параллельных программ. Подробнее об этом вы узнаете из раздела 1.5.

Несмотря на некоторое беспокойство, вызванное необходимостью думать в новой парадигме, первоначальные сложности изучения параллельного программирования быстро уйдут, а вознаграждение за упорство будет бесконечным. Вы поймете, как это волшебно и захватывающе — открыть диспетчер задач Windows и с гордостью обнаружить, что после изменения кода нагрузка на процессор достигает 100 %. Когда вы достаточно освоитесь и станете уверенно писать код высокомасштабируемых систем с использованием функциональной парадигмы, вам будет трудно вернуться к медленному стилю последовательного кода.

Конкурентность — это очередная инновация. Ожидается, что она будет доминировать в компьютерной индустрии и изменит стиль, в котором разработчики пишут программное обеспечение. Эволюция требований к программному обеспечению в отрасли и спрос на высокопроизводительное программное обеспечение, предлагающее пользователям отличный неблокирующийся интерфейс, по-прежнему будут стимулировать применение конкурентности. В полном соответствии с направлением развития аппаратных средств становится очевидным, что конкурентность и параллелизм — это будущее программирования.

1.4. Ловушки параллельного программирования

Конкурентное и параллельное программирование, безусловно, полезно для быстрого реагирования и быстрого выполнения вычислений. Но за этот выигрыш в производительности и скорости реакции придется заплатить. Пока вы использовали последовательные программы, выполнение кода шло по легкому пути предсказуемости и детерминизма. При многопоточном программировании, напротив, требуется немало труда и усилий, чтобы получить правильно работающую программу. Более того, сложно даже рассуждать о нескольких операциях, выполняемых одновременно, поскольку мы привыкли думать последовательно.

Детерминизм

Детерминизм является фундаментальным требованием при создании программного обеспечения, поскольку часто предполагается, что при каждом запуске компьютерная программа возвращает одни и те же результаты. Однако при параллельном выполнении это требование трудно удовлетворить. Внешние обстоятельства, такие как планировщик операционной системы или согласованность кэша (см. главу 4), могут влиять на время выполнения и, следовательно, на последовательность доступа к двум и более потокам, а также изменять одни и те же области памяти. Эти изменения во времени могут влиять на результат программы.

Процесс разработки параллельных программ — нечто большее, чем просто создание кода и порождение потоков. Написание программ, которые выполняются параллельно, настоятельно требует продуманной структуры. Создавая программу, необходимо постоянно задавать себе следующие вопросы.

- ❑ Какие есть возможности применения конкурентности и параллелизма, чтобы радикально повысить вычислительную производительность и получить высокочувствительное приложение?
- ❑ Как такие программы могут в полной мере использовать возможности, предоставляемые многоядерным компьютером?
- ❑ Как организовать связь с одним и тем же местом памяти для разных потоков, обеспечивая при этом потокобезопасность? (Метод называется *потокобезопасным*, если данные и состояние не повреждаются, когда несколько потоков пытаются получить доступ и изменить данные или состояние в одно и то же время.)
- ❑ Как программа может гарантировать детерминированное выполнение?
- ❑ Как можно распараллелить выполнение программы, не подвергая риску качества конечного результата?

Все это непростые вопросы. Но существует ряд шаблонов и методов, которые могут помочь их решить. Например, при наличии побочных эффектов¹ теряется детерминизм вычислений, потому что последовательность выполнения конкурентных задач перестает быть постоянной. Очевидное решение состоит в том, чтобы избегать побочных эффектов в пользу чистых функций. Вы познакомитесь с данными техниками и методами далее в этой книге.

1.4.1. Риски конкурентности

Писать параллельные программы нелегко. При их разработке необходимо учитывать множество сложных элементов. Создать несколько новых потоков или построить очередность из нескольких заданий в пуле потоков относительно просто, но как гарантировать корректность выполнения программы? Когда много потоков постоянно обращаются к общим данным, необходимо подумать о том, как защитить структуру этих данных, чтобы гарантировать ее целостность. Поток должен записывать и изменять область памяти атомарно², без помех, вносимых другими потоками. Реальность такова, что программы, написанные на императивных языках программирования или на языках, в которых значения переменных могут меняться (изменяемые переменные), всегда будут уязвимы для гонки данных, независимо от уровня синхронизации памяти или используемых конкурентных библиотек.

¹ Побочные эффекты возникают тогда, когда метод изменяет какое-либо состояние, находящееся за пределами его области действия, или передает данные во «внешний мир», например обращаясь к базе данных или производя запись в файловую систему.

² Атомарная операция обращается к общей области памяти и завершается за один шаг относительно других потоков.

ПРИМЕЧАНИЕ

Гонка данных возникает тогда, когда два и более потока в одном процессе одновременно обращаются к одной и той же области памяти, и по меньшей мере один из них обновляет ее, в то время как другие потоки считывают это же значение, не используя какие-либо монопольные блокировки для управления доступом к указанной области памяти.

Рассмотрим случай двух потоков, выполняемых параллельно. Оба потока пытаются получить доступ и изменить общее значение x , как показано на рис. 1.11. Для того чтобы поток 1 изменил эту переменную, требуется несколько инструкций CPU: значение должно быть считано из памяти, затем изменено и в итоге записано обратно в память. Если поток 2 попытается прочитать значение из той же ячейки памяти, пока поток 1 записывает обновленное значение, значение x за это время изменится. Точнее, возможно, что поток 1 и поток 2 одновременно считывают значение x , затем поток 1 изменяет значение x и записывает его обратно в память, а поток 2 в то время также изменяет значение x . Результатом будет искажение данных. Такое явление называется *состоянием гонки*.

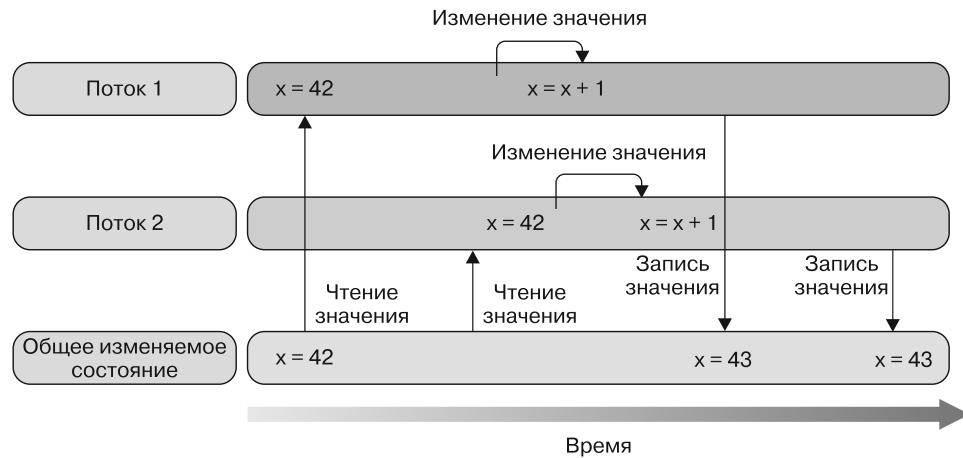


Рис. 1.11. Два потока выполняются параллельно. Оба они пытаются получить доступ к общему значению x и изменить его. Если поток 2 пытается прочитать значение из области памяти, пока поток 1 записывает туда же обновленное значение, значение x изменяется. Результатом этого является искажение данных или возникновение состояния гонки

Сочетание в программе изменяемого состояния и параллелизма — синоним проблем. Решение с точки зрения императивной парадигмы заключается в защите изменяемого состояния путем блокировок и разрешения доступа к памяти только для одного потока за один раз. Такой метод называется *взаимным исключением*, поскольку доступ одного потока к определенной области памяти предотвращает доступ к ней других потоков в то же время. При этом главную роль играет концепция

синхронизации, поскольку, чтобы воспользоваться указанным методом, несколько потоков должны одновременно получать одни и те же данные. Введение блокировок для синхронизации доступа нескольких потоков к общим ресурсам решает проблему искажения данных, но создает большие сложности, которые могут привести к взаимной блокировке.

Рассмотрим случай, показанный на рис. 1.12. Здесь потоки 1 и 2 заблокированы на неопределенное время в ожидании, пока каждый из них завершит работу. Поток 1 получает блокировку А, сразу после чего поток 2 получает блокировку Б. В тот же момент оба потока ждут окончания блокировки, которое никогда не наступит. Это и есть взаимная блокировка.

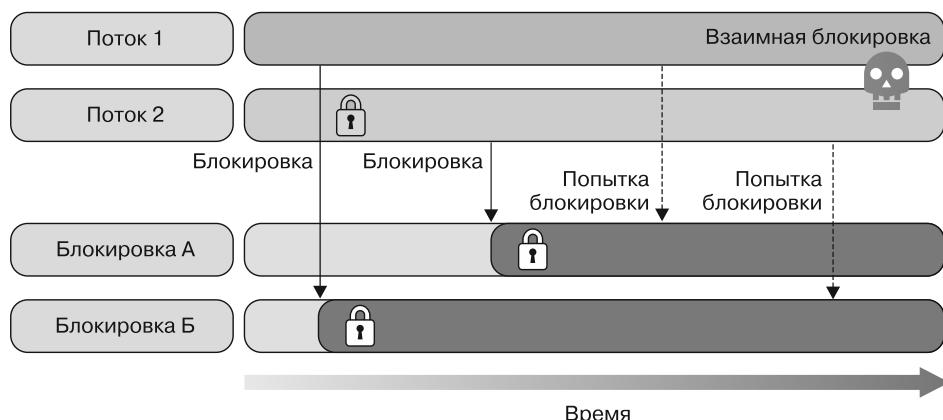


Рис. 1.12. В данном сценарии поток 1 получает блокировку А, а поток 2 — блокировку Б. Затем поток 2 пытается получить блокировку А, а поток 1 — блокировку Б. Но блокировка Б уже занята потоком 2, который ожидает получения блокировки А, а до тех пор не освобождает блокировку Б. В этот момент оба потока ожидают окончания блокировки, которое никогда не наступит. Такая ситуация называется взаимной блокировкой

Ниже приведен краткий список факторов риска при конкурентности. Позже мы рассмотрим подробнее каждый из них, обращая особое внимание на то, как их избежать.

- ❑ Состояние гонки — это состояние, возникающее, когда к совместно применяемому изменяемому ресурсу (например, файлу, изображению, переменной или коллекции) обращается одновременно несколько потоков, создавая несогласованное состояние. Последующее искажение данных делает программу ненадежной и непригодной для использования.
- ❑ Падение производительности — распространенная проблема, возникающая в тех случаях, когда несколько потоков сталкиваются с конфликтом состояний, который требует применения методов синхронизации. Взаимные исключения (мьютексы), как следует из названия, предотвращают параллельное выполнение кода,

заставляя несколько потоков останавливать работу для связи и синхронизации доступа к памяти. Получение и освобождение блокировок вызывает снижение производительности, что замедляет все процессы. По мере увеличения количества ядер затраты на блокировки могут потенциально возрастать. Чем больше задач совместно использует одни и те же данные, тем выше издержки, связанные с блокировками, и тем больше их негативное влияние на вычислительный процесс. В подразделе 1.4.3 будут показаны последствия и издержки, вызванные введением синхронизации блокировок.

- Взаимная блокировка — проблема конкурентности, возникающая из-за использования блокировок. Это происходит, когда существует цикл задач, в котором каждая задача блокируется в ожидании выполнения другой. Поскольку каждая задача ждет, пока выполнится другая задача, все они блокируются на неопределенный срок. Чем больше ресурсов совместно используется потоками, тем больше нужно блокировок, чтобы избежать состояния гонки, и тем выше риск взаимной блокировки.
- Недостаток компоновки — это проблема структуры программы, возникающая из-за введения блокировок в коде. Блокировки затрудняют компоновку. Компоновка способствует упрощению проблем, так как разбивает сложную проблему на более мелкие фрагменты, которые легче решить, а затем снова соединяет их. Компоновка является фундаментальным принципом в ФП.

1.4.2. Эволюция разделяемых состояний

Реальные программы требуют взаимодействия между задачами, такого как обмен информацией для координации работы. Это невозможно реализовать без общих данных, доступных для всех задач. Работа с таким разделяемым состоянием выступает источником большинства проблем, связанных с параллельным программированием, если только общие данные не являются неизменяемыми или если каждая задача не имеет собственную копию. Решение состоит в том, чтобы защитить весь код от проблем конкурентности. Никакой компилятор или другой инструмент не помогут вам разместить эти примитивные блокировки синхронизации в правильном месте кода. Все зависит от вашего мастерства программиста.

Из-за подобных потенциальных проблем сообщество программистов попросту схватилось за голову. В ответ были написаны библиотеки и фреймворки для всех основных объектно-ориентированных языков (таких как C# и Java), которые гарантировали безопасную конкурентность, однако не были частью исходной структуры языка. Данная поддержка представляет собой коррекцию структуры, что подтверждается наличием общей памяти в императивных и объектно-ориентированных средах общего назначения. Между тем функциональные языки не нуждаются в таких гарантиях, поскольку концепция ФП хорошо соответствует конкурентным моделям программирования.

1.4.3. Простой пример из практики: параллельная быстрая сортировка

Алгоритмы сортировки часто используются в технических вычислениях и могут быть в них узким местом. Рассмотрим алгоритм быстрой сортировки Quicksort¹ — вычисление, которое упорядочивает элементы массива. Время выполнения данного вычисления зависит только от быстродействия процессора, и это вычисление хорошо поддается распараллеливанию. Цель примера — продемонстрировать подводные камни преобразования последовательного алгоритма в параллельную версию и обратить внимание на то, что введение в код параллелизма требует дополнительного обдумывания, прежде чем принимать какие-либо решения. В противном случае попытка повысить производительность может привести к противоположному результату.

Quicksort — алгоритм типа «разделяй и властвуй»; сначала он делит большой массив на два меньших подмассива с большими и малыми элементами. Затем Quicksort рекурсивно сортирует подмассивы, что хорошо поддается параллелизации. Алгоритм может работать только на массиве, требуя небольших дополнительных объемов памяти для выполнения сортировки. Он состоит из трех простых этапов, показанных на рис. 1.13.

1. Выбрать опорный элемент.
2. Разделить последовательность на две подпоследовательности в соответствии с их положением относительно опорного элемента.
3. Выполнить сортировку подпоследовательностей методом Quicksort.

Рекурсивные алгоритмы, особенно те, что основаны на принципе «разделяй и властвуй», являются отличными кандидатами для распараллеливания и для реализации вычислений, время которых зависит только от быстродействия процессора.

Библиотека Microsoft Task Parallel Library (TPL), представленная после выхода .NET 4.0, упрощает реализацию и извлечение пользы из параллелизма для данного типа алгоритмов. Используя TPL, вы можете разделить каждый шаг алгоритма и выполнять каждую задачу параллельно, рекурсивно. Это прямолинейная и простая реализация, но следует быть осторожными с уровнем рекурсии, на котором создаются потоки, чтобы не создавать больше задач, чем необходимо.

Чтобы реализовать алгоритм Quicksort, мы воспользуемся ФП-языком F#. Из-за своей внутренней рекурсивной природы идея, лежащая в основе данной реализации, также может быть реализована и на C#, но это потребует императивного подхода с циклом for, с изменением состояния. C# не поддерживает оптимизированные хвостовые рекурсивные функции, в отличие от F#, поэтому существует опасность переполнения стека и появления соответствующего исключения в случае, если значение указателя стека вызовов превысит ограничение стека. В главе 3 мы подробно рассмотрим, как преодолеть такое ограничение C#.

¹ Алгоритм Quicksort изобретен Тони Хоаром (Tony Hoare) в 1960 г. и остается одним из самых известных алгоритмов, имеющих большое практическое значение.

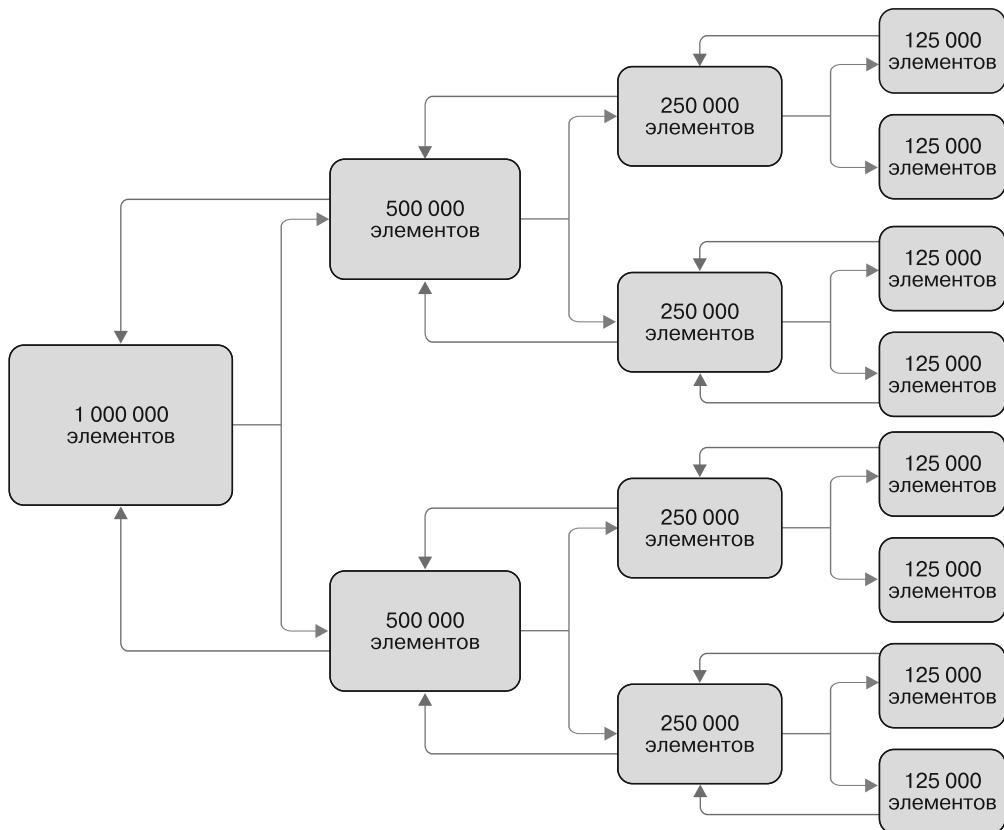


Рис. 1.13. Рекурсивная функция действует по методу «разделяй и властвуй». Каждый блок делится на равные части, где опорный элемент находится в середине последовательности, и так продолжается до тех пор, пока каждая часть кода не будет выполнена независимо от других. Когда все отдельные блоки завершены, они возвращают результат предыдущему вызывающему блоку для сборки. В основе Quicksort лежит идея выбора опорного элемента и разбиения последовательности на подпоследовательности элементов, меньших, чем опорный элемент, и больших, чем опорный элемент, после чего рекурсивная сортировка повторяется для двух меньших последовательностей

В листинге 1.1 показана функция Quicksort, написанная на F#, которая применяет стратегию «разделяй и властвуй». На каждой рекурсивной итерации выбирается опорный элемент, который затем применяется для разделения общего массива. Элементы размещаются вокруг опорного элемента с помощью API List.partition, затем выполняется рекурсивная сортировка списков с каждой стороны опорного элемента. В F# есть отличная встроенная поддержка управления структурами данных. В нашем случае мы используем API List.partition, который возвращает кортеж, содержащий два списка: один из них удовлетворяет заданному условию, а второй — нет.

Листинг 1.1. Простой алгоритм Quicksort

```
let rec quicksortSequential aList =
  match aList with
  | [] -> []
  | firstElement :: restOfList ->
    let smaller, larger =
      List.partition (fun number -> number < firstElement) restOfList
    quicksortSequential smaller @ (firstElement :: -->
      quicksortSequential larger)
```

Выполнение этого алгоритма Quicksort для массива из 1 млн случайных, несортированных целых чисел в моей системе (восемь логических ядер, тактовая частота 2,2 ГГц) занимает в среднем 6,5 с. Но если проанализировать данный алгоритм, то становится очевидной возможность *распараллеливания*. В конце функции `quicksortSequential` мы рекурсивно вызываем `quicksortSequential` для каждого раздела массива, идентифицируемого по `(fun number -> number < firstElement)` `restOfList`. Мы можем переписать эту часть кода для параллельного выполнения, создавая новые задачи с помощью библиотеки TPL (листинг 1.2).

Листинг 1.2. Параллельный алгоритм Quicksort с использованием библиотеки TPL

```
let rec quicksortParallel aList =
  match aList with
  | [] -> []
  | firstElement :: restOfList ->
    let smaller, larger =
      List.partition (fun number -> number < firstElement) restOfList
    let left = Task.Run(fun () -> quicksortParallel smaller)
    let right = Task.Run(fun () -> quicksortParallel larger)
    left.Result @ (firstElement :: right.Result) ← Добавляет результат
    ← каждой задачи
    ← в отсортированный массив
```

Task.Run осуществляет рекурсивные вызовы в задачах, которые могут выполняться параллельно; для каждого рекурсивного вызова задачи создаются динамически

Алгоритм в листинге 1.2 работает параллельно, так что теперь он использует больше ресурсов процессора, распределяя работу по всем доступным ядрам. Но даже при таком улучшенном применении ресурсов общая производительность не соответствует ожиданиям.

Вместо того чтобы уменьшиться, время выполнения резко увеличилось. На прохождение распараллеленного алгоритма Quicksort вместо среднего времени 6,5 с теперь тратится в среднем почти 12 с. Общее время выполнения увеличилось. В данном случае проблема заключается в том, что алгоритм слишком сильно распараллелен. Каждый раз при разбиении внутреннего массива для распараллеливания алгоритма генерируются две новые задачи. Такая схема порождает слишком много задач для имеющегося количества ядер, что повышает издержки на параллелизацию. Это особенно ярко проявляется в алгоритмах типа «разделяй и властвуй», которые предполагают параллелизацию рекурсивной функции. Важно не создавать больше

задач, чем необходимо. Такой неутешительный результат демонстрирует важную характеристику параллелизма: существуют внутренние ограничения на то, сколько дополнительных потоков или процессов будут эффективны для реализации данного алгоритма.

Для того чтобы добиться лучшей оптимизации, можно реорганизовать представленную выше функцию `quicksortParallel`, ограничив рекурсивное распараллеливание в определенной точке. Таким образом, первые рекурсии алгоритма будут выполняться параллельно до самого глубокого уровня, а затем алгоритм вернется к последовательному подходу. Подобная конструкция гарантирует полную нагрузку всех ядер, а издержки, связанные с распараллеливанием, значительно сокращаются.

Этот новый подход к построению алгоритма показан в листинге 1.3. Он учитывает уровень, до которого работает рекурсивная функция; если уровень ниже определенного порога, то распараллеливание прекращается. У функции `quicksortParallelWithDepth` есть дополнительный аргумент `depth`, назначение которого — уменьшить и контролировать количество распараллеливаний рекурсивной функции. При каждом рекурсивном вызове аргумент `depth` уменьшается на единицу; новые задачи создаются до тех пор, пока значение данного аргумента не станет равным нулю. После чего значение, полученное из `Math.Log (float System.Environment.ProcessorCount, 2.) + 4`, передается в `max depth`. Это гарантирует, что каждый уровень рекурсии будет порождать две дочерние задачи, пока не будут задействованы все доступные ядра.

Листинг 1.3. Улучшенный параллельный алгоритм Quicksort с использованием библиотеки TPL

```
let rec quicksortParallelWithDepth depth aList = 
    match aList with
    | [] -> []
    | firstElement :: restOfList ->
        let smaller, larger =
            List.partition (fun number -> number < firstElement) restOfList
        if depth < 0 then
            let left = quicksortParallelWithDepth depth smaller
            let right = quicksortParallelWithDepth depth larger
            left @ (firstElement :: right)
        else
            let left = Task.Run(fun () ->
                quicksortParallelWithDepth (depth - 1) smaller)
            let right = Task.Run(fun () ->
                quicksortParallelWithDepth (depth - 1) larger)
            left.Result @ (firstElement :: right.Result)
```

**Последовательное выполнение QuickSort
с использованием текущего потока**

Одним из важных факторов при выборе количества задач является то, насколько похожим будет прогнозируемое время их решения. В случае с `quicksortParallelWithDepth` продолжительность выполнения задач может существенно различаться, поскольку опорные элементы зависят от несортированных данных. Они не обязательно

приводят к появлению сегментов одинакового размера. Для того чтобы компенсировать неравномерные размеры задач, аргумент `depth` в этом примере вычисляется по формуле, согласно которой создаваемое количество задач превышает количество ядер. По этой формуле максимальное количество задач примерно в 16 раз больше, чем число ядер, так как число задач не может превышать 2^{depth} . Наша цель — сбалансировать рабочую нагрузку Quicksort и не запускать больше задач, чем требуется. Запуск Task на каждой итерации (рекурсии), когда достигается максимальный уровень глубины, превышает максимальную нагрузку на процессоры.

В большинстве случаев Quicksort генерирует несбалансированную рабочую нагрузку, поскольку полученные фрагменты имеют разный размер. Концептуальная формула $\log_2(\text{ProcessorCount}) + 4$ дает значение аргумента `depth` для ограничения и адаптации количества запущенных задач независимо от конкретных случаев¹. Если заменить эту формулу на `depth = log2(ProcessorCount) + 4` и упростить выражение, то увидим, что количество задач составляет $16 * \text{ProcessorCount}$. Ограничение количества подзадач путем измерения глубины рекурсии является чрезвычайно важным методом².

Например, для четырехъядерных машин глубина рассчитывается следующим образом:

```
depth = log2(ProcessorCount) + 4  
depth = log2(2) + 4  
depth = 2 + 4
```

В результате получаем диапазон примерно от 36 до 64 параллельных задач, поскольку на каждой итерации запускаются две задачи для каждой ветви, и это число, в свою очередь, удваивается на каждой следующей итерации. Таким образом, общее разделение между потоками имеет справедливое и подходящее распределение для каждого ядра.

1.4.4. Бенчмаркинг в F#

Мы выполнили пример с Quicksort, использовав цикл «чтение — вычисление — вывод» REPL (Read-Evaluate-Print-Loop) из языка F#. Этот цикл является удобным инструментом для запуска главной части кода, поскольку не требует компиляции программы. REPL отлично подходит для разработки прототипов и анализа данных, так как его применение облегчает процесс программирования. Другим преимуществом является встроенная функциональность `#time`, которая переключает отображение информации о производительности. Когда `#time` включена, в окне F# Interactive отражаются значения реального времени, времени процессора и информация о сборе мусора для каждого раздела интерпретированного и выполненного кода.

¹ Функция \log_2 — это сокращенная запись логарифма по основанию 2. Например, $\log_2(x)$ — логарифм x по основанию 2.

² Напомню, что для любого значения a 2^{a+4} равно 16×2^a и что если $a = \log_2(b)$, то $2^a = b$.

В табл. 1.1 показаны результаты сортировки массива размером 3 Гбайт при включенном режиме 64-битной среды во избежание ограничений по размеру. Сортировка выполнялась на компьютере с восемью логическими ядрами (четыре физических ядра с гиперпоточностью). В табл. 1.1 показано среднее время выполнения программы в секундах за десять запусков.

Таблица 1.1. Бенчмарк сортировки методом Quicksort

Последовательно	Параллельно	Параллельно, четыре потока	Параллельно, восемь потоков
6,52	12,41	4,76	3,50

Важно отметить, что для небольшого массива, размером менее 100 элементов, алгоритмы параллельной сортировки выполняются медленнее, чем последовательная версия, из-за издержек на создание и/или порождение новых потоков. Даже если вы правильно напишете параллельную программу, издержки, создаваемые конкурентными конструкторами, могут перегрузить среду выполнения и производительность программы, вопреки ожиданиям, уменьшится. По этой причине при бенчмаркинге важно проводить сравнения с первоначальным последовательным кодом как базовым, а затем продолжать измерения после каждого изменения, чтобы убедиться, что параллелизм был полезен. Полная стратегия должна учитывать данный фактор и применять параллелизм только в том случае, если размер массива больше порогового (глубина рекурсии), который обычно соответствует количеству ядер, после чего программа по умолчанию возвращается к последовательному поведению.

1.5. Почему для конкурентности выбирают функциональное программирование

Проблема в том, что, по сути, все интересные приложения с конкурентностью требуют преднамеренного и контролируемого изменения разделяемого состояния, такого как экран, файловая система или внутренние структуры данных программы. Поэтому правильное решение состоит в том, чтобы обеспечить механизмы, позволяющие безопасно изменять разделяемые состояния.

*Пейтон Джонс (Peyton Jones), Эндрю Гордон (Andrew Gordon)
и Сигбьорн Финн (Sigbjorn Finne), Concurrent Haskell, «Материалы
23-го симпозиума ACM по основам языков программирования»,
Санкт-Петербург-Бич, Флорида, январь 1996 г.*

Функциональное программирование (ФП), обычно называемое *чистым функциональным программированием*, позволяет свести к минимуму и взять под контроль побочные эффекты. ФП оперирует концепцией преобразования, когда функция создает копию значения x и затем изменяет эту копию, оставляя исходное значение x неизменным

и доступным для использования другими частями программы. Функциональное программирование поднимает вопрос: так ли уж необходимы при разработке программы изменяемость и побочные эффекты? ФП допускает их, но стратегически явно, изолируя данную область от остальной части кода и реализуя методы инкапсуляции.

Основной причиной выбора функциональных парадигм является то, что они позволяют решить проблемы, возникающие в многоядерную эпоху. Высококонкурентные приложения, такие как веб-серверы и аналитические базы данных, страдают от ряда архитектурных проблем. Эти системы должны быть масштабируемыми, чтобы реагировать на большое количество конкурентных запросов, что приводит к необходимости учитывать при проектировании максимальный конфликт ресурсов и высокую частоту планирования. Более того, в таких приложениях часто встречаются состояния гонки и взаимные блокировки, что затрудняет устранение неполадок и отладку кода.

В этой главе мы обсудили ряд типичных проблем, характерных для разработки конкурентных приложений при императивном и объектно-ориентированном стиле программирования. В подобных парадигмах программирования базовыми конструкциями являются объекты. С точки зрения конкурентности, напротив, при оперировании объектами приходится учитывать проблемы, возникающие при переходе от однопоточной программы к интенсивно распараллеленной работе, которая представляет собой совершенно другой, сложный сценарий.

ПРИМЕЧАНИЕ

Поток — это конструкция операционной системы, которая функционирует подобно виртуальному процессору. В любой момент поток может выполняться на физическом CPU в течение заданного периода времени. Когда время выполнения потока истечет, он отключается от процессора, освобождая CPU для другого потока. Поэтому, если один из потоков входит в бесконечный цикл, он не может монополизировать все процессорное время в системе. В конце своего временного интервала поток будет отключен и уступит место другому потоку.

Традиционным решением этих проблем является синхронизация доступа к ресурсам, которая предотвращает конкуренцию между потоками. Но такое решение — обоюдоострый меч, поскольку использование примитивов для синхронизации, таких как `lock` для взаимного исключения, приводит к возможным взаимным блокировкам или возникновению состояния гонки. Фактически состояние переменной (как и следует из слова «*переменная*») может изменяться. В ООП она обычно представляет собой объект, способный изменяться со временем. Из-за этого вы никогда не знаете точно состояние данного объекта и, следовательно, приходится верифицировать его текущее значение, чтобы избежать нежелательного поведения (рис. 1.14).

Важно учитывать, что компоненты систем, в которых реализована концепция ФП, больше не могут мешать друг другу, и их можно использовать в многопоточной среде без задействования стратегий блокировки.

Разработка безопасных параллельных программ с применением общих изменяемых переменных и функций с побочными эффектами требует от программиста значительных усилий: он должен принимать критически важные решения, которые часто приводят к синхронизации в виде блокировки. Избавившись от подобных фундаментальных проблем с помощью функционального программирования, вы также устраняете специфические проблемы, связанные с конкурентностью. Именно поэтому ФП является отличной моделью параллельного программирования. Она прекрасно подходит для программистов, создающих конкурентные программы, позволяя получать корректный код с высокой производительностью в средах с большим количеством потоков и при этом писать простой код. В основе ФП ни переменные, ни состояния не являются изменяемыми и не могут использоваться совместно, а функции не могут иметь побочные эффекты.

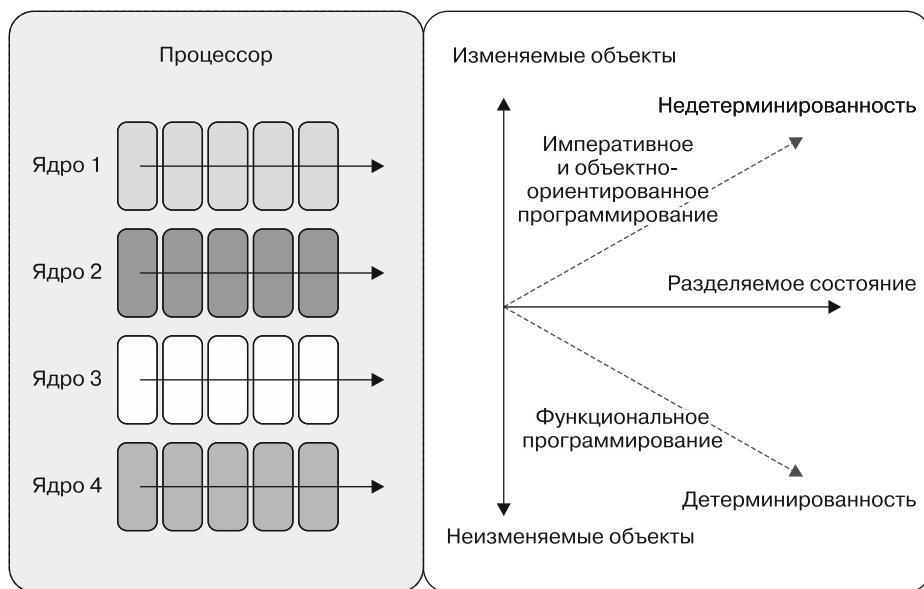


Рис. 1.14. Благодаря тому что в функциональной парадигме неизменяемость является концепцией, принятой по умолчанию, конкурентное программирование в этой парадигме гарантирует детерминированное выполнение программы даже при наличии разделяемых состояний. И наоборот, в императивном и объектно-ориентированном программировании используются изменяемые состояния, которыми трудно управлять в многопоточной среде, и это приводит к появлению недетерминированных программ

ФП – наиболее практичный способ написания конкурентных программ. Попытки писать такие программы на императивных языках приводят к появлению не только сложного кода, но и ошибок, которые трудно обнаруживать, воспроизводить и исправлять.

Как же мы собираемся применять все доступные компьютерные ядра? Ответ прост: нужно задействовать функциональную парадигму!

Преимущества функционального программирования. Есть веские причины изучить ФП, даже если вы не планируете оперировать этим стилем программирования в ближайшем будущем. Но трудно убедить человека тратить время на что-то новое, не показав ему немедленных преимуществ. Выгоды приходят в форме идиоматических особенностей языка, которые поначалу могут показаться непреодолимыми. Однако ФП — это парадигма, которая откроет перед вами новые широкие возможности кодирования и уже вскоре после начала обучения окажет положительное влияние на ваши программы. Всего через несколько недель после начала использования методов ФП читаемость и корректность ваших приложений улучшатся.

Преимущества ФП (с акцентом на конкурентности) включают в себя следующие.

- **Неизменяемость** — свойство, которое предотвращает изменение состояния объекта после его создания. В ФП присвоение значений переменным не является концепцией. После того как значение будет связано с определенным идентификатором, оно не может быть изменено. Функциональный код неизменяем по определению. Неизменяемые объекты могут безопасно передаваться между потоками, что открывает большие возможности оптимизации. Неизменяемость устраниет проблемы искажения памяти (состояние гонки) и взаимные блокировки, так как отсутствуют взаимные исключения.
- **Чистая функция** — функция, не имеющая побочных эффектов. Это означает, что такие функции не меняют никаких входных данных и вообще данных любого типа, расположенных вне тела функции. Функции считаются чистыми, если они прозрачны для пользователя, а возвращаемое ими значение зависит только от входных аргументов. Передавая одни и те же аргументы в чистую функцию, вы всегда получите одинаковый результат, и каждый процесс вернет то же значение, производя последовательное и ожидаемое поведение.
- **Ссылочная прозрачность.** Идея функции, выходные данные которой зависят только от нее и отображают только ее входные данные. Другими словами, каждый раз, когда такая функция получает одни и те же аргументы, результат ее выполнения будет одним и тем же. Ценность этой концепции при параллельном программировании заключается в том, что определение выражения можно заменить его значением и получить тот же смысл. Ссылочная прозрачность гарантирует, что набор функций может быть выполнен в любом порядке и параллельно, без изменения поведения приложения.
- **«Ленивое» выполнение.** Применяется в ФП для получения результата функции по требованию или для отсрочки анализа большого потока данных до тех пор, пока не возникнет такая необходимость.
- **Компонуемость** используется для компоновки функций и создания из простых функций абстракций более высокого уровня. Компонуемость — это самый мощный инструмент для преодоления сложности. Он позволяет описывать и строить решения для сложных задач.

Научившись программировать функционально, вы сможете писать модульный, ориентированный на выражения, концептуально простой код. Сочетание указанных

преимуществ ФП позволит вам понять, что именно делает ваш код, независимо от того, сколько потоков в нем выполняется.

Далее в этой книге будет показано, как применять параллелизм и обходить проблемы, связанные с изменяемыми состояниями и побочными эффектами. Подход к данным концепциям с точки зрения функциональной парадигмы направлен на упрощение и максимальное повышение эффективности кодирования с помощью декларативного программирования.

1.6. Область применения функциональной парадигмы

Изменяться бывает сложно. Часто разработчикам, которые уверенно чувствуют себя в своей области знаний, не хватает мотивации на то, чтобы взглянуть на проблемы программирования под другим углом. Изучать новую парадигму всегда трудно, и требуется время, чтобы перейти на другой стиль разработки. Чтобы изменить перспективу программирования, мало просто изучить новый синтаксис кода для нового языка программирования; требуется изменить способ мышления и подход к решению задач.

Переход с языка, такого как Java, на C# не составляет труда; с точки зрения концепций они одинаковы. Переход от императивной к функциональной парадигме — гораздо более сложная задача. При этом меняются базовые понятия. У вас больше нет состояний. Нет переменных. Нет побочных эффектов.

Но усилия, затраченные на то, чтобы изменить парадигму программирования, будут приносить вам немалые дивиденды. Большинство разработчиков согласятся с тем, что изучение нового языка делает вас лучшим разработчиком. Это можно сравнить с пациентом, которому врач назначил делать зарядку ежедневно в течение 30 мин, чтобы быть здоровым. Пациент знает, что упражнения полезны, но понимает: чтобы выполнять их каждый день, потребуются упорство и определенные жертвы.

Точно так же изучение новой парадигмы само по себе не сложно, но требует само-отверженности, заинтересованности и времени. Я призываю каждого, кто хочет стать лучшим программистом, задуматься об изучении парадигмы ФП. Изучение ФП подобно катанию на американских горках: будут моменты, когда вы почувствуете возбуждение и полет, а затем придет время, когда вам будет казаться, что вы усвоили принципы только для того, чтобы упасть еще ниже — с воплями, — но сама поездка того стоит. Смотрите на изучение ФП как на путешествие, инвестиции в вашу личную и профессиональную карьеру с гарантированным возвратом. Помните, что часть обучения заключается в том, чтобы делать ошибки и развивать навыки, которые позволяют вам избежать таких ошибок в будущем.

На протяжении всего процесса вам следует выделять концепции, трудные для понимания, и попытаться преодолеть эти трудности. Подумайте о том, как использовать данные абстракции на практике, для решения простых задач. Мой опыт показывает, что вы сможете прорваться через все ментальные блокпосты, выяснив, какова

цель каждой концепции, оперируя реальными примерами. В настоящей книге вы познакомитесь с преимуществами ФП, применяемыми для создания конкурентных и распределенных систем. Это узкая область, но, с другой стороны, вы получите несколько отличных фундаментальных концепций, которые сможете использовать в повседневном программировании. Я уверен, что вы получите новое представление о том, как решать сложные проблемы, и станете отличным инженером-программистом, реализуя огромные возможности ФП.

1.7. Зачем использовать F# и C# для функционального конкурентного программирования

Основное внимание в данной книге уделяется разработке и построению высокомасштабируемых и высокопроизводительных систем с использованием функциональной парадигмы для написания корректного конкурентного кода. Это не значит, что вы должны изучить новый язык; вы можете применять функциональную парадигму, прибегая к уже знакомым вам инструментам, таким как многоцелевые языки C# и F#. За последние годы в эти языки было добавлено несколько функциональных возможностей, что упростит вам переход на использование данной новой парадигмы.

Причина, по которой я выбрал данные языки, — принципиально иной подход в них к решению проблем. Оба эти языка программирования могут использоваться для решения одной и той же проблемы совершенно разными способами, что позволяет каждый раз выбирать наилучший инструмент для работы. Благодаря хорошо продуманному набору инструментов вы можете разрабатывать лучшие и более простые решения. В сущности, как разработчик программного обеспечения, вы и должны думать о языках программирования как об инструментах.

В идеале решение должно быть сочетанием проектов на C# и F#, работающих согласованно. Каждый из этих языков поддерживает свою модель программирования, и возможность выбирать для работы каждый раз наилучший инструмент дает огромную выгоду с точки зрения производительности и эффективности. Другим аспектом выбора данных языков является то, что они поддерживают различные конкурентные модели программирования, которые можно комбинировать, в том числе следующие.

- ❑ F# предлагает гораздо более простую модель, чем C#, для асинхронных вычислений, называемую *асинхронными рабочими процессами*.
- ❑ И C#, и F# являются строго типизированными, многоцелевыми языками программирования с поддержкой множества парадигм, включая функциональную, императивную и ООП-технологии.
- ❑ Оба языка выступают частью экосистемы .NET и имеют богатый набор библиотек, которые могут использоваться в равной степени на обоих языках.

- ❑ F# – это изначально функциональный язык программирования, который обеспечивает огромный прирост производительности. В сущности, программы, написанные на F#, как правило, более краткие и приводят к появлению меньшего количества поддерживаемого кода.
- ❑ F# сочетает в себе преимущества функционального декларативного программирования с поддержкой императивного объектно-ориентированного стиля. Это позволяет разрабатывать приложения, применяя существующие навыки объектно-ориентированного и императивного программирования.
- ❑ Благодаря использованию по умолчанию неизменяемых конструкторов F# имеет набор встроенных неблокирующих структур данных. Примером могут служить типы размеченного объединения и записей. Эти типы структурно эквивалентны и не могут быть равными `null`, что приводит к «доверию» целостности данных и более простым сравнениям.
- ❑ В F#, в отличие от C#, настоятельно не рекомендуется использовать значения `null`, также известные как ошибка в миллиард долларов. Вместо этого поощряется применение неизменяемых структур данных. Отсутствие нулевых ссылок помогает свести к минимуму количество ошибок при программировании.

Как появилась нулевая ссылка

Тони Хоар ввел нулевую ссылку в 1965 г., когда создавал объектно-ориентированный язык ALGOL. Спустя 44 года он извинился за свое изобретение, назвав его ошибкой в миллиард долларов. Он сказал: «...Я не мог удержаться от соблазна ввести нулевую ссылку просто потому, что ее было так легко реализовать. Впоследствии это привело к бесчисленным ошибкам, уязвимостям и системным сбоям...»¹

- ❑ Язык F# параллелизуется естественным образом, поскольку в нем по умолчанию используются неизменяемые конструкторы, а благодаря тому, что в его основе лежит .NET, этот язык интегрируется с C#, прибегая к самым современным возможностям на уровне реализации.
- ❑ По своей структуре C# скорее императивный язык, изначально с полной поддержкой ООП. (Я бы определил его как императивный объектно-ориентированный язык.) Функциональная парадигма повлияла на C# только в последние годы, после выхода .NET 3.5. В это время в языке появились такие свойства, как лямбда-выражения и LINQ для списковых включений.
- ❑ C# также обладает отличными инструментами для реализации конкурентности, которые позволяют легко писать параллельные программы и быстро решать сложные проблемы, возникающие в повседневной практике. Действительно, отличная поддержка многоядерной разработки на языке C# является

¹ Из выступления на QCon London в 2009 г.: <http://mng.bz/u74T>.

универсальной; она позволяет быстро разрабатывать и создавать прототипы приложений с высокопараллельной симметричной многопроцессорной обработкой (symmetric multiprocessing, SMP). F# и C# – отличные инструменты для написания конкурентного программного обеспечения; они также предоставляют возможности и варианты для сборки работоспособных решений при их совместном использовании. SMP – это обработка программ несколькими процессорами с общей операционной системой и памятью.

- F# и C# могут взаимодействовать друг с другом. Фактически функция F# может вызывать метод из библиотеки C# и наоборот.

В следующих главах мы обсудим другие параллельные подходы, такие как параллелизм данных, асинхронность и модель программирования с передачей сообщений. Мы построим библиотеки, используя лучшие инструменты, которые предлагает каждый из этих языков программирования, и сравним их с другими языками. Мы также исследуем другие инструменты и библиотеки, такие как TPL и Reactive Extensions (Rx), которые были успешно задуманы, разработаны и реализованы с учетом функциональной парадигмы для получения сборных абстракций.

Очевидно, что индустрии нужна надежная и простая модель конкурентного программирования. Об этом свидетельствует тот факт, что компании – разработчики программного обеспечения вкладывают средства в библиотеки, которые отказываются от уровня абстракции традиционных сложных моделей синхронизации памяти. Примерами таких библиотек более высокого уровня являются Intel Threading Building Blocks (TBB) и Microsoft TPL.

Существуют также интересные проекты с открытым исходным кодом, такие как OpenMP (в нем представлены прагмы – специфические определения компилятора, которые можно использовать для создания новых функций препроцессора и для передачи компилятору информации, специфичной для данной реализации; прагмы можно вставлять в программу, чтобы ее части выполнялись параллельно) и OpenCL (язык низкого уровня для связи с графическими процессорами – GPU). Программирование GPU быстро развивается и получило поддержку Microsoft в виде расширений C++ AMP и Accelerator .NET.

Резюме

- ❑ Не существует «серебряной пули» для решения любых проблем и сложностей конкурентного и параллельного программирования. Вам, как профессионально-му инженеру, нужны разные типы боеприпасов, и вы должны знать, как и когда их использовать, чтобы попасть в цель.
- ❑ Программы должны разрабатываться с учетом конкурентности; программисты не могут продолжать писать последовательный код, закрывая глаза на преимущества параллельного программирования.
- ❑ Закон Мура все еще действует. Он только изменил направление в сторону увеличения количества процессорных ядер вместо повышения скорости одного процессора.
- ❑ При написании конкурентного кода необходимо учитывать различия между конкурентностью, многопоточностью, многозадачностью и параллелизмом.
- ❑ Главное, чего следует избегать в параллельной среде, — это разделяемые изменяемые состояния и побочные эффекты, поскольку они приводят к нежелательному поведению программы и ошибкам.
- ❑ Чтобы избежать ошибок при написании конкурентных приложений, необходимо использовать модели и инструменты программирования, которые повышают уровень абстракции.
- ❑ Функциональная парадигма предоставляет правильные инструменты и принципы для простой и корректной поддержки конкурентности в вашем коде.
- ❑ Функциональное программирование превосходит параллельные вычисления, поскольку неизменяемость является внутренним свойством функциональной парадигмы, что упрощает оперирование общими данными.

Технологии функционального программирования для конкурентных систем

В этой главе:

- решение сложных задач путем компоновки простых решений;
- упрощение функционального программирования путем замыканий;
- повышение производительности программы с помощью технологий функционального программирования;
- использование «ленивых» вычислений.

Написание кода при функциональном программировании может заставить вас почувствовать себя водителем скоростного автомобиля: чтобы разогнаться, нет нужды знать, как работают основные узлы. Из главы 1 вы узнали, что подход ФП к написанию конкурентных приложений позволяет лучше решать проблемы, возникающие при написании таких приложений, чем, например, объектно-ориентированный подход. Ключевые понятия, такие как неизменяемые переменные и чистота, на любом ФП-языке означают: несмотря на то что написание конкурентных приложений остается далеко не простым делом, разработчики все же могут быть уверены — они не столкнутся с некоторыми традиционными ловушками параллельного программирования. Концепция ФП такова, что исключает такие проблемы, как состояние гонки и взаимные блокировки.

В данной главе мы более подробно рассмотрим основные принципы ФП, которые помогут нам пройти квест по написанию высококачественных конкурентных при-

ложений. Вы познакомитесь с основными принципами, узнаете, как они работают в C# (насколько это возможно) и в F# и как соответствуют шаблонам параллельного программирования.

В данной главе я предполагаю, что вы знакомы с основными принципами ФП. Если это не так, то обратитесь к приложению А для получения более подробной информации, необходимой для продолжения чтения книги. В конце главы вы узнаете, как использовать методы функционального программирования, позволяющие компоновать простые функции для решения сложных проблем, выполнять безопасное кэширование и предварительное вычисление данных в многопоточной среде, чтобы ускорить выполнение программы.

2.1. Использование компоновки функций для решения сложных задач

Компоновка функций — объединение их таким образом, что выходные данные одной из них становятся входными для следующей и это приводит к созданию новой функции. Такой процесс может продолжаться бесконечно; функции, объединенные в цепочки, образуют более мощные новые функции для решения сложных задач. Благодаря компоновке обеспечивается модуляризация, что позволяет упростить структуру программы.

Функциональная парадигма дает возможность создавать программы с простой структурой. Основная цель функциональной компоновки — обеспечить простой механизм построения понятного, поддерживаемого, многоразового и лаконичного кода. Кроме того, при компоновке функций без побочных эффектов код остается чистым, что сохраняет логику параллелизма. В принципе, конкурентные программы, основанные на компоновке функций, легче разрабатывать и они получаются менее запутанными, чем программы без такой компоновки.

Функциональная компоновка позволяет создавать и объединять наборы простых функций в единую массивную и более сложную функцию. Почему важно склеить код? Представьте, что вы решаете проблему сверху вниз. Начинаете с большой проблемы, а затем разбиваете ее на меньшие, пока в конце концов они не станут достаточно малыми, чтобы решить их напрямую. Результатом является набор малых решений, которые затем можно соединить, чтобы решить исходную большую проблему. Компоновка — это процесс склеивания больших решений из маленьких кусочков.

Представьте себе функциональную компоновку как конвейерную обработку, в том смысле что результирующее значение одной функции становится входным параметром для следующей функции. При таком сравнении проявляются следующие различия.

При конвейерной обработке выполняется последовательность операций, где вход каждой следующей функции будет выходом предыдущей.

Функциональная компоновка возвращает новую функцию, которая представляет собой сочетание из двух и более функций и не вызывается сразу (вход → функция → выход).

2.1.1. Функциональная компоновка в C#

Язык C# сам по себе не поддерживает функциональную компоновку, что создает семантические проблемы. Но такую функциональность можно легко ввести. Рассмотрим простой случай на C# (листинг 2.1) для определения двух функций с использованием лямбда-выражения.

Листинг 2.1. Функции высшего порядка `grindCoffee` и `brewCoffee` для класса `Espresso` на C#

```
Func<CoffeeBeans, CoffeeGround> grindCoffee = coffeeBeans
    => new CoffeeGround(coffeeBeans);
Func<CoffeeGround, Espresso> brewCoffee = coffeeGround
    => new Espresso(coffeeGround);
```

Функция высшего порядка `grindCoffee`
возвращает делегат `Func`, который
принимает `CoffeeBeans` в качестве аргумента,
а затем возвращает объект `CoffeeGround`

Функция высшего порядка `brewCoffee`
возвращает объект `Espresso` и принимает
в качестве параметра объект `coffeeGround`

Первая функция, `grindCoffee`, принимает в качестве параметра объект `coffeeBeans` и возвращает новый экземпляр класса `CoffeeGround`. Вторая функция, `brewCoffee`, принимает в качестве параметра объект `coffeeGround` и возвращает новый экземпляр `Espresso`. Цель функций — создать объект класса `Espresso`, объединив ингредиенты, получаемые в результате вычислений. Как скомпоновать эти функции? В C# есть возможность выполнять их последовательно, передавая результат первой функции во вторую по цепочке (листинг 2.2).

Листинг 2.2. Компоновка функций в C# (неудачный вариант)

```
CoffeeGround coffeeGround = grindCoffee(coffeeBeans);
Espresso espresso = brewCoffee(coffeeGround);

Espresso espresso = brewCoffee(grindCoffee(coffeeBeans));
```

Неудачный пример
компоновки функций,
где чтение данных
выполняется задом наперед

Сначала выполняется функция `grindCoffee`, которой передается параметр `coffeeBeans`, затем результат `coffeeGround` передается в функцию `brewCoffee`. Второй вариант, который даст тот же результат, — объединить выполнение `grindCoffee` и `brewCoffee`, что позволит реализовать основную идею функциональной компоновки. Но с точки зрения удобства чтения это неудачный шаблон, поскольку он заставляет читать код справа налево, что не является естественным (ни по-английски, ни по-русски. — Примеч. пер.). Было бы неплохо читать код логически слева направо.

Лучшим решением было бы создать параметризованный, специализированный метод расширения, который можно было бы использовать для компоновки любых двух функций с одним или несколькими параметризованными входными аргументами.

тами. В листинге 2.3 определена функция `Compose` и выполнен рефакторинг предыдущего примера. (Параметризованные аргументы выделены жирным шрифтом.)

Листинг 2.3. Функция компоновки в C#

```
static Func<A, C> Compose<A, B, C>(this Func<A, B> f, Func<B, C> g)
    => (n) => g(f(n));
```

```
Func<CoffeeBeans, Espresso> makeEspresso =
    grindCoffee.Compose(brewCoffee);
Espresso espresso = makeEspresso(coffeeBeans);
```

Компилятор F# требует, чтобы в функции использовался один и тот же тип для входных и выходных аргументов

Создает параметризованный метод расширения для любого параметризованного делегата `Func<A, B>`, который принимает в качестве входного аргумента параметризованный делегат `Func<B, C>` и возвращает объединенную функцию `Func<A, C>`

Как показано на рис. 2.1, функция высшего порядка `Compose` составляет цепочку из функций `grindCoffee` и `brewCoffee`, создавая новую функцию `makeEspresso`, которая принимает аргумент `coffeeBeans` и выполняет `brewCoffee` (`grindCoffee(coffeeBeans)`).

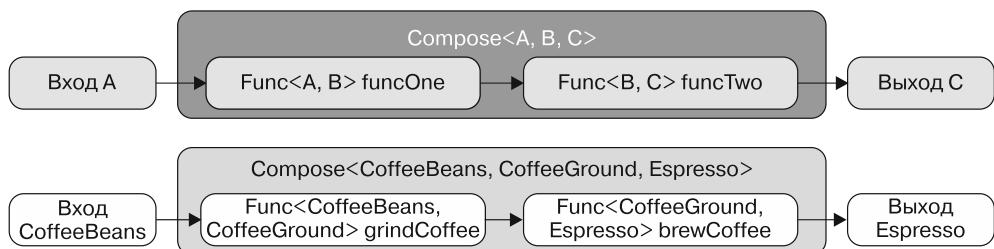


Рис. 2.1. Функциональная компоновка из функции `Func <CoffeeBeans, CoffeeGround> grindCoffee` для функции `Func <CoffeeGround, Espresso> brewCoffee`. Поскольку выходные данные функции `grindCoffee` соответствуют входным данным функции `brewCoffee`, они могут быть скомпонованы в новую функцию, которая отображает входные данные `CoffeeBeans` на выходные данные `Espresso`

В теле функции легко заметить строку, которая выглядит точно так же, как лямбда-выражение `makeEspresso`. Такой метод расширения инкапсулирует понятие скомпонованных функций. Идея заключается в том, чтобы создать функцию, которая возвращает результат использования внутренней функции `grindCoffee`, а затем применяет к данному результату внешнюю функцию `brewCoffee`. Это общепринятый математический шаблон, и его можно представить в записи как `brewCoffee` от `grindCoffee`, то есть функция `grindCoffee`, применяемая к `brewCoffee`. Задействуя методы расширения, можно легко создавать функции высшего порядка (higher-order functions, HOF)¹, чтобы повысить уровень абстракции и создать модульные функции многоразового использования.

¹ Функция высшего порядка (HOF) принимает одну или несколько функций в качестве входных данных и возвращает функцию в качестве результата.

Встроенная в язык компоновочная семантика, как это сделано в F#, помогает структурировать код в декларативном стиле. К сожалению, в C# аналогичного сложного решения нет. В исходном коде, прилагаемом к данной книге, вы найдете библиотеку с несколькими переопределениями методов расширения `Compose`, представляющими подобные полезные многоразовые решения.

2.1.2. Функциональная компоновка в F#

В языке F# функциональная компоновка поддерживается изначально. Фактически определение функции `compose` встроено в язык с помощью оператора `>>`. С помощью этого оператора F# можно создавать новые функции, комбинируя уже существующие.

Рассмотрим простой сценарий, в котором нужно каждый элемент списка увеличить на 4 и умножить на 3. В листинге 2.4 показано, как построить такую функцию с помощью и без помощи функциональной компоновки, так что мы можем сравнить оба подхода.

Листинг 2.4. Поддержка функциональной компоновки в F#

```

let add4 x = x + 4           ← Компилятор F# позволяет выводить типы аргументов
                             для каждой функции без явного обозначения

let multiplyBy3 x = x * 3    ← Компилятор F# проследит, чтобы функция использовала
                             один и тот же тип входных и выходных данных

let list = [0..10]             ← Определение диапазона чисел от 0 до 10. В F# можно описать
                             коллекцию, используя диапазон, обозначенный целыми числами,
                             разделенными оператором диапазона

let newList = List.map(fun x ->
  multiplyBy3(add4(x))) list  ← В F# можно применять операции HOF, используя
                             списковые выражения. Функция высшего порядка map
                             применяет одну и ту же проекцию функции к каждому
                             элементу заданного списка. В F# модули коллекций,
                             такие как List, Seq, Array и Set, принимают аргумент
                             коллекции как последний аргумент

let newList = list |>
  List.map(add4 >> multiplyBy3)  ← Применяет операции HOF, объединяющие функции add4
                                и multiplyBy3 с использованием функциональной композиции

```

В данном примере функции `add4` и `multiplyBy3` применяются к каждому элементу списка с помощью функции `map`, которая в F# является частью модуля `List`. `List.map` выступает эквивалентом статического метода `Select` в LINQ. Компоновка двух функций выполняется посредством последовательного семантического подхода, который вынуждает читать код неестественно, изнутри: `multiplyBy3 (add4 (x))`. Функциональная компоновка, при которой используется более идиоматический F#-код с инфиксным оператором `>>`, позволяет читать код слева направо, как в книге, и результат получается гораздо чище, короче и понятнее.

Другим способом получить функциональную компоновку с простой и модульной семантикой кода является применение технологии, называемой замыканиями.

2.2. Использование замыканий для упрощения функционального мышления

Замыкания предназначены для упрощения функционального мышления; они позволяют среди выполнения управлять состоянием, избавляя разработчика от лишних сложностей. Замыкание — функция первого класса со свободными переменными, ограниченными лексической средой. За всеми этими умными словами скрывается простая концепция: замыкания представляют собой более удобный способ представления функциям доступа к локальному состоянию и передачи данных фоновым операциям. Замыкания — это специальные функции, обеспечивающие неявное связывание для всех нелокальных переменных (также называемых *свободными переменными* или *внешними локальными переменными*). Более того, замыкание обеспечивает для функции доступ к одной или нескольким нелокальным переменным даже при вызове вне пределов непосредственного лексического контекста функции, а тело такой специальной функции может переносить эти *внешние локальные переменные* как единый объект, определенный в его области замыкания. Что еще более важно, замыкание инкапсулирует поведение и передает его, подобно любому другому объекту, предоставляя доступ к контексту, в котором было создано замыкание, для чтения и обновления данных значений.

Свободные переменные и замыкания

Свободной переменной называют переменную, на которую есть ссылка в теле функции, но которая не является ни локальной переменной этой функции, ни ее параметром. Цель замыканий заключается в том, чтобы сделать такие переменные доступными при выполнении функции, даже если исходные переменные выходят за пределы ее области действия.

В ФП, как и в любом другом языке программирования, который поддерживает функции высшего порядка, без поддержки замыканий область действия данных может создавать проблемы и неудобства. Однако в C# и F# компилятор задействует замыкания для увеличения и расширения области действия переменных. Таким образом, данные являются доступными и видимыми в текущем контексте, как показано на рис. 2.2.

В C# замыкания стали доступны, начиная с .NET 2.0; однако использование и определение замыканий упростилось после появления в .NET гармоничного сочетания лямбда-выражений и анонимного метода.

В этом разделе примеры кода написаны на C#, но те же концепции и методы применимы и для F#. В листинге 2.5 определено замыкание с использованием анонимного метода.

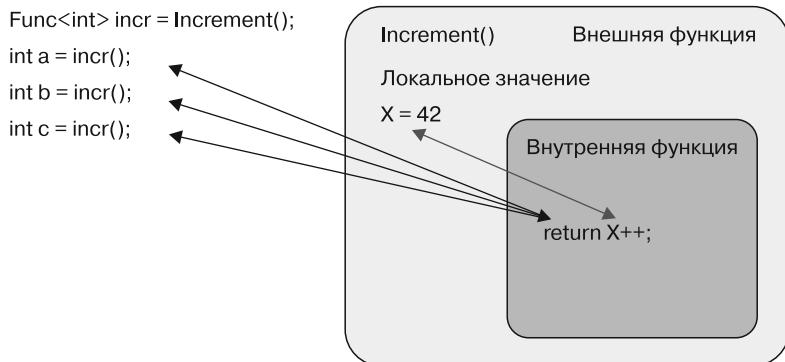


Рис. 2.2. В этом примере использования замыкания локальная переменная X , находящаяся в теле внешней функции `Increment`, представлена в виде функции (`Func<int>`), порожденной внутренней функцией. Важным моментом является тип значения, возвращаемого функцией `Increment`, которая представляет собой функцию, захватывающую замкнутую переменную X , а не саму переменную. Каждый раз, когда запускается ссылка на функцию `incr`, значение захваченной переменной X увеличивается

Листинг 2.5. Замыкание, определенное в C# с использованием анонимного метода

```

string freeVariable = "Я простая переменная"; ← Свободная переменная
Func<string, string> lambda = value => freeVariable + " " + value; ←
                                            Анонимная функция, ссылающаяся на свободную переменную
    
```

В этом примере анонимная функция `lambda` ссылается на свободную переменную `freeVariable`, которая находится в ее области замыкания. Замыкание дает функции доступ к окружающему ее состоянию (в данном случае к переменной `freeVariable`), делая код более ясным и легко читаемым. Репликация одной и той же функциональности без замыкания, очевидно, означает создание класса, который будет использовать функция (и который «знает» о локальной переменной), и передачу данного класса в качестве аргумента. В такой ситуации замыкание помогает среде выполнения управлять состоянием, избегая лишнего и ненужного повторного создания полей. Это одно из преимуществ замыкания: его можно задействовать как переносимый механизм выполнения операций для передачи дополнительного контекста в HOF. Неудивительно, что замыкания часто применяются в сочетании с LINQ. Как видим, замыкания — положительный побочный эффект лямбда-выражений и отличный программный трюк в копилку вашего инструментария.

2.2.1. Захваченные переменные в замыканиях с лямбда-выражениями

Эффективность замыканий возрастает, когда одна и та же переменная может применяться даже в том случае, если она выходит за пределы области действия замыкания. Поскольку переменная была захвачена, она не пропадает при сборке мусора. Преиму-

щество использования замыканий заключается в том, что можно создать переменную уровня метода, которая обычно применяется для реализации кэширования памяти с целью повышения вычислительной производительности. Такие функциональные технологии называются *мемоизацией* и *функциональным предварительным вычислением* и будут рассмотрены далее в этой главе.

В листинге 2.6 использована модель программирования событий (event programming model, EPM) для асинхронной загрузки изображения. Этот пример иллюстрирует работу замыканий с захваченными переменными. Когда загрузка завершается, процесс продолжает обновление пользовательского интерфейса клиентского приложения. В данной реализации задействован асинхронный семантический вызов API. Когда запрос завершается, зарегистрированное событие `DownloadDataCompleted` запускает и выполняет оставшуюся логику.

Листинг 2.6. Регистратор событий с лямбда-выражением, захватывающим локальную переменную

```
void UpdateImage(string url)
{
    System.Windows.Controls.Image image = img; ← Захват экземпляра локального
                                                элемента управления изображением
                                                и запись его в переменную изображения

    var client = new WebClient();
    client.DownloadDataCompleted += (o, e) => ← Регистрация события DownloadDataCompleted
                                                с использованием встроенного
                                                лямбда-выражения

    {
        if (image != null)
            using (var ms = new MemoryStream(e.Result))
            {
                var imageConverter = new ImageSourceConverter();
                image.Source = (ImageSource)
                    => imageConverter.ConvertFrom(ms);
            }
    };
    client.DownloadDataAsync(new Uri(url)); ← Асинхронный запуск DownloadDataAsync
}
}
```

Сначала мы получаем ссылку на элемент управления изображением с именем `img`. Затем с помощью лямбда-выражения регистрируем обратный вызов обработчика события `DownloadDataCompleted`, чтобы обработать его, когда `DownloadDataAsync` завершится. Благодаря замыканию код, находящийся внутри лямбда-блока, может получить непосредственный доступ к состоянию, находящемуся вне его области видимости. Данный доступ позволяет проверить состояние указателя на изображение и, если он не равен `null`, обновить пользовательский интерфейс.

Это довольно простой процесс, но временная шкала вносит интересные корректификаты в его поведение. Метод является асинхронным, а значит, к тому моменту, как сервис возвратит данные, а функция обратного вызова обновит `image`, выполнение метода уже завершится.

Если метод завершился, будет ли локальная переменная `image` доступна? И как тогда станет обновляться изображение? Ответ на этот вопрос — *захваченная переменная*. Лямбда-выражение захватывает локальную переменную `image`, которая,

соответственно, остается доступной, хотя в обычной ситуации она была бы уничтожена. Как видно на данном примере, захваченные переменные следует рассматривать как снимок значений переменных в момент создания замыкания. Если бы мы решили построить такой же процесс без этой захваченной переменной, то нам понадобилась бы переменная уровня класса для хранения значения изображения.

ПРИМЕЧАНИЕ

Переменная, захваченная лямбда-выражением, содержит значение на момент вычисления, а не на момент захвата. Экземпляры и статические переменные могут использоваться и изменяться в теле лямбда-выражения без ограничений.

Чтобы это доказать, проанализируем, что произойдет, если добавить в конце листинга 2.6 еще одну строку, заменив ссылку на изображение нулевым указателем (выделен жирным шрифтом) (листинг 2.7).

Листинг 2.7. Доказательство времени вычисления захваченной переменной

```
void UpdateImage(string url)
{
    System.Windows.Controls.Image image = img;

    var client = new WebClient();
    client.DownloadDataCompleted += (o, e) =>
    {
        if (image != null) {
            using (var ms = new MemoryStream(e.Result))
            {
                var imageConverter = new ImageSourceConverter();
                image.Source = (ImageSource)
                ➔ imageConverter.ConvertFrom(ms);
            }
        }
    };
    client.DownloadDataAsync(new Uri(url));
    image = null;           ➙
}
```

Переменная `image` обнуляется; следовательно, она будет удалена, когда закончится время ее действия. Этот метод завершается до выполнения обратного вызова, поскольку асинхронная функция `DownloadDataAsync` не выполняет блокирование, что приводит к нежелательному поведению

После запуска измененной программы изображение в пользовательском интерфейсе не будет обновляться, потому что указателю присваивается значение `null` перед выполнением тела лямбда-выражения. Несмотря на то что в момент захвата изображение имело ненулевое значение, в момент выполнения кода оно равно `null`. Время жизни захваченных переменных продолжается до тех пор, пока все замыкания, ссылающиеся на них, не будут готовы к сбору мусора.

В F# не существует понятия нулевых объектов, поэтому в данном языке выполнение таких нежелательных сценариев невозможно.

2.2.2. Замыкания в многопоточной среде

Проанализируем сценарий использования замыканий для предоставления данных задаче, которая часто выполняется в другом, не основном, потоке. В ФП замыкание обычно применяется для управления изменяемым состоянием, чтобы ограничить и изолировать область действия изменяемых структур и обеспечить потокобезопасный доступ. Это хорошо подходит для многопоточной среды.

В листинге 2.8 лямбда-выражение вызывает метод `Console.WriteLine` из только что созданного объекта `Task` библиотеки TPL (`System.Threading.Tasks.Task`). Когда начинается выполнение этой задачи, лямбда-выражение строит замыкание, охватывающее локальную переменную `iteration`, которая передается в качестве аргумента методу, выполняемому в другом потоке. В таком случае компилятор автоматически генерирует анонимный класс с данной переменной в качестве открытого свойства.

Листинг 2.8. Замыкание, захватывающее переменные в многопоточной среде

```
for (int iteration = 1; iteration < 10; iteration++)
{
    Task.Factory.StartNew(() => Console.WriteLine("{0} - {1}",
        Thread.CurrentThread.ManagedThreadId, iteration));
}
```

Замыкания иногда приводят к странному поведению программы. Теоретически эта программа должна работать: ожидается, что она напечатает цифры от 1 до 10. Но на практике подобного не происходит; программа напечатает десять раз число 10, потому что одна и та же переменная применяется в нескольких лямбда-выражениях и эти анонимные функции используют значение данной переменной совместно.

Проанализируем другой пример. В листинге 2.9 данные передаются в два разных потока с задействованием лямбда-выражений.

Листинг 2.9. Странное поведение программы при использовании замыканий в многопоточном коде

```
Action<int> displayNumber = n => Console.WriteLine(n);
int i = 5;
Task taskOne = Task.Factory.StartNew(() => displayNumber(i));
i = 7;
Task taskTwo = Task.Factory.StartNew(() => displayNumber(i));

Task.WaitAll(taskOne, taskTwo);
```

Даже если первое лямбда-выражение захватывает переменную `i` до того, как ее значение изменится, оба потока напечатают число 7, потому что переменная `i` изменяется до запуска обоих потоков. Причиной этой тонкой проблемы является изменчивый характер языка C#. Когда замыкание захватывает изменяемую переменную посредством лямбда-выражения, лямбда захватывает не текущее значение переменной, а ссылку на нее. Следовательно, если задача запускается после того, как

изменится значение переменной, на которую указывает ссылка, то будет получено значение, поступившее в память последним, а не то, что было присвоено переменной в момент ее захвата.

Именно по этой причине не стоит вручную кодировать параллельный цикл, надо искать другие решения. Цикл `Parallel.For` из библиотеки TPL устраняет данную ошибку. Одним из возможных решений на C# является создание и захват новой временной переменной для каждого объекта `Task`. Таким образом, при создании новой переменной для нее выделяется новое место в куче и исходное значение сохраняется. В функциональных языках такое сложное и замысловатое решение неприменимо. Посмотрим, как аналогичный сценарий реализуется в F# (листинг 2.10).

Листинг 2.10. Захват переменных с помощью замыкания в многопоточной среде на F#

```
let tasks = Array.zeroCreate<Task> 10

for index = 1 to 10 do
    tasks.[index - 1] <- Task.Factory.StartNew(fun () ->
    ▶ Console.WriteLine index)
```

Запустив эту версию кода, получим ожидаемый результат: программа напечатает числа от 1 до 10. Объяснение состоит в том, что в F# процедурный цикл `for` обрабатывается иначе, чем в C#. Вместо использования изменяемой переменной и обновления ее значения на каждой итерации компилятор F# создает новое неизменяемое значение для каждой итерации, каждый раз со своим местом в памяти. Результатом такого функционального поведения, при котором отдается предпочтение неизменяемым типам, является то, что лямбда захватывает ссылку на неизменяемое значение, которое никогда не изменится.

В многопоточных средах к замыканиям обычно прибегают из-за простоты захвата и передачи переменных в разных контекстах, что требует дополнительного обдумывания. В листинге 2.11 показано, как библиотека .NET TPL может использовать замыкания для выполнения нескольких потоков с применением API `Parallel.Invoke`.

Листинг 2.11. Замыкания, захватывающие переменные в многопоточной среде

```
public void ProcessImage(Bitmap image) {
    byte[] array = image.ToByteArray(ImageFormat.Bmp);
    Parallel.Invoke(
        () => ProcessArray(array, 0, array.Length / 2),
        () => ProcessArray(array, array.Length / 2, array.Length));
}
```

Функции, которые преобразуют изображение в байтовый массив

Функции для параллельной обработки байтового массива, разделенного пополам

В этом примере `Parallel.Invoke` создает две независимые задачи, каждая из которых запускает метод `ProcessArray` для той части массива `array`, чья переменная захвачена и замкнута в лямбда-выражениях.

СОВЕТ

Помните, что компилятор обрабатывает замыкания, выделяя для них объект, который инкапсулирует функцию и ее среду. Поэтому замыкания более затратны с точки зрения распределения памяти, чем обычные функции, и их вызов происходит медленнее.

В контексте параллелизма задач помните о переменных, захваченных в замыканиях: поскольку замыкания захватывают не фактическое значение переменной, а ссылку на переменную, вы можете открыть разделяемый доступ к объекту, значение которого неизвестно. Замыкания — это мощная технология, которую можно использовать для реализации шаблонов, повышающих производительность программ.

2.3. Технология мемоизации и кэширования для ускорения программы

Мемоизация, также известная как *табличное сохранение данных*, — это технология ФП, целью которой является повышение производительности приложения. Ускорение работы программы достигается путем кэширования результатов функции и исключения ненужных дополнительных вычислительных издержек, возникающих из-за многократного выполнения одних и тех же вычислений. Подобное становится возможным благодаря тому, что мемоизация позволяет обойтись без выполнения затратных вызовов функций. Результат предыдущих вычислений с теми же аргументами сохраняется (как показано на рис. 2.3) и выдается, когда понадобится снова получить его для тех же аргументов. Мемоизированная функция сохраняет в памяти результат вычислений, так что при последующих вызовах этот результат можно выдать немедленно.

На первый взгляд такая концепция может показаться сложной, но это простой в реализации метод. При мемоизации используются замыкания, упрощающие преобразование функции в структуру данных, которая облегчает доступ к локальной переменной. Замыкание служит оберткой для каждого вызова мемоизированной функции. Назначением такой локальной переменной — как правило, таблицы поиска — является сохранение результатов внутренней функции в виде значения и применение аргументов, переданных в эту функцию, в качестве ссылочного ключа.

Технология мемоизации хорошо вписывается в многопоточную среду, обеспечивая огромный прирост производительности. Основное преимущество сказывается тогда, когда одна и та же функция многократно применяется к одним и тем же аргументам: ведь с точки зрения вычислительной нагрузки на процессор выполнение функции более затратно, чем доступ к соответствующей структуре данных. Например, чтобы применить к изображению цветовой фильтр, удобно использовать несколько параллельных потоков. Каждый из них обращается к части изображения и изменяет содержащиеся там пиксели. Но цвет фильтра может применяться к набору пикселов, имеющих одинаковые значения. В таком случае, если вычисление будет выдавать один и тот же результат, зачем выполнять его каждый раз заново?

Вместо этого результат может быть кэширован с использованием мемоизации, потоки избавятся от выполнения ненужной работы, и обработка изображения завершится быстрее.

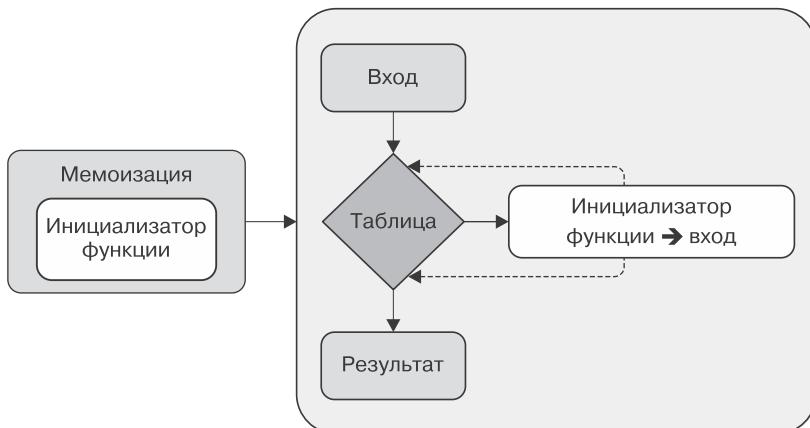


Рис. 2.3. Мемоизация — технология кэширования значений функции, гарантирующая выполнение только одного вычисления. Когда входное значение передается в мемоизированную функцию, внутреннее табличное хранилище проверяет, существует ли результат, связанный с этими входными данными, и если он существует, то возвращается немедленно. В противном случае инициализатор функции запускает вычисление, а затем обновляет внутреннее табличное хранилище и возвращает результат. В следующий раз, когда в мемоизированную функцию передадутся те же входные значения, в табличном хранилище уже будет существовать соответствующий им связанный результат и вычисление будет пропущено

Кэш

Кэш — компонент для хранения данных, так что последующие запросы этих данных могут быть обработаны быстрее; данные, хранящиеся в кэше, могут быть результатом более ранних вычислений или дубликатами данных, хранящихся в другом месте.

В листинге 2.12 показана базовая реализация мемоизированной функции на C#.

Сначала определяется функция `Memoize`, внутри которой используется параметризованная коллекция `Dictionary` в качестве табличной переменной для кэширования. Замыкание захватывает локальную переменную, так что к ней можно получить доступ как от делегата, указывающего на замыкание, так и от внешней функции. При вызове функции высшего порядка сначала сопоставляются входные данные функции с данными, хранящимися в кэше, чтобы установить, был ли такой параметр уже кэширован. Если ключ, соответствующий этому параметру, существует, то таблица кэша возвращает результат. Если же такого ключа нет, то сначала вычисляется значение функции с этим параметром. Затем параметр и соответствующий ему результат добавляются в таблицу кэша, и, наконец, возвращается результат. Важно

отметить, что мемоизация является функцией высшего порядка, так как принимает функцию в качестве входного аргумента и возвращает функцию на выходе.

Листинг 2.12. Простой пример, поясняющий, как работает мемоизация

```
Перебор экземпляров изменяемой коллекции
Dictionary для хранения и поиска значений
static Func<T, R> Memoize<T, R>(Func<T, R> func) <-
where T : IComparable
{
    Dictionary<T, R> cache = new Dictionary<T, R>();
    return arg => {
        if (cache.ContainsKey(arg))
            return cache[arg];
        return (cache[arg] = func(arg));
    };
}
Если ключ не существует, то значение вычисляется,
сохраняется в таблице и возвращается в качестве результата
```

Параметризованная функция `Memoize`, которая требует, чтобы параметризованный тип `T` был сравнимым, потому что это значение применяется для поиска

Использование лямбда-выражения, которое захватывает локальную таблицу с помощью замыкания

Верификация того, что значение аргумента `arg` было вычислено и сохраняется в кэше

Если ключ, переданный в качестве аргумента, существует в таблице, то ассоциированное с ним значение возвращается как результат

СОВЕТ

Поиск по словарю выполняется за постоянное время, но хеш-функция, используемая в словаре, в определенных обстоятельствах может работать медленно. Это относится к строкам, так как время, необходимое для хеширования строки, пропорционально ее длине. В силу чего в определенных сценариях немемоизированные функции работают лучше, чем мемоизированные. Я рекомендую профилировать код, чтобы решить, нужна ли оптимизация и повысится ли производительность в результате мемоизации.

В листинге 2.13 представлена эквивалентная функция `memoize`, реализованная на F#.

Листинг 2.13. Функция `memoize` на F#

```
let memoize func =
    let table = Dictionary<_,>_()
    fun x -> if table.ContainsKey(x) then table.[x]
               else
                   let result = func x
                   table.[x] <- result
                   result
```

Это простой пример с применением определенной ранее функции `memoize`. В листинге 2.14 функция `Greeting` возвращает строку с приветствием, в которое

вставлено имя, переданное функции в качестве аргумента. В сообщении также указано время вызова функции, чтобы отслеживать момент ее выполнения. Из соображений наглядности в коде использована двухсекундная задержка между вызовами функции.

Листинг 2.14. Функция приветствия на C#

```
public static string Greeting(string name)
{
    return $"Warm greetings {name}, the time is
→ {DateTime.Now.ToString("hh:mm:ss")}";
}

Console.WriteLine(Greeting ("Richard"));
System.Threading.Thread.Sleep(2000);
Console.WriteLine(Greeting ("Paul"));
System.Threading.Thread.Sleep(2000);
Console.WriteLine(Greeting ("Richard"));

// вывод
Warm greetings Richard, the time is 10:55:34
Warm greetings Paul, the time is 10:55:36
Warm greetings Richard, the time is 10:55:38
```

В следующем коде выводятся те же сообщения, но с использованием мемоизированной версии функции `Greeting` (листинг 2.15).

Листинг 2.15. Пример приветствия с использованием мемоизированной функции

```
var greetingMemoize = Memoize<string, string>(Greeting); ← Memoize — это функция высшего порядка, поэтому в качестве аргумента ей передается функция, что ограничивает сигнатуру предыдущей функции.
System.Threading.Thread.Sleep(2000); ← Таким образом, memoized может заменить первоначальную,
Console.WriteLine(greetingMemoize ("Richard"));           используя функционал кэширования

// вывод
Warm greetings Richard, the time is 10:57:21 ← Время выполнения функции Greeting не меняется,
Warm greetings Paul, the time is 10:57:23           так как вычисление выполняется только
Warm greetings Richard, the time is 10:57:21 ← один раз, а затем результат запоминается
```

Результат показывает, что первые два вызова выполнились в разное время, как и ожидалось. Но что произошло при третьем вызове? Почему третий вызов функции возвращает сообщение с тем же временем, что и первый? Причина этого — мемоизация.

Первый и третий раз функция `greetingMemoize ("Richard")` была вызвана с одинаковым аргументом. При первом вызове результаты были кэшированы функцией `greetingMemoize`. Результат третьего вызова функции получен не вследствие ее вы-

полнения — он является сохраненным результатом функции с тем же аргументом, и поэтому время в сообщении совпадает.

Так работает мемоизация. Задача мемоизированной функции — найти во внутренней таблице переданный ей аргумент. Если входное значение будет найдено, то функция возвращает ранее вычисленный результат. В противном случае она сохраняет результат в таблице.

2.4. Применение мемоизации для создания быстрого веб-робота

А теперь мы реализуем более интересный пример, применяя то, что узнали в предыдущем разделе. В этом примере мы построим веб-робота, который извлекает и печатает в консоль название страницы каждого посещенного сайта. В листинге 2.16 выполняется код без мемоизации. Затем мы выполним ту же программу повторно, используя технологию мемоизации, и сравним результаты. В итоге мы загрузим содержимое нескольких сайтов, объединив параллельное выполнение и мемоизацию.

Листинг 2.16. Веб-искатель в C #

```
public static IEnumerable<string> WebCrawler(string url) { ←
    string content = GetWebContent(url);
    yield return content;

    foreach (string item in AnalyzeHtmlContent(content))
        yield return GetWebContent(item);
}

static string GetWebContent(string url) { ←
    using (var wc = new WebClient())
        return wc.DownloadString(new Uri(url));
}

static readonly Regex regexLink =
    new Regex(@"(?=<a href='|'>)https?://.*?(?=\1)");
← Извлечение из содержимого веб-страницы ссылок на другие страницы сайта

static IEnumerable<string> AnalyzeHtmlContent(string text) { ←
    foreach (var url in regexLink.Matches(text))
        yield return url.ToString();
}

static readonly Regex regexTitle =
    new Regex("<title>(?<title>.*?)</title>", RegexOptions.Compiled);

static string ExtractWebpageTitle(string textPage) { ←
    if (regexTitle.IsMatch(textPage))
        return regexTitle.Match(textPage).Groups["title"].Value;
    return "No Page Title Found!";
}
```

Функция, которая рекурсивно извлекает и анализирует содержимое главной веб-страницы сайта и связанных страниц

Загрузка содержимого веб-страницы в строковом формате

Извлечение названия страницы сайта

Функция `WebCrawler` загружает содержимое веб-страницы, расположенной по URL-адресу, который был ей передан в качестве аргумента, вызывая метод `GetWebContent`. Затем эта функция анализирует загруженный контент и извлекает из него гиперссылки, содержащиеся на веб-странице, которые отправляются обратно в начальную функцию на обработку. Такая операция повторяется для каждой гиперссылки. В листинге 2.17 представлен веб-робот в действии.

Листинг 2.17. Выполнение веб-робота

```

List<string> urls = new List<string> { ← Инициализация списка
    @"http://www.google.com",
    @"http://www.microsoft.com",
    @"http://www.bing.com",
    @"http://www.google.com"
};

var tabPageTitles = from url in urls ← Использование LINQ-выражения для анализа
    from pageContent in WebCrawler(url)
    select ExtractWebPageTitle(pageContent);

foreach (var tabPageTitle in tabPageTitles)
    Console.WriteLine(tabPageTitle);

// ВЫВОД
Starting Web Crawler for http://www.google.com...
Google
Google Images
...
Web Crawler completed for http://www.google.com in 5759ms
Starting Web Crawler for http://www.microsoft.com...
Microsoft Corporation
Microsoft - Official Home Page
Web Crawler completed for http://www.microsoft.com in 412ms
Starting Web Crawler for http://www.bing.com...
Bing
Msn
...
Web Crawler completed for http://www.bing.com in 6203ms
Starting Web Crawler for http://www.google.com...
Google
Google Images
...
Web Crawler completed for http://www.google.com in 5814ms

```

Для перебора веб-роботом коллекции заданных URL-адресов мы использовали язык запросов LINQ (Language Integrated Query). Когда выражение запроса выполняется в цикле `foreach`, функция `ExtractWebPageTitle` извлекает заголовок из содержимого каждой страницы и печатает его в консоли. Из-за межсетевого характера этой операции функция `GetWebContent` тратит определенное время на завершение загрузки. Одной из проблем в предыдущей реализации кода являются дубликаты

гиперссылок. На веб-страницах часто встречаются повторяющиеся гиперссылки, которые в данном примере приводят к ненужным повторным операциям загрузки. Лучшим решением будет мемоизовать функцию `WebCrawler` (листинг 2.18).

Листинг 2.18. Выполнение веб-робота с использованием мемоизации

```
static Func<string, IEnumerable<string>> WebCrawlerMemoized = 
    Memoize<string, IEnumerable<string>>(WebCrawler);
```

Мемоизованная версия
функции `WebCrawler`

```
var webPageTitles = from url in urls
                    from pageContent in WebCrawlerMemoized(url)
                    select ExtractWebPageTitle(pageContent);
```

Использование
LINQ-выражения
в сочетании
с функцией `memoize`
для анализа веб-страниц

```
foreach (var webPageTitle in webPageTitles)
    Console.WriteLine(webPageTitle);
```

```
// ВЫВОД
Starting Web Crawler for http://www.google.com...
Google
Google Images
...
Web Crawler completed for http://www.google.com in 5801ms
Starting Web Crawler for http://www.microsoft.com...
Microsoft Corporation
Microsoft - Official Home Page
Web Crawler completed for http://www.microsoft.com in 4398ms
Starting Web Crawler for http://www.bing.com...
Bing
Msn
...
Web Crawler completed for http://www.bing.com in 6171ms
Starting Web Crawler for http://www.google.com...
Google
Google Images
...
Web Crawler completed for http://www.google.com in 02ms
```

В этом примере мы реализовали функцию высшего порядка `WebCrawlerMemoized`, которая представляет собой мемоизованную версию функции `WebCrawler`. Как подтверждает вывод, мемоизированная версия кода работает быстрее. В сущности, на то, чтобы второй раз извлечь содержимое с веб-страницы `www.google.com`, ушло всего 2 мс, а не более 5 с, как было без мемоизации.

Для дальнейшего улучшения кода нам потребуется параллельная загрузка веб-страниц. К счастью, поскольку мы обрабатываем запросы с помощью LINQ, для использования нескольких потоков понадобится лишь незначительно изменить код. После выпуска фреймворка .NET 4.0 в LINQ появился метод расширения `AsParallel()`, позволяющий задействовать параллельную версию LINQ (PLINQ). Назначение PLINQ – реализация параллелизма данных; оба эти вопроса будут рассмотрены в главе 4.

Технологии LINQ и PLINQ были разработаны с применением концепций функционального программирования. Особое внимание в них уделено соблюдению стиля декларативного программирования. Это достижимо, поскольку функциональная парадигма имеет тенденцию повышать уровень абстракции по сравнению с другими программными парадигмами. Абстракция позволяет писать код без необходимости знать детали реализации базовой библиотеки, как показано в листинге 2.19.

Листинг 2.19. Запрос веб-робота с использованием PLINQ

```
var webPageTitles = from url in urls.AsParallel()
                    from pageContent in WebCrawlerMemoized(url)
                    select ExtractWebPageTitle(pageContent);
```

Реализация метода расширения, который позволяет LINQ использовать несколько потоков для обработки запроса

Язык запросов PLINQ прост в реализации и способен обеспечить значительный прирост производительности. Здесь показан только один метод — метод расширения `AsParallel`, — но их гораздо больше.

Перед запуском программы нам нужно выполнить еще один рефакторинг — использовать кэши. Поскольку кэши должны быть доступны для всех потоков, их обычно делают статическими. Благодаря внедрению параллелизма становится возможным одновременное обращение нескольких потоков к мемоизированной функции, что может привести к состоянию гонки из-за того, что основная структура данных по своей сути является изменяемой. Проблема состояния гонки обсуждалась в предыдущей главе. К счастью, ее легко решить, как показано в листинге 2.20.

Листинг 2.20. Функция мемоизации в потоковом режиме

```
public Func<T, R> MemoizeThreadSafe<T, R>(Func<T, R> func)
    where T : IComparable
{
    ConcurrentDictionary<T, R> cache = new ConcurrentDictionary<T, R>();
    return arg => cache.GetOrAdd(arg, a => func(a));
}
```

Потокобезопасная мемоизация с использованием конкурентной коллекции `ConcurrentDictionary`

```
public Func<string, IEnumerable<string>> WebCrawlerMemoizedThreadSafe =
    MemoizeThreadSafe<string, IEnumerable<string>>(WebCrawler);
```

```
var webPageTitles =
    from url in urls.AsParallel()
    from pageContent in WebCrawlerMemoizedThreadSafe(url)
    select ExtractWebPageTitle(pageContent);
```

Параллельный анализ веб-страниц с помощью выражения PLINQ

Быстрое решение заключается в замене коллекции `Dictionary` на эквивалентную потокобезопасную версию `ConcurrentDictionary`. Примечательно, что такой рефак-

торинг требует меньше кода. Затем мы реализуем потокобезопасную мемоизованную версию функции `GetWebContent`, которая используется для LINQ-выражения. Теперь веб-робот может выполняться параллельно. При обработке страниц из данного примера двухъядерный компьютер затратил на анализ менее 7 с, в отличие от 18 с в начальной реализации. Обновленный код не только быстрее работает — он также задействует меньше операций сетевого ввода-вывода.

2.5. «Ленивая» мемоизация для повышения производительности

В предыдущем примере веб-робот позволял нескольким параллельным потокам получать доступ к мемоизированной функции с минимальными издержками. Но он не препятствовал многократному выполнению инициализатора функции `func(a)` для одного и того же значения при выполнении выражения. Это может показаться мелочью, но в высококонкурентных приложениях значение данного явления может возрасти многократно (в частности, если инициализация объекта очень затратна). Решение заключается в том, чтобы добавить в кэш объект, который не инициализируется сам, но является функцией, инициализирующей элемент по требованию. Значение результата, полученное от функции-инициализатора, можно обернуть в тип `Lazy` (в листинге 2.21 это место выделено жирным шрифтом). В листинге представлено решение мемоизации, которое представляет собой идеальный вариант с точки зрения потокобезопасности и производительности, без дублирования инициализации элемента кэша.

Листинг 2.21. Потокобезопасная функция мемоизации с надежным «ленивым» выполнением

```
static Func<T, R> MemoizeLazyThreadSafe<T, R>(Func<T, R> func)
    where T : IComparable
{
    ConcurrentDictionary<T, Lazy<R>> cache =
        new ConcurrentDictionary<T, Lazy<R>>(); ←
    return arg => cache.GetOrAdd(arg, a =>
        new Lazy<R>(() => func(a))).Value; }
```

Использование потокобезопасной мемоизации с «ленивым» выполнением

Согласно документации Microsoft, метод `GetOrAdd` не препятствует многократному вызову функции `func` для одного и того же аргумента, но гарантирует, что результат только одного «выполнения функции» будет добавлен в коллекцию. Например, может существовать несколько потоков, конкурентно проверяющих кэш перед тем, как туда будет добавлено кэшированное значение. Кроме того, нельзя гарантировать потокобезопасное выполнение функции `func(a)`. Без такой гарантии в многопоточной среде возможна ситуация, когда несколько потоков будут одновременно обращаться к одной и той же функции, то есть функция `func(a)` также должна быть потокобезопасной. Предлагаемое решение заключается в том, чтобы, избегая примитивных блокировок, использовать конструкцию `Lazy<T>` из

.NET 4.0. Это решение обеспечивает полную безопасность потоков независимо от реализации функции `func` и гарантирует, что функция будет выполняться только один раз.

Подводные камни в функции мемоизации. Использование вариантов мемоизации, представленных в предыдущих примерах кода, является несколько наивным подходом. Хранение данных в простом словаре работает, но это не долгосрочное решение. Объем словаря не ограничен; соответственно, его элементы никогда не удаляются из памяти, а только добавляются в нее, что в какой-то момент может привести к утечкам памяти. Для всех подобных проблем существуют решения. Один из вариантов — реализовать функцию мемоизации, которая использует тип `WeakReference` для хранения результатов, что позволяет собирать результаты перед запуском сборщика мусора (*garbage collector, GC*). После того как во фреймворке .NET 4.0 появилась коллекция `ConditionalWeakDictionary`, эта реализация стала очень простой: словарь принимает в качестве ключа экземпляр типа, который хранится как *слабая ссылка*. Связанные значения сохраняются до тех пор, пока сохраняется ключ. Как только ключ будет утилизирован в процессе сборки мусора, ссылка на данные удаляется и данные также становятся доступными для сборки мусора.

Слабые ссылки — важный механизм для обработки ссылок на управляемые объекты. Обычная ссылка на объект (также известная как сильная ссылка) имеет детерминированное поведение: пока вы ссылаетесь на объект, GC не станет его удалять и он, соответственно, будет продолжать существовать. Но в определенных сценариях желательно сохранить невидимую строку, прикрепленную к объекту, не мешая GC освободить память, занимаемую этим объектом. Если при сборке мусора память освободилась, то такая строка станет неприкрепленной, что можно обнаружить. Если сборка мусора еще не затронула объект, то можно извлечь строку и получить сильную ссылку на объект, которую затем использовать снова. Такая возможность полезна для автоматического управления кэшем, так как позволяет сохранять слабые ссылки на недавно задействованные объекты, не мешая утилизировать сами объекты и, соответственно, оптимизировать ресурсы памяти.

Другой вариант — использовать политику истечения срока действия кэша и сохранять для каждого результата временную метку, определяющую время, в течение которого сохраняется элемент. В таком случае нужно определить постоянное время, по истечении которого элементы аннулируются. Когда это время заканчивается, элемент удаляется из коллекции. В коде для скачивания, прилагаемом к данной книге, содержатся обе реализации.

СОВЕТ

Рекомендуемым методом является применение мемоизации только тогда, когда затраты на выполнение функции выше, чем на хранение всех результатов, вычисляемых во время ее работы. Прежде чем принимать окончательное решение, проведите бенчмаркинг кода с мемоизацией и без нее, используя различные диапазоны значений.

2.6. Эффективная конкурентная упреждающая обработка для уменьшения издержек на затратные вычисления

Упреждающая обработка (предварительная компиляция) является веской причиной использования конкурентности. Упреждающая обработка — это шаблон ФП, в котором вычисления выполняются до того, как будет запущен алгоритм, сразу, как только станут доступны все входные данные функции. Идея параллельной упреждающей обработки заключается в том, чтобы сократить издержки на затратные вычисления, повысить производительность и отзывчивость программы. Эта технология легко применима при параллельных вычислениях, так как на многоядерном оборудовании можно выполнять предварительные вычисления сразу нескольких операций, порождая конкурентно выполняемую задачу, так что данные будут готовы к чтению без задержки.

Предположим, что у нас есть длинный список входных слов и нужно выполнить функцию, которая находит наилучшее нечеткое совпадение⁴ слова в этом списке. В качестве алгоритма нечеткого совпадения будем использовать *сходство Джаро — Винклера*, в котором измеряется степень подобия двух строк. Здесь я не буду описывать полную реализацию этого алгоритма. Ее вы найдете в исходном коде к книге.

Алгоритм Джаро — Винклера

Сходство Джаро — Винклера измеряет степень подобия двух строк. Чем выше сходство Джаро — Винклера, тем больше похожи эти строки. Данная оценка лучше всего подходит для коротких строк, таких как имена собственные. Оценка нормализуется так: 0 означает полное отсутствие подобия, 1 — точное совпадение.

В листинге 2.22 показана реализация функции нечеткого совпадения с использованием алгоритма Джаро — Винклера (выделен жирным шрифтом).

В функции `FuzzyMatch` применяется PLINQ-запрос для параллельного вычисления нечеткого совпадения слова, переданного в качестве аргумента, с массивом строк. Результатом является коллекция совпадений `HashSet`, которая затем упорядочивается по признаку наилучшего соответствия, и функция возвращает первое значение из списка. `HashSet` — эффективная структура данных для поиска.

Логика алгоритма похожа на поиск. Поскольку в списке `List<string> words` могут присутствовать дубликаты, функция сначала создает более эффективную структуру данных, а затем использует ее для выполнения алгоритма нечеткого совпадения. Такая реализация неэффективна, и ее недостаток очевиден: при каждом вызове функция `FuzzyMatch` выполняется для обоих ее аргументов. При каждом

⁴ Нечеткое совпадение — это метод поиска сегментов текста и соответствующих им фрагментов другой строки, которые могут совпадать не на 100 %.

выполнении **FuzzyMatch** внутренняя структура таблицы перестраивается заново, из-за чего всякий положительный эффект теряется.

Листинг 2.22. Реализация нечеткого совпадения на C#

```
public static string FuzzyMatch(List<string> words, string word)
{
    var wordSet = new HashSet<string>(words);
    string bestMatch =
        (from w in wordSet.AsParallel()           ← Параллельное выполнение
         select JaroWinklerModule.Match(w, word))
        .OrderByDescending(w => w.Distance)
        .Select(w => w.Word)
        .FirstOrDefault();
    return bestMatch;                         ← Возвращение наилучшего соответствия
}
```

Удаление возможных дубликатов слов путем создания коллекции HashSet из списка слов. HashSet — эффективная структура данных для поиска

Как повысить эффективность в данном случае? Применив сочетание частичного выполнения функции или частичного приложения и технологии мемоизации из ФП, можно получить предварительное вычисление. Подробнее о частичном выполнении читайте в приложении А. Концепция предварительного вычисления тесно связана с мемоизацией, для которой в данном случае используется таблица, содержащая предварительно вычисленные значения. В листинге 2.23 представлена реализация более быстрой функции поиска нечетких совпадений (выделена жирным шрифтом).

Листинг 2.23. Быстрый поиск нечетких совпадений с использованием предварительного вычисления

Частично выполняемая функция принимает только один параметр и возвращает новую функцию, которая и производит «умный» поиск.

```
► static Func<string, string> PartialFuzzyMatch(List<string> words)
{
    var wordSet = new HashSet<string>(words); ← После создания эффективной
                                                структуры поиска данных она
    return word =>                                хранится в замыкании и используется
        (from w in wordSet.AsParallel()           лямбда-выражением
         select JaroWinklerModule.Match(w, word))
         .OrderByDescending(w => w.Distance)
         .Select(w => w.Word)
         .FirstOrDefault(); ← Новая функция при каждом вызове использует
                           одни и те же данные поиска, что сокращает
                           объем повторяющихся вычислений

}
Func<string, string> fastFuzzyMatch =
➥ PartialFuzzyMatch(words); ← Функция, которая выполняет
                           предварительные вычисления
                           для списка List<string> words
                           и возвращает функцию, принимающую
                           искомое слово в качестве аргумента

string magicFuzzyMatch = fastFuzzyMatch("magic");
► string lightFuzzyMatch = fastFuzzyMatch("light");

Использование функции fastFuzzyMatch
```

Сначала мы создаем частично выполняемую версию функции `PartialFuzzyMatch`. Эта новая функция принимает в качестве аргумента только список `List<string> words` и возвращает новую функцию, которая обрабатывает второй аргумент. Данная стратегия более разумна, поскольку предусматривает обработку первого аргумента сразу, как только он будет передан, путем предварительного вычисления эффективной структуры поиска.

Примечательно, что компилятор использует замыкание для хранения структуры данных, доступной через лямбда-выражение, возвращаемое функцией. Лямбда-выражение — исключительно удобный способ реализации функции с предварительно вычисленным состоянием. Затем можно определить функцию `fastFuzzyMatch`, предоставив аргумент `List<string> words`, который применяется для подготовки базовой таблицы поиска, что приводит к более быстрым вычислениям. Получив `List<string> words`, функция `fastFuzzyMatch` возвращает функцию, которая принимает строковый аргумент `word` и сразу же вычисляет `HashSet` для поиска.

ПРИМЕЧАНИЕ

Функция `fuzzyMatch` в листинге 2.22 скомпилирована как статическая функция, которая составляет набор строк для каждого вызова. В листинге 2.23 `fastFuzzyMatch`, напротив, скомпилирована как статическое свойство, предназначенное только для чтения, и значение данного свойства инициализируется в статическом конструкторе. Это тонкое различие, но оно оказывает огромное влияние на производительность кода.

После указанных изменений время обработки при поиске нечеткого совпадения для строк `magic` и `light` сократилось наполовину по сравнению с вариантом, когда эти значения вычислялись по мере необходимости.

2.6.1. Предварительные вычисления с естественной поддержкой функционального программирования

Теперь рассмотрим эту же реализацию поиска нечетких совпадений, но на языке функционального программирования F#. В листинге 2.24 показана немного другая реализация, по причине внутренней функциональной семантики F# (метод `AsParallel` выделен жирным шрифтом).

Реализация функции `fuzzyMatch` такова, что среда выполнения F# вынуждена генерировать внутренний набор строк для каждого вызова. Частично выполняемая функция `fastFuzzyMatch`, напротив, инициализирует внутренний набор только один раз и затем применяет его для всех последующих вызовов. Предварительные вычисления — это технология кэширования, при которой начальное вычисление выполняется для того, чтобы создать в данном случае готовый к использованию в последующих вызовах набор `HashSet<string>`.

Листинг 2.24. Реализация быстрого поиска нечетких совпадений на F#

Создание HashSet — эффективной структуры для поиска данных, в которой также удаляются повторяющиеся слова

```

let fuzzyMatch (words:string list) =
    let wordSet = new HashSet<string>(words)
    let partialFuzzyMatch word =
        query { for w in wordSet.AsParallel() do
            select (JaroWinkler.getMatch w word) } |>
        Seq.sortBy(fun x -> -x.Distance)
        |> Seq.head
    
```

В F# все функции имеют значения по умолчанию. Сигнатура функции `FuzzyMatch` такова: `(string set -> string -> string)`. Это подразумевает, что данная функция может быть выполнена частично. В таком случае, предоставляя только первый аргумент `wordSet`, мы создаем частично выполняемую функцию `partialFuzzyMatch`


```

    |> fun word -> partialFuzzyMatch word
    let fastFuzzyMatch = fuzzyMatch words
    let magicFuzzyMatch = fastFuzzyMatch "magic"
    let lightFuzzyMatch = fastFuzzyMatch "light"
    
```

Реализация технологии предварительного вычисления путем передачи первого аргумента функции `fuzzyMatch`, которая сразу вычисляет `HashSet` для переданных ей значений

Использование функции `fastFuzzyMatch`

Возвращение функции, которая использует лямбда-выражение для «замыкания» и, в свою очередь, возвращает внутренний набор HashSet

В реализации F# для получения и преобразования данных применяется выражение запроса. Этот подход позволяет задействовать PLINQ так же, как в эквивалентном листинге 2.23, написанном на C#. Но в F# есть более функциональный стиль для распараллеливания операций с последовательностями — параллельные последовательности (PSeq). Используя данный модуль, можно переписать функцию `fuzzyMatch` в форме скомпонованной функции следующим образом.

```

let fuzzyMatch (words:string list) =
    let wordSet = new HashSet<string>(words)
    fun word ->
        wordSet
        |> PSeq.map(fun w -> JaroWinkler.getMatch w word)
        |> PSeq.sortBy(fun x -> -x.Distance)
        |> Seq.head
    
```

Реализации функции `fuzzyMatch` на C# и F# эквивалентны, но на F# функциональное программирование поддерживается по умолчанию. Это упрощает рефакторинг с применением частичного выполнения. Инструкция `PSeq` на F#, использованная в предыдущем фрагменте кода, будет рассмотрена в главе 5.

Станет более понятно, если посмотреть на тип сигнатуры `fuzzyMatch`:

```
string set -> (string -> string)
```

Сигнатура читается как функция, принимающая в качестве аргумента набор строк и возвращающая функцию, которая, в свою очередь, принимает строку в качестве аргумента и возвращает результат в виде строки. Такая цепочка функций позволяет использовать стратегию частичного выполнения, даже не задумываясь об этом.

2.6.2. Пускай победит быстрейший!

Другой пример упреждающего выполнения был подсказан мне оператором однозначного выбора¹, изобретенным Коналом Эллиоттом (Conal Elliott, <http://conal.net>) для реализации его идеи функционального реактивного программирования (ФРП, <http://conal.net/papers/push-pull-frp>). Идея этого оператора проста: он представляет собой функцию, которая принимает два аргумента и конкурентно обрабатывает их, возвращая первый из полученных результатов.

Такая концепция может быть расширена на более чем две параллельные функции. Представим себе, что мы используем данные нескольких метеорологических служб, чтобы узнавать температуру в городе. Мы можем одновременно запустить отдельные задачи для запроса каждой службы и, когда самая быстрая из этих задач вернет ответ, не дожидаться завершения остальных. Функция ожидает выполнения самой быстрой задачи и затем отменяет оставшиеся. В листинге 2.25 показана простая реализация данной идеи без обработки ошибок.

Листинг 2.25. Реализация задачи получения самого быстрого прогноза погоды

```
public Temperature SpeculativeTempCityQuery(string city,
    params Uri[] weatherServices)
{
    var cts = new CancellationTokenSource(); ←
    var tasks = ←
        (from uri in weatherServices
         select Task.Factory.StartNew<Temperature>(() =>
            queryService(uri, city), cts.Token)).ToArray(); ←

    int taskIndex = Task.WaitAny(tasks); ←
    Temperature tempCity = tasks[taskIndex].Result; ←
    cts.Cancel(); ←
    return tempCity; ←
}
```

Использование маркера отмены, чтобы отменить задачу после получения самого быстрого результата

Ожидание завершения самой быстрой задачи

Отмена оставшихся более медленных задач

Применение LINQ-запроса для параллельной обработки отдельных задач для каждой метеорологической службы

Предварительные вычисления — важнейшая технология для реализации любых функций и сервисов, от простых до сложных и более усовершенствованных вычислительных систем. Упреждающее выполнение направлено на рациональное потребление ресурсов процессора, которые в противном случае простаивали бы. Это удобная технология для любой программы, и она может быть реализована на любом языке, который поддерживает замыкания для захвата и выдачи частичных значений.

¹ Elliott C. Functional Concurrency with Unambiguous Choice, November 21, 2008. <http://mng.bz/4mKK>.

2.7. Лень — это хорошо

Одной из основных проблем конкурентности является корректная, потокобезопасная инициализация разделяемых объектов. Когда речь идет об ускорении запуска приложений с конструкторами объектов, требующих значительных вычислительных затрат и времени на выполнение, потребность в них становится еще более актуальной.

«Ленивое» выполнение — технология программирования, используемая для отсрочки выполнения выражения до последнего момента, пока еще возможно. Верьте или нет, но лень способна привести к успеху, и в данном случае она является важным инструментом в вашем арсенале. Пускай это и неочевидно, но возможность «ленивого» выполнения позволяет программе работать быстрее, потому что в таком случае делается только то, что действительно нужно для получения результата запроса, без лишних вычислений. Представим себе, что мы пишем программу, которая выполняет разные длительные операции — вероятно, анализирует большие объемы данных для формирования различных отчетов. Если такие операции будут выполняться одновременно, то системе может не хватить производительности и она зависнет. Кроме того, возможно, не все длительные операции необходимо выполнять немедленно, и если запустить их все сразу, то это приведет к напрасной трате ресурсов и времени.

Лучшей стратегией будет выполнение длительных операций по требованию и только по мере необходимости, что также снизит нагрузку на системную память. По сути, «ленивое» выполнение также приводит к эффективному управлению памятью, повышая производительность благодаря меньшему потреблению памяти. Как видим, быть ленивым в данном случае весьма эффективно. Сокращение ненужных и затратных операций по сбору мусора в управляемых языках программирования, таких как C#, Java и F#, позволяет ускорить работу программ.

2.7.1. Использование строгих языков программирования для лучшего понимания конкурентного поведения

Противоположностью «ленивому» выполнению выступает *энергичное выполнение*, также известное как *строгое выполнение*, — это означает, что выражение выполняется немедленно. C# и F#, как и большинство основных языков программирования, являются строгими языками.

Императивные языки программирования не имеют внутренней модели для хранения и контроля побочных эффектов, поэтому решение об энергичном выполнении в них является разумным. Чтобы понимать, как работает программа, язык со строгим выполнением должен знать последовательность выполнения побочных эффектов (таких как ввод и вывод). Фактически строгий язык анализирует вычисления и на этом основании составляет представление о работе, которая должна быть выполнена.

Поскольку и C#, и F# не являются чистыми ФП-языками, нет никакой гарантии того, что каждое значение в них ссылочно прозрачно; следовательно, они не будут языками программирования с «ленивым» выполнением.

В целом «ленивое» выполнение трудно сочетается с императивными свойствами, которые иногда вносят побочные эффекты, такие как исключения и операции ввода-

вывода, поскольку порядок операций становится недетерминированным. Для получения дополнительной информации я рекомендую почитать работу Джона Хьюза (John Hughes) «Значение функционального программирования» (<http://mng.bz/qp3B>).

В ФП «ленивые» вычисления и побочные эффекты не могут существовать вместе. В императивный язык программирования можно ввести понятие ленивых вычислений, однако в сочетании с побочными эффектами это повышает сложность программы. Фактически «ленивые» вычисления заставляют разработчика снимать ограничения на последовательность выполнения операций и зависимостей в соответствии с тем, какие части программы выполняются. Написание программы с побочными эффектами может стать затруднительным, поскольку для этого требуется ввести понятие функциональной последовательности выполнения, что уменьшает возможность разбиения на модули и снижает степень компонуемости кода. Функциональное программирование направлено на то, чтобы сделать побочные эффекты явными, знать о них и предоставить разработчику инструменты для их изоляции и контроля. В частности, в Haskell используется соглашение функционального программирования об отождествлении функции с побочными эффектами с типом ввода-вывода. При таком определении функция Haskell считывает файл, вызывая побочные эффекты, следующим образом:

```
readFile :: IO()
```

Такое явное определение уведомляет компилятор о наличии побочных эффектов, после чего он применяет оптимизацию и валидацию по мере необходимости.

«Ленивая» оценка становится важной техникой при использовании многоядерных и многопоточных программ. Для поддержки этой технологии Microsoft представила (начиная с Framework 4.0) конструктор параметризованного типа `Lazy<T>`, который потокобезопасным способом упрощает инициализацию объектов с отложенным созданием. В листинге 2.26 представлено определение «ленивого» объекта `Person`.

Листинг 2.26. «Ленивая» инициализация объекта `Person`

```
class Person ← Определение класса Person
{
    public readonly string FullName;
    public Person(string firstName, string lastName)
    {
        FullName = firstName + " " + lastName;
        Console.WriteLine(FullName);
    }
}
Lazy<Person> fredFlintstone = new Lazy<Person>(() =>
    new Person("Fred", "Flintstone"), true); ← Инициализация «ленивого» объекта Person;
                                                // возвращает значение Lazy<Person>,
                                                // которое не определяется до тех пор,
                                                // пока в этом не возникнет необходимость

Person[] freds = new Person[5]; ← Массив из пяти объектов Person
for(int i = 0; i < freds.Length; i++)
    freds[i] = fredFlintstone.Value; ← Экземпляр базового «ленивого» объекта,
                                                // доступный через свойство Value

// вывод
Fred Flintstone
```

Предназначенное только
для чтения поле полного имени,
инициализированное в конструкторе

Инициализация «ленивого» объекта Person;
возвращает значение Lazy<Person>,
 // которое не определяется до тех пор,
пока в этом не возникнет необходимость

Массив из пяти объектов Person

Экземпляр базового «ленивого» объекта,
 // доступный через свойство Value

В данном примере определяется простой класс `Person` с предназначением только для чтения полем, который также выводит в консоль значение `FullName`. Затем создается «ленивый» инициализатор для этого объекта путем передачи делегата фабрики в функцию `Lazy<Person>`, которая отвечает за создание объекта. В нашем случае вместо делегата фабрики удобно использовать лямбда-выражение. Это показано на рис. 2.4.

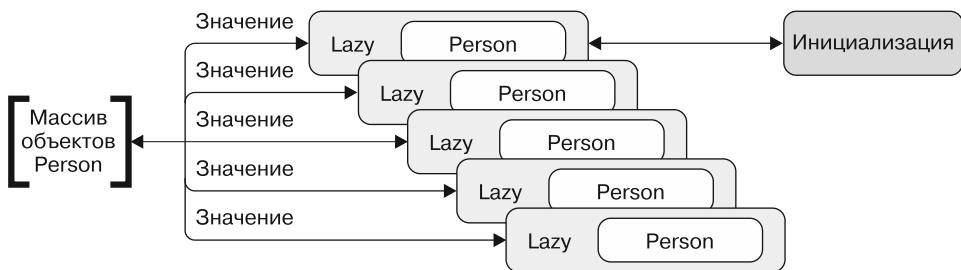


Рис. 2.4. Значение объекта `Person` инициализируется только один раз, при первом обращении к свойству `Value`. Последующие вызовы возвращают одно и то же кэшированное значение. Если есть массив объектов `Lazy<Person>`, то при обращении к элементам массива инициализируется только первый из них, остальные будут многократно использовать результат из кэша

Когда возникает необходимость в фактическом вычислении выражения для применения базового объекта `Person`, мы получаем доступ к свойству `Value` по идентификатору. Поэтому делегат фабрики объекта `Lazy` выполняется только один раз — если значение еще не вычислено. Независимо от того, сколько последует дальше вызовов или сколько потоков будет одновременно обращаться к «ленивому» инициализатору, все они получат один и тот же экземпляр. Чтобы это показать, в листинге создается массив из пяти объектов `Person`, которые инициализируются в цикле `for`. На каждой итерации извлекается объект `Person` путем обращения к его свойству-идентификатору `Value`. Но даже если вызвать такой объект пять раз, вывод (`Fred Flintstone`) все равно происходит только один раз.

2.7.2. «Ленивое» кэширование и потокобезопасный шаблон «Одиночка»

«Ленивые» вычисления .NET рассматриваются как технология кэширования, поскольку при этом запоминается результат выполненных вычислений и программа может работать более эффективно, без повторяющихся и дублирующихся операций.

Поскольку операции выполняются по требованию и, что более важно, только один раз, конструкция `Lazy<T>` является рекомендуемым механизмом реализации шаблона «Одиночка» (Singleton). Шаблон «Одиночка» подразумевает создание

одного экземпляра некоего ресурса, который совместно используется в нескольких частях кода. Данный ресурс нужно инициализировать только один раз, при первом обращении к нему, — именно так ведет себя `Lazy<T>`.

В .NET есть разные способы реализации шаблона «Одиночка», но некоторые из этих методов имеют ограничения, такие как негарантированная безопасность потоков или потерянный «ленивый» экземпляр¹. Конструкция `Lazy<T>` обеспечивает лучшую и более простую структуру одиночки, что гарантирует действительно «ленивое» выполнение и потокобезопасность, как показано в листинге 2.27.

Листинг 2.27. Шаблон «Одиночка» с использованием `Lazy<T>`

```
public sealed class Singleton
{
    private static readonly Lazy<Singleton> lazy =
        new Lazy<Singleton>(() => new Singleton(), true); ← Вызов делегата
    public static Singleton Instance => lazy.Value;           | конструктора Singleton

    private Singleton()
    { }
}
```

Примитив `Lazy<T>` также принимает в качестве необязательного аргумента логический флаг, передаваемый после выполнения лямбда-выражения, который активизирует потокобезопасное поведение. Так можно реализовать сложную и простую версию шаблона блокировки с двойной проверкой (Double-Check Locking).

ПРИМЕЧАНИЕ

При разработке программного обеспечения шаблон Double-Check Locking (также известный как оптимизация блокировки с двойной проверкой) представляет собой шаблон разработки программного обеспечения, используемый для сокращения издержек на установку блокировки. Для этого предварительно, без установки блокировки, проверяется критерий блокировки («подсказка блокировки»).

Данное свойство гарантирует, что инициализация объекта будет потокобезопасной. Когда флаг установлен, что является режимом по умолчанию, то, независимо от того, сколько потоков вызывает одиночку `LazyInitializer`, все потоки получают один и тот же экземпляр, кэшируемый после первого вызова. Это огромное преимущество, без которого пришлось бы вручную защищать потоки и обеспечивать их безопасность для совместно используемого поля.

Важно подчеркнуть, что, если реализация объекта с «ленивым» вычислением является потокобезопасной, это не означает, что автоматически все его свойства также будут потокобезопасными.

¹ См. Implementing Singleton in C#, MSDN, <http://mng.bz/pLf4>.

LazyInitializer

В .NET `LazyInitializer` является альтернативным статическим классом, который работает подобно `Lazy<T>`, но с оптимизированной производительностью при инициализации и более удобным доступом. На самом деле для создания типа `Lazy` нет необходимости в инициализации объекта с помощью оператора `new`, так как его функциональность реализуется посредством статического метода. В следующем простом примере показана «ленивая» инициализация большого изображения с помощью `LazyInitializer`:

```
private BigImage bigImage;
public BigImage BigImage =>
    LazyInitializer.EnsureInitialized(ref bigImage, () => new
BigImage());
```

2.7.3. Поддержка «ленивых» вычислений в F#

Язык F# также поддерживает тип `Lazy<T>` с добавленными «ленивыми» вычислениями, возвращающий также тип `Lazy<T>`, где фактический параметризованный тип, используемый вместо `T`, определяется по результату выражения. Стандартная библиотека F# автоматически обеспечивает взаимное исключение, поэтому чистый код функции является потокобезопасным, одновременно выдавая одно и то же «ленивое» значение из разных потоков. Применение типа `Lazy` в F# немного отличается от применения в C#, где нужно обернуть тип данных `Lazy` в функцию. В следующем примере кода показано «ленивое» вычисление объекта `Person` в F#:

```
let barneyRubble = lazy( Person("barney", "rubble") )
printfn "%s" (barneyRubble.Force().FullName)
```

Функция `barneyRubble` создает экземпляр `Lazy<Person>`, значение которого еще не определено. Затем, чтобы заставить вычислить это значение, мы вызываем метод `Force`, извлекающий значение по требованию.

2.7.4. Мощное сочетание Lazy и Task

Из соображений производительности и масштабируемости в конкурентном приложении полезно сочетать «ленивые» вычисления, которые могут выполняться по требованию, с использованием независимых потоков.

Инициализатор типа `Lazy`, `Lazy<T>`, может применяться для реализации полезного шаблона, позволяющего создавать объекты, требующие асинхронных операций. Рассмотрим класс `Person`, который использовался в предыдущем разделе. Если первое и второе поля имени загружаются из базы данных, то можно задействовать тип `Lazy<Task<Person>>` для отсроченного выполнения ввода-вывода. Интересно, что между `Task<T>` и `Lazy<T>` есть общая черта: оба они выполняют заданное выражение только один раз (листинг 2.28).

Листинг 2.28. «Ленивая» асинхронная инициализация объекта Person

```

Lazy<Task<Person>> person =
    new Lazy<Task<Person>>((async () => ← Асинхронный лямбда-конструктор для типа Lazy
{
    using (var cmd = new SqlCommand(cmdText, conn))
    using (var reader = await cmd.ExecuteReaderAsync())
    {
        if (await reader.ReadAsync())
        {
            string firstName = reader["first_name"].ToString();
            string lastName = reader["last_name"].ToString();
            return new Person(firstName, lastName);
        }
    }
    throw new Exception("Failed to fetch Person");
});

async Task<Person> FetchPerson()
{
    return await person.Value; ← Асинхронно материализованный тип Lazy
}

```

В данном примере делегат возвращает объект типа `Task<Person>`, один раз асинхронно вычисляющий значение и затем возвращающий его при всех вызовах. Это одно из тех решений, которые в итоге улучшают масштабируемость программы. В указанном примере такой прием использован для реализации асинхронных операций с применением ключевых слов `async/await` (которые появились в C# 5.0). Подробнее об асинхронности и масштабируемости читайте в главе 8.

Это полезный прием, который позволяет повысить масштабируемость параллелизм программы, но есть небольшой риск. Поскольку лямбда-выражение является асинхронным, его можно выполнить в любом потоке, вызывающем `Value`, и выражение будет выполняться в контексте. Лучшим решением будет обернуть выражение в базовый класс `Task`, что заставит его асинхронно выполнятся в потоке пула потоков. Этот более предпочтительный шаблон показан в листинге 2.29.

Листинг 2.29. Улучшенный шаблон

```

Lazy<Task<Person>> person =
    new Lazy<Task<Person>>(() => Task.Run(
        async () =>
    {
        using (var cmd = new SqlCommand(cmdText, conn))
        using (var reader = await cmd.ExecuteReaderAsync())
        {
            if(await reader.ReadAsync())
            {
                string firstName = reader["first_name"].ToString();
                string lastName = reader["last_name"].ToString();
                return new Person(firstName, lastName);
            } else throw new Exception("No record available");
        }
    });

```

Резюме

- ❑ При функциональной компоновке результат выполнения одной функции поступает на вход другой, так что создается новая функция. Этую технологию можно использовать в ФП для решения сложных задач, подразделяя их на более мелкие и простые проблемы, которые легче решаются, а затем объединяя данные решения.
- ❑ Замыкание — это встроенный метод делегирования и анонимности, привязанный к родительскому методу; переменные, определенные в теле родительского метода, доступны из анонимного метода. Замыкание является удобным способом предоставить функции доступ к локальному состоянию (которое замкнуто в функции), даже если оно выходит за пределы области видимости. Это основа для разработки сегментов кода в функциональном стиле, включающего в себя мемоизацию, «ленивую» инициализацию и предварительное вычисление для повышения скорости работы.
- ❑ Мемоизация — технология функционального программирования, позволяющая сохранять результаты промежуточных вычислений, вместо того чтобы повторно вычислять их. Мемоизация является одной из форм кэширования.
- ❑ Предварительное вычисление — технология выполнения начального вычисления, при которой генерируется набор результатов, обычно в форме таблицы поиска. Эти предварительно вычисленные значения могут использоваться непосредственно алгоритмом, чтобы избежать ненужных, повторяющихся и затратных вычислений при каждом выполнении кода. Как правило, предварительная компиляция заменяет мемоизацию и применяется в сочетании с частично выполняемыми функциями.
- ❑ «Ленивая» инициализация — еще один вариант кэширования. Точнее, данная технология откладывает выполнение функции-фабрики при создании объекта до тех пор, пока это не понадобится, создавая объект только один раз. Основная цель «ленивой» инициализации — повысить производительность за счет сокращения потребления памяти и отмены ненужных вычислений.

Функциональные структуры данных и неизменяемость



В этой главе:

- разработка параллельных приложений с функциональными структурами данных;
- использование неизменяемости для создания высокопроизводительного кода, не содержащего блокировок;
- реализация параллельных шаблонов с функциональной рекурсией;
- реализация неизменяемых объектов на C# и F#;
- работа с древовидными структурами данных.

Данные поступают на обработку в самых разных видах. Поэтому неудивительно, что многие компьютерные программы строятся вокруг данных и их обработки. Функциональное программирование хорошо вписывается в наш мир, поскольку эта парадигма программирования предназначена именно для преобразования данных. Функциональные преобразования позволяют изменять представление набора структурированных данных, не беспокоясь о побочных эффектах или состоянии. Например, можно преобразовать коллекцию стран в коллекцию городов, используя функцию словаря, и при этом сохранить исходные данные неизменными. Побочные эффекты являются ключевой проблемой конкурентного программирования, поскольку эффекты, возникающие в одном потоке, могут влиять на поведение другого потока.

За последние годы в основных языках программирования появились новые возможности, упрощающие разработку многопоточных приложений. Microsoft, например, создала библиотеку TPL, а в .NET Framework появились ключевые

слова `async/await`, что позволило программистам меньше беспокоиться при реализации параллельного кода. Но все еще остаются проблемы сохранения измененного состояния, защищенного от искажений, если задействованы несколько потоков. Хорошой новостью является то, что ФП позволяет писать код, в котором неизменяемые данные преобразуются без побочных эффектов.

В этой главе вы научитесь писать параллельный код с задействованием функциональных структур данных и неизменяемых состояний, без особых усилий выбирая правильную структуру данных для конкурентной среды, чтобы повысить производительность. *Функциональные структуры данных* повышают эффективность программ за счет совместного использования структур данных разными потоками и параллельной работы без синхронизации.

Для начала мы разработаем в этой главе функциональный список на C# и на F#. Это отличные упражнения, позволяющие понять, как работают неизменяемые функциональные структуры данных. Затем мы рассмотрим неизменяемые древовидные структуры данных, и вы узнаете, как в ФП можно использовать рекурсию для параллельного построения бинарного дерева. Мы изучим пример применения параллельной рекурсии для одновременной загрузки из Интернета нескольких изображений.

В конце главы будет показано, как задействовать неизменяемость и функциональные структуры данных для ускорения параллельной работы программы, избегая ошибок, таких как состояние гонки, при наличии разделяемого изменяемого состояния. Другими словами, если вам нужна конкурентность с твердыми гарантиями корректности, то от изменяемых данных придется отказаться.

3.1. Практический пример: охота на потоконебезопасный объект

Построение программного обеспечения в контролируемой среде обычно не приводит к неприятным неожиданностям. К сожалению, если программа, написанная на локальном компьютере, развертывается на сервере, который не находится под вашим контролем, это может привести к различным изменениям. В среде эксплуатации программы часто сталкиваются с непредвиденными проблемами и непредсказуемо высокими нагрузками. Я уверен, что вам в вашей практике не раз приходилось слышать: «На моей машине все работало».

В процессе реальной эксплуатации программного обеспечения многое может пойти не так и привести к неправильной работе программы. Некоторое время назад мой босс позвонил мне и попросил проанализировать проблему эксплуатации приложения. Это приложение представляло собой простой чат для поддержки клиентов. В программе, написанной на C#, применялись веб-сокеты для прямого обмена данными с концентратором Windows-сервера. Технологической основой для установки двунаправленной коммуникации между клиентом и сервером была библиотека Microsoft SignalR (<http://mng.bz/Fal1>) (рис. 3.1).

Документация MSDN о SignalR

ASP.NET SignalR – это библиотека, предназначенная для разработчиков ASP.NET, которая упрощает процесс добавления в приложения веб-функций реального времени. Веб-функциональность реального времени имеет серверный код, который посыпает подключенными клиентам контент сразу же, как только он становится доступным, вместо того чтобы заставлять сервер ожидать, пока клиент запросит новые данные. SignalR можно использовать для добавления в приложение ASP.NET любых веб-функций реального времени. Чат – это просто типичный пример, а вообще с помощью данной библиотеки можно сделать гораздо больше. Каждый случай, когда пользователь обновляет веб-страницу, чтобы увидеть новые данные, или же страница отправляет длительный запрос на получение новых данных, – кандидат на применение SignalR. В качестве примеров можно привести информационные панели и приложения мониторинга, приложения для совместной работы (такие как одновременное редактирование документов), обновления данных о выполнении задач и формы реального времени.

Перед развертыванием в среде эксплуатации программа прошла все тесты. Однако после развертывания оказалось, что ресурсы сервера на пределе. Нагрузка на процессор постоянно колебалась между 85 и 95 %, что отрицательно сказывалось на общей производительности и не позволяло системе реагировать на входящие запросы. Такой результат был неприемлемым, и проблему было необходимо срочно решать.

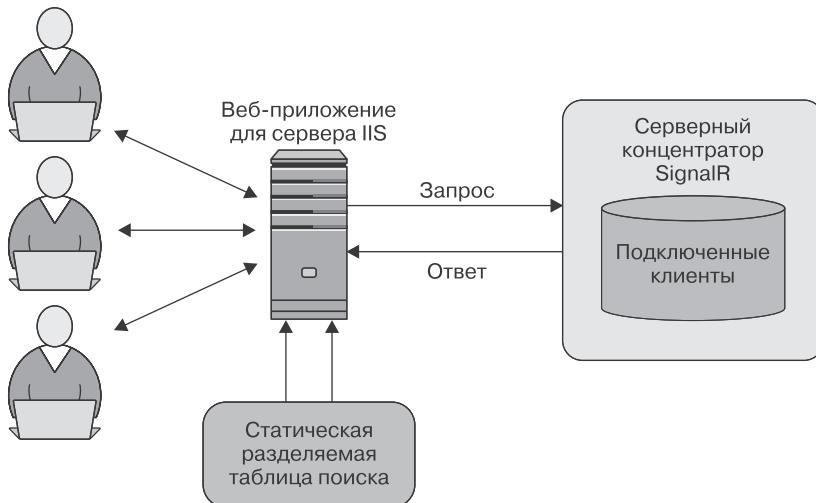


Рис. 3.1. Архитектура веб-серверного приложения чата с использованием концентратора SignalR.

Подключенные клиенты регистрируются в локальном статическом словаре (таблице поиска), экземпляр которого является разделяемым

Как говорил Шерлок Холмс, «отбросьте все невозможное; то, что останется, и будет ответом, каким бы невероятным он ни оказался». Я надел свою шляпу суперсыщика

и, вооружившись лупой, стал рассматривать код. После отладки и тщательного расследования я обнаружил часть кода, которая создавала узкое место.

Для анализа производительности приложения я использовал инструмент профилирования. Выборочное обследование и профилирование приложения — хорошее начальное средство для поиска в нем узких мест. Инструмент профилирования исследует поведение программы при запуске, проверяя время выполнения для обычных данных. Собранные данные представляют собой статистические результаты профилирования для отдельных методов, выполняющих основную работу в приложении. Эти методы представлены в заключительном отчете, и посредством поиска «горячего пути» можно убедиться, что именно они выполняют главную часть работы приложения.

Проблема высокой нагрузки на центральный процессор возникла в методах `OnConnected` и `OnDisconnected` из-за конкуренции за разделяемое состояние. В данном случае разделяемым состоянием был обобщенный тип `Dictionary`, используемый для хранения в памяти подключенных пользователей. *Конфликт потоков* — это ситуация, когда один поток ожидает доступа к объекту, который удерживается другим потоком. Ожидавший поток не может продолжать работу до тех пор, пока другой поток не освободит объект (данний объект заблокирован). Этот проблемный серверный код показан в листинге 3.1.

Листинг 3.1. Написанный на C# концентратор SignalR, который регистрирует соединения в контексте

```
Проверка того, что текущий пользователь
уже подключен и сохранен в словаре
→ static Dictionary<Guid, string> onlineUsers =
    new Dictionary<Guid, string>(); ← Совместный доступ к экземпляру
                                    статического словаря для обработки
                                    состояния подключенных пользователей
→ public override Task OnConnected() { ← Каждое соединение
    Guid connectionId = new Guid (Context.ConnectionId); ← связано с уникальным
    System.Security.Principal.IPrincipal user = Context.User;
    string userName;
    if (!onlineUsers.TryGetValue(connectionId, out userName)){ ←
        RegisterUserConnection (connectionId, user.Identity.Name);
        onlineUsers.Add(connectionId, user.Identity.Name); ←
    }
    return base.OnConnected();
}
public override Task OnDisconnected() {
    Guid connectionId = new Guid (Context.ConnectionId);
    string userName;
    if (onlineUsers.TryGetValue(connectionId, out userName)){ ←
        DeregisterUserConnection(connectionId, userName);
        onlineUsers.Remove(connectionId); ←
    }
    return base.OnDisconnected();
}
```

Добавление и удаление пользователя выполняется после проверки состояния словаря

Операции `OnConnected` и `OnDisconnected` описаются на разделяемый глобальный словарь, который обычно используется в программах этого типа для хранения локального состояния. Обратите внимание, что всякий раз, когда выполняется один из данных методов, лежащая в его основе коллекция вызывается дважды. Логика программы проверяет, существует ли значение `User Connection Id`, и выбирает соответствующее поведение:

```
string userName;  
if (!onlineUsers.TryGetValue(connectionId, out userName)){
```

Видите проблему? При каждом клиентском запросе устанавливается новое соединение и создается новый экземпляр концентратора. Локальное состояние хранится в статической переменной, которая отслеживает текущее пользовательское соединение и совместно используется всеми экземплярами концентратора. Согласно документации Microsoft, «статический конструктор вызывается только один раз, а статический класс остается в памяти в течение всего времени жизни приложения, в котором находится ваша программа»¹.

Ниже представлена коллекция, применяемая для отслеживания пользовательских подключений:

```
static Dictionary<Guid, string> onlineUsers =  
    new Dictionary<Guid, string>();
```

`Guid` — уникальный идентификатор соединения, создаваемый SignalR при установке соединения между клиентом и сервером. Это строка, в которой представлено имя пользователя, определенное при входе в систему. В данном случае программа явно выполняется в многопоточной среде. Каждый входящий запрос представляет собой новый поток; следовательно, будет несколько запросов, одновременно обращающихся к разделяемому состоянию, что рано или поздно приведет к проблемам многопоточности.

В данном отношении документация MSDN предельно ясна: в ней говорится, что коллекция `Dictionary` способна поддерживать несколько конкурентных операций чтения, если только она не будет изменена². Перечисление посредством коллекции, по сути, не является потокобезопасным, поскольку один поток может обновить словарь в тот момент, когда другой поток будет изменять состояние коллекции.

Есть несколько возможных решений, позволяющих избежать этого ограничения. Первый подход заключается в том, чтобы сделать коллекцию потокобезопасной и доступной для операций чтения и записи, выполняемых несколькими потоками, с использованием `lock primitive`. Такое решение является корректным, но снижает производительность.

Предпочтительнее было бы обеспечить такой же уровень безопасности потоков, но без синхронизации; например, с применением неизменяемых коллекций.

¹ Подробнее о статических классах и членах статических классов см.: <http://mng.bz/agzj>.

² Подробнее о потокобезопасности читайте здесь: <http://mng.bz/k8Gg>.

3.1.1. Неизменяемые коллекции .NET: надежное решение

В .NET Framework 4.5 Microsoft представила неизменяемые коллекции, расположенные в пространстве имен `System.Collections.Immutable`. Это очередная часть эволюции инструментов потоковой обработки после библиотеки TPL в .NET 4.0 и ключевых слов `async` и `await` в .NET 4.5.

Неизменяемые коллекции соответствуют концепциям функциональной парадигмы, рассматриваемым в данной главе, и обеспечивают неявную потокобезопасность в многопоточных приложениях, что позволяет решить проблему хранения и контроля изменяемых состояний. Подобно конкурентным коллекциям, они потокобезопасны, но основы их реализации различаются. Любые операции, изменяющие структуры данных, не изменяют исходный экземпляр. Вместо этого они возвращают измененную копию, оставляя исходный экземпляр неизмененным. Неизменяемые коллекции были специально оптимизированы для максимальной производительности. Для минимизации требований по сборке мусора (GC) в них используется шаблон разделения структур (*Structural Sharing*)¹. В качестве примера в следующем фрагменте кода показано создание неизменяемой коллекции на основе обобщенной изменяемой коллекции (команда назначения неизменяемости выделена жирным шрифтом). Затем, путем добавления в коллекцию нового элемента, создается новая коллекция, в результате чего оригинал не изменяется:

```
var original = new Dictionary<int, int>().ToImmutableDictionary();
var modifiedCollection = original.Add(key, value);
```

Любые изменения коллекции, сделанные в одном потоке, не видны другим потокам, поскольку они все еще ссылаются на исходную немодифицированную коллекцию. Именно поэтому неизменяемые коллекции по определению являются потокобезопасными.

В табл. 3.1 показана реализация неизменяемой коллекции для каждой из связанных с ней обобщенных коллекций.

Таблица 3.1. Неизменяемые коллекции в .NET Framework 4.5

Неизменяемая коллекция	Изменяемая коллекция
<code>ImmutableList<T></code>	<code>List<T></code>
<code>ImmutableDictionary<TKey, TValue></code>	<code>Dictionary<TKey, TValue></code>
<code>ImmutableHashSet<T></code>	<code>HashSet<T></code>
<code>ImmutableStack<T></code>	<code>Stack<T></code>
<code>ImmutableQueue<T></code>	<code>Queue<T></code>

¹ Persistent Data Structure, https://en.wikipedia.org/wiki/Persistent_data_structure.

ПРИМЕЧАНИЕ

В предыдущей версии .NET была предпринята попытка создать неизменяемые коллекции с использованием обобщенного метода `ReadOnlyCollection` и метода расширения `AsReadOnly`, который преобразует изменяемую коллекцию в коллекцию, предназначенную только для чтения. Но такая коллекция является лишь оболочкой, предотвращающей изменение базовой коллекции. Поэтому в многопоточной программе если поток изменяет обернутую коллекцию, то коллекция, предназначенная только для чтения, отражает такие изменения. Неизменяемые коллекции решают данную проблему.

В листинге 3.2 представлены два способа создания неизменяемого списка.

Листинг 3.2. Создание неизменяемых коллекций в .NET

```
var list = ImmutableList.Create<int>(); ← Создание пустого неизменяемого списка
list = list.Add(1); ← Добавление в список нового
list = list.Add(2); элемента и возвращение нового списка
list = list.Add(3); ← Создание компоновщика списков
                     для построения определения списка
                     с изменяемой семантикой,
                     затем замораживание коллекции

var builder = ImmutableList.CreateBuilder<int>(); ←
builder.Add(1); ← Добавление нового элемента в компоновщик
builder.Add(2); списка, что изменяет коллекцию
builder.Add(3); ← Закрытие компоновщика списков
list = builder.ToImmutable(); ← для создания неизменяемого списка
```

Второй подход упрощает построение списка путем создания временного компоновщика списков, который используется для добавления элементов в список, а затем запечатывает (замораживает) их, превращая список в неизменяемую структуру.

Что касается проблемы повреждения данных (состояния гонки) в исходной программе чата, то неизменяемые коллекции могут задействоваться в концентраторе Windows-сервера для поддержания состояния открытых соединений SignalR. Это безопасно достигается посредством многопоточного доступа. К счастью, в пространстве имен `System.Collections.Immutable` есть эквивалентная версия `Dictionary` для поиска — `ImmutableDictionary`.

Вы спросите: «Но если коллекция неизменяема, то как ее обновлять с сохранением потокобезопасности?» Для этого можно использовать операторы блокировки операций, связанных с чтением и записью коллекции. Построить потокобезопасную коллекцию с применением блокировки легко, но будет слишком дорогим подходом. Лучшим вариантом является защита записей с помощью единой операции сравнения с обменом (`compare-and-swap`, CAS), которая устраняет необходимость блокировок и оставляет операцию чтения незащищенной. Такая технология без блокировок более масштабируема и работает лучше, чем ее аналог, использующий примитив синхронизации.

Операции CAS

CAS — это специальная инструкция, применяемая в многопоточном программировании как форма синхронизации, при которой операция выполняется атомарно в ячейках памяти. Атомарная операция завершается либо успешно, либо неудачно — но всегда как единое целое.

Атомарными называют операции, изменяющие состояние за один шаг таким образом, что результат является автономным; подобная операция может быть либо выполнена, либо нет, без промежуточных вариантов. Другие параллельные потоки могут видеть только старое или новое состояние. Когда атомарная операция выполняется для разделяемой переменной, потоки не могут наблюдать процесс модификации данной переменной до завершения операции. В сущности, при атомарной операции значение считывается в том виде, в каком оно существует в данный момент времени. Примитивные атомарные операции — это машинные инструкции; в .NET они представлены в классе `System.Threading.Interlocked`, например `Interlocked.CompareExchange` и `Interlocked.Increment`.

Инструкция CAS изменяет разделяемые данные без необходимости установки и снятия блокировки и допускает экстремальные уровни параллелизма. Именно там сияют вершины неизменяемых структур данных, поскольку атомарные операции сводят к минимуму вероятность возникновения проблемы АВА (https://ru.wikipedia.org/wiki/Проблема_ABA и https://en.wikipedia.org/wiki/ABA_problem).

Проблема АВА

Проблема АВА возникает при выполнении атомарной операции CAS, когда один поток приостанавливается перед выполнением CAS, а второй изменяет исходное значение целевого объекта CAS-инструкции. Когда первый поток возобновляется, CAS завершается успешно, несмотря на изменение значения целевого объекта.

Идея состоит в том, чтобы сохранить состояние, которое должно измениться, в одном и, самое главное, изолированном неизменяемом объекте (в данном случае `ImmutableDictionary`). Поскольку объект является изолированным, то не существует разделяемого состояния и, следовательно, нет необходимости в синхронизации.

В листинге 3.3 показана реализация вспомогательного объекта `Atom`. Название объекта навеяно понятием атома из языка программирования Clojure (<https://clojure.org/reference/atoms>). Для выполнения атомарных операций CAS внутри объекта `Atom` используется оператор `Interlocked.CompareExchange`.

Класс `Atom` инкапсулирует ссылочный объект типа `T`, отмеченный как `volatile`¹. Для того чтобы обмен значениями произошел корректно, этот объект должен быть неизменяемым. Свойство `Value` используется для чтения текущего состояния обернутого объекта. Назначением функции `Swap` является выполнение инструкции CAS — передача объекту, вызывающему данную функцию, нового значения, полученного из предыдущего значения с применением делегата `factory`. Операция CAS принимает старое и новое значения и атомарно присваивает объекту `Atom` новое значение только в том случае, если его текущее равно переданному старому значению. Если функция `Swap` не может установить новое значение с помощью `Interlocked.CompareExchange`, то она продолжает повторять попытки, пока операция не завершится успешно.

¹ Подробнее о ключевом слове `volatile` см.: <https://msdn.microsoft.com/en-us/library/x13ttww7.aspx>.

Листинг 3.3. Объект Atom для выполнения инструкций CAS

```
public sealed class Atom<T> where T : class
{
    public Atom(T value)
    {
        this.value = value;
    }

    private volatile T value;
    public T Value => value;

    public T Swap(Func<T, T> factory)
    {
        T original, temp;
        do {
            original = value;
            temp = factory(original);
        }
        while (Interlocked.CompareExchange(ref value, temp, original) != original);
        return original;
    }
}
```

Создание вспомогательного объекта для атомарных инструкций CAS

Получение текущего значения этого экземпляра

Вычисление нового значения, основанного на текущем значении экземпляра

Повторение инструкции CAS до тех пор, пока она не завершится успешно

В листинге 3.4 показано, как использовать класс Atom с объектом `ImmutableDictionary` в контексте серверного концентратора SignalR. В коде реализован только метод `OnConnected`. Такая же концепция применима и для функции `OnDisconnected`.

Листинг 3.4. Потокобезопасная версия ImmutableDictionary с использованием объекта Atom

```
Atom<ImmutableDictionary<Guid, string>> onlineUsers =  
    new Atom<ImmutableDictionary<Guid, string>>  
        (ImmutableDictionary<Guid, string>.Empty); ← Передача объекту Atom пустого  
                                                ImmutableDictionary в качестве  
                                                аргумента для инициализации  
                                                начального состояния  
  
public override Task OnConnected() {  
    Grid connectionId = new Guid (Context.ConnectionId);  
    System.Security.Principal.IPrincipal user = Context.User;  
  
    var temp = onlineUsers.Value; ← Создание временной копии оригинала  
    if(onlineUsers.Swap(d => {  
        if (d.ContainsKey(connectionId)) return d;  
        return d.Add(connectionId, user.Identity.Name);  
    }) != temp) {  
        RegisterUserConnection (connectionId, user.Identity.Name);  
    }  
    return base.OnConnected();  
}  
  
Атомарное обновление базовой неизменяемой коллекции  
с помощью операции обмена, если ключ connectionId не найден  
Регистрация нового пользовательского соединения,  
если оригинал ImmutableDictionary и коллекция,  
возвращенная функцией Swap, различны;  
выполнение обновления
```

Метод Atom Swap обертывает вызов для обновления базового неизменяемого словаря `ImmutableDictionary`. Для того чтобы проверить текущие открытые соединения SignalR, можно в любой момент получить значение свойства Atom Value. Эта операция потокобезопасна, поскольку предназначена только для чтения. Класс Atom является параметризованным, и его можно использовать для атомарного обновления любого типа. Но неизменяемые коллекции имеют специализированный вспомогательный класс, описанный ниже.

Класс `ImmutableInterlocked`

Для того чтобы можно было обновлять неизменяемые коллекции потокобезопасным способом, Microsoft представила класс `ImmutableInterlocked`, расположенный в пространстве имен `System.Collections.Immutable`. Этот класс предоставляет набор функций, обеспечивающих обновление неизменяемых коллекций с применением описанного ранее механизма CAS. Данный класс обеспечивает те же функциональные возможности, что и объект `Atom`. В листинге 3.5 вместо `Dictionary` использован класс `ImmutableDictionary`.

Листинг 3.5. Концентратор, поддерживающий открытые соединения с помощью класса `ImmutableDictionary`

```
static ImmutableDictionary<Guid, string> onlineUsers =
    ImmutableDictionary<Guid, string>.Empty; ← Пустой экземпляр
public override Task OnConnected() { ← класса ImmutableDictionary
    Grid connectionId = new Guid (Context.ConnectionId);
    System.Security.Principal.IPrincipal user = Context.User;
    if(ImmutableInterlocked.TryAdd (ref onlineUsers,
        connectionId, user.Identity.Name)) { ← ImmutableInterlocked пытается
        RegisterUserConnection (connectionId, user.Identity.Name); ← добавить новый элемент
    } ← в неизменяемые коллекции
    return base.OnConnected();
}
public override Task OnDisconnected() {
    Grid connectionId = new Guid (Context.ConnectionId);
    string userName;
    if(ImmutableInterlocked.TryRemove (ref onlineUsers,
        connectionId, out userName)) { ← потокобезопасным способом
        DeregisterUserConnection(connectionId, userName); ←
    } ← ImmutableInterlocked удаляет
    return base.OnDisconnected(); ← элемент. Если элемент существует,
} ← то функция возвращает true
```

Обновление `ImmutableDictionary` выполняется атомарно. Это означает, что пользовательское соединение добавляется только в том случае, если оно не существует. С таким изменением концентратор SignalR работает корректно и не нуждается в блокировках, а на сервере не увеличивается нагрузка на процессор. Но у использования неизменяемых коллекций есть своя цена, и она становится

заметной при частых обновлениях. Например, время, необходимое для добавления в `ImmutableDictionary` 1 млн пользователей, с применением `ImmutableInterlocked` составляет 2,518 с. В большинстве случаев это значение, вероятно, приемлемо, но если вы намерены создать высокопроизводительную систему, то важно исследовать все варианты и выбрать из них наиболее подходящий инструмент.

В целом неизменяемые коллекции идеально подходят в случае разделенных состояний, совместно применяемых несколькими потоками, если количество обновлений невелико. Их значение (состояние) является гарантированно потокобезопасным; его можно безопасно передавать другим потокам. Если же нужна коллекция, способная обрабатывать множество обновлений одновременно, то лучшим решением будет использование конкурентной коллекции .NET.

3.1.2. Конкурентные коллекции .NET: более быстрое решение

В .NET Framework в пространстве имен `System.Collections.Concurrent` представлен ряд потокобезопасных коллекций, предназначенных для упрощения потокобезопасного доступа к разделяемым данным. Конкурентные коллекции — это экземпляры изменяемых коллекций, назначением которых является повышение производительности и масштабируемости многопоточных приложений. Поскольку чтение и обновление таких коллекций может безопасно выполняться несколькими потоками одновременно, то эти коллекции рекомендуется использовать в многопоточных программах вместо аналогичных коллекций из пространства имен `System.Collections.Generic`. В табл. 3.2 показаны конкурентные коллекции, доступные в .NET.

Таблица 3.2. Детали реализации конкурентных коллекций

Конкурентная коллекция	Детали реализации	Метод синхронизации
<code>ConcurrentBag<T></code>	Работает как параметризованный список	При наличии нескольких потоков их доступ координируется с помощью примитивного монитора; в противном случае синхронизация не нужна
<code>ConcurrentStack<T></code>	Параметризованный стек, реализованный с использованием односвязного списка	Без блокировок, с применением технологии CAS
<code>ConcurrentQueue<T></code>	Параметризованная очередь, реализованная с применением связного списка сегментов массива	Без блокировок, с использованием технологии CAS
<code>ConcurrentDictionary<K, V></code>	Параметризованный словарь, реализованный с использованием хеш-таблицы	Без блокировок для операций чтения; для обновлений применяется синхронизация с блокировками

Вернемся к нашему примеру охоты на небезопасный объект в концентраторе SignalR. В этом случае `ConcurrentDictionary` представляет собой лучший вариант, чем `Dictionary`, не обеспечивающий потокобезопасность; из-за частого и большого количества обновлений данный вариант также предпочтительнее, чем `ImmutableDictionary`. Фактически набор `System.Collections.Concurrent` был разработан с целью достижения высокой производительности посредством использования сочетания мелкомодульных¹ шаблонов и шаблонов без блокировок. Эти технологии гарантируют, что потоки, обращающиеся к конкурентной коллекции, будут блокироваться в течение минимального времени или, в некоторых случаях, не будут блокироваться вообще.

`ConcurrentDictionary` обеспечивает масштабируемость при обработке нескольких запросов в секунду. Для присваивания и извлечения значений можно использовать оператор индексирования в виде квадратных скобок, как и в обычном параметризованном словаре, но `ConcurrentDictionary` также предлагает ряд методов, совместимых с конкурентностью, таких как `AddOrUpdate` и `GetOrAdd`. Метод `AddOrUpdate` принимает ключ и значение в качестве параметра, а вторым параметром является делегат. Если такого ключа в словаре нет, то в словарь вставляется новый элемент с заданным ключом и значением. Если ключ в словаре найден, то вызывается делегат и в словаре обновляется значение для заданного ключа. Если действия, выполняемые делегатом, также являются потокобезопасными, то опасность вмешательства другого потока и изменения словаря между чтением одного значения и записью другого исключается.

ПРИМЕЧАНИЕ

Помните, что независимо от того, являются ли методы, представленные в `ConcurrentDictionary`, атомарными и потокобезопасными, данный класс не контролирует делегатов, вызываемых методами `AddOrUpdate` и `GetOrAdd`, — эти делегаты могут быть реализованы без гарантий потокобезопасности.

В листинге 3.6 `ConcurrentDictionary` сохраняет состояние открытых соединений в концентраторе SignalR.

Этот код похож на код из листинга 3.5 с использованием `ImmutableDictionary`, но здесь выше скорость добавления и удаления при большом количестве соединений (`connection`). Например, время, необходимое для добавления 1 млн пользователей в `ConcurrentDictionary`, составляет всего 52 мс, по сравнению с 2,518 с для `ImmutableDictionary`. `ImmutableDictionary`, вероятно, хорошо работает для многих случаев, но если вы хотите создать высокопроизводительную систему, то важно исследовать все варианты и действовать наилучший инструмент.

Следует понимать, как работают эти коллекции. Сначала из-за их изменяемой природы кажется, что они используются без какого-либо стиля ФП. Но коллекции

¹ Подробнее о параллельных вычислениях см.: https://en.wikipedia.org/wiki/Parallel_computing (https://ru.wikipedia.org/wiki/Параллельные_вычисления).

создают внутренний снимок, который имитирует временную неизменяемость, чтобы сохранить потокобезопасность во время итерации, так что данный снимок может быть безопасно вычислен повторно.

Листинг 3.6. Концентратор, поддерживающий открытые соединения с помощью ConcurrentDictionary

```
static ConcurrentDictionary<Guid, string> onlineUsers =  
    new ConcurrentDictionary<Guid, string>(); ← Пустой экземпляр  
  
public override Task OnConnected() { ← Экземпляр onlineUsers класса ConcurrentDictionary  
    Guid connectionId = new Guid (Context.ConnectionId);  
    System.Security.Principal.IPrincipal user = Context.User;  
  
    if(onlineUsers.TryAdd(connectionId, user.Identity.Name)) { ← пытается добавить в словарь новый элемент;  
        RegisterUserConnection (connectionId, user.Identity.Name);  
    } ← если элемент не существует, он добавляется,  
    return base.OnConnected(); ← и пользователь будет зарегистрирован  
}  
  
public override Task OnDisconnected() { ← Экземпляр onlineUsers класса  
    Guid connectionId = new Guid (Context.ConnectionId);  
    string userName;  
    if(onlineUsers.TryRemove (connectionId, out userName)) { ← ConcurrentDictionary удаляет connectionId,  
        DereisterUserConnection(connectionId, userName);  
    } ← если этот элемент существует  
    return base.OnDisconnected();  
}
```

Конкурентные коллекции хорошо работают с алгоритмами, которые реализуют модель поставщика/потребителя¹. Целью шаблона поставщика/потребителя является разделение и балансировка рабочей нагрузки между одним или несколькими поставщиками и одним или несколькими потребителями. Поставщик генерирует данные в независимом потоке и вставляет их в очередь. Потребитель выполняется конкурентно в отдельном потоке, который потребляет данные из очереди. Например, поставщик может загружать изображения и хранить их в очереди, к которой обращается потребитель, выполняющий обработку изображений. Эти две сущности работают независимо одна от другой, и если рабочая нагрузка, поступающая от поставщика, увеличивается, то можно создать нового потребителя, чтобы сбалансировать рабочую нагрузку. Шаблон поставщика/потребителя является одним из наиболее широко используемых шаблонов параллельного программирования; он будет подробно обсуждаться в главе 7, где также будет показан пример его практического применения.

¹ Подробнее о проблеме поставщика-потребителя, также известной как проблема ограниченного буфера, см.: https://en.wikipedia.org/wiki/Producer-consumer_problem.

3.1.3. Шаблон агента передачи сообщений: более быстрое и верное решение

Окончательным решением задачи «Охота на потоконебезопасный объект» было введение в концентратор SignalR локального агента, который создавал возможность асинхронного доступа и тем самым обеспечивал высокую масштабируемость при большом количестве обращений. Агент представляет собой вычислительную единицу, которая обрабатывает сообщения по одному за раз. Сообщения отправляются асинхронно; следовательно, отправителю не приходится ждать ответа, поэтому блокировки не происходит. В данном случае словарь изолирован и доступен только агенту. Агент обновляет коллекцию в одном потоке, что исключает опасность повреждения данных и необходимость блокировки. Такое дополнение масштабируемо, поскольку асинхронный семантический агент способен обрабатывать 3 млн сообщений в секунду, а код работает быстрее, потому что в нем отсутствуют дополнительные служебные данные, которые были ранее необходимы для синхронизации.

Программирование с агентами и передачей сообщений будет подробно рассмотрено в главе 11. Не беспокойтесь, если вы не совсем поймете следующий код; он станет яснее для вас в процессе наших дальнейших исследований, к тому же вы всегда можете обратиться к приложению B. Данный подход требует меньше изменений кода по сравнению с предыдущими решениями, но производительность приложения от этого не пострадала. В листинге 3.7 показана реализация агента на F#.

Листинг 3.7. Написанный на F# агент, обеспечивающий потокобезопасный доступ к изменяемым состояниям

Сообщения для агента
представлены в виде
размеченного объединения

```

→ type AgentMessage =
    | AddIfNoExists of id:Guid * userName:string
    | RemoveIfNoExists of id:Guid
  
```

Полученное
сообщение
сопоставляется
шаблоном
для перехода
к соответствующей
функциональной
ветви

Внутри тела агента даже
изменяемая коллекция является
потокобезопасной, так как она изолирована

Полученное сообщение сопоставляется
с образцом для выбора соответствующей
функциональности

Операция поиска
потокобезопасна,
так как агент выполняет
ее в одном потоке

```

type AgentOnlineUsers() =
    let agent = MailboxProcessor<AgentMessage>.Start(fun inbox ->
        let onlineUsers = Dictionary<Guid, string>()
        let rec loop() = async {
            let! msg = inbox.Receive()
            match msg with
            | AddIfNoExists(id, userName) ->
                let exists, _ = onlineUsers.TryGetValue(id)
                if not exists = true then
                    onlineUsers.Add(id, userName)
                    RegisterUserConnection (id, userName)
            | RemoveIfNoExists(id) ->
                onlineUsers.Remove(id)
        }
        loop()
    )
  
```

```

let exists, userName = onlineUsers.TryGetValue(id)
if exists = true then
    onlineUsers.Remove(id) |> ignore
    DeregisterUserConnection(id, userName)
return! loop() }
loop() )

```

Операция поиска является потокобезопасной, поскольку выполняется однопоточным агентом

В листинге 3.8 представлен рефакторизованный код на C#, в котором реализовано окончательное решение задачи. Вследствие интероперабельности между языками программирования .NET можно написать библиотеку на одном языке, а затем обращаться к ней на другом языке. В данном случае C# обращается к библиотеке, написанной на F#, с помощью кода `MailboxProcessor(Agent)`.

Листинг 3.8. Концентратор SignalR, написанный на C#, с использованием агента, написанного на F#

Использование статического экземпляра агента, написанного на F#, из ссылочной библиотеки

```

static AgentOnlineUsers onlineUsers = new AgentOnlineUsers()

public override Task OnConnected() {
    Guid connectionId = new Guid (Context.ConnectionId);
    System.Security.Principal.IPrincipal user = Context.User;

    onlineUsers.AddIfNoExists(connectionId, user.Identity.Name);
    return base.OnConnected();
}

public override Task OnDisconnected() {
    Guid connectionId = new Guid (Context.ConnectionId);
    onlineUsers.RemoveIfNoExists(connectionId); ← Способы, позволяющие
    return base.OnDisconnected(); ← отправить агенту сообщение
} ← для выполнения потокобезопасных
      операций обновления

```

В результате это окончательное решение позволило устраниТЬ проблему, снизив нагрузку на процессор почти до нуля (рис. 3.2).

Вывод из данного опыта заключается в том, что совместное применение изменяющегося состояния в многопоточной среде не является хорошей идеей. Первоначально коллекция `Dictionary` должна была поддерживать соединения с пользователями, находящимися в сети; изменяемость была почти необходимой. Можно было бы использовать функциональный подход с неизменяемой структурой, но при этом создавать новую коллекцию для каждого обновления, что, вероятно, слишком накладно. Лучшим решением здесь будет применение агента, позволяющего изолировать изменяемость и обеспечить доступ к агенту из вызывающих методов. Это функциональный подход, в котором реализуется естественная потокобезопасность агентов.

Результатом такого подхода является увеличение масштабируемости благодаря асинхронному доступу без блокировки. Это позволяет легко добавлять в тело агента новую логику, например вести журнал и обрабатывать ошибки.

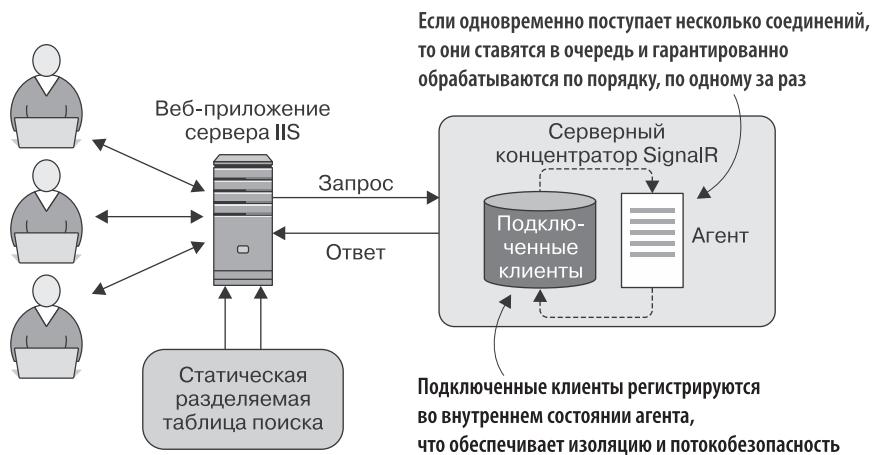


Рис. 3.2. Архитектура веб-сервера для приложения чата с применением концентратора SignalR. Это решение, по сравнению с указанным на рис. 3.1, исключает изменяемый словарь, который совместно использовался несколькими потоками для обработки входящих запросов.

Вместо словаря создан локальный агент, гарантирующий высокую масштабируемость и потокобезопасность в этом многопоточном сценарии

3.2. Безопасное совместное использование потоками функциональных структур данных

Персистентная структура данных (также известная как *функциональная структура данных*) — это структура данных, в которой никакие операции не приводят к постоянным изменениям базовой структуры. *Персистентность* означает, что все версии измененной структуры сохраняются. Другими словами, такая структура данных является неизменяемой, поскольку операции обновления не изменяют структуру данных, а возвращают новую структуру с обновленными значениями.

Персистентность в отношении данных обычно неверно истолковывается как хранение данных в физическом объекте, таком как база данных или файловая система. В ФП функциональная структура данных является долговечной. Большинство традиционных императивных структур данных (таких как `Dictionary`, `List`, `Queue`, `Stack` и др. из `System.Collections.Generic`) кратковременны, их состояние существует только в течение короткого времени между обновлениями. Обновления являются разрушающими, как показано на рис. 3.3.

Функциональная структура данных гарантирует единообразное поведение независимо от того, доступна ли эта структура для нескольких потоков или даже для другого процесса, и при ее использовании не приходится заботиться о возможном изменении данных. Персистентные структуры данных не поддерживают деструктивные обновления; вместо этого они сохраняют свои старые версии.

Разумеется, по сравнению с традиционными императивными структурами данных чисто функциональные структуры данных значительно интенсивнее за-

действуют память, что приводит к существенному ухудшению производительности. К счастью, персистентные структуры данных построены с учетом эффективности; общее состояние между разными версиями структуры данных в них применяется настолько интенсивно, насколько это приемлемо. Такая возможность появляется благодаря неизменяемому характеру функциональных структур данных: поскольку они не могут быть изменены, многократное воздействие разных версий реализуется без усилий. Можно создать новую структуру данных из частей старой, ссылаясь на существующие данные, вместо того чтобы копировать их. Такая технология называется *совместным использованием структур* (см. подраздел 3.3.5). Подобная реализация проще, чем создание новой копии данных каждый раз, когда выполняется обновление, и приводит к повышению производительности.

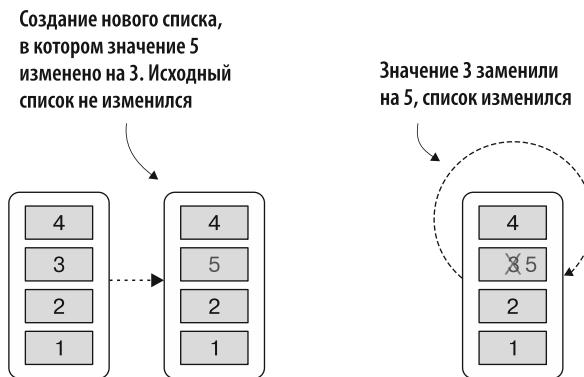


Рис. 3.3. Деструктивное и персистентное обновление списка. Справа показан список, который изменен, чтобы заменить значение 3 на 5, при этом исходный список не сохраняется. Такой процесс называется деструктивным. Слева показан функциональный список: его значения не изменяются, вместо этого создается новый список с обновленным значением

3.3. Наконец-то неизменяемость!

Майкл Фезерс (Michael Feathers) в своей работе «Эффективная работа с устаревшим кодом» дает такое сравнение ООП и ФП.

Объектно-ориентированное программирование делает код понятным, инкапсулируя движущиеся части. Функциональное программирование делает код понятным, сводя количество движущихся частей к минимуму.

*Майкл Фезерс. Эффективная работа с устаревшим кодом
(Prentice Hall, 2004)*

Это означает, что неизменяемость сводит к минимуму изменяемые части кода, что упрощает рассуждения о поведении данных частей. *Неизменяемость* освобождает функциональный код от побочных эффектов. Типичный пример побочного эффекта — разделяемая переменная — является серьезным препятствием для создания

параллельного кода и приводит к недетерминированному выполнению программы. Избавление от побочных эффектов — хороший стиль кодирования.

Например, разработчики .NET Framework решили сделать строки неизменяемыми объектами, используя функциональный подход, чтобы было проще писать хороший код. Как вы помните, неизменяемым называется объект, состояние которого не может быть изменено после его создания. Вам придется потратить какое-то время на то, чтобы принять принцип неизменяемости и научиться писать код в этом стиле; но результат — более понятный синтаксис и более лаконичный код (сокращение ненужного стереотипного кода) — стоит затраченных усилий. Более того, выгодой от такой трансформации данных, по сравнению с изменяемыми данными, будет значительное снижение вероятности ошибок в коде; также упрощается управление взаимодействием и зависимостями между различными частями кодовой базы.

Использование неизменяемых объектов как части модели программирования заставляет каждый поток обрабатывать свою собственную копию данных, что упрощает написание корректного конкурентного кода. Кроме того, теперь вы сможете безопасно организовать одновременный доступ нескольких потоков к разделяемым данным, если этот доступ — только для чтения. Фактически, поскольку вам теперь не нужны блокировки или другие методы синхронизации, опасности возможных взаимных блокировок и возникновения состояния гонки больше не существует (рис. 3.4). Подробно данные технологии обсуждались в главе 1.

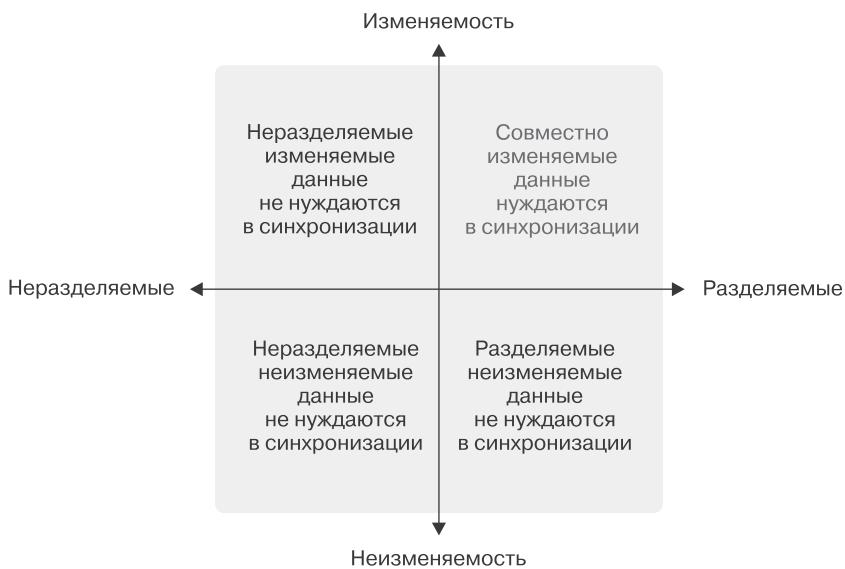


Рис. 3.4. Представление последствий использования изменяемых и неизменяемых состояний, когда они являются разделяемыми или неразделяемыми, в декартовой системе координат

Функциональные языки, такие как F#, по умолчанию являются неизменяемыми, что делает их идеальными в условиях параллелизма. Неизменяемость не заставит ваш код работать быстрее и не сделает программу значительно более масштабиру-

емой, но подготовит код к распараллеливанию посредством незначительных изменений в базе кода.

В объектно-ориентированных языках, таких как C# и Java, бывает трудно писать конкурентные приложения, поскольку в этих языках изменяемость является поведением по умолчанию и не существует инструмента, помогающего предотвратить или компенсировать ее. В императивных языках программирования изменяемые структуры данных считаются совершенно нормальными, и, хотя использование глобальных состояний не рекомендуется, изменяемые состояния обычно применяются в разных частях программы. В параллельном программировании это рецепт катастрофы. К счастью, как уже упоминалось, при компиляции C# и F# оперируют одним и тем же промежуточным языком, что упрощает совместное задействование данными языками одной и той же функциональности. Например, можно определить область и объекты программы на F# и воспользоваться преимуществами типов и краткости в этом языке (что особенно важно, так как его типы по умолчанию являются неизменяемыми). А затем можно разработать программу на C#, используя библиотеку, написанную на F#, которая гарантирует неизменяемое поведение, без дополнительной работы.

Неизменяемость является важным инструментом для построения конкурентных приложений, но использование неизменяемых типов само по себе не ускорит работу программы. Зато оно подготовит код к параллелизму; неизменяемость способствует повышению уровня конкурентности, что на многоядерном компьютере означает более высокую производительность и скорость. Неизменяемые объекты можно безопасно разделять между несколькими потоками, избегая необходимости синхронизации посредством блокировок, что иногда препятствует параллельному запуску программ.

В .NET Framework существует несколько неизменяемых типов. Некоторые из них являются функциональными, другие могут использоваться в многопоточных программах, а есть и такие, что объединяют в себе и то и другое. В табл. 3.3 перечислены характеристики этих типов, подробнее о которых мы поговорим далее в данной главе.

Таблица 3.3. Характеристики неизменяемых типов в .NET Framework

Тип	Язык .NET	Является ли функциональным	Характеристики	Является ли потоко-безопасным	Применение
Список F#	F#	Да	Неизменяемый связанный список с быстрым добавлением и вставкой	Да	Используется в сочетании с рекурсией для построения и перемещения списков из <i>n</i> элементов
Массив	C# и F#	Нет	Нуль-индексированный изменяемый массив, хранящийся в долговременной памяти	Да, с разделением*	Эффективное хранилище данных для быстрого доступа

Продолжение ↗

Таблица 3.3 (продолжение)

Тип	Язык .NET	Является ли функциональным	Характеристики	Является ли потоко-безопасным	Применение
Конкурентные коллекции	C# и F#	Нет	Набор коллекций, оптимизированный для многопоточного чтения и записи	Да	Разделяемые данные в многопоточной программе; отлично подходят для использования в шаблоне «поставщик/потребитель»
Неизменяемые коллекции	C# и F#	Да	Набор коллекций, которые упрощают работу в параллельной вычислительной среде; их значение может свободно передаваться между разными потоками без искажения данных	Да	Сохранение контроля за состоянием при наличии нескольких потоков
Размеченнное объединение (Discriminated Union, DU)	F#	Да	Тип данных, в котором хранится один из нескольких возможных вариантов	Да	Часто используется для моделирования предметной области и представления иерархических структур, таких как абстрактное синтаксическое дерево
Кортеж	C# и F#	Да	Тип, в котором группируются два и более значения любых (возможно, различных) типов	Нет	Используется для возвращения функцией нескольких значений
Кортеж F#	F#	Да	—	Да	—
Тип записи	F#	Да	Представляет агрегаты свойств именованных значений; можно рассматривать как кортеж с именованными членами, к которым можно получить доступ, используя запись через точку	Да	Применяется вместо обычных классов, обеспечивая неизменяемую семантику; как и DU, хорошо подходит для проектирования предметной области и может использоваться на C#*

* Каждый поток работает с отдельной частью массива.

3.3.1. Функциональная структура данных для обеспечения их параллелизма

Неизменяемые структуры данных идеально подходят для их параллелизма, поскольку облегчают разделение данных между в остальном изолированными задачами фактически с нулевым копированием. В сущности, когда несколько потоков параллельно обращаются к разделяемым данным, неизменяемость играет главную роль для безопасной обработки фрагментов данных, которые принадлежат к одной и той же структуре, но кажутся изолированными. Можно достичь того же уровня корректного параллелизма данных, приняв идею функциональной чистоты, то есть использовать вместо неизменяемости функцию, которая позволяет избежать побочных эффектов.

В частности, *чистоте* способствует ключевая функциональность PLINQ. Функция является чистой, если она не имеет побочных эффектов, а возвращаемое ею значение зависит только от входных значений данной функции.

PLINQ — это язык абстракции более высокого уровня. Он располагается поверх многопоточных компонентов, абстрагируя детали нижнего уровня, которые все еще используют упрощенную семантику LINQ. Целью PLINQ является сокращение времени выполнения и повышение общей производительности запроса с применением всех доступных компьютерных ресурсов. (Подробнее о PLINQ читайте в главе 5.)

3.3.2. Влияние неизменяемости на производительность

Некоторые кодировщики считают, что программирование с неизменяемыми объектами неэффективно и значительно снижает производительность. Например, чистый функциональный способ добавить что-то в список означает вернуть новую копию списка с добавленным элементом, оставив исходный список без изменений. Это может привести к увеличению нагрузки на память при сборке мусора. Поскольку каждая модификация возвращает новое значение, сборщику мусора приходится иметь дело с большим количеством короткоживущих переменных. Однако поскольку компилятор знает, что существующие данные неизменяемы, и поскольку данные не будут изменены, компилятор может оптимизировать распределение памяти, многократно используя коллекцию, частично или целиком. Следовательно, влияние неизменяемых объектов на производительность минимально, почти незаметно, так как типичная копия объекта вместо традиционного изменения — это всего лишь поверхностная копия. Таким образом, объекты, на которые ссылается исходный объект, не копируются; копируется только ссылка, представляющая собой небольшую побитовую копию оригинала.

Как появилась сборка мусора в функциональном программировании

В 1959 г., в ответ на проблемы с памятью, обнаруженные в Lisp, Джон Маккарти изобрел сборку мусора. Сборщик мусора пытается удалить мусор и освободить память, занятую объектами, которые больше не полезны для программы. Это форма автоматического управления памятью. Сорок лет спустя сборка мусора была принята в основных языках программирования, таких как Java и C#. Она является усовершенствованной технологией с точки зрения разделяемых структур данных, так как иначе их трудно правильно обслуживать в языках программирования без управления памятью, таких как C и C++, потому что какие-то фрагменты кода должны отвечать за освобождение памяти. Поскольку элементы являются разделяемыми, неясно, какой код должен отвечать за освобождение занимаемой ими памяти. В языках программирования с управлением памятью наподобие C# и F# данный процесс автоматизирован благодаря наличию сборщика мусора.

При современной скорости процессоров это практически нецелесообразная цена за те преимущества, которые достигаются благодаря гарантии потокобезопасности. Стоит учесть, что в настоящее время производительность достигается посредством параллельного программирования, требующего более частого копирования объектов и сильнее нагружающего память.

3.3.3. Неизменяемость в C#

В C# неизменяемость не является конструкцией, поддерживаемой по умолчанию. Но создавать неизменяемые объекты в C# нетрудно; проблема заключается в том, что компилятор не поддерживает такой стиль и программисту приходится самому делать это с помощью кода. Реализация неизменяемости в C# требует дополнительных усилий и усердия. В C# неизменяемый объект может быть создан с использованием ключевого слова `const` или `readonly`.

Ключевым словом `const` может быть декорировано любое поле; единственным предварительным условием является то, что присваивание и объявление должны происходить в одной строке. После объявления и присваивания значения поле `const` не может быть изменено; оно принадлежит уровню класса, и обращаться к нему надо напрямую, а не через экземпляр этого класса.

Другой вариант, декорирование значения с помощью ключевого слова `readonly`, может быть реализован как в описании класса, так и через конструктор при создании экземпляра класса. После инициализации поля, помеченного как `readonly`, значение поля не может быть изменено, и оно является доступным через экземпляр класса. Что более важно, для того чтобы объект остался неизменным, в случае если требуется изменение свойств или состояния, нужно создать новый экземпляр исходного объекта с обновленным состоянием. Помните, что объекты `readonly` в C# выступают неизменяемыми объектами первого уровня и представляют собой лишь неглубокие

копии. В C# объект является неглубокой копией, и неизменяемость в нем гарантируется не всем его полям и свойствам, а только самому объекту. Если у объекта Person есть свойство Address, пред назначенное только для чтения, и это свойство, в свою очередь, представляет собой сложный объект с такими свойствами, как улица, город и почтовый индекс, то данные свойства не наследуют неизменяемое поведение, если только они не помечены как предназначенные только для чтения. И наоборот, неизменяемый объект, все поля и свойства которого помечены как предназначенные только для чтения, является глубоко неизменяемым.

В листинге 3.9 представлен неизменяемый класс Person, реализованный на C#.

Листинг 3.9. Неизменяемый класс Person на C# — поверхностная копия

```
class Address{
    public Address(string street, string city, string zipcode){
        Street = street;
        City = city;
        ZipCode = zipcode;
    }
    public string Street;
    public string City;
    public string ZipCode;
}

class Person {
    public Person(string firstName, string lastName, int age,
    ➔ Address address){
        FirstName = firstName;
        LastName = lastName;
        Age = age;
        Address = address;
    }
    public readonly string FirstName;
    public readonly string LastName;
    public readonly int Age;
    public readonly Address Address;
}
```

Поля объекта Address, не помеченные как предназначенные только для чтения

Поля объекта Person, помеченные как предназначенные только для чтения

В этом коде объект Person является поверхностной копией, поскольку, несмотря на то что поле Address недоступно для изменения (оно помечено как предназначение только для чтения), поля, принадлежащие полю Address, могут быть изменены. Фактически можно создать экземпляр объекта Person с измененным полем Address:

```
Address address = new Address("Brown st.", "Springfield", "55555");
Person person = new Person("John", "Doe", 42, address);
```

Теперь если вы попытаетесь изменить поле Address, то компилятор выдаст исключение (выделено жирным шрифтом), но вы все равно сможете изменить поля объекта address.ZipCode:

```
person.Address = // Ошибка
person.Address.ZipCode = "77777";
```

Это пример объекта — поверхностной копии. Компания Microsoft осознала всю важность программирования на основе неизменяемости в современном контексте и представила в C# 6.0 технологию, позволяющую легко создавать неизменяемые классы (листинг 3.10). Данная технология, называемая *автоматически назначаемыми свойствами, предназначеными только для геттеров*, позволяет объявлять автоматически назначаемые свойства без использования метода-сеттера, так что неявно создается поле, предназначенное только для чтения, с ключевым словом `readonly`. При этом, к сожалению, реализуется поведение поверхностной копии.

Листинг 3.10. Неизменяемый класс на C# с автоматически назначаемыми свойствами, предназначенными только для геттеров

```
class Person {
    public Person(string firstName, string lastName, int age,
    ➔ Address address){
        FirstName = firstName;
        LastName = lastName;
        Age = age;
        Address = address;
    }

    public string FirstName {get;}
    public string LastName {get;}
    public int Age {get;}
    public Address Address {get;}
```

Свойства, предназначенные только для геттеров,
присваиваются непосредственно полям
объекта Person из конструктора

```
➔ public Person ChangeFirstName(string firstName) {
    return new Person(firstName, this.LastName, this.Age, this.Address);
}

➔ public Person ChangeLastName(string lastName) {
    return new Person(this.FirstName, lastName, this.Age, this.Address);
}

➔ public Person ChangeAge(int age) {
    return new Person(this.FirstName, this.LastName, age, this.Address);
}

➔ public Person ChangeAddress(Address address) {
    return new Person(this.firstName, this.LastName, this.Age, address);
}
```

Функции, позволяющие обновлять поля объекта Person
путем создания нового экземпляра без изменения оригинала

Важно заметить, что в такой неизменяемой версии класса `Person` методы, отвечающие за обновление полей `FirstName`, `LastName`, `Age` и `Address`, не изменяют состояния; вместо этого они создают новый экземпляр `Person`. В ООП объекты создаются путем вызова конструктора, после чего устанавливается значение состояния объекта путем обновления его свойств и вызова методов. Такой подход приводит к неудобному и слишком подробному синтаксису построения объектов.

Именно здесь вступают в действие функции серии `Change`, добавленные для изменения свойств объекта `Person`. Используя данные функции, можно применять шаблон «цепочка», также известный как *текущий интерфейс*. Ниже представлен пример такого шаблона, задействованного для создания экземпляра класса `Person` с изменением возраста и адреса.

```
Address newAddress = new Address("Red st.", "Gotham", "123459");
Person john = new Person("John", "Doe", 42, address);
Person olderJohn = john.ChangeAge(43).ChangeAddress(newAddress);
```

Итак, чтобы на C# класс стал неизменным, нужно сделать следующее.

- ❑ Всегда создавайте класс с помощью конструктора, который принимает аргумент, используемый для присваивания состояния объекта.
- ❑ Сделайте поля предназначеными только для чтения и применяйте свойства без публичного сеттера; значения будут присваиваться в конструкторе.
- ❑ Избегайте любых методов, позволяющих изменить внутреннее состояние класса.

3.3.4. Неизменяемость в F#

Как уже отмечалось, язык программирования F# неизменяемый по умолчанию. Следовательно, в нем понятие переменной не существует, поскольку по определению если переменная неизменяется, то она не является переменной. В F# вместо переменных используются идентификаторы, которые ассоциируют (связывают) значение с применением ключевого слова `let`. После того как такая связь установлена, значение не может измениться. Помимо полного набора неизменяемых коллекций, в F# есть встроенные наборы полезных неизменяемых конструкций, предназначенных для чисто функционального программирования (листинг 3.11). Эти встроенные типы — `tuple` и `record`. По сравнению с типами CLI у них есть следующие преимущества:

- ❑ они неизменяемы;
- ❑ их значением не может быть `null`;
- ❑ у них есть встроенная возможность установить структурное равенство и функция сравнения.

В листинге 3.11 демонстрируется использование неизменяемого типа в F#.

Листинг 3.11. Неизменяемые типы F

Кортеж, который
определяет два значения

```
→ let point = (31, 57) ← Можно деконструировать этот кортеж
    let (x,y) = point     и получить доступ к его значениям
```

```
type Person= { First : string; Last: string; Age:int} ← Описание типа объекта Person
let person = { First="John"; Last="Doe"; Age=42} ← с помощью типа записи
```

Экземпляр объекта Person,
в котором используется тип записи

Тип `tuple` представляет собой набор неименованных упорядоченных значений, которые могут принадлежать к разным гетерогенным типам (https://en.wikipedia.org/wiki/Homogeneity_and_heterogeneity). Преимущество `tuple` заключается в том, что данный тип можно использовать сразу же; он идеально подходит для определения временных и легких конструкций, содержащих произвольное количество элементов. Например, `(true, "Hello", 2, 3, 14)` – это кортеж из четырех элементов.

Тип записи `record` похож на кортеж, где все элементы помечены, так что у каждого из значений есть имя. Преимущество записи перед кортежем заключается в том, что метки помогают различать элементы и писать в документации, для чего предназначен каждый из них. Более того, свойства определенных полей записи создаются автоматически. Это удобно, поскольку экономит нажатия клавиши. Запись в F# можно представить как класс C#, у которого все свойства предназначены только для чтения. Наиболее ценной является возможность корректно и быстро реализовывать на C# неизменяемые классы с помощью данного типа. Фактически вы можете создать в своем решении библиотеку на F#, построив модель предметной области с использованием типа `record`, а затем ссылаться на эту библиотеку в проекте на C#. Вот как выглядит код на C#, который ссылается на библиотеку F# через тип `record`:

```
Person person = new Person("John", "Doe", 42)
```

Это простой и эффективный способ создания неизменяемого объекта. Кроме того, для реализации на F# требуется только одна строка кода, а для ее эквивалента на C# – 11 строк с использованием полей, предназначенных только для чтения.

3.3.5. Функциональные списки: объединение ячеек в цепочки

Наиболее распространенной и общепринятой функциональной структурой данных является список – набор однородных типов, применяемых для хранения произвольного количества элементов. В ФП списки представляют собой рекурсивные структуры данных, состоящие из двух взаимосвязанных элементов: `Head` или `Cons` и `Tail`. Назначение `Cons` – предоставить механизм для хранения значения и связи с другими элементами `Cons` через ссылку на объект. Эта ссылка называется указателем `Next`.

У списков также существует специальное состояние, называемое `nil` и соответствующее списку без элементов, последнему звену, которое ни на что не ссылается. Состояние `nil`, или пустой элемент, удобно при рекурсивном прохождении списка, так как указывает на его конец. На рис. 3.5 показан список, построенный из четырех элементов `Cons` и пустого списка. Каждая ячейка (`Head`) содержит число и ссылку на остальной список (`Tail`) – и так до последней ячейки `Cons`, которая определяет пустой список. Эта структура данных похожа на односвязный список (https://ru.wikipedia.org/wiki/Связный_список), где каждый узел цепочки имеет единственную ссылку на другой узел, так что набор связанных узлов образует цепочку.

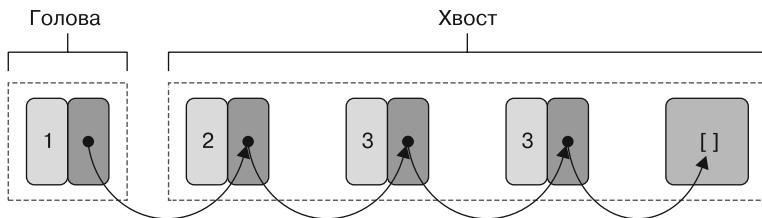


Рис. 3.5. Функциональный список целых чисел, состоящий из четырех чисел и пустого списка (последнее поле [] справа). Каждый элемент имеет ссылку (черная стрелка), связанную с остальной частью списка. Первый элемент слева — это голова списка, связанная с остальной его частью (хвостом)

В функциональных списках операции по добавлению новых и удалению существующих элементов не меняют текущую структуру, а возвращают новую, содержащую обновленные значения. По своей внутренней реализации неизменяемые коллекции могут безопасно обмениваться общими структурами, что ограничивает расход памяти. Эта технология называется *совместным использованием структур*. На рис. 3.6 показано, как совместное использование структур минимизирует потребление памяти при генерации и обновлении функциональных списков.

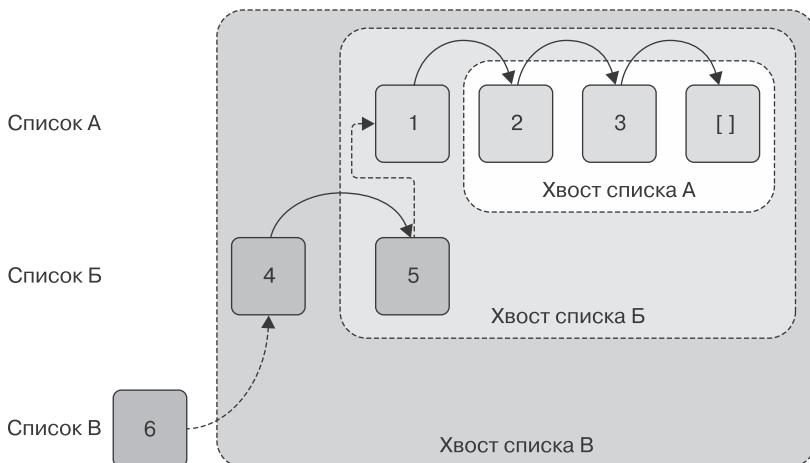


Рис. 3.6. Технология совместного использования структур позволяет оптимизировать пространство в памяти при создании новых списков. Список А состоит из трех элементов и пустой ячейки, список Б — из пяти элементов, а список В — из шести. Каждый элемент связан с остальной частью списка. Например, головной элемент списка Б — это цифра 4, которая связана с хвостом списка (цифры 5, 1, 2, 3 и [])

На рис. 3.6 список А состоит из трех чисел и пустого списка. Если добавить в список А два новых элемента, то технология совместного использования структур создаст впечатление, что построен новый список Б. Однако на самом деле указатель

от двух элементов списка Б связывается с немодифицированным списком А. Тот же сценарий повторяется для списка В. После этого доступны все три списка (А, Б и В), каждый со своими элементами.

Очевидно, что функциональные списки предназначены для обеспечения более высокой производительности путем добавления или удаления элементов в голове списка. На самом деле списки хорошо работают при линейном обходе, а добавление выполняется за постоянное время $O(1)$, потому что новый элемент добавляется в голову предыдущего списка. Однако при случайном доступе такая структура неэффективна, поскольку при каждом поиске приходится просматривать список слева направо, что требует времени $O(n)$, где n — количество элементов в коллекции.

Нотация Big O

Нотация Big O, также известная как *асимптотическая нотация*, представляет собой способ оценки производительности алгоритмов, основанный на размерности проблемы. Здесь слово «*асимптотическая*» описывает поведение функции, если ее входная размерность приближается к бесконечности. Если есть список из 100 элементов, то добавление нового элемента в начало такого списка потребует постоянного времени $O(1)$, потому что операция занимает всего один шаг, независимо от размера списка. И наоборот, при поиске элемента стоимость операции составит $O(100)$, поскольку при наихудшем сценарии, чтобы найти элемент, потребуется 100 проходов по списку. Размер проблемы обычно обозначается как n , а сложность оценивается как $O(n)$.

А как быть со сложностью параллельных программ?

Нотация Big O показывает сложность последовательного выполнения алгоритма, но в случае параллельной программы такое измерение неприменимо. Однако можно описать сложность параллельного алгоритма, введя параметр P , представляющий количество ядер компьютера. Например, при параллельном поиске сложность составляет $O(n/P)$, потому что можно разбить список на сегменты, по одному для каждого ядра, и выполнять поиск одновременно.

В следующем списке представлены наиболее распространенные типы сложности по порядку, начиная с наименее дорогостоящих.

- $O(1)$ — постоянная сложность. Время всегда равно 1, независимо от объема входных данных.
- $O(\log n)$ — логарифмическая сложность. Время возрастает как мантисса логарифма от количества входных данных.
- $O(n)$ — линейная сложность. Время линейно возрастает в зависимости от количества входных данных.
- $O(n \log n)$ — логарифмически-линейная сложность. Время возрастает пропорционально объему входных данных, умноженному на его мантиссу.
- $O(n^2)$ — квадратичная сложность. Время возрастает пропорционально квадрату объема входных данных.

Новый список создается путем добавления элемента в начало существующего списка. Для этого в качестве начального значения выбирается пустой список, и затем новый элемент связывается с существующей структурой списка. Данная операция с `Cons` в голове списка повторяется для всех элементов, и таким образом каждый список завершается пустым состоянием.

Одной из главных прелестей функциональных списков является та легкость, с которой их можно использовать для написания потокобезопасного кода. Фактически функциональные структуры данных могут передаваться по ссылке в вызывающую функцию без риска искажения, как показано на рис. 3.7.

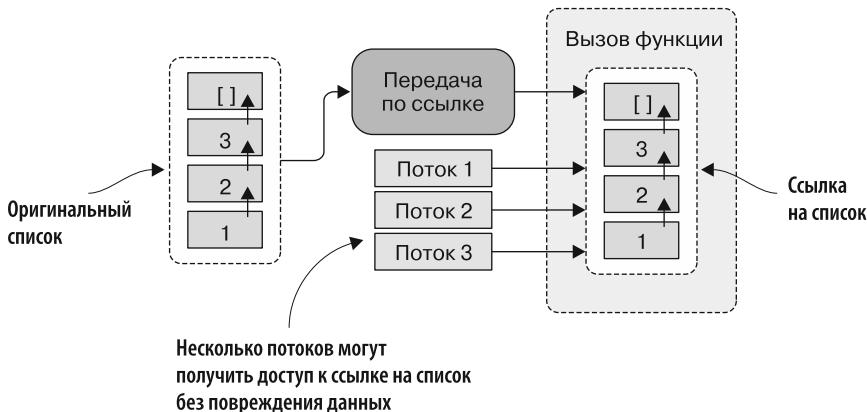


Рис. 3.7. Список передается в вызывающую функцию по ссылке. Поскольку список является неизменяемым, несколько потоков могут получить доступ к ссылке без повреждения данных

По определению, потокобезопасный объект должен сохранять согласованное состояние каждый раз, когда к нему обращаются. Например, вы не должны застать момент, когда при удалении элемента из коллекции данных эта коллекция будет находиться в середине процесса изменения размера. В многопоточной программе выполнение операций над изолированной частью функциональной структуры данных является отличным безопасным способом избежать их разделения.

Функциональные списки в F#

В F# имеется встроенная реализация неизменяемой структуры списка, представленной в виде *связанного списка* (линейная структура данных, состоящая из набора элементов, соединенных в цепочку). Каждый программист хоть раз в жизни писал связанный список. Однако в случае функциональных списков реализация требует немного больше усилий, чтобы гарантировать неизменяемое поведение, при котором однажды созданный список никогда не изменится. К счастью, благодаря поддержке алгебраических типов данных (algebraic data types, ADT) (https://ru.wikipedia.org/wiki/Алгебраический_тип_данных), которые позволяют определить параметризованный рекурсивный тип `List`, в F# списки представлены очень просто.

АДТ является составным типом — иначе говоря, его структура представляет собой сочетание других типов. В F# данные АДТ называются *размеченными объединениями* (discriminated unions, DU) и представляют собой идеальный инструмент моделирования для представления четко определенных наборов данных в рамках одного типа. В DU эти различные формы данных называются *вариантами*.

Рассмотрим представление модели автомобиля, где типы *Car* и *Truck* относятся к одному и тому же базовому типу *Vehicle*. Размеченные объединения хорошо подходят для построения сложных структур данных (таких как связные списки и различные деревья), поскольку являются более простой альтернативой иерархии малых объектов. Например, ниже представлено определение DU для объекта *Vehicle*.

```
type Vehicle=
| Motorcycle of int
| Car of int
| Truck of int
```

Можно рассматривать DU как механизм, который по сравнению с типом предоставляет дополнительное семантическое значение. Например, представленный выше DU можно читать как «тип транспортного средства, которое может быть автомобилем, мотоциклом или грузовиком».

Для того чтобы создать такое же представление в C#, нужно использовать базовый класс *Vehicle* с производными типами *Car*, *Truck* и *Motorcycle*. Но основные возможности DU заключаются в том, что в сочетании с шаблоном легко выбирается ветвь соответствующих вычислений в зависимости от распознанного варианта. Следующая функция F# печатает количество колес для переданного ей транспортного средства.

```
let printWheels vehicle =
match vehicle with
| Car(n) -> Console.WriteLine("Car has {0} wheels", n)
| Motorcycle(n) -> Console.WriteLine("Motorcycle has {0} wheels", n)
| Truck(n) -> Console.WriteLine("Truck has {0} wheels", n)
```

В этом листинге представлен рекурсивный список, построенный на F# с помощью размеченных объединений и удовлетворяющий определению, сформулированному в предыдущем разделе. Список может быть пустым или образованным из элемента и уже существующего списка (листинг 3.12).

Листинг 3.12. Представление списка на F# с использованием размеченных объединений

```
type FList<'a> =
| Empty
| Cons of head:'a * tail:FList<'a>

let rec map f (list:FList<'a>) =
match list with
| Empty -> Empty
| Cons(hd,tl) -> Cons(f hd, map f tl)

let rec filter p (list:FList<'a>) =
```

Пустой вариант

Вариант Cons имеет головной элемент и хвост

Рекурсивная функция, которая проходит по списку с использованием шаблона сопоставления для деконструирования и выполняет преобразование каждого элемента

```
match list with
| Empty -> Empty
| Cons(hd,tl) when p hd = true -> Cons(hd, filter p tl)
| Cons(hd,tl) -> filter p tl
```

Теперь можно создать новый список целых чисел следующим образом:

```
let list = Cons (1, Cons (2, Cons(3, Empty)))
```

В языке F# есть встроенный параметризованный тип `List`, который позволяет переписать реализованный ранее `FList` одним из следующих двух (эквивалентных) способов:

```
let list = 1 :: 2 :: 3 :: []
let list = [1; 2; 3]
```

Тип `List` на F# реализован как односвязный список, который обеспечивает мгновенный доступ к голове списка $O(1)$ и за линейное время $O(n)$ — доступ к произвольному элементу, где (n) — индекс этого элемента.

Функциональные списки в C#

В ООП существует несколько способов представления функционального списка. Решение, принятое в C#, — это параметризованный класс `FList<T>`, позволяющий хранить значения любого типа. Для определения заголовочного элемента и связанного с ним хвостового списка `Flist<T>` в данном классе содержатся автоматически назначаемые свойства, предназначенные только для чтения из геттеров. Свойство `IsEmpty` показывает, содержит ли текущий экземпляр хотя бы одно значение. В листинге 3.13 показана полная реализация такого списка.

Листинг 3.13. Функциональный список на C#

```
public sealed class Flist<T>
{
    private FList(T head, FList<T> tail) ← Создание списка со значением
    {
        Head = head;
        Tail = tail.IsEmpty
            ? FList<T>.Empty : tail;
        IsEmpty = false;
    }
    private FList() ← Создание нового пустого списка
    {
        IsEmpty = true;
    }
    public T Head { get; } ← Свойство Head возвращает
    public FList<T> Tail { get; } ← первый элемент списка
    public bool IsEmpty { get; } ← Свойство Tail возвращает
    public static FList<T> Cons(T head, Flist<T> tail) ← остальную часть связного списка
    {
        return tail.IsEmpty
            ? new FList<T>(head, Empty)
            : tail;
    }
}
```

Это свойство показывает состояние списка

Статический метод гарантирует более удобный синтаксис для создания списков

```

        : new FList<T>(head, tail);
    }
    public FList<T> Cons(T element)
    {
        return FList<T>.Cons(element, this);
    }
    public static readonly FList<T> Empty = new FList<T>(); ← Статический конструктор
}

```

Функция `Cons` обеспечивает удобную семантику для привязки элемента к списку

создает пустой список

В классе `FList<T>` есть приватный конструктор для принудительного создания объекта с использованием либо статического вспомогательного метода `Cons`, либо статического поля `Empty`. Эта последняя опция возвращает пустой экземпляр объекта `FList<T>`, который может применяться для добавления новых элементов с помощью метода экземпляра `Cons`. Задействуя структуру данных `FList<T>`, можно создавать функциональные списки на C# следующим образом:

```

FList<int> list1 = FList<int>.Empty;
FList<int> list1 = list1.Cons(1).Cons(2).Cons(3);
FList<int> list1 = FList<int>.Cons(1, Flist<int>.Empty);
FList<int> list1 = list2.Cons(2).Cons(3);

```

В этом примере кода показано несколько важных свойств построения списка `FList` для целых чисел. Список `list1` создается из исходного состояния `empty list` с использованием поля `Empty` `FList<int>.Empty`, которое является типичным шаблоном, часто применяемым в неизменяемых структурах данных. Затем из этого начального состояния можно, задействуя свободный семантический подход, связать набор значений `Cons` и создать коллекцию `list2`, как продемонстрировано в приведенном примере.

«Ленивые» значения в функциональных списках

В главе 2 было показано, что «ленивые» вычисления — отличное решение, позволяющее избежать лишних повторяющихся операций путем запоминания результатов работы. Более того, код с «ленивыми» вычислениями имеет дополнительные преимущества при потокобезопасной реализации. Эта технология может быть полезна в контексте функциональных списков, так как позволяет отсрочить вычисления и, следовательно, повысить производительность. В языке F# «ленивые» *переходники* (отложенные вычисления) создаются с использованием ключевого слова `lazy`:

```
let thunkFunction = lazy(21 * 2)
```

В листинге 3.14 представлена реализация параметризованного «ленивого» списка.

Для более эффективной обработки пустых состояний в реализации «ленивого» списка «ленивые» вычисления, выполняемые в конструкторе `Cons`, переносятся в хвост, что повышает производительность для последовательных структур данных. Например, выполнение операции `append` задерживается до тех пор, пока из списка не будет извлечен его головной элемент.

Листинг 3.14. Реализация «ленивого» списка на F#

```

Функция append рекурсивно добавляет элемент в список;
ее можно применять для соединения двух списков

type LazyList<'a> =
    | Cons of head:'a * tail:Lazy<'a LazyList> ← Использование DU для определения
    | Empty                                     списка с «ленивым» вычислением хвоста

let empty = lazy(Empty) ← Применение вспомогательной функции
                           для представления пустого списка

let rec append items list =
    match items with
    | Cons(head, Lazy(tail)) -> ← Функция, которая добавляет
        Cons(head, lazy(append tail list))   элемент в начало списка
    | Empty -> list

let list1 = Cons(42, lazy(Cons(21, empty))) ← Создание списка
// val list1: LazyList<int> = Cons (42,Value is not created.) из двух элементов: 42 и 21

let list = append (Cons(3, empty)) list1 ← Добавление значения 3
// val list : LazyList<int> = Cons (3,Value is not created.) в созданный ранее список list1

let rec iter action list = ← Использование функции
    match list with                         для рекурсивного прохода по списку
    | Cons(head, Lazy(tail)) ->
        action(head)
        iter action tail
    | Empty -> ()                           Печать значений списка

list |> iter (printf "%d .. ") ← с помощью функции iter
// 3 .. 42 .. 21 ..

```

3.3.6. Построение персистентной структуры данных: неизменяемое бинарное дерево

В этом разделе вы научитесь строить бинарное дерево (В-дерево) на F#, используя рекурсию и многопоточные процессы. Выражаясь простым языком, *древовидная структура* представляет собой набор узлов, соединенных таким образом, что между ними не образуется замкнутых циклов. Как правило, деревья задействуются там, где важна производительность. (Странно, что в пространствах имен коллекций .NET Framework до сих пор не появилось дерево.) Деревья являются наиболее распространенными и полезными структурами данных в компьютерном программировании и служат основной концепцией в функциональных языках программирования.

Дерево представляет собой полиморфную рекурсивную структуру данных, содержащую произвольное количество поддеревьев — деревьев в дереве. Эта структура используется главным образом для упорядочения данных на основе ключей, что делает ее эффективным инструментом поиска. Вследствие их рекурсивного определения

к деревьям лучше всего прибегать для представления иерархических структур, таких как файловая система или база данных. Кроме того, деревья считаются перспективными структурами данных, которые часто применяются в таких областях, как машинное обучение и проектирование компиляторов. В ФП основным средством построения данных, позволяющим обходить их структуру, является рекурсия, что делает ее дополнительным преимуществом для обработки деревьев.

Деревья используются для представления иерархии и построения сложных структур на основе простых отношений. Такие структуры используются для разработки и реализации множества эффективных алгоритмов. Типичные примеры применения деревьев в XML/Markup — это синтаксический анализ, поиск, сжатие, сортировка, обработка изображений, социальные сети, машинное обучение и деревья принятия решений. Последние широко реализуются в таких областях, как прогнозирование, финансы и игры.

Способность описать дерево, в котором каждый узел может иметь произвольное количество ветвей, например n -арное и В-дерево, оказывается скорее препятствием, чем преимуществом. В этом разделе рассматривается В-дерево — самобалансирующееся дерево, каждый узел которого может иметь от нуля до двух дочерних узлов, а разница в глубине (называемая высотой) дерева между любыми его листьями не превышает единицы. *Глубиной узла* называется количество ребер от данного узла до корневого. В В-дереве каждый узел указывает на два других узла, называемых левым и правым дочерними узлами.

На рис. 3.8 изображено более наглядное представление дерева, где показаны основные свойства этой структуры данных.

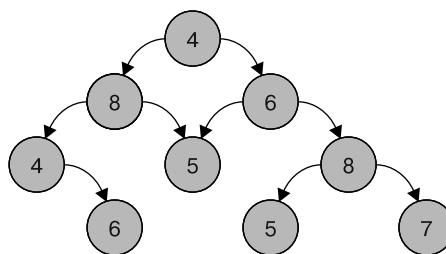


Рис. 3.8. Представление бинарного дерева, каждый узел которого имеет от нуля до двух дочерних узлов. На данном рисунке узел 4 является корневым. Из него выходят две ветви, узлы 8 и 6. Левая ветвь — это ссылка на левое поддерево, а правая ветвь — ссылка на правое поддерево. Узлы без дочерних элементов, 6, 5, 5, 7, называются листьями

У дерева есть специальный узел, называемый *корнем* и не имеющий родителя (узел 4 на рис. 3.8). Корень может быть либо листом, либо узлом с двумя или более дочерними элементами. У каждого родительского узла есть хотя бы один дочерний узел, а у каждого дочернего узла есть один родитель. Узлы без дочерних элементов называются *листьями* (узлы 6, 5, 5, 7 на рисунке), а дочерние узлы одного и того же родителя — *сiblingами*.

Реализация В-деревьев на функциональном языке F#. Реализовать древовидную структуру на F# легко — благодаря поддержке ADT и размеченных объединений. В данном случае DU обеспечивает естественный функциональный способ представления дерева. В листинге 3.15 показано параметризованное определение бинарного дерева на основе DU с особым случаем для пустых ветвей.

Листинг 3.15. Представление неизменяемого В-дерева на F#

```
type Tree<'a> = 
    | Empty                                ← Пустой вариант
    | Node of leaf:'a * left:Tree<'a> * right:Tree<'a>           ←

let tree = 
    Node (20, 
        Node (9, Node (4, Node (2, Empty, Empty), Empty),
               Node (10, Empty, Empty)),
        Empty)                                     ← Экземпляр дерева целых чисел

    Применение варианта узла, который описывает лист
    с параметризованным значением и рекурсивно
    строит ветви левого и правого поддеревьев
```

Элементы В-дерева сохраняются с помощью конструктора типа `Node`, а идентификатор варианта `Empty` представляет собой пустой узел, который не содержит информацию о типе. Вариант `Empty` служит идентификатором местозаполнителя. С помощью такого определения В-дерева можно создавать вспомогательные функции для вставки и верификации элементов дерева. Эти функции реализованы на простом F# с использованием рекурсии и шаблона сопоставления с образцом (листинг 3.16).

Листинг 3.16. Вспомогательные рекурсивные функции для В-дерева

```
let rec contains item tree = 
    match tree with
    | Empty -> false
    | Node(leaf, left, right) ->
        if leaf = item then true
        elif item < leaf then contains item left
        else contains item right

    Использование рекурсии
    для определения функций,
    обходящих древовидную структуру

let rec insert item tree = 
    match tree with
    | Empty -> Node(item, Empty, Empty)
    | Node(leaf, left, right) as node ->
        if leaf = item then node
        elif item < leaf then Node(leaf, insert item left, right)
        else Node(leaf, left, insert item right)

let ``exist 9`` = tree |> contains 9
let ``tree 21`` = tree |> insert 21
let ``exist 21`` = ``tree 21`` |> contains 21
```

Поскольку дерево является неизменяемым, функция вставки возвращает новое дерево с копией только тех узлов, которые находятся на пути вставляемого узла. В функциональном программировании обход дерева DU, чтобы просмотреть все узлы, требует использования рекурсивной функции. Есть три основных способа обхода дерева: в прямом порядке, центрированном и обратном (https://ru.wikipedia.org/wiki/Обход_дерева). Например, при обходе дерева в центрированном порядке сначала обрабатываются узлы, расположенные с левой стороны от корня, затем сам корень и в конце узлы, находящиеся справа, как показано в листинге 3.17.

Листинг 3.17. Функция обхода дерева в центрированном порядке

```
let rec inorder action tree =
  seq {
    match tree with
    | Node(leaf, left, right) ->
        yield! inorder action left
        yield action leaf
        yield! inorder action right
    | Empty -> ()
  }
tree |> inorder (printfn "%d") |> ignore
```

←

Использование функции, которая обходит
древовидную структуру от корня по левым подузлам,
а затем переходит к правым подузлам

←

Применение функции `inorder`
для печати всех значений узлов дерева

Функция `inorder` принимает в качестве аргумента функцию, которая применяется к каждому значению дерева. В данном примере это анонимная лямбда-функция, которая печатает хранящееся в дереве целое число.

3.4. Рекурсивные функции: естественный способ итерирования

Рекурсией называется обманчиво простая концепция программирования — вызов функцией самой себя. Вы когда-нибудь стояли между двумя зеркалами? Кажется, что отражения будут множиться вечно — это и есть рекурсия. Функциональная рекурсия — естественный способ итерирования в ФП, поскольку она позволяет избежать изменения состояния. На каждой итерации в конструктор цикла передается новое значение вместо обновления (изменения) старого. Кроме того, если в программе может быть использована рекурсивная функция, то это сделает программу более модульной, а также предоставит возможности для распараллеливания.

Рекурсивные функции хорошо читаются и обеспечивают эффективную стратегию для решения сложных задач путем разбиения их на более мелкие, хотя и одинаковые, подзадачи. (Это как матрешка, где каждая внутренняя куколка в точности такая же, как и внешняя, только меньше.) Задача в целом может показаться сложной для решения, но меньшие задачи проще решать напрямую, при-

меняя одну и ту же функцию к каждой из них. Возможность разделить задачу на более мелкие, которые могут выполняться по отдельности, делает рекурсивные алгоритмы хорошими кандидатами на распараллеливание. Данный шаблон, также называемый «разделяй и властвуй»¹, приводит к динамическому параллелизму задач, при котором задачи поступают на вычисление по мере прохождения итераций. Для получения дополнительной информации рассмотрите пример в подразделе 1.4.3. Стратегия «разделяй и властвуй» естественным образом применяется для решения задач с рекурсивными структурами данных благодаря присущему ей потенциальну конкурентности.

При рассмотрении рекурсии многие разработчики опасаются падения производительности за время выполнения большого количества итераций, а также исключения `Stackoverflow`. Правильный способ записи рекурсивных функций — использование технологий хвостовой рекурсии и CPS (Continuation Passing Style). Как будет показано в следующих примерах, обе стратегии предоставляют хороший способ минимизировать потребление стека и повысить скорость.

3.4.1. Хвост правильной рекурсивной функции: Оптимизация хвостового вызова

Хвостовой вызов, также известный как оптимизация хвостового вызова (Tail-Call Optimization, TCO), — это вызов подпрограммы, выполняемый как последнее действие процедуры. Если хвостовой вызов может привести к тому, что одна и та же подпрограмма будет снова вызвана в составе цепочки вызовов, то такая подпрограмма называется *хвостовой рекурсивной функцией*, что является частным случаем рекурсии. *Хвостовая рекурсия* — технология, которая преобразует обычную рекурсивную функцию в оптимизированную, способную обрабатывать большие объемы входных данных без каких-либо рисков и побочных эффектов.

ПРИМЕЧАНИЕ

Основной причиной использования хвостового вызова для оптимизации является улучшенная локальность данных, более рациональное применение памяти и кэша. При хвостовом вызове вызываемая функция задействует то же пространство стека, что и вызывающая, и это уменьшает нагрузку на память. При таком вызове немного улучшается использование кэша, поскольку одна и та же память многократно применяется для последующих вызывающих функций и данные могут оставаться в кэше, вместо того чтобы вытеснять более старую строку кэша, чтобы освободить место для новой строки.

¹ Шаблон «разделяй и властвуй» решает задачу путем ее рекурсивного разделения на подзадачи, решения каждой из них независимо от других, а затем снова соединения этих частичных решений в решение исходной задачи.

При хвостовой рекурсии в возвращаемой функции не остается операций, которые надо выполнить после вызова, — последней операцией является вызов функцией самой себя. В качестве примера мы выполним рефакторинг функции вычисления факториала и преобразуем ее в функцию с оптимизированным хвостовым вызовом. В листинге 3.18 показана реализация оптимизированной рекурсивной функции с хвостовым вызовом.

Листинг 3.18. Реализация вычисления факториала на F# методом хвостовой рекурсии

```
let rec factorialTCO (n:int) (acc:int) =
    if n <= 1 then acc
    else factorialTCO (n-1) (acc * n) ←
let factorial n = factorialTCO n 1
```

Последняя операция функции —
это рекурсивный вызов самой себя,
без выполнения каких-либо последующих операций

В этой реализации рекурсивной функции параметр `acc` играет роль аккумулятора. Используя такой аккумулятор и гарантуя, что рекурсивный вызов является последней операцией функции, компилятор оптимизирует выполнение функции для многократного применения одного стекового кадра вместо того, чтобы хранить каждый промежуточный результат рекурсии в разных стековых кадрах (рис. 3.9).

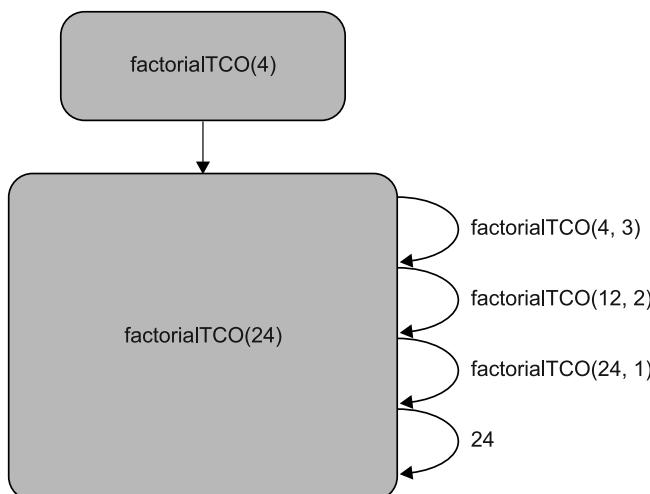


Рис. 3.9. Вычисление факториала методом хвостовой рекурсии позволяет многократно использовать один и тот же стековый кадр

На рисунке показано вычисление факториалов методом хвостовой рекурсии. F# поддерживает рекурсивные функции с хвостовым вызовом, но компилятор C#, к сожалению, не предназначен для оптимизации функций с хвостовой рекурсией.

3.4.2. Оптимизация рекурсивной функции в стиле передачи продолжений

Иногда рекурсивные функции с оптимизированным хвостовым вызовом не являются правильным решением или же их трудно реализовать. В этом случае одним из возможных альтернативных подходов будет CPS — технология передачи результата функции в ее продолжение. Метод CPS используется для оптимизации рекурсивных функций, так как позволяет избежать выделения памяти в стеке. Кроме того, CPS применяется в библиотеке Microsoft TPL: на C# в `async/await`, а на F# — в `async-workflow`.

CPS играет важную роль в параллельном программировании. В следующем примере показана функция `GetMaxCPS`, в которой используется шаблон CPS:

```
static void GetMaxCPS(int x, int y, Action<int> action)
    => action(x > y ? x : y);

GetMaxCPS (5, 7, n => Console.WriteLine(n));
```

Аргумент для передачи продолжения определен как делегат `Action<int>`, который удобно применять для передачи лямбда-выражения. Важно, что функция, построенная таким образом, никогда не возвращает результат напрямую, а передает его в процедуру продолжения. CPS также может использоваться для реализации рекурсивных функций с хвостовым вызовом.

Рекурсивные функции с CPS

Теперь, обладая базовыми знаниями о CPS, мы воспользуемся этим подходом в F# для рефакторинга примера с вычислением факториала, представленного в листинге 3.19. (Реализацию данного примера на C# вы найдете в коде для скачивания.)

Листинг 3.19. Рекурсивная реализация вычисления факториала на F# с использованием CPS

```
let rec factorialCPS x continuation =
    if x <= 1 then continuation()
    else factorialCPS (x - 1) (fun () -> x * continuation())

let result = factorialCPS 4 (fun () -> 1) ←———— Значение результата равно 24
```

Эта функция аналогична предыдущей реализации с аккумулятором; разница заключается в том, что здесь вместо переменной-аккумулятора передается функция. В данном случае функция `factorialCPS` применяет к своему результату функцию `continuation`.

Параллельный рекурсивный обход В-дерева

В листинге 3.20 показан пример рекурсивного обхода древовидной структуры с выполнением действия для каждого элемента. Функция `WebCrawler` из главы 2 строит иерархическое представление веб-ссылок с заданного сайта. Затем она просматривает

HTML-содержимое каждой веб-страницы и ищет ссылки на изображения, которые загружаются параллельно. Примеры кода из главы 2 (см. листинги 2.16–2.19) были призваны познакомить вас с основами параллельных вычислений в целом, а не с типичной процедурой распараллеливания на основе задач. Загрузка любых данных из Интернета — это операция ввода-вывода; в главе 8 будет показано, что их лучше всего выполнять асинхронно.

Листинг 3.20. Параллельная рекурсивная функция, построенная по принципу «разделяй и властвуй»

```

let maxDepth = int(Math.Log(float System.Environment.ProcessorCount,
  2.)+4.)                                ← Порог позволяет избежать создания слишком большого
                                             количества задач, превышающего количество ядер

let webSites : Tree<string> =           ←
  WebCrawlerExample.WebCrawler("http://www.foxnews.com")
  |> Seq.fold(fun tree site -> insert site tree ) Empty

let downloadImage (url:string) =          ← Конструктор fold строит древовидную
  use client = new System.Net.WebClient()   структуру, описывающую иерархию сайта
  let fileName = Path.GetFileName(url)
  client.DownloadFile(url, @"c:\Images\" + fileName) ← Загрузка изображения
                                                       в локальный файл

let rec parallelDownloadImages tree depth = ←
  match tree with
  | _ when depth = maxDepth ->
    tree |> inorder downloadImage |> ignore
  | Node(leaf, left, right) ->
    let taskLeft = Task.Run(fun() ->
      parallelDownloadImages left (depth + 1))
    let taskRight = Task.Run(fun() ->
      parallelDownloadImages right (depth + 1))
    let taskLeaf = Task.Run(fun() -> downloadImage leaf)
    Task.WaitAll([|taskLeft;taskRight;taskLeaf|]) ← Ожидание завершения задач
  | Empty -> ()                                ←

Рекурсивная функция, которая параллельно
обходит древовидную структуру, загружая
одновременно несколько изображений

```

Конструктор `Task.Run` используется для создания и порождения задач. Параллельная рекурсивная функция `parallelDownloadImages` принимает в качестве аргумента глубину рекурсии, которая применяется для ограничения количества создаваемых задач, чтобы оптимизировать потребление ресурсов.

При каждом рекурсивном вызове значение глубины рекурсии увеличивается на единицу; когда оно превышает порог `maxDepth`, остальная часть дерева обрабатывается последовательно. Если для каждого узла дерева создается отдельная задача, то издержки на создание новых задач превысят выгоду от параллельных вычислений. Например, для компьютера с восемью процессорами порождение 50 задач сильно снизит производительность вследствие конкуренции, возникающей в результате обработки этих задач одними и теми же процессорами. Планировщик из библиотеки TPL рассчитан на параллельную обработку большого количества задач, но его поведение подходит не для каждого случая параллелизма динамических задач

(<http://mng.bz/ww1i>); в некоторых ситуациях, как в представленном выше примере параллельной рекурсивной функции, предпочтительнее использовать ручную настройку.

В конце для ожидания завершения задач применяется конструкция `Task.WaitAll`. На рис. 3.10 показано иерархическое представление выполняемых параллельно задач.

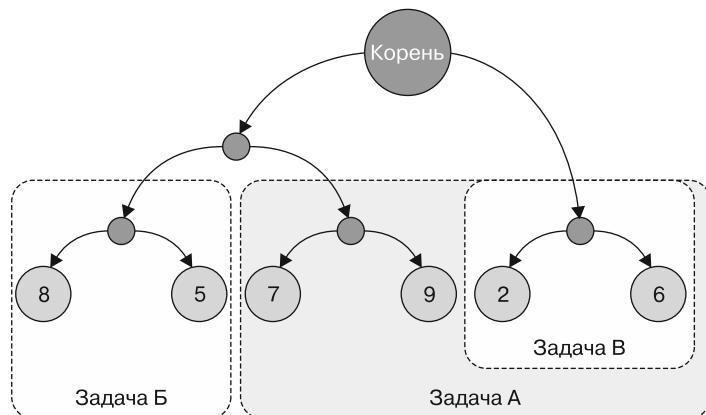


Рис. 3.10. Из корневого узла создается задача В для обработки правой части поддерева.

Этот процесс повторяется для всего поддерева путем выполнения задачи А. Когда задача А завершается, обрабатывается левая сторона поддерева посредством выполнения задачи Б. Данная операция повторяется для всех поддеревьев, на каждой итерации создается новая задача

В табл. 3.4 показано время выполнения параллельной рекурсивной операции `parallelDownloadImages`, измеренное относительно последовательной версии. Контрольный показатель — усредненное значение загрузки 50 изображений (операция повторялась три раза).

Таблица 3.4. Временной тест загрузки 50 изображений с использованием параллельной рекурсии

Последовательно	Параллельно
19,71	4,12

Параллельный калькулятор

Еще один интересный пример использования древовидной структуры — построение параллельного калькулятора. Теперь, после всего, что мы узнали, мы понимаем, что реализация такой программы не является тривиальной. Для определения типа выполняемых операций можно применять ADT в виде размеченных объединений F#:

```
type Operation = Add | Sub | Mul | Div | Pow
```

Теперь можно представить калькулятор как древовидную структуру, где каждой операции соответствует узел, содержащий данные для выполнения конкретного вычисления:

```
type Calculator =
| Value of double
| Expr of Operation * Calculator * Calculator
```

Как вы могли заметить, этот код подобен древовидной структуре, использованной ранее:

```
type Tree<'a> =
| Empty
| Node of leaf:'a * left:Tree<'a> * right:Tree<'a>
```

Единственное различие заключается в том, что в калькуляторе вместо варианта `Empty` в древовидной структуре стоит вариант `value`, так как для выполнения любой математической операции требуется значение. Лист дерева становится типом `Operation`, а левая и правая ветви рекурсивно ссылаются на сам тип `Calculator` точно так же, как и сам `Calculator`.

Теперь можно реализовать рекурсивную функцию, которая проходит по дереву `Calculator` и выполняет операции параллельно. В листинге 3.21 показаны реализация и использование функции `eval`.

Листинг 3.21. Параллельный калькулятор

```
let spawn (op:unit->double) = Task.Run(op) ← Вспомогательная функция для порождения
                                         задачи и запуска операции

let rec eval expr = ← Шаблоны соответствия для DU калькулятора,
                     обеспечивающие прохождение по ветвям дерева
    match expr with
    | Value(value) -> value ← Если вариант expr имеет тип Value,
                               то извлекается и возвращается значение
    | Expr(op, lExpr, rExpr) -> ← Порождение задачи для выполнения
                                    следующего вычисления, которое может
                                    быть выполнением другой операции
        let op1 = spawn(fun () -> eval lExpr)
        let op2 = spawn(fun () -> eval rExpr)

        let apply = Task.WhenAll([op1;op2])
        let lRes, rRes = apply.Result.[0], apply.Result.[1] ← Ожидание выполнения
                                                               операции

        match op with
        | Add -> lRes + rRes ← Получение значений, которые могут быть
        | Sub -> lRes - rRes результиром выполнения других операций,
        | Mul -> lRes * rRes и выполнение текущей операции
        | Div -> lRes / rRes
        | Pow -> System.Math.Pow(lRes, rRes)

Если вариант expr имеет тип Expr, то извлекается операция
и выполняется рекурсивная обработка ветвей с извлечением значений
```

Функция `eval` рекурсивно выполняет набор параллельных операций, представленных в виде древовидной структуры. На каждой итерации переданное выражение проверяется на соответствие шаблону и извлекается значение, если данный вариант является типом `Value`, либо выполняется операция, если это тип `Expr`. Интересно, что рекурсивное повторное выполнение каждой ветви для узла типа `Expr` производится параллельно. Каждая ветвь `Expr` возвращает тип `value`, который вычисляется на каждой дочерней операции (в подузлах). Затем эти значения используются для выполнения последней операции, являющейся корнем дерева операций, в которой они передаются как аргументы для получения конечного результата. Ниже представлен простой набор операций, выполняемых деревом калькулятора для вычисления выражения $2^{10} / 2^9 + 2 * 2$:

```
let operations =
    Expr(Add,
        Expr(Div,
            Expr(Pow, Value(2.0), Value(10.0)),
            Expr(Pow, Value(2.0), Value(9.0))),
        Expr(Mul, Value(2.0), Value(2.0)))  
  
let value = eval operations
```

В настоящем разделе код для определения древовидной структуры данных и выполнения рекурсивной функции на основе задач написан на F#, но его можно реализовать и на C#. Мы не стали приводить весь этот код здесь, но он доступен для скачивания с сайта книги.

Резюме

- ❑ В неизменяемых структурах данных применяются интеллектуальные подходы, такие как совместное использование структур, чтобы свести к минимуму копирование общих элементов и снизить нагрузку на сборщик мусора.
- ❑ Очень важно посвятить некоторое время профилированию производительности приложений, чтобы избежать узких мест и неприятных сюрпризов, когда программа будет работать в среде эксплуатации под большими нагрузками.
- ❑ Для обеспечения потокобезопасности при создании объектов и для повышения производительности в функциональных структурах данных можно использовать «ленивые» вычисления, при которых вычисления откладываются до последнего момента.
- ❑ Функциональная рекурсия — естественный способ итерирования при функциональном программировании, поскольку она позволяет избежать изменения состояния. Кроме того, построение рекурсивных функций делает программу более модульной.
- ❑ Хвостовая рекурсия — это технология преобразования обычной рекурсивной функции в оптимизированную версию, которая позволяет обрабатывать большие объемы входных данных без рисков и побочных эффектов.
- ❑ Стиль передачи вычислений (CPS) — это технология передачи результата выполнения функции в ее продолжение. Данный метод используется для оптимизации рекурсивных функций, поскольку позволяет избежать выделения памяти в стеке. Кроме того, CPS применяется в библиотеке Task Parallel Library в .NET 4.0, в `async/await` на C# и в `async-workflow` на F#.
- ❑ Рекурсивные функции — отличные кандидаты для реализации технологии «разделяй и властвуй», что обеспечивает динамический параллелизм на уровне задач.

Часть II

*Конкурентная
программа: разные
части, разные
подходы*

Данная часть книги посвящена углубленному изучению концепций и областей применения функционального программирования. Мы рассмотрим различные модели конкурентного программирования, уделяя особое внимание выгодам и преимуществам этой парадигмы. Мы изучим такие темы, как Task Parallel Library и шаблоны параллельного программирования, включая Fork/Join, «разделяй и властвуй» и MapReduce. Кроме того, обсудим декларативную компоновку, абстракцию высокого уровня в асинхронных операциях, агентное программирование и семантику передачи сообщений. Вы сразу увидите, как функциональное программирование позволяет компоновать элементы программы, не выполняя их. Эти технологии помогают распараллеливать работу, писать более понятные и эффективные программы с оптимальным потреблением памяти.

Основы обработки больших данных: распараллеливание данных, часть 1

В этой главе:

- важность распараллеливания в мире больших данных;
- применение шаблона Fork/Join;
- написание декларативных параллельных программ;
- ограничения параллельного цикла `for`;
- повышение производительности при распараллеливании данных.

Представьте себе, что вы планируете ужин на четверых и собираетесь приготовить спагетти. Будем считать, что на то, чтобы сварить и подать макароны, требуется 10 мин. Вы приступаете к делу, наполняете водой кастрюлю среднего размера и ставите ее кипятиться. Но тут к вам на ужин приходят еще два приятеля. Очевидно, потребуется больше пасты. Вы можете взять кастрюлю побольше, чтобы закипятить больше воды и сварить больше спагетти, но тогда готовка займет больше времени. Или же можно взять вторую кастрюлю и поставить ее на огонь рядом с первой, чтобы обе партии макарон были готовы одновременно. Примерно так и работает распараллеливание данных. Для того чтобы легко обрабатывать крупные объемы данных, их можно «варить» параллельно.

За последнее десятилетие объем генерируемых данных вырос экспоненциально. По оценкам 2017 г., каждую минуту появляется 4 750 000 «лайков» в Facebook, почти 400 000 твитов, более 2,5 млн сообщений в Instagram и более 4 млн поисковых

запросов в Google. Эти цифры продолжают расти со скоростью 15 % в год. Такое ускорение влияет на коммерческие предприятия, которым теперь приходится быстро анализировать всевозможные большие данные (https://ru.wikipedia.org/wiki/Большие_данные). Как изучить такие огромные объемы данных, сохраняя высокую скорость реакции? Ответ на этот вопрос дает новое поколение технологий, разработанных с учетом распараллеливания данных и, в частности, с упором на способность сохранять высокую производительность при постоянно увеличивающихся объемах информации.

В настоящей главе вы познакомитесь с концепциями, шаблонами проектирования и технологиями быстрой обработки огромных количеств данных. Вы проанализируете проблемы, возникающие при построении параллельных циклов, и узнаете об их решениях. Вы также узнаете, что с помощью функционального программирования в сочетании с распараллеливанием данных можно добиться впечатляющего повышения производительности алгоритмов при минимальных изменениях кода.

4.1. Что такое распараллеливание данных

Распараллеливание данных — это модель программирования, при которой один и тот же набор операций выполняется параллельно для большого количества данных. Такая модель программирования становится все более популярной, поскольку позволяет быстро обрабатывать массивные объемы информации, а проблем с большими данными становится все больше. Распараллеливание позволяет выполнять алгоритм без реорганизации его структуры, тем самым постепенно повышая масштабируемость.

Существует две модели распараллеливания данных: «один поток команд, один поток данных» и «один поток команд, много потоков данных».

- ❑ Один поток команд, один поток данных (Single Instruction Single Data, SISD) используется для определения одноядерной архитектуры. Одноядерная процессорная система выполняет одну задачу за каждый такт процессора; выполнение программы является последовательным и детерминированным. Процессор получает одну инструкцию (по очереди), выполняет работу, необходимую для одного фрагмента данных, и возвращает результаты вычисления. В этой книге такая архитектура процессора не рассматривается.
- ❑ Один поток команд, много потоков данных (Single Instruction Multiple Data, SIMD) — форма параллелизма, достигаемая путем распределения данных между несколькими доступными ядрами. На каждом цикле процессора ко всем данным применяются одни и те же операции. Такой тип параллельной многоядерной процессорной архитектуры обычно используется при распараллеливании данных.

Данные для распараллеливания разбиваются на куски, после чего каждый такой кусок подвергается интенсивным вычислениям и обрабатывается независимо от других. В результате либо производятся новые данные, которые затем агрегируются, либо объем данных снижается до скалярного значения. Если вы пока не знакомы с этими терминами, то они станут вам понятны к концу главы.

Возможность независимой обработки фрагментов данных является ключевым условием значительного увеличения производительности, поскольку устранение зависимостей между блоками данных исключает необходимость синхронизации для доступа к ним и решает любые проблемы, связанные с состоянием гонки (рис. 4.1).

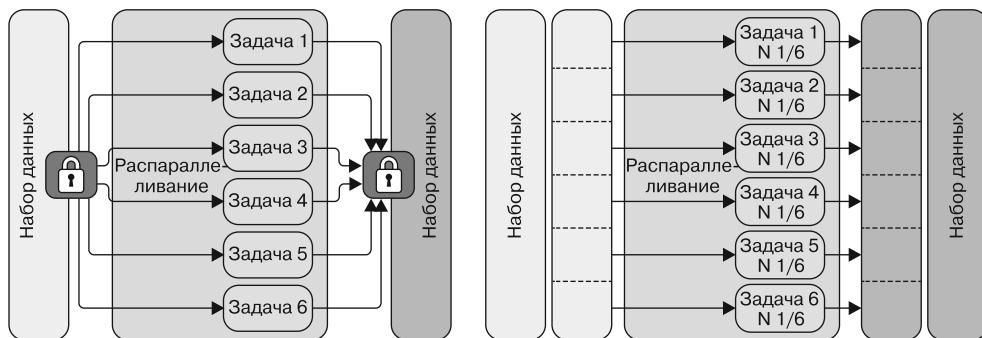


Рис. 4.1. Распараллеливание данных достигается путем разделения набора данных на фрагменты и параллельной обработки каждого из них независимо от других, так что для каждого фрагмента создается отдельная задача. Когда все задачи завершены, набор данных снова собирается.

Показанный слева набор данных обрабатывается несколькими задачами с использованием блокировок для синхронизации доступа к данным в целом. В этом случае синхронизация является источником конфликтов между потоками и причиной дополнительных издержек. Справа показан набор данных, разделенный на шесть частей; каждая задача выполняется для одной шестой части от общего набора данных размером N . Такая конструкция устраняет необходимость блокировок для синхронизации

Распараллеливание данных может быть достигнуто в распределенной системе, если разделить работу между несколькими узлами, или на одном компьютере путем разделения работы на отдельные потоки. В этой главе основное внимание уделяется внедрению и использованию многоядерного оборудования для распараллеливания данных.

4.1.1. Распараллеливание данных и задач

Целью распараллеливания данных является разделение исходного их набора данных и создание достаточного количества задач для максимального использования процессорных ресурсов. Кроме того, каждая задача должна содержать достаточно много операций, чтобы гарантировать максимально короткое время выполнения. Необходимо свести к минимуму количество переключений контекста, которое приводит к дополнительным издержкам.

Существует два варианта распараллеливания данных.

- ❑ *Распараллеливание задач* нацелено на выполнение компьютерных программ на нескольких процессорах одновременно, так что каждый поток отвечает за выполнение своей операции. Это одновременное выполнение множества различных функций для одного и того же или разных наборов данных на нескольких ядрах.

- *Распараллеливание данных* нацелено на разделение исходного набора данных на более мелкие части, соответствующие нескольким задачам, причем все задачи выполняют параллельно одну и ту же инструкцию. Например, распараллеливание данных может использоваться в алгоритме обработки изображений, где каждое изображение или пиксель обновляется параллельно независимыми задачами. И наоборот, при распараллеливании задач набор изображений вычисляется параллельно, но к каждому изображению применяется своя операция (рис. 4.2).



Рис. 4.2. Распараллеливание данных — это одновременное выполнение одной и той же функции для всех элементов набора данных. Распараллеливание задач — это одновременное выполнение нескольких разных функций для одних и тех же или разных наборов данных

Коротко говоря, распараллеливание задач означает выполнение нескольких функций (задач) и направлено на сокращение общего времени вычислений путем одновременного запуска этих задач. Распараллеливание данных сокращает время, затрачиваемое на обработку их набора, путем разделения одного и того же вычислительного алгоритма между несколькими процессорами, так что одни и те же операции выполняются параллельно.

4.1.2. Концепция «естественной параллельности»

При распараллеливании данных алгоритмы, применяемые для их обработки, иногда называются «естественному параллельными», так как обладают особым свойством естественной масштабируемости¹. Это свойство влияет на количество распараллеливаний в алгоритме по мере увеличения количества доступных аппаратных потоков. Чем мощнее компьютер, тем быстрее будет работать алгоритм. При распараллеливании данных следует разрабатывать алгоритмы так, чтобы каждая операция, представленная в виде отдельной задачи и связанная с отдельным аппаратным ядром, осуществлялась независимо от других. Преимущество такой

¹ Подробнее о естественном параллелизме, также известном как «приятный параллелизм», читайте здесь: https://en.wikipedia.org/wiki/Embarrassingly_parallel.

структуры заключается в том, что она автоматически регулирует рабочую нагрузку в процессе выполнения и корректирует разделение данных в соответствии с возможностями компьютера. Такое поведение гарантирует реализацию программы на всех доступных ядрах.

Рассмотрим в качестве примера суммирование большого массива чисел. Любая часть массива может быть просуммирована независимо от остальных. Затем частичные суммы можно сложить и получить тот же результат, что и при последовательном суммировании массива. Не имеет значения, вычисляются частичные суммы на одном процессоре или на разных в одно и то же время. Подобные алгоритмы с высокой степенью независимости называются естественными параллельными проблемами: чем больше процессоров им предоставить, тем быстрее они будут работать. В главе 3 был представлен шаблон «разделяй и властвуй», который обеспечивает естественное распараллеливание. Этот шаблон распределяет работу между несколькими задачами, а затем снова объединяет (сокращает) результат. Естественные параллельные структуры не требуют сложного механизма координации, обеспечивающего естественную автоматическую масштабируемость. Примерами шаблонов проектирования, в которых используется такой подход, являются Fork/Join, параллельное агрегирование (сокращение) и MapReduce. Мы обсудим их далее в этой главе.

4.1.3. Поддержка распараллеливания данных в .NET

Найти в программе код, который может быть распараллелен, — непростая задача. Но существуют общие правила и рекомендации, способные в этом помочь. Прежде всего нужно выполнить профилирование приложения. Такой анализ программы покажет, на что тратится основное время. Именно это место программы затем необходимо изучить более глубоко, чтобы повысить производительность и обнаружить возможности распараллеливания. Руководствуйтесь следующим правилом: возможность распараллеливания есть там, где две и более части исходного кода могут выполняться независимо и параллельно без изменения выходных данных программы. Если же введение параллелизма изменит выход программы, то она не будет детерминированной и может стать ненадежной; в таких случаях распараллеливание неприменимо.

Для обеспечения детерминированных результатов в параллельной программе блоки исходного кода, которые выполняются одновременно, не должны иметь взаимных зависимостей. Фактически программа может быть легко распараллелена, если в ней нет зависимостей или если существующие зависимости могут быть устраниены. Например, в шаблоне «разделяй и властвуй» нет зависимостей между рекурсивными выполнениями функций, так что их можно распараллелить.

Первым кандидатом на распараллеливание является большой набор данных, для которых вычисления, интенсивно использующие процессор, могут выполняться независимо для каждого элемента. Как правило, отличными кандидатами на применение параллелизма бывают все виды циклов (`for`, `while` и `for-each`). Задействуя библиотеку Microsoft TPL, можно легко преобразовать последовательный цикл в параллельный. Эта библиотека предоставляет уровень абстракции, который

упрощает реализацию основных шаблонов, применяемых для распараллеливания данных. Такие шаблоны могут быть реализованы с использованием параллельных конструкций `Parallel.For` и `Parallel.ForEach`, предлагаемых классом `Parallel` из библиотеки `TPL`.

Вот несколько шаблонов, предоставляющих возможности для распараллеливания программ.

- ❑ Последовательные циклы, в которых нет зависимостей между итерациями.
- ❑ Операции сокращения и/или агрегирования, где результаты вычисления между этапами частично объединяются. Данная модель может быть представлена с помощью шаблона `MapReduce`.
- ❑ Блок вычислений, в котором явные зависимости могут быть преобразованы в виде шаблона `Fork/Join`, после чего каждый шаг можно выполнять параллельно.
- ❑ Рекурсивные алгоритмы с использованием подхода «разделяй и властвуй», где каждая итерация может выполняться независимо от других, в отдельном потоке.

В .NET Framework распараллеливание данных также поддерживается посредством языка запросов `PLINQ`, который я рекомендую применять. По сравнению с классом `Parallel` этот язык запросов предлагает более декларативную модель распараллеливания данных и используется для параллельного выполнения произвольных запросов к их источнику. Декларативность в таком случае означает описание того, что именно нужно сделать с данными, а не как это должно быть сделано. Внутри библиотеки `TPL` применяются сложные алгоритмы планирования для эффективного распределения параллельных вычислений между доступными процессорными ядрами. Аналогичные технологии используются в `C#` и `F#`. В следующем разделе мы рассмотрим эти технологии для обоих языков программирования и увидим, что их можно комбинировать так, чтобы они прекрасно дополняли друг друга.

4.2. Шаблон `Fork/Join`: параллельный алгоритм Мандельброта

Самый лучший способ понять, как преобразовать последовательную программу в параллельную, — рассмотреть пример. В этом разделе мы воспользуемся шаблоном `Fork/Join` и преобразуем программу так, чтобы посредством параллельных вычислений получить более высокую производительность.

В шаблоне `Fork/Join` один поток выполняет ветвление и координирует несколько независимых параллельных потоков, а после их завершения объединяет результаты. Параллелизм `Fork/Join` означает выполнение двух основных этапов.

1. Разделить исходную задачу на множество подзадач, которые планируется выполнять независимо друг от друга.
2. Дождаться завершения всех параллельных операций, созданных в результате ветвления, а затем, объединив итоги этих подзадач, получить результат исходной задачи.

В отношении распараллеливания данных рис. 4.3 очень похож на рис. 4.1. Разница заключается только в последнем шаге, где шаблон Fork/Join снова объединяет результаты в один общий результат.

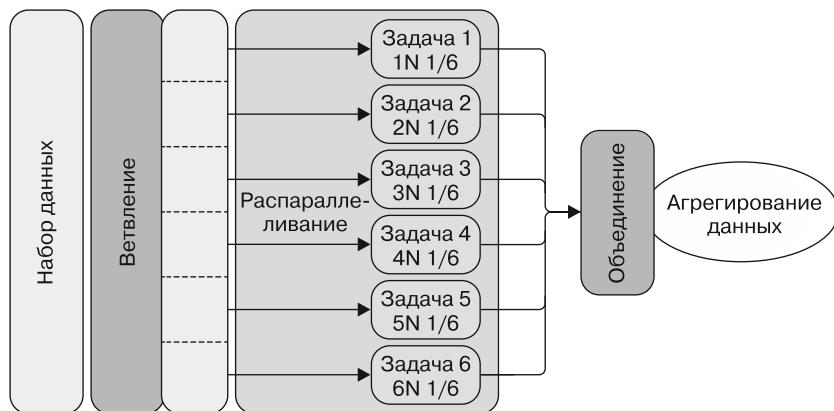


Рис. 4.3. Шаблон Fork/Join разбивает задачу на подзадачи, которые могут выполняться независимо и параллельно. После завершения операций результаты подзадач снова объединяются. Не случайно эта схема часто используется для распараллеливания данных — сходство очевидно

Как видим, этот шаблон хорошо подходит для распараллеливания данных. Шаблон Fork/Join ускоряет выполнение программы, разбивая работу на куски (*fork*) и выполняя каждый блок независимо и параллельно. После завершения всех параллельных операций куски снова объединяются (*join*). В целом Fork/Join — отличный шаблон для реализации структурированного параллелизма, поскольку ветвление и объединение происходят мгновенно (синхронно по отношению к вызывающему объекту), но параллельно (с точки зрения производительности и скорости). Абстракция Fork/Join может быть легко описана с помощью цикла `Parallel.For` из класса `Parallel` библиотеки .NET. Этот статический метод прозрачно выполняет разделение данных и запуск задач.

Исследуем конструкцию цикла `Parallel.For` на примере. Сначала построим последовательный цикл `for`, чтобы нарисовать изображение Мандельброта (рис. 4.4), а затем выполним рефакторинг кода для более быстрой работы. В конце мы оценим достоинства и недостатки этого подхода.

Изображения Мандельброта

Для того чтобы построить набор изображений Мандельброта, нужно взять случайный набор комплексных чисел, выполнить для каждого из них определенную математическую операцию и определить, стремится ли ее результат к бесконечности. Комплексное число представляет собой комбинацию действительного и мнимого чисел. Все числа, о которых вы привыкли думать, являются действительными. Мнимое число — это число, квадрат которого отрицательный. Если рассматривать действительную и мнимую

части каждого числа как координаты изображения, пиксели которого окрашиваются в соответствии с тем, насколько быстро расходится последовательность, если она вообще расходится, то получим изображения Мандельброта. Множество изображений Мандельброта представляет собой сложный контур, детализация которого возрастает по мере увеличения масштаба. Это один из самых известных примеров математической визуализации.

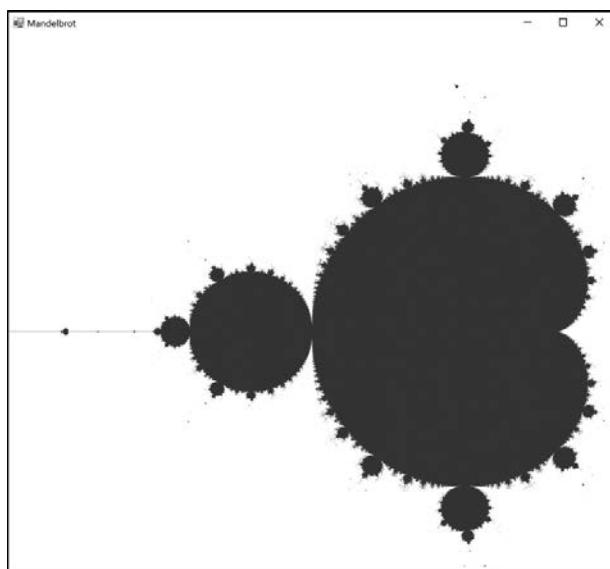


Рис. 4.4. Контур Мандельброта, полученный в результате выполнения кода, представленного в этом разделе

В данном примере детали реализации алгоритма неважны. Важно то, что для каждого пикселя на рисунке (в изображении) выполняются вычисления для каждого присвоенного ему цветового компонента. Эти вычисления являются независимыми, поскольку цвет пикселя не зависит от цветов других пикселов и присвоение цвета может выполняться параллельно. Фактически каждый пиксель может иметь свой цвет, присвоенный ему независимо от цвета других пикселов изображения. Отсутствие зависимостей влияет на стратегию выполнения: все вычисления могут выполняться параллельно.

В данном контексте алгоритм Мандельброта применяется для рисования изображения, соответствующего величине комплексного числа. В естественном представлении этой программы используется цикл `for` для перебора всех значений декартовой плоскости, чтобы присвоить каждой точке соответствующий цвет. Цвет вычисляется по алгоритму Мандельброта. Прежде чем углубиться в основную реализацию, нам

потребуется объект для описания комплексного числа. В листинге 4.1 представлена простая реализация комплексного числа, используемого для выполнения операций над комплексными числами.

Листинг 4.1. Объект, описывающий комплексное число

```
class Complex
{
    public Complex(float real, float imaginary)
    {
        Real = real;
        Imaginary = imaginary;
    }

    public float Imaginary { get; }
    public float Real { get; }

    public float Magnitude
        => (float)Math.Sqrt(Real * Real + Imaginary * Imaginary);
```

Использование автополучателя
обеспечивает неизменяемость

Свойство Magnitude определяет
относительный размер
комплексного числа

Перегруженные операторы выполняют
сложение и умножение комплексных чисел

Класс `Complex` содержит определение свойства `Magnitude`. Интересной частью этого кода являются два перегруженных оператора для объекта `Complex`. Данные операторы используются для сложения и умножения комплексных чисел в алгоритме Мандельброта. В листинге 4.2 показаны две основные функции алгоритма Мандельброта; функция `isMandelbrot` определяет, принадлежит ли комплексное число множеству Мандельброта.

Листинг 4.2. Последовательный алгоритм Мандельброта

```
Func<Complex, int, bool> isMandelbrot = (complex, iterations) =>
{
    var z = new Complex(0.0f, 0.0f);
    int acc = 0;
    while (acc < iterations && z.Magnitude < 2.0)
    {
        z = z * z + complex;
        acc += 1;
    }
    return acc == iterations;
};

for (int col = 0; col < Cols; col++) {
    for (int row = 0; row < Rows; row++) {
```

Функции для определения того,
является ли данное комплексное число
частью множества Мандельброта

Внешний и внутренний цикл
для прохождения
столбцов и строк изображения

```

var x = ComputeRow(row);
var y = ComputeColumn(col);
var c = new Complex(x, y);
var color = isMandelbrot(c, 100) ? Color.Black : Color.White;
var offset = (col * bitmapData.Stride) + (3 * row);
pixels[offset + 0] = color.B; // Синяя составляющая
pixels[offset + 1] = color.G; // Зеленая составляющая
pixels[offset + 2] = color.R; // Красная составляющая
}
}

```

Операции преобразования координат текущего пикселя в параметры комплексного числа

Код для назначения цветовых атрибутов пиксели изображения

Функция для определения цвета пикселя

В коде отсутствует детальная информация о генерации растрового изображения, так как это не имеет отношения к цели данного примера. Полное решение вы найдете в исходном коде к книге.

В данном примере есть два цикла: внешний проходит по столбцам области изображения, а внутренний — по строкам. На каждой итерации выполняются соответственно функции `ComputeColumn` и `ComputeRow`, чтобы преобразовать координаты текущего пикселя в действительную и мнимую части комплексного числа. Затем функция `isMandelbrot` определяет, принадлежит ли данное комплексное число множеству Мандельброта. Эта функция принимает в качестве аргументов комплексное число и число итераций, а возвращает логическое значение, показывающее, принадлежит ли данное комплексное число множеству Мандельброта. Тело функции содержит цикл, который накапливает значение и уменьшает счетчик. Возвращаемое логическое значение истинно, если аккумулятор `acc` равен числу итераций.

В данной реализации программы выполнение функции `isMandelbrot` 1 млн раз занимает 3,666 с — именно столько пикселов составляет изображение Мандельброта. Более быстрым решением будет параллельное выполнение цикла в алгоритме Мандельброта. Как уже отмечалось, в библиотеке TPL есть конструкции, которые можно применять для распараллеливания программы вслепую, что приводит к фантастическому повышению производительности. В данном примере вместо последовательного цикла использована функция более высокого порядка `Parallel.For`. В листинге 4.3 показано параллельное преобразование алгоритма с минимальными изменениями. В частности, сохраняется последовательная структура кода.

Обратите внимание, что распараллелен только внешний цикл, чтобы предотвратить перенасыщение ядер рабочими элементами. Такое простое изменение позволило сократить время выполнения программы на четырехъядерном процессоре до 0,699 с.

Перенасыщение — это вид дополнительных издержек, возникающих при параллельном программировании, когда количество потоков, созданных и управляемых планировщиком для выполнения вычислений, значительно превышает количество доступных аппаратных ядер. В таком случае после распараллеливания приложение может работать медленнее, чем последовательная реализация.

В качестве общего правила я рекомендую распараллеливать дорогостоящие операции на самом высоком уровне. Например, на рис. 4.5 показаны вложенные циклы `for`; я предлагаю применить распараллеливание только для внешнего цикла.

Листинг 4.3. Параллельный алгоритм Мандельброта

```

Func<Complex, int, bool> isMandelbrot = (complex, iterations) =>
{
    var z = new Complex(0.0f, 0.0f);
    int acc = 0;
    while (acc < iterations && z.Magnitude < 2.0)
    {
        z = z * z + complex;
        acc += 1;
    }
    return acc == iterations;
};

System.Threading.Tasks.Parallel.For(0, Cols - 1, col => {
    for (int row = 0; row < Rows; row++) {
        var x = ComputeRow(row);
        var y = ComputeColumn(col);
        var c = new Complex(x, y);
        var color = isMandelbrot(c, 100) ? Color.DarkBlue : Color.White;
        var offset = (col * bitmapData.Stride) + (3 * row);
        pixels[offset + 0] = color.B; // Синяя составляющая
        pixels[offset + 1] = color.G; // Зеленая составляющая
        pixels[offset + 2] = color.R; // Красная составляющая
    }
});
}

```

Параллельная конструкция цикла for применяется
только для внешнего цикла во избежание
перенасыщения процессорных ресурсов

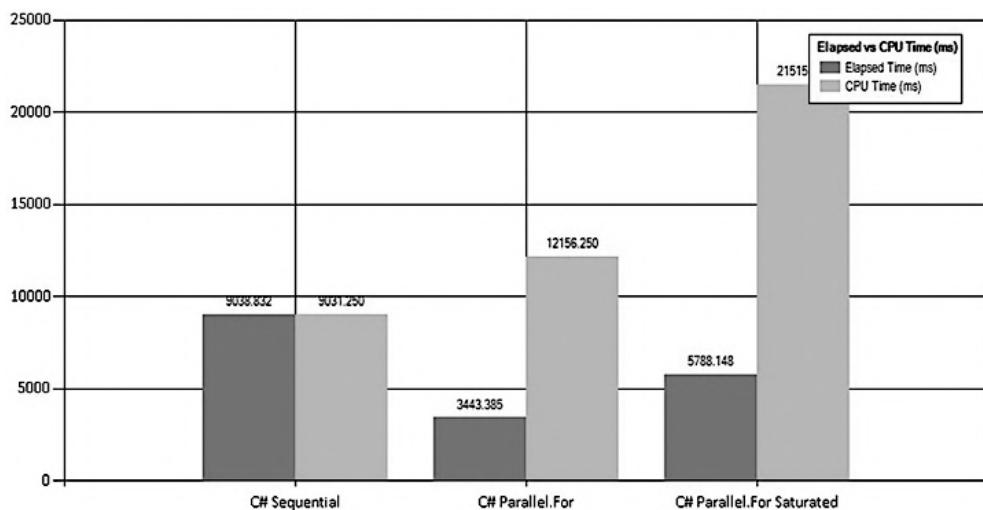


Рис. 4.5. Время выполнения программы с конструкцией Parallel.For по сравнению с последовательным циклом, который выполняется за 9,038 с. Параллельное вычисление занимает 3,443 с. Цикл Parallel.For примерно в три раза быстрее, чем последовательный код. Последний столбец справа показывает время выполнения перенасыщенного параллельного цикла, где и внешний, и внутренний циклы используют конструкцию Parallel.For. Перенасыщенный параллельный цикл занимает 5,788 с, что на 50 % медленнее, чем ненасыщенная версия

Истекшее время и процессорное время

На диаграмме, показанной на рис. 4.5, процессорное время — это время, в течение которого CPU выполнял задание. Истекшее время — общее время выполнения операции, независимо от задержки ресурсов или параллельного выполнения. Обычно истекшее время больше, чем процессорное, но на многоядерной машине данная ситуация меняется.

При выполнении конкурентной программы на многоядерной машине обеспечивается истинный параллелизм. В таком случае процессорное время равно сумме времени выполнения для всех потоков, работающих на других процессорах в это же время. Например, при запуске однопоточной (последовательной) программы на четырехъядерном компьютере истекшее время почти равно процессорному времени, поскольку работает только одно ядро. При параллельном выполнении той же программы с использованием всех четырех ядер истекшее время становится меньше, потому что программа работает быстрее, однако процессорное время увеличивается, потому что оно вычисляется как сумма времени выполнения всех четырех параллельных потоков. Когда программа задействует более одного процессора для выполнения задачи, процессорное время может превышать истекшее.

Таким образом, истекшее время показывает, сколько времени выполняется программа со всеми параллельными вычислениями, а процессорное время демонстрирует, сколько времени выполняются все потоки, игнорируя тот факт, что при параллельном выполнении все эти потоки реализуются одновременно.

Как правило, оптимальное количество рабочих потоков для выполнения параллельной задачи должно быть равно количеству доступных аппаратных ядер, разделенному на средний процент использования ядра одной задачей. Например, для четырехъядерного компьютера с 50%-ным средним использованием ядра для каждой задачи идеальное число рабочих потоков для достижения максимальной пропускной способности равно восьми: $4 \text{ ядра} \times (100\% \text{-ное использование центрального процессора} / 50 \% \text{ среднего использования ядра на одну задачу})$. Любое количество рабочих потоков, превышающее это значение, может привести к дополнительным издержкам вследствие излишнего переключения контекста, что снижает производительность и эффективность применения процессора.

4.2.1. Когда узким местом является сборка мусора: структуры и объекты класса

На примере с изображениями Мандельброта мы показали, как можно преобразовать последовательный алгоритм в более быстрый. Безусловно, мы добились повышения скорости: от 9,038 до 3,443 с на четырехъядерной машине — это более чем втрое быстрее. Возможна ли дальнейшая оптимизация производительности? Планировщик TPL разбивает изображение на части и автоматически распределяет работу между задачами. Неужели же мы можем еще повысить скорость? В данном случае оптимизация подразумевает сокращение потребления памяти, в частности, за счет уменьшения выделения памяти для оптимизации сборки мусора (Garbage Collection, GC).

Когда запускается сборка мусора, выполнение программы приостанавливается до тех пор, пока операция по сборке мусора не завершится.

В рассмотренном примере с алгоритмом Мандельброта на каждой итерации создается новый объект `Complex`, после чего алгоритм решает, принадлежат ли координаты данного пикселя множеству Мандельброта. Объект `Complex` является ссылочным типом — это означает, что новые экземпляры указанного объекта выделяются в куче. Накопление объектов в куче приводит к перегрузке памяти и вынуждает запускать сборку мусора, чтобы освободить пространство.

Ссылочный объект, в отличие от типа-значения, занимает дополнительное место в памяти под указатель, необходимый для доступа к ячейке памяти объекта, выделенного в куче. Экземпляры класса всегда размещаются в куче и доступны через разыменование указателя. Таким образом, поскольку ссылочный объект является копией указателя, его передача обходится дешево с точки зрения памяти: 4 или 8 байт, в зависимости от аппаратной архитектуры. Кроме того, важно учитывать, что объект также имеет фиксированные издержки размером 8 байт для 32-битных процессов и 16 байт для 64-битных процессов. Для сравнения, тип-значение размещается не в куче, а в стеке, что исключает издержки на выделение памяти и сборку мусора.

Помните, что если тип-значение (структура) объявлен как локальная переменная в методе, то он размещается в стеке. Если же тип-значение объявлен как часть ссылочного типа (класса), то структура размещается в памяти, выделяемой под этот объект, — соответственно в куче.

Алгоритм Мандельброта в цикле `for` создает и уничтожает 1 млн объектов `Complex`; при таком высоком уровне выделения памяти создается значительная нагрузка на механизм сборки мусора. Если заменить ссылочный объект `Complex` на тип-значение, то скорость выполнения алгоритма должна увеличиться, поскольку размещение структуры в стеке никогда не потребует запуска сборки мусора и, соответственно, остановки программы на время выполнения этой процедуры. Фактически при передаче в метод типа `value` происходит побитовое копирование данного значения, поэтому создание структуры никогда не приведет к запуску сборки мусора, так как указанная структура не находится в куче.

ПРИМЕЧАНИЕ

Замена ссылочного типа на тип-значение часто приводит к гигантскому скачку производительности. В качестве примера сравним массив объектов с массивом структур на 32-разрядной машине. Предположим, что имеется массив из 1 млн элементов, каждый из которых представлен в виде объекта, содержащего 24 байта данных. Если это ссылочные типы, то общий размер массива составляет 72 Мбайт ($8 \text{ байт памяти на массив} + (4 \text{ байта на указатель} \times 1\ 000\ 000) + (8 \text{ байт служебных данных объекта} + 24 \text{ байта данных}) \times 1\ 000\ 000 = 72 \text{ Мбайт}$). Для того же массива, элементы которого представлены в виде структур, размер составляет всего 24 Мбайт : ($8 \text{ байт памяти на массив} + (24 \text{ байта данных}) \times 1\ 000\ 000 = 24 \text{ Мбайт}$). Интересно, что на 64-разрядной машине размер массива, состоящего из типов-значений, не изменяется, но размер массива из ссылочных типов увеличивается более чем до 40 Мбайт за счет издержек на хранение дополнительного байта указателя.

Оптимизация преобразования объекта `Complex` из ссылочного объекта в тип-значение проста. Нужно лишь заменить ключевое слово `class` на `struct`, как показано ниже. (Полная реализация объекта `Complex` здесь намеренно опущена.) Ключевое слово `struct` преобразует ссылочный тип (`class`) в тип-значение:

```
class Complex {           struct Complex {  
    public Complex(float real,           public Complex(float real,  
                  float imaginary)           float imaginary)  
    {                           {  
        this.Real = real;           this.Real = real;  
        this.Imaginary =           this.Imaginary =  
        imaginary;                 imaginary;  
    }                           }  
}
```

В результате этого простого изменения кода время создания изображения по алгоритму Мандельброта сократилось примерно на 20 % (рис. 4.6).

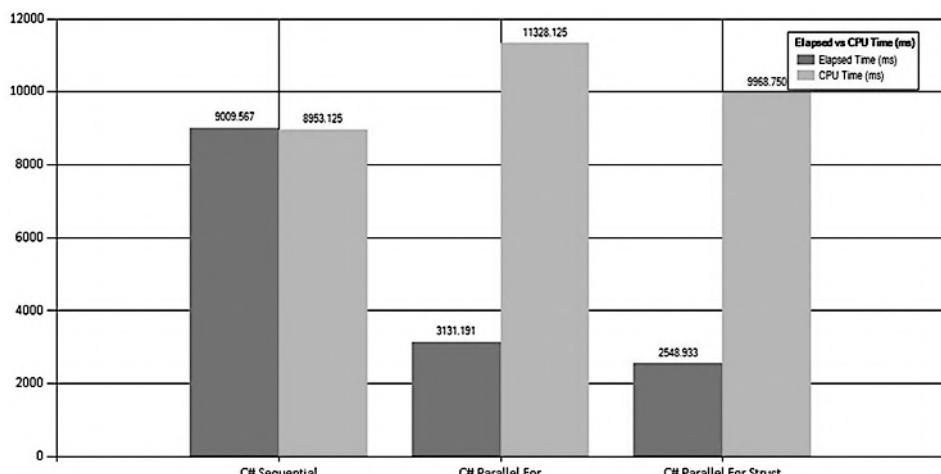


Рис. 4.6. Результаты сравнительных тестов использования конструкции `Parallel.For` в алгоритме Мандельброта, выполняемом на четырехъядерном компьютере с 8 Гбайт оперативной памяти. Последовательный код выполняется за 9,009 с; параллельная версия — за 3,131 с, что почти в три раза быстрее. В правом столбце показана более высокая производительность, достигаемая в параллельной версии кода, в котором для представления комплексного числа вместо ссылочного типа использован тип-значение. Этот код выполняется за 2,548 с, что на 20 % быстрее, чем исходный параллельный код, потому что во время его выполнения не задействованы GC-поколения, замедляющие процесс

Наглядное улучшение: при применении типа `struct` вместо ссылочного типа `class` количество GC-поколений для освобождения памяти сводится к нулю¹. В табл. 4.1 представлено сравнение количества GC-поколений для цикла `Parallel.For` при ис-

¹ Fundamentals of Garbage Collection, <http://mng.bz/v998>.

пользовании большого количества ссылочных типов (`class`) и цикла `Parallel.For` с задействованием большого количества типов-значений (`struct`).

Таблица 4.1. Сравнение количества операций по сборке мусора

Операция	GC gen0	GC gen1	GC gen2
Parallel.For	1390	1	1
Parallel.For для типа-значения struct	0	0	0

Версия кода с использованием ссылочного объекта `Complex` создает в памяти более 4 млн недолговечных элементов, размещаемых в куче¹. Короткоживущий объект хранится в GC поколения 0, и он должен быть удален из памяти раньше, чем объекты поколений 1 и 2. Такое большое количество операций по выделению памяти заставляет выполнять сборку мусора, для чего приостанавливаются все работающие в это время потоки, за исключением тех, что необходимы для GC. Прерванные задачи возобновляются только после завершения сборки мусора. Очевидно, что чем меньше GC-поколений, тем быстрее выполняется приложение.

4.2.2. Оборотная сторона параллельных циклов

В предыдущем разделе мы выполняли последовательную и параллельную версии алгоритма Мандельброта, сравнивая их производительность. Параллельный код был реализован с помощью класса `Parallel` из библиотеки TPL и конструкции `Parallel.For`, обеспечивающей значительное повышение производительности по сравнению с обычными последовательными циклами.

Как правило, шаблон с использованием параллельного цикла `for` полезен для операций, которые могут выполняться независимо для каждого элемента коллекции (элементы которой не зависят один от другого). Например, изменяемые массивы идеально подходят для параллельных циклов, потому что каждый элемент такого массива занимает свое место в памяти и его обновление не приведет к состоянию гонки. Однако при распараллеливании цикла могут возникнуть сложности, способные впоследствии привести к проблемам, которые не являются типичными или даже вообще не встречаются в последовательном коде. Например, в последовательном коде часто используется переменная, играющая роль аккумулятора для чтения или записи. При попытке распараллелить цикл, где применяется такой аккумулятор, существует высокая вероятность получить состояние гонки, поскольку будет осуществляться конкурентный доступ нескольких потоков к таким переменным.

В параллельном цикле `for` степень параллелизма по умолчанию зависит от количества доступных ядер. *Степенью параллелизма* называется число итераций, которые могут выполняться одновременно на данном компьютере. Как правило, чем больше

¹ Dynamic Memory Allocation, <http://mng.bz/w8kA>.

число доступных ядер, тем быстрее выполняется параллельный цикл `for`. Это верно до тех пор, пока не будет достигнута точка падения эффективности, предсказанная по закону Амдала (скорость параллельного цикла зависит от выполняемой работы).

4.3. Измерение производительности

Главная цель написания параллельного кода — это, безусловно, повышение производительности. Увеличение производительности, полученное за счет выполнения параллельной программы на многоядерном компьютере, по сравнению с одноядерным компьютером, называется *ускорением*.

При оценке ускорения следует учесть несколько аспектов. Типичным способом повышения скорости является разделение работы между доступными ядрами. Таким образом, при запуске по одной задаче на процессор ожидается, что на компьютере с n ядрами программа будет выполняться в n раз быстрее, чем исходная последовательная программа. Такой результат называется *линейным ускорением*, которое на практике недостижимо вследствие издержек, связанных с созданием и координацией потоков. В случае параллелизма, требующего создания нескольких потоков и разделения данных между ними, эти издержки только увеличиваются. При измерении ускорения приложения опорной точкой бенчмаркинга считается выполнение программы на одноядерном компьютере.

Линейное ускорение последовательной программы, преобразованной в параллельную версию, вычисляется по следующей формуле: *ускорение = время последовательного выполнения / время параллельного выполнения*. Например, если время выполнения приложения на одноядерном компьютере составляет 60 мин, а на двухъядерном оно уменьшается до 40 мин, то ускорение равно 1,5 ($60 / 40$).

Почему время выполнения не сократилось до 30 мин? Потому что распараллеливание приложения связано с определенными издержками, которые не позволяют получить линейное ускорение, пропорциональное количеству ядер. Эти издержки вызваны созданием новых потоков, что подразумевает конфликты, переключение контекста и планирование потоков.

Измерение производительности и ожидаемое ускорение представляют собой основы для бенчмаркинга, проектирования и реализации параллельных программ. По этой причине параллельное выполнение является дорогим удовольствием — оно вовсе не бесплатно и требует времени на планирование. Внутренние издержки связаны с созданием и согласованием потоков. Иногда, если объем работы слишком мал, издержки, вносимые параллелизмом, могут превышать выгоду от него и, соответственно, перекрывают прирост производительности. Часто масштабы и объем проблемы влияют на структуру кода и время, необходимое для его выполнения. Иногда повышение производительности достигается за счет выбора более масштабируемого решения проблемы.

Еще одним инструментом, позволяющим оценить эффективность инвестиций в параллелизм, является закон Амдала — популярная формула вычисления ускорения для параллельной программы.

4.3.1. Определение предела повышения эффективности по закону Амдала

Итак, теперь вы знаете, что для повышения производительности программы и сокращения общего времени выполнения кода необходимо использовать преимущества параллельного программирования и доступные ресурсы многоядерной системы. Почти в каждой программе есть код, который должен выполняться последовательно для координации параллельного выполнения. В примере с алгоритмом Мандельброта таким последовательным процессом является рендеринг изображения. Другой распространенный пример — шаблон Fork/Join, который запускает параллельное выполнение нескольких потоков, а затем ожидает их завершения, прежде чем продолжить работу.

В 1965 г. Джин Амдал (Gene Amdahl) пришел к выводу, что наличие в программе последовательного кода ставит под угрозу общее повышение производительности. Данная концепция учитывает идею линейного ускорения. Линейное ускорение означает, что время T (количество единиц времени), которое требуется для выполнения задачи на p процессорах, равно T / p (время, необходимое для выполнения задачи на одном процессоре). Это можно объяснить тем, что программы не способны работать полностью параллельно, так что ожидаемое повышение производительности не является линейным — оно ограничено последовательным участком кода.

Закон Амдала гласит, что для набора данных определенного размера максимальное повышение производительности программы, реализованной с применением параллелизма, ограничено временем, необходимым для выполнения последовательной части программы. Согласно закону Амдала, независимо от того, сколько ядер задействовано в параллельных вычислениях, максимальное ускорение программы зависит от доли времени, затрачиваемой на последовательную обработку.

Закон Амдала определяет ускорение параллельной программы с использованием следующих трех переменных:

- базовой продолжительности выполнения программы на одноядерном компьютере;
- количества доступных ядер;
- доли параллельного кода.

Ускорение программы по закону Амдала вычисляется по следующей формуле:

$$\text{ускорение} = 1 / (1 - P + (P / N)).$$

Числитель этой формулы всегда равен 1, поскольку он представляет собой базовое время выполнения программы. В знаменателе переменная N равна количеству доступных ядер, а P — доле параллельного кода.

Например, если параллелизуемый код составляет 70 %, то на четырехядерном процессоре максимальное ожидаемое ускорение составляет 2,13:

$$\text{ускорение} = 1 / (1 - 0,70 + (0,70 / 4)) = 1 / (0,30 + 0,17) = 1 / 0,47 = 2,13 \text{ раза.}$$

Есть ряд условий, которые могут поставить под сомнение результат этой формулы. Когда речь идет о распараллеливании данных, при обработке больших объемов информации часть кода, выполняемая параллельно для анализа данных, оказывает большее влияние на производительность в целом. Более точная формула для оценки повышения производительности в результате распараллеливания описывается законом Густафсона.

4.3.2. Закон Густафсона: еще один шаг вперед в измерении повышения производительности

Закон Густафсона считается развитием закона Амдала; он рассматривает ускорение с другой, более современной точки зрения — с учетом увеличения числа доступных ядер и объема обрабатываемых данных.

Закон Густафсона учитывает переменные, которые отсутствуют в расчете повышения производительности по закону Амдала, получая более реалистичную формулу для современных сценариев вычислений, таких как увеличение параллельной обработки благодаря использованию многоядерного оборудования.

Каждый год объем обрабатываемых данных экспоненциально возрастает, тем самым увеличивая применение параллелизма, распределенных систем и облачных вычислений при разработке программного обеспечения. Сегодня это важный фактор, вследствие которого на смену закону Амдала приходит закон Густафсона.

По закону Густафсона ускорение рассчитывается по следующей формуле:

$$\text{ускорение} = S + (N \times P),$$

где S — количество модулей, обрабатываемых последовательно; P — количество модулей, которые могут выполняться параллельно, а N — количество доступных ядер.

Подведем итоги: закон Амдала позволяет предсказать ускорение, достижимое путем параллелизации последовательного кода, а закон Густафсона помогает вычислить ускорение, достигаемое с использованием существующей параллельной программы.

4.3.3. Ограничения параллельных циклов: сумма простых чисел

В этом разделе будут рассмотрены некоторые ограничения, вытекающие из последовательной семантики параллельного цикла, и технологии преодоления подобных недостатков. Для начала рассмотрим простой пример распараллеливания при суммировании коллекции простых чисел. В листинге 4.4 вычисляется сумма простых чисел для коллекции, состоящей из 1 млн элементов. Такое вычисление — идеальный кандидат на распараллеливание, поскольку на каждой итерации всегда выполняется одна и та же операция. В реализации кода пропущена последовательная версия, время выполнения которой для данного случая составляет 6,551 с.

Это значение будет использовано как опорное для сравнения скорости выполнения параллельной версии кода.

Листинг 4.4. Параллельное суммирование простых чисел с использованием конструкции цикла Parallel.For

```
int len = 10000000;           | Переменная total используется
long total = 0;              | в качестве аккумулятора

Func<int, bool> isPrime = n => {   | Функция isPrime определяет,
    if (n == 1) return false;      | является ли число простым
    if (n == 2) return true;
    var boundary = (int)Math.Floor(Math.Sqrt(n));
    for (int i = 2; i <= boundary; ++i)
        if (n % i == 0) return false;
    return true;
};

Parallel.For(0, len, i => {       | Для доступа к текущему счетчику
    if (isPrime(i))             | в конструкции цикла Parallel.For
        total += i;              | используется анонимная лямбда-функция
});                                | Если значение счетчика i является простым числом,
                                         | то оно прибавляется к аккумулятору total
```

Функция `isPrime` — это простая реализация, используемая для проверки того, является ли данное число простым. Переменная `total` применяется в цикле `for` в качестве аккумулятора для суммирования всех простых чисел коллекции. Время выполнения кода на четырехъядерном компьютере составляет 1,049 с. Скорость выполнения параллельного кода в шесть раз выше, чем скорость последовательного кода. Отлично! Впрочем, не спешите с выводами.

Если мы запустим код снова, то получим другое значение аккумулятора `total`. Код не является детерминированным: при каждом новом запуске результат будет другим, поскольку переменная-аккумулятор `total` открыта для совместного доступа для разных потоков.

Одним из простых решений будет использование блокировки для синхронизации доступа потоков к переменной `total`. Однако затраты на синхронизацию при таком решении снижают производительность. Лучшим выходом будет применение переменной `ThreadLocal<T>` для хранения локального состояния потока во время выполнения цикла. К счастью, одно из переопределений `Parallel.For` предусматривает встроенную конструкцию для создания в потоке локального объекта. Каждый поток имеет собственный экземпляр `ThreadLocal`, что исключает возможность существования нежелательных разделенных состояний. Тип `ThreadLocal<T>` принадлежит пространству имен `System.Threading` и выделен в листинге 4.5 жирным шрифтом.

В коде по-прежнему используется глобальная общая переменная `total`, но другим способом. В этой версии кода третий параметр цикла `Parallel.For` инициализирует локальное состояние, время жизни которого — от первой до последней итерации текущего потока. Таким образом, каждый поток задействует свою локальную переменную для работы с изолированной копией состояния, и эта переменная хранится и читается отдельно, потокобезопасным способом.

Листинг 4.5. Использование Parallel.For с переменными ThreadLocal

```

Parallel.For(0, len,
    () => 0,
    (int i, ParallelLoopState loopState, long tlsValue) => {
        return isPrime(i) ? tlsValue += i : tlsValue;
    },
    value => Interlocked.Add(ref total, value));

```

Переменная `total` используется в качестве аккумулятора

Функция `isPrime` определяет, является ли число простым

Функции инициализации создают защищенные копии переменной `tlsValue` для каждого потока; благодаря переменной `ThreadLocal` каждый поток получает доступ к собственной копии

Поскольку фрагмент данных хранится в управляемой локальной памяти потока (Thread-Local Storage, TLS), как показано в примере, этот фрагмент униклен для потока. В таком случае поток называется *владельцем* этих данных. Назначение локальной памяти потока — избежать издержек на блокировки и синхронизацию для доступа к разделяемому состоянию. В нашем примере копия локальной переменной `tlsValue` присваивается и используется каждым потоком в отдельности для вычисления суммы заданного диапазона коллекции, которая была разделена с помощью алгоритма параллельного разделителя. Параллельный разделитель — это сложный алгоритм, вычисляющий наилучший вариант разделения и распределения фрагментов коллекции между потоками.

После того как поток завершает все итерации, вызывается последний параметр цикла `Parallel.For`, который определяет операцию `join`. Затем в процессе операции `join` результаты всех потоков объединяются. На данном этапе применяется класс `Interlocked`, выполняющий скоростную и потокобезопасную операцию сложения. Этот класс был использован в главе 3 для выполнения операций CAS с безопасным изменением (обменом) значений объекта в многопоточных средах. В классе `Interlock` реализованы и другие полезные операции, такие как инкремент, декремент и обмен переменными.

В этом разделе упоминается важный термин из области параллелизма данных — агрегат. Концепция агрегирования будет подробно рассмотрена в главе 5.

В листинге 4.5 представлена окончательная версия кода, позволяющая получить детерминированный результат со скоростью 1,178 с — почти так же быстро, как и предыдущий вариант. В обмен на незначительные дополнительные издержки мы получили правильный код. При использовании разделяемого состояния в параллельном цикле часто теряется масштабируемость из-за необходимости синхронизации доступа к разделяемым состояниям.

4.3.4. Что может пойти не так в простом цикле

Рассмотрим простой фрагмент кода, в котором суммируются целые числа из заданного массива (листинг 4.6). Задействуя любой язык ООП, вы можете написать примерно такую программу.

Листинг 4.6. Типичный цикл for

```
int sum = 0;
for (int i = 0; i < data.Length; i++)
{
    sum += data[i];
```

← Переменная sum используется как аккумулятор,
значение которого обновляется при каждой итерации

Каждому программисту в его карьере приходилось писать что-то подобное — скопее всего, несколько лет назад, когда программы были однопоточными. Тогда данный код был хорош, но сегодня мы имеем дело с другими сценариями, в которых сложные системы и программы выполняют несколько задач одновременно. В этих новых условиях такой код может содержать малозаметную ошибку в строке суммирования:

```
sum += data[i];
```

Что произойдет, если значения массива изменятся, пока выполняется этот код? В многопоточной программе такой код чреват проблемой изменяемости и не гарантирует согласованности.

Обратите внимание, что изменяемое состояние — не всегда однозначное зло. Если изменяемое состояние является видимым только в пределах одной функции, — это неизящно, но безвредно. Например, суммирование в цикле `for` в предыдущем примере может быть изолировано в функции следующим образом:

```
int Sum(int[] data)
{
    int sum = 0;
    for (int i = 0; i < data.Length; i++)
    {
        sum += data[i];
    }
}
```

Значение `sum` обновляется, но его изменение не видно вне области действия функции. В результате такую реализацию суммирования можно считать чистой функцией.

Чтобы снизить степень сложности и уменьшить вероятность ошибки в программе, нужно повысить уровень абстракции кода. Например, чтобы вычислить сумму числовых значений, можно выразить свое намерение в виде «что я хочу получить» вместо «как это должно быть сделано». Общая функциональность должна быть частью языка, поэтому вы можете выразить свои намерения так:

```
int sum = data.Sum();
```

Действительно, в .NET есть метод расширения `Sum` (<http://mng.bz/f3nF>), он является частью пространства имен `System.Linq`. В этом пространстве имен есть также много других методов, в том числе `List` и `Array`, которые расширяют функциональность для любого объекта `IEnumerable` (<http://mng.bz/2bBv>). Не случайно идеи, лежащие в основе LINQ, происходят из концепций функционального программирования. Пространство имен LINQ повышает неизменяемость; вместо изменяемости оно

оперирует понятием преобразования, так что LINQ-запросы (и лямбда-функции) позволяют преобразовывать наборы структурированных данных из одной формы в другую, не беспокоясь о побочных эффектах или состояниях.

4.3.5. Модель декларативного параллельного программирования

В примере суммирования простых чисел в листинге 4.5 конструктор цикла `Parallel.For` определенно обеспечивает ускорение по сравнению с последовательным кодом и делает это эффективно, хотя такая реализация немного сложнее для понимания и поддержки, чем последовательная версия. Окончательный код едва ли будет понятен разработчику с первого взгляда. В конечном счете цель этого кода состоит в том, чтобы суммировать простые числа коллекции. Было бы неплохо иметь возможность описать назначение программы, представив пошаговую реализацию алгоритма.

Именно здесь в игру вступает PLINQ. Код из листинга 4.7 эквивалентен предыдущей параллельной функции `Sum`, но в нем вместо цикла `Parallel.For` используется PLINQ-запрос (выделен жирным шрифтом).

Листинг 4.7. Параллельное суммирование коллекции с использованием декларативного PLINQ-запроса

```

long total = 0;
Parallel.For(0, len,
    () => 0,
    (int i, ParallelLoopState loopState, long tlsValue) => {
        return isPrime(i) ? tlsValue += i : tlsValue;
},
value => Interlocked.Add(ref total, value));

→ long total = Enumerable.Range(0, len).AsParallel()
    .Where(isPrime).Sum(x => (long)x); ←

```

Сумма значений с преобразованием результата
в тип long во избежание исключения переполнения

Параллельное суммирование
с использованием конструкции Parallel.For

Параллельное суммирование
с применением PLINQ-запроса

Функциональный декларативный подход занимает всего одну строку кода. Разумеется, по сравнению с реализацией цикла `for` это легко читаемая, краткая, хорошо поддерживаемая запись без изменяемых состояний. Конструкция PLINQ представляет собой код в виде цепочки функций, каждая из которых выполняет мелкие действия для выполнения общей задачи. В решении использована агрегирующая часть, реализованная в виде функции более высокого порядка из LINQ/PLINQ API, — в данном случае это функция `Sum()`. Агрегат применяет функцию к каждому следующему элементу коллекции и возвращает агрегированный результат для всех предыдущих элементов. Есть и другие распространенные функции агрегирования: `Average()`, `Max()`, `Min()` и `Count()`.

На рис. 4.7 показаны результаты сравнительных тестов времени выполнения параллельной функции `Sum`.

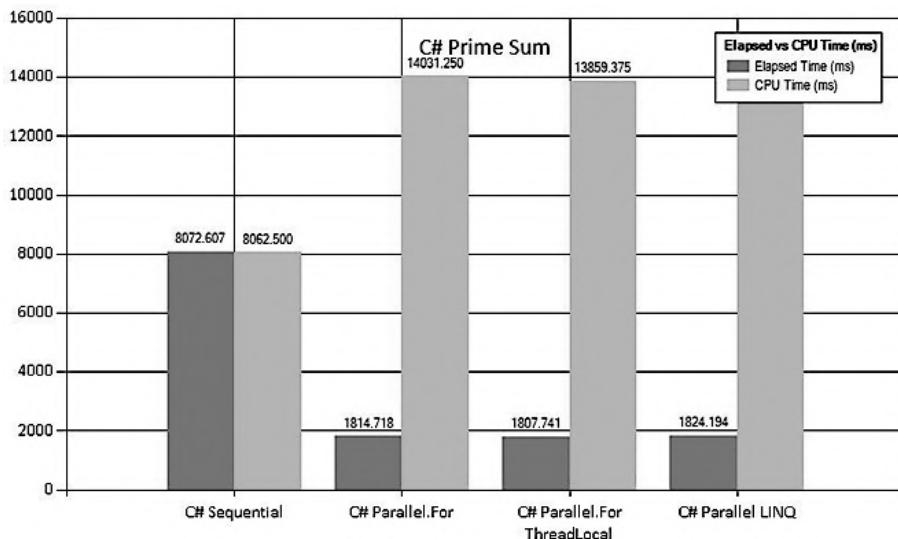


Рис. 4.7. Сравнение производительности суммирования простых чисел. Бенчмаркинг проводился на восьмипроцессорной машине с 8 Гбайт оперативной памяти. Последовательная версия выполняется за 8,072 с. Это значение использовалось как опорное для других версий кода. Выполнение версии с Parallel.For заняло 1,814 с, что примерно в 4,5 раза быстрее, чем последовательный код. Версия с Parallel.For ThreadLocal не намного быстрее, чем параллельный цикл. Наконец, программа с применением PLINQ-запроса является самой медленной среди параллельных версий; на ее выполнение потребовалось 1,824 с

Агрегирование значений во избежание исключения арифметического переполнения

Показанный выше PLINQ-запрос не оптимизирован. Вскоре вы узнаете, как сделать этот код более эффективным. Более того, размер суммируемой последовательности был уменьшен до 10 000 вместо использованного ранее 1 млн, потому что функция `Sum()` в PLINQ скомпилирована для выполнения в проверенном блоке, что вызывает исключение арифметического переполнения. Решение состоит в том, чтобы преобразовать числа из формата integer-32 в формат integer-64 (long) или применять функцию `Aggregate`:

```
Enumerable.Range(0, len) .AsParallel()
    .Aggregate((acc,i) => isPrime(i) ? acc += i : acc);
```

Функция `Aggregate` будет подробно рассмотрена в главе 5.

Резюме

- ❑ Распараллеливание данных направлено на обработку огромных объемов данных путем их разделения и выполнения каждого блока по отдельности, а после завершения указанных операций — объединения результатов. Это позволяет анализировать фрагменты параллельно, получая выигрыш в скорости и производительности.
- ❑ В этой главе использованы следующие ментальные модели распараллеливания данных: Fork/Join, параллельное сокращение данных, параллельная агрегация. Указанные шаблоны проектирования имеют общий принцип: разделение данных и выполнение одной и той же задачи параллельно для каждой из разделенных частей.
- ❑ Задействуя конструкции функционального программирования, можно писать сложный код для обработки и анализа данных в простом декларативном стиле. Эта парадигма позволяет достичь параллелизма при незначительном изменении кода.
- ❑ Профилирование программы — способ понять ее и гарантировать, что изменения, внесенные в код для достижения параллелизма, были полезными. Для этого нужно измерить скорость программы, выполняемой последовательно, и использовать ее как опорную точку при бенчмаркинге для сравнения с измененным кодом.

PLINQ и MapReduce: распараллеливание данных, часть 2

В этой главе:

- использование семантики декларативного программирования;
- изоляция и контроль побочных эффектов;
- реализация и применение параллельной функции Reduce;
- максимизация использования аппаратных ресурсов;
- реализация многоразового параллельного шаблона MapReduce.

В этой главе представлен MapReduce — один из самых распространенных шаблонов функционального программирования в разработке программного обеспечения. Прежде чем приступить к изучению MapReduce, мы проанализируем стиль декларативного программирования, на который делает упор функциональная парадигма, используя PLINQ-запросы и инструкции `PSeq` в F#. Обе эти технологии анализируют оператор запроса в процессе выполнения программы и выбирают наилучшее стратегическое решение о том, как выполнить запрос в соответствии с доступными системными ресурсами. Таким образом, чем больше процессорная мощность компьютера, тем быстрее будет работать код. Применяя эти стратегии, можно уже сейчас разрабатывать код для компьютеров следующего поколения. Затем вы узнаете, как реализовать в .NET параллельную функцию `Reduce`, которую вы сможете много-кратно использовать в повседневной работе, чтобы увеличить скорость выполнения функций агрегирования.

Задействуя принципы ФП, вы сможете применять в своих программах распараллеливание данных, не усложняя код, по сравнению с обычным программированием.

Вместо процедурной семантики ФП предпочитает декларативную: выражать намерение программы, а не описывать шаги для выполнения этой задачи. Такой декларативный стиль программирования упрощает применение параллелизма.

5.1. Краткое введение в PLINQ

Прежде чем углубиться в язык PLINQ, я дам определение его последовательному двойнику, языку запросов LINQ: это расширение .NET Framework, которое обеспечивает стиль декларативного программирования, повышая уровень абстракции и упрощая приложение посредством использования широкого набора операций, позволяющих преобразовать любой объект, который реализует интерфейс `IEnumerable`. Наиболее распространенными операциями LINQ являются отображение, сортировка и фильтрация. Операторы LINQ принимают в виде параметра поведение, которое обычно передается в виде лямбда-выражений. Такие переданные лямбда-выражения применяются к каждому элементу последовательности. С появлением LINQ и лямбда-выражений функциональное программирование в .NET становится реальностью.

Запросы можно выполнять параллельно, используя все ядра среды разработки. Для этого в запросы добавляется расширение `.AsParallel()`, и LINQ превращается в PLINQ. PLINQ можно определить как конкурентный механизм для выполнения LINQ-запросов. Целью параллельного программирования является максимальное использование процессора и повышение пропускной способности в многоядерной архитектуре. Приложение, работающее на многоядерном компьютере, должно распознавать количество доступных процессорных ядер и соответственно масштабировать производительность.

Лучший способ писать параллельные приложения — не думать о параллелизме. PLINQ идеально подходит для данной абстракции, потому что берет на себя выполнение всех основных требований, таких как разбиение последовательностей на более мелкие куски, их независимая обработка и применение логики к каждому элементу подпоследовательности. Звучит знакомо? Это потому, что в основе PLINQ лежит модель Fork/Join (рис. 5.1).

Как правило, каждый раз, когда в коде встречается цикл `for` или `for-each`, который что-то делает с коллекцией и не имеет побочных эффектов вне цикла, есть смысл подумать о преобразовании этого цикла в LINQ-запрос. Затем проведите бенчмаркинг и оцените, имеет ли смысл распараллелить данный запрос с использованием PLINQ.

ПРИМЕЧАНИЕ

Книга посвящена конкурентности, поэтому в дальнейшем здесь будет идти речь только о PLINQ. Но в большинстве случаев те же конструкции, принципы и функции более высокого порядка, используемые для описания запроса, применимы и для LINQ.

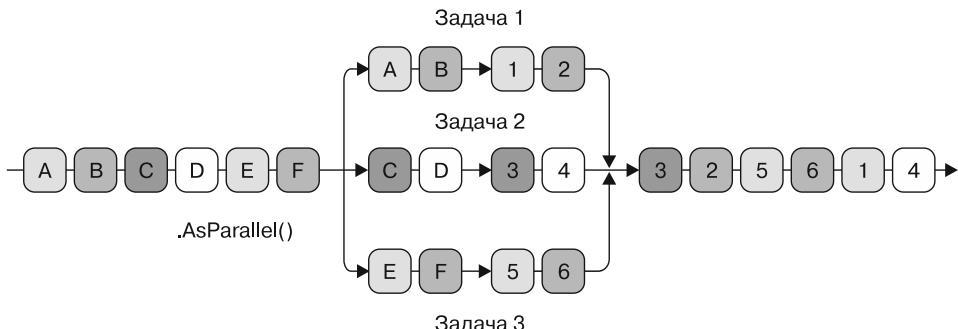


Рис. 5.1. Модель выполнения PLINQ. Преобразовать запрос LINQ в PLINQ легко: нужно только добавить метод расширения AsParallel(), который выполняется параллельно, с использованием шаблона Fork/Join. На этом рисунке входные символы параллельно преобразуются в числа. Обратите внимание на то, что последовательность входных элементов на выходе не сохраняется

Преимущество использования PLINQ по сравнению с параллельным циклом `for` заключается в возможности автоматического агрегирования результатов временной обработки в каждом потоке, который выполняет запрос.

5.1.1. Почему язык PLINQ более функциональный

PLINQ считается идеальной функциональной библиотекой. Но почему? Почему версия кода с использованием PLINQ считается более функциональной, чем первоначальный цикл `Parallel.For`?

При применении `Parallel.For` вы говорите компьютеру, что делать:

- ❑ просмотреть коллекцию;
- ❑ проверить, является ли число простым;
- ❑ если число простое, то прибавить его к локальному аккумулятору;
- ❑ когда все итерации будут пройдены, прибавить значение аккумулятора к общему значению.

Используя запросы LINQ/PLINQ, можно сказать компьютеру, что должно получиться в результате, примерно в такой форме: «В диапазоне значений от 0 до 1 000 000, если число простое, прибавить его к сумме».

ФП предпочтает написание декларативного кода вместо императивного. Код, написанный в декларативном стиле, фокусируется на том, чего нужно достичь, а не на том, как это сделать. В PLINQ-запросах имеется тенденция подчеркивать назначение кода, а не механизм, так что такие запросы гораздо более функциональны.

ПРИМЕЧАНИЕ

В разделе 13.9 будет показано, как с использованием цикла `Parallel.ForEach` создать высокопроизводительный многоразовый оператор, который объединяет в себе функции `filter` и `map`. В этом случае, поскольку детали реализации функции абстрагированы и скрыты от глаз разработчика, параллельная функция `FilterMap` становится оператором более высокого порядка, который удовлетворяет концепции декларативного программирования.

Кроме того, в функциональном программировании поощряется использование функций для повышения уровня абстракции, что позволяет скрыть сложность. В этом отношении PLINQ повышает абстракцию модели конкурентного программирования, обрабатывая выражения запроса, анализируя их структуру и принимая решение, как построить параллельную работу, чтобы получить максимальную скорость.

В ФП также поощряется объединение небольших простых функций для решения сложных задач. Конвейер PLINQ полностью удовлетворяет этому принципу, позволяя объединять методы расширения в цепочки.

Еще одним функциональным аспектом PLINQ является отсутствие изменений. Операторы PLINQ не изменяют исходную последовательность в результате преобразования, а возвращают новую. Следовательно, функциональная реализация PLINQ дает предсказуемые результаты, даже если задачи выполняются параллельно.

5.1.2. PLINQ и чистые функции: параллельный счетчик слов

Рассмотрим пример, в котором программа загружает набор текстовых файлов из заданной папки, анализирует каждый документ и на выходе представляет список из десяти наиболее часто используемых слов. Для этого выполняется такая последовательность операций (рис. 5.2).

1. Выбрать файлы из заданной папки.
2. Просмотреть все файлы.
3. Прочитать содержимое каждого текстового файла.
4. Каждую строку разделить на слова.
5. Преобразовать буквы каждого слова в верхний регистр, чтобы было удобнее сравнивать.
6. Сгруппировать коллекцию по одинаковым словам.
7. Упорядочить коллекцию в порядке убывания частоты слов.
8. Выбрать первые десять результатов.
9. Представить результат в формате таблицы (словаря).

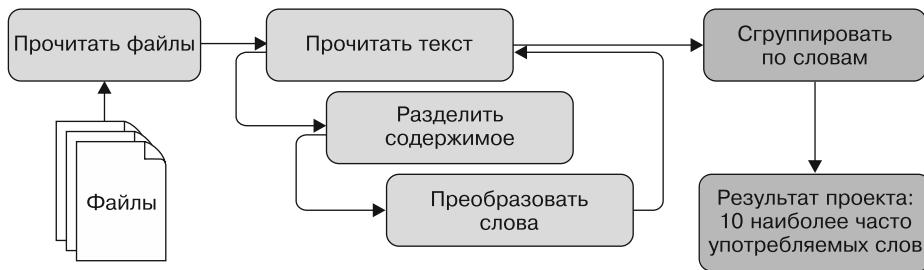


Рис. 5.2. Представление последовательности операций для подсчета частоты употребления каждого слова. Сначалачитываются файлы из заданной папки, затемпрочитывается каждый текстовый файл, его содержимое разделяется на строки и отдельные слова, которые потом группируются

В листинге 5.1 эта функциональность описана в виде метода `WordsCounter`, который принимает в качестве входного аргумента путь к папке, а затем вычисляет, сколько раз каждое слово встречается во всех файлах. Команда `AsParallel` в данном листинге выделена жирным шрифтом.

Листинг 5.1. Параллельная программа подсчета слов с побочными эффектами

```

public static Dictionary<string, int> WordsCounter(string source)
{
    var wordsCount =
        (from filePath in
            Directory.GetFiles(source, "*.txt")
            .AsParallel()
            from line in File.ReadLines(filePath)
            from word in line.Split(' ')
            select word.ToUpper())
        .GroupBy(w => w)
        .OrderByDescending(v => v.Count()).Take(10);
    return wordsCount.ToDictionary(k => k.Key, v => v.Count());
}
  
```

Побочный эффект чтения
файловой системы

Параллельная обработка
последовательности файлов

Упорядочение слов по частоте употребления
и отбор первых десяти значений

Логика программы шаг за шагом соответствует описанной выше последовательности операций. Программа декларативная, легко читаемая и работает параллельно, но в ней есть скрытая проблема — имеется побочный эффект. Метод считывает файлы из файловой системы, генерируя побочный эффект ввода-вывода. Как отмечалось ранее, функция или выражение имеет побочный эффект, если она (оно) изменяет состояние вне своей области действия или если выходные данные зависят не только от входных данных. Передача одного и того же набора входных данных в функцию с побочными эффектами не гарантирует получение всегда одного и того же результата на выходе. Функции этого типа создают проблемы в конкурентном коде, поскольку побочные эффекты подразумевают изменяемость. Примерами функций с побочными эффектами

являются генератор случайных чисел, получение текущего системного времени, чтение данных из файла или из сети, вывод данных в консоль и т. п. Чтобы лучше понять, почему чтение данных из файла является побочным эффектом, представьте себе, что содержимое файла может измениться в любое время, и всякий раз, когда оно будет изменяться, функция будет возвращать другой результат. Более того, чтение файла может привести к ошибке, если за это время он будет удален. Нужно помнить, что при каждом вызове данной функция может возвращать другой результат.

Из-за наличия побочных эффектов возникают следующие вопросы.

- ❑ Действительно ли безопасно запускать этот код параллельно?
- ❑ Является ли результат детерминированным?
- ❑ Как протестировать этот метод?

Функция, на вход которой поступает путь к каталогу файловой системы, может выдать ошибку, если этот каталог не существует или если у программы нет прав на чтение из него. Еще один момент, который следует принять во внимание, — то, что при параллельной работе функции с использованием PLINQ выполнение запроса откладывается до его материализации. *Материализация* — термин, применяемый для указания того, когда именно запрос выполняется и выдает результат. По этой причине при последующей материализации PLINQ-запроса, содержащего побочные эффекты, могут генерироваться разные результаты в зависимости от базовых данных, которые могли измениться за это время. Результат не является детерминированным. Так может случиться, если между вызовами функции файл будет удален из каталога, и тогда будет выдано исключение.

Более того, функции с побочными эффектами (также называемые *нечистыми*) трудно тестировать. Одним из возможных решений будет создание тестового каталога с несколькими текстовыми файлами, которые не могут изменяться. Для проверки правильности функции при таком подходе нужно знать, сколько слов в этих файлах и сколько раз они были использованы. Другим вариантом является имитация каталога и содержащихся в нем данных, но такое решение может оказаться еще сложнее предыдущего. Существует наилучший подход: устраниТЬ побочные эффекты и повысить уровень абстракции, упростив код и отделив его от внешних зависимостей.

Но что такое побочные эффекты? Что такое чистые функции и зачем их создавать?

5.1.3. Исключение побочных эффектов посредством чистых функций

Одним из принципов функционального программирования является чистота. *Чистыми функциями* называются функции без побочных эффектов; функции, результат которых не зависит от состояния, способного изменяться со временем. Другими словами, чистые функции всегда возвращают одно и то же значение при одних и тех же входных данных. В листинге 5.2 показаны примеры чистых функций на C#.

Листинг 5.2. Чистые функции на C #

```
public static string AreaCircle(int radius) =>
    Math.Pow(radius, 2) * Math.PI; ← У этих функций нет побочных эффектов,
public static int Add(int x, int y) => x + y; ← поэтому их выходные результаты
                                                никогда не изменяются
```

Данный листинг является примером побочных эффектов, которые представляют собой функции, изменяющие состояние, так как они меняют значение глобальных переменных. Поскольку переменные живут внутри блока, в котором они объявлены, переменная, определенная глобально, создает возможную коллизию и делает программу менее удобной для чтения и поддержки. Наличие глобальной переменной требует дополнительной проверки ее текущего значения в любой точке и при каждом вызове. Основная проблема с побочными эффектами заключается в том, что в конкурентной среде программа с ними становится непредсказуемой и противоречивой, поскольку побочный эффект представляет собой один из вариантов изменяемости.

Представьте себе, что мы передаем функции один и тот же аргумент и каждый раз получаем разные результаты. Говорят, что функция имеет побочный эффект, если она выполняет любое из следующих действий:

- выполняет операцию ввода-вывода (чтение/запись в файловую систему, базу данных или консоль);
- изменяет глобальное состояние или любое состояние вне ее области действия;
- выдает исключения.

На первый взгляд, устранение побочных эффектов из программы может показаться крайне ограничивающим, но у кода, написанного в этом стиле, есть много преимуществ:

- легко определить, правильно ли работает программа;
- легко компоновать функции для создания нового поведения;
- легко изолировать и, следовательно, тестировать и получить программу, менее подверженную ошибкам;
- легко организовать параллельное выполнение. Поскольку чистые функции не имеют внешних зависимостей, последовательность их выполнения (вычислений) не имеет значения.

Как видим, введение чистых функций в ваш инструментарий сразу приносит пользу коду. Более того, результат применения чистых функций зависит только от их входных данных, что вводит свойство *ссылочной прозрачности*.

Ссылочная прозрачность

Ссылочная прозрачность имеет фундаментальное значение для функций, свободных от побочных эффектов. Она предоставляет возможность заменить вызов функции для определенного набора параметров на возвращаемое значение без изменения результата

программы, и именно поэтому она является желательным свойством. Используя ссылочную прозрачность, можно заменить выражение его значением — и ничего не изменится. Данная концепция заключается в возможности представить результат любой чистой функции непосредственно, без вычисления. Последовательность вычислений неважна, и многократное выполнение функции всегда приводит к одному и тому же результату, поэтому такое действие может быть легко распараллелено.

Математика всегда ссылочно-прозрачна. Для определенной функции и определенных входных значений словари этой функции всегда будут иметь один и тот же результат для одних и тех же входных данных. Например, функция вида $f(x) = y$ является чистой, если для одного и того же значения x мы всегда получим один и тот же результат y при отсутствии изменений внутренних или внешних состояний.

Разумеется, в случае, если программа делает что-либо полезное, побочные эффекты неизбежны. Функциональное программирование не запрещает побочные эффекты, но поощряет их минимизацию и изоляцию.

5.1.4. Изоляция и контроль побочных эффектов: рефакторинг параллельного счетчика слов

Пересмотрим пример счетчика слов, представленный в листинге 5.1. Как можно изолировать и контролировать побочные эффекты в этом коде?

```
static Dictionary<string, int> WordsCounter(string source) {
    var wordsCount = (from filePath in
        Directory.GetFiles(source, "*.txt") ←
        .AsParallel()
        from line in File.ReadLines(filePath)
        from word in line.Split(' ')
        select word.ToUpper())
        .GroupBy(w => w)
        .OrderByDescending(v => v.Count()).Take(10);
    return wordsCount.ToDictionary(k => k.Key, v => v.Count());
}
```

Эту функцию можно разделить на чистую функцию, образующую основу кода, и пару функций с побочными эффектами. Нельзя избежать побочного эффекта ввода-вывода, но можно отделить его от чистой логики. В листинге 5.3 логика подсчета слов в каждом файле вынесена в отдельную функцию и побочные эффекты изолированы от нее.

Новая функция `PureWordsPartitioner` является чистой, ее результат зависит только от входного аргумента. Эта функция не имеет побочных эффектов, и ее корректность легко проверить. Метод `WordsPartitioner`, наоборот, отвечает за чтение текстового файла из файловой системы, что представляет собой побочный эффект, а также за агрегирование результатов анализа.

Жирным шрифтом выделен побочный эффект, возникающий в результате чтения файловой системы

Листинг 5.3. Разделение логики и побочных эффектов

```

static Dictionary<string, int> PureWordsPartitioner
    (IEnumerable<IEnumerable<string>> content) =>
    (from lines in content.AsParallel() ←
        from line in lines
        from word in line.Split(' ')
        select word.ToUpper())
        .GroupBy(w => w)
        .OrderByDescending(v => v.Count()).Take(10)
        .ToDictionary(k => k.Key, v => v.Count()); ←

static Dictionary<string, int> WordsPartitioner(string source)
{
    var contentFiles =
        (from filePath in Directory.GetFiles(source, "*.txt")
            let lines = File.ReadLines(filePath)
            select lines);
    return PureWordsPartitioner(contentFiles); ←
}

```

Чистая функция без побочных эффектов;
операция ввода-вывода здесь удалена

Результат функции без побочных эффектов
может быть распараллелен без проблем

Вызов функции без побочных
эффектов из нечистой функции

**Код для объединения результатов
в один словарь с исключением дубликатов**

Как видно из этого примера, разделение чистой и нечистой частей кода не только облегчает тестирование и оптимизацию чистых частей, но также позволяет лучше контролировать побочные эффекты программы и помогает не создавать нечистых частей больше, чем необходимо. Проектирование с чистыми функциями и отделение побочных эффектов от чистой логики — два базовых принципа, которые при функциональном мышлении выдвигаются на первый план.

5.2. Агрегирование и сокращение данных в параллельных программах

В функциональном программировании *сверткой* (fold) или *сокращением и накоплением* называется функция более высокого порядка, которая сокращает исходную структуру данных, обычно представленную в виде последовательности элементов, в одно значение. Например, функция сокращения может вернуть среднее значение, выполнить суммирование, найти максимальное или минимальное значение последовательности чисел.

Функция свертки принимает начальное значение, обычно называемое *аккумулятором*, которое используется как контейнер для хранения промежуточных результатов. В качестве второго аргумента требуется бинарное выражение, играющее роль

функции *сокращения*; оно применяется к каждому элементу последовательности и возвращает новое значение аккумулятора. В общем случае при сокращении берется бинарный оператор, то есть функция с двумя аргументами, и производится вычисление по вектору или набору элементов размера n , обычно слева направо. Иногда для первой операции с первым элементом используется специальное начальное значение, поскольку предыдущего значения для этого случая не существует. На каждой итерации входными данными для бинарного выражения являются текущий элемент последовательности и значение аккумулятора; возвращаемое значение перезаписывается в аккумулятор. Конечным результатом будет последнее значение аккумулятора (рис. 5.3).

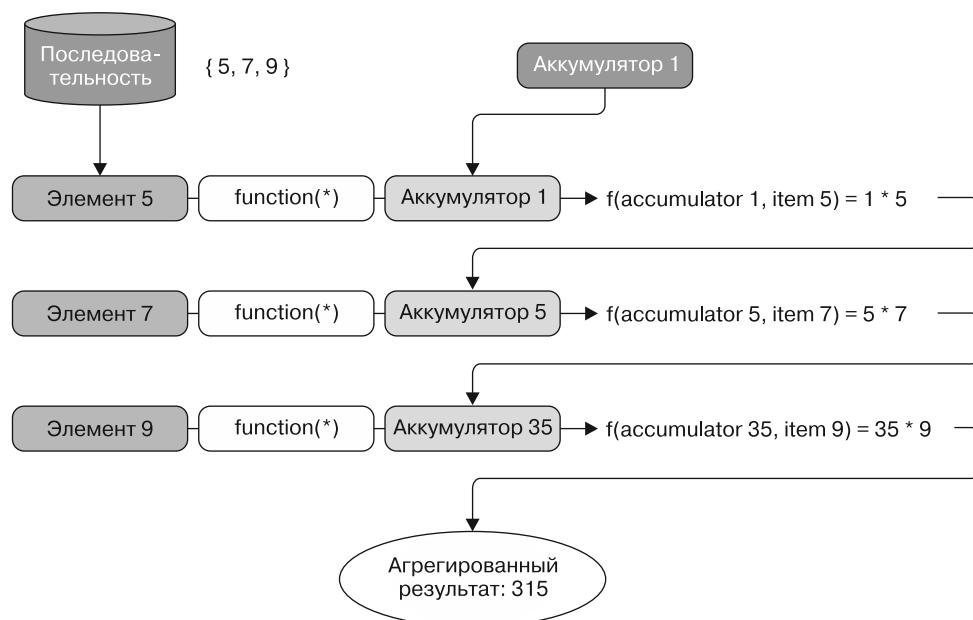


Рис. 5.3. Функция свертки сокращает последовательность до одного значения. В данном случае функция (f) выполняет умножение: она берет начальное значение аккумулятора, равное 1. На каждой итерации для последовательности $(5, 7, 9)$ функция выполняет вычисления для текущего элемента и аккумулятора. Результат вычислений используется для обновления аккумулятора новым значением

Функция свертки бывает двух видов — правая и левая, в зависимости от того, где находится первый элемент обрабатываемой последовательности. Правая свертка начинается с первого элемента списка идвигается вперед; левая свертка начинается с последнего элемента списка идвигается назад. В этом разделе рассматривается правая свертка, так как она используется чаще всего. Здесь и далее до конца раздела под словом «свертка» будет подразумеваться *правая свертка*.

ПРИМЕЧАНИЕ

При выборе между функциями правой и левой свертки следует учитывать несколько особенностей их влияния на производительность. При обработке списка скорость правой свертки составляет $O(1)$, поскольку в этом случае элемент добавляется в начало списка, что занимает постоянное время. Для левой свертки требуется время $O(n)$, поскольку в таком случае, чтобы добавить элемент, нужно пройти весь список. Левая свертка не может использоваться для обработки или генерации бесконечных списков, поскольку для того, чтобы начать свертку с последнего элемента, должен быть известен размер списка.

Функция свертки исключительно полезна и интересна: с ее помощью можно выразить в терминах агрегирования различные операции, такие как фильтрация, отображение и суммирование. Функция свертки — это, вероятно, самая трудная для изучения среди других функций для обработки списков, но и одна из самых мощных.

Если вы еще не читали статью Джона Хьюза (John Hughes) «Значение функционального программирования» (*Why Functional Programming Matters*, www.cs.kent.ac.uk/people/staff/dmr/whyfp90.pdf), то рекомендую прочесть. В ней подробно обсуждается широкая применимость и важность свертки в функциональном программировании. В листинге 5.4 показана реализация нескольких полезных функций на F# с помощью свертки.

Листинг 5.4. Реализация функций max и map на F# с помощью свертки

```

Функция map на F#
с использованием свертки
let map (projection:'a -> 'b) (sequence:seq<'a>) =
    sequence |> Seq.fold(fun acc item -> (projection item)::acc) []
Функция max на F#
с использованием свертки
let max (sequence:seq<int>) =
    sequence |> Seq.fold(fun acc item -> max item acc) 0
Функция filter на F#
с использованием свертки
let filter (predicate:'a -> bool) (sequence:seq<'a>) =
    sequence |> Seq.fold(fun acc item ->
        if predicate item = true then item::acc else acc) []
Функция length для вычисления размера
коллекции на F# с использованием свертки
let length (sequence:seq<'a>) =
    sequence |> Seq.fold(fun acc item -> acc + 1) 0

```

Эквивалентом свертки в языке LINQ на C# является агрегирование (Aggregate). В листинге 5.5 показана функция агрегирования на C#, примененная для реализации других полезных функций.

Благодаря поддержке параллелизма при генерации списков в .NET, включая операторы LINQ Aggregate и Seq.fold, реализация этих функций на C# и F# может быть легко преобразована для конкурентных вычислений. Более подробно это преобразование обсуждается в следующих разделах.

Листинг 5.5. Реализация функций Filter и Length с помощью LINQ-запросов на C# с использованием агрегирования

```

IEnumable<T> Map<T, R>(IEnumable<T> sequence, Func<T, R> projection){
    return sequence.Aggregate(new List<R>(), (acc, item) => {
        acc.Add(projection(item));
        return acc;
    });
}

int Max(IEnumable<int> sequence) { ←
    return sequence.Aggregate(0, (acc, item) => Math.Max(item, acc));
}

IEnumable<T> Filter<T>(IEnumable<T> sequence, Func<T, bool> predicate){
    return sequence.Aggregate(new List<T>(), (acc, item) => {
        if (predicate(item))
            acc.Add(item);
        return acc;
    });
}

int Length<T>(IEnumable<T> sequence) { ←
    return sequence.Aggregate(0, (acc, _) => acc + 1);
}

```

Функция Map с использованием
LINQ Aggregate на C#

Функция Max с использованием
LINQ Aggregate на C#

Функция Filter с использованием
LINQ Aggregate на C#

Функция Length с использованием
LINQ Aggregate на C#

5.2.1. Усечение: одно из многих преимуществ свертки

Многократное использование и удобство поддержки — лишь некоторые из преимуществ, обеспечиваемые благодаря функции свертки. Но есть одно свойство этой функции, заслуживающее особого внимания. Функция свертки может применяться для повышения производительности запросов с генерацией списков. Генерация списков — это конструкция, аналогичная LINQ/PLINQ в C# и предназначенная для облегчения списковых запросов к уже существующим спискам (https://en.wikipedia.org/wiki/List_comprehension).

Как функция свертки может повысить скорость выполнения спискового запроса независимо от параллелизма? Для того чтобы ответить на этот вопрос, проанализируем простой запрос PLINQ. Мы видели, что использование функциональных конструкций, таких как LINQ/PLINQ в .NET, преобразует исходную последовательность с исключением изменяемости, которая в строго определенных языках программирования, таких как F# и C#, часто приводит к созданию излишних промежуточных структур данных. В листинге 5.6 показан запрос PLINQ, который фильтрует и затем преобразует последовательность чисел для определения удвоенной суммы четных значений. Параллельное вычисление выделено жирным шрифтом.

Листинг 5.6. PLINQ-запрос для параллельного суммирования удвоенных значений четных чисел

```

var data = new int[100000];
for(int i = 0; i < data.Length; i++)
    data[i]=i;

long total =
    data.AsParallel()
        .Where(n => n % 2 == 0)
        .Select(n => n + n)
        .Sum(x => (long)x);

```

Из-за параллелизма последовательность обработки значений непостоянна; но ее результат детерминирован, поскольку операция суммирования коммутативна

← Значение преобразуется в тип long: Sum (x => (long)x), чтобы избежать исключения переполнения

В этих нескольких строках кода для каждого PLINQ-запроса `Where` и `Select` генерируются промежуточные последовательности, которые приводят к выделению лишней памяти. В случае преобразования больших последовательностей затраты на сборку мусора для освобождения памяти становятся все выше, что отрицательно сказывается на производительности. Размещение объектов в памяти обходится дорого; следовательно, нужна оптимизация, которая позволит избежать излишнего выделения памяти и тем самым ускорить выполнение функциональных программ. К счастью, создания таких ненужных структур данных часто можно избежать. Исключение промежуточных структур данных для уменьшения размера временной памяти называется *усечением*. Эту технологию можно легко использовать с помощью функции свертки более высокого порядка, получившей в LINQ имя `Aggregate`. Указанная функция позволяет избежать выделения памяти для промежуточных структур данных путем объединения нескольких операций, таких как `filter` и `map`, в один шаг — в противном случае потребовалось бы выделение памяти для каждой операции. В следующем примере кода показан PLINQ-запрос для параллельного суммирования удвоенных значений четных чисел с помощью оператора `Aggregate`:

```
long total = data.AsParallel().Aggregate(0L, (acc, n) =>
    n % 2 == 0 ? acc + (n + n) : acc);
```

У функции PLINQ `Aggregate` есть несколько перегрузок; в данном случае первым аргументом `0` является начальное значение аккумулятора `acc`, которое передается и обновляется на каждой итерации. Второй аргумент — это функция, выполняющая операцию для каждого элемента последовательности и обновляющая значение аккумулятора `acc`. В теле данной функции объединяется поведение ранее определенных расширений PLINQ `Where`, `Select` и `Sum`, так что получается тот же результат. Единственное отличие — время выполнения: исходный код выполнялся за 13 мс; обновленная версия кода с применением усечения — за 8 мс.

Усечение — это эффективный инструмент оптимизации при использовании «жадных» структур данных, таких как списки и массивы; но «ленивые» коллекции ведут себя иначе. Вместо генерации промежуточных структур данных «ленивые» последовательности хранят функцию, подлежащую отображению, и исходную структуру данных. Но и в этом случае мы получим улучшение производительности по сравнению с функцией без усечения.

5.2.2. Свертки в PLINQ: функции агрегирования

Концепции, которые касаются функции свертки, можно применить и к PLINQ-запросам на F# и C#. Как уже отмечалось, в PLINQ есть эквивалент функции `fold`, называемый `Aggregate`. `Aggregate` в PLINQ — это правая свертка. Вот одна из перегруженных сигнатур данной функции:

```
public static TAccumulate Aggregate<TSource, TAccumulate>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func);
```

Эта функция принимает три аргумента, которые отображаются на исходную последовательность: исходную последовательность для обработки, начальное значение аккумулятора и функцию `func`, обновляющую аккумулятор для каждого элемента.

Лучший способ понять, как работает `Aggregate`, — рассмотреть пример. В примере, представленном во врезке, описывается распараллеливание кластеризации методом k -средних с использованием PLINQ-запросов и функции `Aggregate`. На этом примере видно, насколько проще и эффективнее становится программа благодаря данной конструкции.

Кластеризация методом k -средних

Кластеризация методом k -средних, также называемая алгоритмом Ллойда, является неконтролируемым алгоритмом машинного обучения, распределяющим набор точек данных по кластерам, каждый из которых сосредоточен вокруг своего центроида. Центроидом кластера называется сумма его точек, разделенная на общее количество точек. Центроид представляет собой центр масс геометрической формы, имеющей однородную плотность.

Алгоритм кластеризации методом k -средних принимает входные данные и значение k , равное количеству кластеров, а затем случайным образом помещает центроиды в эти кластеры. В качестве параметра данный алгоритм принимает количество кластеров, которые нужно найти, и делает первоначальное предположение о центре каждого из них. Идея состоит в том, чтобы генерировать множество центроидов, создающих центры кластеров. Каждая точка данных связана с ближайшим центроидом. Расстояние вычисляется по простой евклидовой функции расстояния (https://en.wikipedia.org/wiki/Euclidean_distance и https://ru.wikipedia.org/wiki/Евклидова_метрика). Затем каждый центроид перемещается в среднее положение для всех связанных с ним точек. Центроид вычисляется как сумма его точек, разделенная на размер кластера. На каждой итерации выполняются следующие действия.

1. Сумма вычисляется как сумма точек в каждом кластере.
2. Сумма каждого кластера делится на количество точек в этом кластере.
3. Выполняется переназначение (отображение) точек каждого кластера на ближайший центроид.

4. Эти операции повторяются до тех пор, пока положение кластера не стабилизируется. После произвольно выбранного количества итераций вычисления прекращаются, потому что иногда алгоритм не сходится.

Процесс является итерационным — это означает, что он повторяется до тех пор, пока алгоритм не достигнет конечного результата или не будет превышено максимальное количество итераций. В процессе работы алгоритм на каждой итерации непрерывно корректирует и обновляет значения центроидов для лучшей кластеризации входных данных.

В качестве источника входных данных для алгоритма кластеризации методом k -средних мы будем использовать полученные из открытых источников данные о качестве белого вина (рис. 5.4), доступные по адресу¹ <http://mng.bz/9mdt>.

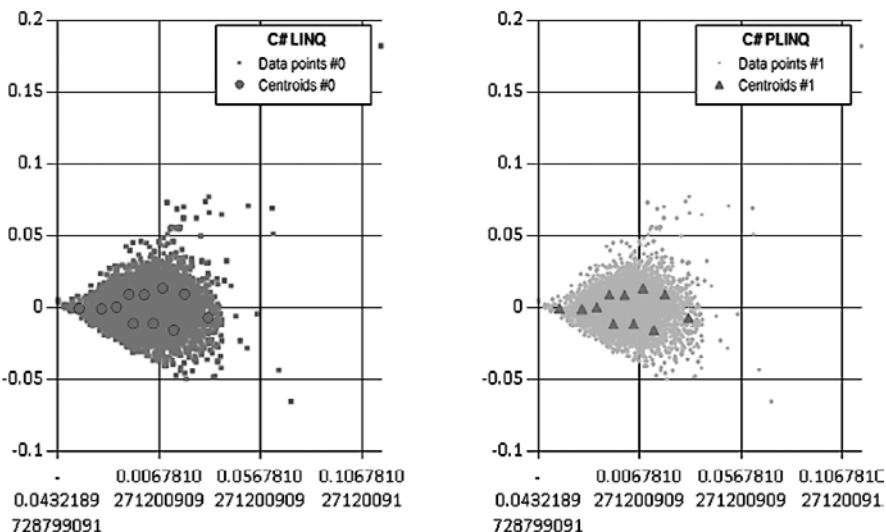


Рис. 5.4. Результат выполнения алгоритма кластеризации методом k -средних с использованием LINQ-запросов на C# для последовательной версии кода и PLINQ-запросов на C# для распараллеленной версии. Центроиды в обоих кластерах обозначены жирными точками. На каждом рисунке показана одна итерация алгоритма кластеризации методом k -средних с 11 центроидами в кластере. На каждой итерации алгоритма вычисляется центроид каждого кластера, и затем каждая точка назначается кластеру с ближайшим центроидом

Полная реализация программы кластеризации методом k -средних опущена из-за длины кода; в листингах 5.7 и 5.8 показаны только важные фрагменты кода. Однако полная реализация как на F#, так и на C# доступна для скачивания в виде исходного кода.

¹ В некоторых столбцах оригинальной таблицы данные представлены в формате даты, что неверно. Следует выбрать для них числовой тип. — Примеч. ред.

Рассмотрим две основные функции: `GetNearestCentroid` и `UpdateCentroids`. Функция `GetNearestCentroid` используется для обновления кластеров, как показано в листинге 5.7. Для каждого набора входных данных эта функция находит ближайший центроид, назначенный кластеру, к которому принадлежат входные данные (выделено жирным шрифтом).

Листинг 5.7. Поиск ближайшего центроида (обновление кластеров)

```
double[] GetNearestCentroid(double[][] centroids, double[] center){
    return centroids.Aggregate((centroid1, centroid2) => ←
        Dist(center, centroid2) < Dist(center, centroid1)
        ? centroid2
        : centroid1);
}
```

Функция LINQ Aggregate используется
для нахождения ближайшего центроида

Для сравнения расстояний между центроидами и поиска ближайшего в реализации `GetNearestCentroid` использована функция `Aggregate`. На этом этапе, если входные данные любого кластера не обновятся, потому что не будет найден более близкий центроид, алгоритм завершается и возвращает результат.

На следующем этапе, показанном в листинге 5.8 и выполняемом после обновления кластеров, положение центроидов обновляется. Функция `UpdateCentroids` вычисляет центр каждого кластера и переносит его центроид в эту точку. Затем, получив обновленные значения центроидов, алгоритм повторяет предыдущий шаг, запуская `GetNearestCentroid`, пока не найдет ближайший результат. Такие операции продолжаются до тех пор, пока не будет выполнено условие сходимости и позиции центров кластера не стабилизируются. В листинге 5.8 жирным шрифтом выделены команды, которые будут подробно рассмотрены после листинга.

В следующей реализации алгоритма кластеризации методом k -средних используются принципы ФП, последовательностные выражения с PLINQ и некоторые из множества встроенных функций для управления данными.

Функция `UpdateCentroids` требует больше вычислений, поэтому использование PLINQ-запросов позволяет эффективно распараллелить код, тем самым повысив скорость.

ПРИМЕЧАНИЕ

Даже если центроиды не перемещаются на плоскости, их индексы в результате массива могут изменяться из-за природы функций `GroupBy` и `AsParallel`.

PLINQ-запрос в теле функции `UpdateCentroids` выполняет агрегирование в два этапа. На первом этапе используется функция `GroupBy`, принимающая в качестве аргумента функцию, которая предоставляет ключ, применяемый для агрегирования. В данном случае ключ вычисляется предыдущей функцией `GetNearestCentroid`. На втором этапе выполняется отображение, которое запускает функцию `Select`, вычисляющую центры новых кластеров для каждой точки. Это вычисление вы-

полняется функцией `Aggregate`, принимающей на вход список точек (координаты положения каждого центроида), и вычисляет их центры, отображаемые на тот же кластер, с помощью локального аккумулятора `acc`, как показано в листинге 5.8.

Листинг 5.8. Обновление положения центроидов

```
double[][] UpdateCentroids(double[][] centroids)
{
    var partitioner = Partitioner.Create(data, true);
    var result = partitioner.AsParallel()
        .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
        .GroupBy(u => GetNearestCentroid(centroids, u))
        .Select(points =>
            points
                .Aggregate(
                    seed: new double[N], ←
                    func: (acc, item) => ←
                        acc.Zip(item, (a, b) => a + b).ToArray() ←
                    .Select(items => items / points.Count())
                    .ToArray());
            return result.ToArray();
}
```

The diagram shows annotations for the code:

- Настраиваемый разделитель для достижения максимальной производительности**: Points to the line `var partitioner = Partitioner.Create(data, true);`.
- Параллельный запрос от разделителя**: Points to the line `partitioner.AsParallel()`.
- Использование функции Aggregate для нахождения центра центроидов в кластере; начальное значение представляет собой массив чисел типа double размера N (размерность данных)**: Points to the `.Aggregate` call. It includes a note about the initial value being a `new double[N]` array.
- Использование функции Zip для объединения положений центроидов и последовательности аккумуляторов**: Points to the `.Zip` call within the `.Aggregate` block.

Принудительное использование параллелизма независимо от формы запроса, минуя анализ PLINQ по умолчанию, который может решить выполнять часть операций последовательно

Аккумулятор представляет собой массив чисел с удвоенной точностью, размера N , что соответствует *размерности* (количеству характеристик/измерений) данных для обработки. Значение N определяется как константа в родительском классе, поскольку оно никогда не изменяется и может быть безопасно разделено. Функция `Zip` объединяет ближайшие центроиды (точки) и последовательности аккумуляторов. Затем центр кластера повторно вычисляется путем усреднения положения точек в кластере.

Детали реализации этого алгоритма не имеют решающего значения; ключевым моментом является то, что описание алгоритма точно и непосредственно переводится в PLINQ-запросы с использованием функции `Aggregate`. Если попытаться реализовать ту же функциональность без функции `Aggregate`, то получим программу с уродливыми и запутанными циклами и с изменяемыми общими переменными.

В листинге 5.9 показан эквивалент функции `UpdateCentroids` без использования функции `Aggregate`. Фрагмент, выделенный жирным шрифтом, будет рассмотрен после листинга.

На рис. 5.5 приводятся результаты бенчмаркинга алгоритма кластеризации методом k -средних. Бенчмаркинг выполнялся на четырехъядерном процессоре с 8 Гбайт оперативной памяти. Были протестированы следующие алгоритмы: последовательный с LINQ-запросами, параллельный с PLINQ-запросами и параллельный с PLINQ-запросами и с использованием специального разделителя.

Листинг 5.9. Функция UpdateCentroids, реализованная без Aggregate

```
double[][] UpdateCentroidsWithMutableState(double[][] centroids)
{
    var result = data.AsParallel()
        .GroupBy(u => GetNearestCentroid(centroids, u))
        .Select(points => {
            var res = new double[N];
            foreach (var x in points)
                for (var i = 0; i < N; i++)
                    res[i] += x[i];
            var count = points.Count();
            for (var i = 0; i < N; i++)
                res[i] /= count;
            return res;
        });
    return result.ToArray();
}
```

Императивный цикл
для вычисления центра
центроидов в кластере

Использование
изменяемого состояния

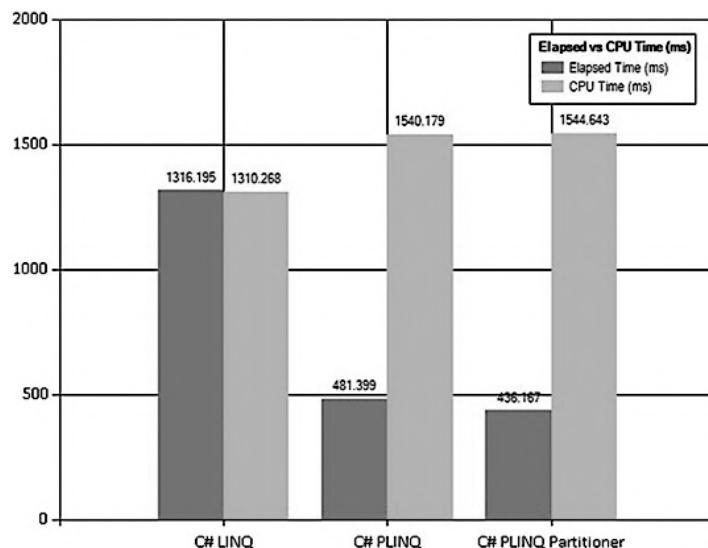


Рис. 5.5. Бенчмаркинг выполнения алгоритма кластеризации методом k-средних на четырехъядерной машине с 8 Гбайт оперативной памяти. Были протестированы следующие алгоритмы: последовательный с LINQ-запросами и параллельный с PLINQ-запросами и вариантом настраиваемого разделителя. Параллельный PLINQ выполняется за 0,481 с — в три раза быстрее, чем последовательная версия LINQ, выполняемая за 1,316 с. Небольшое улучшение дает PLINQ с настраиваемым разделителем, который выполняется за 0,436 с — на 11 % быстрее, чем исходная версия PLINQ

ПРИМЕЧАНИЕ

Если на многопроцессорном компьютере используется несколько потоков, то для выполнения задачи может задействоваться более одного процессора. В этом случае процессорное время может быть больше, чем истекшее время.

Результаты тестов впечатляют. Параллельная версия алгоритма кластеризации методом k -средних с применением PLINQ выполняется на четырехъядерной машине в три раза быстрее, чем последовательная версия. Версия PLINQ с разделителем, показанная в листинге 5.8, работает на 11 % быстрее, чем обычная версия PLINQ. В функции `UpdateCentroids` использовано интересное расширение PLINQ. Расширение `WithExecutionMode (ParallelExecutionMode.ForceParallelism)` применяется для уведомления планировщика TPL о том, что запрос должен выполняться конкурентно.

У `ParallelExecutionMode` есть два параметра настройки: `ForceParallelism` и `Default`. Перечисление `ForceParallelism` принудительно активизирует параллельное выполнение. Значение `Default` откладывает выполнение PLINQ-запроса до принятия соответствующего решения о выполнении.

В общем случае нет гарантии, что PLINQ-запрос будет выполняться параллельно. Планировщик TPL не выполняет распараллеливание каждого запроса автоматически; он может решить выполнить весь запрос или его часть последовательно, учитывая такие факторы, как размер и сложность операций и текущее состояние доступных ресурсов компьютера. Издержки, связанные с выполнением распараллеливания, могут оказаться больше, чем полученное ускорение. Но иногда возникает потребность принудительно использовать параллелизм, поскольку вы знаете о выполнении запроса больше, чем может определить PLINQ по результатам анализа. Например, вы можете знать, что делегат обходится дорого и, следовательно, запрос точно выиграет от распараллеливания.

Другим интересным расширением, используемым в функции `UpdateCentroids`, является специальный разделитель. При распараллеливании k -средних мы разделили входные данные на куски, чтобы избежать параллелизма с чрезмерно тонкой детализацией:

```
var partitioner = Partitioner.Create(data, true)
```

Класс `Partitioner<T>` — это абстрактный класс, допускающий статическое и динамическое секционирование. По умолчанию TPL `Partitioner` имеет встроенные стратегии, которые автоматически обрабатывают секционирование, обеспечивая хорошую производительность для многих источников данных. Цель TPL `Partitioner` — найти баланс между слишком большим количеством разделов (что приводит к значительным издержкам) и слишком малым количеством разделов (с недостаточным использованием доступных ресурсов). Но бывают ситуации, когда разбиение по умолчанию неприемлемо и можно получить более высокую производительность PLINQ-запроса, применяя стратегию настраиваемого секционирования.

В представленном фрагменте кода создается специальный разделитель с использованием перегруженной версии метода `Partitioner.Create`. Этот метод принимает в качестве аргумента источник данных и флаг, показывающий, какую стратегию применять — динамическую или статическую. Когда флаг равен `true`, стратегия разделителя является динамической, в противном случае — статической. Статическое секционирование часто обеспечивает ускорение на многоядерных компьютерах с небольшим количеством ядер (два или четыре). Динамическое секционирование направлено на то, чтобы сбалансировать нагрузку между задачами,

выбирая произвольный размер кусков, а затем постепенно увеличивая их длину после каждой итерации. Есть комплексные стратегии, позволяющие создавать сложные разделители (<http://mng.bz/48UP>).

Как работает секционирование

В PLINQ существует четыре вида алгоритмов разбиения.

- Секционирование по диапазонам работает с источником данных определенного размера, таким как массив:

```
int[] data = Enumerable.Range(0, 1000).ToArray();
data.AsParallel().Select(n => Compute(n));
```

- Секционирование с расслоением — противоположность секционированию по диапазонам. Размер источника данных не предопределен, поэтому PLINQ-запрос выбирает элементы по очереди и назначает их задачам до тех пор, пока источник данных не опустеет. Основное преимущество этой стратегии заключается в возможности регулирования нагрузки между задачами:

```
IEnumerable<int> data = Enumerable.Range(0, 1000);
data.AsParallel().Select(n => Compute(n));
```

- Хеш-секционирование использует код хеш-значения. Элементам с одним и тем же хеш-кодом назначается одна и та же задача (например, когда PLINQ-запрос выполняет операцию `GroupBy`).
- Блочное секционирование работает с увеличением размера блока; каждая задача извлекает из источника данных блок элементов, размер которого увеличивается на каждой следующей итерации. По мере увеличения размеров блоков задачи становятся более загруженными.

5.2.3. Реализация параллельной функции Reduce на PLINQ

Итак, теперь вы знаете о возможностях операций агрегирования, которые благодаря низкому потреблению памяти и оптимизации посредством усечения особенно эффективны для масштабируемой параллелизации на многоядерном оборудовании. Низкая пропускная способность памяти возникает из-за того, что функции агрегирования производят меньше данных, чем потребляют. Например, такие функции агрегирования, как `Sum()` и `Average()`, уменьшают набор элементов до одного значения. В этом и заключается концепция сокращения: это функция, которая сокращает последовательность элементов до одного значения. Среди списковых расширений PLINQ нет специальной функции сокращения `Reduce`, как в списковых включениях F# и других языках функционального программирования, таких как Scala и Elixir. Но, познакомившись с функцией `Aggregate`, вы сможете легко реализовать функцию многократного использования `Reduce`. В листинге 5.10 показаны два варианта реализации функции `Reduce`. Аннотированный код выделен жирным шрифтом.

Листинг 5.10. Параллельная реализация функции Reduce с использованием Aggregate

```

Эта функция определяет,
является ли число простым
На каждой итерации функция func применяется
к текущему элементу, а предыдущее значение
используется в качестве аккумулятора

static TSource Reduce<TSource>(this ParallelQuery<TSource> source,
                                  Func<TSource, TSource, TSource> reduce) =>
    ParallelEnumerable.Aggregate(source,
                                  (item1, item2) => reduce(item1, item2)); ←

static TValue Reduce<TValue>(this IEnumerable<TValue> source, TValue seed,
                               Func<TValue, TValue, TValue> reduce) =>
    source.AsParallel()
        .Aggregate(
            seed: seed,
            updateAccumulatorFunc: (local, value) => reduce(local, value), ←
            combineAccumulatorsFunc: (overall, local) =>
                reduce(overall, local), ←
            resultSelector: overall => overall); ←
    Объединение промежуточных
    результатов каждого потока
    (результатов секционирования)

int[] source = Enumerable.Range(0, 100000).ToArray();
int result = source.AsParallel()
    .Reduce((value1, value2) => value1 + value2); ←
    Использование функции
    Reduce, которой передается
    анонимная лямбда-функция
    в качестве функции
    сокращения

Возвращение конечного результата;
здесь возможна трансформация выходных данных

```

Первая функция `Reduce` принимает два аргумента: последовательность для сокращения и делегата (функцию) для применения к этой последовательности. У делегата есть два параметра: частичный результат и следующий элемент коллекции. В основной реализации используется функция `Aggregate`, в которой первый элемент исходной последовательности играет роль аккумулятора.

Второй вариант функции `Reduce` принимает дополнительный параметр `seed`, который используется в качестве начального значения для запуска сокращения с первым элементом последовательности для агрегирования. В этой версии функции объединяются результаты из нескольких потоков. Такое действие создает потенциальную зависимость как от исходной коллекции, так и от результата. По этой причине каждый поток задействует свое локальное хранилище, которое является неразделяемой памятью и применяется для кэширования частичных результатов. Когда все операции завершены, частичные результаты объединяются в конечный результат.

Функция `updateAccumulatorFunc` вычисляет частичный результат для потока. Функция `combAccumulatorsFunc` объединяет частичные результаты в конечный результат. Последний параметр `resultSelector` используется для выполнения над конечными результатами операции, определяемой пользователем. В данном случае возвращается исходное значение. Остальная часть кода является примером применения функции `Reduce` для параллельного вычисления суммы заданной последовательности.

Ассоциативность и коммутативность при детерминированном агрегировании

При агрегировании, которое выполняется параллельно с использованием PLINQ (или PSeq), функция `Reduce` применяется к данным не в таком порядке, как при последовательных вычислениях. В листинге 5.8 последовательный результат вычислялся не в той последовательности, что параллельный результат, но результаты на выходе гарантированно совпадают, потому что оператор `+` (плюс), используемый для обновления расстояний центроидов, обладает специальными свойствами ассоциативности и коммутативности. Вот строка кода, применяемая для поиска ближайшего центроида:

```
Dist(center, centroid2) < Dist(center, centroid1)
```

А это код, используемый для поиска обновлений центроидов:

```
points
    .Aggregate(
        seed: new double[N],
        func: (acc, item) => acc.Zip(item, (a, b) => a + b).ToArray())
    .Select(items => items / points.Count())
```

В функциональном программировании математические операторы — это функции. Оператор `+` (плюс) является двоичным оператором, поэтому он принимает два значения и манипулирует ими для получения результата.

Функция называется *ассоциативной*, если последовательность ее применения не влияет на результат. Это свойство важно для *операций сокращения*. Операторы `+` (плюс) и `*` (умножение) являются ассоциативными, потому что:

$$(a + b) + c = a + (b + c); \\ (a * b) * c = a * (b * c).$$

Функция называется *коммутативной*, если последовательность operandов не влияет на ее результат, при условии что все operandы учтены. Это свойство важно для *операций объединения*. Операторы `+` (плюс) и `*` (умножение) коммутативны, поскольку:

$$a + b + c = b + c + a; \\ a * b * c = b * c * a.$$

Почему это имеет значение

Используя эти свойства, можно разделять данные между несколькими потоками, каждый из которых будет работать независимо от других, со своим фрагментом данных, обеспечивая параллелизм и возвращая в конце правильный результат. Сочетание подобных свойств позволяет реализовать такие параллельные шаблоны, как «разделяй и властвуй», Fork/Join и MapReduce.

Для того чтобы параллельное агрегирование в PLINQ PSeq работало корректно, применяемая операция должна быть и ассоциативной, и коммутативной. Хорошей новостью является то, что многие из наиболее распространенных функций сокращения удовлетворяют обоим этим условиям.

5.2.4. Параллельное списковое включение в F#: PSeq

Теперь вы понимаете, что декларативное программирование приводит к распараллеливанию данных, и язык PLINQ для этого особенно удобен. PLINQ предоставляет методы расширения и функции более высокого порядка, которые можно использовать как в C#, так и в F#. Однако модуль — оболочка функциональности, предоставляемой в PLINQ для F#, делает код более идиоматичным, чем при работе непосредственно с PLINQ. Этот модуль называется PSeq, и он является параллельным эквивалентом функциональности модуля Seq для вычисления выражений. В F# модуль Seq представляет собой тонкую оболочку над классом .NET `IEnumerable<T>`, которая позволяет имитировать аналогичную функциональность. В F# все встроенные контейнеры последовательностей, такие как массивы, списки и наборы, являются подтипами типа Seq.

Итак, если параллельный LINQ-запрос хорошо подходит для применения в вашем коде, то модуль PSeq — лучший способ задействовать его в F#. В листинге 5.11 показана реализация функции `updateCentroids` с использованием PSeq на F# (выделено жирным шрифтом).

Листинг 5.11. Реализация функции `updateCentroids` на идиоматическом F# с использованием Pseq

```
let updateCentroids centroids =
    data
    |> PSeq.groupBy (nearestCentroid centroids)
    |> PSeq.map (fun (_,points) ->
        Array.init N (fun i ->
            points |> PSeq.averageBy (fun x -> x.[i])))
    |> PSeq.sort
    |> PSeq.toArray
```

В этом коде используется конвейерный оператор `|>` F#, позволяющий строить конвейерную семантику для вычисления последовательности операций в виде цепочки выражений. Применяемые с помощью функций `PSeq.groupBy` и `PSeq.map` операции более высокого порядка следуют тому же шаблону, что и исходная функция `updateCentroids`. Функция `map` эквивалентна `Select` в PLINQ.

Функция агрегирования `PSeq.averageBy` полезна, поскольку заменяет шаблонный код (необходимый в PLINQ), который не имеет такой встроенной функциональности.

5.2.5. Параллельные массивы в F#

Модуль `PSeq` содержит множество знакомых и полезных функциональных конструкций, таких как `map` и `reduce`, но подобные функции по своей сути ограничены тем фактом, что вынуждены воздействовать на последовательности, а не на разделяемые диапазоны. Поэтому функции, предоставляемые модулем `Array.Parallel` из стандартной библиотеки F#, обычно масштабируются намного эффективнее при увеличении количества ядер в машине (листинг 5.12).

Листинг 5.12. Параллельное суммирование простых чисел с использованием F# `Array.Parallel`

```
let len = 10000000
let isPrime n =
    if n = 1 then false
    elif n = 2 then true
    else
        let boundary = int (Math.Floor(Math.Sqrt(float(n))))
        [2..boundary - 1] > Seq.forall(fun i -> n % i <> 0)

let primeSum =
    [|0.. len|]
    |> Array.Parallel.filter (fun x-> isPrime x) ← Эта функция определяет,
                                                является ли число простым
    |> Array.sum                                Здесь используется встроенный
                                                модуль параллельного массива F#;
                                                функция filter разработана
                                                на базе функции Array.Parallel.choose.
                                                Подробнее см. в исходном коде,
                                                выложенном онлайн
```

Модуль `Array.Parallel` содержит версии многих обычных функций высшего порядка для обработки массивов, распараллеленных с помощью класса `Parallel` из библиотеки TPL. Эти функции, как правило, гораздо эффективнее, чем их эквиваленты из `PSeq`, поскольку они работают не с линейными последовательностями, а со смежными диапазонами массивов, разделенных на куски. Модуль `Array.Parallel`, входящий в состав стандартной библиотеки F#, включает в себя параллельные версии нескольких полезных операторов агрегирования, в первую очередь `map`. Функция `filter` разработана с использованием функции `Array.Parallel.choose` (см. исходный код к книге).

Стратегии распараллеливания данных: сводная таблица. Мы рассмотрели основные шаблоны проектирования программного обеспечения, сформировавшиеся в функциональном программировании и используемые для быстрой параллельной обработки данных. Краткая информация об этих шаблонах сведена в табл. 5.1.

Таблица 5.1. Шаблоны параллельной обработки данных, рассмотренные ранее в этой книге

Шаблон	Определение	Достоинства и недостатки
«Разделяй и властвуй»	Рекурсивно разбивает проблему на более мелкие проблемы до тех пор, пока они не станут достаточно мелкими, чтобы решить их напрямую. Для всех рекурсивных вызовов создаются независимые задачи, которые выполняются параллельно. Самым популярным примером алгоритма «разделяй и властвуй» является сортировка Quicksort	Если рекурсивных вызовов слишком много, этот шаблон может создавать дополнительные издержки, связанные с тем, что параллельная обработка перегружает процессоры

Шаблон	Определение	Достоинства и недостатки
Fork/Join	<p>Этот шаблон предназначен для разделения (fork) исходного набора данных на куски с целью параллельной обработки. После завершения работы всех параллельных частей эти куски объединяются (join).</p> <p>Параллельная часть алгоритма может быть реализована с использованием рекурсии, как в шаблоне «разделяй и властвуй», до достижения определенной степени детализации задачи</p>	Обеспечивает эффективное распределение нагрузки
Aggregate/Reduce	<p>Цель этого шаблона – объединить все элементы исходного набора данных в одно значение путем независимого параллельного выполнения задач с отдельными элементами.</p> <p>Это первый уровень оптимизации, который следует учитывать при распараллеливании циклов с разделяемым состоянием</p>	Элементы набора данных, который следует сократить в результате параллельной обработки, должны удовлетворять свойству ассоциативности. Используя ассоциативный оператор, можно объединить любые два элемента набора данных в один элемент

Абстракции параллельного программирования, представленные в табл. 5.1, могут быть быстро реализованы с использованием функций многоядерной разработки, доступных в .NET. В остальной части этой книги будут проанализированы и другие шаблоны. В следующем разделе мы рассмотрим параллельный шаблон MapReduce.

5.3. Параллельный шаблон MapReduce

MapReduce – это шаблон, впервые описанный в 2004 г. в статье «MapReduce: упрощенная обработка данных в крупных кластерах» Джейфри Дина (Jeffrey Dean) и Санджея Гемавата (Sanjay Ghemawat) (<https://research.google.com/archive/mapreduce-osdi04.pdf>).

Особенно интересные решения MapReduce предлагает для анализа больших данных; посредством параллельной обработки этот шаблон позволяет перемалывать огромные объемы данных. Он чрезвычайно хорошо масштабируется и используется в ряде крупнейших распределенных приложений по всему миру. Кроме того, он предназначен для обработки и генерации больших наборов данных, распределемых между несколькими компьютерами. В Google указанный шаблон реализован на большом кластере машин и может обрабатывать несколько терабайт данных одновременно. Структура и принципы этого шаблона применимы как к одной машине (одноядерной) в меньшем масштабе, так и к мощным многоядерным машинам.

В этой главе основное внимание уделяется распараллеливанию данных на одном многоядерном компьютере, но те же концепции могут быть применены и для распределения работы между несколькими компьютерами, объединенными в сеть.

В главах 11 и 12 будет рассмотрена модель агентного программирования (и модель актора), которая может быть использована для такого сетевого распределения задач.

Идея модели MapReduce (как показано на рис. 5.6) основана на функциональной парадигме и, как следует из ее названия, происходит от концепций, известных как комбинаторы *map* (карта) и *reduce* (сокращение). Программы, написанные в этом более функциональном стиле, могут быть распараллелены на большом кластере машин без специальных знаний о конкурентном программировании. Фактическая среда выполнения может затем распределить данные, спланировать выполнение и организовать обработку любых потенциальных сбоев.

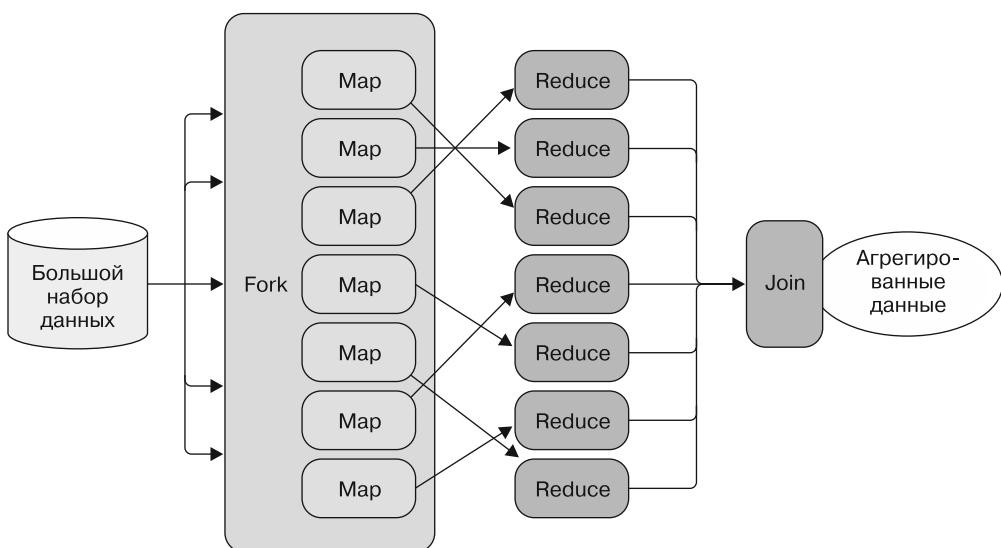


Рис. 5.6. Схематическая иллюстрация этапов вычисления по шаблону MapReduce. Шаблон MapReduce состоит из двух основных операций: отображения (map) и сокращения (reduce).

Функция Map применяется ко всем элементам и генерирует промежуточные результаты, которые объединяются с помощью функции Reduce. Этот шаблон похож на Fork/Join, потому что после разделения данных на блоки в нем параллельно выполняются задачи отображения и сокращения — для каждого блока в отдельности, независимо от других. На рисунке исходный набор данных разбивается на блоки, которые благодаря отсутствию зависимостей могут обрабатываться по отдельности. Затем с помощью функции Map каждый блок преобразуется в другую форму. Все функции Map выполняются одновременно. После завершения каждой операции отображения блока результат передается на следующий этап, на котором выполняется агрегирование с помощью функции Reduce. (Агрегирование можно сравнить со стадией объединения в шаблоне Fork/Join)

Модель MapReduce полезна в тех областях, где необходимо выполнить огромное количество параллельных операций. Примерами областей, в которых широко используется MapReduce, являются машинное обучение, обработка изображений, интеллектуальный анализ данных, распределенная сортировка.

В целом эта модель программирования основана на следующих пяти простых принципах. Их последовательность не обязательна и может быть изменена в зависимости от потребностей.

1. Итерирование входных данных.
2. Вычисление пар «ключ/значение» для каждого набора входных данных.
3. Группировка всех промежуточных значений по ключу.
4. Итерирование полученных групп.
5. Сокращение каждой группы.

Общая идея MapReduce заключается в использовании сочетания отображений и сокращений для запроса потока данных. Для этого можно отобразить доступные данные в другой формат, создавая новый элемент данных в другом формате для каждого исходного элемента данных. Во время выполнения операции `Map` также можно изменить последовательность элементов, как до, так и после отображения. Операции `Map` сохраняют прежнее количество элементов. Если элементов много, то, чтобы ответить на запрос, их количество можно уменьшить. Можно отфильтровать входной поток, отбрасывая ненужные элементы.

Шаблоны MapReduce и GroupBy

Функция `Reduce` сокращает входной поток до одного значения. Иногда вместо получения одного значения требуется сократить большое количество входных элементов, группируя их в соответствии с неким условием. При этом фактически количество элементов не уменьшается, они только группируются, но затем можно сократить каждую группу, агрегировав ее в одно значение. Например, можно вычислить сумму значений в каждой группе, если группа содержит элементы, которые можно суммировать.

Можно объединить элементы в один агрегированный элемент и вернуть только те, которые дают искомый ответ. Отображение перед сокращением — один из способов сделать это, но также можно выполнить операцию `Reduce` и перед `Map` или даже выполнить `Reduce`, `Map`, затем снова `Reduce` и т. д. Таким образом, шаблон MapReduce выполняет отображение (преобразование данных из одного формата в другой и упорядочение данных) и сокращение (фильтрацию, группировку или агрегирование) данных.

5.3.1. Функции `Map` и `Reduce`

Шаблон MapReduce состоит из следующих двух основных этапов.

- ❑ Функция `Map` получает входные данные и выполняет их отображение, так что на ее выходе образуются промежуточные результаты в формате пар «ключ/значение». Значения с одним и тем же ключом затем объединяются и передаются на второй этап.

- Функция `Reduce` агрегирует результаты, полученные от `Map`, применяя функцию к значениям, связанным с одним и тем же промежуточным ключом, и создавая, как правило, меньший набор значений.

Важным аспектом `MapReduce` является то, что результат, полученный на этапе `Map`, должен быть совместим с форматом входных данных этапа `Reduce`. Это свойство приводит к функциональной композиционности.

5.3.2. Использование `MapReduce` совместно с галереей пакетов NuGet

В данном разделе вы научитесь реализовывать и применять шаблон `MapReduce` с использованием программы загрузки и анализа пакетов из онлайн-галереи NuGet. NuGet – это система управления пакетами, предназначенная для платформы разработки Microsoft, включая .NET, а галерея NuGet – центральный репозиторий пакетов, применяемый всеми разработчиками пакетов. На момент написания этой книги существовало более 800 000 пакетов NuGet. Цель нашей программы – упорядочить пакеты по степени важности и выделить пять наиболее важных пакетов NuGet. Важность пакета будем определять как сумму рейтинга пакета и рейтингов всех его зависимостей.

Вследствие внутренней связи между `MapReduce` и ФП код из листинга 5.13 будет реализован на F# с использованием `PSeq` для поддержки распараллеливания данных. Версию кода на C# вы найдете в исходном коде к книге.

Эту же базовую идею можно применять для поиска другой информации, такой как зависимости используемого пакета, зависимости их зависимостей и т. д.

ПРИМЕЧАНИЕ

Загрузка всей информации о версиях всех пакетов NuGet занимает некоторое время. В исходном коде к книге есть файл с архивом (`nuget-latestversions.model`), размещенный в подпапке `Models`. Если вы захотите обновить его до последних версий, удалите этот файл, запустите приложение и запаситесь терпением. Будет создан новый архив и сохранен до следующего использования.

В листинге 5.13 определены функции `Map` и `Reduce`. Функция `Map` преобразует входные данные о пакетах NuGet в структуру данных, состоящую из пар «ключ/значение», где ключ – имя пакета, а значение – его рейтинг (число в формате `float`). Эта структура данных определена как последовательность типов «ключ/значение», поскольку у каждого пакета могут быть зависимости, рейтинг которых будет вычисляться как часть общего рейтинга. Функция `Reduce` принимает в качестве аргумента имя пакета и последовательность связанных с ним пар «рейтинг/значение». Этот набор входных данных получен на выходе предыдущей функции `Map`.

Листинг 5.13. Объект PageRank, инкапсулирующий функции Map и Reduce

```

type PageRank (ranks:seq<string*float>) =
    let mapCache = Map.ofSeq ranks ← Внутренняя таблица для хранения
                                в памяти коллекции NuGet,
                                состоящей из пар «имя/рейтинг»
    let getRank (package:string) =
        match mapCache.TryFind package with ← Если пакет NuGet не найден,
        | Some(rank) -> rank
        | None -> 1.0
                                то по умолчанию его рейтинг равен 1.0
    member this.Map (package:NuGet.NuGetPackageCache) =
        let score =
            (getRank package.PackageName)
            /float(package.Dependencies.Length) ← Функция для вычисления среднего
                                рейтинга зависимостей NuGet
        package.Dependencies ←
        |> Seq.map (fun (Domain.PackageName(name, _), _, _) -> (name, score))
    member this.Reduce (name:string) (values:seq<float>) =
        |> (name, Seq.sum values)

```

**Функция, которая сводит к одному значению
все рейтинги, относящиеся к данному пакету**

Объект PageRank инкапсулирует функции Map и Reduce, обеспечивая легкий доступ к одним и тем же базовым рейтингам структуры данных. Затем нужно построить ядро программы MapReduce. Используя стиль ФП, можно смоделировать многократно применяемую функцию MapReduce, передав ей на вход функции, задействуемые на этапах Map и Reduce. В листинге 5.14 представлена реализация функции mapF.

Листинг 5.14. Функция mapF для первого этапа шаблона MapReduce

Устанавливает порог распараллеливания
в виде произвольного значения M

```

let mapF M (map:'in_value -> seq<'out_key * 'out_value>)
           (inputs:seq<'in_value>) = ← Принудительный порог
                                распараллеливания
    inputs
    |> PSeq.withExecutionMode ParallelExecutionMode.ForceParallelism ←

```

|> PSeq.withDegreeOfParallelism M | Отображение элементов
входной коллекции

|> PSeq.collect (map) | Группировка отображенных элементов по ключу,
генерированному функцией отображения

|> PSeq.groupBy (fst)

|> PSeq.toList | Материализация последовательности,
чтобы не превысить порог распараллеливания

Функция `mapF` принимает в качестве первого параметра целое число `M`, которое определяет уровень распараллеливания. Этот аргумент намеренно стоит первым, потому что он упрощает частичное применение функции при ее многократном использовании с одним и тем же входным значением. Внутри тела `mapF` уровень параллелизма задается с помощью `PSeq.withDegreeOfParallelism M`. Данный метод расширения также применяется в PLINQ. Назначение такой конфигурации — ограничить количество потоков, которые могут выполняться параллельно, и не случайно немедленно материализованный запрос реализует последнюю функцию `PSeq.toList`. Если опустить `PSeq.withDegreeOfParallelism`, то заданный уровень параллелизма не будет гарантирован.

В случае одиночной многоядерной машины иногда бывает полезно ограничить количество работающих потоков для каждой функции. Поскольку в параллельном шаблоне MapReduce этапы `Map` и `Reduce` выполняются одновременно, может быть полезно ограничить ресурсы, выделяемые для каждого из данных этапов. Например, можно определить значение `maxThreads` как:

```
let maxThreads = max (Environment.ProcessorCount / 2, 1)
```

и использовать его для того, чтобы на выполнение каждого из двух этапов MapReduce выделялось не более половины системных ресурсов.

Вторым аргументом функции `mapF` является основная функция `map`, которая работает с каждым входным значением и возвращает последовательность пар «ключ/значение». Тип выходной последовательности может отличаться от типа входных данных. Последним аргументом будет последовательность входных значений для обработки.

После функции `map` надо реализовать агрегирование `reduce`. В листинге 5.15 показана реализация функции агрегирования `reduceF`, выполняющая второй этап шаблона и выдающая итоговый результат.

Листинг 5.15. Функция `reduceF` для реализации второго этапа MapReduce

```
let reduceF R (reduce:'key -> seq<'value> -> 'reducedValues)
    (inputs:('key * seq<'key * 'value>) seq) =
    |> PSeq.withExecutionMode ParallelExecutionMode.ForceParallelism
    |> PSeq.withDegreeOfParallelism R
    |> PSeq.map (fun (key, items) ->
        |> Seq.map (snd)
        |> reduce key)
    |> PSeq.toList
```

Первый аргумент `R` функции `reduceF` имеет ту же цель, что и аргумент `M` в предыдущей функции `mapF` — установить уровень параллелизма. Второй аргумент — это

функция `reduce`, которая обрабатывает каждую пару «ключ/значение» входного набора данных. В примере с пакетом NuGet ключ представляет собой строку, содержащую имя пакета, а последовательность значений — список рейтингов, связанных с этим пакетом. Последний входной аргумент представляет собой набор пар «ключ/значение», полученный на выходе функции `mapF`. Функция `reduceF` генерирует конечный результат.

После того как функции `map` и `reduce` определены, остается самое простое: собрать все вместе (выделено жирным шрифтом) (листинг 5.16).

Листинг 5.16. Функция `mapReduce`, скомпонованная из функций `mapF` и `reduceF`

```
let mapReduce
    (inputs:seq<'in_value>)
    (map:'in_value -> seq<'out_key * 'out_value>)
    (reduce:'out_key -> seq<'out_value> -> 'reducedValues)
    M R =
  inputs |> (mapF M map >> reduceF R reduce)
```

Функции `map` и `reduce` созданы
 с использованием оператора
 предварительной компоновки
 функций `>> F#`

Поскольку результат функции `map` подается на вход функции `reduce`, эти функции можно легко скомпоновать. Такой функциональный подход показан в листинге 5.16 при реализации функции `mapReduce`. Аргументами функции `mapReduce` выступают базовые функции `mapF` и `reduceF` — по той же причине. Важной частью этого кода является последняя строка. Используя встроенный конвейерный оператор F# (`|>`) и оператор предварительной компоновки функций (`>>`), можно собрать все блоки воедино.

В следующем коде показано, как теперь можно использовать функцию `mapReduce` из листинга 5.16 для расчета рейтинга пакета NuGet:

```
let executeMapReduce (ranks:(string*float)seq) =
let M,R = 10,5
let data = Data.loadPackages()
let pg = MapReduce.Task.PageRank(ranks)
mapReduce data (pg.Map) (pg.Reduce) M R
```

Класс `pg` (`PageRank`) определен в листинге 5.13. Он обеспечивает реализацию функций `map` и `reduce`. Произвольные значения `M` и `R` соответствуют количеству обработчиков, создаваемых на каждом этапе MapReduce. После реализации функций `mapF` и `reduceF` выполняется их компоновка, чтобы реализовать функцию `mapReduce`, которую удобно использовать в качестве новой функции.

Как и ожидалось, последовательная реализация (рис. 5.7) является самой медленной. Поскольку параллельные версии инструкций `PSeq` в F# и PLINQ в C# за-действуют одну и ту же базовую библиотеку, их скорости почти одинаковы. Версия `PSeq` из-за дополнительных накладных расходов, вызванных оболочкой, работает немного медленнее и имеет более высокое процессорное время. Самый быстрый вариант MapReduce — параллельная версия с PLINQ-запросами и настраиваемым разделителем, которую можно найти в исходном коде к книге.

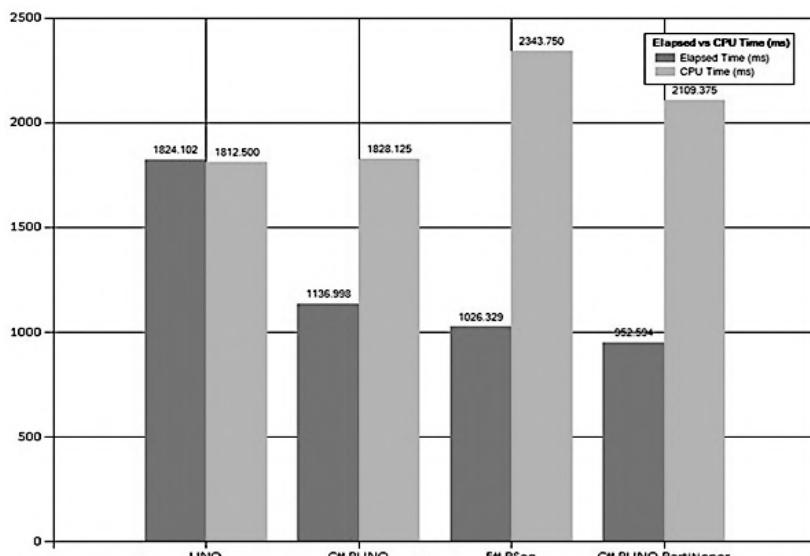


Рис. 5.7. Бенчмаркинг алгоритма MapReduce на четырехъядерном компьютере с 8 Гбайт оперативной памяти. Протестированы следующие алгоритмы: последовательный с LINQ-запросами, параллельный с инструкциями PSeq в F# и с PLINQ-запросами с вариантом настраиваемого разделителя. Параллельная версия MapReduce с использованием PLINQ-запросов выполняется за 1,136 с, что на 38 % быстрее, чем последовательная версия с использованием обычных LINQ-запросов на C#. Производительность инструкций PSeq, как и ожидалось, почти эквивалентна PLINQ, поскольку в их основе лежит одна и та же технология. Самое быстрое решение — параллельный алгоритм на C# с применением PLINQ и настраиваемым разделителем. Он выполнился за 0,952 с, что примерно на 18 % быстрее, чем обычный PLINQ, и в два раза быстрее опорного варианта (последовательной версии)

Ниже представлен результат выполнения для пяти наиболее важных пакетов NuGet.

```
Microsoft.NETCore.Platforms :      6033.799540
Microsoft.NETCore.Targets :        5887.339802
System.Runtime :                  5661.039574
Newtonsoft.Json :                 4009.295644
NETStandard.Library :             1876.720832
```

В MapReduce все формы сокращения, выполняемые параллельно, могут давать результаты, отличные от последовательной версии, если эта операция не является ассоциативной.

MapReduce и немного математики

Свойства ассоциативности и коммутативности, представленные ранее в этой главе, гарантируют корректность и детерминированное поведение функций агрегирования. При параллельном функциональном программировании применение математиче-

ских моделей является общим правилом для обеспечения точности при реализации программы. Но для этого все не требуется глубокое знание математики.

Можете ли вы определить значение x в следующих уравнениях?

$$9 + x = 12.$$

$$2 < x < 4.$$

Вы дали ответ 3 для обеих функций? Отлично! Поздравляю: ваших познаний в математике достаточно для написания детерминированных конкурентных программ в функциональном стиле с использованием методов линейной алгебры (https://ru.wikipedia.org/wiki/Линейная_алгебра).

Математика для упрощения параллелизма: моноиды

Свойство ассоциативности приводит к стандартному способу, известному как моноид (<https://wiki.haskell.org/Monoid>), который легко совместить со многими типами значений. Термин «*монаид*» (не путать с монадой: <https://wiki.haskell.org/Monad>) — математический, но для того, чтобы применить эту концепцию в компьютерном программировании, особых познаний в математике не требуется. По сути, моноиды — операции, у которых типы входных и выходных данных совпадают и которые удовлетворяют правилам ассоциативности, нейтральности и замыкания.

Об ассоциативности говорилось в предыдущем разделе. Свойство *нейтральности* требует, чтобы при многократном вычислении результат не изменялся. Например, агрегирование, являющееся ассоциативным и коммутативным, может выполняться на одном или нескольких этапах сокращения для получения конечного результата и это не влияет на тип результата. Правило *замыкания* требует, чтобы типы входных и выходных данных функции были одинаковыми. Например, операция сложения принимает в качестве входных параметров два числа и возвращает результат также в виде числа. Такое правило может быть выражено в .NET посредством сигнатуры функции `Func<T, T, T>`, гарантирующей, что все аргументы принадлежат к одному и тому же типу, в отличие от сигнатуры функции `Func<T1, T2, R>`, которая такой гарантии не дает.

В примере с k -средним функция `UpdateCentroids` удовлетворяет данным законам, поскольку операции, используемые в алгоритме, являются моноидальными — страшное слово, за которым скрывается простая концепция. Эта операция — сложение (для сокращения).

Функция сложения принимает два числа и выдает результат того же типа. В таком случае нейтральный элемент равен 0 (нулю), поскольку значение 0 может быть добавлено к результату операции без его изменения. Умножение также является моноидом с нейтральным элементом 1 : значение числа, умноженное на 1 , не изменяется.

Почему важно, чтобы операция возвращала результат того же типа, что и входные данные? Потому что это позволяет связывать и компоновать несколько объектов с помощью моноидальной операции, упрощая введение параллелизма для этих операций.

Например, если операция ассоциативна, то можно свернуть структуру данных и, соответственно, сократить список. Но если мы имеем дело с моноидом, то можем

сократить список, используя свертку (функция `Aggregate`), которая будет более эффективной для ряда операций, а также допускает параллелизм.

Для вычисления факториала числа 8 операции умножения, выполняемые параллельно на двухъядерном процессоре, выглядят примерно так, как показано в табл. 5.2.

Таблица 5.2. Параллельное вычисление факториала числа 8

	Ядро 1	Ядро 2
Шаг 1	$M1 = 1 * 2$	$M2 = 3 * 4$
Шаг 2	$M3 = M2 * 5$	$M4 = 6 * M1$
Шаг 3	$M5 = M4 * 7$	$M6 = 8 * M3$
Шаг 4	Простой	$M7 = M6 * M5$
Результат	40320	

Тот же результат можно получить посредством параллельного агрегирования на F# или C# и уменьшить список чисел от 1 до 8 до одного значения:

```
[1..8] |> PSeq.reduce (*)
Enumerable.Range(1,8).AsParallel().Reduce((a,b)=> a * b);
```

Поскольку умножение является моноидальной операцией для типа `integer`, мы можем быть уверены, что результат выполнения параллельной операции будет детерминированным.

ПРИМЕЧАНИЕ

При использовании параллелизма задействованы многие факторы, поэтому важно постоянно проводить бенчмаркинг и измерять ускорение алгоритма, применяя последовательную версию в качестве опорной. Фактически в некоторых случаях параллельный цикл может работать медленнее, чем его последовательный эквивалент. Если последовательность слишком мала для распараллеливания, то дополнительные издержки, введенные для координации задач, могут иметь негативные последствия. Для таких сценариев лучше подходит последовательный цикл.

Резюме

- ❑ Параллельные алгоритмы с использованием LINQ-запросов и инструкций `PSeq` в F# основаны на функциональной парадигме и предназначены для распараллеливания данных, упрощения кода и повышения производительности. По умолчанию уровнем параллелизма для этих технологий является логический процессор. Данные технологии обрабатывают основные процессы, связанные с разделением последовательностей на фрагменты меньшего размера, устанавливают степень параллелизма в зависимости от количества логических ядер машины и обрабатывают каждую подпоследовательность в отдельности.
- ❑ PLINQ и `PSeq` в F# – это технологии абстракции более высокого порядка, которые являются надстройкой над многопоточными компонентами. Цель этих технологий – сокращение времени выполнения запроса посредством максимального использования доступных компьютерных ресурсов.
- ❑ .NET Framework позволяет использовать настраиваемые технологии, чтобы получить максимальную производительность при анализе данных. Имеет смысл применять типы-значения вместо ссылочных типов, чтобы избежать возможных проблем с памятью, которые иначе могут создать узкое место из-за слишком частой сборки мусора.
- ❑ Написание чистых функций или функций без побочных эффектов упрощает проверку корректности программы. Кроме того, поскольку чистые функции являются детерминированными, то при передаче им одних и тех же исходных данных результат не изменяется. Последовательность выполнения не имеет значения, поэтому функции без побочных эффектов могут легко выполняться параллельно.
- ❑ Проектирование с использованием чистых функций и отделение побочных эффектов от чистой логики – два основных принципа, которые выходят на первый план при функциональном мышлении.
- ❑ Усечение – это технология, позволяющая избежать создания промежуточных структур данных, сократить объем выделяемой временной памяти и таким образом повысить производительность приложения. Такую технологию можно легко сочетать с функцией `Aggregate` более высокого порядка в LINQ. Она объединяет несколько операций в одну – например, `filter` и `map`, которые в противном случае потребовали бы выделения памяти для каждой операции.
- ❑ Функции записи, являющиеся ассоциативными и коммутативными, допускают реализацию параллельных шаблонов, таких как «разделяй и властвуй», `Fork/Join` или `MapReduce`.
- ❑ Шаблон `MapReduce` состоит из двух основных этапов: отображения (`Map`) и сокращения (`Reduce`). Функция `Map` применяется ко всем элементам и производит промежуточные результаты, которые объединяются с помощью функции `Reduce`. Такой шаблон похож на `Fork/Join`, поскольку после разделения данных на фрагменты функции `Map` и `Reduce` применяются ко всем задачам независимо и параллельно.

Потоки событий реального времени: функциональное реактивное программирование

В этой главе:

- что такое потоки событий, доступные для запроса;
- работа с реактивными расширениями (Rx);
- как сделать события значениями первого класса, сочетая F# и C#;
- обработка высокоскоростных потоков данных;
- реализация шаблона «издатель — подписчик».

Мы привыкли реагировать на события, ежедневно происходящие в нашей жизни. Пошел дождь — мы берем с собой зонтик. Стемнело — мы щелкаем выключателем, чтобы включить свет. То же самое касается и наших приложений, когда программа должна реагировать на события, вызванные действиями пользователя или чем-то, что произошло в приложении (другими словами — обрабатывать события). Почти каждая программа должна обрабатывать события, будь то получение HTTP-запроса для веб-страницы на сервере, уведомление от вашей любимой социальной сети, изменения в файловой системе или просто нажатие кнопки.

Сегодня перед приложениями стоит задача: реагировать не просто на отдельные события, а на постоянный большой поток событий в (почти) реальном времени. Рассмотрим обычный смартфон. Мы зависим от этого устройства: оно должно быть постоянно подключено к Интернету, постоянно отправлять и получать данные. Подобный обмен данными между множеством устройств можно сравнить с рабо-

той миллиардов датчиков, которые получают и передают информацию и при этом должны анализировать ее в реальном времени. Кроме того, такой непрерывный массивный поток уведомлений из Интернета продолжает быть, как струя из пожарного шланга, и конструкция системы должна не только обрабатывать уведомления, но и выдерживать обратное давление (https://en.wikipedia.org/wiki/Back_pressure).

Обратное давление означает ситуацию, когда поставщик событий слишком сильно опережает потребителя. Это может привести к резким скачкам потребления памяти и, возможно, к выделению большего количества системных ресурсов для потребителя до тех пор, пока он не обработает все запросы. Подробнее об обратном давлении читайте далее в настоящей главе.

Согласно прогнозам, в 2020 г. к Интернету будет подключено более 50 млрд устройств. Более того, в ближайшее время поток цифровой информации не собирается замедляться, а будет только расширяться! По этой причине способность управлять высокоскоростными потоками данных и анализировать их в реальном времени по-прежнему будет иметь первоочередное значение в области анализа данных (больших данных) и цифровой информации.

Существует множество проблем, связанных с использованием традиционной парадигмы программирования при реализации таких систем обработки данных в реальном времени. Какие технологии и инструменты можно применять, чтобы упростить модель программирования событий? Как конкурентно управлять несколькими событиями, не обладая конкурентным мышлением? Ответы на эти вопросы лежат в области реактивного программирования.

В вычислениях реактивное программирование — это парадигма программирования, которая поддерживает непрерывное взаимодействие со своей средой, но скорость данного взаимодействия определяется средой, а не программой.

Жерар Берри (Gérard Berry). Программирование в реальном времени: языки специального и общего назначения (Inria, 1989), <http://mng.bz;br08>

Говоря простыми словами, *реактивное программирование* — это программирование с постоянными асинхронными потоками событий. Кроме того, оно сочетает в себе описанные в предыдущих главах преимущества функционального программирования для обеспечения конкурентности с инструментарием реактивного программирования, позволяющим сделать событийно-управляемое программирование очень полезным, доступным и безопасным. Более того, применяя к потокам разнообразные операторы более высокого порядка, можно легко достичь различных вычислительных целей.

Дочитав эту главу до конца, вы узнаете, как реактивное программирование позволяет избежать проблем, возникающих при использовании императивных методов построения реактивных систем. Вы научитесь разрабатывать и внедрять событийно-управляемые приложения с поддержкой асинхронности — отзывчивые, масштабируемые и слабосвязанные.

6.1. Реактивное программирование: обработка больших событий

Реактивное программирование (не путайте с функциональным реактивным программированием) — это парадигма программирования, которая фокусируется на асинхронном приеме и асинхронной обработке событий как потока данных, где не поток вычислений управляет потоком данных, а поступление новой информации приводит в движение логику.

Типичным примером реактивного программирования является электронная таблица, ячейки которой содержат лiteralные значения или формулы, такие как $C1 = A1 + B1$, или, на языке Excel, $C1 = \text{Sum}(A1:B1)$. В данном случае значение в ячейке $C1$ вычисляется на основании значений других ячеек. Когда значение в одной из этих ячеек, $B1$ или $A1$, изменяется, значение формулы автоматически пересчитывается и ячейка $C1$ обновляется, как показано на рис. 6.1.



Рис. 6.1. Электронная таблица Excel является реактивной: ячейка C1 реагирует на изменение значений в ячейках A1 и B1 по формуле $\text{Sum}(A1:B1)$

Тот же принцип применим и при обработке данных для уведомления системы об изменении состояния. Анализ коллекций данных является обычным требованием при разработке программного обеспечения. Во многих случаях код выигрывает от использования реактивного обработчика событий. Реактивный обработчик событий, в отличие от обычного обработчика с ограниченной гибкостью, рассчитанного на простые сценарии, позволяет задействовать композиционную реактивную семантику для описания таких операций, как *Filter* и *Map*, что дает возможность элегантно и лаконично описывать обработку событий.

Метод реактивного программирования для обработки событий отличается от традиционного подхода тем, что события рассматриваются в нем как потоки. Это дает возможность легко и выразительно манипулировать событиями в декларативном стиле, с различными свойствами событий, такими как возможность фильтрации, отображения и слияния. Например, можно создать веб-сервис, который бы филь-

тровал поток событий, выделяя их подмножество на основе определенных правил. В полученном решении реактивное программирование используется для распознавания предполагаемого поведения путем описания операций в декларативном стиле, что является одним из принципов ФП. Это одна из причин, почему данный стиль обычно называют функциональным реактивным программированием; но указанный термин требует дальнейшего пояснения.

Что такое *функциональное реактивное программирование* (ФРП)? Технически ФРП представляет собой парадигму программирования, основанную на *значениях, изменяющихся во времени*, с использованием набора простых композиционных реактивных операторов (*behavior* и *event*), которые, в свою очередь, позволяют строить более сложные операторы. Эта парадигма программирования обычно применяется для разработки пользовательских интерфейсов, в робототехнике и играх, а также для разрешения проблем в распределенных и сетевых системах. Благодаря мощному упрощенному композиционному аспекту ФРП принципы ФРП используются в ряде современных технологий для разработки сложных систем. Например, ФРП лежит в основе языков программирования Elm (<http://elm-lang.org>) и Yampa (<https://wiki.haskell.org/Yampa>).

С точки зрения индустрии ФРП представляет собой набор различных, но взаимосвязанных технологий функционального программирования, объединенных общей целью обработки событий. Но из-за того, что одни и те же слова используются в разных сочетаниях, возникает путаница вследствие сходства и неправильного представления терминов.

- ❑ Функциональное программирование — это парадигма, которая рассматривает вычисление как определение значения выражения и позволяет избежать изменяющихся состояний и изменяемых данных.
- ❑ Реактивное программирование — это парадигма реализации приложений с компонентом реального времени.

Реактивное программирование становится все более важным в контексте обработки потоков в реальном времени для анализа больших данных. Преимуществами реактивного программирования являются более эффективное использование вычислительных ресурсов на многоядерном и многопроцессорном оборудовании, что обеспечивает простой и удобный способ организации асинхронных вычислений, неблокирующих вычислений и ввода-вывода без блокировок. Аналогично ФРП предлагает корректную абстракцию, дающую возможность создавать очень выгодные, доступные, безопасные и компонуемые события-управляемые программы. Эти аспекты позволяют разрабатывать реактивные программы реального времени с чистым и понятным кодом, которые легко поддерживать и расширять без ущерба для производительности.

Концепция реактивного программирования — это неблокирующий асинхронный код, возвращающий управление от «запроса» к «ожиданию» событий, как показано на рис. 6.2. Данный принцип называется *инверсией управления* (<http://martinfowler.com/bliki/InversionOfControl.html>) и также известен как «принцип Голливуда» («не звоните нам, мы сами позвоним»).

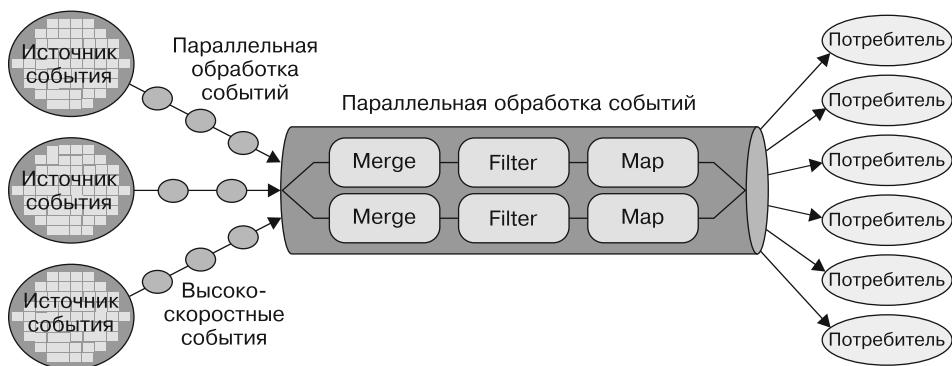


Рис. 6.2. Реактивное программирование в режиме реального времени означает построение неблокирующих (асинхронных) операций для работы с большими объемами высокоскоростных последовательностей событий в реальном времени с одновременной, возможно, параллельной обработкой нескольких событий

Реактивное программирование направлено на обработку высокоскоростных последовательностей событий в реальном времени с упрощением конкурентного аспекта одновременной (параллельной) обработки нескольких событий.

Сейчас все более важной становится возможность писать приложения, способные реагировать на события с высокой скоростью. На рис. 6.3 показана система, обрабатывающая огромное количество твитов в минуту. Эти сообщения отправляются буквально с миллионов устройств, представляющих собой источники событий, в систему, которая анализирует, преобразует, а затем отправляет твиты тем, кто зарегистрирован для их чтения. Обычно сообщение твита сопровождается хештегом, создающим выделенный канал и группу интересов. Система использует хештеги для фильтрации и разбиения уведомлений по темам.

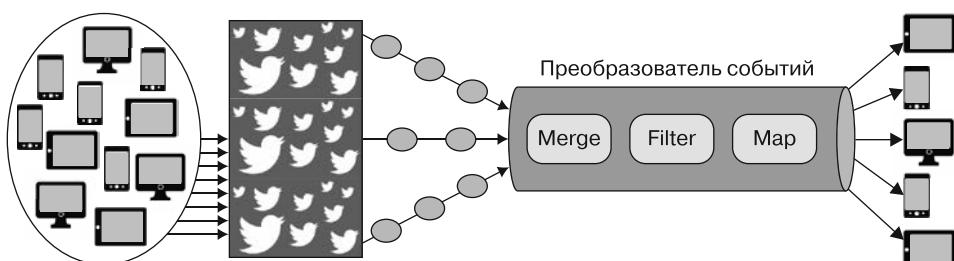


Рис. 6.3. Миллионы устройств представляют собой богатый источник событий, способный отправлять огромное количество твитов в минуту. Реактивная система реального времени может обрабатывать большие массы твитов в виде потока событий, применяя неблокирующие (асинхронные) операции (Merge, Filter и Map), а затем отправляя твиты адресатам (потребителям)

Каждый день миллионы устройств отправляют и получают уведомления, которые могли бы переполнить и, возможно, разрушить систему, если бы она не была рассчи-

тана на обработку такого устойчивого потока, состоящего из большого количества событий. Как бы вы написали такую систему?

Между ФП и реактивным программированием существует тесная взаимосвязь. Реактивное программирование задействует функциональные конструкторы для достижения композиционной абстракции событий. Как уже отмечалось, для обработки событий можно использовать операции более высокого порядка, такие как `map`, `filter` и `reduce`. Термином ФРП обычно обозначают реактивное программирование, но это не совсем верно.

ПРИМЕЧАНИЕ

ФРП — очень объемная тема; в данной главе рассматриваются только основные принципы этой парадигмы. Для более глубокого изучения ФРП рекомендую книгу «Функциональное реактивное программирование» Стивена Блэкхита (Stephen Blackheath) и Энтона Джонса (Anthony Jones), Manning Publications, 2016, www.manning.com/books/functional-reactive-programming.

6.2. Инструментарий .NET для реактивного программирования

.NET Framework поддерживает обработку событий на основе модели делегирования. Обработчик событий подписчика регистрирует цепочку событий и инициирует события при вызове. Используя парадигму императивного программирования, обработчики событий нуждаются в изменяемом состоянии, чтобы отслеживать подписки для регистрации обратного вызова. Этот вызов обертывает поведение в функцию, чтобы ограничить компонуемость.

Вот типичный пример регистрации события нажатия кнопки, при которой используется обработчик событий и анонимная лямбда-функция:

```
public MainWindow()
{
    myButton.Click += new System.EventHandler(myButton_Click);

    myButton.Click += (sender, args) => MessageBox.Show("Bye!");
}
void myButton_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hello!");
}
```

Данный шаблон является основной причиной того, что события в .NET сложно составлять и практически невозможно преобразовывать. В итоге это приводит к случайным утечкам памяти. Как правило, использование модели императивного программирования требует наличия разделяемого изменяемого состояния для коммуникации между событиями; в этом состоянии потенциально могут скрываться нежелательные побочные эффекты. При реализации сложных комбинаций событий

код, написанный в стиле императивного программирования, становится все более запутанным. Кроме того, явный обратный вызов функций ограничивает возможности выразить функциональность в декларативном стиле. В результате получаем программу, которую трудно понять и со временем становится невозможно расширять и отлаживать. Вдобавок события .NET не поддерживают конкурентное программирование; нельзя создавать события в отдельных потоках, что делает события .NET малопригодными для современных реактивных и масштабируемых приложений.

ПРИМЕЧАНИЕ

Потоки событий представляют собой неограниченные потоки обработки данных, поступающих из множества источников. Эти данные анализируются и преобразуются асинхронно, посредством операций, объединенных в конвейер.

События в .NET – первый шаг к реактивному программированию. События с самого начала были частью .NET Framework. На заре существования .NET Framework события в основном применялись при работе с графическими пользовательскими интерфейсами (graphical user interfaces, GUI). В настоящее время их потенциал применяется гораздо полнее. Компания Microsoft реализовала в .NET Framework возможность представлять и обрабатывать события как значения первого класса, используя модуль F# Event (и Observable), а также реактивные расширения .NET Reactive Extensions (Rx). Реактивные расширения позволяют легко создавать эффективные события в декларативном стиле. Кроме того, можно обрабатывать события как поток данных, в котором инкапсулированы логика и состояние, тем самым гарантируя отсутствие в коде побочных эффектов и изменяемых переменных. Теперь ваш код будет вполне соответствовать функциональной парадигме, целью которой являются асинхронный прием и обработка событий.

6.2.1. Улучшенное решение: комбинаторы событий

В настоящее время большинство систем получают обратные вызовы и обрабатывают события по мере того, как они происходят. Но если рассматривать события как потоки, подобные спискам или другим коллекциям, то можно использовать технологии работы с коллекциями или обработки событий, что делает обратные вызовы ненужными. Представление списка в F#, описанное в главе 5, содержит набор функций более высокого порядка, таких как `Filter` и `Map`, для работы со списками в декларативном стиле:

```
let squareOfDigits (chars:char list)
    |> List.filter (fun c -> Char.IsDigit c && int c % 2 = 0)
    |> List.map (fun n -> int n * int n)
```

В этом коде функция `squareOfDigits` принимает список символов и возвращает квадрат чисел, представленных в списке. Первая функция, `filter`, возвращает список элементов, для которых данный предикат истинный; в указанном случае

символы являются четными числами. Вторая функция, `map`, преобразует каждый элемент `n` в целое число и вычисляет его квадрат `n*n`. Оператор конвейера (`|>`) выстраивает операции в цепочку вычислений. Другими словами, результат операции в левой части уравнения будет использоваться как аргумент для следующей операции в конвейере.

Этот же код можно представить на языке LINQ, что более приемлемо для C#:

```
List<int> SquareOfDigits(List<char> chars) =>
    chars.Where(c => char.IsDigit(c) && char.GetNumericValue(c) % 2 == 0)
        .Select(c => (int)c * (int)c).ToList();
```

Такой выразительный стиль программирования идеально подходит для работы с событиями. По сравнению с C# у F# есть преимущество: он допускает внутреннюю (естественную) обработку событий как значений первого класса — это означает, что события можно передавать как данные. Кроме того, можно написать функцию, которая бы принимала событие в качестве аргумента и генерировала новое событие. Соответственно, события могут передаваться в функции, объединенные оператором конвейера (`|>`), как любые другие значения. Такие структура и метод использования событий в F# основаны на комбинаторах, выглядящих как программирование с применением списков вместо последовательностей. Комбинаторы событий представлены в модуле F# `Event`, который может задействоваться для компоновки событий:

```
textBox.KeyPress
|> Event.filter (fun c -> Char.IsDigit c.KeyChar && int c.KeyChar % 2 = 0)
|> Event.map (fun n -> int n.KeyChar * n.KeyChar)
```

В этом коде событие клавиатуры `KeyPress` рассматривается как поток, который фильтруется, чтобы игнорировать события, не представляющие интереса, так что окончательное вычисление выполняется только в том случае, если нажатые клавиши являются цифрами. Главное преимущество использования функций более высокого порядка — более четкое *разделение обязанностей*¹. Для того чтобы достичь такого же уровня выразительности и компонуемости в C#, нужно применять .NET Rx, как это будет кратко описано далее в настоящей главе.

6.2.2. Совместимость .NET с комбинаторами F#

Комбинаторы событий F# позволяют писать код с применением алгебры событий, с помощью которой можно разделить сложные события на более простые. Можно ли использовать преимущества модуля комбинаторов событий F# для написания более декларативного кода на C#? Да, можно.

Оба языка программирования .NET, F# и C#, задействуют одну и ту же *общезыковую среду выполнения* (Common Language Runtime, CLR), и оба они компилируются

¹ Этот принцип проектирования делит компьютерную программу на секции, каждая из которых имеет свое назначение. Такое разделение обязанностей упрощает разработку и обслуживание компьютерных программ (https://en.wikipedia.org/wiki/Separation_of_concerns).

в промежуточный язык (Intermediate Language, IL), соответствующий спецификации *общезыковой инфраструктуры* (Common Language Infrastructure, CLI), что позволяет применять один и тот же код.

Вообще говоря, все языки .NET распознают события. Однако в F# события используются как значения первого класса и, следовательно, требуют лишь немного дополнительного внимания. Чтобы события F# гарантированно можно было задействовать в других языках .NET, нужно сообщить об этом компилятору, декорировав событие атрибутом [`<CLIEvent>`]. Для создания сложных обработчиков событий, которые можно применять в коде C#, можно использовать встроенный композиционный аспект комбинаторов событий F# — это удобно и эффективно.

Чтобы лучше понять, как работают комбинаторы событий F# и как их можно легко задействовать в других языках программирования .NET, рассмотрим следующий пример. В листинге 6.1 показано, как реализовать простую игру в угадывание секретного слова, используя комбинаторы событий F#.

Код регистрирует два события: `KeyPress` (нажатие клавиши), поступающее от элемента управления WinForms и передаваемое в конструкцию `KeyPressedCombinators`, и `Elapsed` (прошедшее время) от `System.Timers.Timer`. Пользователь вводит текст — в данном случае допускаются только буквы (без цифр) — либо пока не будет угадано секретное слово, либо пока не истечет время по таймеру (задаваемое значением `interval`). Когда пользователь нажимает клавишу, комбинаторы `filter` и `event` преобразуют источник события в новое событие посредством выполнения *цепочки выражений*. Если время истекает прежде, чем будет угадано секретное слово, появляется сообщение «Игра окончена»; в противном случае, если секретное слово соответствует введенному, появляется сообщение «Вы выиграли!».

Листинг 6.1. Использование комбинатора F# Event для управления событиями нажатия клавиш

Функция `map` из модуля `Event` регистрирует и преобразует событие таймера, чтобы выдать уведомление о нажатии клавиши X

```

type KeyPressedEventCombinators(secretWord, interval,
  control:#System.Windows.Forms.Control) =
  let evt =
    let timer = new System.Timers.Timer(float interval)
    let timeElapsed = timer.Elapsed |> Event.map(fun _ -> 'X')
    let keyPressed = control.KeyPress
      |> Event.filter(fun kd -> Char.IsLetter kd.KeyChar)
      |> Event.map(fun kd -> Char.ToLower kd.KeyChar)
    timer.Start()
    keyPressed
    |> Event.merge timeElapsed
    |> Event.scan(fun acc c ->
      if c = 'X' then "Игра окончена"
      else
        let word = sprintf "%s%c" acc c
        if word = secretWord then "Вы выиграли!"

```

Создание и запуск экземпляра `System.Timers.Timer`

Регистрация события `KeyPress` с помощью модуля F# Event для фильтрации и публикации только событий нажатия строчных букв

Слияние фильтров для обработки события в целом. Когда истекает время по таймеру или происходит нажатие клавиши, событие выдает уведомление

```

else word
) String.Empty
)
[<CLIEvent>]
member this.OnKeyDown = evt ←
    Функция scan поддерживает внутреннее состояние
    с информацией о нажатой клавише и принудительно
    выдает результат каждого вызова функции аккумулятора
    Раскрытие события F# для других языков
    программирования .NET через специальный атрибут CLIEvent

```

У типа `KeyPressedEventArgs` есть параметр конструктора `control`, который ссылается на любой объект, производный от `System.Windows.Forms.Control`. Аннотация `#` в F# называется гибким типом — это означает, что данный параметр совместим с определенным базовым типом (<http://mng.bz/FSp2>).

Событие `KeyPress` связано с базовым элементом управления `System.Windows.Forms.Control`, переданным в конструктор типа, и его поток событий входит в конвейер комбинаторов событий F# для дальнейшей обработки. Событие `OnKeyDown` декорировано атрибутом `[<CLIEvent>]` для раскрытия (публикации), что позволяет сделать событие видимым для других языков .NET. Таким образом можно оформить подписку на событие, и оно может быть использовано в коде на C#. Так обеспечивается реактивная программируемость посредством ссылки на проект библиотеки F#. На рис. 6.4 представлен конвейер комбинаторов событий F#, где поток событий `KeyPress` проходит через последовательность функций, объединенных в цепочку.

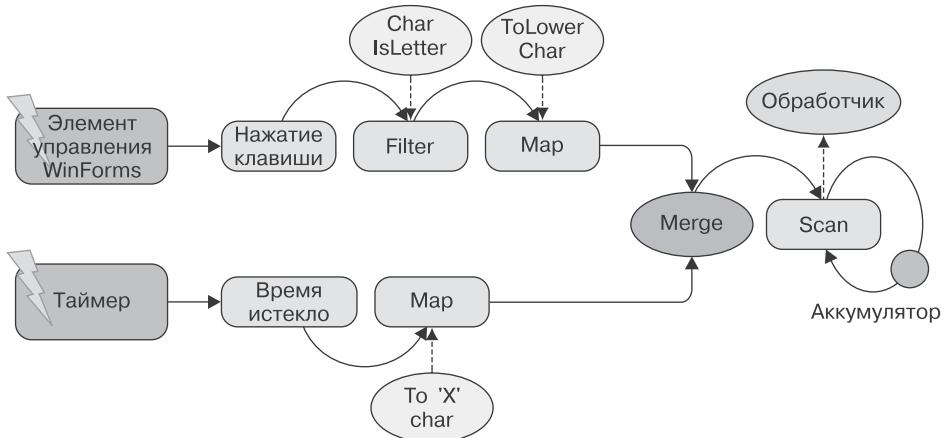


Рис. 6.4. Конвейер — комбинатор событий. На данной схеме видно, как два потока событий управляют каждый своим набором событий, прежде чем эти потоки будут объединены и переданы в аккумулятор. При нажатии клавиши на элементе управления WinForms событие `Filter` проверяет, является ли нажатая клавиша буквой, а затем событие `Map` преобразует нажатую букву в строчную версию перед сканированием. Когда истекает время по таймеру, оператор `Map` передает функции `Scan` значение `X`, означающее «отсутствие значения»

Цепочка комбинаторов событий, показанная на рис. 6.4, является сложной, и при этом демонстрирует простоту выражения столь сложной конструкции кода благодаря использованию событий в качестве значений первого класса. Комбинаторы событий F#

повышают уровень абстракции, чтобы упростить применение операций более высокого порядка для событий. Это делает код более удобочитаемым и понятным по сравнению с аналогичной программой, написанной в императивном стиле. Реализация программы с использованием типичного императивного стиля потребовала бы создания двух разных событий, передающих состояние таймера и поддерживающих состояние текста в виде разделяемого изменяемого состояния. Функциональный подход с комбинаторами событий позволяет избавиться от разделяемого неизменяемого состояния; кроме того, события становятся компонуемыми.

Подводя итог, отмечу основные преимущества применения комбинаторов событий F#.

- ❑ Компонуемость: можно определять события, захватывающие сложную логику, как набор из более простых событий.
- ❑ Декларативность: код, написанный с использованием комбинаторов событий F#, основан на принципах функционального программирования; следовательно, комбинаторы событий описывают то, что нужно выполнить, а не то, как выполнять задачу.
- ❑ Интероперабельность: комбинаторы событий F# могут совместно применяться в языках .NET, поэтому сложность кода может быть скрыта в библиотеке.

6.3. Реактивное программирование в .NET: реактивные расширения (Rx)

Библиотека .NET Rx упрощает создание асинхронных ситуационных программ с использованием наблюдаемых последовательностей. Библиотека Rx сочетает в себе простую семантику LINQ-стиля для управления коллекциями и обширные возможности асинхронных моделей программирования, что позволяет использовать чистые шаблоны `async/await` из .NET 4.5. Это мощное сочетание образует набор инструментов, позволяющий обрабатывать потоки событий в таком же простом, компонуемом и декларативном стиле, что и для коллекций данных (например, `List` и `Array`). Rx предоставляет предметно-ориентированный язык (Domain-Specific Language, DSL), который обеспечивает значительно более простой и гибкий API для обработки сложной асинхронной событийно-управляемой логики. Rx можно использовать для разработки отзывчивых пользовательских интерфейсов и для увеличения масштабируемости серверных приложений.

Коротко говоря, Rx представляет собой набор расширений для интерфейсов `IObservable<T>` и `IObserver<T>`, которые обеспечивают обобщенный механизм принудительной передачи уведомлений на основе шаблона «Наблюдатель» из книги «Банды четырех» (Gang of Four, GoF).

Шаблон проектирования «Наблюдатель» основан на событиях и является одним из самых распространенных шаблонов ООП. Он публикует изменения состояния объекта (наблюдаемого), делая их доступными для других объектов (наблюдателей), которые подписываются на уведомления, сообщающие обо всех изменениях наблюдаемого объекта (рис. 6.5).

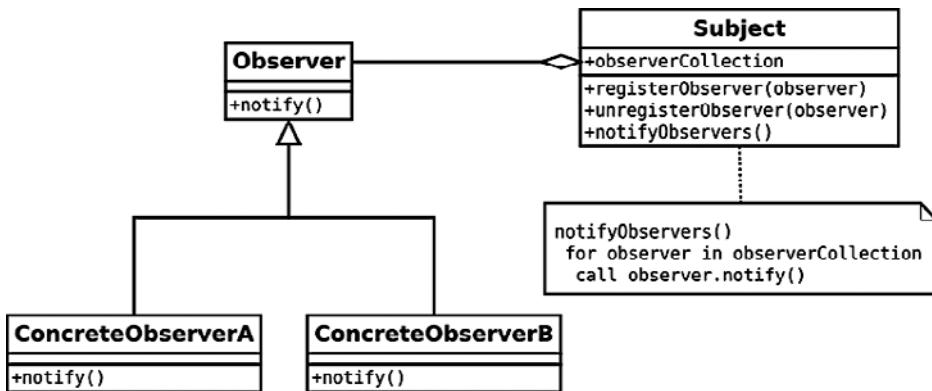


Рис. 6.5. Первоначальный шаблон «Наблюдатель» из книги GoF

Используя терминологию GoF, можно сказать, что интерфейсы **I_{Observe}rable** являются *субъектами*, а интерфейсы **I_{Observe}r** — *наблюдателями*. Эти интерфейсы, представленные в .NET 4.0 как часть пространства имен **System**, являются важным компонентом модели реактивного программирования.

Книга «Банды четырех»

В этой книге по разработке программного обеспечения, написанной Мартином Фаулером (Martin Fowler) и его соавторами, отражены шаблоны проектирования программного обеспечения в ООП. Название книги — «Шаблоны проектирования: элементы многоразового объектно-ориентированного программного обеспечения» — оказалось слишком длинным, особенно для электронной переписки, поэтому прозвище «книга «Банды четырех»» стало популярным способом ссылки на нее. Авторы часто упоминаются как «Банда четырех» (Gang of Four, GoF).

Ниже представлено определение для сигнатур интерфейсов **I_{Observe}r** и **I_{Observe}rable** на C#:

```

public interface IObserver<T>
{
    void OnCompleted();
    void OnError(Exception exception);
    void OnNext(T value);
}
public interface IObservable<T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

```

Эти интерфейсы реализуют шаблон «Наблюдатель», который позволяет создавать с помощью Rx события **observable** на основании существующих событий .NET CLR. На рис. 6.6 представлена схема исходного шаблона «Наблюдатель» из

книги GoF, выполненная на языке Unified Modeling Language (UML), — возможно, это прояснит суть данного шаблона.

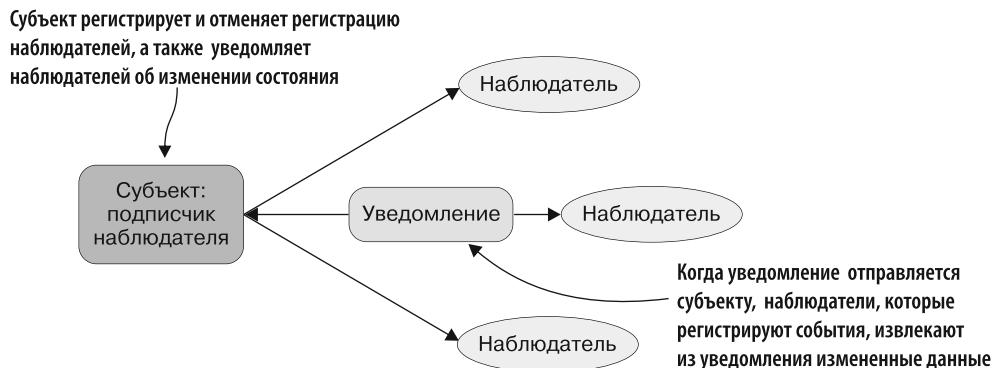


Рис. 6.6. В основе шаблона «Наблюдатель» лежит объект `Subject`, который обслуживает список зависимостей (называемых наблюдателями) и автоматически уведомляет наблюдателей о любых изменениях состояния `Subject`. Этот шаблон определяет отношения «один ко многим» между подписчиками-наблюдателями, так что при изменении состояния объекта все его зависимости получают уведомление и обновляются автоматически

Функциональный интерфейс `IObservable<T>` (www.lambdafaq.org/what-is-a-functionalinterface) реализует только метод `Subscribe`. Когда этот метод вызывается наблюдателем, инициируется уведомление для публикации нового элемента посредством метода `IObserver<T>.OnNext`. Интерфейс `IObservable`, как следует из названия, можно рассматривать как источник данных, которые находятся под постоянным наблюдением, и при любых изменениях состояния все зарегистрированные наблюдатели автоматически получают уведомления. Аналогичным образом через методы `IObserver<T>.OnError` и `IObserver<T>.OnCompleted` публикуются уведомления об ошибках и завершении работы соответственно. Метод `Subscribe` возвращает объект `IDisposable`, который действует как обработчик для подписанного наблюдателя. Когда вызывается метод `Dispose`, соответствующий наблюдатель отсоединяется от `Observable` и перестает получать уведомления. Подведем итог.

- ❑ `IObserver<T>.OnNext` предоставляет наблюдателю новые данные или информацию о состоянии.
- ❑ `IObserver<T>.OnError` указывает на ошибку поставщика.
- ❑ `IObserver<T>.OnCompleted` показывает, что наблюдатель завершил отправку уведомлений наблюдателям.

Те же интерфейсы применяются в качестве базового определения для типа F# `Ievent<'a>`. Этот тип является интерфейсом, используемым для реализации описанных ранее комбинаторов событий F#. Как видим, одни и те же принципы применяются с несколько другим подходом и позволяют получить такую же структуру. Возможность описать в коде несколько асинхронных источников событий — основное преимущество Rx.

ПРИМЕЧАНИЕ

.NET Rx можно загрузить с помощью команды `Install-Package System.Reactive` и подключить к проекту через диспетчер пакетов NuGet.

6.3.1. От LINQ/PLINQ к Rx

Как обсуждалось в главе 5, поставщики запросов LINQ/PLINQ в .NET действуют как механизм обработки последовательности, хранящейся в памяти. Концептуально данный механизм основан на модели приема (pull model) — это означает, что элементы коллекций принимаются из запроса во время его выполнения. Такое поведение представлено в виде шаблона итератора `IEnumerable<T>` — `IEnumerator<T>`, который может вызвать блокировку, пока ожидает поступления данных для итерации. Rx, напротив, рассматривает события как поток данных, реагируя на запросы по мере поступления событий. Это модель принудительной передачи (push model), в которой события поступают и автономно проходят через запрос. Обе модели показаны на рис. 6.7.

При реактивной модели приложение является пассивным и не блокирует процесс извлечения данных.



Рис. 6.7. Сравнение моделей приема и передачи. В основе шаблона `IEnumerable/IEnumerator` лежит модель приема, которая запрашивает новые данные у источника данных.

Шаблон `IObservable/IObserver`, напротив, основан на модели принудительной передачи, согласно которой объект получает уведомление, когда новые данные становятся доступны для отправки потребителю

F#: вдохновение для Rx

В интервью на Microsoft's Channel 9 (<http://bit.ly/2v8exjV>) Эрик Мейер (Erik Meijer), идейный вдохновитель Rx, отметил, что язык F# стал источником вдохновения для создания реактивных расширений. Одной из самых воодушевляющих идей, лежащих в основе фреймворка Reactive, стали компонуемые события, впервые появившиеся в F#.

6.3.2. IObservable и Ienumerable: дуальные интерфейсы

Реализованная в Rx модель на основе принудительной передачи событий абстрагируется интерфейсом `IObservable<T>`, который является дуальным для интерфейса `IEnumerable<T>`¹. Термин «дуальность» может показаться сложным, но на самом деле это простая и мощная концепция. Дуальность можно сравнить с двумя сторонами монеты, где задняя сторона просвечивает через переднюю.

В контексте информатики эта концепция используется в законе Де Моргана², где на основе дуальности между конъюнкцией `&&` (AND) и дизъюнкцией `||` (OR) доказывается, что отрицание распространяется как на конъюнкцию, так и на дизъюнкцию:

```
!(a || b) == !a && !b
!(a && b) == !a || !b
```

Подобно инверсии LINQ, где LINQ — набор методов расширения в интерфейсе `IEnumerable`, реализующих модель приема для коллекций, Rx предоставляет набор методов расширения для интерфейса `IObservable`, реализующих модель принудительной передачи событий. На рис. 6.8 показаны дуальные отношения между этими интерфейсами.

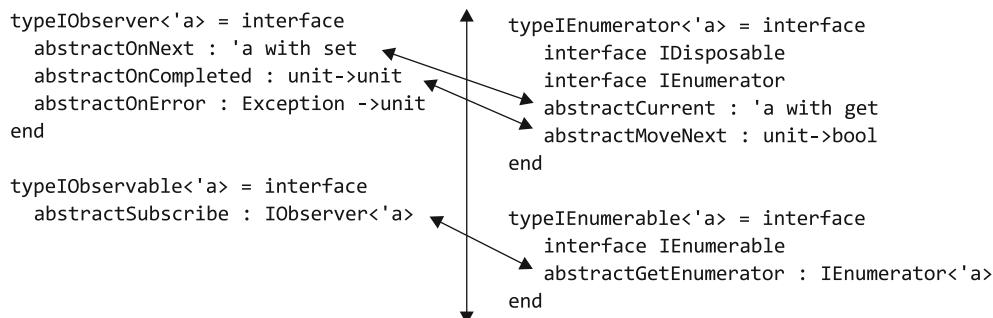


Рис. 6.8. Дуальные отношения между интерфейсами `IObserver` и `IEnumerator`, а также интерфейсами `IObservable` и `IEnumerable`. Такие дуальные отношения достигаются путем изменения направления стрелок в функциях — другими словами, вход и выход меняются местами

¹ Термин происходит от слова «дуальность». Подробнее см.: [https://en.wikipedia.org/wiki/Dual_\(category_theory\)](https://en.wikipedia.org/wiki/Dual_(category_theory)).

² Подробнее см.: https://en.wikipedia.org/wiki/De_Morgan%27s_laws.

Как показано на рис. 6.8, интерфейсы `IObservable` и `IObserver` получаются из соответствующих интерфейсов `IEnumerable` и `IEnumerator` путем изменения направления стрелок.

Замена направления стрелки означает, что вход и выход метода меняются местами. Например, сейчас свойство интерфейса `IEnumerator` имеет такую сигнатуру:

```
Unit (or void in C#) -> get 'a
```

Если повернуть стрелку для данного свойства, то получим его двойник: `Unit <- set 'a`. Эта сигнатура в противоположном интерфейсе `IObserver` соответствует методу `OnNext`, сигнатура которого выглядит так:

```
set 'a -> Unit (or void in C#)
```

Функция `GetEnumerator` не принимает аргументов и возвращает объект `IEnumerator<T>`, который, в свою очередь, возвращает следующий элемент списка посредством функций `MoveNext` и `Current`. Обратный метод, `IEnumerable`, может использоваться для перемещения `IObservable`, который передает данные подписчику `IObserver`, вызывая его методы.

6.3.3. Реактивные расширения в действии

Важной характеристикой Rx является создание комбинаций из существующих событий. Это позволяет достичь уровня абстракции и компонуемости, что с помощью других средств было невозможно. В .NET события — это вариант асинхронного источника данных, который может быть использован в Rx. Чтобы преобразовать существующие события в наблюдаемые, Rx принимает событие и возвращает объект `EventPattern`, аргументами которого являются отправитель и событие. Например, событие `keypressed` преобразуется в реактивное наблюдаемое событие (выделено жирным шрифтом):

```
Observable.FromEventPattern<KeyPressedEventArgs>(this.textBox,
                                                     nameof(this.textBox.KeyPress));
```

Как видим, в Rx есть широкие возможности многоразовой обработки событий.

Воспользуемся фреймворком Rx, реализовав на C# эквивалент игры в секретное слово, описанной ранее с использованием комбинаторов событий F# `KeyPressed-EventCombinators`. В листинге 6.2 показана реализация игры с применением этого шаблона и соответствующего реактивного фреймворка.

Листинг 6.2. Rx KeyPressedEventCombinators в C#

Метод Rx FromEventPattern преобразует событие .NET в наблюдаемое	LINQ-подобные семантические функции Select и Where регистрируют, компилируют и трансформируют события для уведомления подписчиков
-----------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------

```
var timer = new System.Timers.Timer(timerInterval);
var timerElapsed = Observable.FromEventPattern<ElapsedEventArgs>
    (timer, "Elapsed").Select(_ => 'X');
var keyPressed = Observable.FromEventPattern<KeyPressEventArgs>
    (this.textBox, nameof(this.textBox.KeyPress));
    .Select(kd => Char.ToLower(kd.EventArgs.KeyChar))
    .Where(c => Char.IsLetter(c));
```

```

timer.Start();

timerElapsed
    .Merge(keyPressed)
    .Scan(String.Empty, (acc, c) =>
{
    if (c == 'X') return "Игра окончена";
    else
    {
        var word = acc + c;
        if (word == secretWord) return "Вы выиграли!";
        else return word;
    }
}).Subscribe(value =>
    this.label.BeginInvoke(
        Action(() => this.label.Text = value)));

```

Объединение фильтров позволяет обрабатывать их целиком. Когда событие инициируется, происходит передача уведомления

Функция Scan отслеживает внутреннее состояние нажатых клавиш и передает результат каждого вызова функции аккумулятора

Метод `Observable.FromEventPattern` устанавливает связь между событием .NET и объектом Rx `I Observable`, который служит оберткой для `Sender` и `EventArgs`. В листинге 6.2 императивные события C# для обработки нажатия клавиши (`KeyPressEventArgs`) и времени, прошедшего согласно таймеру (`ElapsedEventArgs`), преобразуются в наблюдаемые объекты и объединяются для обработки как единый поток событий. Теперь можно построить всю обработку событий как единую и сжатую цепочку выражений.

Реактивные расширения являются функциональными

Фреймворк Rx обеспечивает функциональный подход для асинхронной обработки событий как потока. Функциональный аспект означает декларативный стиль программирования, при котором используется меньше переменных для обслуживания состояния и предотвращения изменений, так что можно объединять события в цепочку выражений.

6.3.4. Потоковая передача в реальном времени с помощью Rx

Поток событий — это канал, по которому последовательность событий подается в виде значений в порядке поступления. Потоки событий поступают из разных источников, таких как социальные сети, фондовый рынок, смартфоны или компьютерная мышь. Обработка потоков событий в реальном времени направлена на потребление потока постоянно поступающих данных, которые могут быть представлены в различных формах. Потребление данных, зачастую поставляемых с высокой скоростью, может быть огромным — это похоже на попытку напиться из пожарного шланга. Возьмем, к примеру, постоянно изменяющиеся цены на акции, которые

надо проанализировать, а затем отправить результат нескольким потребителям, как показано на рис. 6.9.

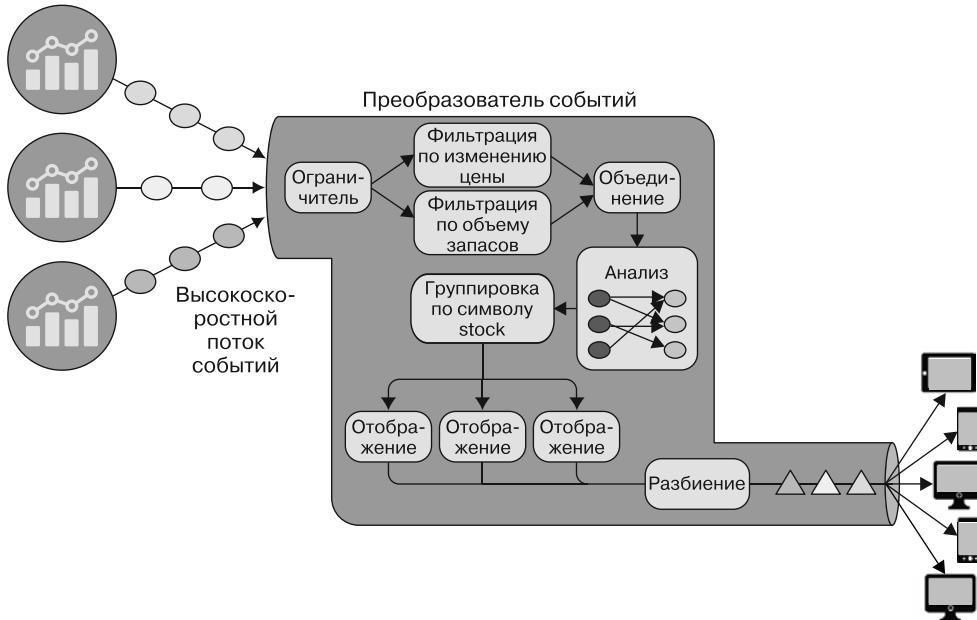


Рис. 6.9. Потоки событий из разных источников поставляют данные преобразователю событий, который применяет операции более высокого порядка, а затем уведомляет подписанных наблюдателей

Фреймворк Rx хорошо подходит для такого сценария, поскольку позволяет обрабатывать несколько асинхронных источников данных посредством комбинаций высокопроизводительных операций по преобразованию и фильтрации любого из этих потоков данных. Rx использует интерфейс `IObservable<T>` для поддержки списка зависимых интерфейсов `IObserver<T>`, которые автоматически получают уведомления о любых изменениях событий и данных.

6.3.5. От событий к наблюдаемым объектам F#

Как мы помним, в F# события используются для создания настраиваемых конструкций обратного вызова. Кроме того, этот язык поддерживает альтернативный и более прогрессивный механизм настраиваемых обратных вызовов, которые являются более компонуемыми, чем события. Язык F# рассматривает события .NET как значения типа `IEvent<'T>`, унаследованные от интерфейса `IObservable<'T>` того же типа, что и в Rx. По этой причине модуль `Observable` входит в состав основной сборки F# `Fsharp.Core`. Данный модуль включает в себя, кроме значений интерфейса `IObservable`, также набор полезных функций, которые считаются подмножеством Rx.

Например, в следующем фрагменте кода наблюдаемые объекты F# (выделены жирным шрифтом) используются для обработки событий нажатия клавиш и таймера из примера KeyPressedEventCombinators (см. листинг 6.2):

```
let timeElapsed = timer.Elapsed |> Observable.map(fun _ -> 'X')
let keyPressed = control.KeyPress
    |> Observable.filter(fun c -> Char.IsLetter c)
    |> Observable.map(fun kd -> Char.ToLower kd.KeyChar)

let disposable =
keyPressed
|> Observable.merge timeElapsed
|> Observable.scan(fun acc c ->
    if c = 'X' then "Game Over"
    else
        let word = sprintf "%s%c" acc c
        if word = secretWord then "You Won!"
        else word
) String.Empty
|> Observable.subscribe(fun text -> printfn "%s" text)
```

При создании реактивных систем на F# можно выбрать (и использовать) либо `Observable`, либо `Event`; но, чтобы избежать утечек памяти, лучше выбрать `Observable`. При применении модуля F# `Event` скомпонованные события присоединяются к начальному событию, и у них нет механизма отмены подписки, что может привести к утечке памяти. Модуль `Observable`, напротив, предоставляет оператору `subscribe` возможность зарегистрировать функцию обратного вызова. Этот оператор возвращает объект `IDisposable`, который может использоваться, чтобы остановить обработку потока событий и отменить регистрацию всех подписанных обработчиков `Observable` (или `Event`) в конвейере посредством одного вызова метода `Dispose`.

6.4. Укрощение потока событий: анализ эмоций в Twitter посредством Rx-программирования

В наш век цифровой информации, когда к Интернету подключены миллиарды устройств, программы должны сопоставлять, объединять, фильтровать и анализировать данные в реальном времени. Скорость обработки данных переместилась в область аналитики реального времени, сократив время ожидания доступа к информации практически до нуля. Реактивное программирование — прекрасный способ удовлетворить требования высокой производительности, поскольку он совместим с конкурентностью, хорошо масштабируется и обеспечивает компонуемую асинхронную семантику для обработки данных.

По оценкам, в Соединенных Штатах генерируется примерно 24 млн твитов в час, что составляет почти 7000 сообщений в секунду. Это огромное количество данных, нуждающихся в обработке, создает серьезную проблему при потреблении такого высоконагруженного потока. Соответственно, система должна быть сконструи-

рована таким образом, чтобы справиться с возникающим обратным давлением. Например, в случае с Twitter это обратное давление может быть вызвано тем, что потребитель реального потока не способен выдержать скорость, с которой поставщики генерируют события.

Обратное давление

Обратное давление возникает в том случае, когда компьютерная система не в состоянии обрабатывать входные данные достаточно быстро. Тогда она начинает буферизировать поступающие данные, пока объем буфера не уменьшится до уровня ухудшения чувствительности системы или, что еще хуже, пока не приведет к исключению вследствие недостатка памяти. В случае итерирования набора элементов `IEnumerable` потребитель работает в режиме контролируемого приема; элементы обрабатываются с контролируемой скоростью. В `IObservable` элементы принудительно передаются потребителю. В этом случае `IObservable` потенциально может генерировать значения быстрее, чем подписанные наблюдатели способны их обрабатывать. Такой сценарий создает избыточное обратное давление, усиливая нагрузку на систему. Для уменьшения обратного давления в Rx предусмотрены операторы `Throttle` и `Buffer`.

На рис. 6.10 показан пример кода на F#, иллюстрирующий поток анализа в реальном времени для определения текущего настроения (эмоции) твитов, опубликованных в Соединенных Штатах.

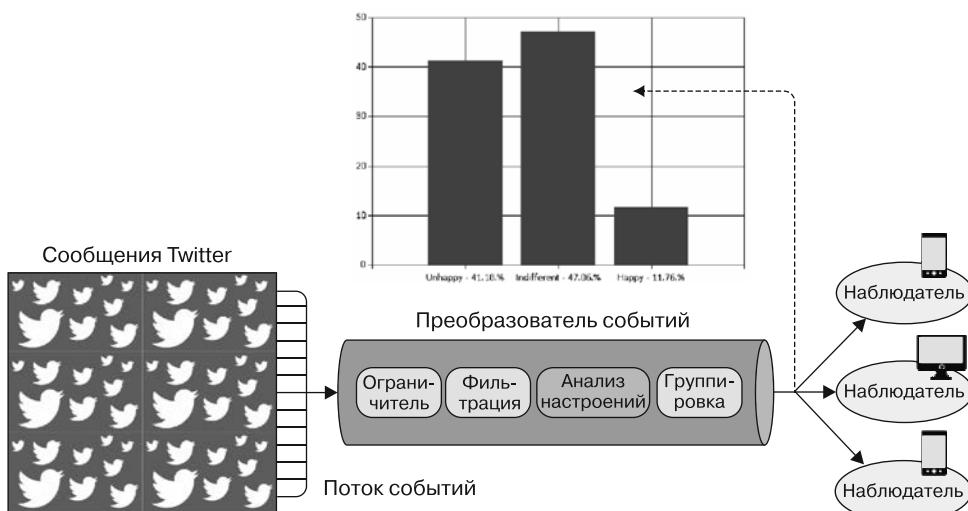


Рис. 6.10. Сообщения Twitter порождают высокоскоростной поток событий, направленный к потребителю. Поэтому важно, чтобы такие инструменты, как Rx, справлялись с непрерывным волем уведомлений. Сначала поток ограничивается (функция `Throttle`), затем сообщения фильтруются, анализируются и группируются по эмоциям. В результате получается поток данных, состоящий из входящих твитов и отражающий последнее состояние эмоций, значения которых постоянно обновляют диаграмму и поступают к подписанным наблюдателям в виде уведомлений

В этом примере продемонстрирована встроенная в F# поддержка наблюдаемых объектов, которая отсутствует в C#. Но эта функциональность может быть воспроизведена и на C#, с помощью .NET Rx либо путем ссылки на библиотеку F# и использования ее функций, в коде которых реализованы наблюдаемые объекты.

Анализ потока твитов выполняется путем их потребления и извлечения информации из каждого сообщения. Анализ эмоций выполняется с помощью библиотеки Stanford CoreNLP. Результат анализа передается в постоянно обновляемую анимированную диаграмму, которая принимает на входе объект `IEnumerable` и автоматически обновляет график по мере изменения данных.

Stanford CoreNLP

Stanford CoreNLP (<http://nlp.stanford.edu>) — это библиотека естественно-языкового анализа, написанная на Java. Ее можно интегрировать в .NET с помощью моста IKVM (www.ikvm.net). В данной библиотеке есть несколько инструментов, включая средства анализа эмоций, которые определяют эмоцию предложения. Для того чтобы установить библиотеку Stanford CoreNLP, можно воспользоваться NuGet-пакетом `Install-Package Stanford.NLP.CoreNLP` (www.nuget.org/packages/Stanford.NLP.CoreNLP); этот пакет также настраивает мост IKVM. Подробнее о том, как работает библиотека CoreNLP, читайте в онлайн-документации.

В листинге 6.3 показана функция анализа эмоций и настройка для подключения библиотеки Stanford CoreNLP.

Листинг 6.3. Определение эмоции предложения с помощью библиотеки CoreNLP

```

let properties = Properties()
properties.setProperty("annotators", "tokenize,ssplit,pos,parse,emotion")
➥ |> ignore

IO.Directory.SetCurrentDirectory(jarDirectory)
let stanfordNLP = StanfordCoreNLP(properties)           ↪ Задание свойств и создание
                                                               экземпляра StanfordCoreNLP

type Emotion =
    | Unhappy
    | Indifferent
    | Happy           ↪ Размеченное объединение классифицирует
                           эмоции для каждого текстового сообщения

let getEmotionMeaning value =
    match value with
    | 0 | 1 -> Unhappy
    | 2 -> Indifferent
    | 3 | 4 -> Happy           ↪ Присваивание значений от 0 до 4,
                           соответствующих эмоциям

let evaluateEmotion (text:string) =
    let annotation = Annotation(text)
    stanfordNLP.annotate(annotation)

let emotions =

```

```

let emotionAnnotationClassName =
  SentimentCoreAnnotations.SentimentAnnotatedTree().getClass()
  let sentences = annotation.get(CoreAnnotations.SentencesAnnotation()).
    getClass())
  :?> java.util.ArrayList
  [ for s in sentences ->
    let sentence = s :?> Annotation
    let sentenceTree = sentence.get(emotionAnnotationClassName)
  :?> Tree
    let emotion = NNCoreAnnotations.getPredictedClass(sentenceTree)
    getEmotionMeaning emotion]
  (emotions.[0])

```

← Аналisis текстового сообщения
и вывод связанной эмоции

В этом коде на F# размеченнное объединение определяет разные уровни эмоции (варианты значений): `Unhappy` (печальный), `Indifferent` (безразличный) и `Happy` (веселый). Затем вычисляется процент распределения данных значений среди твитов. Функция `valuEmotion` выполняет текстовый анализ средствами библиотеки Stanford и возвращает полученное значение (вариант эмоции).

Чтобы получить поток твитов, я воспользовался библиотекой Tweetinvi (<https://github.com/linv1/tweetinvi>). У нее есть хорошо документированный API, и, что более важно, она рассчитана на конкурентное выполнение потоков и управление много-поточными сценариями. Эту библиотеку можно загрузить и установить из NuGet-пакета `TweetinviAPI`.

ПРИМЕЧАНИЕ

Twitter предоставляет отличный API для разработчиков. Для того чтобы получить ключ и секретный доступ, достаточно лишь создать учетную запись Twitter и учетную запись приложения (<https://apps.twitter.com>). Обладая этой информацией, можно отправлять и получать твиты и взаимодействовать с API Twitter.

В листинге 6.4 показано, как создать экземпляр библиотеки `Tweetinvi` и как получить доступ к настройке параметров взаимодействия с Twitter.

Листинг 6.4. Настройка параметров для подключения библиотеки `Twitterinvi`

```

let consumerKey = "<your Key>"
let consumerSecretKey = "<your secret key>"
let accessToken = "<your access token>"
let accessTokenSecret = "<your secret access token>

let cred = new TwitterCredentials(consumerKey, consumerSecretKey,
  accessToken, accessTokenSecret)
let stream = Stream.CreateSampleStream(cred)
stream.FilterLevel <- StreamFilterLevel.Low

```

Этот простой код создает экземпляр потока Twitter. В листинге 6.5 показаны основные операции программирования Rx (выделены жирным шрифтом), где для обработки и анализа потока событий используется сочетание Rx и F#-модуля `Observable`.

Листинг 6.5. Анализ твитов посредством наблюдаемого конвейера

```

let emotionMap =
    [(Unhappy, 0)
     (Indifferent, 0)
     (Happy, 0)] |> Map.ofSeq
    
```

Генерирование потока событий от API Twitter

```

let observableTweets =
    stream.TweetReceived |>
    Observable.throttle(TimeSpan.FromMilliseconds(100.))
    
```

Управление скоростью поступления событий во избежание переполнения у потребителя

```

|> Observable.filter(fun args ->
    args.Tweet.Language = Language.English)
    
```

Фильтрация входящих сообщений — отбор лишь тех, что написаны на английском языке

```

|> Observable.groupBy(fun args ->
    evaluateEmotion args.Tweet.FullText)
    
```

Разделение сообщений для анализа эмоций

```

|> Observable.selectMany(fun args ->
    args |> Observable.map(fun i ->
        (args.Key, (max 1 i.Tweet.FavoriteCount))))
    
```

Преобразование сообщений в последовательность эмоций с подсчетом лидеров

```

|> Observable.scan(fun sm (key,count) ->
    match sm |> Map.tryFind key with
    | Some(v) -> sm |> Map.add key (v + count)
    | None -> sm ) emotionMap
    
```

Поддержка состояния общего разделения сообщений по эмоциям

```

|> Observable.map(fun sm ->
    let total = sm |> Seq.sumBy(fun v -> v.Value)
    sm |> Seq.map(fun k ->
        let percentageEmotion = ((float k.Value) * 100.)
        / (float total)
        let labelText = sprintf "%A - %.2f.%%" (k.Key)
        percentageEmotion
        (labelText, percentageEmotion)
    ))
    
```

Вычисление общего процента эмоций и возвращение наблюдаемого объекта для обновления диаграммы в реальном времени

Результатом выполнения конвейера `observableTweets` является объект `IDisposable`, который используется для прекращения приема твитов и удаления подписки из данных подписанного наблюдаемого объекта. В `TweetInv` есть обработчик события `TweetReceived`, уведомляющий подписчиков о поступлении нового твита. Наблюдаемые объекты объединяются в цепочку, образуя наблюдаемый конвейер. На каждом шаге возвращается новый наблюдаемый объект, который принимает события от исходного наблюдаемого объекта, а затем запускает сформированное в результате события из заданной функции.

Первым этапом в канале наблюдаемых объектов является управление обратным давлением, которое образуется вследствие высокой скорости поступающих событий. При написании Rx-кода помните, что процесс может быть перегружен, если события потока будут поступать слишком быстро.

На рис. 6.11 в показанной слева системе нет проблем с обработкой входящих потоков событий, поскольку частота уведомлений имеет постоянную пропускную способность (желательный поток). Система, показанная справа, борется за то, чтобы не отставать от огромного количества уведомлений (справиться с обратным

давлением), которые она постоянно получает, что может потенциально привести к краху системы. Чтобы избежать сбоя, в системе организовано ограничение потоков событий. Результатом является разная скорость уведомлений у наблюдаемого объекта и наблюдателя.

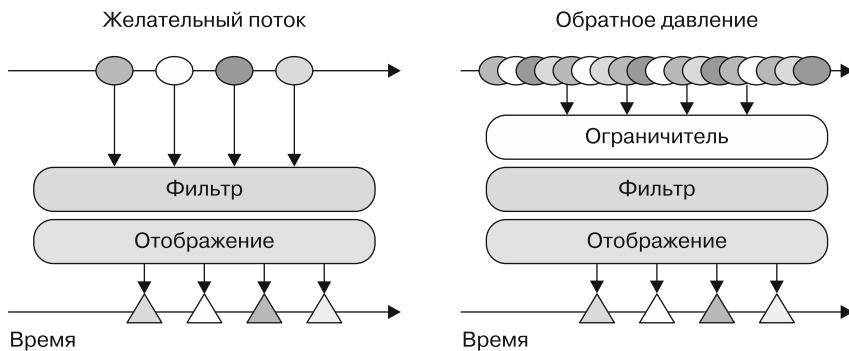


Рис. 6.11. Обратное давление может отрицательно влиять на отзывчивость системы. Но можно снизить скорость входящих событий и сохранить работоспособность системы, используя функцию ограничителя (throttle) для компенсации различия в скорости между наблюдаемым объектом и наблюдателем

Чтобы избежать проблемы обратного давления, создается уровень защиты в виде функции `throttle`, которая управляет скоростью сообщений, не позволяя им поступать слишком быстро:

```
stream.TweetReceived
| > Observable.throttle(TimeSpan.FromMilliseconds(50.))
```

Функция `throttle` уменьшает скорость поступления данных до уровня, соответствующего определенной частоте (ритму), как показано на рис. 6.9 и 6.10. Функция `throttle` извлекает последнее значение из пакета данных в последовательности наблюдаемых объектов, игнорируя любое значение, которое поступило менее чем через указанный интервал после предыдущего. В листинге 6.5 максимальная частота распространения событий ограничена до одного события каждые 50 мс.

Как Rx-операторы `throttle` и `buffer` позволяют справиться с большими объемами событий

Следует помнить, что у функции `throttle` возможны деструктивные эффекты. Это значит, что сигналы, поступающие с более высокой скоростью, чем заданная частота, не буферизуются, а теряются. Так происходит потому, что функция `throttle` отбрасывает сигнал из наблюдаемой последовательности, если интервал между этим и предыдущим сигналом меньше заданного. Оператор `throttle` также называется

сглаживанием *дребезга* (*debounce*), при котором устанавливается минимальный интервал между сообщениями и сообщения с большей скоростью перестают поступать.

Функция `buffer` полезна в тех случаях, когда обрабатывать сигналы по одному обходится слишком дорого и, следовательно, предпочтительнее обрабатывать их пакетами, собранными за счет введения задержки. Но при использовании функции `buffer` с событиями большого размера возникает проблема. В таких событиях сигналы хранятся в памяти в течение определенного периода времени, и система может столкнуться с проблемой переполнения памяти. Назначение оператора `buffer` состоит в том, чтобы спрятать определенную последовательность сигналов и затем повторно опубликовать их либо по истечении заданного времени, либо когда буфер будет заполнен.

Например, в следующем коде на C# объект получает либо все события, которые произошли за секунду, либо последние 50 сигналов, в зависимости от того, какое из условий выполнится раньше.

```
Observable.Buffer(TimeSpan.FromSeconds(1), 50)
```

В примере с анализом эмоций твитов метод расширения `Buffer` может применяться таким образом:

```
stream.TweetReceived.Buffer(TimeSpan.FromSeconds(1), 50)
```

Следующим шагом в конвейере является фильтрация событий, которые неважны для анализа (команда выделена жирным шрифтом):

```
|> Observable.filter(fun args -> args.Tweet.Language = Language.English)
```

Функция `filter` гарантирует, что будут обрабатываться только твиты, написанные по-английски. Объект `Tweet` из твита имеет ряд свойств, к которым можно получить доступ, в том числе отправителя сообщения, хештег и координаты (*местоположение*).

Затем оператор Rx `groupBy` предоставляет возможность разделить последовательность на ряд наблюдаемых групп, в зависимости от функции-селектора. Каждый из наблюдаемых подобъектов соответствует уникальному значению ключа, в котором содержатся все элементы, имеющие такое же значение ключа, — так же как это делается в LINQ и SQL:

```
|> Observable.groupBy(fun args -> evaluateEmotion args.Tweet.FullText)
|> Observable.selectMany(fun args -> args |> Observable.map(fun i ->
  (args.Key, i.Tweet.FavoriteCount)))
```

В данном случае ключом, по которому разделяется поток событий, является эмоция. Функция `evaluateEmotion`, которая ведет себя как групповой селектор, вычисляет и классифицирует эмоцию для каждого входящего сообщения. Каждый вложенный наблюдаемый объект может иметь собственную уникальную операцию; для дальнейшей подписки на эти группы наблюдаемых объектов используется оператор `selectMany`, объединяющий их в один объект. Затем с помощью функции `map` последовательность преобразуется в новую последовательность пар (кортеж),

состоящую из значений Tweet-Emotion и количества лайков (предпочтений) для данного твита.

После разделения и анализа данные должны быть агрегированы в осмысленный формат. Это делает функция `scan` наблюдаемого объекта, передавая результат каждого вызова в функцию `accumulator`. Возвращаемый наблюдаемый объект будет генерировать уведомления для каждого вычисленного значения состояния, как показано на рис. 6.12.

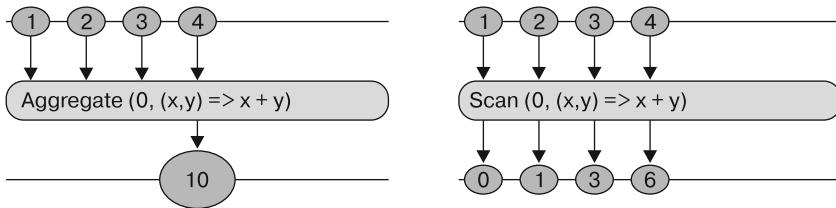


Рис. 6.12. Функция `aggregate` возвращает одно значение, которое выступает аккумулятором всех значений, полученных в результате выполнения заданной функции (x, y) относительно исходного аккумулятора 0. Функция `scan` возвращает значение для каждого элемента коллекции, что является результатом выполнения заданной функции для аккумулятора на текущей итерации

Функция `scan` аналогична функции `fold` или `Aggregate` в LINQ, но только она возвращает не одно значение, а промежуточные вычисления, полученные на каждой итерации (в следующем фрагменте кода выделено жирным шрифтом). Более того, она удовлетворяет парадигме функционального программирования, сохраняя неизменяемое состояние. Функции агрегирования (такие как `scan` и `fold`) принадлежат к разряду операций, называемых в ФП *катаморфизмами* (<https://wiki.haskell.org/Catamorphisms>):

```
< этот код выполняет наблюдение за твитами и анализ эмоций >
|> Observable.scan(fun sm (key,count) ->
  match sm |> Map.tryFind key with
  | Some(v) -> sm |> Map.add key (v + count)
  | None -> sm) emotionMap
```

Функция `scan` принимает три аргумента: наблюдаемый объект, который передается концептуально в виде потока твитов для анализа эмоций, анонимная функция для записи в аккумулятор базовых значений наблюдаемого объекта и аккумулятор `emotionMap`. Результатом выполнения функции `scan` является обновленный аккумулятор, который передается на следующую итерацию. Начальное состояние аккумулятора в предыдущем коде используется функцией `scan` в пустой F#-функции `Map`, что эквивалентно неизменяемому параметризованному словарю .NET `Dictionary<K, V>`, где ключом является одна из эмоций, а значением — количество соответствующих этой эмоции твитов. Функция аккумулятора `scan` обновляет элементы коллекции, записывая в них новые вычисленные типы, и возвращает обновленную коллекцию в качестве нового аккумулятора.

Последняя операция в конвейере — это выполнение функции `map`, используемой для преобразования исходных наблюдаемых объектов в представление общего процента твитов, проанализированных и разделенных в соответствии с эмоциями:

```
|> Observable.map(fun sm ->
    let total = sm |> Seq.sumBy(fun v -> v.Value)
    sm |> Seq.map(fun k ->
        let percentageEmotion = ((float k.Value) * 100.) / (float total)
        let labelText = sprintf "%A - %.2f.%" (k.Key) percentageEmotion
        (labelText, percentageEmotion)
    ))
)
```

Функция преобразования выполняется один раз для каждого подписанного наблюдателя. Функция `map` вычисляет общее количество твитов для переданного наблюдаемого объекта, который содержит значение аккумулятора, полученное от предыдущей функции `scan`:

```
sm |> Seq.sumBy(fun v -> v.Value)
```

Результат возвращается в формате, где представлен процент каждой эмоции из полученной таблицы соответствий. Заключительный наблюдаемый объект передается в `LiveChart`, где в реальном времени генерируются обновления. Теперь, когда код готов, можно использовать функцию `StartStreamAsync()`, чтобы запустить процесс отслеживания событий и приема твитов и позволить наблюдаемым объектам отправлять подписчикам уведомления:

```
LiveChart.Column(observableTweets,Name= sprintf "Tweet
➥ Emotions").ShowChart()
do stream.StartStreamAsync()
```

«Холодные» и «горячие» наблюдаемые объекты

Существует два вида наблюдаемых объектов: «горячие» и «холодные». «Горячий» наблюдаемый объект представляет собой постоянный поток данных, который передает уведомления независимо от наличия у него подписчиков. Например, поток твитов является «горячим» потоком данных, поскольку они будут продолжать поступать независимо от состояния подписчиков. «Холодный» наблюдаемый объект — это поток событий, который всегда передает уведомления по запросу в начале потока, независимо от того, переходят ли подписчики в режим приема после того, как событие будет передано.

Подобно модулю `Event` в F#, модуль `Observable` определяет набор комбинаторов для использования интерфейса `IObservable<T>`. Модуль `Observable` в F# включает в себя операции `add`, `filter`, `partition`, `merge`, `choose` и `scan`. Для получения дополнительной информации обратитесь к приложению B.

В предыдущем примере функции наблюдаемых объектов `groupBy` и `selectMany` являются частью фреймворка Rx. Этот пример иллюстрирует полезные для разработчика свойства F#: возможность комбинировать и составлять инструменты, чтобы результат наилучшим образом соответствовал поставленной задаче.

SelectMany: монадический оператор связывания. SelectMany — мощный оператор, соответствующий оператору bind (или flatMap) в других языках программирования. Этот оператор производит одно монадическое значение из другого и имеет общую сигнатуру монадического связывания:

$M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b,$

где M — любой надтип, который ведет себя как контейнер. В случае наблюдаемых объектов он имеет следующую сигнатуру:

`IObservable'T' > -> ('T -> IObservable'R') -> IObservable'R'`

В .NET есть несколько типов, соответствующих указанной сигнатуре: `IObservable`, `IEnumerable` и `Task.Monads` ([https://en.wikipedia.org/wiki/Monad_\(functional_programming\)](https://en.wikipedia.org/wiki/Monad_(functional_programming))). Несмотря на репутацию сложных типов, их можно описать простыми словами: это контейнеры, которые инкапсулируют и абстрагируют заданную функциональность, чтобы обеспечить возможность компоновки надтипов без побочных эффектов. В принципе, работу с монадами можно представить как работу с коробками (контейнерами), которые распаковываются в последний момент — тогда, когда в них возникает необходимость.

Главная цель монадических вычислений — обеспечить возможность компоновки там, где она не может быть достигнута другими способами. Например, используя монады в C#, можно напрямую сложить целое число и тип `Task` из пространства имен `System.Threading.Tasks` для целочисленных значений (`Task<int>`) (выделено жирным шрифтом):

```
Task<int> result = from task in Task.Run<int>(() => 40)
                     select task + 2;
```

Операция bind, или SelectMany, принимает надтип и применяет функцию к его базовому значению, возвращая другой надтип. *Надтип* — это оболочка вокруг другого типа, например `IEnumerable<int>`, `Nullable<bool>` или `Ibservable< Tweets>`. Смысл операции bind зависит от типа монады. В `Ibservable` обрабатывается каждое событие из входных наблюдаемых объектов и создается новый наблюдаемый объект. Полученные наблюдаемые объекты затем объединяются, чтобы получить выходной наблюдаемый объект, как показано на рис. 6.13.

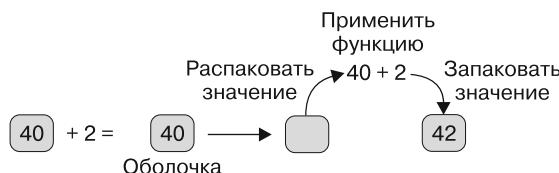


Рис. 6.13. Надтип можно рассматривать как специальный контейнер, позволяющий применить функцию непосредственно к базовому типу (в данном случае к числу 40). Надтип работает как оболочка, содержащая значение, которое может быть извлечено для использования заданной функции, после чего результат возвращается обратно в контейнер

Связка SelectMany не только *объединяет* значения данных, но так же, как оператор, преобразует и затем объединяет вложенные монадические значения. Базовая

теория монад применяется в языке LINQ, который, в свою очередь, используется компилятором .NET для интерпретации шаблона `SelectMany` и применения монадического поведения. Например, путем реализации метода расширения `SelectMany` для типа `Task` (выделено жирным шрифтом в следующем фрагменте кода) компилятор распознает шаблон и интерпретирует его как монадическую связку, что позволяет использовать специальную компоновку:

```
Task<R> SelectMany<T, R>(this Task<T> source, Func<T, Task<R>> selector) =>
    source.ContinueWith(t => selector(t.Result)).Unwrap();
```

Благодаря применению этого метода предыдущий код на основе LINQ будет компилироваться и вычисляться как `Task<int>`, что даст на выходе число 42. Монады играют роль импорта в функциональном конкурентном программировании и будут более подробно рассмотрены в главе 7.

6.5. Шаблон «издатель — подписчик» в Rx

Шаблон «издатель — подписчик» позволяет любому числу издателей асинхронно обмениваться данными с любым количеством подписчиков через канал событий. Как правило, для осуществления такой коммуникации используется промежуточный концентратор, который получает уведомления, а затем пересыпает их подписчикам. Благодаря Rx становится возможным эффективно описать шаблон «издатель — подписчик» посредством встроенных инструментов и конкурентной модели.

Идеальным кандидатом для такой реализации является тип `Subject`. Он реализует интерфейс `ISubject`, который представляет собой сочетание `IObservable` и `IObserver`. В результате `Subject` ведет себя как наблюдатель и наблюдаемый объект одновременно, что позволяет ему играть роль брокера, перехватывая уведомления в качестве наблюдателя и рассыпая их всем своим наблюдателям. `IObserver` и `IObservable` можно рассматривать как интерфейсы подписчика и издателя соответственно (рис. 6.14).

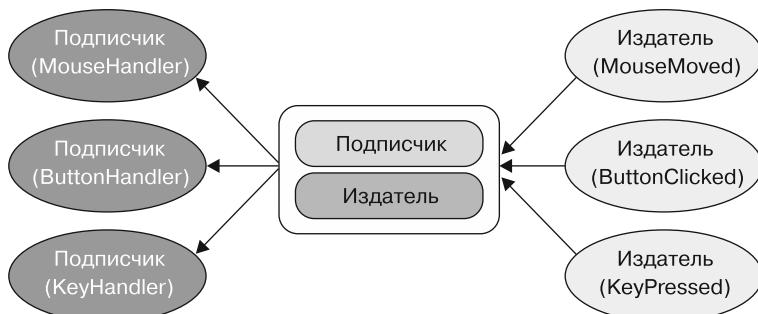


Рис. 6.14. В шаблоне «издатель — подписчик» концентратор управляет коммуникацией между любым количеством подписчиков (наблюдателей) и любым количеством издателей (наблюдаемых объектов). Концентратор, также известный как брокер, получает уведомления от издателей; затем эти уведомления пересыпаются подписчикам

Преимуществом использования типа `Subject` из Rx для представления шаблона «издатель — подписчик» является возможность вводить в уведомления перед публикацией дополнительную логику, такую как `merge` и `filter`.

6.5.1. Использование типа `Subject` для создания мощного концентратора в шаблоне «издатель — подписчик»

Тип `Subject` — это компонент Rx, целью которого является синхронизация значений, создаваемых наблюдаемыми объектами, и наблюдателей, потребляющих данные значения. `Subject` не полностью соответствует функциональной парадигме, поскольку поддерживает потенциально изменяемые состояния или управляет ими. Тем не менее `Subject` полезен для создания наблюдаемых объектов, таких как события, и идеально подходит для реализации шаблона «издатель — подписчик».

Тип `Subject` реализует интерфейс `ISubject` (в следующем фрагменте кода выделен жирным шрифтом), принадлежащий пространству имен `System.Reactive.Subjects`:

```
interface ISubject<T, R> : IObservable<R>, IObserver<T> { }
```

или `ISubject<T>`, если источник и результат относятся к одному и тому же типу.

Поскольку `Subject<T>` и, следовательно, `ISubject<T>` являются наблюдателями, в них реализованы методы `OnNext`, `OnCompleted` и `OnError`. Поэтому при их вызове для всех подписчиков-наблюдателей выполняются одни и те же методы.

В стандартной конфигурации Rx есть несколько реализаций класса `Subject`, каждая из которых имеет свое поведение. Кроме того, если ни один из существующих вариантов `Subject` не удовлетворяет вашим потребностям, вы можете написать собственную реализацию. Единственное требование к нестандартной реализации класса `Subject` — она должна соответствовать реализации интерфейса `ISubject`.

Существуют следующие готовые варианты `Subject`.

- ❑ `ReplaySubject` ведет себя как обычный `Subject`, но сохраняет все полученные сообщения, делая их доступными как для существующих, так и для будущих подписчиков.
- ❑ `BehaviorSubject` всегда сохраняет последнее полученное значение, делая его доступным для будущих подписчиков.
- ❑ `AsyncSubject` обеспечивает асинхронную работу, маршрутизируя только последнее уведомление, полученное во время ожидания сообщения `OnComplete`.

ПРИМЕЧАНИЕ

Тип `Subject` является «горячим», что делает его уязвимым для потери уведомлений, переданных от источника (наблюдаемого объекта) при отсутствии принимающих наблюдателей. Чтобы этого не произошло, хорошо подумайте, прежде чем использовать тип `Subject`, особенно если необходимо, чтобы подписчики получили все сообщения. Примером «горячего» наблюдаемого объекта является движение мыши, когда оно происходит постоянно и уведомления о нем передаются независимо от того, существуют ли принимающие наблюдатели.

6.5.2. Rx и конкурентность

В основе фреймворка Rx лежит модель принудительной передачи с поддержкой многопоточности. Но важно помнить, что по умолчанию Rx является однопоточным, а параллельные конструкции, которые позволяют объединять асинхронные источники, необходимо активировать с помощью планировщиков Rx.

Одной из основных причин введения конкурентности в Rx-программировании является уменьшение нагрузки на поток событий и управление данной нагрузкой. Это позволяет выполнять несколько конкурентных задач, таких как обеспечение отзывчивости пользовательского интерфейса, чтобы освободить текущий поток.

ПРИМЕЧАНИЕ

Реактивные расширения позволяют комбинировать асинхронные источники, используя параллельные вычисления. Потенциально эти асинхронные источники могут генерироваться независимо от параллельных вычислений. Rx справляется со сложностями, связанными с компоновкой таких источников, позволяет сосредоточиться на их композиционном аспекте и писать код в декларативном стиле.

Более того, Rx позволяет управлять потоком входящих сообщений в виде конкретных потоков и получать вычисления с высоким уровнем конкурентности. Rx – это система асинхронных запросов потоков событий, которой требуется определенный уровень управления конкурентностью.

В режиме многопоточности Rx-программирование повышает эффективность использования вычислительных ресурсов на многоядерном оборудовании, что повышает производительность вычислений. В этом случае сообщения могут поступать одновременно из разных контекстов выполнения. Фактически несколько асинхронных источников могут быть результатом работы отдельных параллельных вычислений и при этом объединяться в один конвейер `Observable`. Другими словами, наблюдатели и наблюдаемые объекты имеют дело с асинхронными операциями, выполняемыми над последовательностью значений в рамках модели принудительной передачи. В конечном итоге Rx справляется со всеми сложностями, связанными с управлением доступом к этим уведомлениям, и позволяет избежать основных проблем конкурентности: снаружи все выглядит так, как если бы задачи выполнялись в одном потоке.

Если просто использовать тип `Subject` (или любые другие наблюдаемые объекты Rx), то код не станет автоматически конкурентным и не будет работать быстрее. По умолчанию операция принудительной передачи сообщений нескольким подписчикам от типа `Subject` выполняется в том же потоке. Более того, уведомления отправляются всем подписчикам последовательно, в порядке их подписки, и, возможно, блокируют операцию до ее завершения.

Фреймворк Rx устраняет это ограничение благодаря методам `ObserveOn` и `SubscribeOn`, которые позволяют регистрировать планировщик `Scheduler` для поддерж-

ки конкурентности. Планировщики Rx предназначены для конкурентного создания и обработки событий, повышая отзывчивость и масштабируемость и одновременно снижая сложность. Они обеспечивают уровень абстракции, расположенный поверх модели конкурентности, которая позволяет выполнять операции с потоком движения данных, и она не должна быть отражена непосредственно в базовой конкурентной реализации. Более того, в планировщиках Rx интегрирована возможность отмены задач, обработки ошибок и передачи состояния. Все планировщики Rx реализуют интерфейс `IScheduler`, принадлежащий пространству имен `System.Reactive.Concurrency`.

ПРИМЕЧАНИЕ

Рекомендуемыми встроенными планировщиками для .NET Framework после .NET 4.0 являются `TaskPoolScheduler` либо `ThreadPoolScheduler`.

Метод `SubscribeOn` определяет, какой планировщик следует активировать для поддержки очереди сообщений, выполняемой в другом потоке. Метод `ObserveOn` определяет, в каком потоке будет выполняться функция обратного вызова. Этот метод предназначен для планировщика, который обрабатывает выходные сообщения, и программирования пользовательского интерфейса (например, для обновления интерфейса WPF). Метод `ObserveOn` используется главным образом для программирования пользовательских интерфейсов и взаимодействия с `SynchronizationContext` (<http://bit.ly/2wiVBxu>).

В случае программирования пользовательского интерфейса оба оператора, `SubscribeOn` и `ObserveOn`, могут быть объединены, чтобы лучше управлять потоками и выбирать, какой из них будет выполняться на каждом шаге наблюдаемого конвейера.

6.5.3. Реализация на Rx многоразового шаблона «издатель — подписчик»

Вооружившись знаниями о классе `Subject` в Rx, мы можем гораздо проще описать многоразовый параметрический объект `Pub-Sub`, сочетающий в себе публикацию и подписку на один и тот же источник. В этом разделе мы сначала, используя тип `Subject` в Rx, построим конкурентный концентратор для шаблона «издатель — подписчик». А затем выполним рефакторинг представленного в предыдущем примере анализатора эмоций Twitter, воспользовавшись новой и более простой функциональностью, предоставляемой концентратором издания — подписки на основе Rx.

В реализации реактивного концентратора издания — подписки для подписки используется класс `Subject`; затем значения передаются наблюдателям с рассылкой многоадресных уведомлений от источников к наблюдателям. В листинге 6.6 показана реализация класса `RxPubSub`, в котором Rx применяется для создания параметрического объекта `Pub-Sub`.

Листинг 6.6. Реактивный шаблон «издатель — подписчик» на C#

Внутреннее состояние наблюдателей

```

public class RxPubSub<T> : IDisposable
{
    private ISubject<T> subject; ← Частный субъект уведомляет всех зарегистрированных
    private List<IObserver<T>> observers = new List<IObserver<T>>(); состояния наблюдаемых объектов
    private List<IDisposable> observables = new List<IDisposable>(); ←

    public RxPubSub(ISubject<T> subject)
    {
        this.subject = subject;
    }
    public RxPubSub() : this(new Subject<T>()) { } ← Внутреннее состояние
                                                наблюдаемых объектов

    public IDisposable Subscribe(IObserver<T> observer)
    {
        observers.Add(observer); ← Конструктор создает экземпляр
        subject.Subscribe(observer); чтобы можно было удалить наблюдателя
        return new Subscription<T>(observer, observers);
    }
                                                AddPublisher оформляет подписку на наблюдаемый объект, по умолчанию
                                                используя TaskPoolScheduler для обработки конкурентных уведомлений
    public IDisposable AddPublisher(IObservable<T> observable) =>
        observable.SubscribeOn(TaskPoolScheduler.Default).Subscribe(subject); ←

    public IObservable<T> AsObservable() => subject.AsObservable(); ←

    public void Dispose()
    {
        observers.ForEach(x => x.OnCompleted()); ← IObservable<T> от внутреннего объекта ISubject
        observers.Clear(); ← предназначен для применения
                            операций более высокого порядка
    }
                                                Удаление всех подписчиков
                                                после удаления объекта
                                                к уведомлениям о событиях
}

```

Внутренний класс ObserverHandler служит оберткой для IObserver, создавая объект IDisposable, который используется для остановки потока уведомлений и удаления его из коллекции наблюдателей

```

class ObserverHandler<T> : IDisposable ←
{
    private IObserver<T> observer;
    private List<IObserver<T>> observers;

    public ObserverHandler(IObserver<T> observer,
    List<IObserver<T>> observers)
    {
        this.observer = observer;
        this.observers = observers;
    }

    public void Dispose()
    {
        observer.OnCompleted();
        observers.Remove(observer);
    }
}

```

Экземпляр класса RxPubSub может быть создан либо с помощью конструктора с указанием версии Subject, либо с помощью главного конструктора, который создает объект Subject по умолчанию и передает его дальше. Кроме частного поля Subject, существует еще два частных поля коллекций: коллекция наблюдателей observers и коллекция подписанных наблюдаемых объектов observables.

Коллекция observers прежде всего поддерживает состояние наблюдателей, подписанных на Subject через новый экземпляр класса Subscription. Этот класс предоставляет метод отмены подписки Dispose через интерфейс IDisposable, вызов которого позволяет удалить конкретного наблюдателя.

Вторая частная коллекция называется observables. Наблюдаемые объекты представляют собой список интерфейсов IDisposable, которые получаются в результате регистрации каждого наблюдаемого объекта с помощью метода AddPublisher. Для того чтобы отменить регистрацию наблюдаемого объекта, существует открытый метод Dispose.

В этой реализации подписка на Subject создается планировщиком TaskPoolScheduler:

```
observable.SubscribeOn(TaskPoolScheduler.Default)
```

TaskPoolScheduler составляет план обработки для каждого наблюдателя. Каждый блок обработки запускается в отдельном потоке с использованием текущего предоставленного объекта TaskFactory (<http://bit.ly/2vaemTA>). Незначительно изменив код, можно применить любой другой планировщик.

Подписанные наблюдаемые объекты из внутреннего Subject отображаются через интерфейс I Observable, полученный путем вызова метода AsObservable. Это свойство используется для применения к уведомлениям о событиях операций более высокого порядка:

```
public I Observable<T> AsObservable() => subject.AsObservable();
```

Причиной применения интерфейса I Observable в Subject является то, что он гарантирует: никто не сможет выполнить обратное преобразование в I Subject и наделать ошибок. Объекты Subject — это компоненты с хранимым состоянием, поэтому рекомендуется изолировать доступ к ним посредством инкапсуляции; в противном случае есть вероятность повторной инициализации или непосредственного обновления Subject.

6.5.4. Анализ эмоций твитов с помощью Rx-класса Pub-Sub

В листинге 6.7 для обработки потока эмоций твитов использован реактивный класс C# Pub-Sub (RxPubSub). Этот листинг — еще один пример того, как можно легко организовать взаимодействие между языками программирования C# и F# и позволить им сосуществовать в одном решении. Из библиотеки F#, реализованной в разделе 6.4, взят наблюдаемый объект, который принудительно передает поток эмоций

твитов, поэтому на него легко подписываются внешние наблюдатели. (Команды наблюдаемых объектов выделены жирным шрифтом.)

Листинг 6.7. Реализация наблюдаемых объектов для эмоций твитов

```
let tweetEmotionObservable(throttle:TimeSpan) =
    Observable.Create(fun (observer:IObserver<_>) ->
        let cred = new TwitterCredentials(consumerKey, consumerSecretKey,
            accessToken, accessTokenSecret)
        let stream = Stream.CreateSampleStream(cred)
        stream.FilterLevel <- StreamFilterLevel.Low
        stream.StartStreamAsync() |> ignore
        stream.TweetReceived
        |> Observable.throttle(throttle)
        |> Observable.filter(fun args ->
            args.Tweet.Language = Language.English)
        |> Observable.groupBy(fun args ->
            evaluateEmotion args.Tweet.FullText)
        |> Observable.selectMany(fun args ->
            Observable.map(fun tw ->
                TweetEmotion.Create tw.Tweet args.Key))
        |> Observable.subscribe(observer.OnNext)
    )
```

Observable.Create создает наблюдаемый объект с помощью заданной функции, которая принимает в качестве параметра наблюдателя; тот наблюдатель подписывается на возвращаемый наблюдаемый объект

Наблюдаемый объект подписывается на метод **OnNext** у наблюдателя, чтобы принудительно передавать ему изменения

В листинге показана реализация объекта `tweetEmotionObservable` с помощью оператора фабрики наблюдаемого объекта `Create`. Этот оператор принимает функцию, параметром которой является наблюдатель, а сама функция ведет себя как наблюдаемый объект, вызывая его методы.

Оператор `Observable.Create` регистрирует наблюдателя, переданного в функцию, и начинает принудительно передавать уведомления по мере их поступления. Наблюдаемый объект определяется из метода `subscribe`, который принудительно передает уведомления наблюдателю `observer`, вызывающему метод `OnNext`. В листинге 6.8 показана эквивалентная реализация `tweetEmotionObservable` на C# (выделена жирным шрифтом).

Листинг 6.8. Реализация `tweetEmotionObservable` на C#

```
var tweetObservable = Observable.FromEventPattern<TweetEventArgs>(stream,
    "TweetReceived");

Observable.Create<TweetEmotion>(observer =>
{
    var cred = new TwitterCredentials(
        consumerKey, consumerSecretKey, accessToken, accessTokenSecret);
    var stream = Stream.CreateSampleStream(cred);
    stream.FilterLevel = StreamFilterLevel.Low;
    stream.StartStreamAsync();

    return Observable.FromEventPattern<TweetReceivedEventArgs>(stream,
        "TweetReceived")
```

```

.Throttle(throttle)
.Select(args => args.EventArgs)
.Where(args => args.Tweet.Language == Language.English)
.GroupBy(args =>
    evaluateEmotion(args.Tweet.FullText))
.SelectMany(args =>
    args.Select(tw => TweetEmotion.Create(tw.Tweet, args.Key)))
.Subscribe(o=>observer.OnNext(o));
});

```

Метод `FromEventPattern` преобразует .NET-событие CLR в наблюдаемый объект. В данном случае событие `TweetReceived` преобразуется в `Iobservable`.

Единственное различие между реализациями на C# и F# заключается в том, что для кода на F# не требуется создание Observable `tweetObservable` с использованием `FromEventPattern`. Фактически на F# обработчик события `TweetReceived` автоматически становится наблюдаемым объектом при передаче в конвейер `stream.TweetReceived |> Observable`. `TweetEmotion` — это тип-значение (структура), который несет информацию об эмоции твита (выделен жирным шрифтом) (листинг 6.9).

Листинг 6.9. Структура `TweetEmotion` для хранения свойств твита

```

[<Struct>]
type TweetEmotion(tweet:ITweet, emotion:Emotion) =
    member this.Tweet with get() = tweet
    member this.Emotion with get() = emotion

    static member Create tweet emotion =
        TweetEmotion(tweet, emotion)

```

В листинге 6.10 показана реализация класса `RxTweetEmotion`, унаследованного от класса `RxPubSub` и подписанного на `Iobservable` для управления уведомлениями об эмоциях твитов (выделено жирным шрифтом).

Листинг 6.10. Реализация класса `TweetEmotion`, унаследованного от `RxPubSub`

```

Класс RxTweetEmotion
унаследован от RxPubSub
→ class RxTweetEmotion : RxPubSub<TweetEmotion>
{
    public RxTweetEmotion(TimeSpan throttle) ← Передача значения throttle
    {
        var obs = TweetsAnalysis.tweetEmotionObservable(throttle)
            .SubscribeOn(TaskPoolScheduler.Default); ← в конструктор, а затем
            base.AddPublisher(obs); ← в определение tweetEmotionObservable
    }
}

```

Конкурентное выполнение уведомлений Tweet-Emotions
 посредством TaskPoolScheduler. Это полезно при обработке
 конкурентных сообщений и наличии нескольких наблюдателей

Класс `RxTweetEmotion` создает и регистрирует в базовом классе наблюдаемый объект `tweetEmotionObservable` с помощью метода `AddPublisher` через наблюдаемый

объект `obs`, который активизирует уведомление из внутреннего `TweetReceived`. На следующем шаге нужно зарегистрировать наблюдателей, чтобы сделать что-то полезное.

6.5.5. Наблюдатели в действии

Реализация класса `RxTweetEmotion` готова. Но без подписки наблюдателей невозможно ни уведомлять о происходящих событиях, ни реагировать на них. Чтобы создать реализацию интерфейса `IObserver`, можно описать класс, который бы наследовал и реализовывал каждый из методов этого интерфейса. К счастью, в Rx есть набор вспомогательных функций, упрощающих такую работу. Метод `Observer.Create()` позволяет описывать новых наблюдателей:

```
IObserver<T> Create<T>(Action<T> onNext,
                        Action<Exception> onError,
                        Action onCompleted)
```

У данного метода есть несколько переопределений, так что ему можно передавать практически любую реализацию методов `OnNext`, `OnError` и `OnCompleted` и получать на выходе объект `IObserver<T>`, вызывающий предоставленные ему функции.

Эти вспомогательные функции Rx сводят к минимуму количество типов, создаваемых в программе, а также ограничивают излишнее распространение классов. Вот пример `IObserver`, который выводит в консоль только твиты с позитивными эмоциями:

```
var tweetPositiveObserver = Observer.Create<TweetEmotion>(tweet => {
    if (tweet.Emotion.IsHappy)
        Console.WriteLine(tweet.Tweet.Text);
});
```

После создания наблюдателя `tweetPositiveObserver` его экземпляр регистрируется в экземпляре реализованного ранее класса `RxTweetEmotion`, который уведомляет каждого подписанного наблюдателя о получении твита с позитивной эмоцией:

```
var rxTweetEmotion = new RxTweetEmotion(TimeSpan.FromMilliseconds(150));
IDisposable postTweets = rxTweetEmotion.Subscribe(tweetPositiveObserver);
```

Для каждого подписанного наблюдателя возвращается экземпляр интерфейса `IDisposable`. Этот интерфейс может использоваться для того, чтобы наблюдатель прекратил получать уведомления и чтобы отменить регистрацию (удалить) наблюдателя у издателя, вызвав метод `Dispose`.

6.5.6. Удобное выражение объектов в F#

Выражение объектов в F# – удобный способ реализовать на лету любой экземпляр анонимного объекта, основанный на известном существующем интерфейсе (или нескольких интерфейсах). Выражения объектов в F# работают аналогично методу `Observer.Create()`, но могут применяться к любому заданному интерфейсу.

Кроме того, вследствие поддержки функциональной совместимости экземпляры, созданные путем выражения объектов в F#, можно передавать в другие языки программирования .NET.

В следующем коде показано, как посредством выражения объекта в F# можно создать экземпляр `IObserver<TweetEmotion>`, чтобы отображать в консоли только грустные эмоции:

```
let printUnhappyTweets() =
    { new IObserver<TweetEmotion> with
        member this.OnNext(tweet) =
            if tweet.Emotion = Unhappy then
                Console.WriteLine(tweet.Tweet.text)

        member this.OnCompleted() = ()
        member this.OnError(exn) = () }
```

Цель выражения объектов состоит в том, чтобы избежать излишнего кода, необходимого для определения и создания новых именованных типов. Экземпляр, полученный из выражения объекта, может быть использован в проекте C# посредством ссылки на библиотеку F# и импортирования соответствующего пространства имен. Ниже показано, как можно применить выражение объекта F# в коде C#:

```
IObserver<TweetEmotion> unhappyTweetObserver = printUnhappyTweets();

IDisposable disposable = rxTweetEmotion.Subscribe(unhappyTweetObserver);
```

Экземпляр наблюдателя `unhappyTweetObserver` определяется с помощью выражения объекта F#; затем он подписывается на `rxTweetEmotion`, после чего готов принимать уведомления.

Резюме

- ❑ В парадигме реактивного программирования используются неблокирующие асинхронные операции с высокой скоростью обработки последовательности событий. Эта парадигма программирования ориентирована на асинхронный прием и обработку последовательности событий в виде потока событий.
- ❑ Rx рассматривает поток событий как последовательность событий. Rx позволяет использовать ту же выразительную семантику программирования, что и LINQ, и применять к событиям операции более высокого порядка, такие как `filter`, `map` и `reduce`.
- ❑ Rx в .NET обеспечивает полную поддержку многопоточного программирования. В сущности, Rx позволяет обрабатывать несколько событий одновременно, в том числе параллельно. Кроме того, Rx интегрируется с программированием клиентов, что позволяет напрямую обновлять графические интерфейсы.
- ❑ Планировщики Rx предназначены для конкурентной генерации и обработки событий, повышения отзывчивости и масштабируемости, а также уменьшения сложности. Планировщики Rx обеспечивают абстракцию поверх модели конкурентности, что позволяет выполнять операции по перемещению потоков данных без непосредственного отображения в базовой конкурентной реализации.
- ❑ Язык программирования F# рассматривает события как значения первого класса, а это значит, что их можно передавать как данные. Такой подход является основой, которая влияет на комбинаторы событий, позволяющие программировать обработку событий как регулярной последовательности.
- ❑ Специальные комбинаторы событий в F# могут описываться и использоваться в других языках программирования .NET. Этот мощный стиль программирования позволяет упростить традиционную событийно-управляемую модель программирования.
- ❑ Реактивное программирование имеет значительные преимущества перед асинхронным выполнением при создании компонентов и компоновке рабочих процессов. Более того, возможности Rx, позволяющие справиться с обратным давлением, имеют решающее значение, дающее возможность избежать перегрузки и неограниченного потребления ресурсов.
- ❑ Rx помогает решить проблему обратного давления при постоянных резких скачках плотности уведомлений, позволяя управлять высокоскоростными потоками событий, чтобы они не перегружали потребителей.
- ❑ Rx предоставляет набор инструментов для реализации полезных реактивных шаблонов, таких как «издатель — подписчик».



Функциональный параллелизм на основе задач

В этой главе:

- параллелизм на основе задач и семантика декларативного программирования;
- компоновка параллельных операций с применением функциональных комбинаторов;
- максимальное использование ресурсов с помощью Task Parallel Library;
- реализация параллельного функционального шаблона «конвейер».

Парадигма параллелизма на основе задач предусматривает разделение программы и параллельное выполнение каждой части, тем самым сокращая общую продолжительность выполнения. Эта парадигма нацелена на распределение задач между процессорами для максимального задействования процессоров и повышения производительности. Традиционно, чтобы организовать параллельное выполнение программы, нужно было разбить ее код на отдельные функциональные области, а затем выполнять его в разных потоках. В этом сценарии для синхронизации доступа нескольких потоков к разделяемым ресурсам используются примитивные блокировки. Цель блокировок — избежать состояния гонки и повреждения памяти, обеспечивая конкурентное взаимное исключение. Главными причинами применения блокировок является устаревшая структура проекта, требующая ожидания завершения выполнения текущего потока, прежде чем ресурс будет доступен другим потокам для продолжения работы.

Более новый, улучшенный механизм состоит в том, чтобы для продолжения работы передать остальную часть вычислений функции обратного вызова (которая

выполняется после завершения выполнения потока). В функциональном программировании этот метод называется *стилем продолжений* (Continuation-Passing Style, CPS). Из данной главы вы узнаете, как применять этот механизм для параллельного запуска нескольких задач без блокировки выполнения программы. Вы также научитесь с помощью указанной технологии реализовывать параллельные программы на основе задач, изолируя побочные эффекты; вы освоите функциональную компоновку, которая упрощает распараллеливание задач в коде. Компоновка является одним из наиболее важных свойств ФП, она облегчает применение декларативного стиля программирования. Если код легко читается, его также легко поддерживать. Используя ФП, вы задействуете в своих программах параллелизм на основе задач, не усложняя их, по сравнению с обычным программированием.

7.1. Краткое введение в параллелизм

Параллелизм на основе задач — процесс параллельного запуска нескольких независимых задач на нескольких процессорах. Данная парадигма подразумевает разделение вычислений на ряд задач меньшего размера и выполнение этих меньших задач в нескольких потоках. Благодаря одновременному выполнению нескольких функций сокращается время выполнения.

Как правило, выполнение параллельных задач начинается из одной точки, с одними и теми же данными и может либо прекращаться в автономном режиме, независимо друг от друга, либо полностью завершаться для всей группы задач. Когда компьютерная программа одновременно вычисляет разные автономные выражения, используя одни и те же исходные данные, — это параллелизм задач. В основе такой концепции лежат небольшие вычислительные блоки, называемые *будущими значениями* (*futures*). На рис. 7.1 показано сравнение распараллеливания данных и задач.

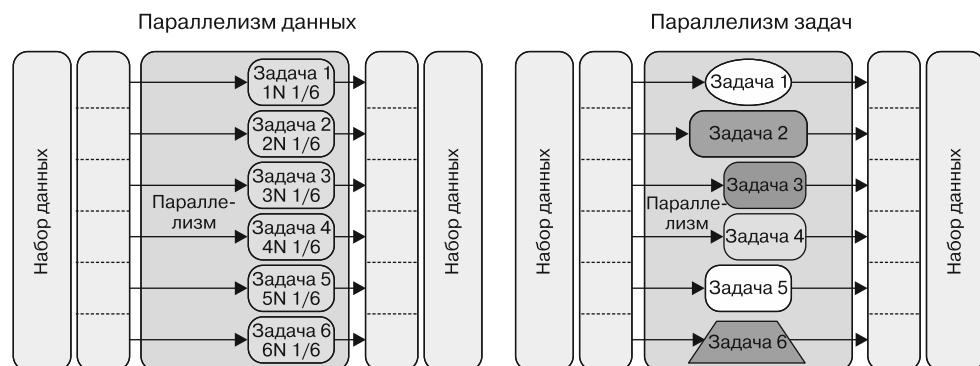


Рис. 7.1. Параллелизм данных — это одновременное выполнение одной и той же функции для разных элементов набора данных. Параллелизм задач — одновременное выполнение нескольких разных функций для одинаковых или разных наборов данных

Параллелизм задач — это не параллелизм данных

В главе 4 были показаны различия между параллелизмом задач и параллелизмом данных. Напомню: эти парадигмы находятся на противоположных концах спектра. *Параллелизм данных* возникает, когда одна операция применяется к нескольким наборам входных данных. *Параллелизм задач* возникает, когда выполняется несколько разных операций и у каждой из них — собственный набор входных данных. Параллелизм задач используется для одновременного запроса и вызова нескольких веб-API или для записи данных на разные серверы баз данных. Короче говоря, при параллелизме задач распараллеливаются функции, а при параллелизме данных — данные.

Для того чтобы достичь максимальной производительности при параллелизме задач, нужно регулировать количество выполняемых задач в зависимости от возможностей их распараллеливания в системе, что определяется количеством доступных ядер процессора и, вероятно, их текущей нагрузкой.

7.1.1. Зачем нужны параллелизм задач и функциональное программирование

В предыдущих главах были представлены примеры кода с использованием параллелизма данных и составных задач. Шаблоны распараллеливания данных, такие как «разделяй и властвуй», Fork/Join и MapReduce, направлены на решение вычислительной задачи разделения и параллельного вычисления независимых задач меньшего размера. В конце, когда все задачи завершены, их выходные данные объединяются в конечный результат.

Однако при реальном параллельном программировании мы обычно имеем дело с различными и более сложными структурами, которые не так легко разделить и уменьшить. Например, вычисление задачи, обрабатывающей входные данные, может опираться на результат других задач. В этом случае разработка и распределение работы между несколькими задачами отличаются от таких в модели параллелизма данных и могут быть очень сложными. Подобная сложность вызывается зависимостями между задачами: связи между ними могут быть запутанными, время выполнения может изменяться — и все это усложняет распределение заданий.

Параллелизм задач позволяет справиться с подобными сценариями, предоставляемыми нам, разработчикам, набор инструментов, шаблонов и в случае программирования .NET Framework — обширную библиотеку, которая упрощает программирование на основе параллелизма задач. ФП также упрощает композиционный аспект задач, позволяя держать под контролем побочные эффекты и управлять их зависимостями в стиле декларативного программирования.

Принципы функционального программирования играют важную роль в написании эффективных и детерминированных программ на основе параллелизма задач. Эти функциональные концепции обсуждались в первых главах книги. Итак, вот список рекомендаций для написания параллельного кода.

- ❑ Задачи должны состоять из функций, не имеющих побочных эффектов, что приводит к ссыпочной прозрачности и детерминированному коду. Благодаря чистым функциям программа получается более предсказуемой, так как функции всегда ведут себя одинаково независимо от внешнего состояния.
- ❑ Помните, что чистые функции могут выполняться параллельно, поскольку последовательность их выполнения не имеет значения.
- ❑ Если побочные эффекты неизбежны, управляйте ими локально, проводя вычисления в функциях с изолированным выполнением.
- ❑ Избегайте разделения данных между задачами, применяя подход защищенной копии.
- ❑ Если не удается избежать разделения данных между задачами, используйте неизменяемые структуры.

ПРИМЕЧАНИЕ

Защищенная копия — это механизм, который уменьшает (или устраняет) негативные побочные эффекты, возникающие из-за изменения общего изменяемого объекта. Идея состоит в создании копии исходного объекта, который можно безопасно использовать совместно; его изменение не влияет на исходный объект.

7.1.2. Поддержка параллелизма задач в .NET

Начиная с первой версии, .NET Framework поддерживает параллельное выполнение кода посредством многопоточности. Многопоточные программы основаны на независимых исполняемых модулях, называемых *потоками* (*thread*). Потоки представляют собой легкие процессы, обеспечивающие многозадачность в рамках одного приложения. (Класс *Thread* принадлежит пространству имен *.Threading* из библиотеки Base Class Library (BCL).) Потоки обрабатываются посредством CLR. Создание новых потоков является затратным с точки зрения вычислительных расходов и памяти. Например, размер стека памяти, необходимого для создания потока в процессоре на основе архитектуры x86, составляет около 1 Мбайт, поскольку он включает в себя стек, локальное хранилище потоков и переключения контекста.

К счастью, в .NET Framework есть класс *ThreadPool*, который помогает решить эти проблемы производительности. В сущности, он позволяет оптимизировать затраты, связанные со сложными операциями, такими как создание, запуск и уничтожение потоков.

Кроме того, .NET `ThreadPool` позволяет многократно использовать существующие потоки — чем чаще, тем лучше, — чтобы минимизировать затраты, связанные с созданием новых потоков. На рис. 7.2 показано сравнение двух процессов.

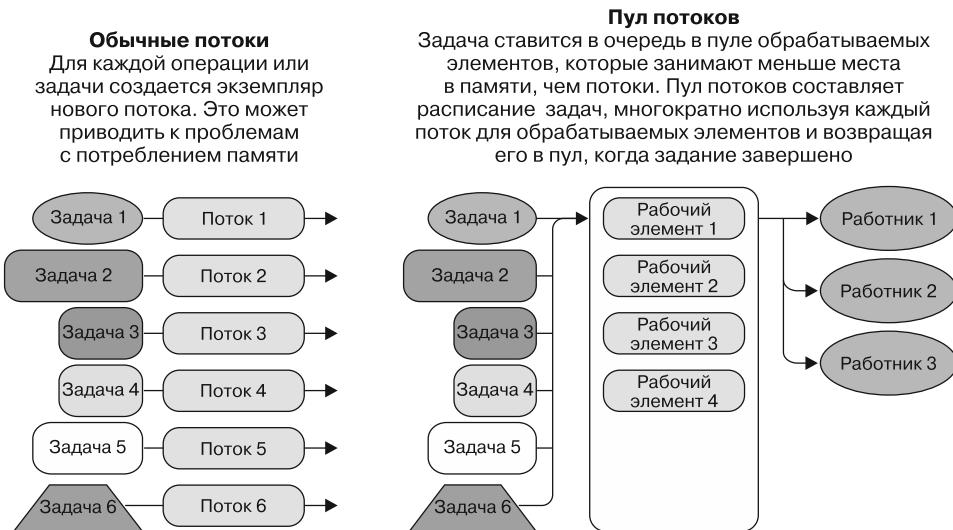


Рис. 7.2. Если использовать обычные потоки, приходится создать экземпляр нового потока для каждой операции или задачи. Это может приводить к проблемам потребления памяти. При применении пула потоков, наоборот, задача ставится в очередь в пуле обрабатываемых элементов, которые занимают в памяти значительно меньше места, чем потоки. Затем пул потоков планирует выполнение задач, многократно используя один поток для разных обрабатываемых элементов и возвращая его обратно в пул, когда выполнение задания завершено

Класс `ThreadPool`

В .NET Framework существует статический класс `ThreadPool`, загружающий набор потоков во время инициализации многопоточного приложения, а затем по мере необходимости повторно использующий эти потоки вместо того, чтобы создавать новые при запуске новых задач. Таким образом, класс `ThreadPool` ограничивает количество потоков, выполняемых в каждой точке приложения, позволяя избежать вычислительных расходов на создание и уничтожение потоков. При параллельных вычислениях `ThreadPool` оптимизирует производительность и повышает скорость отклика приложения, избегая переключений контекста.

В состав класса `ThreadPool` входит статический метод `QueueUserWorkItem`, который принимает функцию (делегат), представляющую асинхронную операцию.

В листинге 7.1 сравнивается обычный запуск потока и запуск потока с помощью статического метода `ThreadPool.QueueUserWorkItem`.

Листинг 7.1. Порождение потоков с помощью ThreadPool.QueueUserWorkItem

```

Action<string> downloadSite = url => {
    var content = new WebClient().DownloadString(url);
    Console.WriteLine($"The size of the web site {url} is
→ {content.Length}");
};

var threadA = new Thread(() => downloadSite("http://www.nasdaq.com"));
var threadB = new Thread(() => downloadSite("http://www.bbc.com"));

threadA.Start();
threadB.Start();
threadA.Join();
threadB.Join(); ← Потоки должны запускаться явно и давать
                    возможность ожидать (присоединяться) для завершения

ThreadPool.QueueUserWorkItem(o => downloadSite("http://www.nasdaq.com"));
ThreadPool.QueueUserWorkItem(o => downloadSite("http://www.bbc.com")); ←
                    Метод ThreadPool.QueueUserWorkItem сразу запускает
                    операцию в стиле «запустить и забыть» — это означает,
                    что рабочий элемент должен создавать побочные эффекты,
                    чтобы результаты вычислений стали видимыми

```

Поток запускается явно, но в классе `Thread` есть возможность дождаться потока, используя метод экземпляра `Join`. Однако каждый поток создает дополнительную нагрузку на память, что вредно для среды выполнения. Запуск асинхронных вычислений с применением `ThreadPool QueueUserWorkItem` выполняется просто, но у этой технологии есть несколько ограничений, из-за которых она вызывает серьезные сложности при разработке параллельных систем на основе задач:

- ❑ нет встроенного механизма уведомлений о завершении асинхронных операций;
- ❑ нет простого способа вернуть результат выполнения обработчика `ThreadPool`;
- ❑ нет встроенного механизма распространения исключений вплоть до исходного потока;
- ❑ отсутствует простой способ координации зависимых асинхронных операций.

Чтобы преодолеть эти ограничения, Microsoft вводит в TPL понятие задач, доступных через пространство имен `System.Threading.Tasks`. Концепция задач является рекомендуемым подходом для построения параллельных систем на основе задач в .NET.

7.2. Библиотека параллельных задач в .NET

В библиотеке TPL, входящей в состав .NET, реализован ряд дополнительных оптимизаций `ThreadPool`, включая сложный алгоритм захвата задачи `TaskScheduler` (<http://mng.bz/j4K1>) для динамического масштабирования степени конкурентности, как показано на рис. 7.3. Этот алгоритм гарантирует эффективное использование доступных процессорных ресурсов системы для достижения максимальной общей производительности конкурентного кода.

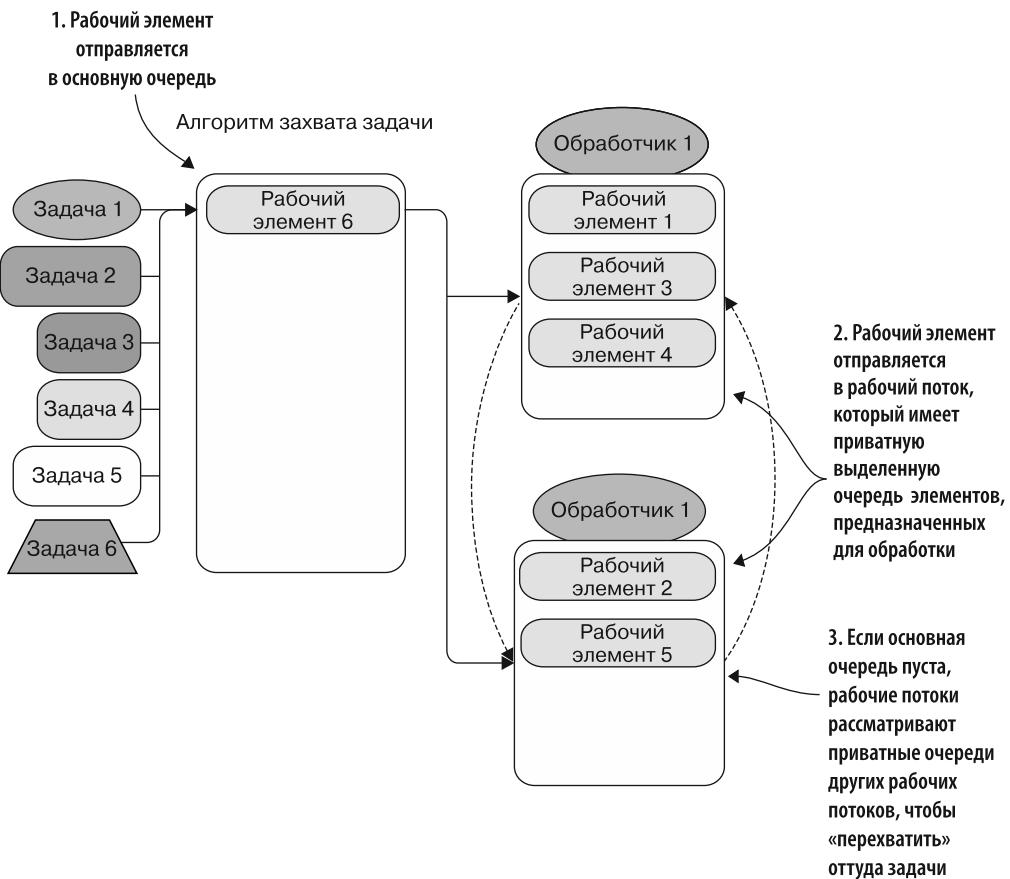


Рис. 7.3. Для оптимизации планировщика в библиотеке TPL используется алгоритм захвата задач. Сначала рабочие элементы ставятся в основную очередь (шаг 1). Затем они отправляются в один из потоков обработчиков, у которого есть приватная выделенная очередь рабочих элементов, предназначенных для обработки (шаг 2). Если основная очередь пуста, обработчики заглядывают в приватные очереди других обработчиков, чтобы «перехватить» оттуда задачи (шаг 3)

После появления в Microsoft TPL концепции на основе задач вместо традиционной ограниченной модели на основе потоков благодаря появлению ряда новых типов упростился процесс реализации в программах конкурентности и параллелизма. Кроме того, благодаря объекту `Task` TPL обеспечивает возможность отмены состояния и управления состоянием, что позволяет обрабатывать и распространять исключения и управлять выполнением рабочих потоков. TPL абстрагирует детали реализации от разработчика, предоставляя ему средства контроля над параллельным выполнением кода.

Использование модели программирования на основе задач позволяет практически без усилий вводить параллелизм в программу и конкурентно выполнять части кода, преобразовав их в задачи.

ПРИМЕЧАНИЕ

TPL предоставляет необходимую инфраструктуру для оптимального использования процессорных ресурсов независимо от того, выполняется параллельная программа на одно- или многоядерном компьютере.

Есть несколько способов вызвать параллельные задачи. В этой главе рассматриваются соответствующие методы реализации параллелизма на основе задач.

7.2.1. Выполнение параллельных операций с помощью Parallel.Invoke из библиотеки TPL

Используя .NET TPL, можно составить расписание выполнения задач несколькими способами, самым простым из которых является метод `Parallel.Invoke`. Этот метод принимает в качестве аргумента произвольное количество действий (делегатов) в виде `ParamArray` и создает задачу для каждого переданного делегата. К сожалению, сигнатура действия-делегата не имеет входных аргументов и возвращает `void`, что противоречит принципам функционального программирования. В императивных языках программирования функции, возвращающие `void`, используются для обработки побочных эффектов.

После завершения всех задач метод `Parallel.Invoke` возвращает управление основному потоку, и продолжается выполнение этого потока. Важной особенностью метода `Parallel.Invoke` является то, что обработка исключений, синхронный вызов и планирование выполняются незаметно для разработчика.

Рассмотрим сценарий, в котором требуется выполнить параллельно несколько независимых, гетерогенных задач, а затем, после их завершения, продолжить работу. К сожалению, PLINQ и параллельные циклы (рассмотренные в предыдущих главах) в данном случае неприменимы, поскольку они не поддерживают гетерогенные операции. Это типичный случай для использования метода `Parallel.Invoke`.

ПРИМЕЧАНИЕ

Гетерогенные задачи представляют собой набор операций, выполняемых в общем случае независимо друг от друга и имеющих разные результаты или разные типы результатов.

В листинге 7.2 параллельно выполняются несколько функций, обрабатывающих три входных изображения, после чего результат сохраняется в файловой системе. Чтобы избежать нежелательных изменений, каждая функция создает локальную защищенную копию входного изображения. Представленный ниже пример кода написан на F#; но данная концепция применима ко всем языкам программирования .NET.

В этом коде `Parallel.Invoke` создает и запускает три независимые задачи, по одной для каждой функции, и блокирует выполнение основного потока, пока не завершится выполнение всех задач. Вследствие достигаемого параллелизма общее время выполнения равно времени выполнения самого медленного метода.

Листинг 7.2. Выполнение нескольких гетерогенных задач с помощью Parallel.Invoke

```

→ let convertImageTo3D (sourceImage:string) (destinationImage:string) =
    let bitmap = Bitmap.FromFile(sourceImage) :?> Bitmap ← Создание копии
    let w,h = bitmap.Width, bitmap.Height
    for x in 20 .. (w-1) do
        for y in 0 .. (h-1) do
            let c1 = bitmap.GetPixel(x,y) ← Вложенные циклы для доступа
            let c2 = bitmap.GetPixel(x - 20,y) к пикселям изображения
            let color3D = Color.FromArgb(int c1.R, int c2.G, int c2.B)
            bitmap.SetPixel(x - 20 ,y,color3D)
    bitmap.Save(destinationImage, ImageFormat.Jpeg) ← Сохранение созданного
                                                    изображения
                                                    в файловой системе

Добавление 3D-эффекта ко входному изображению
Создание изображения, цвета которого преобразованы в оттенки серого
→ let setGrayscale (sourceImage:string) (destinationImage:string) =
    let bitmap = Bitmap.FromFile(sourceImage) :?> Bitmap ←
    let w,h = bitmap.Width, bitmap.Height
    for x = 0 to (w-1) do
        for y = 0 to (h-1) do ← Вложенные циклы для доступа
            let c = bitmap.GetPixel(x,y) к пикселям изображения
            let gray = int(0.299 * float c.R + 0.587 * float c.G + 0.114 *
→ float c.B)
            bitmap.SetPixel(x,y, Color.FromArgb(gray, gray, gray))
    bitmap.Save(destinationImage, ImageFormat.Jpeg) ← Создание изображения
                                                    с фильтрацией красного цвета

→ let setRedscale (sourceImage:string) (destinationImage:string) =
    let bitmap = Bitmap.FromFile(sourceImage) :?> Bitmap ←
    let w,h = bitmap.Width, bitmap.Height
    for x = 0 to (w-1) do
        for y = 0 to (h-1) do ← Вложенные циклы для доступа
            let c = bitmap.GetPixel(x,y) к пикселям изображения
            bitmap.SetPixel(x,y, Color.FromArgb(int c.R,
→ abs(int c.G - 255), abs(int c.B - 255)))
    bitmap.Save(destinationImage, ImageFormat.Jpeg)

System.Threading.Tasks.Parallel.Invoke(
    Action(fun ()-> convertImageTo3D "MonaLisa.jpg" "MonaLisa3D.jpg"),
    Action(fun ()-> setGrayscale "LadyErmine.jpg" "LadyErmineRed.jpg"),
    Action(fun ()-> setRedscale "GinevraBenci.jpg" "GinevraBenciGray.jpg"))

Сохранение созданного изображения
в файловой системе

```

ПРИМЕЧАНИЕ

В этом коде для изменения растрового изображения намеренно использованы методы GetPixel и SetPixel. Как известно, это медленные методы (особенно GetPixel); но в данном примере мы как раз и хотим протестировать параллелизм, генерируя небольшие дополнительные накладные расходы, чтобы создать дополнительную нагрузку на процессор. В реальной программе, если нужно обработать пиксели всего изображения, лучше преобразовать все изображение в байтовый массив и проитерировать его.

Интересный нюанс: метод `Parallel.Invoke` можно использовать для реализации шаблона Fork/Join, в котором несколько операций выполняются параллельно, а затем, когда все они будут завершены, их результаты объединяются. На рис. 7.4 показаны изображения до и после обработки.



Рис. 7.4. Изображения, полученные в результате обработки с помощью кода, представленного в листинге 7.2. Полную реализацию этого примера вы найдете в исходном коде к книге

Несмотря на удобство параллельного выполнения нескольких задач, `Parallel.Invoke` ограничивает управление параллельным выполнением вследствие сигнатуры с типом `void`. Такой метод не возвращает ресурсов, из которых можно было бы извлечь сведения о состоянии и результате выполнения каждой отдельной задачи, как успешном, так и неудачном. `Parallel.Invoke` может либо завершиться успешно, либо генерировать исключение в виде экземпляра `AggregateException`. В последнем случае любое исключение, возникшее во время выполнения, будет отложено и заново выдано после завершения всех задач. С точки зрения ФП исключения — это побочные эффекты, которых следует избегать. Поэтому ФП предоставляет лучший механизм обработки ошибок, и он будет рассмотрен в главе 11.

В сущности, есть два важных ограничения, которые следует учитывать при использовании метода `Parallel.Invoke`:

- сигнатура метода возвращает `void`, что не дает возможности применять компоновку;
- последовательность выполнения задач не гарантирована, что ограничивает разработку вычислений с зависимостями.

7.3. Проблема `void` в C#

В императивных языках программирования, таких как C#, обычно используются методы и делегаты, которые не возвращают значения (`void`), например метод `Parallel.Invoke`. Сигнатура таких методов не позволяет применять их при компоновке. Две функции могут быть скомпонованы только в том случае, если формат выхода одной из них соответствует формату входа другой.

В изначально функциональных языках программирования, таких как F#, у каждой функции есть возвращаемое значение, в том числе специальный случай с типом `unit`, который подобен `void`, но рассматривается как значение, концептуально практически не отличающееся от логического или целого.

`Unit` — это тип любого выражения, если у него нет какого-либо другого определенного значения; например, если функция только выводит данные на экран. Такая функция не должна возвращать никаких конкретных результатов, поэтому подобные функции могут возвращать `unit`, чтобы код оставался валидным. Данный тип в F# — эквивалент `void` в C#. Причина, по которой в F# не используется `void`, состоит в том, что каждый валидный фрагмент кода должен иметь тип возвращаемого значения, тогда как `void` означает отсутствие возвращаемого значения. Программист, пишущий код на функциональном языке, вместо концепции `void` подумает о `unit`. В F# тип `unit` записывается как `()`. Эта конструкция обеспечивает компонуемость функций.

В принципе, язык программирования не обязательно должен поддерживать методы с возвращаемыми значениями. Но метод без определенного типа возвращаемого значения (`void`) предполагает, что данная функция имеет некий побочный эффект, и это затрудняет параллельное выполнение задач.

Решение проблемы void в C #: тип unit. В функциональном программировании функция определяет отношения между входными и выходными значениями. Это похоже на формулировку математических теорем. Например, в случае чистой функции возвращаемое значение определяется только его входными значениями.

В математике каждая функция возвращает значение. В ФП функция является отображением, а у отображения должно быть значение, которое оно будет отображать. В основных императивных языках программирования, таких как C#, C++ и Java, эта концепция отсутствует. Эти языки рассматривают функции, возвращающие `void`, как методы, ничего не возвращающие, а не как функции, которые могут вернуть нечто значимое.

В C# можно реализовать тип `Unit` в виде `struct` с единственным значением, используемым как возвращаемый тип для метода, который иначе возвращал бы `void`. В библиотеке Rx, описанной в главе 6, применяется другой подход: здесь тип `unit` представлен как часть библиотеки. В листинге 7.3 показана реализация типа `Unit` на C#, заимствованная из Microsoft Rx.

Листинг 7.3. Реализация типа Unit в C#

```

Вспомогательный статический метод
для получения экземпляра типа Unit
public struct Unit : IEquatable<Unit> ← Структура Unit, которая реализует интерфейс
{
    public static readonly Unit Default = new Unit(); ← IEquatable, чтобы создать описание
    public override int GetHashCode() => 0; ← типаориентированного метода
    public override bool Equals(object obj) => obj is Unit; ← для определения равенства
    public override string ToString() => "("; ←
    public bool Equals(Unit other) => true; ← Переопределение базовых
    public static bool operator ==(Unit lhs, Unit rhs) => true; ← методов для описания
    public static bool operator !=(Unit lhs, Unit rhs) => false; } ← операции равенства
                                                                значений типа Unit
                                                                Два значения типа Unit
                                                                всегда равны между собой

```

Структура `Unit` реализует интерфейс `IEquatable` таким образом, чтобы все значения типа `Unit` были равны между собой. Но какова реальная выгода от использования типа `Unit` в качестве значения в системе языковых типов? Каково его практическое применение?

Бот два основных ответа на этот вопрос.

- ❑ Тип `Unit` может использоваться для публикации подтверждения о том, что выполнение функции завершено.
- ❑ Тип `Unit` полезен для написания параметризованного кода, включая случай, когда требуется параметризованная функция первого класса, что уменьшает дублирование.

Например, тип `Unit` позволяет избежать дублирования кода при реализации `Action<T>`, `Func<T, R>` или функций, возвращающих `Task` или `Task<T>`. Рассмотрим

функцию, которая выполняет `Task<TInput>` и преобразует результат вычислений в тип `TResult`:

```
 TResult Compute<TInput, TResult>(Task<TInput> task,
                                     Func<TInput, TResult> projection) => projection(task.Result);

 Task<int> task = Task.Run<int>(() => 42);
 bool isTheAnswerOfLife = Compute(task, n => n == 42);
```

У этой функции два аргумента. Первый из них — `Task<TInput>`, который определяет выражение. Результат передается во второй аргумент, делегат `Func<TInput, TResult>`, чтобы применить преобразование и вернуть окончательное значение.

ПРИМЕЧАНИЕ

Данная реализация кода предназначена только для демонстрационных целей. Блокировать выполнение задачи для получения результата, как это делает функция `Compute` в предыдущем фрагменте кода, не рекомендуется. Правильный подход будет рассмотрен в разделе 7.4.

Как бы вы преобразовали функцию `Compute` так, чтобы она выводила результат? Вместо проекции делегата `Func<T>` вам пришлось бы написать новую функцию, с типом делегата `Action`. Вот сигнатура нового метода:

```
 void Compute<TInput>(Task<TInput> task, Action<TInput> action) =>
    action(task.Result);

 Task<int> task = Task.Run<int>(() => 42);
 Compute(task, n => Console.WriteLine($"Is {n} the answer of life?
➥ {n == 42}"));
```

Также важно отметить, что тип делегата `Action` имеет побочный эффект: в данном случае выводит результат в консоль, то есть эта функция концептуально аналогична предыдущей.

В идеале было бы повторно использовать ту же функцию, а не дублировать код для функции с типом делегата `Action` в качестве аргумента. Для этого нужно передать делегату `Func` тип `void`, что в C# невозможно. Это тот случай, когда тип `Unit` позволяет избавиться от дублирования кода. Благодаря определению типа `struct Unit` можно задействовать уже имеющуюся функцию, которая принимает делегат `Func`, чтобы описать то же поведение, что и функция с типом делегата `Action`:

```
 Task<int> task = Task.Run<int>(() => 42);

 Unit unit = Compute(task, n => {
    Console.WriteLine($"Is {n} the answer of life? {n == 42}");
    return Unit.Default;});
```

Таким образом, вводя тип `Unit`, можно написать на C# единую функцию `Compute` для обработки обоих случаев — и возврата значения, и вычисления побочного эффекта. В итоге функция, возвращающая тип `Unit`, указывает на наличие побочных эффектов, что является важной информацией при написании конкурентного кода.

Кроме того, в ФП существуют такие языки, как Haskell, где тип `Unit` позволяет компилятору различать чистые и нечистые функции, чтобы применять более детальную оптимизацию.

7.4. Стиль продолжений: функциональный поток управления

В основе принципа продолжения задач лежит функциональная идея парадигмы CPS, обсуждавшейся в главе 3. Такой подход дает возможность управлять выполнением программы в виде продолжения, передавая результат текущей функции в следующую функцию. По сути, продолжение функции — это делегат, представляющий «то, что будет дальше». Стиль CPS является альтернативой традиционному потоку управления в императивном стиле программирования, где все команды выполняются по очереди, одна за другой. Вместо этого при использовании CPS функция передается методу в качестве аргумента, явно определяя следующую операцию, которая должна быть выполнена после завершения самого данного метода. Это позволяет создавать собственные команды управления потоком.

7.4.1. Зачем нужен CPS

Главным преимуществом применения CPS в конкурентной среде является возможность избежать неудобной блокировки потоков, которая отрицательно сказывается на производительности программы. Например, неэффективно заставлять метод ждать завершения одной или нескольких задач, блокируя основной поток выполнения до тех пор, пока не будут завершены его дочерние задачи. Часто родительская задача, которая в таком случае является основным потоком, могла бы продолжаться, но сразу это сделать невозможно, потому что ее поток все еще выполняет другую задачу. CPS решает подобную проблему, так как позволяет потоку сразу вернуться к вызывающей задаче, не дожидаясь завершения выполнения дочерних элементов. Таким образом гарантируется, что продолжение будет вызвано сразу после завершения.

Единственный недостаток явного использования CPS заключается в том, что сложность кода может быстро возрастать, так как CPS делает программы более длинными и менее понятными. Далее в этой главе вы узнаете, как бороться с подобной проблемой, объединив TPL и функциональные парадигмы и таким образом абстрагировав сложность кода, сделав его гибким и простым в применении. CPS имеет следующие полезные преимущества:

- ❑ продолжение функций может быть скомпоновано в виде цепочки операций;
- ❑ в продолжении можно указать условия, при которых вызывается функция;
- ❑ функция-продолжение может вызывать другие продолжения;
- ❑ функцию-продолжение можно легко отменить в любой момент во время выполнения или даже до запуска.

В .NET Framework задача является абстракцией классического (традиционного) потока .NET (<http://mng.bz/DK6K>), представляющего собой независимый асинхронный рабочий блок. Объект Task принадлежит пространству имен `System.Threading.Tasks`. Более высокий уровень абстракции, предоставляемый типом `Task`, упрощает реализацию конкурентного кода и облегчает управление жизненным циклом для каждой операции задачи. Например, можно проверить состояние вычисления и убедиться, что операция завершена, удалена или отменена. Кроме того, задачи можно компоновать в цепочки операций с использованием продолжений, что допускает декларативное и текущее программирование.

В листинге 7.4 показано, как создавать и выполнять операции с применением типа `Task`. В коде задействованы функции из листинга 7.2.

Листинг 7.4. Создание и запуск задач

```
Запуск метода convertImageTo3D с использованием
статического вспомогательного
метода StartNew Task
Task monaLisaTask = Task.Factory.StartNew(() =>
    convertImageTo3D("MonaLisa.jpg", "MonaLisa3D.jpg")); ←

Task ladyErmineTask = new Task(() =>
    setGrayscale("LadyErmine.jpg", "LadyErmine3D.jpg"));
ladyErmineTask.Start(); ←
Запуск метода setGrayscale путем
создания нового экземпляра Task,
а затем вызова метода Start()
экземпляра Task
Task ginevraBenciTask = Task.Run(() =>
    setRedscale("GinevraBenci.jpg", "GinevraBenci3D.jpg")); ←
Запуск метода setRedscale с использованием упрощенного статического метода Run(),
который запускает задачу с общими свойствами по умолчанию
```

В этом коде показаны три способа создания и выполнения задачи.

- ❑ Первый способ создает и сразу запускает новую задачу, используя встроенный конструктор метода `Task.Factory.StartNew`.
- ❑ Второй способ создает новый экземпляр задачи, которому нужно указать функцию в качестве параметра конструктора — эта функция является телом задачи. Затем, чтобы начать вычисление, `Task` вызывает метод экземпляра `Start`. Этот способ более гибкий: он позволяет отложить выполнение задачи до вызова функции `Start`; таким образом, объект `Task` может быть передан другому методу, который сам решит, когда запланировать выполнение задачи.
- ❑ Третий способ создает объект `Task` и сразу вызывает метод `Run` для планирования задачи. Это удобный способ создания задач и работы с ними с использованием стандартного конструктора, который применяет стандартные значения параметров.

Первые два способа лучше всего подходят, если нужен конкретный вариант создания экземпляра задачи, например, с параметром `LongRunning`. Как правило, задачи способствуют естественному способу изолирования данных, которые зависят от функций и коммуникации через связанные с ними входные и выходные значения, как показано в схематическом примере на рис. 7.5.



Рис. 7.5. Когда две задачи скомпонованы, то выходные данные первой задачи становятся входными для второй. Это касается и компоновки функций

ПРИМЕЧАНИЕ

Объект Task может быть создан с разными параметрами, позволяющими управлять этим объектом и определяющими его поведение. Например, параметр `TaskCreationOptions.LongRunning` уведомляет основного планировщика о том, что задача будет долговременной. В этом случае может быть создан дополнительный выделенный поток, в обход планировщика пула потоков, и планировщик пула потоков не будет влиять на работу данного потока. Подробнее о `TaskCreationOptions` читайте в документации Microsoft MSDN, доступной онлайн (<http://bit.ly/2uxg1R6>).

7.4.2. Ожидание завершения задачи: модель продолжения

Мы видели, как можно использовать задачи для распараллеливания независимых рабочих блоков. Но обычно структура кода сложнее, чем просто выполнение операций в режиме «запустить и забыть». Большинство параллельных вычислений на основе задач требуют более сложной координации между конкурентными операциями, где на последовательность выполнения влияют базовые алгоритмы и поток управления программой. К счастью, библиотека .NET TPL предоставляет механизмы координации задач.

Начнем с примера последовательного выполнения нескольких операций и поэтапно перепроектируем и реорганизуем программу, чтобы улучшить компонуемость кода и повысить производительность. Мы начнем с последовательной реализации, а затем постепенно применим различные технологии, чтобы получить максимальную общую вычислительную производительность.

В листинге 7.5 реализована программа распознавания лиц, которая позволяет выделять определенные лица из заданного изображения. В этом примере мы возьмем изображения президентов США на 20-, 50- и 100-долларовых купюрах, используя ту их сторону, где напечатано изображение президента. Программа будет распознавать лицо президента на каждом изображении и возвращать новое изображение, на котором обнаруженнное лицо заключено в квадратную рамку. В данном примере мы сосредоточимся на важных особенностях кода, не отвлекаясь на детали реализации пользовательского интерфейса. Полный исходный кодложен на сайте издательства.

Листинг 7.5. Функция распознавания лиц на C#

Классификатор для распознавания лиц в изображении

```
Bitmap DetectFaces(string fileName) {
    var imageFrame = new Image<Bgr, byte>(fileName); ← Экземпляр изображения Emgu.CV
    var cascadeClassifier = new CascadeClassifier(); ← для взаимодействия
    var grayframe = imageFrame.Convert<Gray, byte>(); ← с библиотекой OpenCV
    var faces = cascadeClassifier.DetectMultiScale(
        grayframe, 1.1, 3, System.Drawing.Size.Empty); ← Процесс
    foreach (var face in faces) ← распознавания лица
        imageFrame.Draw(face,
            new Bgr(System.Drawing.Color.BurlyWood), 3); ← Выделение распознанного лица (лиц)
    return imageFrame.ToBitmap(); ← путем заключения его в квадратную рамку
}

void StartFaceDetection(string imagesFolder) {
    var filePaths = Directory.GetFiles(imagesFolder);
    foreach (string filePath in filePaths) {
        var bitmap = DetectFaces(filePath);
        var bitmapImage = bitmap.ToBitmapImage(); ← Обработанное изображение
        Images.Add(bitmapImage); ← добавляется в наблюдаемую коллекцию Images, чтобы обновить
    } ← пользовательский интерфейс
}
```

Функция `DetectFaces` загружает изображение из файловой системы, используя заданный путь и имя файла, а затем распознает наличие в нем каких-либо лиц. За распознавание лиц отвечает библиотека `Emgu.CV`. Библиотека `Emgu.CV` — это оболочка .NET, которая обеспечивает согласованность с языками программирования, такими как C# и F#. Оба данных языка могут взаимодействовать с базовой библиотекой обработки изображений Intel OpenCV и вызывать ее функции¹. Функция `StartFaceDetection` инициирует выполнение, получает путь к обрабатываемым изображениям в файловой системе, а затем последовательно выполняет распознавание лиц в цикле `for-each`, вызывая функцию `DetectFaces`. Результатом является новый тип `BitmapImage`, который добавляется к наблюдаемой коллекции `Images` для обновления пользовательского интерфейса. На рис. 7.6 показан ожидаемый результат — обнаруженные лица заключены в рамки.

Первым шагом для повышения производительности программы будет параллельное выполнение функции распознавания лиц, при котором для каждого обрабатываемого изображения создается новая задача (листинг 7.6).

В этом коде выражение LINQ создает коллекцию `IEnumerable` из `Task<Bitmap>`, которая построена с помощью удобного метода `Task.Run`. После построения коллекции задач программа запускает независимое выполнение в цикле `for-each`; но от этого производительность не улучшается. Проблема в том, что задачи все равно

¹ OpenCV (Open Source Computer Vision Library, библиотека компьютерного зрения с открытым исходным кодом) — высокопроизводительная библиотека обработки изображений от Intel (<https://opencv.org>).

выполняются последовательно, одна за другой. Цикл обрабатывает задачи по одной, ожидая завершения предыдущей, прежде чем перейти к следующей. Этот код не работает параллельно.



Рис. 7.6. Результат процесса распознавания лиц. Справа показаны изображения с распознанными лицами, заключенными в квадратную рамку

Листинг 7.6. Реализация программы распознавания лиц с распараллеливанием задач

```
void StartFaceDetection(string imagesFolder)
{
    var filePaths = Directory.GetFiles(imagesFolder);

    var bitmaps = from filePath in filePaths
                  select Task.Run<Bitmap>(() => DetectFaces(filePath)); ←
    foreach (var bitmap in bitmaps) {
        var bitmapImage = bitmap.Result;
        Images.Add(bitmapImage.ToBitmapImage());                         | Последовательный запуск
    }                                                               | задач из TPL для обработки
}                                                               | каждого изображения
```

Вы можете возразить, что если выбрать другой подход, например использовать `Parallel.ForEach` или `Parallel.Invoke` для вычисления функции `DetectFaces`, то можно избежать подобной проблемы и гарантировать параллелизм. Но, сделав так, вы поймете, почему это не очень хорошая идея.

Перестроим программу так, чтобы это исправить, но прежде проанализируем, в чем заключается основная проблема. Коллекция `IEnumerable` из `Task<Bitmap>`, сгенерированная с помощью LINQ-выражения, материализуется во время выполнения цикла `for-each`. На каждой итерации извлекается `Task<Bitmap>`, но здесь задача

не выполняется и даже не запускается. Дело в том, что коллекция `IEnumerable` рассчитана на ленивое выполнение, поэтому вычисление основной задачи начинается в последний момент, во время ее материализации. Соответственно, когда результат задачи по обработке растрового изображения внутри цикла становится доступным через свойство `Task<Bitmap>.Result`, задача блокирует присоединяющийся поток до окончания ее выполнения. Выполнение возобновится после того, как задача завершит вычисление и вернет результат.

Чтобы написать масштабируемое программное обеспечение, нужно убрать оттуда блокируемые потоки. В предыдущем коде при доступе к ресурсу `Result` задачи пул потоков, скорее всего, создаст новый поток, поскольку задача еще не завершена. Это увеличивает потребление ресурсов и ухудшает производительность.

После подобного анализа становится ясным, что для обеспечения параллельности необходимо предпринять следующее (рис. 7.7):

- убедиться, что задачи выполняются параллельно;
- избегать блокировки основного рабочего потока и ожидания завершения каждой задачи.

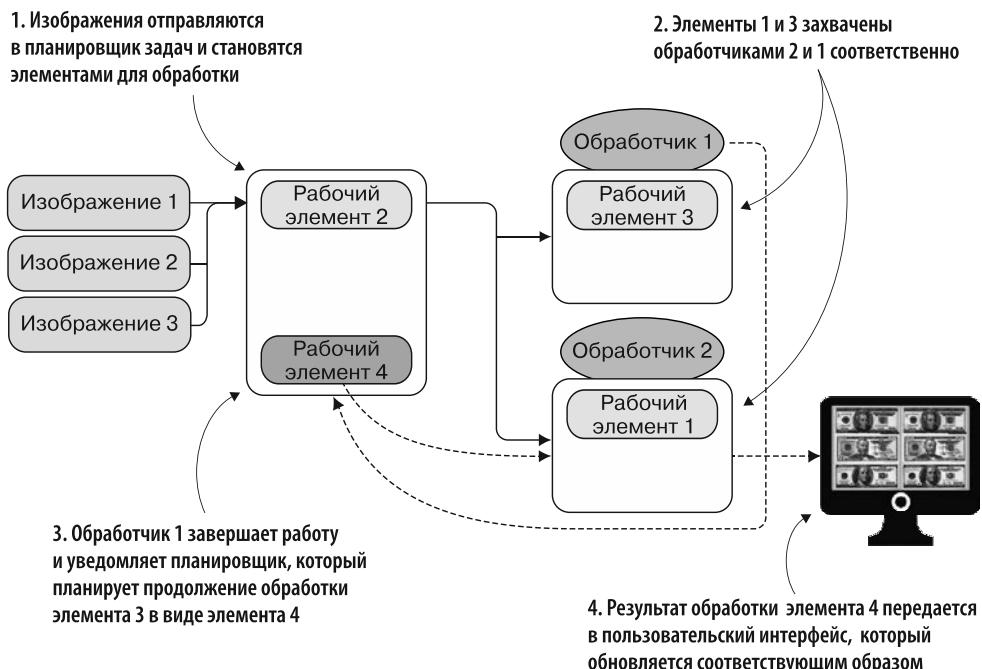


Рис. 7.7. Изображения передаются в планировщик задач, становясь элементами для обработки (шаг 1). Затем элементы 3 и 1 «захватываются» обработчиками 1 и 2 соответственно (шаг 2).

Обработчик 1 завершает работу и уведомляет планировщик задач, который распределяет оставшуюся часть работы для продолжения в виде нового элемента 4, являющегося продолжением элемента 3 (шаг 3). После обработки элемента 4 результат передается в пользовательский интерфейс, который обновляется соответствующим образом (шаг 4)

Вот как можно исправить проблемы, чтобы обеспечить параллельное выполнение кода и сократить потребление памяти (листинг 7.7).

Листинг 7.7. Правильная реализация функции DetectFaces с параллельным выполнением задач

```
ThreadLocal<CascadeClassifier> CascadeClassifierThreadLocal =
    new ThreadLocal<CascadeClassifier>(() => new CascadeClassifier()); ←

Bitmap DetectFaces(string fileName) {
    var imageFrame = new Image<Bgr, byte>(fileName); ←
    var cascadeClassifier = CascadeClassifierThreadLocal.Value; ←
    var grayframe = imageFrame.Convert<Gray, byte>(); ←
    var faces = cascadeClassifier.DetectMultiScale(grayframe, 1.1, 3, ←
        System.Drawing.Size.Empty); ←

    foreach (var face in faces)
        imageFrame.Draw(face, new Bgr(System.Drawing.Color.BurlyWood), 3);
    return imageFrame.ToBitmap();
} ←

void StartFaceDetection(string imagesFolder) {
    var filePaths = Directory.GetFiles(imagesFolder); ←
    var bitmapTasks = ←
        (from filePath in filePaths
         select Task.Run<Bitmap>(() => DetectFaces(filePath))).ToList(); ←
        LINQ-выражение принимает
        путь к файлам и запускает
        параллельную обработку изображений ←

    foreach (var bitmapTask in bitmapTasks)
        bitmapTask.ContinueWith(bitmap => { ←
            var bitmapImage = bitmap.Result;
            Images.Add(bitmapImage.ToBitmapImage());
        }, TaskScheduler.FromCurrentSynchronizationContext()); ←
} ←

TaskScheduler FromCurrentSynchronizationContext ←
 выбирает контекст для планирования обработки
 с соответствующим пользовательским интерфейсом ←
Продолжение задачи гарантирует
 отсутствие блокировок; операция допускает
 продолжение работы до своего завершения ←
```

В данном примере, из соображений простоты структуры кода, мы исходили из предположения, что все вычисления завершаются успешно. Иначе потребовалось бы несколько изменить код, но, к счастью, при настоящих параллельных вычислениях для этого не потребуется блокировка потоков (путем продолжения выполнения задачи после завершения других задач). Основная функция `StartFaceDetection` гарантирует параллельное выполнение задач, материализуя LINQ-выражение сразу после вызова `ToList()` для коллекции `IEnumerable`, состоящей из элементов `Task<Bitmap>`.

ПРИМЕЧАНИЕ

Разрабатывая программу вычислений большого количества задач, запустите LINQ-запрос и проследите, чтобы он материализовался сразу. Иначе от него не будет никакой пользы, поскольку параллелизм будет потерян и задачи будут вычисляться последовательно в цикле `for-each`.

Затем с помощью объекта `ThreadLocal` создается защитная копия `CascadeClassifier` для каждого потока, обращающегося к функции `DetectFaces`. `CascadeClassifier` загружает в память локальный ресурс, который не является потокобезопасным. Чтобы обеспечить потокобезопасность, для каждого потока создается локальная переменная `CascadeClassifier`, которая и выполняет данную функцию. В этом заключается назначение объекта `ThreadLocal` (подробнее данный вопрос обсуждался в главе 4).

Затем в функции `StartFaceDetection` в цикле `for-each` для каждой задачи из списка `Task<Bitmap>` создается продолжение, вместо того чтобы блокировать ее выполнение в случае, если задача не завершена. Поскольку `bitmapTask` является асинхронной операцией, то нет гарантии, что выполнение задачи завершится до обращения к свойству `Result`. При продолжении задачи рекомендуется использовать функцию `ContinueWith` для доступа к результату как части продолжения. Определение продолжения задачи аналогично созданию обычной задачи, но функция, переданная в метод `ContinueWith`, принимает в качестве аргумента тип `Task<Bitmap>`. Этот аргумент представляет собой предыдущую задачу, которая может быть применена для проверки состояния вычислений и соответствующего ветвления.

После завершения предыдущей задачи функция `ContinueWith` начинает выполняться как новая задача. Продолжение задачи выполняется в захваченном текущем контексте синхронизации — `TaskScheduler.FromCurrentSynchronizationContext`, который автоматически выбирает соответствующий контекст для планирования работы с соответствующим потоком пользовательского интерфейса.

ПРИМЕЧАНИЕ

При вызове функции `ContinueWith` можно организовать запуск новой задачи только в том случае, если первая задача завершится с определенными условиями, — например, будет отменена (после установки флага `TaskContinuationOptions.OnlyOnCanceled`) или если возникнет исключение (после установки флага `TaskContinuationOptions.OnlyOnFaulted`).

Как уже отмечалось, мы могли бы использовать `Parallel.ForEach`, но проблема в том, что при таком подходе программа ждет, пока будут закончены все операции, и только потом продолжает работу, отчего блокируется основной поток. Более того, при этом становится сложнее напрямую обновлять пользовательский интерфейс, так как операции выполняются в разных потоках.

7.5. Стратегии компоновки операций для выполнения задач

Продолжения — это самое главное преимущество TPL. Например, можно выполнить несколько продолжений одной и той же задачи и создать цепочку продолжений задач, звенья которой имеют зависимости между собой. Более того, базовый планировщик может, задействуя продолжения задач, в полной мере воспользоваться механизмом перехвата заданий и оптимизировать планирование с учетом доступных ресурсов в процессе выполнения программы.

Давайте используем продолжение задач в примере с распознаванием лиц. Итоговая версия кода работает параллельно, обеспечивая резкий рост производительности. Но программу можно дополнительно оптимизировать, повысив масштабируемость. Функция `DetectFaces` выполняет ряд операций последовательно, как цепочку вычислений. Чтобы улучшить реализацию ресурсов и повысить общую производительность, целесообразно разделить задачи и последующие их продолжения для каждой операции `DetectFaces` и выполнять их в отдельных потоках.

Чтобы использовать продолжения задач, нужно внести в код простые изменения. В листинге 7.8 показана новая функция `DetectFaces`, где каждый шаг алгоритма распознавания лиц выполняется в виде отдельной, независимой задачи.

Листинг 7.8. Функция DetectFaces с использованием продолжения задач

```
Task<Bitmap> DetectFaces(string fileName)
{
    var imageTask = Task.Run<Image<Bgr, byte>>(
        () => new Image<Bgr, byte>(fileName)
    );
    var imageFrameTask = imageTask.ContinueWith(
        image => image.Result.Convert<Gray, byte>()
    );
    var grayframeTask = imageFrameTask.ContinueWith(
        imageFrame => imageFrame.Result.Convert<Gray, byte>()
    );
    var facesTask = grayframeTask.ContinueWith(grayFrame =>
    {
        var cascadeClassifier = CascadeClassifierThreadLocal.Value;
        return cascadeClassifier.DetectMultiScale(
            grayFrame.Result, 1.1, 3, System.Drawing.Size.Empty);
    });
    var bitmapTask = facesTask.ContinueWith(faces =>
    {
        foreach (var face in faces.Result)
            imageTask.Result.Draw(
                face, new Bgr(System.Drawing.Color.BurlyWood), 3);
        return imageTask.Result.ToBitmap();
    });
    return bitmapTask;
}

Продолжение задачи передает результат обработки  
в прикрепленную функцию без блокировки
```

Данный код работает так, как ожидалось; время выполнения не улучшилось, но теперь программа может обрабатывать большее количество изображений, сохранив низкое потребление ресурсов. Это достигается благодаря интеллектуальной оптимизации `TaskScheduler`, из-за которой код стал более громоздким и трудноизменяемым. Например, если добавить в код обработку ошибок или возможность

отмены, то он становится запутанным и неструктурированным, его трудно понимать и поддерживать. Его необходимо улучшить. Ключом к контролируемой сложности программного обеспечения является компоновка.

Наша цель состоит в том, чтобы иметь возможность применять семантику в стиле LINQ для компоновки функций, выполняемых в программе распознавания лиц, как показано ниже (имена команд и модулей выделены жирным шрифтом):

```
from image in Task.Run<Emgu.CV.Image<Bgr, byte>()
from imageFrame in Task.Run<Emgu.CV.Image<Gray, byte>>()
from faces in Task.Run<System.Drawing.Rectangle[]>()
select faces;
```

Это пример того, как применение математических шаблонов может помочь в использовании декларативной композиционной семантики.

7.5.1. Использование математических шаблонов для оптимальной компоновки

Продолжение задач позволяет применять компоновку. Как компоновать задачи? В целом в компоновке функций участвуют две функции: результат первой подается на вход второй, таким образом получается одна функция из двух. В главе 2 мы реализовали подобную функцию `Compose` на C# (выделена жирным шрифтом):

```
Func<A, C> Compose<A, B, C>(this Func<A, B> f, Func<B, C> g) =>
    (n) => g(f(n));
```

Можем ли мы задействовать эту функцию для объединения двух задач? Непосредственно — нет. Во-первых, при такой компоновке возвращаемый типом функции должен быть надтип `Task`, как показано ниже (выделено жирным шрифтом):

```
Func<A, Task<C>> Compose<A, B, C>(this Func<A, Task<B>> f,
                                         Func<B, Task<C>> g) => (n) =>
    ➔ g(f(n));
```

Но здесь возникает проблема: этот код не компилируется. Тип, возвращаемый функцией `f`, не соответствует входному типу функции `g`: функция `f(n)` возвращает тип `Task`, не соответствующий входному типу `B` функции `g`.

Решением является реализация функции, которая обращается к базовому значению надтипа (в данном случае к задаче), а затем передает значение в следующую функцию. Это типичный шаблон, который в ФП называется монадой; монада — еще один шаблон проектирования, такой же как декоратор и адаптер. Мы уже познакомились с данной концепцией в подразделе 6.4.1, но давайте проанализируем ее идею глубже, чтобы использовать эту концепцию для улучшения программы распознавания лиц.

Монады — это математические шаблоны, которые управляют выполнением побочных эффектов, инкапсулируя логику программы, поддерживая функциональную

чистоту и предоставляя мощный инструментарий для компоновки вычислений, поддерживающих надтипы. Согласно определению монады, чтобы определить монадический конструктор, есть две функции реализации: `Bind` и `Return`.

Монадические операторы `Bind` и `Return`

Оператор `Bind` принимает экземпляр надтипа, извлекает базовое значение и затем вызывает функцию для этого извлеченного значения, возвращая новый надтип. При необходимости данная функция будет выполняться и в будущем. Здесь в сигнатуре `Bind` роль надтипа играет объект `Task`:

```
Task<R> Bind<T, R>(this Task<T> m, Func<T, Task<R>> k)
```

Значение `Return` — это оператор, который обворачивает любой тип `T` в экземпляр надтипа. В примере с типом `Task` сигнатура выглядит так:

```
Task<T> Return(T value)
```

ПРИМЕЧАНИЕ

То же самое относится и к другим надтипам: например, можно заменить тип `Task` на другой надтип, такой как `Lazy` или `Observable`.

Законы монады

В конечном счете для того чтобы определить корректную монаду, операции `Bind` и `Return` должны удовлетворять следующим законам монад.

1. *Левая единица*. Применение к значению, обернутому операцией `Return`, операции `Bind` и последующая передача значения в функцию эквивалентны передаче значения непосредственно в функцию:

```
Bind(Return value, function) = function(value)
```

2. *Правая единица*. Возвращение значения, обернутого в `Bind`, эквивалентно значению, обернутому непосредственно:

```
Bind(elevated-value, Return) = elevated-value
```

3. *Ассоциативность*. Передача значения в функцию `f`, результат которой передается во вторую функцию `g`, эквивалентна компоновке функций `f` и `g` и передаче им начального значения:

```
Bind(elevated-value, f(Bind(g(elevated-value))) =  
Bind(elevated-value, Bind(f.Compose(g), elevated-value))
```

Теперь, используя эти монадические операции, можно исправить ошибку в предыдущей функции `Compose` и скомпоновать надтипы `Task`, как показано ниже:

```
Func<A, Task<C>> Compose<A, B, C>(this Func<A, Task<B>> f,  
Func<B, Task<C>> g) => (n) => Bind(f(n), g);
```

Сила монад в том, что они позволяют описать произвольные операции с надтипаами. В частности, для надтипа Task с помощью монад можно реализовать комбинаторы функций, позволяющие компоновать асинхронные операции различными способами, как показано на рис. 7.8.

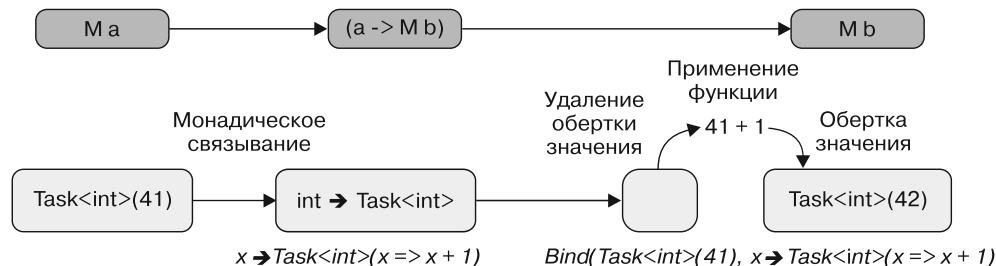


Рис. 7.8. Монадический оператор Bind принимает значение надтипа Task, которое действует как контейнер (обертка) для значения 42, после чего применяет к нему функцию $x \rightarrow \text{Task}(\text{int})(x => x + 1)$, где x — это число 41, с которого удалена обертка. В сущности, оператор Bind разворачивает значение надтипа ($\text{Task}(\text{int})(41)$), а затем применяет к нему функцию $(x + 1)$ и возвращает новое значение надтипа ($\text{Task}(\text{int})(42)$)

Как ни парадоксально, но такие монадические операторы уже встроены в .NET Framework в виде операторов LINQ. Определение LINQ SelectMany прямо соответствует монадической функции Bind. В листинге 7.9 показаны операторы Bind и Return, применяемые к типу Task. Затем эти функции задействуются для реализации семантики в стиле LINQ, что позволяет скомпоновать асинхронные операции как монады. Код, написанный на F#, используется на C#, что обеспечивает простую интероперабельность между этими языками программирования (код, на который следует обратить внимание, выделен жирным шрифтом).

Листинг 7.9. Расширение Task на F# позволяет использовать для задач операторы в стиле LINQ

TaskCompletionSource инициализирует поведение
в виде Task, поэтому его можно рассматривать как задачу

```
[<Sealed; Extension; CompiledName("Task")>]
type TaskExtensions =
    // 'T -> M<'T>
    static member Return value : Task<'T> = Task.FromResult<'T>(value)
    // M<'T> * ('T -> M<'U>) -> M<'U>
    static member Bind (input : Task<'T>, binder : 'T -> Task<'U>) =
        let tcs = new TaskCompletionSource<'U>()
        input.ContinueWith(fun (task:Task<'T>) -
            if (task.IsFaulted) then
                tcs.SetException(task.Exception.InnerException)
            elif (task.IsCanceled) then
```

Монадический оператор Return
принимает любой тип T
и возвращает Task<T>

Оператор Bind принимает объект Task как надтип, применяет
функцию к базовому типу и возвращает новый надтип Task<U>

```

    tcs.SetCanceled()
else
try
    (binder(task.Result)).ContinueWith(fun
    => (nextTask:Task<'U>) -> tcs.SetResult(nextTask.Result)) |> ignore ←
        with
            | ex -> tcs.SetException(ex)) |> ignore
    tcs.Task

static member Select (task : Task<'T>, selector : 'T -> 'U) : Task<'U> =
    task.ContinueWith(fun (t:Task<'T>) -> selector(t.Result))

static member SelectMany(input:Task<'T>, binder:'T -> Task<'I>,
projection:'T -> 'I -> 'R): Task<'R> =
    TaskExtensions.Bind(input,
        fun outer -> TaskExtensions.Bind(binder(outer), fun inner ->
            TaskExtensions.Return(projection outer inner))) ←

static member SelectMany(input:Task<'T>, binder:'T -> Task<'R>) : Task<'R>
= TaskExtensions.Bind(input,
    fun outer -> TaskExtensions.Bind(binder(outer), fun inner ->
        TaskExtensions.Return(inner))) ←
LINQ-оператор SelectMany работает
как монадический оператор Bind

```

Реализация операции `Return` очевидна, но операция `Bind` немного сложнее. Определение `Bind` можно многократно использовать для создания других комбинаторов в стиле LINQ для задач, таких как `Select`, и двух вариантов операторов `SelectMany`. В теле функции `Bind` функция `ContinueWith` из базовой сущности используется для извлечения результата вычислений входной задачи. Затем функция связывания применяется к результату входной задачи. В конце выходные данные продолжения `nextTask` являются результатом `tcs TaskCompletionSource`. Возвращаемая задача — это экземпляр базового объекта `TaskCompletionSource`, вводимый для инициализации задачи из любой операции, которая будет запускаться и завершаться в будущем. Назначение объекта `TaskCompletionSource` заключается в создании задачи, которой можно управлять и которую можно обновлять вручную, чтобы было видно, когда и как выполняется та или иная операция. Главное преимущество типа `TaskCompletionSource` заключается в возможности создания задач, которые не задерживают выполнение потоков.

TaskCompletionSource

Назначение объекта `TaskCompletionSource<T>` — предоставить управление и возможность обратиться к произвольной асинхронной операции как к `Task<T>`. При создании `TaskCompletionSource` (<http://bit.ly/2vDOmSN>) базовые свойства задачи становятся доступными посредством набора методов для управления временем жизни и завершением задачи, таких как `SetResult`, `SetException` и `SetCanceled`.

Применение шаблона монады к операциям с задачами

Теперь, когда у нас есть LINQ-операция `SelectMany` для задач, можно переписать функцию `DetectFaces`, используя выразительный и понятный запрос (код, на который необходимо обратить внимание, в листинге 7.10 выделен жирным шрифтом).

Этот код демонстрирует широкие возможности монадического шаблона, представляя семантику компоновки для надтипов, таких как задачи. Код монадических операций сконцентрирован в реализации двух операторов — `Bind` и `Return`, благодаря чему он становится удобным и легко отлаживаемым. Например, для ведения журнала функциональности или специальной обработки ошибок нужно изменить всего одно место в коде, что весьма удобно.

Листинг 7.10. Функция `DetectFaces` с использованием продолжения задачи на основе LINQ-выражения

```
Task<Bitmap> DetectFaces(string fileName) {
    Func<System.Drawing.Rectangle[], Image<Bgr, byte>, Bitmap>
    drawBoundries =
        (faces, image) => {
            faces.ForEach(face => image.Draw(face, new
    Bgr(System.Drawing.Color.BurlyWood), 3)); ←
            return image.ToBitmap(); ←
        };
```

Распознанные лица заключаются
в квадратную рамку

```
    return from image in Task.Run(() => new Image<Bgr, byte>(fileName))
           from imageFrame in Task.Run(() => image.Convert<Gray,
    byte>())
           from bitmap in Task.Run(() =>
    CascadeClassifierThreadLocal.Value.DetectMultiScale(imageFrame,
    1.1, 3, System.Drawing.Size.Empty)).Select(faces =>
        drawBoundries(faces, image))
           select bitmap; ←
```

Компоновка задач с использованием LINQ-подобных
операторов Task, определенных с помощью
монадических операторов Task

}

В листинге 7.10 операторы `Return` и `Bind` написаны на F# и используются на C# для демонстрации простой интероперабельности между этими языками программирования. Реализацию на C# вы найдете в исходном коде к книге. Как видим, монады необходимы для изящной компоновки надтипов; монада-продолжение показывает, как с помощью монад можно легко описать сложные вычисления.

Применение преобразований с помощью скрытого шаблона fmap-функциона

Одной из важных функций ФП является отображение (`Map`), которое преобразует один тип данных в другой. Сигнатура функции `Map` выглядит так:

`Map : (T -> R) -> [T] -> [R]`

Примером такой функции на C# является LINQ-оператор `Select`, который представляет собой функцию отображения для типов `IEnumerable`:

```
Ienumerable<R> Select<T,R>(Ienumerable<T> en, Func<T, R> projection)
```

В функциональном программировании подобная концепция называется *функцией*, а функция отображения называется *fmap*. Функторы — это, в сущности, отображаемые типы. В F# существует много таких типов:

```
Seq.map : ('a -> 'b) -> 'a seq -> 'b seq  
List.map : ('a -> 'b) -> 'a list -> 'b list  
Array.map : ('a -> 'b) -> 'a [] -> 'b []  
Option.map : ('a -> 'b) -> 'a Option -> 'b Option
```

Идея отображения кажется простой, но когда дело доходит до отображения надтипов, начинаются сложности. Именно здесь становится полезным шаблон функтора.

Функтор удобно представлять как контейнер, в который обернут надтип и который предоставляет способ преобразования обычной функции в функцию, способную обрабатывать значения, содержащиеся в данном контейнере. Для типа `Task` сигнатура функтора выглядит так:

```
fmap : ('T -> 'R) -> Task<'T> -> Task<'R>
```

Эта функция ранее уже была реализована для типа `Task` в виде оператора `Select` в наборе операторов в стиле LINQ, созданных для задач, построенных на F#. При вычислении последнего LINQ-выражения в функции `DetectFaces` оператор `Select` отображает (выполняет операцию `map`) входную задачу `Task<Rectangle []>` на `Task<Bitmap>`:

```
from image in Task.Run(() => new Image<Bgr, byte>(fileName))  
from imageFrame in Task.Run(() => image.Convert<Gray, byte>())  
from bitmap in Task.Run(() =>  
    CascadeClassifierThreadLocal.Value.DetectMultiScale  
        (imageFrame, 1.1, 3, System.Drawing.Size.Empty))  
    .select(faces => drawBoundries(faces, image))  
select bitmap;
```

Концепция функторов становится полезной при работе с другим функциональным шаблоном — applicативными функторами, которые будут описаны в главе 10.

ПРИМЕЧАНИЕ

Концепция функторов и монад происходит из раздела математики, называемого *теорией категорий*¹, но для того, чтобы использовать эти шаблоны, знание данной области математики не обязательно.

¹ Подробнее см.: https://wiki.haskell.org/Category_theory.

Возможности монад

Монады — это элегантное решение для компоновки надтипов. Назначением монад является управление функциями с побочными эффектами, такими как операции ввода-вывода, и обеспечение механизма для выполнения операций непосредственно над результатами ввода-вывода, так что не приходится нарушать чистоту программы, получая значения от нечистых функций. Именно в этом и заключается главная польза монад при проектировании и реализации конкурентных приложений.

7.5.2. Рекомендации по использованию задач

Ниже приведены несколько рекомендаций по использованию задач.

- ❑ В качестве возвращаемых значений рекомендуется применять неизменяемые типы. Так проще гарантировать корректность кода.
- ❑ Избегайте задач, вызывающих побочные эффекты; задачи должны взаимодействовать с остальной частью программы только посредством возвращаемых значений.
- ❑ Для продолжения вычислений рекомендуется использовать модель продолжения задач — это позволяет избежать ненужных блокировок.

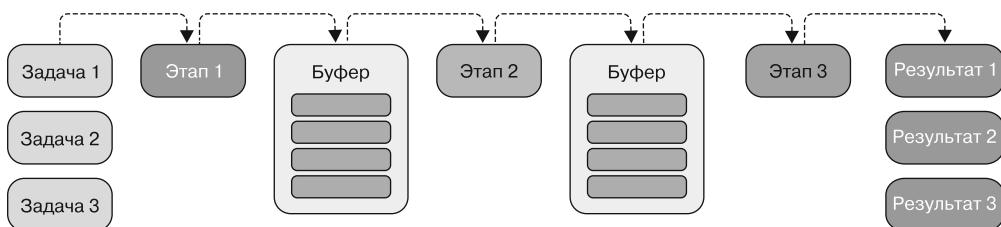
7.6. Параллельный функциональный шаблон конвейера

В этом разделе мы реализуем одну из самых распространенных технологий координации — шаблон конвейера. В целом конвейер представляет собой последовательность вычислительных операций, скомпонованных в виде цепочки этапов, где каждый этап зависит от выходных данных его предшественника и обычно выполняет преобразование данных, поступающих на вход. Шаблон конвейера можно представить себе как сборочную линию на заводе, где все элементы собираются по-этапно. Продвижение по всей цепочке описывается в виде функции; каждый раз при поступлении новых входных данных для выполнения этой функции используется очередь сообщений. Очередь сообщений является неблокирующей, поскольку она выполняется в отдельном потоке; поэтому, даже если прохождение этапов конвейера занимает некоторое время, они не будут блокировать отправителя входных данных и не помешают ему отправлять в цепочку новые данные.

Этот шаблон похож на шаблон «поставщик — потребитель», где поставщик управляет одним или несколькими рабочими потоками для генерации данных. Потребителей, которые получают данные, создаваемые производителем, может быть один или несколько. Конвейеры позволяют потребителям обрабатывать эти наборы данных параллельно. Реализация конвейера, представленная в настоящем разделе, несколько отличается от традиционной, показанной на рис. 7.9.

Традиционный параллельный конвейер

Конвейер создает буфер между всеми этапами. Этот буфер работает как параллельный шаблон «поставщик — потребитель». Буферов почти столько же, сколько этапов. Каждый обрабатываемый элемент сначала отправляется на этап 1; результат обработки передается в первый буфер, который координирует параллельную работу и передает его на этап 2. И так далее до конца конвейера, пока не будут выполнены все этапы



Функциональный параллельный конвейер

Конвейер объединяет все этапы в один, как бы скомпоновав несколько функций. Каждый обрабатываемый элемент передается в этот скомпонованный набор этапов для параллельной обработки с использованием TPL и оптимизированного планировщика

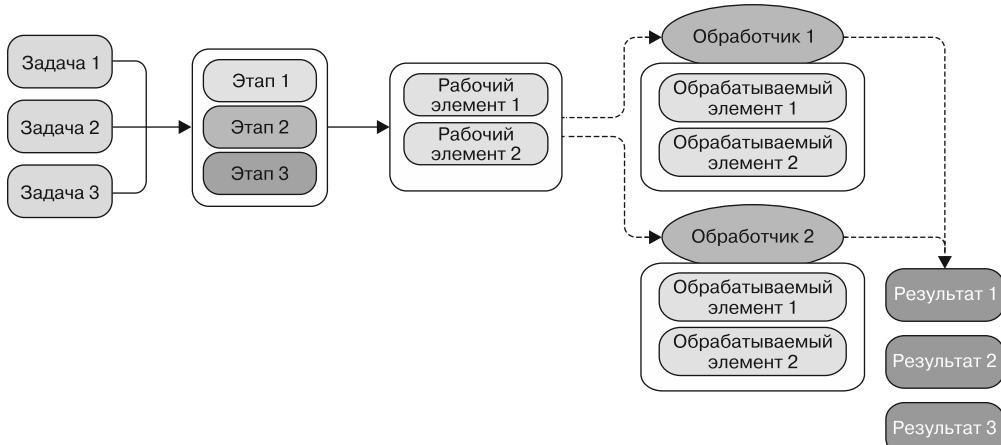


Рис. 7.9. Традиционный конвейер создает буфер перед каждым этапом, который работает как параллельный шаблон «поставщик — потребитель». Количество буферов на единицу меньше, чем количество этапов. В такой структуре каждый обрабатываемый элемент отправляется на начальный этап, затем результат передается в первый буфер, который координирует параллельную работу, чтобы затем передать этот результат на второй этап. Процесс продолжается до конца конвейера, когда будут выполнены все этапы. Функциональный параллельный конвейер, наоборот, объединяет все этапы в один, компонуя несколько функций. Затем посредством объекта Task каждый обрабатываемый элемент передается на эти скомпонованные этапы для параллельной обработки с использованием TPL и оптимизированного планировщика

Традиционный шаблон конвейера с последовательными этапами имеет ускорение, измеряемое в виде пропускной способности. Это ускорение ограничено пропускной способностью самого медленного этапа. Каждый элемент, попавший в конвейер, должен пройти через данный этап. Традиционный шаблон конвейера не масштабируется автоматически, в зависимости от количества ядер, так как он ограничен числом этапов.

Только линейный конвейер, в котором количество этапов соответствует количеству доступных логических ядер, может в полной мере использовать мощность компьютера. В компьютере с восемью ядрами конвейер, состоящий из четырех этапов, способен применять только половину ресурсов, оставляя 50 % ядер бездействующими.

ФП поощряет компоновку, на которой основана концепция конвейера. В листинге 7.11 построен конвейер с применением этого принципа: каждый этап представлен в виде отдельной функции, после чего работа распределяется параллельно, с полным использованием доступных ресурсов. Теоретически каждая функция действует как продолжение предыдущей, в стиле продолжений. В листинге представлен код на F#, реализующий конвейер, который затем задействуется на C#. В исходном коде к книге вы найдете полную реализацию конвейера на обоих языках программирования. В данном случае функциональность конвейера описана интерфейсом `IPipeline`.

Листинг 7.11. Интерфейс `IPipeline`

```

Функция, реализующая
принцип гибкого API

[<Interface>]
type IPipeline<'a, 'b> =
    abstract member Then : Func<'b, 'c> -> IPipeline<'a, 'c>
    abstract member Enqueue : 'a * Func<('a * 'b), unit> -> unit
    abstract member Execute : (int * CancellationToken) -> IDisposable
    abstract member Stop : unit -> unit

    Конвейер можно остановить в любой момент;
    эта функция запускает базовый маркер отмены

Интерфейс, определяющий
функции конвейера

Функция, передающая
в конвейер новые входные
значения для обработки

Запуск конвейера
  
```

Функция `Then` является ядром конвейера, где входная функция скомпонована из предыдущей с применением преобразования. Эта функция возвращает новый экземпляр конвейера, предоставляя удобный и гибкий API для построения процесса.

Функция `Enqueue` отвечает за передачу обрабатываемых элементов в конвейер для обработки. В качестве аргумента она принимает обратный вызов `Callback`, который используется по окончании выполнения конвейера для дальнейшей обработки результата. Такая конструкция обеспечивает гибкость и позволяет применить произвольную функцию к каждому переданному элементу.

Функция `Execute` начинает вычисление. Его входные аргументы определяют размер внутреннего буфера и маркер отмены, позволяющий остановить конвейер по требованию. Эта функция возвращает тип `IDisposable`, который можно использовать для запуска маркера отмены, чтобы остановить конвейер. Ниже представлена полная реализация конвейера (код, на который следует обратить внимание, в листинге 7.12 выделен жирным шрифтом).

Листинг 7.12. Параллельный функциональный шаблон конвейера

```
[<Struct>]
type Continuation<'a, 'b>(input:'a, callback:Func<('a * 'b), unit>) =
    member this.Input with get() = input
    member this.Callback with get() = callback ← Структура Continuation инкапсулирует входное значение для каждой задачи и обратный вызов, который будет выполняться после окончания вычислений

type Pipeline<'a, 'b> private (func:Func<'a, 'b>) as this =
    let continuations = Array.init 3 (fun _ -> new BlockingCollection<Continuation<'a, 'b>>(100)) ← Инициализация коллекции BlockingCollection, которая буферизует обработку

    let then' (nextFunction:Func<'b, 'c>) =
        Pipeline(func.Compose(nextFunction)) :> IPipeline<_, _> ← Функциональная компоновка для объединения текущей функции конвейера и новой переданной функции. Возвращает новый конвейер. Компоновка функций

    let enqueue (input:'a) (callback:Func<('a * 'b), unit>) = ← описана в главе 2
        BlockingCollection<Continuation<_, _>>.AddToAny(continuations,
    ← Continuation(input, callback)) ← Функция Enqueue передает обработку в буфер

    let stop() = for continuation in continuations do continuation.
        CompleteAdding() ← BlockingCollection получает уведомление о завершении и останавливает конвейер

    let execute blockingCollectionPoolSize
        (cancellationToken:CancellationToken) = ← Регистрация маркера отмены, при активации которого запускается функция останова
        cancellationToken.Register(Action(stop)) |> ignore ← Запуск задач для параллельного вычисления

        for i = 0 to blockingCollectionPoolSize - 1 do
            Task.Factory.StartNew(fun ()->
                while (not <| continuations.All(fun bc -> bc.IsCompleted))
                    && (not <| cancellationToken.IsCancellationRequested) do
                    let continuation = ref
    ← unchecked.defaultof<Continuation<_, _>>
                    BlockingCollection.TakeFromAny(continuations,
    ← continuation)
                    let continuation = continuation.Value
                    continuation.Callback.Invoke(continuation.Input,
    ← func.Invoke(continuation.Input)),
```

```

    cancellationToken, TaskCreationOptions.LongRunning,
➥ TaskScheduler.Default) |> ignore
static member Create(func:Func<'a, 'b>) =
    Pipeline(func) :> IPipeline<_,_>
}

interface IPipeline<'a, 'b> with
    member this.Then(nextFunction) = then' nextFunction
    member this.Enqueue(input, callback) = enqueue input callback
    member this.Stop() = stop()
    member this.Execute (blockingCollectionPoolSize,cancellationToken) =
        execute blockingCollectionPoolSize cancellationToken
    { new IDisposable with member self.Dispose() = stop() }
}

```

Статический метод создает новый экземпляр конвейера

Внутренняя структура `Continuation` используется для прохождения по конвейеру функций вычисления элементов. В реализации конвейера задействуется внутренний буфер, состоящий из массива конкурентных коллекций `BlockingCollection<Collection>`, обеспечивающего потокобезопасность при параллельном вычислении элементов. Аргумент этого конструктора коллекции определяет максимальное количество элементов, которые могут находиться в буфере одновременно. В данном случае это значение равно 100 для каждого буфера.

Каждый элемент, передаваемый в конвейер, добавляется в коллекцию, которая затем будет обрабатываться параллельно. Функция `Then` компонует функцию-аргумент `nextFunction` с функцией `func`, передаваемой в конструктор конвейера. Обратите внимание, что для компоновки функций `func` и `nextFunction` используется функция `Compose`, описанная в главе 2 (см. листинг 2.3):

```
Func<A, C> Compose<A, B, C>(this Func<A, B> f, Func<B, C> g) => (n) => g(f(n));
```

Когда конвейер запускает процесс, он применяет к каждому входному значению заключительную из скомпонованных функций. Параллелизм в конвейере достигается благодаря функции `Execute`, которая генерирует отдельную задачу для каждого экземпляра `BlockingCollection`. Это гарантирует создание буфера для каждого запускаемого потока. Задачи создаются с помощью параметра `LongRunning` для планирования выделенного потока. Конкурентная коллекция `BlockingCollection` обеспечивает потокобезопасный доступ к элементам, сохраненным с помощью статических методов `TakeFromAny` и `AddToAny`, распределяющих элементы и рабочую нагрузку между работающими потоками. Такое распределение используется для управления связями между входом и выходом конвейера, которые ведут себя как потоки поставщиков и потребителей.

ПРИМЕЧАНИЕ

Используя `BlockingCollection`, не забудьте вызвать `GetConsumingEnumerable`, поскольку класс `BlockingCollection` реализует `IEnumerable<T>`. Функция преобразования экземпляров блокирующей коллекции не принимает значений.

Конструктор конвейера объявлен закрытым во избежание прямого обращения. Вместо этого новый экземпляр конвейера инициализируется с помощью статического метода `Create`, что позволяет создать гибкий API для управления конвейером.

Такая структура конвейера напоминает параллельную модель «поставщик — потребитель», способную управлять конкурентным обменом данными между несколькими поставщиками и несколькими потребителями.

В листинге 7.13 реализованный ранее конвейер применяется для рефакторинга программы `DetectFaces`, представленной в предыдущем разделе. Гибкий API является удобным способом описать и скомпоновать этапы конвейера на C#.

Листинг 7.13. Усовершенствованный код программы `DetectFaces` с использованием параллельного конвейера

```
var files = Directory.GetFiles(ImagesFolder);

var imagePipe = Pipeline<string, Image<Bgr, byte>>
    .Create(filePath => new Image<Bgr, byte>(filePath))
    .Then(image => Tuple.Create(image, image.Convert<Gray, byte>()))
    .Then(frames => Tuple.Create(frames.Item1,
        CascadeClassifierThreadLocal.Value.DetectMultiScale(frames.Item2, 1.1,
            3, System.Drawing.Size.Empty)))
    .Then(faces =>{
        foreach (var face in faces.Item2)
            faces.Item1.Draw(face,
                new Bgr(System.Drawing.Color.BurlyWood), 3);
        return faces.Item1.ToBitmap();
    });
    };
```

Создание конвейера с использованием гибкого API

```
imagePipe.Execute(cancellationToken);
```

Запускает выполнение конвейера. Маркер отмены может остановить его в любой момент

```
foreach (string fileName in files)
    imagePipe.Enqueue(file, (_,
        => Images.Add(bitmapImage)));
```

На каждой итерации путь к файлу передается в конвейер и ставится в очередь; работа конвейера не блокируется

Используя разработанный нами конвейер, мы значительно улучшили структуру кода.

ПРИМЕЧАНИЕ

В реализации конвейера на F#, представленной в предыдущем разделе, использован делегат `Func`, который можно легко передать в C#. В исходном коде к книге вы найдете реализацию этого же конвейера, где вместо делегата `Func` из .NET применены функции F#. Такой вариант лучше подходит для проектов, полностью построенных на F#. При применении в C# собственных функций F# интероперабельность обеспечивается благодаря вспомогательному методу расширения `ToFunc`. Метод расширения `ToFunc` вы также найдете в исходном коде.

Это элегантное определение конвейера можно применять для построения процесса распознавания лиц в изображениях с использованием красивого, гибкого API. Каждая функция компонуется поэтапно, а затем вызывается функция `Execute` для запуска конвейера. Поскольку базовая обработка конвейера выполняется параллельно, цикл для передачи путей к файлам — последовательный. Функция конвейера `Enqueue` является неблокирующей, поэтому она не снижает производительность. Позже, при возврате обработанного изображения, результат обновляется посредством функции обратного вызова `Callback`, переданной в функцию `Enqueue`, и соответствующим образом обновляется пользовательский интерфейс. В табл. 7.1 показан бенчмаркинг для сравнения реализованных подходов.

Таблица 7.1. Бенчмаркинг обработки 100 изображений с использованием четырех логических ядер с 16 Гбайт оперативной памяти. Результаты в секундах представляют среднее время из трех запусков каждой реализации

Последовательный цикл	Параллельный цикл	Параллельное продолжение	Параллельное сочетание LINQ-запросов	Параллельный конвейер
68,57	22,89	19,73	20,43	17,59

Как видно по результатам теста, в среднем при загрузке 100 изображений по три раза параллельный конвейер является самым быстрым. Это также самый выразительный и лаконичный шаблон.

Резюме

- Параллельные программы на основе задач создаются с учетом функциональной парадигмы, что позволяет гарантировать более надежный и менее уязвимый (или с меньшей вероятностью поврежденный) код с такими функциональными свойствами, как неизменяемость, изоляция побочных эффектов и защитное копирование. Это упрощает проверку корректности кода.
- Microsoft TPL поддерживает функциональную парадигму в виде использования стиля продолжений, что обеспечивает удобный способ для создания цепочек неблокируемых операций.
- Метод, возвращающий `void` в коде C#, представляет собой строковый сигнал, который может создавать побочные эффекты. Метод с выходным значением типа `void` не позволяет реализовывать компоновку в задачах с использованием продолжений.
- ФП снимает покровы таинственности с математических шаблонов, позволяющих упростить компоновку параллельных задач в декларативном и гибком стиле программирования. (Шаблоны монад и функтора скрыты в LINQ.) Те же шаблоны могут применяться для реализации монадических операций в задачах в семантическом LINQ-стиле.
- Функциональный параллельный конвейер представляет собой шаблон, предназначенный для компоновки последовательности операций в одну функцию, которая затем применяется одновременно к последовательности входных значений, поставленных в очередь для обработки. Конвейеры часто бывают полезны в тех случаях, когда элементы данных принимаются из потока событий в реальном времени.
- Зависимость задач — ахиллесова пята параллелизма. Параллелизм заканчивается там, где две и более операции не могут выполняться до завершения других операций. Очень важно использовать инструменты и шаблоны, позволяющие максимально увеличить параллелизм. Ключами к этому решению являются функциональный конвейер, CPS и математические шаблоны, например монада.



Асинхронность задач – путь к победе

В этой главе:

- понятие модели асинхронного программирования на основе задач (Task-based Asynchronous Programming, TAP);
- параллельное выполнение большого количества асинхронных операций;
- настройка параметров асинхронного потока выполнения операций.

Вот уже несколько лет, как асинхронное программирование является одной из актуальных тем. Сначала асинхронное программирование использовалось главным образом на стороне клиента для предоставления потребителям быстро реагирующего, высококачественного пользовательского интерфейса. Для поддержки отзывчивого GUI асинхронное программирование должно обеспечивать устойчивую связь с серверной частью в обоих направлениях, иначе из-за задержек времени отклика замедлится. Примером такой проблемы связи является зависание окна приложения в течение нескольких секунд, пока серверная часть приложения старается справиться с командами, отправленными пользователями.

Компаниям необходимо учитывать растущие требования и запросы клиентов, при этом обеспечивая быстрый анализ данных. Использование асинхронного программирования на стороне сервера приложения является решением, позволяющим системе сохранять высокую скорость отклика независимо от количества запросов. Более того, модель асинхронного программирования (Asynchronous Programming Model, APM) является выгодной с точки зрения бизнеса. Компании начинают понимать, что разработка программного обеспечения на основе этой модели обходится

не так дорого, поскольку количество серверов, требуемых для обслуживания запросов, значительно сокращается за счет использования неблокирующей (асинхронной) системы ввода-вывода, по сравнению с синхронной системой ввода-вывода, применяющей блокировки. Имейте в виду, что понятия масштабируемости и асинхронности никак не связаны со скоростью или быстродействием. Не беспокойтесь, если пока что эти термины вам незнакомы; они будут описаны в следующих разделах.

Асинхронное программирование станет важным дополнением к вашим навыкам разработчика, поскольку программирование надежных, быстро реагирующих и масштабируемых программ очень востребовано сейчас и останется таким в будущем. Эта глава поможет вам понять семантику производительности, связанную с АРМ, и то, как писать масштабируемые приложения. В конце данной главы вы узнаете, как использовать асинхронность для параллельной обработки нескольких операций ввода-вывода независимо от доступных аппаратных ресурсов.

8.1. Модель асинхронного программирования (АРМ)

Слово «*асинхронный*» происходит от сочетания греческих слов *asyn* (что означает «не вместе») и *chronos* (что означает «время») и описывает действия, происходящие не одновременно. В контексте асинхронного выполнения программы слово «асинхронный» относится к операции, начинаящейся с некоего запроса, который может быть успешным, а может — и нет и который завершится в какой-то точке в будущем. В общем случае асинхронные операции выполняются независимо от других процессов, не дожидаясь результата, тогда как при синхронных операциях программа ожидает завершения предыдущей операции, прежде чем перейти к другой задаче.

Представьте, что вы находитесь в ресторане, где работает только один официант. Он подходит к вашему столику, принимает заказ, идет на кухню, чтобы разместить этот заказ, и остается там, ожидая, когда еда будет приготовлена и готова к сервировке! Если бы в ресторане был только один столик, то данный процесс был бы вполне хорош, но что, если столиков много? В таком случае процесс будет медленным и вы не получите хорошего обслуживания. Одно из решений — нанять несколько официантов, возможно, по одному на каждый столик. Но это повысит накладные расходы ресторана из-за увеличения заработной платы и будет крайне неэффективным. Более эффективное и рациональное решение заключается в том, чтобы официант доставлял заказ на кухню шеф-повару, а затем продолжал обслуживать другие столики. Когда повар закончит готовить еду, официант получит уведомление из кухни, что он может забрать еду и доставить ее на столик. В этом случае официант сможет своевременно обслуживать несколько столиков.

Та же концепция применима и в компьютерном программировании. Несколько операций выполняются асинхронно; начинает выполняться одна операция, затем, в ожидании ее завершения, продолжает выполняться другая, а потом, после получения данных, выполнение первой операции возобновляется.

ПРИМЕЧАНИЕ

Понятие «продолжение» означает стиль продолжений (continuation-passing style, CPS). Этот стиль представляет собой форму программирования, в которой функция сама определяет, что делать дальше. Она может принять решение продолжить операцию или сделать что-то совершенно другое. Как вы вскоре увидите, в основе APM лежит стиль CPS (описанный в главе 3).

Асинхронные программы не простоявают в бездействии, ожидая завершения очередной операции, — например, запросив данные от веб-сервиса или ответ от базы данных.

8.1.1. В чем ценность асинхронного программирования

Асинхронное программирование — отличная модель, которую можно использовать в каждой программе, где есть блокирующие операции ввода-вывода. При синхронном программировании, когда вызывается метод, вызывающий объект оказывается заблокирован до тех пор, пока не завершится выполнение текущего метода. При операциях ввода-вывода время, в течение которого вызывающий объект должен ожидать возврата управления, чтобы продолжить выполнение остального кода, зависит от текущей выполняемой операции.

Часто приложения используют большое количество внешних сервисов, которые выполняют операции, требующие заметного для пользователя времени. По этой причине жизненно важно делать программы асинхронными. Обычно разработчикам удобно мыслить последовательно: отправив запрос или запустив выполнение метода, дождаться ответа, а затем обработать его. Но высокопроизводительное и масштабируемое приложение не может себе позволить синхронно ждать завершения действия. Более того, если приложение объединяет результаты нескольких операций, то для достижения хорошей производительности необходимо выполнить все эти операции одновременно.

А что произойдет, если управление так и не вернется к вызывающему объекту, потому что во время операции ввода-вывода что-то пошло не так? Если вызывающий объект никогда не вернет себе управление, то программа может зависнуть.

Рассмотрим многопользовательское серверное приложение — например, обычный интернет-магазин, в котором для каждого входящего запроса программа должна обратиться к базе данных. Если программа рассчитана на синхронное выполнение операций (рис. 8.1), то для каждого входящего запроса создается только один выделенный поток. В этом случае каждое следующее обращение к базе данных блокирует текущий поток, которому принадлежит входящий запрос, и он ждет, пока не вернется ответ базы данных с результатом. В течение этого времени пул потоков должен создавать новые потоки для удовлетворения каждого входящего запроса; эти потоки также будут блокировать выполнение программы в ожидании ответа от базы данных.

Если приложение получает большое количество одновременных запросов (сотни или даже тысячи), то система может перестать отвечать на запросы, пытаясь создать множество потоков, необходимых для обработки запросов. Так будет продолжаться до тех пор, пока не исчерпается пул потоков и появится риск исчерпания ресурсов. Все это может привести к слишком большому потреблению памяти или, что еще хуже, к сбою системы.

Когда ресурсы пула потоков исчерпываются, последующие входящие запросы ставятся в очередь и ждут обработки, в результате чего система не реагирует на действия пользователя. Еще важнее то, что когда приходит ответ от базы данных, то заблокированные потоки освобождаются и обработка запросов продолжается, — это может спровоцировать высокую частоту переключений контекста, что отрицательно сказывается на производительности. В результате клиентские запросы к сайту замедляются, пользовательский интерфейс перестает реагировать, в итоге компания теряет потенциальных клиентов и доходы.

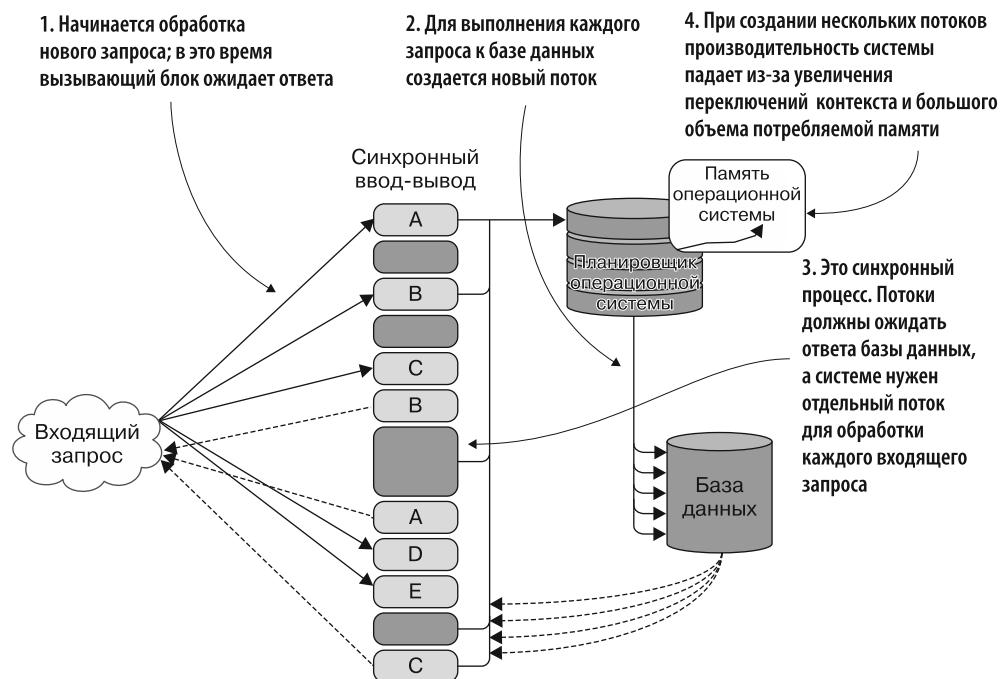


Рис. 8.1. Серверы, которые синхронно обрабатывают входящие запросы, не масштабируются

Очевидно, что именно эффективность является главной причиной перехода к асинхронной модели выполнения операций, при которой потоки не должны ожидать завершения операций ввода-вывода, что позволяет планировщику повторно использовать эти потоки для обслуживания других входящих запросов. Когда поток, развернутый для асинхронной операции ввода-вывода, оказывается

бездействующим — возможно, в ожидании ответа от базы данных, как показано на рис. 8.1, — планировщик может отправить этот поток обратно в пул потоков, где он будет доступен для других операций. А когда база данных выполнит запрос, планировщик уведомит пул потоков о необходимости выделить доступный поток для продолжения операции с результатом, поступившим от базы данных.

В серверных программах асинхронное программирование позволяет эффективно справляться с большим количеством конкурентных операций ввода-вывода, рационально повторно используя ресурсы во время их простоя и избегая создания новых ресурсов (рис. 8.2). Это оптимизирует потребление памяти и повышает производительность.

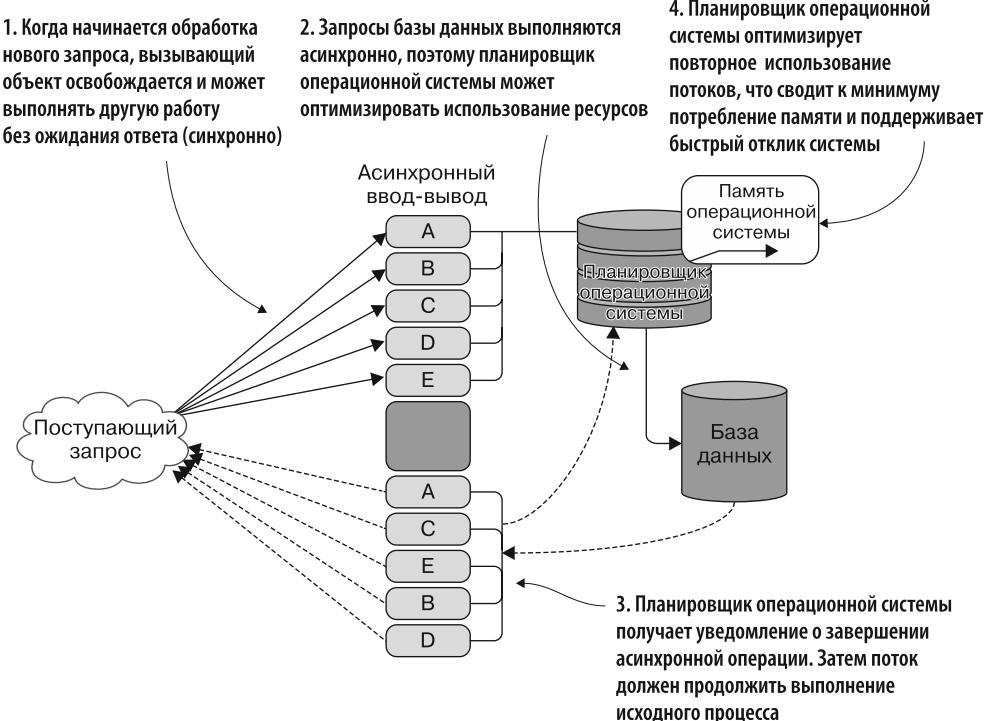


Рис. 8.2. Несколько операций асинхронного ввода-вывода могут запускаться параллельно без ограничений, а после завершения будут возвращаться к вызывающему блоку, что обеспечивает масштабируемость системы

Пользователи предъявляют высокие требования к современным приложениям, с которыми они взаимодействуют. Сегодня, чтобы удовлетворить требования пользователей, приложения должны взаимодействовать с внешними ресурсами, такими как базы данных и веб-сервисы, работать с дисками или API на базе REST. Кроме того, современные приложения должны извлекать и преобразовывать огромные объемы данных, участвовать в облачных вычислениях и реагировать на

уведомления от параллельных процессов. Для того чтобы справиться с этими сложными взаимодействиями, APM предоставляет возможность выполнять вычисления без блокировки выполняемых потоков, что улучшает доступность (надежность системы) и пропускную способность. В результате заметно повышаются производительность и масштабируемость.

Это особенно актуально для серверов с большим количеством одновременных операций ввода-вывода. В таком случае APM позволяет обрабатывать множество конкурентных операций с низким потреблением памяти вследствие небольшого количества задействованных потоков. Даже в случае небольшого (несколько тысяч) количества одновременных операций асинхронный подход выгоден, поскольку он обеспечивает выполнение операций ввода-вывода вне пула потоков .NET.

Применяя в приложении асинхронное программирование, вы получите следующие преимущества кода:

- ❑ разделенные операции выполняют минимум действий в областях, критически важных для производительности;
- ❑ увеличение доступности потоковых ресурсов позволяет системе повторно использовать одни и те же ресурсы без необходимости создавать новые;
- ❑ более эффективное применение планировщика пула потоков позволяет масштабировать серверные программы.

8.1.2. Асинхронное программирование и масштабируемость

Масштабируемость означает, что система способна реагировать на увеличение количества запросов путем добавления ресурсов и, соответственно, ускорять работу за счет параллелизма. Система, разработанная с учетом такой возможности, будет продолжать хорошо работать в условиях длительного поступления большого количества входящих запросов, которые могут нагружать ресурсы приложения. Последовательная масштабируемость достигается различными средствами: например, повышением пропускной способности памяти и процессора, распределением рабочей нагрузки и улучшением качества кода. Если вы создаете приложение на базе APM, то оно, скорее всего, будет масштабируемым.

Помните, что масштабируемость не обязательно означает высокую скорость. В общем случае масштабируемая система не всегда работает быстрее, чем немасштабируемая. Фактически асинхронная операция не выполняется быстрее, чем ее синхронный эквивалент. Главная выгода заключается в сведении к минимуму узких мест по производительности в приложении и оптимизации потребления ресурсов, что позволяет другим асинхронным операциям работать параллельно и в конечном счете ускорять работу.

Сегодня масштабируемость жизненно важна для удовлетворения все возрастающих потребностей мгновенного реагирования. Например, в высоконагруженных веб-приложениях, используемых, скажем, при торговле акциями или в социальных

сетях, критически важно, чтобы эти приложения имели быстрое время отклика и были способны конкурентно управлять огромным количеством запросов. Людям естественно мыслить последовательно, выполняя действия одно за другим, по очереди. Из соображений простоты программы изначально писались так же, одна операция за другой, что сейчас является неуклюжим и трудоемким подходом. Сейчас появилась потребность в новой модели — APM, позволяющей писать неблокирующие приложения, способные при необходимости выполнять операции не по порядку, обеспечивая практически неограниченные возможности.

8.1.3. Операции с ограничениями процессора и ввода-вывода

В вычислениях с ограничениями процессора выполнение метода занимает несколько процессорных циклов, причем для каждого процессора выделяется один поток, выполняющий работу. Асинхронные вычисления с ограничениями ввода/вывода, наоборот, не связаны с количеством процессорных ядер. Сравнение этих двух вариантов показано на рис. 8.3. Как уже упоминалось, при вызове асинхронного метода поток выполнения сразу возвращается к вызывающему объекту и продолжает выполнение текущего метода, в то время как ранее вызванная функция выполняется в фоновом режиме, тем самым предотвращая блокировку. Термины «неблокирующий» и «асинхронный» обычно взаимозаменяемы, поскольку означают аналогичные понятия.

Вычисления с ограничениями процессора получают входные данные с клавиатуры, чтобы выполнить некоторую работу, а затем вывести результат на экран. На одноядерной машине переход к следующему вычислению возможен только в том случае, если выполнено предыдущее

Вычисления с ограничениями ввода-вывода выполняются независимо от процессора, и операции могут выполняться где угодно. В данном случае одновременно происходит несколько асинхронных обращений к базе данных. Позже, когда операция будет завершена, вызывающий объект получит уведомление (посредством обратного вызова)

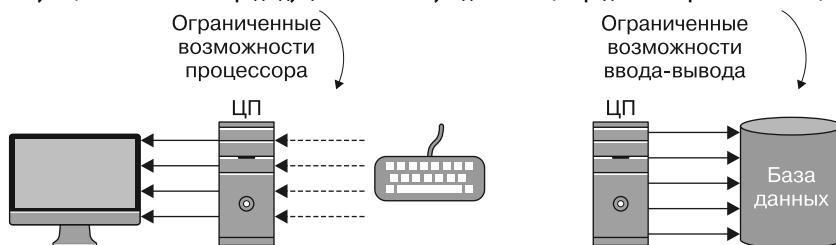


Рис. 8.3. Сравнение операций с ограничениями процессора и ввода-вывода

Вычисления с ограничениями процессора — это операции, время которых тратится на интенсивное взаимодействие с процессором с постоянной нагрузкой на аппаратные ресурсы. Поэтому для них целесообразно выделять по одному потоку на процессор, так что время выполнения определяется скоростью каждого процессора. Для *вычислений с ограничениями ввода-вывода*, наоборот, количество выполняемых

потоков не имеет отношения к количеству доступных процессоров, и время выполнения зависит от периода, затраченного на ожидание завершения операций ввода-вывода, которое зависит только от возможностей драйверов ввода-вывода.

8.2. Неограниченный параллелизм при асинхронном программировании

Асинхронное программирование обеспечивает простой способ выполнения нескольких задач независимо и, следовательно, параллельно. Возможно, вам приходят в голову вычисления, связанные с процессорами, которые можно распараллелить с использованием модели программирования на основе задач (глава 7). Но по сравнению с вычислениями с ограничениями процессора особенность АРМ определяется тем, что по своей вычислительной природе это модель с ограничениями ввода-вывода, и данный факт позволяет преодолеть аппаратное ограничение одного рабочего потока для каждого процессорного ядра.

Асинхронные вычисления, не имеющие ограничений процессора, выигрывают от того, что позволяют выполнять несколько потоков на одном процессоре. На одноядерном компьютере можно выполнять сотни или даже тысячи операций ввода-вывода, поскольку характер асинхронного программирования позволяет использовать параллелизм для выполнения операций ввода-вывода, причем количество таких операций может на порядок превышать количество доступных процессорных ядер в компьютере. Это можно сделать, поскольку асинхронные операции ввода-вывода перемещают работу в другое место, не влияя на ресурсы локального процессора, которые остаются свободными, что дает возможность выполнять дополнительную работу в локальных потоках. Чтобы продемонстрировать эти неограниченные возможности, рассмотрим пример в листинге 8.1, где выполняется 20 асинхронных операций (выделены жирным шрифтом). Подобные операции могут выполняться параллельно, независимо от количества доступных ядер.

Время выполнения этого примера с полностью асинхронной реализацией составляет 1,546 с на четырехядерной машине. Такая же синхронная реализация выполняется за 11,230 с (синхронный код опущен, но вы найдете его в исходном коде для этой книги). Время меняется в зависимости от скорости и пропускной способности сети, но асинхронный код примерно в семь раз быстрее, чем синхронный.

При работе в режиме с ограничениями процессора на одноядерном устройстве при одновременном запуске двух и более потоков улучшения производительности не наблюдается. Производительность может даже ухудшиться из-за дополнительных издержек. Это касается и многоядерных процессоров, если число работающих потоков намного превышает количество ядер. Асинхронность не увеличивает процессорный параллелизм, но повышает производительность и сокращает количество требуемых потоков. Несмотря на множество попыток удешевить потоки операционной системы (снизить потребление памяти и издержки на их создание), их выделение создает большой стек памяти, что не позволяет решать проблемы, требующие очень больших асинхронных операций. Данный вопрос подробно обсуждался в подразделе 7.1.2.

Листинг 8.1. Параллельные асинхронные вычисления

```

let httpAsync (url : string) = async {
    let req = WebRequest.Create(url)
    let! resp = req.AsyncGetResponse()
    use stream = resp.GetResponseStream()
    use reader = new StreamReader(stream)
    let! text = reader.ReadToEndAsync()
    return text
}

let sites =
    [ "http://www.live.com"; "....." "http://www.fsharp.org";
      "http://news.live.com"; "http://www.digg.com";
      "http://www.yahoo.com"; "http://www.amazon.com";
      "http://news.yahoo.com"; "http://www.microsoft.com";
      "http://www.google.com"; "http://www.netflix.com";
      "http://news.google.com"; "http://www.maps.google.com";
      "http://www.bing.com"; "http://www.microsoft.com";
      "http://www.facebook.com"; "http://www.docs.google.com";
      "http://www.youtube.com"; "http://www.gmail.com";
      "http://www.reddit.com"; "http://www.twitter.com"; ]

```

Асинхронное чтение содержимого сайта

Список произвольных сайтов для загрузки

sites
|> Seq.map httpAsync

Создание последовательности асинхронных операций для выполнения

|> Async.Parallel

Запуск параллельного выполнения нескольких асинхронных вычислений

|> Async.RunSynchronously

Запуск программы и ожидание результата, что приемлемо для консольной работы или в целях тестирования. Рекомендуемый подход — избегать блокировок, как будет показано далее

Асинхронность или параллелизм?

Параллелизм обеспечивает прежде всего производительность приложения, а также упрощает интенсивную работу с процессорами с применением нескольких потоков, используя преимущества современных многоядерных компьютерных архитектур. Асинхронность — это расширенный вариант конкурентности, ориентированный на операции с ограничениями ввода-вывода, а не процессора. Асинхронное программирование решает проблему простоев (всего, что требует много времени для выполнения).

8.3. Поддержка асинхронности в .NET

АПМ входит в состав Microsoft .NET Framework, начиная с самой первой версии (v1.1). Эта модель позволяет выгружать работу из основного выполняемого потока в другие рабочие потоки с целью ускорения реакции и обеспечения масштабируемости.

Исходный шаблон асинхронного программирования подразумевает разделение функции с длительным временем выполнения на две части. Одна часть отвечает

за запуск асинхронной операции (`Begin`), а другая вызывается при завершении операции (`End`).

В следующем коде показана синхронная (блокирующая) операция,читывающая данные из файлового потока, а затем обрабатывающая сгенерированный байтовый массив (код, на который следует обратить внимание, выделен жирным шрифтом):

```
void ReadFileBlocking(string filePath, Action<byte[]> process)
{
    using (var fileStream = new FileStream(filePath, FileMode.Open,
                                             FileAccess.Read, FileShare.Read))
    {
        byte[] buffer = new byte[fileStream.Length];
        int bytesRead = fileStream.Read(buffer, 0, buffer.Length);
        process(buffer);
    }
}
```

Для того чтобы преобразовать этот код в эквивалентную асинхронную (*неблокирующую*) операцию, нужны уведомления в виде *обратного вызова*, которые позволяют продолжать выполнение, начиная с исходной вызывающей точки (того места, откуда была вызвана функция), пока не закончится операция асинхронного ввода-вывода. В этот момент обратный вызов сохраняет соответствующее состояние из функции `Begin`, как показано в листинге 8.2 (код, на который следует обратить внимание, выделен жирным шрифтом). Затем, когда обратный вызов вернет управление, состояние будет перезаписано (восстановлено в исходном представлении).

Листинг 8.2. Асинхронное чтение данных из файловой системы

Создание экземпляра FileStream с использованием опции Asynchronous.
Обратите внимание: поток удаляется не здесь, чтобы впоследствии избежать ошибки доступа к удаленному объекту после завершения выполнения Async

```
IAsyncResult ReadFileNoBlocking(string filePath, Action<byte[]> process)
{
    var fileStream = new FileStream(filePath, FileMode.Open,
                                    FileAccess.Read, FileShare.Read, 0x1000,
                                    FileOptions.Aynchronous)
    byte[] buffer = new byte[fileStream.Length];
    var state = Tuple.Create(buffer, fileStream, process); ←
    return fileStream.BeginRead(buffer, 0, buffer.Length,
                               EndReadCallback, state);
}

void EndReadCallback(IAsyncResult ar) ←
{
    var state = ar.AsyncState;
    as (Tuple<byte[], FileStream, Action<byte[]>>)
    using (state.Item2) state.Item2.EndRead(ar);
    state.Item3(state.Item1);
}

Обратный вызов восстанавливает состояние
в исходной форме для доступа к базовым значениям
```

Пояснения к коду:

- Передача состояния в функцию обратного вызова. Процесс функции передается как часть кортежа
- Запуск функции `BeginRead`. `EndReadCallback` передается в качестве функции обратного вызова для уведомления о завершении операций
- Удаление `FileStream` и обработка данных

Почему асинхронная версия операции, использующая шаблон `Begin/End`, является неблокирующей? Потому что при запуске операции ввода-вывода поток в контексте возвращается в пул потоков для выполнения другой полезной работы, когда в этом возникнет необходимость. В .NET планировщик пула потоков обеспечивает планирование работы для всего пула потоков, управляемого CLR.

СОВЕТ

Флаг `FileOptions.Asynchronous` передается в качестве аргумента в конструктор `FileStream`, что гарантирует действительно асинхронную операцию ввода-вывода на уровне операционной системы. Он отправляет уведомление в пул потоков, чтобы избежать блокировки. В предыдущем примере `FileStream` не удаляется в вызове `BeginRead`, чтобы впоследствии, при выполнении асинхронного вычисления, избежать ошибки доступа к удаленному объекту.

Считается, что разрабатывать программы на основе АРМ труднее, чем их последовательные версии. АРМ-программа требует больше кода, этот код сложнее, его труднее читать и писать. Код может получиться еще более запутанным, если в нем несколько асинхронных операций выполняются последовательно. В следующем примере показан набор асинхронных операций, которым требуется уведомление для продолжения работы. Отправка уведомлений осуществляется посредством обратного вызова.

Обратный вызов

Обратный вызов — это функция, используемая для ускорения работы программы. При асинхронном программировании с применением обратного вызова создаются новые потоки для независимого запуска методов. При асинхронном запуске программа уведомляет вызывающий поток о любых изменениях, в том числе о неудачном завершении, отмене, продолжении и завершении выполнения, с помощью повторно выполняемой функции, используемой для регистрации продолжения другой функции. Этот процесс требует некоторого времени.

Такая цепочка асинхронных операций создает в коде набор вложенных обратных вызовов, известный как *проклятие обратных вызовов* (`callback hell`, <http://callbackhell.com>). Код на основе обратных вызовов является источником проблем, поскольку заставляет программиста передавать управление, ограничивая выразительность, и, что более важно, при этом теряется семантический аспект компоновки.

Ниже приведен концептуальный пример кода, где выполняется чтение из потока файлов, затем данные сжимаются и отправляются в сеть (код, на который нужно обратить внимание, выделен жирным шрифтом):

```
IAsyncResult ReadFileNoBlocking(string filePath)
{
    // сохранить контекст и запустить BeginRead
}
void EndReadCallback(IAsyncResult ar)
```

```
{  
    // получить данные от Read и восстановить состояние,  
    // затем запустить BeginWrite (сжатие)  
}  
void EndCompressCallback(IAsyncResult ar)  
{  
    // получить данные от Write и восстановить состояние,  
    // затем запустить BeginWrite (отправить данные в сеть)  
}  
void EndWriteCallback(IAsyncResult ar)  
{  
    // получить данные от Write и восстановить состояние, завершить процесс  
}
```

Как бы вы поступили, чтобы добавить новую функциональность в этот процесс? Данный код не простой в обслуживании! Как построить такую же последовательность асинхронных операций, чтобы избежать проклятия обратных вызовов? И где и как разместить обработку ошибок и освобождение ресурсов? Все это сложные задачи!

В целом асинхронный шаблон Begin/End более или менее работоспособен для единичного вызова, но в случае последовательности асинхронных операций он позорно проваливается. Далее в этой главе я покажу, как справиться с подобными исключениями и отменами.

8.3.1. Асинхронное программирование нарушает структуру кода

Как видно из предыдущего кода, проблема, возникающая в результате применения традиционной модели АРМ, — это отсутствие связи между временем выполнения начальной операции (`Begin`) и ее уведомлением обратного вызова (`End`). Такая структура программы ломает код надвое: операция делится на две части, что нарушает императивную последовательную структуру программы. Соответственно, операция продолжается и завершается в разных областях видимости и, возможно, в разных потоках, что усложняет отладку и обработку исключений, а также делает невозможным управление областями транзакций.

В целом при использовании шаблона АРМ трудно сохранять состояние между асинхронными вызовами. Для продолжения работы приходится передавать состояние в каждое продолжение через обратный вызов. Для управления передачей состояния между этапами асинхронного конвейера требуется выделенный конечный автомат.

В предыдущем примере для передачи состояния между функцией `fileStream.BeginRead` и ее обратным вызовом `EndReadCallback` был создан выделенный объект `state`, обеспечивающий доступ к потоку, буферу байтового массива и процессу функции:

```
var state = Tuple.Create(buffer, fileStream, process);
```

При завершении операции объект `state` был восстановлен, чтобы получить доступ к базовым объектам для продолжения работы.

8.3.2. Асинхронное программирование на основе событий

Осознав внутренние проблемы АРМ, Microsoft внедрила (начиная с .NET 2.0) альтернативный шаблон, названный асинхронным программированием на основе событий (Event-based Asynchronous Programming, EAP)¹. Модель EAP была первой попыткой решить проблемы АРМ. Идея, лежащая в основе EAP, заключается в том, чтобы создать обработчик события, позволяющий уведомить асинхронную операцию о завершении задачи. Это событие используется вместо семантики уведомлений посредством обратного вызова.

Поскольку событие вызывается в корректном потоке и обеспечивает прямой доступ к элементам пользовательского интерфейса, EAP имеет ряд преимуществ по сравнению с АРМ. Кроме того, данная модель поддерживает генерацию отчетов о ходе выполнения, отмену операций и обработку ошибок — и все это происходит незаметно для разработчика.

EAP обеспечивает более простую модель асинхронного программирования, чем АРМ, и основана на стандартном механизме событий .NET, не требуя использования специального класса и обратных вызовов. Но и она не идеальна, поскольку продолжает разделять код на вызовы методов и обработчики событий, усложняя логику программы.

8.4. Асинхронное программирование на основе задач в C#

По сравнению со своим предшественником, .NET АРМ, модель асинхронного программирования на основе задач (Task-based Asynchronous Programming, ТАР) нацелена на упрощение реализации асинхронных программ и компоновки последовательностей конкурентных операций. Благодаря модели ТАР обе предыдущие модели — и АРМ, и EAP — уходят в прошлое, поэтому, если вы пишете асинхронный код на C#, то рекомендуется использовать ТАР. Модель ТАР обеспечивает ясный декларативный стиль для написания асинхронного кода, напоминающего асинхронный рабочий процесс F#, под влиянием которого он и был создан. Асинхронный рабочий процесс F# будет подробно рассмотрен в следующей главе.

В C# (начиная с версии 5.0) объекты `Task` и `Task<T>` при поддержке ключевых слов `async` и `await` стали основными компонентами для моделирования асинхронных операций. ТАР решает проблему обратного вызова, позволяя сосредоточиться исключительно на синтаксическом аспекте, минуя трудности, возникающие при рассуждениях о последовательности событий, описанных в коде. Асинхронные функции в C# 5.0 решают проблему задержки, которая возникает во всех случаях, когда выполнение чего-либо занимает заметное время.

¹ Подробнее см. здесь: <http://mng.bz/2287>.

Идея состоит в том, чтобы выполнить асинхронный метод, возвращая задачу (также называемую *future-объектом*), изолирующую и инкапсулирующую длительную операцию, которая завершится в какой-то момент в будущем, как показано на рис. 8.4.

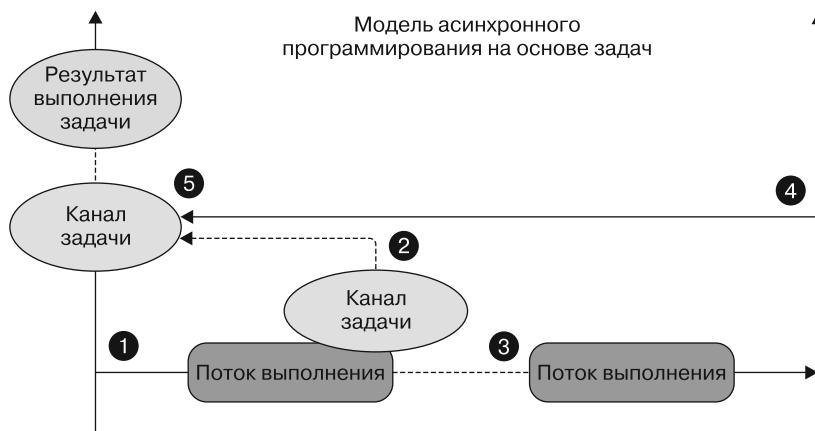


Рис. 8.4. Задача действует как канал для потока выполнения, который может продолжать работу, пока объект, вызывающий операцию, получает дескриптор задачи. Когда операция завершится, задача получает уведомление и доступ к основному результату

Задачи на рис. 8.4 выполняются в такой последовательности.

1. Операция ввода-вывода запускается асинхронно в отдельном потоке выполнения. Для обработки операции создается новый экземпляр задачи.
2. Созданная задача возвращается вызывающему объекту. Задача содержит обратный вызов, который действует как канал между вызывающим объектом и асинхронной операцией. По этому каналу поступает информация о завершении операции.
3. Поток выполнения продолжает выполнение операции, в то время как основной поток, которому принадлежит объект, вызывающий операцию, доступен для выполнения другой работы.
4. Операция завершается асинхронно.
5. Задача получает уведомление, и результат становится доступен для объекта, вызвавшего операцию.

Объект `Task`, возвращаемый из выражения `async/await`, содержит сведения об инкапсулированном вычислении и ссылку на результат вычисления, который станет доступен после завершения самой операции. Эти данные включают в себя состояние задачи, ее результат, если она завершена, и информацию об исключении, если оно возникнет.

Мы уже использовали конструкции .NET Task и Task<T> в предыдущей главе, в частности для вычислений с ограничениями процессора. Та же модель в сочетании с ключевыми словами `async/await` может применяться для операций с ограничениями ввода-вывода.

ПРИМЕЧАНИЕ

В пуле потоков есть две группы: рабочие потоки и потоки ввода-вывода. Рабочие потоки предназначены для заданий с ограничениями процессора; потоки ввода-вывода более эффективны для выполнения операций с ограничениями ввода-вывода. В пуле потоков хранится кэш рабочих потоков, потому что создание потока сопряжено со значительными затратами. Пул потоков CLR хранит отдельные пулы каждого типа, чтобы избежать ситуации, когда из-за высокого спроса на рабочие потоки все потоки, доступные для выполнения собственных обратных вызовов ввода-вывода, будут заняты, что может привести к взаимоблокировке. Только представьте себе, что было бы с приложением, использующим большое количество рабочих потоков, каждый из которых ожидал бы завершения ввода-вывода.

Коротко говоря, ТАР состоит из следующих элементов:

- конструкций Task и Task<T> для представления асинхронных операций;
- ключевого слова `await` для ожидания завершения задачи асинхронной операции, в то время как текущий поток не блокируется и свободен для выполнения другой работы.

Например, если операция выполняется в отдельном потоке, ее необходимо обернуть в объект Task:

```
Task<int[]> processDataTask = Task.Run(() => ProcessMyData(data));  
// выполнять другую работу  
var result = processDataTask.Result;
```

Результат вычислений доступен через свойство `Result`, блокирующее вызывающий метод до завершения задачи. Для задач, которые не возвращают результат, можно вместо этого вызвать метод `Wait`. Но так делать не рекомендуется. Чтобы избежать блокировки вызывающего потока, можно использовать ключевые слова `async/await` для объекта Task:

```
Task<int[]> processDataTask = Task.Run(async () => ProcessMyData(data));  
// выполнять другую работу  
var result = await processDataTask;
```

Ключевое слово `async` уведомляет компилятор о том, что данный метод выполняется асинхронно без блокировки. Таким образом, вызывающий поток будет освобожден для выполнения другой работы. После завершения задачи один из свободных рабочих потоков возобновит обработку.

ПРИМЕЧАНИЕ

Методы, помеченные как `async`, могут возвращать `void`, `Task` или `Task<T>`, но рекомендуется ограничить использование сигнатуры с `void`. Такую сигнатуру следует применять только в точках входа верхнего уровня программы и в обработчиках событий пользовательского интерфейса. Лучше задействовать тип `Task<Unit>`, описанный в предыдущей главе.

Вот предыдущий пример, преобразованный для асинхронного чтения файлового потока в стиле ТАР (код, на который следует обратить внимание, выделен жирным шрифтом):

```
async void ReadFileNoBlocking(string filePath, Action<byte[]> process)
{
    using (var fileStream = new FileStream(filePath, FileMode.Open,
                                FileAccess.Read, FileShare.Read, 0x1000,
                                FileOptions.Asynchronous))
    {
        byte[] buffer = new byte[fileStream.Length];
        int bytesRead = await fileStream.ReadAsync(buffer, 0, buffer.Length);
        await Task.Run(async () => process(buffer));
    }
}
```

Метод `ReadFileNoBlocking` помечен как `async`; это контекстное ключевое слово используется для определения асинхронной функции и позволяет применять ключевое слово `await` для данного метода. Цель конструкции `await` — информировать компилятор C# о том, что надо транслировать код в *продолжение* задачи, которая не будет блокировать текущий контекстный поток, освобождая этот поток для выполнения другой работы.

ПРИМЕЧАНИЕ

Функциональность `async/await` в C# основана на регистрации обратного вызова в виде продолжения, которое начнет выполняться, когда выполнение задачи в контексте завершится. Такой код легко реализовать в гибком и декларативном стиле.

Ключевые слова `async/await` являются связующими звеньями для монады продолжения, которая реализуется как монадический оператор связывания с использованием функции `ContinuesWith`. Этот подход хорошо сочетается с построением цепочки методов, поскольку каждый метод возвращает задачу, имеющую свой метод `ContinuesWith`. Однако для того, чтобы получить результат и передать его следующему методу, требуется непосредственная работа с задачами. Более того, если нужно построить цепочку из большого количества задач, то для того, чтобы получить нужное значение, приходится перебрать все результаты. Вместо этого нам нужен более обобщенный подход, который можно использовать в разных методах и на произвольном уровне цепочки. Именно это и предлагает модель программирования `async/await`.

Внутри продолжение реализовано с помощью функции `ContinuesWith` объекта `Task`; эта функция запускается при завершении асинхронной операции. Преимущество сборки продолжения компилятором заключается в том, что сохраняется структура программы и асинхронные методы выполняются без необходимости обратных вызовов или вложенных лямбда-выражений.

Данный асинхронный код имеет ясную семантику, построенную в виде последовательного потока. В общем случае, когда вызывается метод, помеченный как `async`, поток выполнения работает синхронно, пока очередь не дойдет до задачи, которая помечена ключевым словом `await` и еще не завершена. Когда поток выполнения встречает ключевое слово `await`, он приостанавливает вызываемый метод и возвращает управление вызвавшему его объекту до тех пор, пока ожидаемая задача не будет завершена; таким образом, поток выполнения не блокируется. Когда операция завершится, ее результат будет развернут и привязан к переменной содержимого, а затем поток продолжает выполнение оставшейся работы.

Интересным аспектом ТАР является то, что поток выполнения захватывает контекст синхронизации и возвращается к потоку, который продолжает выполнение, что позволяет обновлять пользовательский интерфейс напрямую, без выполнения дополнительных действий.

8.4.1. Анонимные асинхронные лямбда-функции

Возможно, вы заметили в предыдущем коде любопытную деталь — анонимная функция была помечена как `async`:

```
await Task.Run(async () => process(buffer));
```

Как видим, не только обычные именованные методы, но и анонимные методы могут быть помечены как `async`. Вот другой вариант записи для создания анонимной асинхронной лямбда-функции:

```
Func<string, Task<byte[]>> downloadSiteIcone = async domain =>
{
    var response = await new
        HttpClient().GetAsync($"http://{domain}/favicon.ico");
    return await response.Content.ReadAsByteArrayAsync();
}
```

Это также называется *асинхронной лямбда-функцией*¹. Она похожа на обычное лямбда-выражение, только в начале стоит модификатор `async`, чтобы разрешить использование ключевого слова `await` в теле функции. Асинхронные лямбда-функции полезны в тех случаях, когда нужно передать в метод потенциально длительно выполняемый делегат. Если метод принимает `Func<Task>`, то можно передать ему асинхронную лямбда-функцию и получить все преимущества асинхронности. Как и любое другое лямбда-выражение, асинхронная лямбда-функция поддерживает

¹ Методы и лямбда-функции с модификатором `async` называются асинхронными функциями.

замыкание для захвата переменных и асинхронный запуск операции только при вызове делегата.

Такие функции являются простым способом быстрого описания асинхронных операций. Внутри асинхронных лямбда-функций `await`-выражения могут ожидать выполнения задач. Это приводит к тому, что остальная часть асинхронного выполнения незаметно для разработчика станет продолжением ожидаемой задачи. В анонимных асинхронных лямбда-функциях действуют те же правила, что и в обычных асинхронных методах. Вы можете использовать такие функции для краткости кода и для создания замыканий.

8.4.2. Монадический контейнер `Task<T>`

В предыдущей главе мы увидели, что тип `Task<T>` можно рассматривать как специальную обертку, которая в случае успешного выполнения в итоге предоставляет значение типа `T`. Тип `Task<T>` — монадическая структура данных, и это, кроме прочего, означает, что такой тип может быть легко скомпонован с другими типами. Разумеется, подобная концепция относится и к типу `Task<T>`, используемому в TAP.

Монадический контейнер

Пора дополнить понятие монады, представленное ранее в книге. В контексте `Task` можно представить монаду, действующую как контейнер. Монадический контейнер — мощный инструмент компоновки, применяемый в функциональном программировании, чтобы описать способ объединения операций в цепочку и при этом избежать опасного и нежелательного поведения. В сущности, монады означают, что вы работаете с *обернутыми, замкнутыми значениями*, такими как типы `Task` и `Lazy`, которые развертываются только в тот момент, когда в них возникает необходимость. Например, монады позволяют взять значение и применить к нему ряд независимых преобразований способом, инкапсулирующим побочные эффекты. Сигнатура типа монадической функции вызывает потенциальные побочные эффекты, обеспечивая представление результата вычислений и фактических побочных эффектов, которые возникли в результате выполнения функции.

Учитывая это, теперь мы можем легко определить монадические операторы `Bind` и `Return`. В частности, оператор `Bind` использует подход с продолжениями базовой асинхронной операции и обеспечивает гибкий стиль семантического программирования с компоновкой. Вот определение этих операторов, включая отображение функтора (*functor map, fmap*):

```
static Task<T> Return<T>(T task)=> Task.FromResult(task);

static async Task<R> Bind<T, R>(this Task<T> task, Func<T, Task<R>> cont)
    => await cont(await task.ConfigureAwait(false)).ConfigureAwait(false);

static async Task<R> Map<T, R>(this Task<T> task, Func<T, R> map)
    => map(await task.ConfigureAwait(false));
```

Благодаря использованию ключевого слова `await` определения функций `Map` и `Bind` получились гораздо проще, чем реализация `Task<T>` для вычислений с ограничениями процессора, представленная в предыдущей главе. Функция `Return` заключает `T` в контейнер `Task<T>`. Метод `ConfigureAwait1` в методе расширения `Task` удаляет текущий контекст пользовательского интерфейса. Так рекомендуется делать для повышения производительности в тех случаях, когда код не нуждается в обновлении или не требует взаимодействия с интерфейсом пользователя. Теперь с помощью этих операторов можно составить набор асинхронных вычислений в виде цепочки операций. В листинге 8.3 происходит асинхронная загрузка изображения (пиктограммы) из сети.

Листинг 8.3. Асинхронная загрузка изображения (пиктограммы) из сети

```
async Task DownloadIconAsync(string domain, string fileDestination)
{
    using (FileStream stream = new FileStream(fileDestination,
                                                FileMode.Create, FileAccess.Write,
                                                FileShare.Write, 0x1000, FileOptions.Asynchronous))
        await new HttpClient()
            .GetAsync($"http://{domain}/favicon.ico")           Асинхронное отображение
            .Bind(async content => await                         результата предыдущей операции
                  content.Content.ReadAsByteArrayAsync())
            .Map(bytes => Image.FromStream(new MemoryStream(bytes)))
            .Tap(async image =>                                ←
                  await SaveImageAsync(fileDestination,           Функция Tap выполняет
                  ImageFormat.Jpeg, image));                      побочный эффект
```

Привязывает асинхронную операцию,
разворачивая результат Task

В этом коде метод `DownloadIconAsync` использует экземпляр объекта `HttpClient`, чтобы асинхронно получить `HttpResponseMessage` путем вызова метода `GetAsync`. Цель ответного сообщения — получить содержимое HTTP-запроса (в этом случае изображение) в виде массива байтов. Данныечитываются оператором `Task.Bind`, а затем преобразуются в изображение с помощью оператора `Task.Map`. Функция `Task.Tap` (также известная как *k-комбинатор*) применяется, чтобы упростить построение конвейера, вызвать побочный эффект с заданными входными данными и вернуть исходное значение. Ниже представлена реализация функции `Task.Tap`:

```
static async Task<T> Tap<T>(this Task<T> task, Func<T, Task> action)
{
    await action(await task);
    return await task;
}
```

Оператор `Tap` чрезвычайно полезен для того, чтобы связать `void`-функции (такие как запись в журнал, вывод в файл или на HTML-страницу) без необходимости создавать дополнительный код. Для этого оператор `Tap` передает себя в функцию и возвращает

¹ Подробнее см. здесь: <http://mng.bz/T8US>.

сам себя. Он разворачивает базовый надтип, применяет действие для создания побочного эффекта, а затем снова обертывает исходное значение и возвращает его. В данном примере побочным эффектом является сохранение изображения в файловой системе. Функция `Tap` может использоваться и для других побочных эффектов.

На данном этапе эти монадические операторы могут применяться для определения LINQ-шаблона, реализующего функции `Select` и `SelectMany`, аналогично типу `Task` в предыдущей главе, а также для того, чтобы можно было задействовать LINQ-семантику компоновки:

```
static async Task<R> SelectMany<T, R>(this Task<T> task,
                                         Func<T, Task<R>> then) => await Bind(await task);

static async Task<R> SelectMany<T1, T2, R>(this Task<T1> task,
                                         Func<T1, Task<T2>> bind, Func<T1, T2, R> project)
{
    T taskResult = await task;
    return project(taskResult, await bind(taskResult));
}
static async Task<R> Select<T, R>(this Task<T> task, Func<T, R> project)
=> await Map(task, project);

static async Task<R> Return<R>(R value) => Task.FromResult(value);
```

Оператор `SelectMany` — одна из многих функций, способных расширять асинхронную LINQ-семантику. Задача функции `Return` — преобразовать значение `R` в `Task<R>`. Модель программирования `async/await` в C# основана на задачах и, как отмечалось в предыдущей главе, по своей природе близка к монадической концепции операторов `Bind` и `Return`. Следовательно, можно определить многие операторы LINQ-запросов, которые опираются на `SelectMany`. Важным моментом является то, что использование шаблонов, таких как монады, позволяет создать множество многоразовых комбинаторов и упростить применение технологий, позволяющих улучшить компонуемость и читаемость кода с использованием LINQ-семантики.

ПРИМЕЧАНИЕ

В подразделе 7.15.3 спецификации C# содержится список операторов, которые могут быть реализованы для поддержки всего синтаксиса LINQ.

Вот предыдущий пример функции `DownloadIconAsync`, преобразованный с использованием семантики LINQ-выражений:

```
async Task DownloadIconAsync(string domain, string fileDestination)
{
    using (FileStream stream = new FileStream(fileDestination,
                                                FileMode.Create, FileAccess.Write, FileShare.Write,
                                                0x1000, FileOptions.Asynchronous))
    await (from response in new HttpClient()
           .GetAsync($"http://{domain}/favicon.ico")
           from bytes in response.Content.ReadAsByteArrayAsync()
           select Bitmap.FromStream(new MemoryStream(bytes)))
           .Tap(async image => (await image).Save(fileDestination));
}
```

С использованием LINQ-версии замыкание `from` извлекает внутреннее значение `Task` из операции `async` и соединяет его со связанным значением. Таким образом, благодаря базовой реализации можно обойтись без ключевых слов `async/await`.

ТАР можно задействовать для распараллеливания вычислений на C#, но, как мы видели, распараллеливание выступает лишь одним из аспектов ТАР. Еще более заманчивым предложением является написание асинхронного кода, который легко, с минимальными затратами поддается компоновке.

8.5. Асинхронное программирование на основе задач: практический пример

Программы, выполняющие многочисленные операции ввода-вывода, которые требуют много времени, являются хорошими кандидатами для демонстрации возможностей асинхронного программирования и мощного набора инструментов, который ТАР предоставляет разработчику. В качестве примера мы рассмотрим в этом разделе применение ТАР для реализации программы, которая загружает с HTTP-сервера и анализирует историю фондового рынка нескольких компаний. Результаты отображаются в диаграмме, размещаемой в приложении с пользовательским интерфейсом Windows Presentation Foundation (WPF). Затем символы обрабатываются параллельно, а выполнение программы оптимизировано с учетом улучшений.

В данном случае будет логично выполнять операции асинхронно и параллельно. Каждый раз, когда мы захотим прочитать данные из сети с помощью какого-либо клиентского приложения, мы будем вызывать неблокирующие методы, преимущество которых заключается в том, что пользовательский интерфейс сохраняет высокую скорость реакции (рис. 8.5).

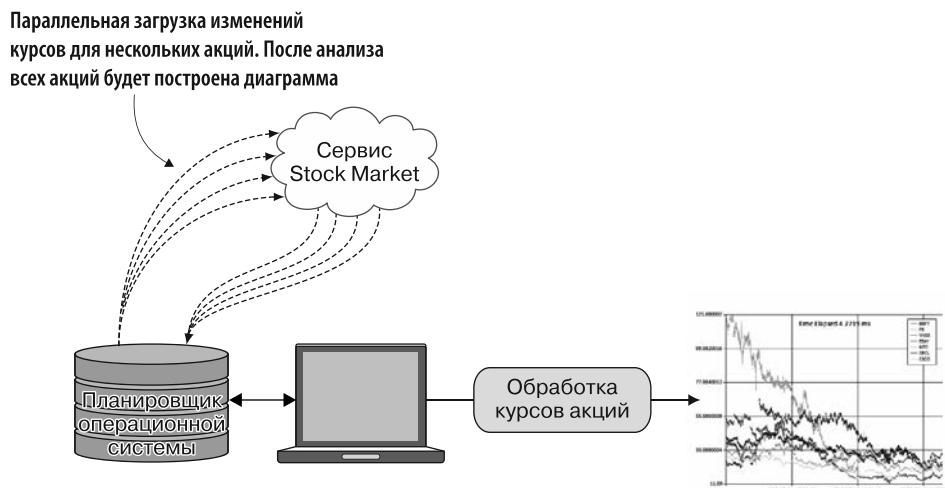


Рис. 8.5. Асинхронная параллельная загрузка изменений курсов акций. Количество запросов может превышать количество доступных ядер, но мы все равно можем увеличить степень параллелизма

В листинге 8.4 показана основная часть программы. Для построения диаграммы мы воспользуемся элементом управления Microsoft Windows.Forms.DataVisualization¹. Рассмотрим, как работает модель асинхронного программирования на .NET. Сначала мы определим структуру данных StockData для хранения ежедневных изменений курсов акций:

```
struct StockData
{
    public StockData(DateTime date, double open,
                     double high, double low, double close)
    {
        Date = date;
        Open = open;
        High = high;
        Low = low;
        Close = close;
    }
    public DateTime Date { get; }
    public Double Open { get; }
    public Double High { get; }
    public Double Low { get; }
    public Double Close { get; }
}
```

Для каждой акции существует несколько точек изменения данных; StockData в виде структуры, состоящей из значений разных типов, может повысить производительность благодаря оптимизации памяти. В листинге 8.4 реализованы асинхронная загрузка и анализ изменений курсов акций (код, на который нужно обратить внимание, выделен жирным шрифтом).

Листинг 8.4. Анализ изменений курсов акций

Метод, который анализирует строку данных
об изменениях курсов акций и возвращает массив StockData

```
async Task<StockData[]> ConvertStockHistory(string stockHistory)
{
    → return await Task.Run(() => {
        string[] stockHistoryRows =
            stockHistory.Split(Environment.NewLine.ToCharArray(),
                               StringSplitOptions.RemoveEmptyEntries);
        return (from row in stockHistoryRows.Skip(1)
                let cells = row.Split(',')
                let date = DateTime.Parse(cells[0])
                let open = double.Parse(cells[1])
                let high = double.Parse(cells[2])
                let low = double.Parse(cells[3])
                let close = double.Parse(cells[4])
                select new StockData(date, open, high, low, close))
                           .ToArray();
    });
}
```

**Использование
асинхронного
анализатора
изменений курсов
акций в формате CSV**

¹ Подробнее см. здесь: <http://mng.bz/Jvo1>.

```

async Task<string> DownloadStockHistory(string symbol)
{
    string url =
        $"http://www.google.com/finance/historical?q={symbol}&output=csv";
    var request = WebRequest.Create(url);
    using (var response = await request.GetResponseAsync()
        .ConfigureAwait(false))
        .ConfigureAwait(false)) ← Веб-запрос для асинхронного
    using (var reader = new StreamReader(response.GetResponseStream()))
        return await reader.ReadToEndAsync().ConfigureAwait(false); ← получения HTTP-ответа
}
} ← Создание функции чтения потока для асинхронного чтения
    содержимого HTTP-ответа; весь CSV-текст читается за один проход

async Task<Tuple<string, StockData[]>> ProcessStockHistory(string symbol)
{
    string stockHistory = await DownloadStockHistoryAsync(symbol);
    StockData[] stockData = await ConvertStockHistory(stockHistory); ← Метод ProcessStockHistory
    return Tuple.Create(symbol, stockData); ← асинхронно выполняет
} ← Новый экземпляр кортежа содержит информацию об изменениях
    курса каждой акции, проанализированную и передаваемую в диаграмму
    загрузку и обработку
    изменений курсов акций

async Task AnalyzeStockHistory(string[] stockSymbols)
{
    var sw = Stopwatch.StartNew(); ← «Ленивая» коллекция асинхронных
    IEnumerable<Task<Tuple<string, StockData[]>>> stockHistoryTasks = ← операций обрабатывает данные
        stockSymbols.Select(stock => ProcessStockHistory(stock)); ← об изменениях курсов акций
    var stockHistories = new List<Tuple<string, StockData[]>>(); ←
    foreach (var stockTask in stockHistoryTasks)
        stockHistories.Add(await stockTask); ← Вывод диаграммы
    ShowChart(stockHistories, sw.ElapsedMilliseconds);
}
} ← Асинхронное выполнение операций, по очереди

```

Выполнение кода начинается с создания веб-запроса для получения HTTP-ответа от сервера, чтобы получить базовый `ResponseStream` для загрузки данных. Для выполнения операций ввода-вывода в коде используются методы экземпляра `GetReponseAsync()` и `ReadToEndAsync()`, что может занять много времени. Поэтому такие методы выполняются асинхронно, с применением шаблона ТАР. Затем код создает экземпляр `StreamReader` для чтения данных в формате значений, разделенных запятыми (comma-separated values, CSV). После этого CSV-данные преобразуются в понятную форму — объект `StockData` — с использованием LINQ-выражения и функции `ConvertStockHistory`. Эта функция выполняет преобразование данных с помощью метода `Task.Run1`, который выполняет заданное лямбда-выражение в `ThreadPool`.

Функция `ProcessStockHistory` асинхронно загружает и преобразует изменения курсов акций, после чего возвращает объект `Tuple`. В данном случае возвращается

¹ Microsoft рекомендует использовать метод `Task`, который выполняется в пользовательском интерфейсе и время вычисления которого составляет более 50 мс.

типа `Task<Tuple<string, StockData[]>>`. Интересно, что в этом методе, когда в конце создается кортеж, нет никакого объекта `Task`. Такое поведение возможно, поскольку метод помечен ключевым словом `async`, и компилятор автоматически переносит результат в тип `Task`, чтобы он соответствовал сигнатуре. В ТАР, когда метод обозначен как `async`, операции по обертыванию и развертыванию, необходимые для превращения результата в объект `Task` (и наоборот), всегда выполняются компилятором. Полученные данные отправляются в метод `ShowChart` для отображения изменений курсов акций и прошедшего времени. (Реализацию `ShowChart` вы найдете в исходном коде к книге.)

Остальная часть кода не требует пояснений. Время выполнения этой программы, включая загрузку, обработку и визуализацию изменений курсов акций для семи компаний, составило 4,272 с. На рис. 8.6 показаны результаты изменений цен на акции Microsoft (MSFT), EMC, Yahoo (YHOO), eBay (EBAY), Intel (INTC) и Oracle (ORCL).

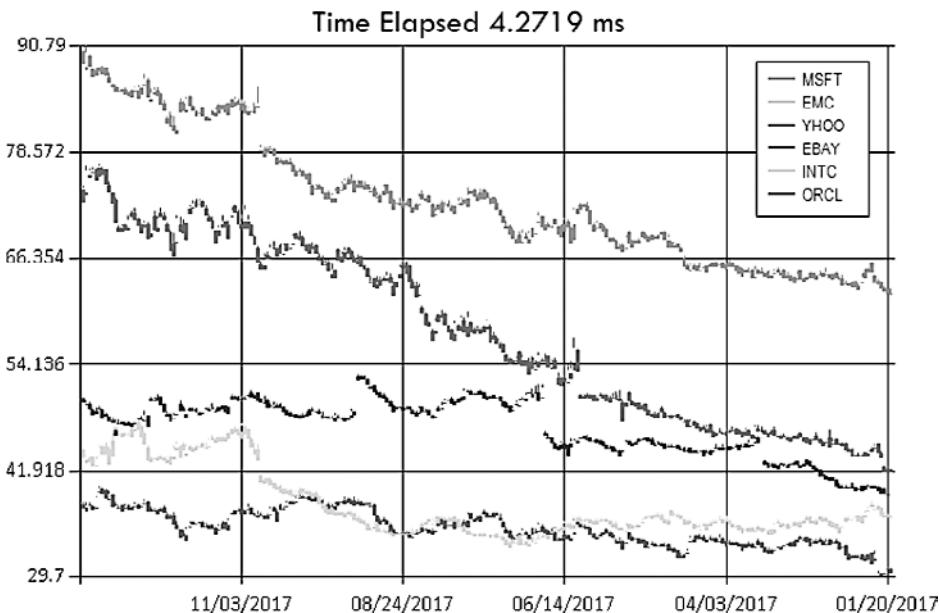


Рис. 8.6. Диаграмма колебаний цен на акции в течение заданного времени

Как видим, ТАР возвращает задачи, обеспечивая естественную композиционную семантику для других методов с одним и тем же возвращаемым типом `Task`. Рассмотрим, что происходит на протяжении всего процесса. В данном примере для загрузки и анализа истории фондового рынка использовался сервис Google (см. листинг 8.4). Это общая архитектура масштабируемого сервиса с аналогичным поведением (рис. 8.7).

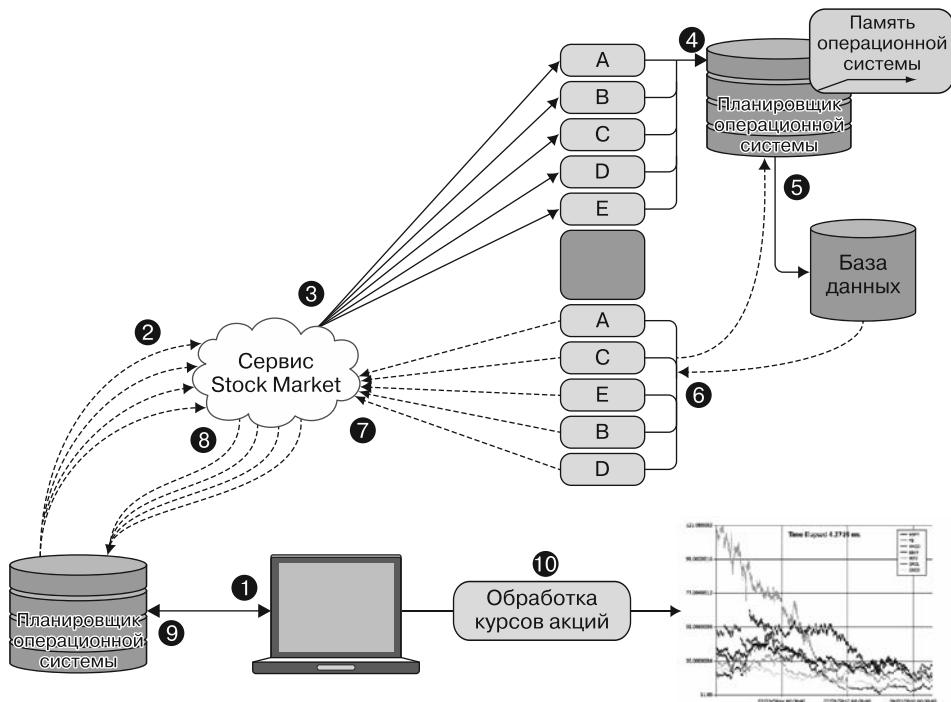


Рис. 8.7. Модель асинхронного программирования для параллельной загрузки данных из сети

Ниже представлена последовательность действий, которые выполняет служба фондового рынка при обработке запросов.

1. Пользователь асинхронно отправляет несколько запросов, чтобы параллельно загрузить историю изменений курсов акций. Пользовательский интерфейс сохраняет отзывчивость.
2. Пул потоков планирует работу. Поскольку это операции с ограничениями ввода-вывода, то количество асинхронных запросов, выполняемых параллельно, может превышать доступное количество локальных процессорных ядер.
3. Сервис Stock Market получает HTTP-запросы, и работа передается во внутреннюю программу, которая уведомляет планировщик пула потоков об асинхронной обработке входящих запросов с обращением к базе данных.
4. Поскольку код является асинхронным, планировщик пула потоков может спланировать работу так, чтобы оптимизировать использование локальных аппаратных ресурсов. Таким образом, количество потоков, необходимых для выполнения программы, сведено к минимуму, система сохраняет отзывчивость, потребление памяти остается низким, а сервер является масштабируемым.

5. Запросы к базе данных обрабатываются асинхронно без блокировки потоков.
6. Когда база данных завершает работу, результат отправляется обратно вызывающему объекту. На этом этапе планировщик пула потоков получает уведомление, и поток переназначается для выполнения остальной части работы.
7. Ответы на запросы по мере их выполнения отправляются обратно вызывающему объекту в сервис Stock Market.
8. Пользователь начинает получать ответы от сервиса Stock Market.
9. Пользовательский интерфейс получает уведомление, и выделяется поток для продолжения остальной работы без блокировок.
10. Выполняется анализ полученных данных, и по ним строится диаграмма.

Использование асинхронного подхода означает, что все операции выполняются параллельно, но общее время отклика по-прежнему определяется временем самого медленного исполнителя. При синхронном подходе, наоборот, каждый добавляемый исполнитель увеличивает время отклика.

8.5.1. Отмена асинхронных операций

При выполнении асинхронной операции полезно преждевременно завершить выполнение, прежде чем оно будет завершено по требованию. Это хорошо работает для длительных, неблокирующих операций, когда удобно запланировать возможность их отмены, чтобы избежать появления задач, которые могут зависнуть. Например, желательно иметь возможность отменить операцию загрузки изменений курсов акций, если загрузка длится больше определенного времени.

Начиная с версии 4.0, в .NET Framework представлены разнообразные удобные варианты совместной поддержки отмены операций, выполняемых в другом потоке. Данный механизм — простой и полезный инструмент для управления потоком выполнения задач. Концепция совместной отмены позволяет запросить прекращение уже запущенной операции без применения кода (рис. 8.8). Для этого код должен поддерживать отмену выполнения. Рекомендуется построить программу так, чтобы возможности отмены операций были как можно шире.

В .NET существуют следующие типы для отмены операций Task и async:

- ❑ CancellationTokenSource отвечает за создание маркера отмены и отправку запросов отмены всех копий данного маркера;
- ❑ CancellationToken — структура, используемая для контроля состояния текущего маркера.

Отмена отслеживается и запускается в .NET Framework `System.Threading.CancellationToken` с применением модели отмены.

ПРИМЕЧАНИЕ

Отмена рассматривается как особый вид исключения типа `OperationCancelledException`. Для вызывающего кода это исключение является условным сигналом, служащим уведомлением об отмене.

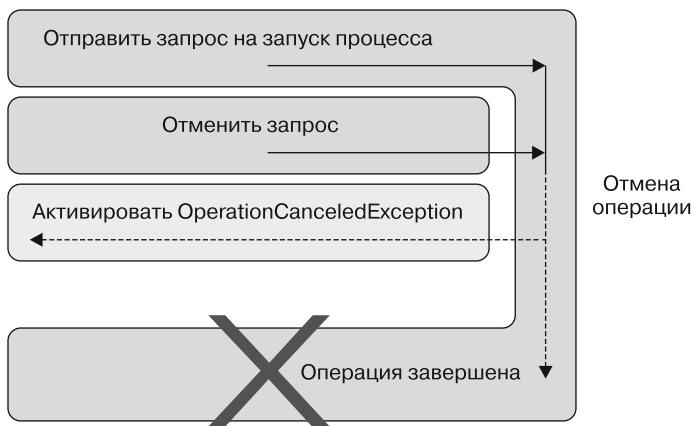


Рис. 8.8. После запроса на запуск процесса отправляется запрос аннулирования. Он останавливает оставшуюся часть выполнения операции, которая возвращается вызывающему объекту в форме OperationCanceledException

Поддержка отмены в модели TAP

В TAP поддержка отмены операций реализована изначально; фактически каждый метод, возвращающий задачу, имеет как минимум одно переопределение с маркером отмены в качестве параметра. В такой ситуации можно передать маркер отмены при создании задачи, тогда асинхронная операция будет проверять состояние маркера и отменит вычисление в случае инициации соответствующего запроса.

Чтобы отменить загрузку изменений курсов акций, нужно передать методу `Task` экземпляр `CancellationToken` в качестве аргумента, а затем вызвать метод `Cancel`. Эта технология показана в листинге 8.5 (выделена жирным шрифтом).

Листинг 8.5. Отмена асинхронной задачи

```

    Передача CancellationToken в методе для отмены
CancellationSource cts = new CancellationTokenSource(); ←

async Task<string> DownloadStockHistory(string symbol,
                                         CancellationToken token) ←
{
    string stockUrl =
        $"http://www.google.com/finance/historical?q={symbol}&output=csv";
    var request = await new HttpClient().GetAsync(stockUrl, token); ←
    return await request.Content.ReadAsStringAsync();
} ←
    Создание экземпляра
    CancellationTokenSource
    Передача CancellationToken
    в метод для отмены
    Активация маркера отмены
    cts.Cancel(); ←

```

Некоторые методы программирования изначально не поддерживают отмену. В таких случаях важно проверять их вручную. В листинге 8.6 показано, как встроить

поддержку отмены в предыдущем примере анализа фондового рынка, где нет асинхронных методов с возможностью преждевременного прекращения операций.

Листинг 8.6. Проверка и отмена асинхронной операции вручную

```
List<Task<Tuple<string, StockData[]>>> stockHistoryTasks =
    stockSymbols.Select(async symbol => {
        var url =
            $"http://www.google.com/finance/historical?q={symbol}&output=csv";
        var request = HttpWebRequest.Create(url);
        using (var response = await request.GetResponseAsync())
        using (var reader = new StreamReader(response.GetResponseStream()))
        {
            token.ThrowIfCancellationRequested();

            var csvData = await reader.ReadToEndAsync();
            var prices = await ConvertStockHistory(csvData);

            token.ThrowIfCancellationRequested();
            return Tuple.Create(symbol, prices.ToArray());
        }
    }).ToList();
```

В тех случаях, когда метод `Task` не имеет встроенной поддержки отмены, рекомендуемый шаблон — добавить несколько маркеров `CancellationToken` в виде параметров асинхронного метода и регулярно проверять, не произошла ли отмена. Затем удобнее всего сообщить об ошибке с помощью метода `ThrowIfCancellationRequested`, поскольку операция завершится без возвращения результата.

Любопытно, что в листинге 8.7 маркер `CancellationToken` (выделен жирным шрифтом) поддерживает регистрацию обратного вызова, который будет выполнен сразу после получения сообщения об отмене. В этом листинге метод `Task` загружает содержимое с сайта Manning и отменяется сразу после использования маркера отмены.

Листинг 8.7. Обратный вызов маркера отмены

```
CancellationTokenSource tokenSource = new CancellationTokenSource();
CancellationToken token = tokenSource.Token;

Task.Run(async () =>
{
    var webClient = new WebClient();
    token.Register(() => webClient.CancelAsync()); ← Регистрация обратного вызова,
    var data = await webClient                                который отменяет загрузку
        .DownloadDataTaskAsync("http://www.manning.com");   экземпляра WebClient
}, token);

tokenSource.Cancel();
```

В этом коде обратный вызов зарегистрирован для того, чтобы остановить выполнение базовой асинхронной операции в случае активации `CancellationToken`.

Данный полезный шаблон открывает возможность журналирования отмены и запуска события, позволяющего уведомить обработчик событий о том, что операция была отменена.

Поддержка совместной отмены

Использование `CancellationTokenSource` упрощает создание комбинированного маркера, состоящего из нескольких других маркеров. Такой шаблон полезен, когда есть несколько возможных причин отмены операции. Это могут быть нажатие кнопки, уведомление от системы или передача сообщения об отмене от другой операции. Метод `CancellationTokenSource.CreateLinkedTokenSource` генерирует источник отмены. Он сработает, если сработает любой из указанных маркеров (код, на который следует обратить внимание, в листинге 8.8 выделен жирным шрифтом).

Листинг 8.8. Маркеры совместной отмены

```
Несколько экземпляров CancellationToken
CancellationTokenSource ctsOne = new CancellationTokenSource();
CancellationTokenSource ctsTwo = new CancellationTokenSource();
CancellationTokenSource ctsComposite = CancellationTokenSource.
    CreateLinkedTokenSource(ctsOne.Token, ctsTwo.Token);

CancellationToken ctsCompositeToken = ctsComposite.Token;
Создание комбинированного
Task.Factory.StartNew(async () => {
    var webClient = new WebClient();
    ctsCompositeToken.Register(() => webClient.CancelAsync());

    var data = await webClient
        .DownloadDataTaskAsync(http://www.manning.com);
}, ctsComposite.Token); ←
Комбинированный маркер отмены передается так же,
как обычный; затем задача отменяется путем вызова
метода Cancel() любым из маркеров, образующих составной маркер
```

В этом листинге на основе двух маркеров отмены создается связанный источник отмены. Затем используется новый составной маркер. Он будет отменен, если будет отменен любой из первоначальных маркеров. Маркер отмены — это, в сущности, потокобезопасный флаг (логическое значение), который уведомляет своего владельца о том, что операция `CancellationTokenSource` была отменена.

8.5.2. Асинхронная компоновка на основе задач с монадическим оператором Bind

Как уже отмечалось, `async Task<T>` — это монадический тип, другими словами — контейнер, в котором можно применять монадические операторы `Bind` и `Return`. Проанализируем, чем данные функции могут быть полезны при написании программы. В листинге 8.9 оператор `Bind` позволяет объединить последовательность

асинхронных операций в цепочку вычислений. Оператор `Return` возвращает значение в монаду (контейнер или надтип).

ПРИМЕЧАНИЕ

Напомню: оператор `Bind` применяется к асинхронному типу `Task<T>`, что позволяет объединить две асинхронные операции в конвейер, передавая результат первой операции, когда он станет доступным, во вторую.

В общем случае асинхронная функция `Task` принимает аргумент произвольного типа '`T`', возвращает вычисление типа `Task<'R>` (с сигнатурой `T -> Task<'R>`) и может быть скомпонована посредством оператора `Bind`. Данный оператор говорит следующее: «Когда будет вычислено значение '`R`' из функции (`g: 'T -> Task<'R>`), этот результат надо передать в функцию (`f: 'R -> Task<'U>`)».

Функция `Bind` показана на рис. 8.9 для наглядности, поскольку она встроена в систему.

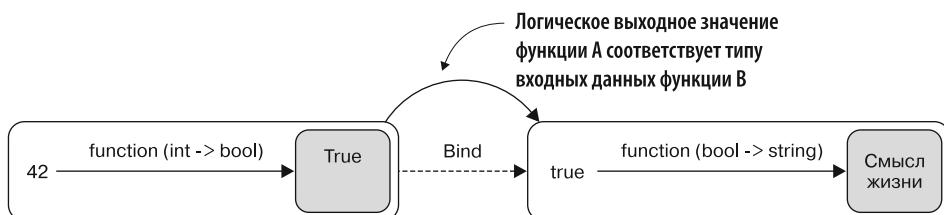


Рис. 8.9. Оператор `Bind` компонует две функции, которые возвращают результат, обернутый в тип `Task`; значение, возвращаемое после вычисления первой задачи, соответствует формату входных данных второй функции

С помощью этой функции `Bind` (выделенной в листинге жирным шрифтом) можно упростить структуру кода в программе анализа курсов акций. Идея заключается в том, чтобы объединить несколько функций.

Листинг 8.9. Использование оператора `Bind`

```
async Task<Tuple<string, StockData[]>> ProcessStockHistory(string symbol)
{
    return await DownloadStockHistory(symbol)
        .Bind(stockHistory => ConvertStockHistory(stockHistory))
        .Bind(stockData => Task.FromResult(Tuple.Create(symbol,
                                                       stockData)));
}
```

Компоновка асинхронной операции в стиле продолжений

Асинхронные вычисления `Task` скомпонованы путем вызова оператора `Bind` в первой операции `async`, передачи результата во вторую асинхронную операцию и т. д. Результатом является асинхронная функция, аргументом которой выступает значение, возвращаемое первой задачей, когда она будет завершена. Эта функция возвращает вторую задачу, которая использует результат первой задачи в качестве входных данных для своих вычислений.

Такой код и декларативен, и выразителен, поскольку полностью соответствует функциональной парадигме. Теперь мы задействовали монадический оператор — тот, что основан на монаде продолжения.

8.5.3. Отсрочка асинхронного вычисления, обеспечивающая компоновку

В модели ТАР, реализованной в C#, функция, возвращающая задачу, начинает выполнение немедленно. Такое поведение с немедленным выполнением асинхронного выражения называется *горячей задачей*, которая, к сожалению, негативно влияет на возможности компоновки. Функциональный способ обработки асинхронных операций состоит в том, чтобы отложить выполнение до тех пор, пока это не будет необходимо, что позволяет использовать компоновку и более тонко управлять процессом выполнения.

Существуют следующие три варианта реализации АРМ.

- Горячие задачи. Асинхронный метод возвращает задачу, представляющую уже выполняемое задание, которое в итоге выдаст значение. Эта модель применяется в C#.
- Холодные задачи. Асинхронный метод возвращает задачу, которую вызывающий объект должен явно запустить. Эта модель часто используется в традиционном подходе на основе потоков.
- Генераторы задач. Асинхронный метод возвращает задачу, которая в итоге выдаст значение. Выполнение такой задачи начнется тогда, когда будет предоставлено продолжение. Это предпочтительный вариант для функциональной парадигмы, поскольку он позволяет избежать побочных эффектов и изменений. (Подобная модель применяется в F# для запуска асинхронных вычислений.)

Как выполнить асинхронную операцию по запросу, используя модель ТАР в C#? Можно задействовать тип `Lazy<T>` в качестве оболочки для вычисления `Task<T>` (см. главу 2), но будет проще перенести асинхронное вычисление в делегат `Func<T>`, который будет выполнять базовую операцию только при явном запуске. В следующем фрагменте кода эта концепция применяется к примеру с изменениями курса акций, где определяется функция `onDemand` для «ленивого» вычисления `Task`-выражения `TaskStockHistory`:

```
Func<Task<string>> onDemand = async () => await ownloadStockHistory("MSFT");  
string stockHistory = await onDemand();
```

С точки зрения кода, чтобы получить основную задачу асинхронного выражения `DownloadStockHistory`, нужно получить и явно выполнить функцию `onDemand` как обычную `Func` с помощью оператора `()`.

Обратите внимание: в этом коде есть небольшой дефект. Функция `onDemand` выполняет асинхронное вычисление, у которого должен быть предварительно зафиксированный аргумент (в данном случае `"MSFT"`).

Но как передать функции другой код акций? Решение заключается в каррировании и частичном применении — в ФП-технологиях, которые упрощают многократное

использование абстрактных функций, поэтому их можно специализировать. (Подробнее это объясняется в приложении А.)

Каррирование и частичное применение

В языках функционального программирования функция *каррируется*, когда она должна иметь несколько параметров, но принимает только один, а возвращает функцию, которая принимает следующий параметр, и т. д. Например, функция с сигнатурой $A \rightarrow B \rightarrow C$ принимает один аргумент A и возвращает функцию $B \rightarrow C$. При переводе в код C# с использованием делегатов эта функция определяется как `Func<A, Func<B, C>>`,

Этот механизм позволяет применить функцию частично, вызвав ее с несколькими параметрами и создав новую функцию, которая применяется только к переданным аргументам. Одна и та же функция может иметь разные интерпретации в зависимости от количества переданных параметров.

Ниже представлена каррированная версия функции `onDemand`, которая принимает строку (символ) в качестве аргумента, затем передает эту строку во внутреннее выражение `Task` и возвращает функцию типа `Func<Task<string>>`:

```
Func<string, Func<Task<string>>> onDemandDownload = symbol =>
    async () => await DownloadStockHistoryAsync(symbol);
```

Теперь эта каррированная функция может быть частично использована для создания *специализированных* функций для заданной строки (в нашем случае к коду акции); эта строка затем будет потреблена обернутой задачей `Task` при выполнении функции `onDemand`. Ниже представлена частично примененная функция для создания специализированной функции `onDemandDownloadMSFT`:

```
Func<Task<string>> onDemandDownloadMSFT = onDemandDownload("MSFT");
string stockHistoryMSFT = await onDemandDownloadMSFT();
```

Такая технология разделения асинхронных операций показывает, что можно строить произвольную, сколь угодно сложную логику, не выполняя ничего, пока вы не решите запустить процесс.

8.5.4. Повторная попытка в случае, если что-то пошло не так

Общей проблемой при работе с асинхронными операциями ввода-вывода и особенно с сетевыми запросами является появление неожиданных факторов, ставящих под угрозу успешное завершение операций. В таких ситуациях желательно иметь возможность повторить операцию, если предыдущая попытка не удалась. Например, в ходе выполнения HTTP-запроса методом `DownloadStockHistory` могут возникнуть такие проблемы, как плохое качество подключения к Интернету или недоступные удаленные серверы. Но эти проблемы могут носить временный характер, и опера-

ция, которая однажды не сработала, может завершиться успешно, если повторить ее через несколько минут.

Шаблон нескольких попыток является обычной практикой решения временных проблем. В контексте асинхронных операций данная модель достигается путем создания функции-обертки, реализуемой посредством ТАР и возвращающей задачи. Это изменяет выполнение асинхронного выражения, показанного в предыдущем разделе. Затем, если существует несколько проблем, эта функция применяет логику повторного вызова заданное количество раз с заданной задержкой между попытками. В листинге 8.10 показана реализация асинхронной функции `Retry` как метода расширения.

Листинг 8.10. Асинхронная операция `Retry`

```
Если маркер не передан, то маркеру по умолчанию
(CancellationToken) присваивается значение CancellationToken.None           CancellationToken cts используется
                                                               для остановки текущего выполнения

async Task<T> Retry<T>(Func<Task<T>> task, int retries, TimeSpan delay,
    CancellationToken cts = default(CancellationToken)) =>
    await task().ContinueWith(async innerTask => {
        cts.ThrowIfCancellationRequested();
        if (innerTask.Status != TaskStatus.Faulted)
            return innerTask.Result;           ← Возвращает результат, если
        if (retries == 0)                  | асинхронная операция была успешна
            throw innerTask.Exception ?? throw new Exception();   ← Задерживает выполнение
        await Task.Delay(delay, cts);      | асинхронной операции,
        return await Retry(task, retries - 1, delay, cts);       | если произошел сбой
    }).Unwrap();                      ← Повторение асинхронной операции, при этом
                                    | счетчик повторов уменьшается на единицу

Если достигнут предел повторов,
то функция выдаст исключение
```

Первым аргументом является асинхронная операция, которая будет выполняться повторно. Эта функция описана как «ленивая», ее выполнение обернуто в `Func<>`, потому что вызов операции приводит к немедленному запуску задачи. При возникновении исключения операция `Task<T>` перехватывает обработку ошибок через свойства `Status` и `Exception`. Для того чтобы убедиться, что выполнить асинхронную операцию действительно не удалось, нужно проверить эти свойства. Если операция завершилась неудачно, вспомогательная функция `Retry` выдерживает заданный интервал и повторяет ту же операцию, уменьшая счетчик попыток, — и т. д., пока значение счетчика не станет равно нулю. С помощью этой вспомогательной функции `Retry<T>` можно перестроить функцию `DownloadStockHistory` следующим образом для выполнения операции веб-запроса с учетом логики повторов:

```
async Task<Tuple<string, StockData[]>> ProcessStockHistory(string symbol)
{
    string stockHistory =
        await Retry(() => DownloadStockHistory(symbol), 5,
                    TimeSpan.FromSeconds(2));
    StockData[] stockData = await ConvertStockHistory(stockHistory);
    return Tuple.Create(symbol, stockData);
}
```

В данном случае логика повторов должна выполняться не более пяти раз с задержкой 2 с между попытками. Вспомогательная функция `Retry<T>` обычно размещается в конце рабочего процесса.

8.5.5. Обработка ошибок в асинхронных операциях

Как мы помним, большинство асинхронных операций имеют ограничения ввода-вывода; существует высокая вероятность того, что во время их выполнения что-то пойдет не так. В предыдущем разделе было рассмотрено решение для устранения сбоев путем применения логики повтора. Другой подход заключается в объявлении функционального комбинатора, который связывает асинхронную операцию с резервной операцией. Если первая операция завершится неудачей, то вступит в силу резервная функция. Важно объявить резервную функцию как задачу, выполняемую другим способом. В листинге 8.11 показан код, в котором определен комбинатор `Otherwise`, принимающий две задачи и передающий выполнение во вторую, если первая завершится неудачно.

Листинг 8.11. Комбинатор резервных задач

```
static Task<T> Otherwise<T>(this Task<T> task,
                               Func<Task<T>> otherTask) ←
    => task.ContinueWith(async innerTask => {
        if (innerTask.Status == TaskStatus.Faulted) return await orTask();
        return innerTask.Result;
    }).Unwrap();
```

otherTask обернута в `Func<T>`
для выполнения по требованию

Если `innerTask` завершится неудачно,
то будет выполнена задача `orTask`

Для завершения задачи в типе `Task` реализована концепция успешного или неуспешного завершения. Это отображается в свойстве `Status`, которое принимает значение `TaskStatus.Faulted` при возникновении исключения во время выполнения задачи. Мы можем поменять пример с анализом изменений курса акций в стиле ФП, применив комбинатор `Otherwise`.

В листинге 8.12 приведен код, в котором объединены поведение повтора, комбинатор `Otherwise` и монадические операторы для компоновки асинхронных операций.

Обратите внимание, что метод расширения `ConfigureAwait` в коде отсутствует. Комбинатор `Otherwise` запускает функцию `DownloadStockHistory` для обеих асинхронных операций, первичной и резервной. В стратегии резервирования используются одни и те же функции для загрузки курсов акций, но веб-запросы указывают на разные конечные точки сервисов (URL). Если первый сервис оказался недоступен, то применяется второй.

Эти две конечные точки предоставляются функциями `googleSourceUrl` и `yahooSourceUrl`, которые строят URL-адреса для HTTP-запроса. Такой подход требует измене-

ния сигнатуры функции `DownloadStockHistory`, которая теперь является функцией более высокого порядка `Func<string, string> sourceStock`. Эта функция частично применяется к функциям `googleSourceUrl` и `yahooSourceUrl`. В результате возникают две новые функции, `googleService` и `yahooService`. Они передаются в качестве аргументов в комбинатор `Otherwise`, который в итоге обертывается в логику `Retry`. Затем операторы `Bind` и `Map` используются для компоновки операций в виде рабочего процесса — и все это не выходя за пределы `async Task`. Все операции гарантированно будут полностью асинхронными.

Листинг 8.12. Использование комбинатора `Otherwise` для резервного поведения

```

Вспомогательная функция, которая генерирует
конечные точки для извлечения
изменений курса акций с заданным кодом

→ Func<string, string> googleSourceUrl = (symbol) =>
    $"http://www.google.com/finance/historical?q={symbol}&output=csv";

→ Func<string, string> yahooSourceUrl = (symbol) =>
    $"http://ichart.finance.yahoo.com/table.csv?s={symbol}";

async Task<string> DownloadStockHistory(Func<string, string> sourceStock,
                                         string symbol)
{
    string stockUrl = sourceStock(symbol); ← Генерирование конечной точки shareUrl
    var request = WebRequest.Create(stockUrl); ← с использованием переданной функции sourceStock
    using (var response = await request.GetResponseAsync())
    using (var reader = new StreamReader(response.GetResponseStream()))
        return await reader.ReadToEndAsync();
} ← Создание производной функции DownloadStockHistory для частичного
      применения вспомогательной функции конечных точек и кода акции

async Task<Tuple<string, StockData[]>> ProcessStockHistory(string symbol)
{
    Func<Func<string, string>, Func<string, Task<string>>> downloadStock =
        service => stock => DownloadStockHistory(service, stock); ←
            service => stock => DownloadStockHistory(service, stock); ← Частичное применение
            Func<string, Task<string>> googleService =
                downloadStock(googleSourceUrl); ← функции DownloadStockHistory.
            Func<string, Task<string>> yahooService =
                downloadStock(yahooSourceUrl); ← downloadStock создает сервис
                                                изменений курсов акций
    return await Otherwise(() => googleService(symbol)) ← Функция Retry применяет
        .Retry(()=> yahooService(symbol)), 5, TimeSpan.FromSeconds(2) ←
        .OperadorOtherwise выполняет
        .Bind(data => ConvertStockHistory(data)) операцию googleService; если это не удаётся,
        .Map(prices => Tuple.Create(symbol, prices)); ← то выполняется операция yahooService

}
} ← Использование оператора-функциона Map
      Монадический оператор Bind объединяет
      две асинхронные операции Task, Retry и ConvertStockHistory
      для преобразования результата

```

8.5.6. Асинхронная параллельная обработка изменений фондового рынка

Поскольку функция `Task` описывает операции, выполнение которых занимает значительное время, вполне логично желание выполнять их параллельно, насколько это возможно. В примере с кодом анализа изменений курса акций есть один интересный нюанс. При материализации LINQ-выражения асинхронный метод `ProcessStockHistory` выполняется в цикле `for-each`, вызывая задачи по очереди и ожидая результата. Подобные вызовы являются неблокирующими, но поток выполнения — последовательный; каждая задача ожидает завершения предыдущей, и только потом запускается сама. Это неэффективно.

В фрагменте далее показано ошибочное поведение при последовательном запуске асинхронных операций с использованием цикла `for-each`:

```
async Task ProcessStockHistory (string[] stockSymbols)
{
    var sw = Stopwatch.StartNew();
    IEnumerable<Task<Tuple<string, StockData[]>>> stockHistoryTasks =
        stockSymbols.Select(stock => ProcessStockHistory(stock));

    var stockHistories = new List<Tuple<string, StockData[]>>();

    foreach (var stockTask in stockHistoryTasks)
        stockHistories.Add(await stockTask);

    ShowChart(stockHistories, sw.ElapsedMilliseconds);
}
```

Предположим, что теперь мы хотим запустить эти вычисления параллельно и визуализировать диаграмму, когда все будет завершено. Данная структура аналогична шаблону `Fork/Join`. Здесь несколько асинхронных вычислений будут запускаться параллельно. Затем программа станет ожидать завершения всех операций, после чего результаты будут объединены, и выполнение продолжится. В листинге 8.13 показана параллельная обработка акций.

В этом листинге коллекция акций преобразуется в список задач с использованием асинхронного лямбда-выражения в LINQ-методе `Select`. Важно материализовать LINQ-выражение, вызывая функцию `ToList()`, которая отправляет задачи на параллельное выполнение только один раз. Это возможно благодаря свойству горячей задачи, которое означает, что задача будет выполняться сразу после ее определения.

СОВЕТ

По умолчанию в .NET допускается одновременное существование не более двух соединений с открытым запросом; чтобы ускорить процесс, нужно изменить значение предельного количества соединений `ServicePointManager.DefaultConnectionLimit = stocks.Length`.

Листинг 8.13. Параллельный анализ изменения курсов акций

```
async Task ProcessStockHistory( )
{
    var sw = Stopwatch.StartNew();
    string[] stocks = new[] { "MSFT", "FB", "AAPL", "YHOO",
                             "EBAY", "INTC", "GOOG", "ORCL" };

    List<Task<Tuple<string, StockData[]>>> stockHistoryTasks =
        stocks.Select(async stock => await
            ProcessStockHistory(stock)).ToList(); ←

    Tuple<string, StockData[]>[] stockHistories =
        await Task.WhenAll(stockHistoryTasks); →
    ShowChart(stockHistories, sw.ElapsedMilliseconds);
}

Асинхронное, без блокировок
ожидание выполнения всех задач
Оператор List гарантирует материализацию LINQ-запроса, который,
в свою очередь, параллельно выполняет основные операции
```

Метод `Task.WhenAll` (аналогичный `Async.Parallel` в F#) является частью TPL, и его назначение состоит в объединении результатов нескольких задач в один массив задач, а затем в асинхронном ожидании для завершения работы:

```
Tuple<string, StockData[]>[] result = await Task.WhenAll(stockHistoryTasks);
```

В данном случае время выполнения программы в результате упало с 4,272 до 0,534 с.

8.5.7. Асинхронная параллельная обработка данных фондового рынка после завершения выполнения задач

Есть и другое, лучшее решение: вместо того, чтобы ожидать завершения загрузки всех курсов акций, обрабатывать результаты изменений курсов акций по мере их поступления. Это хороший шаблон для повышения производительности. В данном случае он также сокращает нагрузку на поток пользовательского интерфейса, позволяя визуализировать данные по частям. Рассмотрим код анализа фондового рынка, где несколько фрагментов данных об изменениях курса акций загружаются из Интернета, а затем используются для обработки изображения при визуализации в пользовательском интерфейсе. Если ожидать, пока будут проанализированы все данные, и только потом обновлять пользовательский интерфейс, то программа будет вынуждена выполнять последовательную обработку в потоке пользовательского интерфейса. В листинге 8.14 показано более рациональное решение, которое состоит в том, чтобы обрабатывать и обновлять диаграмму по возможности конкурентно. Технически этот шаблон называется *чредованием*. Важный код выделен жирным шрифтом.

Листинг 8.14. Обработка результатов анализа фондового рынка по мере завершения каждой задачи

```
async Task ProcessStockHistory()
{
    var sw = Stopwatch.StartNew();
    string[] stocks = new[] { "MSFT", "FB", "AAPL", "YHOO",
                             "EBAY", "INTC", "GOOG", "ORCL" };

    List<Task<Tuple<string, StockData[]>>> stockHistoryTasks =
        stocks.Select(ProcessStockHistory).ToList();
```

ToList() материализует LINQ-выражение, гарантируя параллельное выполнение основных задач

```
while (stockHistoryTasks.Count > 0) ← Запуск выполнения в цикле while,
{                                         пока не будут реализованы все асинхронные задачи
```

↑

```
    Task<Tuple<string, StockData[]>> stockHistoryTask =
        await Task.WhenAny(stockHistoryTasks); ←
```

↑

```
    stockHistoryTasks.Remove(stockHistoryTask);
    Tuple<string, StockData[]> stockHistory = await stockHistoryTask;
```

↑

```
    ShowChartProgressive(stockHistory); ← Удаление завершенной операции
}                                         из списка, который используется в предикате цикла while
```

↑

```
Оператор Task.WhenAny асинхронно
ожидает завершения первой операции
```

↑

```
Отправка результата асинхронной операции,
который будет отображаться на диаграмме
```

По сравнению с предыдущей версией в этот код внесены два изменения:

- ❑ цикл `while` удаляет задачи по мере их поступления, пока не останется ни одной;
- ❑ вместо `Task.WhenAll` используется `Task.WhenAny`. Этот метод асинхронно ожидает завершения первой задачи и возвращает свой экземпляр.

В данной реализации не учтены ни исключения, ни отмены. Вместо этого можно проверить состояние задачи `taskHistoryTask` перед дальнейшей обработкой и применением условной логики.

Резюме

- ❑ В .NET можно писать асинхронные программы посредством асинхронного программирования на основе задач (Task-based Asynchronous Programming, TAP) на C#, что является предпочтительной моделью.
- ❑ Модель асинхронного программирования позволяет эффективно выполнять большое количество параллельных операций ввода-вывода, грамотно освобождая ресурсы во время их простоя и избегая создания новых ресурсов, тем самым оптимизируя потребление памяти и повышая производительность.
- ❑ Тип `Task<T>` представляет собой монадическую структуру данных, что означает, кроме прочего, то, что он легко может быть скомпонован с другими задачами декларативным и эффективным способом.
- ❑ Асинхронные задачи могут выполняться и компоноваться с использованием монадических операторов, что приводит к семантике в LINQ-стиле. Преимуществом такого подхода является четкий и понятный декларативный стиль программирования.
- ❑ Выполнение сравнительно длительных операций с применением асинхронных задач позволяет повысить производительность и отзывчивость приложения, особенно если оно опирается на один или несколько удаленных сервисов.
- ❑ Количество асинхронных вычислений, которые могут выполняться параллельно одновременно, не зависит от количества доступных процессоров, а время выполнения определяется периодом ожидания завершения операций ввода-вывода, ограниченных только возможностями драйверов ввода-вывода.
- ❑ В основе модели ТАР лежит тип `Task`, обогащенный ключевыми словами `async` и `await`. Эта асинхронная модель программирования реализует функциональную парадигму в форме использования стиля продолжений (CPS).
- ❑ С помощью ТАР можно легко реализовать эффективные шаблоны, такие как параллельная загрузка нескольких ресурсов и их обработка по мере доступности, вместо того чтобы ожидать окончания загрузки всех ресурсов.



Асинхронное функциональное программирование на F#

В этой главе:

- взаимодействие с асинхронными вычислениями;
- реализация асинхронных операций в функциональном стиле;
- расширение рабочего процесса посредством асинхронных вычислительных выражений;
- согласование параллелизма с асинхронными операциями;
- координация отмены параллельных асинхронных вычислений.

В главе 8 асинхронное программирование было описано как задача (task), выполняемая независимо от основного потока приложения, возможно, в отдельной вычислительной среде или по сети, на разных процессорах. Этот метод приводит к параллелизму, когда приложения способны выполнять огромное количество операций ввода-вывода на одноядерной машине. С точки зрения скорости выполнения программы и передачи данных это очень многообещающая идея, позволяющая отказаться от традиционного пошагового подхода к программированию.

В языках программирования F# и C# реализована немного другая, но тоже весьма элегантная абстракция для представления асинхронных вычислений, что делает их идеальными инструментами, отлично подходящими для моделирования реальных проблем. В главе 8 вы увидели, как можно использовать асинхронную модель программирования на C#. В этой главе мы рассмотрим, как сделать то же самое на F#. Данная глава поможет вам понять семантику производительности асинхронного рабочего процесса на F#, чтобы писать эффективные высокопроизводительные программы для обработки операций с ограничениями ввода/вывода.

Мы обсудим подход, реализованный в F#, проанализируем его уникальные особенности и то, как они влияют на структуру кода, я покажу, как можно легко реализовать и компоновать эффективные асинхронные операции в функциональном стиле. Я также научу вас писать неблокирующие операции ввода-вывода, чтобы повысить общую производительность, эффективность и пропускную способность приложений при конкурентном выполнении нескольких асинхронных операций, не беспокоясь об аппаратных ограничениях.

Вы сами увидите, что можно применять функциональные концепции для написания асинхронных вычислений. Затем вы поймете, как использовать эти концепции для обработки побочных эффектов и взаимодействия с реальным миром, сохраняя преимущества семантики компоновки, так, что код останется кратким, понятным и легко поддерживаемым. В конце этой главы вы узнаете, как следует реализовать параллелизм в современных приложениях и использовать возможности многоядерных процессоров для эффективной работы и эффективного управления большим количеством операций в функциональном стиле.

9.1. Аспекты асинхронного функционального программирования

Асинхронная функция — это структурный элемент, означающий обычную функцию или метод F#, который возвращает асинхронное вычисление. Современные модели асинхронного программирования, такие как асинхронный рабочий процесс F# и `async/await` в C#, являются функциональными, поскольку применение функционального программирования позволяет опытному программисту писать простой и декларативный процедурный код, который выполняется асинхронно и параллельно.

С самого начала F# поддерживал применение семантического определения асинхронного программирования, которое напоминало синхронный код. Не случайно несколько функциональных элементов, реализованных впоследствии в C#, появились в этом языке под влиянием функционального подхода F# с его асинхронным рабочим процессом для реализации асинхронной модели `async/await`, заменившей традиционную императивную модель APM. Более того, и асинхронная задача C#, и асинхронный рабочий процесс F# являются монадическими контейнерами, которые упрощают разделение общей функциональности на универсальные, многократно используемые компоненты.

9.2. Что такое асинхронный рабочий процесс F#

Язык функционального программирования F# обеспечивает полную поддержку асинхронного программирования:

- ❑ он интегрируется с моделью асинхронного программирования, предоставляемой .NET;
- ❑ он предлагает идиоматическую функциональную реализацию APM;
- ❑ он поддерживает совместимость с моделью программирования на основе задач C#.

Асинхронный рабочий процесс в F# предназначен для удовлетворения требований функциональной парадигмы; он повышает возможности компоновки, обеспечивает простоту и выразительность неблокирующих вычислений путем сохранения последовательной структуры кода. По определению асинхронный рабочий процесс построен на вычислительных выражениях, общем компоненте ядра языка F#, который обеспечивает монадическую семантику для описания последовательности операций в стиле продолжений (CPS).

Ключевой особенностью асинхронного рабочего процесса является сочетание неблокирующих вычислений с простой асинхронной семантикой, которая напоминает линейный поток управления.

9.2.1. Стиль прохождения продолжений в вычислительных выражениях

Многопоточный код, как известно, устойчив к императивному стилю написания программ. Но, используя CPS и функциональную парадигму, можно сделать код гораздо короче и проще в написании. Представьте себе, что вы пишете программы на старой версии .NET Framework, в которой нет модели программирования `async/await` (см. главу 8). В этом случае вам нужно выполнить серию операций `Task`, где входные данные каждой операции зависят от результата предыдущей; код может стать сложным и запутанным. В следующем примере программа загружает изображение из хранилища Azure Blob и сохраняет полученные байты в файл.

Для простоты код, который не имеет значения для примера, умышленно опущен; код, на который необходимо обратить внимание, выделен жирным шрифтом. Полную реализацию программы вы найдете в исходном коде, выложенном на сайте издательства.

```
let downloadCloudMediaBad destinationPath (imageReference : string) =
    log "Creating connection..."
    let taskContainer = Task.Run<CloudBlobContainer>(fun () ->
        getCloudBlobContainer())
    log "Get blob reference...";
    let container = taskContainer.Result
    let taskBlockBlob = Task.Run<CloudBlob>(fun () ->
        container.GetBlobReference(imageReference))
    log "Download data..."
    let blockBlob = taskBlockBlob.Result
    let bytes = Array.zeroCreate<byte> (int blockBlob.Properties.Length)
    let taskData = Task.Run<byte[]>(fun () -> blockBlob.
        DownloadToByteArray(bytes, 0)|>ignore; bytes)
    log "Saving data..."
    let data = taskData.Result
    let taskComplete = Task.Run(fun () ->
        File.WriteAllBytes(Path.Combine(destinationPath,imageReference), data))
    taskComplete.Wait()
    log "Complete"
```

Конечно, это утрированный пример, призванный проиллюстрировать тот факт, что использование традиционных инструментов (и устаревшего мышления) для написания конкурентного кода приводит к появлению многословных и непрактичных программ. Разумеется, неопытному разработчику легче писать код в таком стиле, потому что ему проще рассуждать последовательно. Но результатом станет немасштабируемая программа, в которой каждое вычисление `Task` будет вызывать метод экземпляра `Result`, что является плохой практикой. В данной ситуации для того, чтобы решить проблему масштабируемости, достаточно хоть немного знать CPS. Для этого мы сначала определим функцию, используемую для объединения операций в конвейер:

```
let bind(operation:unit -> 'a, continuation:'a -> unit) =
    Task.Run(fun () -> continuation(operation())) |> ignore
```

Функция `bind` принимает функцию продолжения (`'a -> unit`), которая вызывается, когда готов результат операции (`unit -> 'a`). Главное здесь заключается в том, что при этом не блокируется вызывающий поток, который, следовательно, может продолжить выполнение полезного кода. Когда результат будет готов, вызывается продолжение, позволяющее продолжить вычисление. Теперь можно воспользоваться этой функцией `bind` и создать более быструю версию предыдущего кода:

```
let downloadCloudMediaAsync destinationPath (imageReference : string) =
    bind( (fun () -> log "Creating connecton..."; getCloudBlobContainer()),
        fun connection ->
            bind( (fun () -> log "Get blob reference...";
                    connection.GetBlobReference(imageReference)),
                fun blockBlob ->
                    bind( (fun () -> log "Download data..."
                            let bytes = Array.zeroCreate<byte> (int
→ blockBlob.Properties.Length)
                            blockBlob.DownloadToByteArray(bytes, 0) |> ignore
                            bytes), fun bytes ->
                        bind( (fun () -> log "Saving data...";
                                File.WriteAllBytes(Path.Combine(destinationPath,imageReference),
→ bytes)), fun () -> log "Complete"))))

["Bugghina01.jpg"; "Bugghina02.jpg"; "Bugghina003.jpg"] |> Seq.iter
downloadCloudMediaAsync "Images")
```

Запустив код, вы заметите, что функция `bind` выполняет базовую анонимную лямбда-операцию в отдельном потоке. Каждый раз, когда вызывается функция `bind`, поток извлекается из пула потоков, а затем, когда функция завершается, поток возвращается назад в пул.

Асинхронный рабочий процесс F# основан на той же концепции CPS, что и при моделировании вычислений, которые трудно выполнять последовательно.

ПРИМЕЧАНИЕ

Асинхронные функции на протяжении их времени жизни можно перебрасывать между любым количеством потоков.

На рис. 9.1 показано сравнение между синхронно и асинхронно обрабатываемыми входящими запросами.

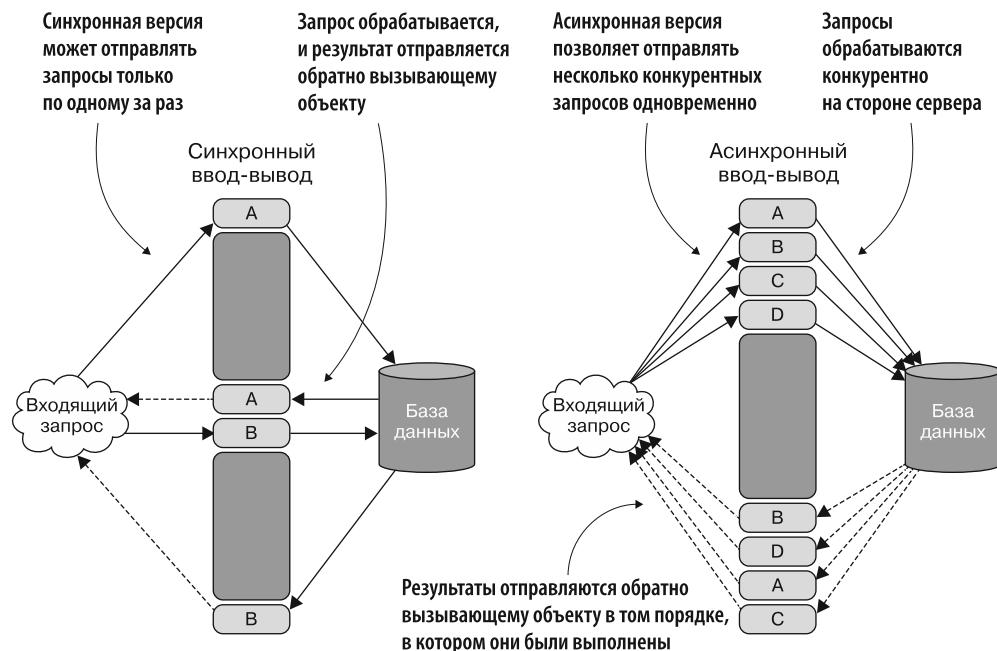


Рис. 9.1. Сравнение операционных систем с синхронными (блокирующими) и асинхронными (неблокирующими) операциями ввода-вывода. Синхронная версия может отправлять запросы только по одному; после обработки запроса результат отправляется обратно вызывающему объекту. Асинхронная версия позволяет отправлять несколько конкурентных запросов одновременно; после того как эти запросы будут конкурентно обработаны на стороне сервера, они отправляются обратно вызывающему объекту в том порядке, в котором были выполнены

Асинхронный рабочий процесс F# также включает в себя функции-продолжения для обработки отмены и исключения. Прежде чем углубиться в детали асинхронного рабочего процесса, рассмотрим следующий пример.

9.2.2. Асинхронный рабочий процесс в действии: параллельные операции сохранения данных из Azure Blob

Предположим, что ваш начальник решил, что цифровые медиакомпании должны храниться не только локально, но и в облаке. С этой целью он попросил вас создать простое средство для загрузки и скачивания, а также синхронизации и проверки того, что нового появилось в облаке. Для того чтобы в этом сценарии медиафайлы были представлены в виде двоичных данных, вы разрабатываете программу

загрузки набора изображений из сетевого хранилища Azure Blob и визуализирует эти изображения в клиентском приложении, разработанном на основе WPF. Хранилище Azure Blob (<http://mng.bz/X1FB>) — это облачный сервис Microsoft, хранящий неструктурированные данные в формате blobs (binary large objects — «большие двоичные объекты»). Эта служба позволяет хранить данные любого типа, что отлично подходит для хранения медиафайлов вашей компании в виде двоичных данных (рис. 9.2).

ПРИМЕЧАНИЕ

Примеры кода, представленные в этой главе, написаны на F#, однако те же концепции применимы и на C#. Версии этих примеров кода, переведенные на C#, вы найдете в коде данной книги.

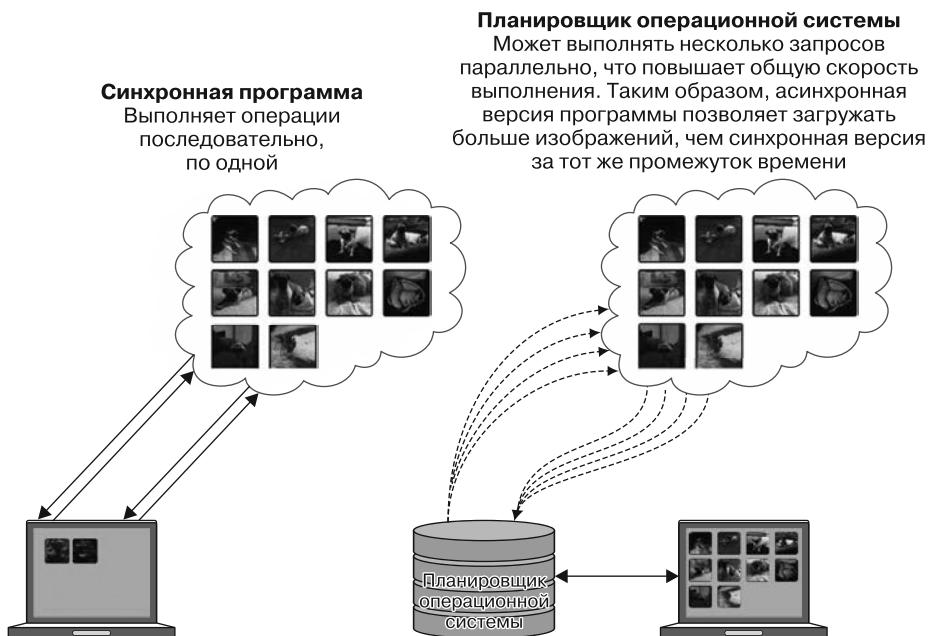


Рис. 9.2. Модель синхронного и асинхронного программирования. Синхронная программа выполняет операции последовательно, по одной. Асинхронная версия может запускать несколько запросов параллельно, повышая общую скорость выполнения программы. В результате асинхронная версия программы позволяет загружать больше изображений за то же время, по сравнению с синхронной версией

Как уже упоминалось, для обеспечения визуальной обратной связи программа запускается как клиентское приложение WPF. С помощью `FileSystemWatcher` (<http://mng.bz/DcRT>) это приложение прослушивает создаваемые файлами события, собирая информацию об изменениях файлов в локальной папке. Когда изображения будут загружены и сохранены в этой локальной папке, `FileSystemWatcher` активирует соответствующее событие и синхронизирует обновления локальной коллекции файлов

с путем изображения, которое затем отображается в контроллере пользовательского интерфейса WPF. (Реализация кода клиентского приложения WPF UI здесь не рассматривается, так как не имеет отношения к основной теме этой главы.)

Сравним синхронную и асинхронную версии программы, показанные на рис. 9.2. Синхронная версия программы выполняет каждый шаг последовательно, перебирая коллекцию изображений и загружая их из хранилища Azure Blob в обычном цикле `for`. Это простая структура, но она не масштабируется. Асинхронная версия программы, наоборот, способна обрабатывать несколько запросов параллельно, что увеличивает количество загружаемых изображений за тот же отрезок времени.

Рассмотрим асинхронную версию программы более подробно. На рис. 9.3 программа начинает работу с отправки запроса в хранилище Azure Blob, чтобы установить соединение с облачным blob-контейнером. После того как соединение будет установлено, извлекается дескриптор медиапотока blob-данных, чтобы начать загрузку изображения. Данныечитываются из потока и, наконец, сохраняются в локальной файловой системе. Затем эта операция повторяется для следующего изображения и т. д., пока все изображения не будут обработаны.



Рис. 9.3. Асинхронная загрузка изображения из сети (из хранилища Azure Blob)

Каждая операция загрузки занимает в среднем 0,89 с (проверено на пяти выполнениях программы), а общее время загрузки ста изображений составляет 89,28 с. Эти значения могут варьироваться в зависимости от пропускной способности сети. Очевидно, что время последовательного выполнения нескольких синхронных операций ввода-вывода равно сумме времен выполнения всех отдельных операций — сравните это с асинхронным подходом, где при параллельном запуске общее время выполнения программы равно времени выполнения самой медленной операции.

ПРИМЕЧАНИЕ

У хранилища Azure Blob есть API для загрузки blob-данных прямо в локальный файл — `DownloadToFile`; но в данном коде мы намеренно создали много операций ввода-вывода, чтобы обратить ваше внимание на проблему синхронного выполнения блокирующих операций ввода-вывода.

В листинге 9.1 показана реализация асинхронного рабочего процесса программы для асинхронной загрузки изображений из хранилища Azure Blob (код, на который следует обратить внимание, выделен жирным шрифтом).

Листинг 9.1. Реализация асинхронного рабочего процесса для загрузки изображений

Анализ и создание соединения с хранилищем Azure

```

let getCloudBlobContainerAsync() : Async<CloudBlobContainer> = async {
    let storageAccount = CloudStorageAccount.Parse(azureConnection)
    let blobClient = storageAccount.CreateCloudBlobClient()
    Создание blob-клиента let container = blobClient.GetContainerReference("media") ← Извлечение ссылки на медиаконтейнер
    let! _ = container.CreateIfNotExistsAsync() ← Асинхронное создание контейнера, если его еще не существует
    return container }

let downloadMediaAsync(blobNameSource:string) (fileNameDestination:string)=
    async {
        let! container = getCloudBlobContainerAsync() ← Преобразование функции в асинхронную
        let blockBlob = container.GetBlockBlobReference(blobNameSource)
        let! (blobStream : Stream) = blockBlob.OpenReadAsync() ←

        use FileStream = new FileStream(fileNameDestination, FileMode.Create,
    FileAccess.Write, FileShare.None, 0x1000, FileOptions.Asynchronous)
        let buffer = Array.zeroCreate<byte> (int blockBlob.Properties.Length)
        let rec copyStream bytesRead = async {
            match bytesRead with
            | 0 -> FileStream.Close(); blobStream.Close()
            | n -> do! FileStream.AsyncWrite(buffer, 0, n) ←
                    let! bytesRead = blobStream.AsyncRead(buffer, 0, buffer.
    Length) ←
                    return! copyStream bytesRead }
        let! bytesRead = blobStream.AsyncRead(buffer, 0, buffer.Length) ←
        do! copyStream bytesRead } ← Добавление семантики вычислительного выражения с оператором !, чтобы зарегистрировать продолжение рабочего процесса
    
```

Обратите внимание: этот код выглядит почти так же, как и последовательный. Единственными изменениями, необходимыми для преобразования кода из синхронного в асинхронный, являются фрагменты, выделенные жирным шрифтом.

Назначение этого кода понятно сразу, его легко интерпретировать благодаря последовательной структуре. Такое упрощение кода является результатом подхода, основанного на шаблонах. Данный подход используется в компиляторе F#

для обнаружения вычислительных выражений, а в случае асинхронного рабочего процесса благодаря этому у разработчика создается иллюзия, что обратные вызовы исчезли. Без обратных вызовов программа не подвержена инверсии управления, как в АРМ, что позволяет писать на F# чистый асинхронный код с акцентом на возможности компоновки.

Обе функции, `getCloudBlobContainerAsync` и `downloadMediaAsync`, обернуты в выражение `async` (объявление рабочего процесса), которое превращает код в блок, способный выполняться асинхронно. Функция `getCloudBlobContainerAsync` создает ссылку на контейнер `media`. Возвращаемым типом этой асинхронной операции для идентификации контейнера является тип `Task<CloudBlobContainer>`, который вместе с `Async<CloudBlobContainer>` обрабатывается базовым выражением асинхронного рабочего процесса (что это такое, вы узнаете далее в этой главе). Ключевой особенностью асинхронного рабочего процесса является сочетание неблокирующих вычислений с облегченной асинхронной семантикой, которая напоминает линейный поток управления. Это упрощает структуру программы по сравнению с традиционным асинхронным программированием на основе обратных вызовов посредством «синтаксического сахара».

Методы, выполняемые асинхронно, связаны с другой конструкцией, которая использует оператор `!` (произносится как «банг» — «хлоп!» или «бабах!»). Этот оператор является основой асинхронного рабочего процесса — именно он уведомляет компилятор F# о том, что следует интерпретировать функцию эксклюзивно. В теле привязки `let!` выражение регистрируется как обратный вызов в контексте будущего выполнения остальной части асинхронного рабочего процесса, а также извлекается основной результат из `Async<'T>`.

В выражении:

```
let! bytesRead = blobStream.AsyncRead(buffer, 0, buffer.Length)
```

возвращаемым типом `blobStream.AsyncRead` является `Async<int>` — количество байтов, считанных асинхронной операцией. Эти байты извлекаются в виде значения `bytesRead`. Функция `rec copyStream` рекурсивно и асинхронно копирует `blobStream` в `fileStream`. Обратите внимание на то, что функция `copyStream` определена внутри другого асинхронного рабочего процесса для захвата (замыкания) значений потока, к которым можно получить доступ и скопировать их. Этот код можно переписать в императивном стиле, сохранив его поведение, следующим образом:

```
let! bytesRead = blobStream.AsyncRead(buffer, 0, buffer.Length)
let mutable bytesRead = bytesRead
while bytesRead > 0 do
    do! fileStream.AsyncWrite(buffer, 0, bytesRead)
    let! bytesReadTemp = blobStream.AsyncRead(buffer, 0, buffer.Length)
    bytesRead <- bytesReadTemp
fileStream.Close(); blobStream.Close()
```

Изменение переменной `bytesRead` инкапсулировано и изолировано внутри основной функции `downloadMediaAsync` и является потокобезопасным.

Кроме `let!`, существуют следующие конструкторы асинхронного рабочего процесса:

- ❑ `use!` — работает как `let!` для одноразовых ресурсов, которые освобождаются, когда выходят за пределы области видимости;
- ❑ `do!` — связывает асинхронный рабочий процесс с типом `Async<unit>`;
- ❑ `return` — возвращает результат выражения;
- ❑ `return!` — использует связанный асинхронный рабочий процесс, возвращая значение выражения.

Асинхронный рабочий процесс F# основан на полиморфном типе данных `Async<'a>`. Этот тип означает произвольное асинхронное вычисление, которое будет материализовано в будущем и которое возвращает значение типа `'a`. Данная концепция похожа на модель TAP в C#. Основное отличие состоит в том, что в F# тип `Async<'a>` не «горячий», то есть требует явную команду для запуска операции.

Когда асинхронный рабочий процесс доходит до примитива `start`, в системе планируется обратный вызов и поток выполнения освобождается. Затем, когда выполнение асинхронной операции завершается, базовые механизмы уведомляют об этом рабочий процесс и результат передается на следующий шаг в потоке кода.

Настоящая магия заключается в том, что асинхронный рабочий процесс будет завершен позднее, но вам не нужно беспокоиться об ожидании результата, поскольку после завершения операции результат будет передан в качестве аргумента в функцию продолжения. Обо всем этом позаботится компилятор, органично преобразовывая вызовы элементов `Bind` в конструкции продолжения. Такой механизм неявно используется для записи в CPS — структурированная программа, основанная на обратных вызовах и расположенная в теле его выражения, позволяет использовать линейный стиль кодирования для последовательности операций.

Вся асинхронная модель выполнения построена на продолжениях, где при выполнении асинхронного выражения сохраняется возможность создания функции, зарегистрированной как обратный вызов (рис. 9.4):

```
bind(fun() -> log "Creating connection...";  
     getCloudBlobContainer(), fun connection ->  
      bind(fun() -> log "Get blob reference...";  
           connection.GetBlobReference(imageReference)), fun  
            blockBlob ->  
  
              async {  
                log "Creating connection...";  
                let! connection = getCloudBlobContainerAsync()  
                log "Get blob reference...";  
                let blockBlob =  
                  connection.GetBlobReference(imageReference)  
                ...
```

```

bind(fun() -> log "Creating connection...";
      getCloudBlobContainer()), fun connection ->
bind(fun() -> log "Get blob reference...";
      connection.GetBlobReference(imageReference)), fun blockBlob ->

    async {
      log "Creating connection...";
      let! connection = getCloudBlobContainerAsync()
      log "Get blob reference...";
      let blockBlob = connection.GetBlobReference(imageReference)
      ...
    }
  
```

Рис. 9.4. Сравнение функции Bind с версией вычислительных выражений

Преимущества использования асинхронного рабочего процесса заключаются в следующем:

- код выглядит последовательным, но ведет себя асинхронно;
- код простой, его легко обсуждать (потому что он похож на последовательный), что упрощает обновление и внесение изменений;
- асинхронная семантика с возможностью компоновки;
- встроенная поддержка отмены;
- простая обработка ошибок;
- удобство распараллеливания.

9.3. Асинхронные вычислительные выражения

Вычислительные выражения — это особенность F#, которая означает полиморфную конструкцию, используемую для настройки спецификации и поведения кода, и приводит нас к стилю компоновочного программирования. В онлайн-документации MSDN дано отличное определение вычислительных выражений:

Вычислительные выражения в F# обеспечивают удобный синтаксис для написания вычислений, которые могут быть выстроены в последовательность и объединены с использованием конструкций и привязок потока управления. Вычислительные выражения могут использоваться для обеспечения удобного синтаксиса монад — свойства функционального программирования, применяемого для управления данными, а также для управления побочными эффектами и их реализации в функциональных программах¹.

Вычислительные выражения являются полезным механизмом для написания вычислений, которые выполняют управляемую последовательность действий в виде шагов ввода. Первый шаг служит источником входных данных для второго

¹ Подробнее см. здесь: <http://mng.bz/n1uZ>.

шага, его выходные данные являются входными для третьего и т. д. по всей цепочке выполнения, если только не возникает исключение, в случае которого выполнение завершается преждевременно, пропуская оставшиеся шаги.

Вычислительные выражения можно рассматривать как расширения языка программирования, поскольку они позволяют настраивать специализированные вычисления, сокращая избыточный код и оставляя тяжеловесные конструкции вне поля зрения разработчика, чтобы уменьшить сложность кода. Вычислительные выражения можно использовать для ввода дополнительного кода на каждом шаге вычислений при выполнении таких операций, как автоматическое журналирование, валидация, управление состоянием и т. п.

Модель асинхронного программирования F# — асинхронный рабочий процесс — основана на вычислительных выражениях, которые также используются для определения других конструкций, таких как последовательности и выражения запроса. Шаблон асинхронного рабочего процесса F# представляет собой «синтаксический сахар», интерпретируемый компилятором как вычислительное выражение. В асинхронном рабочем процессе компилятор должен получить инструкцию интерпретировать выражения рабочего процесса как асинхронные вычисления. Уведомления семантически передаются путем обертывания выражений в асинхронном блоке, который написан с использованием фигурных скобок и идентификатора `async`, поставленного в самом начале блока, например: `async {expression}`.

Если компилятор F# интерпретирует вычисление как асинхронный рабочий процесс, он делит выражение на части, распределяемые между асинхронными вызовами. Такое преобразование, называемое *извлечением «сахара»*, основано на элементарных составных частях, предоставляемых компоновщику вычислений в зависимости от контекста (в данном случае асинхронного рабочего процесса).

F# поддерживает вычислительные выражения посредством специального типа `builder`, связанного с обычным монадическим синтаксисом. Как вы помните, есть два основных монадических оператора для построения вычислений — `Bind` и `Return`.

Для асинхронного рабочего процесса общий монадический тип заменяется и определяется с помощью специализированного типа `Async`:

`async.Bind: Async<'T> → ('T → Async<'R>) → Async<'R>`

► `async.Return: 'T → Async<'T>`
Обертка общего типа 'T
в надтип Async<'T>

Асинхронная операция `Async<'T>`
передается как первый аргумент,
а продолжение `('T → Async <'R>)` —
как второй аргумент

Асинхронный рабочий процесс скрывает нестандартные операции в виде примитивов компоновщика вычислений и восстанавливает оставшуюся часть вычислений в продолжении. Нестандартные операции связаны в теле выражения конструкций компоновщика с помощью оператора `!`. Вовсе не случайно определение вычислительных выражений через операторы `Bind` и `Return` идентично монадическому определению, в котором используются те же монадические операторы. Вычислительное выражение можно представить себе как шаблон продолжения монады.

9.3.1. Различия между вычислительными выражениями и монадами

Вычислительные выражения также можно представить как общий монадический синтаксис для языка F#, который вообще тесно связан с монадами. Основное различие между вычислительными выражениями и монадами заключается в их происхождении. Монады строго соответствуют математической абстракции, тогда как вычислительные выражения в F# – это свойство языка, которое предоставляет разработчику набор инструментов для написания программ с вычислениями и которое может иметь, а может и не иметь монадическую структуру.

F# не поддерживает классы типов, поэтому невозможно написать вычислительное выражение, которое было бы полиморфным по типу вычислений. В F# можно выбрать вычислительное выражение с наиболее специализированным поведением и подходящим синтаксисом (как вскоре будет показано на примере).

Классы типов

Класс типа – это конструкция, которая обеспечивает определенный полиморфизм, достигаемый путем применения неких ограничений к типам переменных. Классы типов сродни интерфейсам – они также определяют поведение, но имеют больше возможностей. Компилятор предоставляет специализированное поведение и синтаксис для типа, определяемого посредством этих ограничений. В .NET можно рассматривать классы типов как интерфейсы, которые определяют поведение, распознаваемое компилятором, а затем предоставляют конкретную реализацию на основе определения этого типа. В конечном счете тип может стать экземпляром класса типа, если этот класс поддерживает такое поведение.

Код, написанный с применением шаблона вычислительных выражений, в итоге преобразуется в выражение, которое использует базовые примитивы, реализованные в контексте компоновщиком вычислений. Эта концепция станет понятнее, когда мы рассмотрим ее на примере.

В листинге 9.2 показана очищенная от «синтаксического сахара» версия функции `downloadMediaAsync`, где компилятор преобразует вычислительное выражение в цепочку вызовов методов. Этот развернутый код показывает, как поведение каждой отдельной асинхронной части инкапсулируется в соответствующем примитиве, который является элементом сборщика вычислений. Ключевое слово `async` сообщает компилятору F# о том, что нужно создать экземпляр `AsyncBuilder`, который реализует важные элементы асинхронного рабочего процесса – `Bind`, `Return`, `Using`, `Combine` и т. д. В листинге 9.2 показано, как компилятор преобразует вычислительное выражение в цепочку вызовов методов из кода, представленного в листинге 9.1. (Код, на который следует обратить внимание, выделен жирным шрифтом.)

Листинг 9.2. Вычислительное выражение DownloadGediaAsyncDesugared

```

Оператор Bind — это версия оператора let!
без «синтаксического сахара»
let downloadMediaAsync(blobName:string) (fileNameDestination:string) =
    async.Delay(fun() ->
        ▶ async.Bind(getCloudBlobContainerAsync(), fun container ->
            let blockBlob = container.GetBlockBlobReference(blobName)
            async.Using(blockBlob.OpenReadAsync(), fun (blobStream:Stream) -> ←
                Оператор Using — это аналогичное
                преобразование оператора use!
                let sizeBlob = int blockBlob.Properties.Length
                async.Bind(blobStream.AsyncRead(sizeBlob), fun bytes ->
                    use fileStream = new FileStream(fileNameDestination,
                        FileMode.Create, FileAccess.Write, FileShare.None, bufferSize,
                        ▶ FileMode.Create, FileAccess.Write, FileShare.None, bufferSize,
                        ▶ FileMode.Create, FileAccess.Write, FileShare.None, bufferSize)
                    async.Bind(fileStream.AsyncWrite(bytes, 0, bytes.Length), fun () ->
                        fileStream.Close()
                        blobStream.Close()
                        async.Return())))) ←
                Возвращение оператора,
                который завершает
                вычислительное выражение
)

```

В этом коде компилятор преобразует связывающую конструкцию `let!` в вызов операции `Bind`, которая извлекает значение из вычислительного типа и выполняет оставшуюся часть вычислений, преобразованных в продолжение. Операция `Using` выполняет вычисления, где тип результирующего значения представляет собой ресурс, который может быть удален. Первый элемент цепочки — `Delay` — обертывает выражение так, чтобы его выполнением можно было управлять целиком; впоследствии это выражение может запускаться по требованию.

На каждом шаге вычислений выполняется один и тот же шаблон: элемент компоновщика вычислений, такой как `Bind` или `Using`, запускает операцию и обеспечивает продолжение, которое выполняется при завершении операции, так что вам не придется ожидать результата.

9.3.2. AsyncRetry: построение собственных вычислительных выражений

Как уже отмечалось, вычислительные выражения представляют собой интерпретацию на основе шаблонов (например, LINQ/PLINQ). Это означает, что компилятор может по реализации элементов `Bind` и `Return` сделать вывод, что данная конструкция типа является монадическим выражением. Следуя нескольким простым спецификациям, вы можете построить собственное вычислительное выражение или даже расширить существующее, чтобы добавить в выражение особую коннотацию и желаемое поведение.

Вычислительные выражения могут содержать большое количество стандартных языковых конструкций (табл. 9.1), но большинство этих определений элементов являются необязательными и могут использоваться в соответствии с потребностями конкретной реализации. Обязательными и базовыми элементами для представления компилятору валидного вычислительного выражения являются `Bind` и `Return`.

Таблица 9.1. Операторы вычислительных выражений

Bind : $M<'a> * ('a \rightarrow M<'b>) \rightarrow M<'b>$	Операции <code>let!</code> и <code>do!</code> , преобразованные для вычислительного выражения
Return : $'a \rightarrow M<'a>$	Операция <code>return</code> , преобразованная для вычислительного выражения
Delay : $(unit \rightarrow M<'a>) \rightarrow M<'a>$	Используется для гарантии того, что побочные эффекты в вычислительном выражении будут выполнены тогда, когда это ожидается
Yield : $'a \rightarrow M<'a>$	Операция <code>yield</code> , преобразованная для вычислительного выражения
For : $seq<'a> * ('a \rightarrow M<'b>) \rightarrow M<'b>$	Цикл <code>for ... do ...</code> , преобразованный для вычислительного выражения. $M<'b>$ можно заменить на $M<unit>$
While : $(unit \rightarrow bool) * M<'a> \rightarrow M<'a>$	Блок <code>while-do</code> , преобразованный для вычислительного выражения. $M<'b>$ можно заменить на $M<unit>$
Using : $'a * ('a \rightarrow M<'b>) \rightarrow M<'b>$ when $'a : IDisposable$	Привязки <code>use</code> , преобразованные для вычислительного выражения
Combine : $M<'a> \rightarrow M<'a> \rightarrow M<'a>$	Построение последовательности, преобразованное для вычислительного выражения. Первый $M<'a>$ можно заменить на $M<unit>$
Zero : $unit \rightarrow M<'a>$	Пустые ветви <code>else</code> из <code>if/then</code> , преобразованные для вычислительного выражения
TryWith : $M<'a> \rightarrow M<'a> \rightarrow M<'a>$	Пустые привязки <code>try/with</code> , преобразованные для вычислительного выражения
TryFinally : $M<'a> \rightarrow M<'a> \rightarrow M<'a>$	Привязки <code>try/finally</code> , преобразованные для вычислительного выражения

Построим вычислительное выражение, которое можно использовать в примере, показанном в листинге 9.2. Прежде всего функция `downloadMediaCompAsync` выполняет асинхронное подключение к сервису Azure Blob. Но что случится, если соединение будет разорвано? Будет выдано сообщение об ошибке, и вычисление прекратится. Мы можем проверить, находится ли клиент в сети, и потом попытаться подключиться еще раз; но общим правилом при выполнении сетевых операций является повторение попытки несколько раз, прежде чем прерывать операцию.

В листинге 9.3 мы построим вычислительное выражение, которое успешно выполняет асинхронную операцию несколько раз, с задержкой в несколько миллисекунд между попытками, прежде чем прекратить операцию (код, на который следует обратить внимание, выделен жирным шрифтом).

Листинг 9.3. Вычислительное выражение AsyncRetryBuilder

```

Операция выполнила максимально разрешенное
число повторов, и было выдано сообщение
об ошибке, которое остановило данное вычисление

type AsyncRetryBuilder(max, sleepMilliseconds : int) =
    let rec retry n (task:Async<'a>) (continuation:'a -> Async<'b>) =
        async {
            try
                let! result = task
                let! conResult = continuation result
                return conResult
            with error -
                if n = 0 then return raise error
                else
                    do! Async.Sleep sleepMilliseconds
                    return! retry (n - 1) task continuation }

```

Запуск рабочего процесса задачи в блоке try-catch

Возвращение самого вычисления

Получение значения внутри блока async

member x.ReturnFrom(f) = f

member x.Return(v) = async { return v }

member x.Delay(f) = async { return! f() }

member x.Bind(task:Async<'a>, continuation:'a -> Async<'b>) =

retry max task continuation

member x.Bind(t : Task, f : unit -> Async<'R>) : Async<'R> =

async.Bind(Async.AwaitTask t, f)

Привязка асинхронной функции и ее продолжения, запуск функции повтора. Если функция завершается успешно, то результат передается в функцию продолжения

Обертывание функции внутри блока азупс, что позволяет создавать вложенные вычисления внутри асинхронного рабочего процесса

Демонстрация совместимости операций на основе задач

AsyncRetryBuilder — это сборщик вычислений, применяемый для определения значения, необходимого для построения вычисления. В листинге 9.4 показано, как использовать сборщик вычислений (код, на который следует обратить внимание, выделен жирным шрифтом).

Листинг 9.4. Использование AsyncRetryBuilder для определения значения конструкции

Определение значения, идентифицирующего вычислительное выражение.

В случае исключения это значение позволит повторить попытку выполнения кода три раза с задержкой 250 мс между попытками

```

let retry = AsyncRetryBuilder(3, 250)

```

```

let downloadMediaCompAsync(blobNameSource:string)
    (fileNameDestination:string) =
async {
    let! container = retry {
        return! GetCloudBlobContainerAsync() }
    ... Остальная часть кода без изменений

```

Повторное выполнение вычислительного выражения может быть вложенным, располагаясь внутри асинхронного рабочего процесса

В случае исключения экземпляра `retry` вычислительного выражения `AsyncRetryBuilder` трижды повторяет попытку выполнить код с задержкой 250 мс между попытками. Теперь вычислительное выражение `AsyncRetryBuilder` можно использовать в сочетании с асинхронным рабочим процессом, запуская и асинхронно повторяя (в случае отказа) операцию `downloadMediaCompAsync`. Обычно создается глобальный идентификатор значения вычислительного выражения, который может быть многоократно использован в разных частях программы. Например, асинхронный рабочий процесс и выражение последовательности можно получить в любом месте кода, не создавая новое значение.

9.3.3. Расширение асинхронного рабочего процесса

Компилятор F# позволяет не только создавать специальные вычислительные выражения, но и расширять уже существующие. Асинхронный рабочий процесс — прекрасный пример вычислительного выражения, которое может быть усовершенствовано. В листинге 9.4 соединение с контейнером Azure Blob устанавливается посредством выполнения асинхронной операции `getCloudBlobContainerAsync`, реализация которой показана ниже:

```
let getCloudBlobContainerAsync() : Async<CloudBlobContainer> = async {
    let storageAccount = CloudStorageAccount.Parse(azureConnection)
    let blobClient = storageAccount.CreateCloudBlobClient()
    let container = blobClient.GetContainerReference("media")
    let! _ = container.CreateIfNotExistsAsync()
    return container }
```

В теле функции `getCloudBlobContainerAsync` операция `CreateIfNotExistsAsync` возвращает тип `Task`, который неудобен для использования в контексте асинхронного рабочего процесса. К счастью, в асинхронный инструментарий F# входит оператор `Async.AwaitTask`¹, который позволяет ожидать операцию `Task` и обрабатывать ее как асинхронное вычисление F#. У многих асинхронных операций .NET есть возвращаемые типы `Task`, производные от параметрического типа `Task<'T>`. Эти операции, предназначенные главным образом для работы с C#, изначально несовместимы с асинхронными вычислениями F#.

Как выйти из этого положения? Расширить вычислительное выражение. В листинге 9.5 показан расширенный вариант модели асинхронного рабочего процесса F#, который можно использовать не только для асинхронных операций, но и для типов `Task` и `Observable`. Асинхронные вычислительные выражения требуют конструкций типов, которые могут создавать наблюдаемые объекты и задачи, в отличие от чисто асинхронных рабочих процессов. Эта модель позволяет ожидать любых событий, создаваемых потоками и задачами `Event` и `IEnumerable` из операций `Task`.

¹ `Async.AwaitTask` создает вычисления, которые ожидают определенную задачу и возвращают ее результат.

Такие расширения вычислительных выражений, как видим, абстрагируют использование оператора `Async.AwaitTask` (соответствующие команды выделены жирным шрифтом).

Листинг 9.5. Расширение асинхронного рабочего процесса для поддержки `Task<'a>`

```
type Microsoft.FSharp.Control.AsyncBuilder with
    member x.Bind(t:Task<'T>, f:'T -> Async<'R>) : Async<'R> =
        async.Bind(Async.AwaitTask t, f) ←

    member x.Bind(t:Task, f:unit -> Async<'R>) : Async<'R> =
        async.Bind(Async.AwaitTask t, f) ←

    member x.Bind (m:'a IObservable, f:'a -> 'b Async) =
        async.Bind(Async.AwaitObservable m, f) ←

    member x.ReturnFrom(computation:Task<'T>) =
        x.ReturnFrom(Async.AwaitTask computation)
```

Расширение
оператора `AsyncBuilder`,
чтобы его можно было
выполнять для других
надтипов

`AsyncBuilder` позволяет вводить функции для расширенных манипуляций с другими типами оберток, такими как `Task` и `Observable`, тогда как функция `Bind` в расширении дает возможность получить внутреннее значение, содержащееся в `Observable` (или `IEvent`), используя операторы `let!` и `do!`. Этот метод позволяет избавиться от дополнительных функций, таких как `Async.AwaitEvent` и `Async.AwaitTask`.

В первой строке кода компилятор получает уведомление о том, что нужно обратиться к оператору `AsyncBuilder`, который управляет преобразованием асинхронных вычислительных выражений. После этого расширения компилятор может определить, какую операцию `Bind` следует использовать, в зависимости от сигнатуры выражения, зарегистрированной через связывание `let!`. Теперь мы можем использовать асинхронные операции типа `Task` и `Observable` в асинхронном рабочем процессе.

9.3.4. Отображение асинхронных операций: функтор `Async.map`

Продолжим расширять возможности асинхронного рабочего процесса F#. Асинхронный рабочий процесс F# предоставляет широкий набор операторов; но пока еще нет встроенной поддержки функции `Async.map` (также известной как *функтор*) с сигнатурой типа:

`('a → 'b) → Async<'a> → Async<'b>`

Функтор — это шаблон отображения согласно структуре, которое обеспечивается за счет поддержки реализации функции с двумя аргументами, называемой `map` (более известной как `fmap`). Например, оператор `Select` в LINQ/PLINQ является функтором для надтипа `IEnumerable`. Как правило, функторы используются в C# для реализации гибких API в LINQ-стиле, которые также применяются для типов (или контекстов), отличных от коллекций.

В главе 7, где был описан тип функтора, вы узнали, как реализовать функтор (выделен жирным шрифтом) для надтипа Task:

```
Task<T> fmap<T, R>(this Task<T> input, Func<T, R> map) =>
    input.ContinueWith(t => f(t.Result));
```

Эта функция имеет сигнатуру ($T \rightarrow R \rightarrow Task<T> \rightarrow Task<R>$, поэтому она принимает функцию отображения $T \rightarrow R$ в качестве первого аргумента (другими словами, она переходит от значения типа T к значению типа R , в нотации C# — $Func<T, R>$), а затем обновляет тип $Task<T>$, принимаемый в качестве второго аргумента, и возвращает $Task<R>$. Применяя этот шаблон к асинхронному рабочему процессу F#, получим следующую сигнатуру функции `Async.map`:

```
('a -> 'b) -> Async<'a> -> Async<'b>
```

Здесь первый аргумент — это функция `'a -> 'b`, второй — `Async<'a>`, а результат — `Async<'b>`. Ниже представлена реализация `Async.map`:

```
module Async =
    let inline map (func:'a -> 'b) (operation:Async<'a>) = async {
        let! result = operation
        return func result }
```

`let! result = operation` выполняет асинхронную операцию и развертывает тип `Async<'a>`, возвращая тип `a`. Затем мы можем передать значение `'a` в функцию `func:'a -> 'b`, которая преобразует `'a` в `'b`. В итоге, когда вычислено значение `'b`, оператор `return` обертыывает результат `'b` в тип `Async<>`.

Ключевое слово `inline`

Ключевое слово `inline` в F# используется для определения функции, встроенной в вызывающий код путем изменения тела функции непосредственно в коде вызывающего объекта. Наиболее ценным применением ключевого слова `inline` является вложение функций более высокого порядка в точке вызова, где их аргументы также являются встроенными, для создания единой, полностью оптимизированной части кода. F# также может встраиваться между скомпилированными сборками, потому что `inline` передается через метаданные .NET.

Функция `map` применяет операцию к объектам, находящимся внутри контейнера `Async`¹, и возвращает контейнер той же формы. Функция `Async.map` интерпретируется как функция с двумя аргументами, где значение, к которому применяется функция, обернуто в контекст F# `Async`. Тип F# `Async` добавляется как на входе, так и на выходе.

Основной целью функции `Async.map` является управление (проектирование) результата вычисления `Async` без выхода из контекста. Вернемся к примеру с храни-

¹ Полиморфные типы можно рассматривать как контейнеры для значений другого типа.

лицем Azure Blob. Здесь мы можем использовать функцию `Async.map` для загрузки и преобразования изображения следующим образом (код, на который следует обратить внимание, выделен жирным шрифтом):

```
let downloadBitmapAsync(blobNameSource:string) = async {
    let! token = Async.CancellationToken
    let! container = getCloudBlobContainerAsync()
    let blockBlob = container.GetBlockBlobReference(blobNameSource)
    use! (blobStream : Stream) = blockBlob.OpenReadAsync()
    return Bitmap.FromStream(blobStream) }

let transformImage (blobNameSource:string) =
    downloadBitmapAsync(blobNameSource)
    |> Async.map ImageHelpers.setGrayscale
    |> Async.map ImageHelpers.createThumbnail
```

Функция `Async.map` содержит асинхронные операции загрузки изображения `blobNameSource` из хранилища Azure Table с функциями преобразования `setGrayscale` и `createThumbnail`.

ПРИМЕЧАНИЕ

Функции `ImageHelpers` были определены в главе 7, поэтому здесь их определения умышленно опущены. Полную реализацию вы найдете в исходном коде к книге.

В этом фрагменте использованы такие преимущества функции `Async.map`, как возможность компоновки и дальнейшая инкапсуляция.

9.3.5. Распараллеливание асинхронных рабочих процессов: `Async.Parallel`

Вернемся к примеру загрузки 100 изображений из хранилища Azure Blob с использованием асинхронного рабочего процесса F#. В разделе 9.2 мы создали функцию `downloadMediaAsync`, которая загружает из облака одно blob-изображение, используя асинхронный рабочий процесс. Пора связать куски в единое целое и выполнить код. Но вместо перебора списка изображений и выполнения операций по очереди асинхронный рабочий процесс F# предоставляет изящную альтернативу: `Async.Parallel`.

Идея состоит в том, чтобы скомпоновать все асинхронные вычисления и выполнить их все сразу. Параллельная компоновка асинхронных вычислений является эффективной благодаря масштабируемости пула потоков .NET и контролируемому выполнению переопределяемых операций, таких как веб-запросы в современных операционных системах.

Используя функцию F# `Async.Parallel`, можно параллельно загрузить сотни изображений (код, на который следует обратить внимание, в листинге 9.6 выделен жирным шрифтом).

Листинг 9.6. Функция Async.Parallel позволяет загрузить все изображения параллельно

```

let retry = RetryAsyncBuilder(3, 250) ← Определение вычислительного выражения Retry

let downloadMediaCompAsync (container:CloudBlobContainer)
    (blobMedia:IListBlobItem) = retry { ← Выполнение асинхронных
        operations with retry
    Возвращение
    изображения
    из операции let blobName = blobMedia.Uri.Segments.[blobMedia.Uri.Segments.Length-1]
    let blockBlob = container.GetBlockBlobReference(blobName)
    let! (blobStream : Stream) = blockBlob.OpenReadAsync()
    return Bitmap.FromStream(blobStream)
}

let transformAndSaveImage (container:CloudBlobContainer)
    (blobMedia:IListBlobItem) =
    downloadMediaCompAsync container blobMedia
    |> Async.map ImageHelpers.setGrayscale
    |> Async.map ImageHelpers.createThumbnail
    |> Async.tap (fun image -> ← Функция tap применяет побочные эффекты к данным,
        let mediaName = ← поступающим на ее вход; результат игнорируется
        blobMedia.Uri.Segments.[blobMedia.Uri.Segments.Length - 1]
        image.Save(mediaName))

let downloadMediaCompAsyncParallel() = retry { ← Выполнение асинхронных операций
    with retry
    Получение
    списка
    загружаемых
    изображений let! container = getCloudBlobContainerAsync()
    let computations =
        container.ListBlobs()
        |> Seq.map(transformAndSaveImage container) ← Бесконфликтное разделение аргумента
                                                    CloudBlobContainer между параллельными
                                                    неблокирующими вычислениями,
                                                    поскольку он доступен только для чтения
    return! Async.Parallel computations } ← Создание последовательности
                                            неблокирующих вычислений загрузки,
                                            которые не выполняются сразу,
                                            поскольку требуют явных запросов

let cancelOperation() =
    downloadMediaCompAsyncParallel()
    |> Async.StartCancelable ← Агрегирование последовательности
                                асинхронных вычислений в единый
                                асинхронный рабочий процесс, который
                                выполняет все операции параллельно

```

Функция StartCancelable выполняет асинхронное вычисление с явными запросами без блокировки текущего потока и предоставляет маркер, который может использоваться для остановки вычислений

Функция Async.Parallel принимает произвольный набор асинхронных операций и возвращает единый асинхронный рабочий процесс, который будет запускать все вычисления параллельно и ожидать, когда все они будут завершены. Функция Async.Parallel координирует работу с планировщиком пула потоков, чтобы максимально задействовать ресурсы. Для этого используется шаблон Fork/Join, что приводит к резкому росту производительности.

Библиотечная функция Async.Parallel принимает список асинхронных вычислений и создает одно асинхронное вычисление, которое запускает отдельные вычисления параллельно и ожидает их завершения, чтобы обработать весь набор как единое целое. Когда все операции будут завершены, функция возвратит результаты,

агgregированные в одном массиве. После этого можно будет перебрать этот массив и получить результаты для дальнейшей обработки.

Обратите внимание на минимальное изменение кода и синтаксис, необходимый для преобразования вычислений, выполняющих операции по одной, в код, где они выполняются параллельно. Кроме того, это преобразование не требует согласования синхронизации и блокировки памяти.

Оператор `Async.tap` асинхронно применяет функцию к значению, переданному на его вход, игнорирует результат и возвращает исходное значение. Оператор `Tap` был впервые описан в листинге 8.3. Здесь представлена его версия с использованием асинхронного рабочего процесса F# (выделена жирным шрифтом):

```
let inline tap (fn:'a -> 'b) (x:Async<'a>) =
    (Async.map fn x) |> Async.Ignore |> Async.Start; x
```

Эту и другие полезные функции `Async` вы найдете в прилагаемом к данной книге исходном коде, в библиотеке `FunctionalConcurrencyLib`.

Время выполнения параллельной загрузки изображений с использованием асинхронного рабочего процесса F# в сочетании с `Async.Parallel` составляет 10,958 с. Этот результат примерно на 5 с быстрее, чем АРМ, что приблизительно в восемь раз быстрее, чем исходная синхронная реализация. Основные выгоды включают в себя структуру кода, удобство чтения и поддержки, а также возможность компоновки.

Используя асинхронный рабочий процесс, мы получили простую асинхронную семантику для выполнения неблокирующего вычисления, которое обеспечивает ясный, понятный код, удобный для поддержки и обновления. Кроме того, благодаря функции `Async.Parallel` можно легко запускать параллельно множество асинхронных вычислений с минимальными изменениями кода, что резко повышает производительность.

Тип `Async` не является горячим

Отличительным функциональным аспектом асинхронного рабочего процесса является время его выполнения. В F#, когда вызывается асинхронная функция, возвращаемый тип `Async<'a>` представляет собой вычисление, которое будет реализовано только после явного запроса. Это свойство позволяет моделировать и компоновать несколько асинхронных функций, которые могут выполняться условно, по требованию. Такое поведение в корне отличается от поведения асинхронных операций C# TAP (`async/await`), которые запускают выполнение немедленно.

В итоге реализация расширения типа `Async.StartCancelable` запускает асинхронный рабочий процесс без блокировки вызывающего потока, используя новый `CancellationToken`, и возвращает `IDisposable`, который отменяет рабочий процесс при его удалении. Мы не использовали функцию `Async.Start`, поскольку она

не предоставляет семантику продолжений, что во многих случаях полезно, так как позволяет применить операцию к результату вычислений. В данном примере при завершении вычислений выводится сообщение; но тип результата доступен для дальнейшей обработки.

Ниже представлена реализация оператора `Async.StartCancelable`, более сложного по сравнению с `Async.Start` (выделен жирным шрифтом):

```
type Microsoft.FSharp.Control.Async with
    static member StartCancelable(op:Async<'a>) (tap:'a -> unit)(?onCancel)=
        let ct = new System.Threading.CancellationTokenSource()
        let onCancel = defaultArg onCancel ignore
        Async.StartWithContinuations(op, tap, ignore, onCancel, ct.Token)
    { new IDisposable with
        member x.Dispose() = ct.Cancel() }
```

В базовой реализации функции `Async.StartCancelable` используется оператор `Async.StartWithContinuations`, который обеспечивает встроенную поддержку отмены. При передаче асинхронной операции `op:Async<'a>` (после завершения первого аргумента) результат передается как продолжение во второй аргумент функции `tap: 'a -> unit`. Необязательный параметр `onCancel` представляет собой запускаемую функцию; в данном случае это отмена основной операции `op:Async<'a>`. Результат `Async.StartCancelable` — это анонимный объект, создаваемый динамически на основе интерфейса `IDisposable`, который отменит операцию в случае вызова метода `Dispose`.

Асинхронный API в F#

В состав модуля `Async` F# входит ряд функций, позволяющих создавать или использовать в программе асинхронные рабочие процессы. Данные функции применяются для запуска других функций, предоставляя различные способы создания асинхронного рабочего процесса. Это может быть как фоновый поток, так и объект `Task` из .NET Framework, или же запуск вычисления в текущем потоке.

Пожалуй, стоит рассмотреть подробнее использованные нами операторы `Async` F# `Async.StartWithContinuations`, `Async.Ignore` и `Async.Start`.

Async.StartWithContinuations

Функция `Async.StartWithContinuations` выполняет асинхронный рабочий процесс, начинающийся сразу же в текущем потоке операционной системы, и после его завершения передает соответственно результат, исключение и отмену (`OperationCancelledException`) для одной из заданных функций. Если поток, который инициирует выполнение, имеет свой ассоциированный с ним `SynchronizationContext`, то продолжения, запускаемые по окончании функции, будут использо-

зователь этот `SynchronizationContext` для передачи результатов. Эта функция удобна для обновления графических пользовательских интерфейсов. Она принимает в качестве аргументов три функции, которые вызываются, если асинхронное вычисление завершается успешно, вызывает исключение или отменяется соответственно.

Эта функция имеет следующую сигнатуру: `Async<'T> -> ('T -> unit) * (exn -> unit) * (OperationCanceledException -> unit) -> unit`. `Async.StartWithContinuations` не поддерживает возвращаемое значение, поскольку результат вычисления обрабатывается внутри, функцией, предназначенней для успешного завершения работы (листинг 9.7).

Листинг 9.7. `Async.StartWithContinuations`

```
let computation() = async {
    use client = new WebClient()
    let! manningSite =
        client.AsyncDownloadString(Uri("http://www.manning.com"))
    return manningSite
}

Async.StartWithContinuations(computation(), ←
    (fun site-> printfn "Size %d" site.Length), ←
    (fun exn->printfn"exception-%s" <| exn.ToString()), ←
    (fun exn->printfn"cancel-%s" <| exn.ToString()) ←
    )
```

Асинхронное вычисление возвращает длинную строку

Асинхронное вычисление начинается немедленно, используя текущий поток операционной системы

Вычисление завершается успешно и вызывается продолжение, которое выводит размер загруженного сайта

Операция генерирует исключение; выполняется продолжение исключения, выводящее информацию об исключении

Операция отменяется; вызывается продолжение отмены, которое выводит информацию об отмене

Async.Ignore

Оператор `Async.Ignore` принимает вычисление и возвращает рабочий процесс, который выполняет вычисление исходного выражения, игнорирует его результат и возвращает тип `unit`. Его сигнатура: `Async.Ignore: Async<'T> -> Async<unit>`.

Возможны два варианта использования `Async.Ignore`:

```
Async.Start(Async.Ignore computationWithResult())
```

```
let asyncIgnore = Async.Ignore >> Async.Start
```

Во втором случае создается функция `asyncIgnore`, в которой посредством компоновки функций объединяются операторы `Async.Ignore` и `Async.Start`. В листинге 9.8 показан полный пример, в котором результат асинхронной операции игнорируется с использованием функции `asyncIgnore` (выделена жирным шрифтом).

Листинг 9.8. Функция Async.Ignore

```
let computation() = async {
    use client = new WebClient()
    let! manningSite =
        client.AsyncDownloadString(Uri("http://www.manning.com"))
    printfn "Size %d" manningSite.Length
    return manningSite
}
```

← | Это асинхронное вычисление возвращает длинную строку

```
Async.Ignore (computation())
```

← | Вычисление выполняется асинхронно, результат отбрасывается (игнорируется)

Если нужно вычислить результат асинхронных операций без блокировки, в чистом CPS-стиле, то оператор `Async.StartWithContinuations` — лучший вариант.

Async.Start

Функция `Async.Start`, показанная в листинге 9.9, не поддерживает возвращаемое значение; в сущности, она выполняет асинхронное вычисление типа `Async<unit>`. Оператор `Async.Start` выполняет вычисления асинхронно, поэтому процесс вычисления должен сам определить способ связи и возврата конечного результата. Эта функция ставит асинхронный рабочий процесс в очередь для выполнения в пуле потоков и сразу, не дожидаясь завершения, возвращает управление вызывающему объекту. Поэтому операция может быть завершена в другом потоке.

Сигнатура этой функции — `Async.Start: Async<unit> -> unit`. В качестве необязательного аргумента она принимает `cancelationToken` (листинг 9.9).

Листинг 9.9. Функция Async.Start

```
let computationUnit() = async {
    do! Async.Sleep 1000
    use client = new WebClient()
    let! manningSite =
        client.AsyncDownloadString(Uri("http://www.manning.com"))
    printfn "Size %d" manningSite.Length
}
```

← | Создание асинхронного вычисления для загрузки сайта с односекундной задержкой с целью имитации интенсивных вычислений

← | Вывод размера сайта из тела выражения

```
Async.Start(computationUnit())
```

← | Выполнение вычисления без блокировки вызывающего потока

Поскольку `Async.Start` не поддерживает возвращаемое значение, размер сайта выводится внутри выражения, где это значение доступно. А как быть, если вычисление все же возвращает значение, но вы не можете изменить асинхронный рабочий процесс? Тогда можно извлечь результат асинхронного вычисления, выполнив функцию `Async.Ignore` перед началом операции.

9.3.6. Поддержка отмены асинхронного рабочего процесса

При выполнении асинхронной операции полезно при необходимости иметь возможность прекратить выполнение досрочно, прежде чем оно будет завершено. Это хорошо работает для длительных неблокирующих операций, отмена которых позволяет избежать задач, которые иначе зависли бы. Например, можно отменить операцию загрузки 100 изображений из хранилища Azure Blob, если время загрузки превысит определенное значение. Асинхронный рабочий процесс F# поддерживает отмену изначально как автоматический механизм; когда рабочий процесс отменяется, он также отменяет все дочерние вычисления.

Как правило, желательно согласовать маркеры отмены и сохранить контроль над ними. В этих случаях можно предоставить собственные маркеры, но во многих других ситуациях можно добиться аналогичных результатов с меньшим количеством кода, используя встроенный маркер по умолчанию из асинхронного модуля F#. В начале асинхронной операции эта базовая система передает предоставленный ей маркер `CancellationToken` или, если таковой не предоставлен, присваивает рабочему процессу произвольный маркер и отслеживает, не поступит ли запрос отмены. Сборщик вычислений `AsyncBuilder` проверяет статус маркера отмены во время выполнения каждой операции связывания (`let!`, `do!`, `return!`, `use!`). Если маркер помечен как маркер отмены, то рабочий процесс прерывается.

Этот сложный механизм облегчает вашу работу, если не нужно выполнять ничего особенного для поддержки отмены. Кроме того, асинхронный рабочий процесс F# поддерживает неявное генерирование и распространение маркеров отмены в процессе его выполнения, и во время асинхронных вычислений все вложенные асинхронные операции автоматически включаются в иерархию отмены.

F# поддерживает отмену в разных формах. Первая из них — посредством функции `Async.StartWithContinuations`, которая отслеживает маркер по умолчанию и отменяет рабочий процесс, если этот маркер помечен как отмененный. При активации маркера отмены вместо функции, вызываемой в случае успешного завершения, вызывается функция, ответственная за обработку маркера отмены. Другие варианты включают в себя передачу маркера отмены вручную или расчет на стандартный `Async.Default CancellationToken`, который активирует `Async.CancellationToken` (в листинге 9.10 выделен жирным шрифтом).

В листинге 9.10 показано, как реализовать поддержку отмены в примере с загрузкой изображений с помощью `Async.Parallel` (см. листинг 9.6). В этом примере маркер отмены передается вручную, поскольку в автоматической версии с использованием `Async.Default CancellationToken` отсутствует изменение кода — есть только функция отмены последней асинхронной операции.

Мы создали экземпляр объекта `CancellationTokenSource`, который передает маркер отмены в асинхронное вычисление, начиная операцию с помощью функции `Async.Start` и передавая `CancellationToken` в качестве второго аргумента. Затем мы отменяем операцию, которая прерывает все вложенные операции.

Листинг 9.10. Отмена асинхронного вычисления

```

let tokenSource = new CancellationTokenSource() ←
    Экземпляр
    CancellationTokenSource
    используется
    для создания
    CancellationToken

let container = getCloudBlobContainer()
let parallelComp() =
    container.ListBlobs()
    |> Seq.map(fun blob -> downloadMediaCompAsync container blob)
    |> Async.Parallel

Async.Start(parallelComp() |> Async.Ignore, tokenSource.Token) ←
    Генерация маркера отмены и передача его для асинхронного
    вычисления, чтобы остановить выполнение по требованию

tokenSource.Cancel() ←

```

В листинге 9.11 `Async.TryCancelled` добавляет функцию к асинхронному рабочему процессу. Данная функция будет вызываться, когда будет активирован маркер отмены. Это альтернативный способ создания дополнительного кода, который будет выполняться в случае отмены. В следующем листинге показано, как использовать функцию `Async.TryCancelled`, которая имеет дополнительное преимущество — возвращает значение, обеспечивая возможность компоновки. (Код, на который стоит обратить внимание, выделен жирным шрифтом.)

Листинг 9.11. Отмена асинхронного вычисления с уведомлением

```

let onCancelled = fun (cnl:OperationCanceledException) ->
    printfn "Operation cancelled!" ←
        Функция, вызываемая для обработки
        исключения OperationCanceledException
        в случае отмены операции

let tokenSource = new CancellationTokenSource()

let tryCancel = Async.TryCancelled(parallelComp(), onCancelled) ←
    Функция parallelComp обернута в оператор Async.TryCancelled
    для обработки случаев нестандартного поведения,
    возникающих при отмене операции

Async.Start(tryCancel, tokenSource.Token)

```

`TryCancelled` — это асинхронный рабочий процесс, который можно использовать в сочетании с другими вычислениями. Его выполнение начинается по требованию при наличии явного запроса посредством функции запуска, такой как `Async.Start` или `Async.RunSynchronously`.

Async.RunSynchronously. Функция `Async.RunSynchronously` блокирует текущий поток во время выполнения рабочего процесса и продолжает выполнять текущий поток, когда рабочий процесс завершается. Этот подход идеально подходит для использования в интерактивном сеансе F# в целях тестирования и в консольных приложениях, поскольку подразумевает ожидание завершения асинхронного вычисления. Однако не рекомендуется использовать этот способ асинхронных вычислений в программах с GUI, поскольку он блокирует пользовательский интерфейс.

Сигнатура этой функции — `Async<'T> -> 'T`. В качестве необязательных аргументов она принимает значение задержки и `cancellationToken`. В листинге 9.12

показан простейший способ выполнения асинхронного рабочего процесса (выделен жирным шрифтом).

Листинг 9.12. Async.RunSynchronously

```

let computation() = async {
    do! Async.Sleep 1000
    use client = new WebClient()
    return! client.AsyncDownloadString(Uri("www.manning.com"))
}

let manningSite = Async.RunSynchronously(computation()) ← Выполнение вычисления
printfn "Size %d" manningSite.Length ← Вывод размера
загруженного сайта

```

9.3.7. Управление параллельными асинхронными операциями

Модель программирования `Async.Parallel` — отличное средство для реализации параллелизма ввода-вывода на основе шаблона Fork/Join. Fork/Join позволяет выполнить набор вычислений так, что в определенных точках кода выполнение операций распараллеливается на несколько ветвей, которые затем объединяются в последующей точке, и выполнение возобновляется.

Но, поскольку `Async.Parallel` опирается на пул потоков, гарантируется максимальная степень параллелизма и, следовательно, повышается производительность. Кроме того, существуют случаи, когда запуск большого количества асинхронных рабочих процессов может отрицательно сказаться на производительности. В частности, асинхронный рабочий процесс выполняется полуупреждающим способом, когда после запуска большого количества операций (более 10 000 на компьютере с 4 Гбайт оперативной памяти) асинхронные рабочие процессы помещаются в очередь, и даже если они не являются блокирующими и не ждут завершения длительных операций, другой рабочий процесс удаляется из очереди. Это крайний случай, способный ухудшить производительность параллельных вычислений, поскольку потребление памяти программой пропорционально количеству готовых к запуску рабочих процессов, которое может значительно превышать количество ядер процессора.

Еще один случай, на который следует обратить внимание, — это когда асинхронные операции, способные выполняться параллельно, ограничены внешними факторами. Например, при запуске консольного приложения, которое выполняет веб-запросы, максимальное количество конкурентных HTTP-соединений, разрешенных объектом `ServicePoint`¹, по умолчанию равно двум. В рассмотренном примере с хранилищем `Azure Blob` мы связываем `Async.Parallel` с параллельным

¹ Используется для получения или задания максимально допустимого количества конкурентных подключений.

выполнением нескольких длительных операций, но в итоге, если не изменить базовую конфигурацию, параллельно будут выполняться только два веб-запроса. Для достижения максимальной производительности кода рекомендуется настроить параллелизм программы, отрегулировав количество конкурентных вычислений.

В листинге 9.13 показан код, в котором реализованы две функции, `ParallelWithThrottle` и `ParallelWithCatchThrottle`. Эти функции могут быть использованы для изменения количества асинхронных операций, выполняемых конкурентно.

Листинг 9.13. ParallelWithThrottle и ParallelWithCatchThrottle

```

type Result<'a> = Result<'a, exn> ← Определение псевдонима для Result<'a>
module Result =
    let ofChoice value = ← Вспомогательная функция для отображения
        match value with ← между типами размеченные объединения Choice и Result
        | Choice1Of2 value -> Ok value
        | Choice2Of2 e -> Error e
    selector ← Функция selector применяет проекцию
    computations = async { ← к результату асинхронного вычисления
        let parallelWithCatchThrottle (selector:Result<'a> -> 'b) ←
            Максимальное ← Перечисление асинхронных вычислений
            количество ← для параллельного выполнения
            конкурентных ←
            асинхронных операций ←
            use semaphore = new SemaphoreSlim(throttle) ← Элемент блокировки, используемый
                для ограничений асинхронных вычислений
            let throttleAsync (operation:Async<'a>) = async { ←
                Выполнение ← Функция, применяемая
                вычисления ← для запуска каждого вычисления
                с защитой ← и ограничивающая параллелизм
                результата ← посредством элемента блокировки
                в случае ←
                исключения ←
                try ←
                    do! semaphore.WaitAsync()
                    let! result = Async.Catch operation
                    return selector (result |> Result.ofChoice)
                finally ←
                    semaphore.Release() |> ignore }
                Завершение ←
                вычисления ← Отображение результата с помощью типа
                и отмена ← размеченному объединению Result,
                блокировки ← а затем передача результата в функцию selector
            let parallelWithThrottle throttle computations =
                parallelWithCatchThrottle id throttle computations

```

Функция `parallelWithCatchThrottle` создает асинхронное вычисление, которое выполняет все заданные асинхронные операции, сначала ставя их в очередь как рабочие элементы и используя шаблон Fork/Join. Параллелизм является настраиваемым, так что большинство управляемых вычислений выполняются одновременно.

В листинге 9.13 функция `Async.Catch` используется для защиты параллельного асинхронного вычисления от неудачного завершения. Функция `parallelWithCatchThrottle` не генерирует исключений, вместо этого возвращая массив типов F# `Result`.

Вторая функция, `parallelWithThrottle`, является вариантом первой, в которой вместо аргумента `selector` используется `id`. Функция `id` в F# называется *функцией идентификации*, она является ярлыком для операции, которая возвращает сама себя: `(fun x -> x)`. В данном примере `id` используется для того, чтобы пропустить `selector` и вернуть результат операции без применения какого-либо преобразования.

В версии F# 4.1 появился тип `Result<'TSuccess, 'TError>` — удобное размеченнное объединение, которое поддерживает потребляющий код — теперь этот код может генерировать ошибку, не выполняя обработку исключений. Размеченное объединение `Result` обычно используется для представления и сохранения ошибки, которая может возникнуть во время выполнения.

Первая строка кода в предыдущем листинге определяет псевдоним типа `Result<'a>` для `Result<'a, exn>`, который предполагает, что в данном случае второй аргумент — это всегда исключение (`exn`). Цель создания псевдонима типа `Result<'a>` — упростить сопоставление шаблонов по `result`:

```
let! result = Async.Catch operation
```

Можно по-разному обрабатывать исключения в асинхронных операциях F#. Наиболее характерным является использование `Async.Catch` в качестве обертки, которая защищает вычисление путем перехвата всех исключений в исходном вычислении. В функции `Async.Catch` используется более функциональный подход, поскольку вместо функции в качестве аргумента для обработки ошибки она возвращает размеченное объединение, состоящее из элементов `Choice<'a, exn>`, где `'a` — это тип результата асинхронного рабочего процесса, а `exn` — исключение. Базовые значения результата `Choice<'a, exn>` могут быть извлечены посредством сопоставления с шаблоном. Обработка ошибок в функциональном программировании будет подробно рассмотрена в главе 10.

ПРИМЕЧАНИЕ

Недетерминированное поведение асинхронных параллельных вычислений означает, что мы не знаем, какое из асинхронных вычислений первым даст сбой. Но асинхронный комбинатор `Async.Parallel` сообщает о первом сбое из всех вычислений и отменяет остальные задачи, вызывая маркер отмены для группы задач.

`Choice<'T, exn>` — это размеченное объединение¹, состоящее из двух элементов:

- ❑ `Choice10f2 of 'T` содержит результат для случая успешного завершения рабочего процесса;
- ❑ `Choice20f2 of exn` соответствует сбою рабочего процесса и содержит активизированное исключение.

Обработка исключений посредством этой функциональной конструкции позволяет создавать асинхронный код в виде скомпонованной и естественной конвейерной структуры.

¹ Для получения дополнительной информации см.: <http://mng.bz/03fl>.

ПРИМЕЧАНИЕ

Функция `Async.Catch` сохраняет информацию об ошибке, что упрощает диагностику проблемы. Использование `Choice<_,_>` позволяет применять систему типов для создания вариантов обработки как результатов, так и ошибок.

`Choice<'T, 'U>` — это размеченное объединение, встроенное в ядро F#, что весьма удобно; однако в данном случае мы можем создать лучшее представление результата асинхронного вычисления, заменив размеченное объединение `Choice` более осмысленным `Result<'a>`¹ (листинг 9.14). (Код, на который стоит обратить внимание, выделен жирным шрифтом.)

Листинг 9.14. ParallelWithThrottle для загрузки из хранилища Azure Table

```

let maxConcurrentOperations = 100           ← Установка предела для максимального
                                             количества параллельных операций
ServicePointManager.DefaultConnectionLimit <- maxConcurrentOperations ← Установка значения
                                             DefaultConnectionLimit,
                                             которое по умолчанию равно двум
let downloadMediaCompAsyncParallelThrottle() = async {
    let! container = getCloudBlobContainerAsync()
    let computations =
        container.ListBlobs()           ← Создание списка асинхронных операций
        |> Seq.map(fun blobMedia -> transformAndSaveImage container blobMedia)

    return! Async.parallelWithThrottle
          maxConcurrentOperations computations } ← Выполнение асинхронных операций
                                             с ограничением параллелизма

```

В этом коде посредством `ServicePointManager.DefaultConnectionLimit` устанавливается предельное значение для количества параллельных запросов `maxConcurrentOperations`, равное 100. Это же значение передается в качестве аргумента `parallelWithThrottle` для ограничения конкурентных запросов. `maxConcurrentOperations` — произвольное число, которое может быть сколь угодно большим, но я рекомендую вам протестировать и измерить время выполнения и потребление памяти вашей программы, чтобы определить, какое значение является наилучшим для производительности.

¹ Описано в главе 4.

Резюме

- ❑ Асинхронное программирование позволяет параллельно загружать несколько изображений, избегая аппаратных зависимостей и высвобождая неограниченные вычислительные возможности.
- ❑ Язык функционального программирования F# обеспечивает полную поддержку асинхронного программирования, интегрируясь в модель асинхронного программирования, предоставляемую .NET. Он также имеетстроенную функциональную реализацию APM, называемую асинхронным рабочим процессом, которая может взаимодействовать с моделью программирования на основе задач, реализованной в C#.
- ❑ В основе асинхронного рабочего процесса F# лежит тип `Async<'a>`. Он определяет вычисление, которое будет завершено в будущем. Это обеспечивает отличные возможности компоновки, потому что вычисления не начинаются сразу. Для асинхронного вычисления требуется явный запрос запуска.
- ❑ Время выполнения нескольких последовательных синхронных операций ввода-вывода равно сумме времени выполнения каждой из операций — в отличие от асинхронного подхода, при котором операции выполняются параллельно, и поэтому общее время отклика равно времени выполнения самой медленной операции.
- ❑ Используя стиль продолжений, соответствующий функциональной парадигме, можно писать гораздо более краткий код. Этот код легко сделать многопоточным.
- ❑ Вычислительные выражения F#, в частности, в форме асинхронного рабочего процесса образуют цепочку и выполняют набор вычислений асинхронно, не блокируя выполнение другой работы.
- ❑ Вычислительные выражения могут быть расширены для работы с разными надтипами без необходимости покидать текущий контекст. Также можно создавать собственные вычислительные выражения, позволяющие расширить возможности компилятора.
- ❑ Для обработки особых случаев возможно создание адаптированных асинхронных комбинаторов.

Функциональные комбинаторы для быстрого конкурентного программирования

В этой главе:

- обработка исключений в стиле функционального программирования;
- использование встроенных комбинаторов задач;
- реализация нестандартных асинхронных комбинаторов и условных операторов;
- выполнение параллельных асинхронных гетерогенных вычислений.

В двух предыдущих главах вы узнали, как использовать асинхронное программирование для разработки масштабируемых и эффективных систем. Мы применили средства функционального программирования для компоновки, управления и оптимизации параллельного выполнения нескольких задач. В этой главе мы еще больше повысим уровень абстракции для описания асинхронных вычислений в функциональном стиле.

Сначала мы рассмотрим способы управления исключениями в функциональном стиле, обращая особое внимание на асинхронные операции. Затем мы изучим *функциональные комбинаторы* — полезный инструмент программирования для построения наборов вспомогательных функций. Функциональные комбинаторы позволяют нам создавать сложные функции, компонуя их из небольших и более кратких операторов. Такие комбинаторы и методики позволяют сделать код более удобным для обслуживания и более производительным, расширят наши возможности по на-

писанию конкурентных вычислений и обработке побочных эффектов. В конце главы мы узнаем, как организовать взаимодействие между C# и F#, вызывая и передавая асинхронные функции между данными языками.

Из всех глав настоящей книги эта — одна из самых сложных, потому что касается теории функционального программирования, и лексика здесь первоначально может восприниматься как жаргон. Но чем больше усилий — тем выше награда...

Концепции, описанные в данной главе, снабдят вас исключительными инструментами, которые позволяют легко и просто создавать сложные конкурентные программы. Обычному программисту не обязательно знать, как работает сборщик мусора (garbage collector, GC) в .NET, поскольку он выполняется в фоновом режиме. Но если разработчик понимает детали функционирования GC, то он может максимально эффективно использовать память и повысить производительность программы.

В этой главе мы пересмотрим примеры из главы 9, несколько усложнив их. Примеры будут написаны на C# или F#, в зависимости от того, какой из данных языков программирования лучше соответствует рассматриваемой идеи. Но все эти концепции применимы к обоим языкам программирования, и в большинстве случаев вы найдете альтернативный пример в исходном коде к книге.

Эта глава поможет вам понять семантику компоновки при обработке ошибок в стиле функционального программирования и функциональные комбинаторы, так что вы сможете писать эффективные программы для конкурентной (и параллельной) безопасной обработки асинхронных операций с минимальными усилиями и высокой производительностью.

В конце главы вы научитесь использовать встроенные асинхронные комбинаторы, а также разрабатывать и внедрять эффективные комбинаторы собственной разработки, которые отлично соответствуют требованиям ваших приложений. Вы сможете повысить уровень абстракции в сложных и медленных частях кода, чтобы без особых усилий упростить структуру программы, управлять потоком и сократить время выполнения.

10.1. Поток выполнения не всегда проходит по «счастливому пути»: обработка ошибок

При разработке программного обеспечения может возникнуть множество неожиданных проблем. Корпоративные приложения, как правило, являются распределенными и зависят от нескольких внешних систем, что может привести к различным проблемам. Вот несколько примеров таких проблем:

- ❑ потеря сетевого подключения во время веб-запроса;
- ❑ приложениям не удалось обменяться данными с сервером;
- ❑ при обработке данные случайно обнулились;
- ❑ возникли исключения.

Наша цель как разработчиков — писать надежный код, который бы учитывал эти проблемы. Но решение потенциальных проблем может создавать свои сложности. В реальных приложениях поток выполнения не всегда проходит по «счастливому пути», когда поведение по умолчанию является безошибочным (рис. 10.1). Чтобы предупредить возникновение исключений и упростить процесс отладки, необходимо использовать логику валидации, проверку значений, журналирование и ветвление кода. Как правило, компьютерные программисты склонны злоупотреблять и даже чрезмерно злоупотреблять исключениями. Например, генерация исключений в коде является обычным делом, и если в этом контексте отсутствует соответствующий обработчик, то объект, вызывающий данный фрагмент кода, вынужден обрабатывать исключение на несколько уровней выше по стеку вызовов.

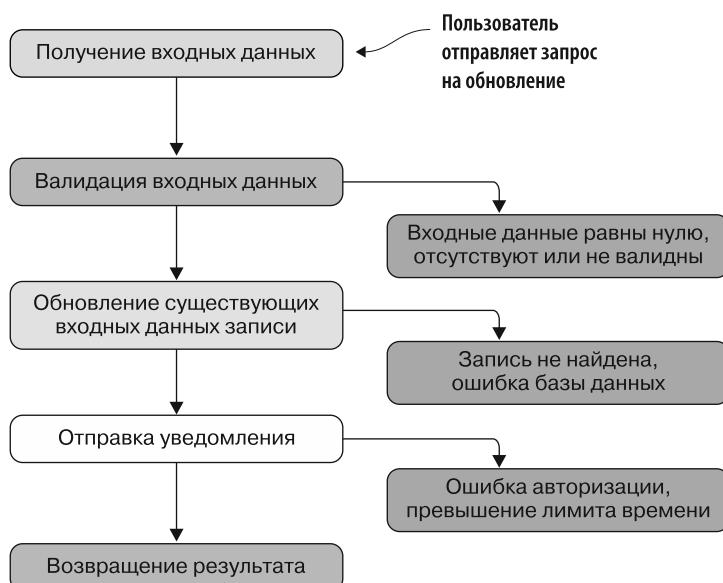


Рис. 10.1. Пользователь отправляет запрос на обновление, которое может легко отклониться от «счастливого пути». Мы часто пишем код, думая, что что-то не может пойти не так. Но при создании качественного кода необходимо учитывать исключения и возможные проблемы, такие как валидация, сбой или ошибки, мешающие правильному выполнению кода

При асинхронном программировании обработка ошибок важна — она позволяет обеспечить безопасное выполнение приложения. Предполагается, что асинхронная операция завершится; но что, если что-то пойдет не так и операция не прекратится никогда? Парадигмы функционального и императивного программирования предлагают разные подходы для обработки ошибок.

- ❑ Обработка ошибок при императивном программировании основана на побочных эффектах. В императивных языках для создания побочных эффектов исполь-

зуются блоки `try-catch` и операторы `throw`. Эти побочные эффекты нарушают нормальное выполнение программы, так что код становится трудно понять. При применении традиционного стиля императивного программирования наиболее распространенным подходом при обработке ошибок является защита методов от возникновения ошибки и возврат значения `null`, если результат пуст. Эта концепция обработки ошибок широко распространена, но обработка ошибок таким образом в императивных языках — не лучший вариант, потому что она создает большие возможности для ошибок кода.

- В функциональном программировании основное внимание уделяется сведению к минимуму и контролю побочных эффектов, поэтому при обработке ошибок обычно стремятся избегать изменения состояний и генерации исключений. Например, если операция завершается неудачно, то она должна возвращать структурное представление результата, которое включает в себя уведомление об успешном или неудачном завершении.

Проблема обработки ошибок в императивном программировании. В .NET Framework легко перехватывать ошибки асинхронных операций и реагировать на них. Один из способов сделать это — обернуть весь код, принадлежащий одному и тому же асинхронному вычислению, в блок `try-catch`.

Для того чтобы проиллюстрировать проблему обработки ошибок и понять, как ее можно решить в функциональном стиле, вернемся к примеру загрузки изображений из хранилища Azure Blob (см. главу 9). В листинге 10.1 показано, как защитить метод `DownloadImageAsync` от исключений, которые могут возникнуть во время его выполнения (выделены жирным шрифтом).

Это кажется простым и понятным: сначала вызывающая программа `RunDownloadImageAsync` вызывает `DownloadImageAsync` и обрабатывает возвращенное изображение. В данном примере кода уже предполагается, что что-то может пойти не так, и ядро выполнения обернуто в блок `try-catch`. Делать ставку на «счастливый путь» — сценарий выполнения, когда все идет правильно, — роскошь, которую программист не может себе позволить при создании надежных приложений.

Как видим, когда мы начинаем учитывать возможные сбои, ошибки ввода и процедуру журналирования, метод начинает превращаться в длинный стереотипный код. Если удалить оттуда строки обработки ошибок, то останется всего девять строк значимых основных функций; еще 21 строка — это стереотипное оформление, предназначеннное для обработки ошибок и журналирования.

Подобный нелинейный поток выполнения программы может быстро стать запутанным, поскольку трудно отследить все существующие взаимосвязи между операторами `throw` и `catch`. Более того, из-за исключений неясно, где именно возникают ошибки. Можно обернуть процедуру валидации в инструкции `try-catch` непосредственно в том месте, где она вызывается, или же вставить блок `try-catch` на пару уровней выше. Становится трудно понять, намеренно ли было сгенерировано исключение при ошибке.

В листинге 10.1 тело метода `DownloadImageAsync` обернуто в блок `try-catch` для защиты программы в случае возникновения исключения. Но при этом не выполняется обработка ошибок; исключение передается дальше, а данные об ошибках заносятся в журнал. Цель блока `try-catch` состоит в том, чтобы предотвратить исключение; для этого небезопасный фрагмент кода заключается внутри такого блока. Но если генерируется исключение, среда выполнения создает отслеживание стека всех вызовов функций, ведущих к инструкции, которая вызвала ошибку.

Листинг 10.1. Метод `DownloadImageAsync` с традиционной императивной обработкой ошибок

```
static async Task<Image> DownloadImageAsync(string blobReference)
{
    try
    {
        var container = await Helpers.GetCloudBlobContainerAsync();
        ➔ ConfigureAwait(false);
        CloudBlockBlob blockBlob = container.
        ➔ GetBlockBlobReference(blobReference);
        using (var memStream = new MemoryStream())
        {
            await blockBlob.DownloadToStreamAsync(memStream);
        }
        ➔ ConfigureAwait(false);
        return Bitmap.FromStream(memStream);
    }
}
catch (StorageException ex)
{
    Log.Error("Azure Storage error", ex);
    throw;
}
catch (Exception ex)
{
    Log.Error("Some general error", ex);
    throw;
}
async RunDownloadImageAsync()
{
    try
    {
        var image = await DownloadImageAsync("Bugghina0001.jpg");
        ProcessImage(image);
    }
    catch (Exception ex)
    {
        HanldingError(ex);
        throw;
    }
}
```

Где-то в верхней части стека вызовов

Обработка и перенаправление ошибок, чтобы поднять исключение по стеку вызовов

Отслеживание операций, которые могут вызвать исключение

`DownloadImageAsync` выполняется, но какие меры предосторожности должны использоваться, чтобы гарантировать обработку всех возможных ошибок? Следует ли на всякий случай также обернуть вызывающий объект в блок `try-catch`?

```
Image image = await DownloadImageAsync("Buggina001.jpg");
```

Как правило, вызывающая функция отвечает за защиту кода, проверяя состояние объектов на валидность перед их использованием. Но что случится, если такой проверки состояния нет? Ответ прост: будет больше проблем и ошибок.

Кроме того, когда одна и та же функция `DownloadImageAsync` появится в нескольких местах кода, программа станет еще сложнее, поскольку разным вызывающим объектам может понадобиться разная обработка ошибок; это приводит к утечкам памяти и излишне сложным моделям предметной области.

10.2. Комбинаторы ошибок: `Retry`, `Otherwise` и `Task.Catch` в C#

В главе 8 мы определили два метода расширения для типа `Task`: `Retry` и `Otherwise` (резервный сценарий), определяющие логику для обработки исключений асинхронных операций `Task`. К счастью, поскольку асинхронные операции являются уязвимыми для исключений вследствие внешних факторов, тип `Task` в .NET имеет встроенную обработку ошибок с помощью свойств `Status` и `Exception`, как будет показано далее (листинг 10.2) (методы `Retry` и `Otherwise` выделены жирным шрифтом).

Листинг 10.2. Обновление функций `Otherwise` и `Retry`

```
static async Task<T> Otherwise<T>(this Task<T> task,
➥ Func<Task<T>> orTask) =>
    task.ContinueWith(async innerTask => {
        if (innerTask.Status == TaskStatus.Faulted)
            return await orTask();
        return await Task.FromResult<T>(innerTask.Result);
    }).Unwrap();
```

Резервная функция на случай, если что-то пойдет не так


```
static async Task<T> Retry<T>(Func<Task<T>> task, int retries, TimeSpan
➥ delay, CancellationToken cts = default(CancellationToken))
    => await task().ContinueWith(async innerTask =>
{
    cts.ThrowIfCancellationRequested();
    if (innerTask.Status != TaskStatus.Faulted)
        return innerTask.Result;
    if (retries == 0)
        throw innerTask.Exception ?? throw new Exception();
    await Task.Delay(delay, cts);
    return await Retry(task, retries - 1, delay, cts);
}).Unwrap();
```

Повторное выполнение функции указанное количество раз с заданной задержкой между попытками

Функции `Retry` и `Otherwise` рекомендуется использовать для управления ошибками в коде. Например, с помощью этих вспомогательных функций можно переписать вызов метода `DownloadImageAsync` следующим образом:

```
Image image = await AsyncEx.Retry(async () =>
    await DownloadImageAsync("Bugghina001.jpg")
    .Otherwise(async () =>
        await DownloadImageAsync("Bugghina002.jpg"),
        5, TimeSpan.FromSeconds(2));
```

Благодаря применению функций `Retry` и `Otherwise` в предыдущем коде функция `DownloadImageAsync` меняет поведение и ее выполнение становится более безопасным. Если что-то пойдет не так, когда функция `DownloadImageAsync` будет извлекать изображение `Bugghina001`, будет выполнена резервная операция — загрузка альтернативного изображения. Логика `Retry`, которая включает в себя поведение `Otherwise` (резервный сценарий), повторяется до пяти раз с задержкой две секунды между операциями, пока одна из них не завершится успешно (рис. 10.2).

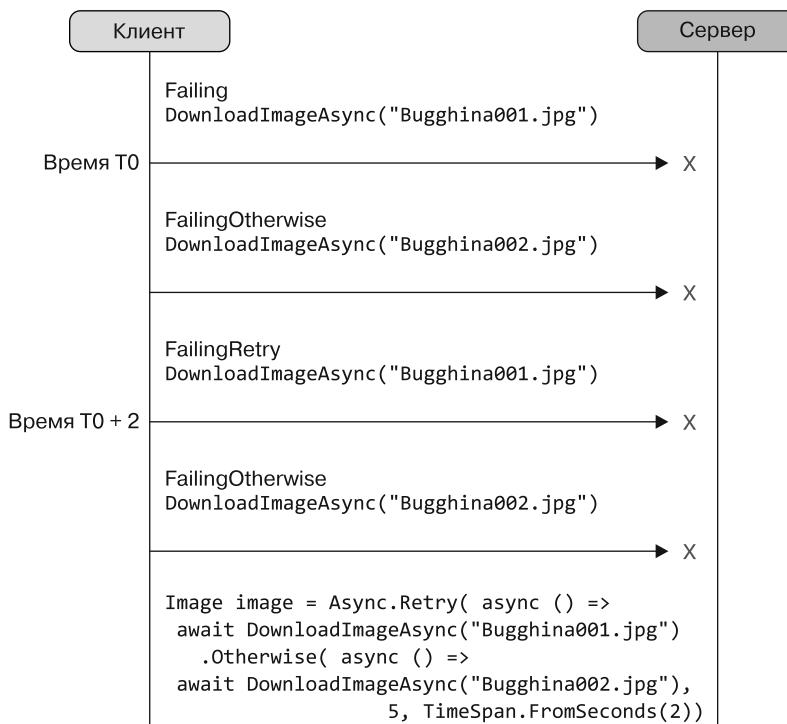


Рис. 10.2. Клиент отправляет на сервер два запроса, чтобы в случае сбоя применить стратегии `Otherwise` (резервный сценарий) и `Retry`. Выполнение этих запросов (`DownloadImageAsync`) является безопасным, поскольку в обоих случаях для обработки потенциальных проблем используются стратегии `Retry` и `Otherwise`

Кроме того, можно определить еще один метод расширения, например функцию `Task.Catch`, специально предназначенную для обработки исключений, генерируемых при выполнении асинхронных операций (листинг 10.3).

Листинг 10.3. Функция Task.Catch

```
static Task<T> Catch<T, TError>(this Task<T> task,
➥ Func<TError, T> onError) where TError : Exception
{
    var tcs = new TaskCompletionSource<T>(); ← Экземпляр TaskCompletionSource
    task.ContinueWith(innerTask =>           возвращает тип Task, чтобы сохранить
    {                                         согласованность в асинхронной модели
        if (innerTask.IsFaulted && innerTask?.Exception?.InnerException
    ➥ is TError)
            tcs.SetResult(onError((TError)innerTask.Exception.
    ➥ InnerException));
        else if (innerTask.IsCanceled)
            tcs.SetCanceled();
        else if (innerTask.IsFaulted)
            tcs.SetException(innerTask?.Exception?.InnerException ???
    ➥ throw new InvalidOperationException());
        else
            tcs.SetResult(innerTask.Result);
    });
    return tcs.Task;                         Назначение Result или Exception для TaskCompletionSource
}                                         в зависимости от результата Task
```

Преимуществом функции `Task.Catch` является описание конкретных вариантов исключений в качестве конструкторов типов. В следующем фрагменте показан пример обработки `StorageException` в контексте задачи с хранилищем Azure Blob (код, на который следует обратить внимание, выделен жирным шрифтом):

```
static Task<Image> CatchStorageException(this Task<Image> task) =>
    task.Catch<Image, StorageException>(ex => Log($"Azure Blob
➥ Storage Error {ex.Message}"));
```

Метод расширения `CatchStorageException` может применяться так, как показано в следующем фрагменте кода:

```
Image image = await DownloadImageAsync("Bugghina001.jpg")
    .CatchStorageException();
```

Правда, подобная структура может нарушать принцип нелокальности, поскольку код, используемый для восстановления после возникновения исключения, отличается от первоначального вызова функции. Кроме того, это никак не поддерживается компилятором, так что невозможно уведомить разработчика о том, что в объекте,зывающем метод `DownloadImageAsync`, применяется обработка ошибок, так как его возвращаемый тип — это обычный примитив `Task`, который не требует валидации и не выполняет ее. В последнем случае, если обработка ошибок пропущена или о ней забыли, может возникнуть исключение, способное вызвать непредвиденные

побочные эффекты. Эти эффекты могут повлиять на всю систему (помимо вызывающей функции), что приведет к катастрофическим последствиям, таким как сбой приложения. Как видим, исключения разрушают способность рассуждать о коде. Более того, структурированный механизм генерирования и перехвата исключений в императивном программировании имеет недостатки, которые противоречат принципам функционального проектирования. Например, функции, генерирующие исключения, не могут быть скомпонованы или объединены в цепочки, подобно другим функциональным артефактам.

Обычно код читают чаще, чем пишут, поэтому неудивительно, что рекомендуется всячески упрощать его понимание и обсуждение. Чем проще код, тем меньше ошибок он содержит и тем легче обслуживать программное обеспечение в целом. Использование исключений при управлении потоком выполнения программ скрывает намерения программиста, поэтому считается плохой практикой. К счастью, можно относительно легко избежать создания сложного и загроможденного кода.

Решение состоит в том, чтобы вместо генерации исключений явно возвращать значения, указывающие на успешное или неудачное завершение операции. Это придает ясность тем частям кода, которые потенциально подвержены ошибкам. В следующих разделах я покажу два возможных подхода, которые соответствуют парадигме функционального программирования и позволяют упростить семантическую структуру обработки ошибок.

10.2.1. Обработка ошибок в функциональном программировании: исключения при управлении потоком выполнения

Изменим метод `DownloadImageAsync`, на этот раз с обработкой ошибок в функциональном стиле. Сначала мы рассмотрим пример кода, а затем, в листинге 10.4, углубимся в детали. Новый метод `DownloadOptionImage` перехватывает исключения в блоке `try-catch`, как в предыдущей версии кода, но здесь результат имеет тип `Option` (выделен жирным шрифтом).

Листинг 10.4. Тип Option для обработки ошибок в функциональном стиле

```
async Task<Option<Image>> DownloadOptionImage(string blobReference)
{
    try
    {
        var container = await Helpers.GetCloudBlobContainerAsync();
        ConfigureAwait(false);
        CloudBlockBlob blockBlob = container.
        GetBlockBlobReference(blobReference);
        using (var memStream = new MemoryStream())
        {
            await
            blockBlob.DownloadToStreamAsync(memStream).ConfigureAwait(false);
        }
    }
}
```

Выходное значение функции — это компоновка `Task`, в которую обернут тип `Option`

```
        return Option.Some(Bitmap.FromStream(memStream)); ◀
    }
}
catch (Exception)
{
    return Option.None;
}
```

Результат типа Option — это либо значение Some
в случае успешного завершения операции,
либо None (ничего) в случае ошибки

Тип `Option` уведомляет объект, вызывающий функцию, что операция `DownloadOptionImage` вернула некий конкретный результат, который нуждается в специальной обработке. Фактически тип `Option` в качестве результата может быть равен либо `Some`, либо `None`. Следовательно, объект, вызывающий функцию `DownloadOptionImage`, вынужден проверять результат на значение. Если он содержит значение, то операция завершилась успешно; если же это не так, значит, произошел сбой. Подобная валидация требует от программиста написания кода для обработки обоих возможных результатов. Использование данной конструкции делает код предсказуемым, позволяя избежать побочных эффектов и дает возможность применять `DownloadOptionImage` при компоновке.

Управление побочными эффектами с помощью типа Option. В функциональном программировании понятия `null`-значений не существует. В функциональных языках, таких как Haskell, Scala и F#, такая проблема решается путем обертывания значений `null` в тип `Option`. В F# тип `Option` является решением проблемы исключений с `null`-указателями; это размеченное объединение (discriminated union, DU) с двумя состояниями, которое используется для обертывания значения (`Some`) или отсутствия значения (`None`). Это можно представить как коробку, в которой или что-то находится, или нет. Концептуально можно рассматривать тип `Option` как нечто, что присутствует или отсутствует. Символически определение типа `Option` выглядит так:

```
type Option<'T> =
| Some of value:T
| None
```

Вариант `Some` означает, что данные хранятся в ассоциированном внутреннем значении `T`; вариант `None` означает, что данных нет. Например, `Option<Image>` может содержать или не содержать изображение. На рис. 10.3 показано сравнение между примитивом, который может принимать значение `null`, и эквивалентным типом `Option`.

Экземпляр типа `Option` создается путем вызова либо метода `Some(value)`, который представляет собой положительный ответ, либо `None`, что эквивалентно возвращению пустого значения. В F# не нужно специально определять тип `Option`, так как он входит в состав стандартной библиотеки F# в комплекте с богатым набором вспомогательных функций.

В C# есть тип `Nullable<T>`, ограниченный типами значений. Первоначальным решением является создание общей структуры, в которую обернуто значение. Использование типа значения (`struct`) важно для уменьшения расхода памяти

и идеально подходит для того, чтобы избежать исключений нулевых ссылок путем присвоения значения `null` самому типу `Option`.

	Объект со значением	Пустой объект
Обычный примитив	<code>string x = "value"</code>	<code>string x = null</code>
Тип Option	<code>Optional<string> x = Some("value")</code>	<code>Optional<string> x = None</code>

Рис. 10.3. Наглядное сравнение обычных примитивов, которые могут принимать значение `null` (верхняя строка), и типов `Option` (нижняя строка). Основное различие состоит в том, что обычный примитив может иметь либо валидное, либо невалидное значение (`null`) без уведомления вызывающего объекта, тогда как тип `Option` обертывает примитив, предлагая вызывающему объекту проверить, является ли внутреннее значение валидным

Чтобы тип `Option` можно было применять многократно, мы воспользуемся параметризованной структурой C# `struct Option<T>`; она служит оберткой для произвольного типа, который может содержать, а может и не содержать значение. Базовая структура `Option<T>` имеет свойство `value` типа `T` и флаг `HasValue`, который указывает, установлено значение или нет.

Реализовать тип `Option` в C# легко, поэтому мы здесь не прилагаем подробный пример. Если вы захотите глубже разобраться в реализации типа `Option` на C#, обратитесь к исходному коду данной книги, выложенному на сайте издательства. Более высокий уровень абстракции, достигаемый с помощью типа `Option<T>`, позволяет реализовать функции более высокого порядка (Higher-Order Functions, HOF), такие как `Match` и `Map`, что упрощает компоновочную структуру кода и в данном случае благодаря функции `Match` позволяет использовать сопоставление шаблонов и деконструктивную семантику:

```
R Match<R>(Func<R> none, Func<T, R> some) => hasValue ? some(value) :  
  none();
```

Функция `Match` принадлежит экземпляру типа `Option`. Это удобная конструкция, благодаря которой исключаются лишние преобразования типов и улучшается читаемость кода.

10.2.2. Обработка ошибок с помощью `Task<Option<T>>` в C#

В листинге 10.4 я проиллюстрировал, как тип `Option` защищает код от ошибок, делая программу более защищенной от исключений с нулевыми указателями, и предположил, что компилятор поможет избежать случайных ошибок. В отличие от значений

null тип Option заставляет разработчика писать логику, позволяющую проверить, существует ли данное значение, тем самым смягчая многие проблемы со значениями null и error.

Вернемся к примеру с хранилищем Azure Blob. Используя тип Option и HOF Match, можно выполнить функцию DownloadOptionImage, возвращаемым типом которой является Task<Option<Image>>:

```
Option<Image> imageOpt = await DownloadOptionImage ("Bugghina001.jpg");
```

При применении возможности компоновки, свойственной типам Task и Option и их расширенным HOF, запись в стиле функционального программирования (выделена жирным шрифтом) выглядит следующим образом:

```
DownloadOptionImage ("Bugghina001.jpg")
    .Map(opt => opt.Match(
        some: image => image.Save("ImageFolder\Bugghina.jpg"),
        none: () => Log("There was a problem downloading
            ➔ the image")));
```

Этот окончательный код является гибким и выразительным и, что более важно, позволяет сократить количество ошибок, поскольку компилятор заставляет вызывающий объект учитывать все возможные результаты — как успешный, так и неудачный.

10.2.3. Тип AsyncOption в F#: сочетание Async и Option

Такой же подход к обработке исключений с использованием типа Task<Option<T>> применим и в F#. Такая же технология может быть применена в асинхронном рабочем процессе F# для достижения подхода, более свойственного этому языку.

Улучшением F#, по сравнению с C#, является поддержка *псевдонимов типов*, также называемых *сокращениями типов*. Псевдоним типа используется для того, чтобы не писать каждый раз полную сигнатуру и упростить работу с кодом. Например, псевдоним типа Async<Option<'T>> выглядит так:

```
type AsyncOption<'T> = Async<Option<'T>>
```

Определение AsyncOption<'T> можно задействовать непосредственно в коде вместо Async<Option<'T>> с тем же поведением. Другое назначение псевдонима типа — ослабление связи между использованием типа и его реализацией. В листинге 10.5 показана эквивалентная F#-реализация функции DownloadOptionImage, ранее созданной на C#.

Функция downloadOptionImage асинхронно загружает изображение из хранилища Azure Blob. Функция Async.map с сигнатурой ('a → 'b) -> Async<'a> → Async<'b> обертывает результат функции и обеспечивает доступ к внутреннему значению. В таком случае параметрический тип 'a — это Option<Image>.

Листинг 10.5. Реализация и использование псевдонима типа AsyncOption на F#

```

Блок try-with безопасно управляет
потенциальными ошибками
let downloadOptionImage(blobReference:string) : AsyncOption<Image> = ←
    async {
        try
            let! container = Helpers.getCloudBlobContainerAsync()
            let blockBlob = container.GetBlockBlobReference(blobReference)
            use memStream = new MemoryStream()
            do! blockBlob.DownloadToStreamAsync(memStream)
            return Some(Bitmap.FromStream(memStream)) ←
        with
            | _ -> return None ← Создание типа Option
    } ← со значением Some (изображение)

downloadOptionImage "Bugghina001.jpg"
|> Async.map(fun imageOpt -> ← Применение функции более высокого
    match imageOpt with ← порядка Async.map, которая обращается
    | Some(image) -> do! image.SaveAsync("ImageFolder\Bugghina.jpg") ← к внутреннему значению Option и проектирует его
    | None -> log "There was a problem downloading the image" ← Шаблон сопоставляет тип Option
    ← для деконструирования
Реализация метода расширения изображения SaveAsync
есть в исходном коде, выложенном онлайн ← и доступа к обернутому значению Image

```

ПРИМЕЧАНИЕ

Один из важных моментов, показанных в листинге 10.5, заключается в том, что линейная реализация асинхронного кода допускает обработку исключений тем же способом, что и в синхронном коде. Этот механизм, основанный на блоке try-with, гарантирует, что возникающие исключения станут подниматься по уровням стека через неблокирующие вызовы и в итоге будут обработаны обработчиком исключений. Данный механизм гарантированно корректно работает, несмотря на наличие нескольких потоков, которые в это время выполняются параллельно.

Удобно, что функции, принадлежащие модулю F# `Async`, могут применяться к псевдониму `AsyncOption`, потому что это тип `Async`, служащий оберткой для `Option`. Функция внутри оператора `Async.map` извлекает значение `Option`, которое является шаблоном, соответствующим выбору поведения для выполнения сценария, в зависимости от значения `Option` — `Some` или `None`.

10.2.4. Функциональная асинхронная обработка ошибок, свойственная F#

Сейчас функция F# `downloadOptionImage` безопасно загружает изображение, гарантировая, что в случае проблемы исключение будет перехвачено, при этом не ставится под угрозу стабильность приложения. Но по возможности следует избегать самого

блока `try-with`, эквивалентного `try-catch` в C#, поскольку он поощряет использование нечистого (с побочными эффектами) стиля программирования. В контексте асинхронных вычислений модуль F# `Async` обеспечивает присущий этому языку функциональный подход, используя функцию `Async.Catch` как оболочку, которая защищает вычисления.

ПРИМЕЧАНИЕ

Функция `Async.Catch` была описана в главе 9. Она требует более функционального подхода, поскольку вместо функции в качестве аргумента для обработки ошибок возвращает размеченное объединение `Choice<'a, exn>`, где `'a` — это тип результата асинхронного рабочего процесса, а `exn` — исключение.

`Async.Catch` можно использовать для безопасного выполнения и сопоставления асинхронных операций с типом `Choice<'a, exn>`. Чтобы уменьшить количество стереотипного кода, а также, в большинстве случаев, упростить код, можно создать вспомогательную функцию, которая служит оберткой для `Async<'T>` и возвращает `AsyncOption<'T>` посредством оператора `Async.Catch`. Эта реализация показана в следующем фрагменте кода. Вспомогательная функция `Choice` дополняет модуль F# `Option`, целью которого являются отображение и преобразование типа `Choice` в тип `Option`:

```
module Option =
    let ofChoice choice =
        match choice with
        | Choice1Of2 value -> Some value
        | Choice2Of2 _ -> None
module AsyncOption =
    let handler (operation:Async<'a>) : AsyncOption<'a> = async {
        let! result = Async.Catch operation
        return (Option.ofChoice result)
    }
```

`Async.Catch` используется для преобразования типа `Async<'T>` в `Async<Choice<'T, exn>>` при обработке исключений. Этот тип `Choice` затем преобразуется в `Option<'T>` посредством простого преобразования функции `ofChoice`. Функция-обработчик `AsyncOption` может безопасно выполнять асинхронные операции `Async<'T>` и преобразовывать их в тип `AsyncOption`.

В листинге 10.6 показана реализация функции `downloadOptionImage` без необходимости защищать код посредством блока `try-with`. Функция `AsyncOption.handler` управляет выходом, независимо от того, завершилась операция успешно или нет. Теперь, если возникает ошибка, `Async.Catch` перехватит ее и преобразует в тип `Option` посредством функции `Option.ofChoice` (выделена жирным шрифтом).

Листинг 10.6. Псевдоним типа `AsyncOption` в действии

```
let downloadAsyncImage(blobReference:string) : Async<Image> = async {
    let! container = Helpers.getCloudBlobContainerAsync()
    let blockBlob = container.GetBlockBlobReference(blobReference)
```

```

use memStream = new MemoryStream()
do! blockBlob.DownloadToStreamAsync(memStream)
return Bitmap.FromStream(memStream)
}

downloadAsyncImage "Bugghina001.jpg"
|> AsyncOption.handler
|> Async.map(fun imageOpt ->
    match imageOpt with
    | Some(image) -> image.Save("ImageFolder\Bugghina.jpg")
    | None -> log "There was a problem downloading the image")
|> Async.Start

```

Функция `AsyncOption.handler` — это многоразовый и компонуемый оператор, который может применяться к любой асинхронной операции.

10.2.5. Сохранение семантики исключений с помощью типа `Result`

В подразделе 10.2.2 вы увидели, как тип `Option` используется в функциональной парадигме для обработки ошибок и контроля побочных эффектов. В контексте обработки ошибок `Option` действует как контейнер — область, в которой побочные эффекты исчезают и растворяются, не создавая нежелательного поведения в программе. В функциональном программировании понятие заключения опасного кода, способного вызывать ошибки, в замкнутую область, не ограничивается типом `Option`.

В этом разделе мы сохраним семантику обработки ошибок, но воспользуемся типом `Result`, который позволяет разделять программу и создавать в ней ветви с разными вариантами поведения в зависимости от типа ошибки. Предположим, что в рамках реализации приложения мы хотим упростить отладку или сообщить объекту, вызывающему функцию, подробную информацию об исключении, если что-то пойдет не так. В этом случае подход с использованием типа `Option` не годится для решения задачи, поскольку в качестве информации о том, что что-то пошло не так, он передает `None` (ничего). Когда передается результат `Some`, его значение недвусмысленно, но результат `None` не дает никакой информации, кроме очевидной. Отбрасывая исключение, невозможно диагностировать проблему.

Вернемся к нашему примеру с загрузкой изображения из хранилища Azure Blob. Если что-то пойдет не так во время получения данных, это могут быть разные ошибки, возникающие в разных ситуациях, таких как потеря сетевого соединения или невозможность найти файл/изображение. В любом случае вам необходимо знать подробности об ошибках, чтобы выбрать правильную стратегию восстановления после исключения.

В листинге 10.7 метод `DownloadOptionImage` из предыдущего примера извлекает изображение из хранилища Azure Blob. Тип `Option` (выделен жирным шрифтом)

используется для обработки выходных данных более безопасным способом, управляя событиями ошибок.

Листинг 10.7. Тип Option, не сохраняющий данные об ошибке

```
async Task<Option<Image>> DownloadOptionImage(string blobReference)
{
    try
    {
        CloudStorageAccount storageAccount =
            CloudStorageAccount.Parse("<Azure Connection>");
        CloudBlobClient blobClient =
            storageAccount.CreateCloudBlobClient();
        CloudBlobContainer container =
            blobClient.GetContainerReference("Media");
        await container.CreateIfNotExistsAsync();

        CloudBlockBlob blockBlob = container.
            GetBlockBlobReference(blobReference);
        using (var memStream = new MemoryStream())
        {
            await blockBlob.DownloadToStreamAsync(memStream).
                ConfigureAwait(false);
            return Some(Bitmap.FromStream(memStream));
        }
    }
    catch (StorageException)
    {
        return None; ←
    }
    catch (Exception)
    {
        return None; ←
    }
}
```

Тип Option возвращает None в обоих случаях, независимо от типа исключения

Ограничение данной реализации кода заключается в том, что объект, вызывающий метод `DownloadOptionImage`, независимо от типа исключения — `StorageException` или общее `Exception`, — не получает никакой информации о самом исключении и поэтому не может выбрать соответствующую стратегию для восстановления.

Есть ли лучший способ? Как этот метод может предоставить информацию о возможной ошибке и избежать побочных эффектов? Решение состоит в том, чтобы вместо типа `Option<'T>` использовать полиморфный `Result<'TSuccess, 'TError>`.

`Result<'TSuccess, 'TError>` может применяться для обработки ошибок в функциональном стиле, а также содержать данные о причине потенциального отказа. На рис. 10.4 показано сравнение примитива с возможным значением `null`, эквивалентного типа `Option` и типа `Result`.

В некоторых языках программирования, таких как Haskell, структура `Result` называется `Either` и представляет собой логическое разделение двух значений, никогда не возникающих одновременно. Например, `Result<int, string>` моделирует два случая и может принимать одно из двух значений: либо `int`, либо `string`.

	Объект со значением	Пустой объект
Обычный примитив	string x = "value"	string x = null
Тип Option	Optional<string> x = Some("value")	Optional<string> x = None
Тип Result	Result<string> x = Success("value")	Result<string> x = Failure(error)
		Если что-то пойдет не так, в Failure можно будет найти описание ошибки

Рис. 10.4. Сравнение обычного примитива с возможным значением null (верхняя строка), типа Option (вторая строка) и типа Result (нижняя строка). Значение Result Failure обычно используется как обертка ошибки в случае, если что-то пойдет не так

Структура `Result<'TSuccess, 'TFailure>` также может использоваться для защиты кода от непредсказуемых ошибок, что делает его более безопасным по типу и свободным от побочных эффектов, легко устранивая исключения на ранних стадиях, вместо того чтобы распространять их.

Тип Result в F#

Тип `Result`, представленный в главе 9, — это удобное размеченное объединение F#, которое поддерживает потребляющий код, способный генерировать ошибку без необходимости реализации обработки исключений.

Начиная с F# 4.1, тип `Result` входит в состав стандартной библиотеки F#. Если вы используете более ранние версии F#, то можете легко определить этот тип и его вспомогательные функции буквально в несколько строк:

```
Type Result<'TSuccess, 'TFailure> =
| Success of 'TSuccess
| Failure of 'TFailure
```

Для того чтобы организовать взаимодействие между библиотекой ядра F# и C# для совместного использования одной и той же структуры типа F# `Result` и избежать дублирования кода, необходимы лишь минимальные усилия. Методы расширения F#, которые облегчают взаимодействие с типом `Result` в C#, вы найдете в исходном коде к книге.

В листинге 10.8 приводится пример реализации типа `Result` на C#, адаптированный таким образом, чтобы быть полиморфным только в одном конструкторе типа и принудительно использовать тип `Exception` в качестве альтернативного значения для обработки ошибок. Соответственно, система типов вынуждена распознавать ошибки, логика обработки ошибок становится более явной и предсказуемой. В данном листинге из соображений краткости были опущены некоторые детали реализации, полный текст входит в состав исходного кода к книге.

Листинг 10.8. Параметрический тип Result<T> в C#

```

Свойства для представления
значений, соответствующих
успешным или неудачным
операциям

Конструкторы, передающие значение
успешной операции или исключение
в случае неудачного завершения

=> IsOk ? okMap(Ok) : failureMap(Error);
{ if (IsOk) okAction(Ok); else errorAction(Error);}

Нейвные операторы
автоматически преобразуют
любой примитивный тип в Result

```

```

Эта удобная функция Match деконструирует
тип Result и применяет логику выбора поведения

```

```

struct Result<T>
{
    public T Ok { get; }
    public Exception Error { get; }
    public bool IsFailed { get => Error != null; }
    public bool IsOk => !IsFailed;

    public Result(T ok)
    {
        Ok = ok;
        Error = default(Exception);
    }
    public Result(Exception error)
    {
        Error = error;
        Ok = default(T);
    }

    public R Match<R>(Func<T, R> okMap, Func<Exception, R> failureMap)
        => IsOk ? okMap(Ok) : failureMap(Error);

    public void Match(Action<T> okAction, Action<Exception> errorAction)
        { if (IsOk) okAction(Ok); else errorAction(Error); }

    public static implicit operator Result<T>(T ok) =>
        new Result<T>(ok);
    public static implicit operator Result<T>(Exception error) =>
        new Result<T>(error);

    public static implicit operator Result<T>(Result.Ok<T> ok) =>
        new Result<T>(ok.Value);
    public static implicit operator Result<T>(Result.Failure error) =>
        new Result<T>(error.Error);
}

```

Интересная часть этого кода — последние строки, где благодаря нейвным операторам упрощается преобразование в `Result` во время присваивания примитивов. Такое автоматическое преобразование в тип `Result` должно использоваться любой функцией, которая потенциально может возвращать ошибку.

Вот, например, простая синхронная функция, загружающая содержимое заданного файла в виде байтов. Если файл не существует, то возвращается исключение `FileNotFoundException`:

```

static Result<byte[]> ReadFile(string path)
{
    if (File.Exists(path))
        return File.ReadAllBytes(path);
    else
        return new FileNotFoundException(path);
}

```

Как видим, функция `ReadFile` возвращает `Result<byte[]>`, служащий оберткой либо для результата успешного выполнения функции, который представляет собой байтовый массив, либо, в случае сбоя, для исключения `FileNotFoundException`. Оба возвращаемых типа, `Ok` и `Failure`, неявно преобразуются без объявления типа.

ПРИМЕЧАНИЕ

Назначение класса `Result` аналогично описанному ранее типу `Option`. Тип `Result` позволяет рассуждать о коде, не углубляясь в детали реализации. Это достигается путем предоставления выбора из двух вариантов: `Ok`, который возвращает значение при успешном выполнении функции, и `Failure Error` — для возвращения значения в случае сбоя функции.

10.3. Обработка исключений при асинхронных операциях

Полиморфный класс `Result` в C# — это многократно используемый компонент, который рекомендуется применять для ограничения побочных эффектов функций, способных генерировать исключения. Чтобы указать, что функция может завершиться неудачно, вывод обернут в тип `Result`. В листинге 10.9 показана предыдущая функция `DownloadOptionImage`, измененная в соответствии с моделью типа `Result` (изменения выделены жирным шрифтом). Новая функция называется `DownloadResultImage`.

Важно то, что тип `Result` предоставляет объекту, вызывающему функцию `DownloadResultImage`, информацию, необходимую для выбора соответствующего способа обработки каждого из возможных результатов, включая различные варианты ошибок. В этом примере, поскольку функция `DownloadResultImage` обращается к удаленному сервису, она, кроме эффекта `Result`, также имеет эффект `Task` (для асинхронных операций). В примере с хранилищем Azure (из листинга 10.9) при получении текущего состояния изображения эта операция обращается к онлайн-хранилищу медиаданных. Как я уже отмечал, рекомендуется сделать ее асинхронной и обернуть тип `Result` в `Task`. В функциональном программировании при реализации асинхронных операций с обработкой ошибок эффекты `Task` и `Result` обычно используются совместно.

Прежде чем углубиться в детали совместного применения типов `Result` и `Task`, создадим несколько вспомогательных функций для упрощения кода. Статический класс `ResultExtensions` определяет серию полезных функций более высокого порядка для типа `Result`, таких как `bind` и `map`, применение которых обеспечивает удобную краткую семантику для описания общих потоков обработки ошибок. Из соображений краткости в листинге 10.10 показаны только вспомогательные функции, которые обрабатывают типы `Task` и `Result` (выделены жирным шрифтом). Другие перегруженные функции опущены, их полная реализация содержится в примерах, выложенных на сайте издательства.

Листинг 10.9. Функция DownloadResultImage: обработка ошибок и сохранение семантики

```
async Task<Result<Image>> DownloadResultImage(string blobReference)
{
    try
    {
        CloudStorageAccount storageAccount =
            CloudStorageAccount.Parse("<Azure Connection>");
        CloudBlobClient blobClient =
            storageAccount.CreateCloudBlobClient();
        CloudBlobContainer container =
            blobClient.GetContainerReference("Media");
        await container.CreateIfNotExistsAsync();

        CloudBlockBlob blockBlob = container.
            GetBlockBlobReference(blobReference);
        using (var memStream = new MemoryStream())
        {
            await blockBlob.DownloadToStreamAsync(memStream).
                ConfigureAwait(false);
            return Image.FromStream(memStream); ←
        }
    }
    catch (StorageException exn)
    {
        return exn;
    }
    catch (Exception exn)
    {
        return exn;
    }
}
```

Неявные операторы типа Result позволяют автоматически обертывать примитивные типы в Result, который, в свою очередь, обернут в Task

Листинг 10.10. Вспомогательные функции Task<Result<T>> для семантики компоновки

```
static class ResultExtensions
{
    public static async Task<Result<T>> TryCatch<T>(Func<Task<T>> func)
    {
        try
        {
            return await func();
        }
        catch (Exception ex)
        {
            return ex;
        }
    }

    static async Task<Result<R>> SelectMany<T, R>(this Task<Result<T>>
        resultTask, Func<T, Task<Result<R>>> func)
    {
```

```

    Result<T> result = await resultTask.ConfigureAwait(false);
    if (result.IsFailed)
        return result.Error;
    return await func(result.Ok);
}

static async Task<Result<R>> Select<T, R>(this Task<Result<T>> resultTask,
➥ Func<T, Task<R>> func)
{
    Result<T> result = await resultTask.ConfigureAwait(false);
    if (result.IsFailed)
        return result.Error;
    return await func(result.Ok).ConfigureAwait(false);
}

static async Task<Result<R>> Match<T, R>(this Task<Result<T>> resultTask,
➥ Func<T, Task<R>> actionOk, Func<Exception, Task<R>> actionError)
{
    Result<T> result = await resultTask.ConfigureAwait(false);
    if (result.IsFailed)
        return await actionError(result.Error);
    return await actionOk(result.Ok);
}
}

```

Функция `TryCatch` обертыывает заданную операцию в блок `try-catch`, чтобы в случае возникновения проблем защитить код от любых исключений. Эта функция позволяет объединить любое вычисление `Task` с типом `Result`. В следующем фрагменте кода функция `ToByteArrayAsync` асинхронно преобразует заданное изображение в байтовый массив:

```

Task<Result<byte[]>> ToByteArrayAsync(Image image)
{
    return TryCatch(async () =>
    {
        using (var memStream = new MemoryStream())
        {
            await image.SaveImageAsync(memStream, image.RawFormat);
            return memStream.ToArray();
        }
    });
}

```

Внутренняя функция `TryCatch` гарантирует, что, независимо от поведения внутри операции, будет возвращен тип `Result`, в который будет обернут либо успешный (массив байтов `Ok`), либо неуспешный (исключение `Error`) результат.

Методы расширения `Select` и `SelectMany`, являющиеся частью класса `ResultExtensions`, в функциональном программировании обычно известны как `Bind` (или `flatMap`) и `Map` соответственно. Но в контексте .NET и, в частности, в C#, рекомендуется использовать имена `Select` и `SelectMany`, поскольку они соответствуют

соглашению LINQ; компилятор распознает и обрабатывает эти функции как LINQ-выражения, чтобы упростить семантическую структуру их компоновки. Теперь с помощью операторов более высокого порядка из класса `ResultExtensions` можно легко и изящно построить цепочку действий, которые работают с внутренним значением `Result`, не выходя из контекста.

В листинге 10.11 показано, как объект, вызывающий `DownloadResultImage`, может обрабатывать поток выполнения при успешном или неудачном завершении операции, а также строить последовательность операций (код, на который стоит обратить внимание, выделен жирным шрифтом).

Листинг 10.11. Компоновка операций `Task<Result<T>>` в функциональном стиле

```

Использование HOF для простой компоновки функций
с возвращением составного типа Task<Result<T>>

async Task<Result<byte[]>> ProcessImage(string nameImage, string
destinationImage){
    return await DownloadResultImages(nameImage)
        .Map(async image => await ToThumbnail(image))
        .Bind(async image => await ToByteArrayAsync(image))
        .Tap(async bytes =>
            await File.WriteAllBytesAsync(destinationImage,
                bytes));
```

→ →

Реализация метода расширения WriteAllBytesAsync
содержится в исходном коде,ложенном онлайн

Как видно из сигнатуры функции `ProcessImage`, сообщение в документации о том, что функция может возвращать результаты в виде ошибок, является одним из преимуществ использования типа `Result`. Функция `ProcessImage` сначала загружает заданное изображение из хранилища Azure Blob, затем преобразует его в формат миниатюры с помощью оператора `Bind`. Этот оператор проверяет предыдущий экземпляр `Result` и, если он успешен, выполняет переданный ему делегат. В противном случае оператор `Bind` возвращает предыдущий результат. Оператор `Map` также проверяет предыдущее значение `Result` и действует соответствующим образом, извлекая массив байтов из изображения.

Цепочка операций продолжается до тех пор, пока одна из них не завершится неудачей. Если происходит сбой, то остальные операции пропускаются.

ПРИМЕЧАНИЕ

Функция `Bind` работает с обернутыми значениями, в данном случае это `Task<Result<Image>>`. В отличие от нее функция `Map` работает с развернутыми типами.

Полученный в итоге байтовый массив сохраняется по указанному пути (`destinationImage`), или же, в случае возникновения ошибки, производится запись в журнал. Вместо того чтобы обрабатывать ошибки для каждого вызова в отдельности,

следует добавить обработку ошибок в конце цепочки вычислений. Таким образом логика обработки ошибок размещается в предсказуемом месте кода, что облегчает чтение и обслуживание.

Вы должны понимать, что в случае неудачного завершения любой из этих операций остальные задачи будут пропущены. Ни одна из них не будет выполнена до первой функции, обрабатывающей ошибку (рис. 10.5). В данном примере ошибка обрабатывается функцией `Match` (с лямбда-выражением `actionError`). Если вызов функции не завершился успешно, важно выполнить логику компенсации.

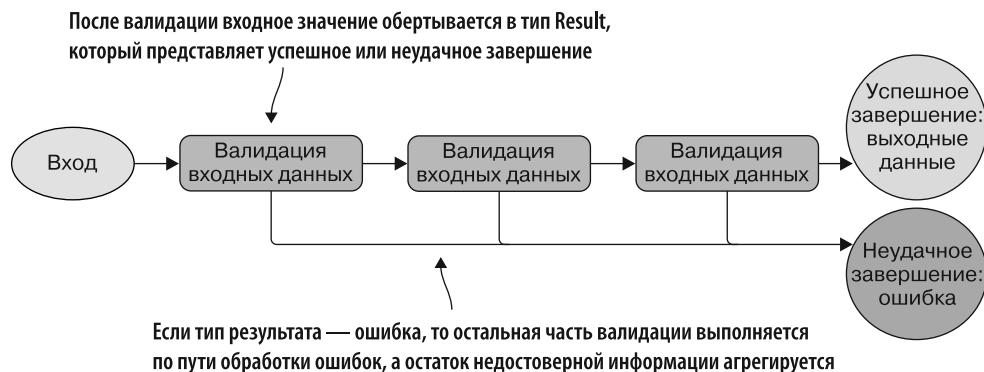


Рис. 10.5. Тип `Result` обрабатывает операции таким образом, что если на каком-либо шаге произойдет сбой, то остальная последовательность задач будет пропущена, вплоть до первой функции, обрабатывающей ошибку. На данном рисунке, если какая-либо из валидаций выдает ошибку, остальная часть вычислений пропускается до обработчика `Failure`

Поскольку извлекать внутреннее значение из типа `Result` сложно и неудобно, мы воспользуемся механизмами компоновки при функциональной обработке ошибок. Эти механизмы вынуждают вызывающий объект всегда обрабатывать результат — как успешный, так и неудачный. Применяя это свойство типа `Result`, можно сделать программный поток декларативным и простым для понимания. Раскрытие намерений имеет решающее значение, если вы хотите повысить читабельность кода. Представление класса `Result` (и скомпонованного типа `Task<Result<T>`) без побочных эффектов помогает показать, может ли метод завершиться неудачно или же не будет сигнализировать системе о том, что что-то пошло не так. Вообще, система типов становится полезным помощником в разработке программного обеспечения, определяя, как следует обрабатывать успешные, и неудачные результаты.

Тип `Result` обеспечивает условный поток, представленный в функциональном стиле высокого уровня. Здесь вы сами выбираете стратегию для устранения ошибки и регистрируете данную стратегию как обработчик. Когда код нижнего уровня обнаруживает ошибку, он может выбрать обработчик, не нагружая стек вызовов. Это открывает дополнительные возможности: можно решить проблему и продолжить работу.

10.3.1. Моделирование обработки ошибок в F# с помощью Async и Result

В предыдущем разделе обсуждалась концепция комбинирования типов `Task` и `Result` для обеспечения безопасной и декларативной обработки ошибок в функциональном стиле. Кроме TPL, вычислительные выражения асинхронного рабочего процесса в F# обеспечивают более естественный функциональный подход. В этом разделе дается рецепт обработки исключений, в котором показано, как можно комбинировать тип `Async` F# со структурой `Result`.

Прежде чем углубиться в подробности модели обработки ошибок в F# для асинхронных операций, следует определить необходимую структуру типов. Во-первых, чтобы соответствовать контексту обработки ошибок (в частности), как описано в главе 9, следует определить псевдоним типа `Result<'a>` для `Result<'a, exn>`, который предполагает, что второе значение всегда является исключением (`exn`). Псевдоним `Result<'a>` упрощает сопоставление с образцом и деконструирование по типу `Result<'a, exn>`:

```
Result<'TSuccess> = Result<'TSuccess, exn>
```

Во-вторых, конструкция типа `Async` должна обернуть эту структуру в `Result<'a>` — так мы определим новый тип, который будет использоваться в параллельных операциях и сигнализировать о том, что операция завершена. Типы `AsyncResult<'a>` и `Result<'a>` следует рассматривать как один тип. Это легко, если применять типы псевдонимов, которые действуют как комбинаторная структура:

```
typeAsyncResult<'a> = Async<Result<'a>>
```

Тип `AsyncResult<'a>` содержит значение асинхронного вычисления, успешного или неудачного. В случае возникновения исключения информация об ошибке сохраняется. Концептуально `AsyncResult` — это отдельный тип.

Теперь, черпая вдохновение из типа `AsyncOption`, описанного в подразделе 10.2.2, определим вспомогательную функцию `AsyncResult.handler`, которая обертывает выходные данные в тип `Result` и выполняет вычисления. Для этого функция F# `Async.Catch` описывает идеальное соответствие. В листинге 10.12 показано специальное альтернативное представление `AsyncResult.Catch`, которое называется `AsyncResult.handler`.

Обработчик `AsyncResult.handler` в F# — мощный оператор, который перенаправляет поток выполнения в случае ошибки. В двух словах, `AsyncResult.handler` запускает в фоновом режиме функцию `Async.Catch` для обработки ошибок и использует функцию `ofChoice` для сопоставления результата вычислений (размеченного объединения `Choice<Choice1Of2, Choice2Of2>`) с вариантами, представленными в размеченном объединении `Result<'a>`, в зависимости от которых затем результат вычисления принимает значение `OK` или `Error` соответственно (`ofChoice` был описан в главе 9).

Листинг 10.12. ОбработчикAsyncResult для захвата и обертывания асинхронных вычислений

```
module Result =
    let ofChoice value =
        match value with
        | Choice1Of2 value -> Ok value
        | Choice2Of2 e -> Error e

    let ofResult result =
        match result with
        | Ok value -> Ok value
        | Error e -> Error e

module AsyncResult =
    let handler (operation:Async<'a>) : AsyncResult<'a> = async {
        let! result = Async.Catch operation
        return (Result.ofChoice result) }
```

Сопоставление функции из размеченного объединения Choice, которое является возвращаемым типом оператора Async.Catch, с вариантами размеченного объединения типа Result. Эта функция отображения была определена в главе 9

Запуск асинхронной операции с использованием оператора Async.Catch для защиты от возможных ошибок

Выходные данные функции сопоставляются в соответствии с типом Result

10.3.2. Расширение типа F# AsyncResult посредством операторов монадического связывания

Прежде чем двигаться дальше, мы определим монадические вспомогательные функции для работы с типом AsyncResult (листинг 10.13).

Листинг 10.13. Функции высшего порядка, расширяющие тип AsyncResult

```
module AsyncResult =
    let retn (value:'a) : AsyncResult<'a> = value |> Ok |> async.Return
    let map (selector : 'a -> Async<'b>) (asyncResult : AsyncResult<'a>)
        = async {
            let! result = asyncResult
            let! result = selector result
            return! result }
    let bind (selector : 'a -> AsyncResult<'b>) (asyncResult : AsyncResult<'a>)
        = async {
            let! result = asyncResult
            let! result = selector result
            return! result }
    let bimap success failure operation = async {
        let! result = operation
        match result with
        | Ok v -> return! success v |> handler
        | Error x -> return! failure x |> handler }
```

Обертывание произвольного заданного значения в тип AsyncResult

Сопоставление типа AsyncResult, выполняющего внутреннюю асинхронную операцию asyncResult, и применение заданной функции селектора к результату

Связывание монадического оператора, выполняющего заданную функцию, с надтипом AsyncResult

Выполнение функции, соответствующей успеху или неудаче, для операции типа AsyncResult путем выбора для результата вычислений соответствующей ветви размеченного объединения — Ok или Error соответственно

Использование функции AsyncResult.handler для обработки успешного или неудачного результата асинхронной операции

Операторы высшего порядка `map` и `bind` являются общими функциями, используемыми для компоновки.

Следующие реализации очевидны.

- ❑ Функция `retn` обертывает произвольное значение '`a` в надтип `AsyncResult<'a>`.
- ❑ Синтаксическая конструкция `let!` в операторе `map` извлекает содержимое `Async`, выполняет его и ожидает результата, который имеет тип `Result<'a>`. Затем к значению `Result`, содержащемуся в случае `Ok`, применяется функция `selector`, используя `AsyncResult.handler`, поскольку результат вычисления может быть как успешным, так и неудачным. В итоге возвращается результат, обернутый в тип `AsyncResult`.
- ❑ Функция `bind` применяет стиль продолжений (Continuation Passing Style, CPS), чтобы передать функцию, которая будет выполнять успешное вычисление для дальнейшей обработки результата. Функция продолжения `selector` совмещает два типа, `Async` и `Result`, и имеет сигнатуру `'a -> AsyncResult<'b>`.
- ❑ Если внутренний `Result` является успешным, то для него выполняется функция продолжения `selector`. Синтаксис `return!` означает, что возвращаемое значение уже обернуто.
- ❑ Если внутренний `Result` сообщает о сбое, возвращается сообщение о неудачном завершении асинхронной операции.
- ❑ Синтаксис возврата в операциях `map`, `retn` и `bind` обертывает значение `Result` в тип `Async`.
- ❑ В операции `bind` синтаксис `return!` означает, что значение уже обернуто и для него не надо вызывать операцию `return`.
- ❑ Целью функции `bimap` является выполнение асинхронной операции `AsyncResult` и последующее ветвление потока выполнения, который, в зависимости от результата, выполняет одну из функций продолжения: `success` либо `failure`.

Чтобы сделать код более лаконичным, вместо этого можно использовать встроенную функцию `Result.map`, позволяющую превратить значение в функцию, которая работает с типом `Result`. Затем, если передать выходные данные в `Async.map`, результирующая функция будет работать с асинхронным значением. Применяя этот стиль компоновочного программирования, можно переписать функцию `AsyncResult map`, например, следующим образом:

```
module AsyncResult =  
    let map (selector : 'a -> 'b) (asyncResult : AsyncResult<'a>) =  
        asyncResult |> Async.map (Result.map selector)
```

Использовать ли такой стиль программирования — ваш личный выбор, поэтому следует рассмотреть компромисс между сжатостью и удобством чтения кода.

Функция более высокого порядка F# AsyncResult в действии

Посмотрим, как выполнить тип `AsyncResult` и его функции более высокого порядка: `bind`, `map` и `return`. Преобразуем код C# из листинга 10.7, который загружает изображение из хранилища Azure Blob, в более свойственный F# код с обработкой ошибок в контексте асинхронной операции.

Мы продолжаем рассматривать пример с хранилищем Azure Blob, чтобы упростить понимание двух подходов путем прямого сравнения, преобразуя уже знакомые вам функции (рис. 10.6).

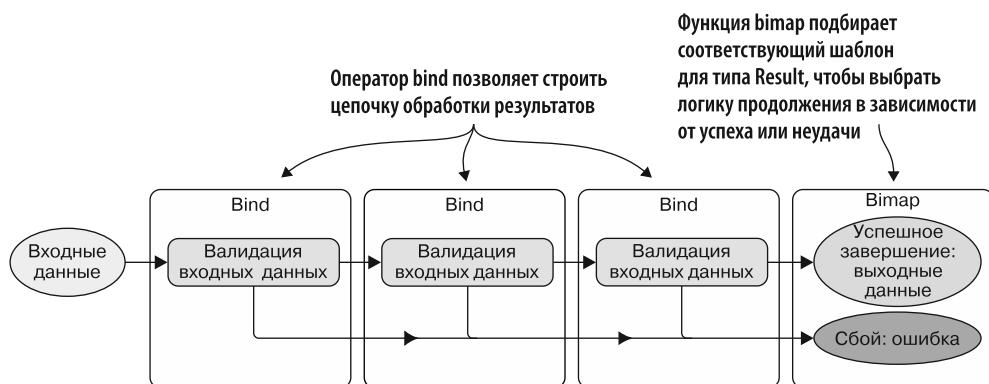


Рис. 10.6. Логика валидации может быть скомпонована быстро и с минимальными усилиями путем применения операторов более высокого порядка `bind` и `bimap`. Кроме того, в конце конвейера функция `bimap` выбирает шаблон для типа `Result`, чтобы перенаправить логику продолжения либо по ветви `success`, либо по ветви `failure` в удобном и декларативном стиле

В листинге 10.14 представлена функция `processImage`, реализованная с использованием типа F# `AsyncResult` и его компоновочных операторов более высокого порядка (выделены жирным шрифтом).

Листинг 10.14. Использование функций более высокого порядка `AsyncResult` для быстрой компоновки

```
let processImage(blobReference:string) (destinationImage:string)
  : AsyncResult<unit> =
  async {
    let storageAccount = CloudStorageAccount.Parse("< Azure Connection >")
    let blobClient = storageAccount.CreateCloudBlobClient()
    let container = blobClient.GetContainerReference("Media")
    let! _ = container.CreateIfNotExistsAsync()
    let blockBlob = container.GetBlockBlobReference(blobReference)
    use memStream = new MemoryStream()
    do! blockBlob.DownloadToStreamAsync(memStream)
```

```

    return Bitmap.FromStream(memStream) }
|> AsyncResult.handler
|> AsyncResult.bind(fun image -> toThumbnail(image))
|> AsyncResult.map(fun image -> toByteArrayAsync(image))
|> AsyncResult.bitmap
    (fun bytes -> FileEx.
        ↪ WriteAllBytesAsync(destinationImage, bytes))
    (fun ex -> logger.Error(ex) |> AsyncResult.ret)

```

Стиль компактной компоновки оператора
более высокого порядка AsyncResult

AsyncResult.ret обертывает функцию logger
в надтип AsyncResult, чтобы соответствовать
сигнатурае выходных данных

Поведение функции `processImage` похоже на связанный с C# метод `processImage` из листинга 10.7; единственным отличием является определение типа `AsyncResult`. Семантика благодаря встроенному конвейерному оператору F# (`|>`) функции `AsyncResult handler`, `bind`, `map` и `bitmap` объединяются в компактную цепочку. Этот стиль — ближайший эквивалент концепции текущих интерфейсов (или цепочек методов), используемых в версии кода на C#.

Повышение уровня абстракции AsyncResult на F# с помощью вычислительных выражений

Предположим, что мы хотим повысить абстракцию синтаксиса по сравнению с кодом, представленным в листинге 10.12, чтобы вычисления `AsyncResult` можно было объединять в цепочки и компоновать, используя конструкции потока управления. В главе 9 мы создали собственные *вычислительные выражения* (Computational Expressions, CE) F# для повторного выполнения асинхронных операций в случае ошибок. В F# CE являются безопасным способом управления сложностью и изменением состояний. Они предоставляют удобный синтаксис для управления данными, потоком и побочными эффектами в функциональных программах.

В контексте асинхронных операций, обернутых в тип `AsyncResult`, можно использовать CE для элегантной обработки ошибок, что позволяет сосредоточиться на «счастливом пути». Благодаря монадическим операторам `AsyncResult bind` и `return` реализация связанных вычислительных выражений требует минимальных усилий для получения удобной и компактной семантики программирования.

В следующем коде определяются монадические операторы (выделены жирным шрифтом) для компоновщика вычислений, в которых объединяются типы `Result` и `Async`:

```

Type AsyncResultBuilder () =
    Member x.Return m = AsyncResult.ret m
    member x.Bind (m, f:'a -> AsyncResult<'b>) = AsyncResult.bind f m
    member x.Bind (m:Task<'a>, f:'a -> AsyncResult<'b>) =
        AsyncResult.bind f (m |> Async.AwaitTask |> AsyncResult.handler)
    Ember x.ReturnFrom m = m

Let asyncResult = AsyncResultBuilder()

```

Если необходима поддержка более сложного синтаксиса, то можно добавить и другие функции-члены `AsyncResultBuilder` СЕ; здесь представлена лишь минимальная реализация, необходимая для данного примера. Единственная строка кода, требующая пояснений, — это тип `Bind` с `Task<'a>`:

```
member x.Bind (m:Task<'a>, f) =
    AsyncResult.bind f (m |> Async.AwaitTask
    ↪ |> AsyncResult.handler)
```

В этом случае, как объяснялось в подразделе 9.3.3, вычислительные выражения F# позволяют вводить функции, распространяющие возможности манипулирования на другие обертывающие типы, в данном случае на `Task`, тогда как функция `Bind` в расширении позволяет выбирать внутреннее значение, содержащееся в надтипе, с использованием операторов `let!` и `do!`. Эта технология устраниет потребность в дополнительных функциях, таких как `Async.AwaitTask`. В исходном коде к книге вы найдете более полную реализацию СЕ `AsyncResultBuilder`, но дополнительные подробности реализации СЕ не имеют отношения к теме этой книги.

Простое СЕ оперирует асинхронными вызовами, которые возвращают тип `Result` и могут быть полезны для выполнения вычислений, способных завершиться неудачей. Затем эти вызовы позволяют объединить результаты в цепочку. Давайте снова преобразуем функцию `processImage`, но на сей раз вычисления будут выполняться внутри СЕ `AsyncResultBuilder` (в листинге 10.15 выделены жирным шрифтом).

Листинг 10.15. Использование `AsyncResultBuilder`

```
let processImage (blobReference:string) (destinationImage:string)
    : AsyncResult<unit> =
    asyncResult {
        let storageAccount = CloudStorageAccount.Parse("<Azure Connection>")
        let blobClient = storageAccount.CreateCloudBlobClient()
        let container = blobClient.GetContainerReference("Media")
        let! _ = container.CreateIfNotExistsAsync()
        let blockBlob = container.GetBlockBlobReference(blobReference)
        use memStream = new MemoryStream()
        do! blockBlob.DownloadToStreamAsync(memStream)
        let image = Bitmap.FromStream(memStream)
        let! thumbnail = toThumbnail(image)
        return! toByteArrayAsyncResult thumbnail
    }
    |> AsyncResult.bitmap (fun bytes ->
    FileEx.WriteAllBytesAsync(destinationImage, bytes))
    (fun ex -> logger.Error(ex) |> async.Return.retn)
```

Блок кода обернут в `asyncResult`, чтобы оператор `bind` выполнялся в контексте вычислительных выражений `AsyncResultBuilder`

Теперь осталось лишь обернуть операции в блок СЕ `asyncResult`. Компилятор распознает монадический (СЕ) шаблон и обрабатывает вычисления соответствующим образом. Встретив оператор связывания `let!`, компилятор автоматически преобразует операции вычислительных выражений `AsyncResult.Return` и `AsyncResult.Bind` в заданном контексте.

10.4. Абстрагирование операций с функциональными комбинаторами

Предположим, что нам нужно загрузить и проанализировать историю биржевого кода или историю изменений курсов нескольких акций, чтобы сравнить их и выбрать лучшие варианты для покупки. Считается, что загрузка данных из Интернета является операцией с ограничениями ввода/вывода, которая должна выполняться асинхронно. Но предположим, что мы хотим создать более сложную программу, в которой загрузка биржевых данных зависит от других асинхронных операций (рис. 10.7), например:

- ❑ если индекс NASDAQ или NYSE положительный;
- ❑ если последние шесть месяцев акции имели положительную динамику;
- ❑ если пакет акций удовлетворяет какому-то количеству положительных критериев покупки.

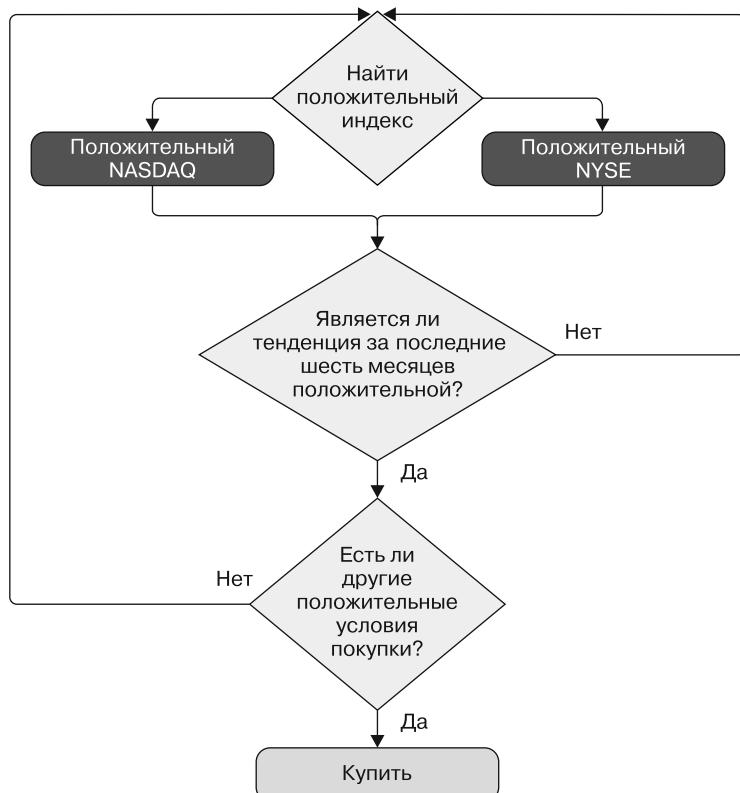


Рис. 10.7. Эта схема представляет собой последовательное дерево принятия решений при покупке акций. Каждый шаг, вероятно, включает в себя операцию ввода-вывода для асинхронного опроса внешнего сервиса. Необходимо быть внимательными, чтобы поддерживать этот последовательный поток, асинхронно проходя по всему дереву принятия решений

А что, если выполнить поток, показанный на рис. 10.7, для каждого интересующего вас биржевого кода? Как бы вы объединили условную логику этих операций, сохраняя асинхронную семантику, чтобы распараллелить выполнение? Как бы вы разработали программу?

Ответ на этот вопрос — *функциональные асинхронные комбинаторы*. В следующих разделах будут рассмотрены характеристики функциональных комбинаторов, причем особое внимание будет уделено комбинаторам асинхронным. Я покажу, как использовать встроенную поддержку .NET Framework, а также как создавать и настраивать ваши собственные асинхронные комбинаторы, чтобы добиться максимальной производительности программы, применяя гибкий и декларативный стиль функционального программирования.

10.5. Коротко о функциональных комбинаторах

Императивная парадигма использует для управления потоком выполнения программы процедурные механизмы, такие как операторы *if-else* и циклы *for/while*. Это противоречит стилю функционального программирования. Покинув императивный мир, вы научитесь находить альтернативы, позволяющие восполнить данный пробел. Хорошим решением является использование функциональных комбинаторов, которые управляют потоком выполнения программы. Механизмы функционального программирования упрощают объединение нескольких решений более мелких задач в единую абстракцию, решающую более крупную проблему.

Абстракция — это основа функционального программирования, позволяющая разрабатывать приложения, не заботясь о деталях реализации, давая сконцентрироваться на более важной семантике программы высокого уровня. Абстракция фиксирует суть того, что делает отдельная функция или вся программа, облегчая выполнение задач.

В функциональном программировании *комбинатором* называется либо функция без свободных переменных (<https://wiki.haskell.org/Pointfree>), либо шаблон для компоновки и объединения любых типов. Это второе определение и будет центральной темой данного раздела.

С практической точки зрения *функциональные комбинаторы* — это программные конструкции, которые позволяют объединять и связывать примитивные артефакты, такие как другие функции (или другие комбинаторы), и вести себя как объединенные части управляющей логики, генерируя расширенные варианты поведения. Кроме того, функциональные комбинаторы способствуют модульности, помогающей абстрагировать функции в компоненты, которые можно рассматривать и повторно использовать независимо, с кодифицированным значением, следующим из правил, управляющих их компоновкой. Эта концепция уже встречалась нам, когда мы знакомились с определением асинхронных функций (комбинаторов), таких как *Otherwise* и *Retry* для модели асинхронного программирования на основе задач в C# (Task-based Asynchronous Programming, TAP) и *AsyncResult.handler* в F#.

В контексте конкурентного программирования основной причиной использования комбинаторов является возможность реализации программы, которая обрабатывает побочные эффекты без ущерба для декларативной и компоновочной семантики. Это возможно благодаря тому, что комбинаторы абстрагируются от деталей реализации разработки и позволяют скрыть обработку побочных эффектов с целью предложить функции, которые легко компоновать. В частности, в данном разделе рассматриваются комбинаторы, позволяющие компоновать асинхронные операции.

Если побочные эффекты ограничены областью видимости одной функции, то поведение, вызывающее эту функцию, является идемпотентным. *Идемпотентность* означает, что операция может применяться многократно без изменения результата за пределами первоначального приложения — эффект не изменяется. Такие идемпотентные функции можно объединять в цепочки, описывающие сложное поведение, где побочные эффекты изолированы и находятся под контролем.

10.5.1. Встроенные асинхронные комбинаторы TPL

Асинхронный рабочий процесс F# и .NET TPL предоставляют разработчику набор встроенных комбинаторов, таких как `Task.Run`, `Async.StartWithContinuation`, `Task.WhenAll` и `Task.WhenAny`. Их можно легко расширить для реализации полезных комбинаторов, чтобы компоновать и строить более сложные шаблоны на основе задач. Например, такие операторы F#, как `Task.WhenAll` и `Async.Parallel`, используются для асинхронного ожидания нескольких асинхронных операций; затем внутренние результаты данных операций группируются для продолжения. Это продолжение — ключевая особенность, которая предоставляет возможности для компоновки потока программы в более сложные структуры, такие как реализация шаблонов `Fork/Join` и «разделяй и властвуй».

Чтобы оценить преимущества комбинаторов, начнем с простого примера на C#. Предположим, что нам нужно выполнить три асинхронные операции и вычислить сумму их результатов, дожидаясь получения каждого из них по очереди. Обратите внимание, что выполнение каждой операции занимает одну секунду:

```
async Task<int> A() { await Task.Delay(1000); return 1; }
async Task<int> B() { await Task.Delay(1000); return 3; }
async Task<int> C() { await Task.Delay(1000); return 5; }

int a = await A();
int b = await B();
int c = await C();

int result = a + b + c;
```

Результат (9) вычисляется за 3 с, по 1 с на каждую операцию. А что будет, если выполнить эти три метода параллельно? Существуют методы, помогающие координировать выполнение нескольких задач в фоновом режиме. Самым простым

решением для конкурентного выполнения нескольких задач будет запустить их последовательно, а затем собрать ссылки на них. TPL-оператор `Task.WhenAll` принимает массив задач `params` и возвращает задачу, в которой сообщается, когда будут завершены все остальные задачи. Если исключить из последнего примера промежуточные переменные, то получим более компактный код:

```
var results = (await Task.WhenAll(A(), B(), C())).Sum();
```

Результаты возвращаются в массив, к которому затем применяется LINQ-оператор `Sum()`. С этими изменениями результат вычисляется всего за одну секунду. Теперь задача представляет собой полностью асинхронную операцию и предоставляет синхронные и асинхронные возможности для объединения с операцией, принимающей ее результаты, и т. д. Это позволяет создавать полезные библиотеки комбинаторов, с помощью которых можно компоновать задачи и строить большие шаблоны.

10.5.2. Использование комбинатора `Task.WhenAny` для избыточности и чередования

Преимущество использования задач заключается в том, что они сильно расширяют возможности компоновки. Как только у нас появляется тип, способный представлять произвольную асинхронную операцию, мы можем писать комбинаторы для этого типа, которые позволяют комбинировать и компоновать асинхронные операции множеством способов.

Например, TPL-оператор `Task.WhenAny` позволяет разрабатывать параллельные программы, в которых одна задача, состоящая из нескольких асинхронных операций, должна быть выполнена прежде, чем основной поток сможет продолжить обработку. Такое поведение асинхронного ожидания завершения первой операции, обрабатывающей заданный набор задач, и последующего уведомления основного потока о дальнейшей обработке упрощает разработку сложных комбинаторов. Примерами свойств, которые следуют из этих комбинаторов, являются избыточность, чередование и ограничение.

ИЗБЫТОЧНОСТЬ

Многократное выполнение асинхронной операции и выбор того варианта, который завершился первым.

ЧЕРЕДОВАНИЕ

Запускается несколько операций, но они обрабатываются в порядке завершения. Это свойство обсуждалось в подразделе 8.5.7.

Рассмотрим пример: вы хотите как можно скорее купить билет на самолет. Вам нужно связаться с несколькими веб-сервисами авиакомпаний, но в зависимости от веб-трафика у каждого сервиса может быть свое время отклика. В этом случае для связи

с несколькими веб-сервисами можно использовать оператор `Task.WhenAny`, выдающий один результат — тот, который будет получен раньше всех (листинг 10.16).

Листинг 10.16. Избыточность с помощью оператора `Task.WhenAny`

```

Использование CancellationToken для отмены
операций, выполнение которых продолжается
после завершения первой операции
    var cts = new CancellationTokenSource(); ←
    Func<string, string, string, CancellationToken, Task<string>>
    → GetBestFlightAsync = async (from, to, carrier, token) => {
        string url = $"flight provider{carrier}";
        using(var client = new HttpClient()) {
            HttpResponseMessage response = await client.GetAsync(url, token);
            return await response.Content.ReadAsStringAsync();
        };
    };

    var recommendationFlights = new List<Task<string>>() ← Список асинхронных операций
    { ← для параллельного выполнения
        GetBestFlightAsync("WAS", "SF", "United", cts.Token),
        GetBestFlightAsync("WAS", "SF", "Delta", cts.Token),
        GetBestFlightAsync("WAS", "SF", "AirFrance", cts.Token),
    };

    Task<string> recommendationFlight = await Task. ← Оператор Task.WhenAny ожидает
        WhenAny(recommendationFlights); ← завершения первой операции
    while (recommendationFlights.Count > 0)
    {
        try
        {
            var recommendedFlight = await recommendationFlight; ← Получение результата в блоке try-catch,
            cts.Cancel(); ← чтобы учесть возможные исключения.
            BuyFlightTicket("WAS", "SF", recommendedFlight); ← Даже если первая задача завершится
            break; ← успешно, последующие задачи
        } ← могут закончиться неудачей
        catch (WebException)
        {
            recommendationFlights.Remove(recommendedFlight); ←
        }
    }

Если операция выполнена успешно, то остальные
вычисления, которые все еще выполняются, отменяются

```

В этом коде оператор `Task.WhenAny` возвращает задачу, которая была завершена первой. Важно знать, успешно ли завершилась операция, чтобы в случае ошибки отбросить результат и дождаться завершения следующего вычисления. Код должен обрабатывать исключения с использованием блока `try-catch`, где неудачное вычисление удаляется из списка рекомендованных асинхронных операций. Когда появится первая успешно завершенная задача, необходимо гарантировать, что остальные, все еще работающие операции будут отменены.

10.5.3. Использование комбинатора Task.WhenAll в асинхронном цикле for-each

Оператор `Task.WhenAll` асинхронно ожидает выполнения нескольких асинхронных вычислений, представленных в виде задач. Предположим, что мы хотим разослать по электронной почте сообщения всем, кто есть в нашем списке контактов. Чтобы ускорить этот процесс, желательно отправить электронную почту всем получателям параллельно, не дожидаясь завершения отправки каждого сообщения перед отправкой следующего. При таком сценарии было бы удобно обрабатывать список электронных писем в цикле `for-each`. Как обеспечить асинхронную семантику операции при параллельной рассылке электронных писем? Решением является реализация оператора `ForEachAsync` на основе метода `Task.WhenAll` (листинг 10.17).

Листинг 10.17. Асинхронный цикл `for-each` с `Task.WhenAll`

```
static Task ForEachAsync<T>(this IEnumerable<T> source,
➥ int maxDegreeOfParallelism, Func<T, Task> body)
{
    return Task.WhenAll(
        from partition in Partitioner.Create(source).
        GetPartitions(maxDegreeOfParallelism)
        select Task.Run(async () =>
    {
        using (partition)
            while (partition.MoveNext())
                await body(partition.Current);
    }));
}
```

Для каждого раздела перечисления оператор `ForEachAsync` выполняет функцию, которая возвращает объект `Task`, представляющий собой завершение обработки данной группы элементов. Поскольку работа начинается асинхронно, можно обеспечить конкурентность и параллелизм, вызывая тело для каждого элемента и ожидая в конце завершения выполнения всех элементов, а не каждого по очереди.

Созданный `Partitioner` ограничивает количество операций, которые могут выполняться параллельно, чтобы избежать создания большего количества задач, чем необходимо. Это максимальное значение уровня параллелизма зависит от разделения входных данных и определяется количеством фрагментов `maxDegreeOfParallelism`. Начало выполнения каждой задачи планируется для каждого раздела. Пакеты `ForEachAsync` помогают создавать меньше задач, чем существующее количество рабочих элементов. Это обеспечивает значительно лучшую общую производительность, особенно если в теле цикла выполняется небольшое количество операций для каждого элемента.

ПРИМЕЧАНИЕ

Последний пример по своей природе подобен `Parallel.ForEach`. Основное различие заключается в том, что `Parallel.ForEach` является синхронным методом и в нем используются синхронные делегаты.

Теперь можно применять оператор `ForEachAsync` для асинхронной отправки нескольких электронных писем (листинг 10.18).

Листинг 10.18. Использование асинхронного цикла `for-each`

```
async Task SendEmailsAsync(List<string> emails)
{
    SmtpClient client = new SmtpClient();
    Func<string, Task> sendEmailAsync = async emailTo =>
    {
        MailMessage message = new MailMessage("me@me.com", emailTo);
        await client.SendMailAsync(message);
    };
    await emails.ForEachAsync(Environment.ProcessorCount, sendEmailAsync);
}
```

Мы рассмотрели несколько простых примеров, которые показывают, как использовать встроенные TPL-комбинаторы `Task.WhenAll` и `Task.WhenAny`. Раздел 10.6 будет посвящен созданию собственных комбинаторов и компоновке уже существующих, в которых применяются принципы F# и C#. Вы увидите, что есть бесконечное множество комбинаторов. Мы рассмотрим несколько наиболее распространенных, которые используются для реализации асинхронного логического потока в программах — `ifAsync`, асинхронного AND и асинхронного OR.

Прежде чем приступить к созданию асинхронных комбинаторов, рассмотрим функциональные шаблоны, которые обсуждались ранее. Это напоминание приведет нас к новому функциональному шаблону, используемому для компоновки гетерогенных конкурентных функций. Не стоит беспокоиться, если вы не знакомы с этим термином: скоро мы восполним данный пробел.

10.5.4. Обзор математических шаблонов: что мы уже знаем?

В предыдущих главах были представлены концепции моноидов, монад и функторов, которые относятся к разделу математики под названием «*теория категорий*». Кроме того, мы обсудили их важное значение для функционального программирования и функциональной конкурентности.

Лексикон теории категорий

Теория категорий — это раздел математики, определяющий любую коллекцию объектов, которые могут соотноситься друг с другом посредством морфизмов любыми разумными способами, такими как компоновка и ассоциативность. *Морфизмы* — это модное слово, которое определяет то, что может изменяться; представьте себе применение функции `map` (или `select`) для преобразования одной математической структуры в другую. По сути, теория категорий состоит из объектов и стрелок, которые связывают эти объекты между собой, обеспечивая основы для компоновки. Теория категорий — мощная

идея, порожденная необходимостью систематизировать математические понятия, основанные на общей структуре. Под «зонтик» теории категорий подпадают многие полезные концепции. Вам не понадобится математическое образование, чтобы понять и использовать их обширные возможности, которые в большинстве своем касаются создания новых способов компоновки.

В программировании данные математические шаблоны применяются для управления побочными эффектами и сохранения функциональной чистоты. Эти шаблоны представляют интерес благодаря свойствам абстракции и компонуемости. Абстракция способствует возможности компоновки, а сочетание этих свойств служит опорой для функционального и конкурентного программирования. В следующих разделах мы пересмотрим определения этих математических понятий.

Использование моноидов для распараллеливания данных

Моноид, как объяснялось ранее, является бинарной ассоциативной операцией с тождеством; он помогает объединять значения одного типа. Свойство ассоциативности позволяет без особых усилий выполнять вычисления параллельно, предоставляя возможность разделять задачу на куски и вычислять их независимо. Затем, когда все блоки вычислений будут завершены, их результаты объединяются. Оказывается, множество интересных параллельных операций обладают свойствами ассоциативности и коммутативности, будучи выраженным посредством моноидов: `Map-Reduce` и `Aggregation` в различных формах, таких как `sum`, `variance`, `average`, `concatenation`, и т. п. Например, .NET PLINQ для корректного распараллеливания работы использует моноидальные операции, которые являются ассоциативными и коммутативными.

В следующем примере кода, основанном на материале главы 4, показано, как применять PLINQ для распараллеливания суммирования степени сегмента массива. Набор данных разбивается на подмассивы, которые накапливаются отдельно, каждый в своем потоке, с использованием аккумулятора, инициализированного для начального числа. В итоге все аккумуляторы будут объединены с использованием функции конечного уменьшения (функция `AsParallel`, выделена жирным шрифтом):

```
var random = new Random();
var size = 1024 * Environment.ProcessorCount;
int[] array = Enumerable.Range(0, size).Select(_ =>
    random.Next(0, size)).ToArray();

long parallelSumOfSquares = array.AsParallel()
    .Aggregate(
        seed: 0,                                     ←———— Начальное значение для каждого раздела
        updateAccumulatorFunc: (partition, value) =>
    partition + (int)Math.Pow(value, 2),
        combineAccumulatorsFunc: (partitions, partition) =>
    partitions + partition,
        resultSelector: result => result);
```

Несмотря на непредсказуемую последовательность вычислений по сравнению с последовательной версией кода, результат является детерминированным благодаря свойствам ассоциативности и коммутативности оператора +.

Функторы для отображения надтипов

Функтор — это шаблон отображения структур надтипов, который архивируется и предоставляет поддержку двухпараметрической функции, называемой `Map` (также известной как `fmap`). Согласно сигнатуре типа, функция `Map` принимает в качестве первого аргумента функцию ($T \rightarrow R$), которая в C# переводится в `Func<T, R>`.

Функтор применяет преобразование к заданному входному типу T и возвращает тип R . Функтор обрабатывает функции только с одним входным аргументом.

LINQ/PLINQ-оператор `Select` можно считать функтором для надтипа `IEnumerable`. Как правило, функторы используются в C# для реализации гибких API в стиле LINQ, которые применяются для типов, отличных от коллекций. В главе 7 был реализован функтор для надтипа `Task` (функция `Map` выделена жирным шрифтом):

```
static Task<R> Map<T, R>(this Task<T> input, Func<T, R> map) =>
    input.ContinueWith(t => map(t.Result));
```

Функция `Map` принимает функцию отображения ($T \rightarrow R$) и функтор (обернутый в контекст) `Task<T>` и возвращает новый функтор `Task<R>`, содержащий результат применения функции к значению, снова заключенный в замыкание.

Следующий код из главы 8 загружает пиктограмму с заданного сайта и преобразует ее в растровое изображение. Оператор `Map` применяется для построения цепочки асинхронных вычислений (код, на который следует обратить внимание, выделен жирным шрифтом):

```
Bitmap icon = await new HttpClient()
    .GetAsync($"http://{domain}/favicon.ico")
    .Bind(async content => await
        content.Content.ReadAsByteArrayAsync())
    .Map(bytes =>
        Bitmap.FromStream(new MemoryStream(bytes)));
```

Эта функция имеет сигнатуру ($T \rightarrow R$) \rightarrow `Task <T>` \rightarrow `Task <R>`, которая означает, что в качестве первого аргумента она принимает функцию отображения $T \rightarrow R$, преобразующего значение типа T в значение типа R , затем обновляет тип `Task<T>`, предоставляемый в качестве второго входного аргумента, и возвращает `Task<R>`.

Функтор — это не что иное, как структура данных, которую можно использовать для отображения функций с целью обернуть значения в надтип, изменить их и затем поместить обратно в обертку. Причина, по которой `fmap` возвращает тот же надтип, состоит в том, чтобы сохранить возможность продолжения цепочки операций. По сути, функторы создают контекст или абстракцию, позволяющую безопасно манипулировать исходными значениями и применять к ним операции, не изменения ни одно из них.

Использование монад для компоновки без побочных эффектов

Монады — мощное средство компоновки, используемое в функциональном программировании, чтобы избежать опасного и нежелательного поведения (побочных эффектов). Монады позволяют принимать значение и применять к нему серию независимых преобразований, инкапсулируя побочные эффекты. Сигнатура типа монадической функции вызывает потенциальные побочные эффекты, обеспечивая представление как результата вычисления, так и фактических побочных эффектов, которые возникли в ходе выполнения. Монадическое вычисление представляется в виде параметрического типа $M<'a>$, где параметр типа определяет тип значения (или значений), полученных в результате монадического вычисления (внутренний тип может быть, например, `Task` или `List`). При написании кода с применением монадических вычислений внутренний тип напрямую не используется. Вместо этого выполняются две операции, которые должны обеспечивать все монадические вычисления: `Bind` и `Return`.

Данные операции определяют поведение монады и имеют следующие сигнатуры типов (для некоторых монад типа $M<'a>$, которые могут быть заменены на `Task<'a>`):

```
Bind: ('a -> M<'b>) -> M<'a> -> M<'b>
Return: 'a -> M<'a>
```

Оператор `Bind` принимает экземпляр надтипа, извлекает из него внутреннее значение и выполняет для этого значения функцию, возвращая новое значение надтипа:

```
Task<R> Bind<R, T>(this Task<T> task, Func<T, Task<R>> continuation)
```

В данной реализации, как видим, оператор `SelectMany` встроен в библиотеку LINQ/PLINQ.

`Return` — это оператор, который преобразует (обращивает) любой тип в контекст надтипа (монаду, такую как `Task`), обычно преобразуя немонадическое значение в монадическое. Например, `Task.FromResult` создает `Task<T>` из любого заданного типа `T` (выделен жирным шрифтом):

```
Task<T> Return<T>(T value) => Task.FromResult(value);
```

Такие монадические операторы необходимы для LINQ/PLINQ и делают возможными многие другие операторы. Например, предыдущий код, который загружает пиктограмму с заданного сайта и преобразует ее в растровый формат, может быть переписан с использованием монадических операторов (выделены жирным шрифтом) следующим образом:

```
Bitmap icon = await (from content in new HttpClient().GetAsync($"http://
{domain}/favicon.ico")
                     from bytes in content.Content.ReadAsByteArrayAsync()
                     select Bitmap.FromStream(new MemoryStream(bytes));
```

Монада — удивительно универсальный шаблон для компоновки функций с преобразованием в надтипы и сохранением возможности применять функции к экземплярам внутренних типов. Монады также предоставляют технологии для удаления дублирующегося и громоздкого кода; они значительно упрощают многие проблемы программирования.

Почему важны законы

Как мы видели, каждый из упомянутых математических шаблонов должен удовлетворять определенным законам, чтобы его свойства раскрылись в полной мере. Но почему это так важно? Причина в том, что законы помогают нам рассуждать о программе, предоставляя информацию об ожидаемом поведении типа в контексте. В частности, конкурентная программа должна быть детерминированной; следовательно, детерминированный и предсказуемый способ рассуждения о коде помогает убедиться в его корректности. Если для объединения двух моноидов применяется некая операция, то можно предположить, что в соответствии с законами моноидов вычисления являются ассоциативными и тип результата также будет моноидом. Для написания конкурентных комбинаторов важно доверять законам, которые следуют из абстрактного интерфейса, таким как монады и функторы.

10.6. Окончательный вариант параллельной компоновки аппликативного функтора

Мы уже обсудили, как можно использовать функтор (`fmap`) для преобразования функции, принимающей один аргумент, так, чтобы она могла работать с надтипаами. Вы также научились использовать монадические операторы `Bind` и `Return` для управляемой и быстрой компоновки надтипов. Но это еще не все! Предположим, что у нас есть функция из *обычного мира*: например, метод, который обрабатывает изображение для создания миниатюры `Thumbnail` для заданного растрового объекта `Bitmap`. Как бы вы применили такую функциональность к значениям из *мира надтипов* `Task<Bitmap>?`

ПРИМЕЧАНИЕ

Под обычным миром в данном случае понимаются функции, которые выполняются для обычных примитивов, таких как растровое изображение. В отличие от них функции из мира надтипов оперируют надтипами: например, в результате обертывания растрового изображения получается `Task<Bitmap>`.

Ниже представлена функция `ToThumbnail`, предназначенная для обработки заданного изображения (код, на который нужно обратить внимание, выделен жирным шрифтом):

```
Image ToThumbnail (Image bitmap, int maxPixels)
{
    var scaling = (bitmap.Width > bitmap.Height)
        ? maxPixels / Convert.ToDouble(bitmap.Width)
        : maxPixels / Convert.ToDouble(bitmap.Height);
    var width = Convert.ToInt32(Convert.ToDouble(bitmap.Width) * scaling);
    var height = Convert.ToInt32(Convert.ToDouble(bitmap.Height) * scaling);
    return new Bitmap(bitmap.GetThumbnailImage(width, height, null,
    IntPtr.Zero));
}
```

Конечно, можно получить значительное количество различных форм компоновки, используя базовые функции, такие как `map` и `bind`, но у них есть ограничение: эти функции принимают на входе только один аргумент. Как интегрировать в рабочие процессы функции с несколькими аргументами, учитывая, что обе функции, и `map`, и `bind`, принимают в качестве входных данных унарную функцию? Решение заключается в применении *аппликативных функторов*.

Для того чтобы понять, почему следует применять шаблон (технологию) Applicative Functor, рассмотрим пример. В состав функтора входит оператор `map` для преобразования функций с одним и только одним аргументом.

Обычно функции, выполняющие отображение в надтипы, принимают несколько аргументов, например, рассмотренный ранее метод `ToThumbnail`, который в качестве первого аргумента принимает изображение, а в качестве второго — максимальный размер преобразованного изображения в пикселях. Проблема таких функций заключается в том, что их нелегко обернуть в другие контексты. Если мы загружаем изображение — предположим для простоты, из хранилища Azure Blob с помощью функции `DownloadImageAsync`, как в предыдущих примерах, — а затем хотим применить преобразование с помощью функции `ToThumbnail`, то мы не можем использовать функтор `map` из-за несовпадения сигнатур типов. Функция `ToThumbnail` (в листинге 10.19 выделена жирным шрифтом) принимает два аргумента, а функция `map` принимает на вход только одну функцию-аргумент.

Листинг 10.19. Ограничение компоновки функтора отображения Task

```
Task<R> map<T, R>(this Task<T> task, Func<T, R> map) =>
    task.ContinueWith(t => map(t.Result));

static async Task<Image> DownloadImageAsync(string blobReference)
{
    var container = await Helpers.GetCloudBlobContainerAsync().
        ConfigureAwait(false);
    CloudBlockBlob blockBlob = container.
        GetBlockBlobReference(blobReference);
    using (var memStream = new MemoryStream())
    {
        await blockBlob.DownloadToStreamAsync(memStream).
            ConfigureAwait(false);
        return Bitmap.FromStream(memStream);
    }
}
static async Bitmap CreateThumbnail(string blobReference, int maxPixels)
{
    Image thumbnail =
        await DownloadImageAsync("Bugghina001.jpg")
            .map(ToThumbnail); ←————| Ошибка компиляции
    return thumbnail;
}
```

Проблема этого кода заключается в том, что при попытке применить `ToThumbnail` к методу расширения отображения `Task — map(ToThumbnail)` он не компилируется. Компилятор выдает исключение из-за несоответствия сигнатур.

Как применить функцию сразу к нескольким контекстам? Как изменить для этого функцию, которая принимает несколько аргументов? Именно здесь в игру вступают аппликативные функторы, позволяющие применить многопараметрическую функцию к надтипу. В листинге 10.20 аппликативные функторы использованы для компоновки функций `ToThumbnail` и `DownloadImageAsync`. Они позволяют привести в соответствие сигнатуры типов и сохранить асинхронную семантику (выделены жирным шрифтом).

Листинг 10.20. Улучшенный вариант компоновки асинхронной операции

```
Static Func<T1, Func<T2, TR>> Curry<T1, T2, TR>(this Func<T1, T2, TR> func)
➥ => p1 => p2 => func(p1, p2);

static async Task<Image> CreateThumbnail(string blobReference, int maxPixels)
{
    Func<Image, Func<int, Image>> ToThumbnailCurried =
    ➤ Curry<Image, int, Image>(ToThumbnail); ← Каррированная функция

    ➤ Image thumbnail = await TaskEx.Pure(ToThumbnailCurried)
        .Apply(DownloadImageAsync(blobReference))
        .Apply(TaskEx.Pure(maxPixels));

    return thumbnail;
} ← Использование аппликативной функции
      для построения цепочки вычислений
      без выхода за пределы контекста Task
```

Преобразование функции
`ToThumbnailCurried` для надтипа `Task`

Рассмотрим этот листинг внимательно, чтобы выяснить все подробности. Функция `Curry` является частью вспомогательного статического класса, используемого для обеспечения стиля функционального программирования в C#. В данном случае каррированная версия метода `ToThumbnail` — это функция, которая принимает на входе изображение и возвращает функцию. А она, в свою очередь, принимает целое число (`int`), обозначающее максимально допустимый размер (в пикселях), и возвращает на выходе тип `Image: Func<Image, Func<int, Image>>` `ToThumbnailCurried`. Затем эта унарная функция обертыивается в контейнерный тип `Task` и перегружается, чтобы посредством ее каррирования можно было определить большее количество аргументов.

На практике функция, которая принимает более одного аргумента — в данном случае `ToThumbnail`, — каррируется и обертывается в тип `Task` с помощью метода расширения `Task Pure`. Затем результат `Task<Func<Image, Func<int, Image>>>` передается через аппликативный функтор `Apply`, который добавляет свои выходные данные, `Task<Image>`, в следующую функцию, применяемую к `DownloadImageAsync`.

В итоге последний оператор аппликативного функтора `Apply` обрабатывает переходный параметр `maxPixels`, преобразованный в надтип с помощью метода расширения `Pure`. С точки зрения оператора-функтора `map`, каррированная функция `ToThumbnailCurried` частично применяется к аргументу `image`, а затем обертывается в тип `Task`. Следовательно, ее сигнатура, по идее, должна быть такой:

`Task<ToThumbnailCurried (Image)>`

Функция `ToThumbnailCurried` принимает на входе `image`, а затем возвращает частично примененную функцию в виде делегата `Func<int, Image>`, определение сигнатуры которого корректно соответствует входу аппликативного функтора: `Task<Func<int, Image>>`.

Функцию `Apply` можно рассматривать как частичное применение для обернутых функций, следующее значение которых предоставляется для каждого вызова в форме обернутого значения (надтипа). Таким образом, можно превратить любой аргумент функции в обернутое значение.

Каррирование и частичное применение

Каррирование — это технология преобразования функции с несколькими аргументами в набор функций с одним аргументом, которые всегда возвращают значение. Язык программирования F# выполняет каррирование автоматически; в C# каррирование необходимо активировать вручную, или, точнее, с помощью специальной вспомогательной функции, как показано в примере. Именно поэтому сигнатуры функций в функциональном программировании представляются с несколькими символами `->`, которые обычно являются символами функции. Рассмотрим, например, функцию с такой сигнатурой: `string -> int -> Image`.

Эта функция имеет два аргумента, `string` и `int`, и возвращает `Image`. Но сигнатуру подобной функции следует читать так: это функция, которая имеет только один аргумент, `string`, и возвращает новую функцию с сигнатурой `int -> Image`.

Может показаться, что из-за отсутствия поддержки каррирования и частичного применения использовать аппликативные функторы в C# неудобно. Но, немного по-практиковавшись, вы преодолеете первоначальный барьер и увидите действительные возможности и гибкость технологии аппликативных функторов.

Частичное применение означает, что функция с несколькими аргументами не должна применяться ко всем ее входным данным сразу. Это можно представить так: применение функции к первому аргументу дает не значение, а функцию с $n - 1$ аргументами. Таким образом можно привязать многопараметрическую функцию к одной задаче и получить асинхронную операцию с $n - 1$ аргументами. Остается лишь решить проблему применения задачи функции к задаче аргумента — именно для этого используется аппликативный шаблон.

Подробнее о каррировании читайте в приложении А.

Назначение шаблона аппликативного функтора — обернуть функцию и применить ее к обернутому контексту, а затем применить вычисление (преобразование) к определенному надтипу. Поскольку и значение, и функция используются в одном и том же обернутом контексте, их можно объединить.

Проанализируем функции `Pure` и `Apply`. Аппликативный функтор — это шаблон, реализованный в виде двух определенных ниже операций, где `AF` — любой надтип (выделены жирным шрифтом):

Pure : `T -> AF<R>`

Apply : `AF<T -> R> -> AF<T> -> F<R>`

Интуитивно понятно, что оператор `Pure` обертыывает значение и помещает его в область надтипа, что эквивалентно монадическому оператору `Return`. Название `Pure` по соглашению соответствует определению аппликативного функтора. Но в случае аппликативных функций данный оператор обертывает функцию. Оператор `Apply` — это двухпараметрическая функция, оба параметра которой принадлежат одному домену надтипов.

Из примера кода, представленного в подразделе «Функторы для отображения надтипов» предыдущего раздела, видно, что аппликативный функтор — это любой контейнер (надтип), который предлагает способ преобразования обычной функции в функцию, оперирующую содержащимися в надтипе значениями.

ПРИМЕЧАНИЕ

Аппликативный функтор — это конструкция, представляющая собой нечто среднее между функтором `map` и монадой `bind`. Аппликативные функторы могут работать с обернутыми функциями, поскольку их значения обернуты в контекст, как и в случае функтора, но в данном случае обернутое значение является функцией. Это полезно, если нужно применить функцию, которая находится внутри функтора, к значению, которое также находится внутри функтора.

Аппликативные функторы полезны при построении последовательности параллельных действий без необходимости получения каких-либо промежуточных результатов. Фактически если задачи независимы, то их выполнение может быть скомпоновано и распараллелено с использованием аппликативности. Примером является выполнение множества конкурентных действий, которые читают и преобразуют части структуры данных в определенном порядке, а затем объединяют результаты, как показано на рис. 10.8.

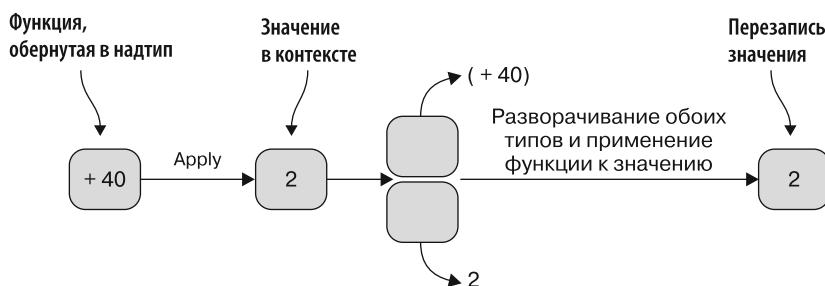


Рис. 10.8. Оператор `Apply` реализует функцию, обернутую в надтип и применяемую к значению в контексте. Этот процесс приводит к развертыванию обоих значений; затем, поскольку первое значение является функцией, оно автоматически применяется ко второму значению. Наконец, результат снова обертывается в контекст надтипа

В контексте надтипа `Task` оператор принимает значение `Task<T>` и обернутую функцию `Task<(T -> R)>` (в C# она преобразована в `Task<Func <T, R >>`), а затем

возвращает новое значение `Task<R>`, созданное путем применения исходной функции к значению `Task<T>`:

```
static Task<R> Apply<T, R>(this Task<Func<T, R>> liftedFn, Task<T> task) {  
    var tcs = new TaskCompletionSource<R>();  
    liftedFn.ContinueWith(innerLiftTask =>  
        task.ContinueWith(innerTask =>  
            tcs.SetResult(innerLiftTask.Result(innerTask.Result))  
        ));  
    return tcs.Task;  
}
```

Ниже представлен вариант оператора `Apply`, определенный для `async Task` в мире TAP, который может быть реализован вместо варианта `async/await`:

```
static async Task<R> Apply<T, R> (this Task<Func<T, R>> f, Task<T> arg)  
    => (await f.ConfigureAwait(false))  
        (await arg.ConfigureAwait(false));
```

Поведение у обеих функций `Apply` одинаковое, несмотря на различные реализации. Первое входное значение `Apply` — это функция, обернутая в `Task: Task<Func<T, R>>`. На первый взгляд данная сигнатура может показаться странной, но вспомним, что в функциональном программировании функции рассматриваются как значения и могут передаваться так же, как строки или целые числа.

Теперь можно легко расширить оператор `Apply` до сигнатур, которая принимает несколько входных данных. Вот пример такой функции:

```
static Task<Func<b, c>> Apply<a, b, c>(this Task<Func<a, b, c>> liftedFn,  
    Task<a> input) =>
```

```
    Apply(liftedFn.map(Curry), input);
```

Обратите внимание, что это разумная реализация, поскольку в ней функция `Curry` применяется к `Task<Func<a, b, c>> liftedFn`, используя функтор `map`, а затем результат применяется к обернутому входному значению с использованием оператора `Apply` с меньшим количеством аргументов, как было определено ранее. С помощью этой технологии мы продолжим расширять оператор `Apply`, чтобы в качестве входных данных он мог принимать обернутую функцию с любым количеством параметров.

ПРИМЕЧАНИЕ

Последовательность аргументов в операторе `Apply` обратная по сравнению со стандартной сигнатурой, поскольку этот оператор реализован как статический метод для упрощения его использования.

Оказывается, обычные и аппликативные функторы хорошо сочетаются, позволяя упростить компоновку, в том числе компоновку выражений, выполняемых параллельно. При передаче функтору `map` функции с несколькими аргументами тип результата совпадает с типом входных данных функции `Apply`.

Существует и другой способ реализации applicативного функтора — с использованием монадических операторов `bind` и `return`. Но такой подход не позволяет выполнять код параллельно, поскольку выполнение каждой операции зависит от результата предыдущей операции.

С помощью applicативного функтора легко составить набор вычислений без ограничения количества аргументов, принимаемых каждым выражением. Предположим, что нам нужно объединить два изображения, чтобы получить третье, представляющее собой наложение исходных изображений друг на друга, которое затем вставляется в рамку определенного размера. В листинге 10.21 показано, как это сделать (функция `Apply` выделена жирным шрифтом).

Листинг 10.21. Распараллеливание цепочки вычислений с помощью applicативных функторов

```
static Image BlendImages(Image imageOne, Image imageTwo, Size size)
{
    var bitmap = new Bitmap(size.Width, size.Height);
    using (var graphic = Graphics.FromImage(bitmap)) {
        graphic.InterpolationMode = InterpolationMode.HighQualityBicubic;
        graphic.DrawImage(imageOne,
            new Rectangle(0, 0, size.Width, size.Height),
            new Rectangle(0, 0, imageOne.Width, imageTwo.Height),
            GraphicsUnit.Pixel);
        graphic.DrawImage(imageTwo,
            new Rectangle(0, 0, size.Width, size.Height),
            new Rectangle(0, 0, imageTwo.Width, imageTwo.Height),
            GraphicsUnit.Pixel);
        graphic.Save();
    }
    return bitmap;
}
async Task<Image> BlendImagesFromBlobStorageAsync(string blobReferenceOne,
➥ string blobReferenceTwo, Size size)
{
    Func<Image, Func<Image, Func<Size, Image>>> BlendImagesCurried =
        Curry<Image, Image, Size, Image>(BlendImages);
    Task<Image> imageBlended =
        TaskEx.Pure(BlendImagesCurried)
            .Apply(DownloadImageAsync(blobReferenceOne))
            .Apply(DownloadImageAsync(blobReferenceTwo))
            .Apply(TaskEx.Pure(size));
    return await imageBlended;
}
```

При первом вызове функции `Apply` с задачей `DownloadImageAsync(blobReferenceOne)` она немедленно возвращает новую задачу, не дожидаясь завершения задачи `DownloadImageAsync`; таким образом, программа сразу приступает к созданию второй задачи `DownloadImageAsync(blobReferenceTwo)`. В результате обе выполняются параллельно.

Код предполагает, что все функции имеют одинаковый тип входных и выходных данных; но это не ограничение. До тех пор пока тип выходных данных выражения

совпадает с типом входных данных следующего выражения, вычисление все еще выполняется и является валидным. Обратите внимание, что в листинге 10.21 каждый вызов запускается независимо от других, поэтому вызовы выполняются параллельно и общее время выполнения для завершения `BlendImagesFromBlobStorageAsync` определяется наибольшим временем, требуемым для завершения вызовов `Apply`.

Apply или bind?

Различия в поведении между операторами `bind` и `Apply` описываются сигнатурой их функций. Например, в контексте асинхронного рабочего процесса в операторе `bind` сначала выполняется тип `Async` и ожидает завершения, чтобы начать вторую асинхронную операцию. Оператор `bind` следует использовать, когда выполнение асинхронной операции зависит от значения, возвращаемого другой асинхронной операцией.

В сигнатуре оператора `Apply` обе асинхронные операции представлены в качестве аргументов. Этот оператор следует использовать, когда асинхронные операции могут выполняться независимо одна от другой.

Данные концепции верны и для других надтипов, таких как тип `Task`.

В данном примере особое внимание уделено аспекту компоновки конкурентных функций. Вместо этого можно было бы создать специальные методы, которые бы со-вмещали изображения напрямую, но в более крупной схеме рассмотренный подход позволяет гибко комбинировать более сложные варианты поведения.

10.6.1. Расширение асинхронного рабочего процесса F# с помощью операторов applicативных функторов

Продолжая знакомство с applicативными функторами, в этом разделе мы изучим на практике уже знакомые нам концепции задач, позволяющие расширить асинхронный рабочий процесс F#. Обратите внимание, что F# поддерживает TPL, поскольку он является частью экосистемы .NET, а applicативные функторы основаны на применяемом типе `Task`.

В листинге 10.22 реализованы два applicативных функтора, `pure` и `apply`, которые специально определены внутри модуля `Async`, чтобы расширить этот тип. Обратите внимание, что, поскольку ключевое слово `pure` в F# зарезервировано для использования в будущем, компилятор выдаст предупреждение.

Функция `apply` параллельно выполняется для двух входных параметров, `funAsync` и `opAsync`, используя шаблон `Fork/Join`, а затем возвращает результат применения выходных данных первой функции по отношению ко второй.

Обратите внимание, что реализация оператора `apply` выполняется параллельно, поскольку каждая асинхронная функция начинает выполнение с помощью оператора `Async.StartChild`.

Листинг 10.22. Асинхронный applicative-функция в F#

```
module Async =
    let pure value = async.Return value           ← Обертывание значения в Async
    let apply funAsync opAsync = async {
        let! funAsyncChild = Async.StartChild funAsync
        let! opAsyncChild = Async.StartChild opAsync
        let! funAsyncRes = funAsyncChild
        let! opAsyncRes = opAsyncChild
        return funAsyncRes opAsyncRes
    }
```

Параллельный запуск
двух асинхронных функций

Ожидание результатов

Оператор Async.StartChild

Оператор `Async.StartChild` выполняет вычисление, которое начинается в асинхронном рабочем процессе, и возвращает маркер (типа `Async<'T>`), который можно использовать для ожидания завершения операции. Его сигнатура выглядит следующим образом:

```
Async.StartChild : Async<'T> * ?int -> Async<Async<'T>>
```

Этот механизм позволяет выполнять несколько асинхронных вычислений одновременно. Если родительское вычисление запрашивает результат, а дочернее вычисление еще не завершено, то родительское вычисление приостанавливается до тех пор, пока не будет завершено дочернее.

Какие возможности предоставляют эти функции? Концепции applicative-функции, представленные в C#, применимы и здесь, но предоставляемый F# семантический стиль компоновки приятнее. Использование конвейерного оператора F# (`|>`) для передачи промежуточного результата функции следующей функции позволяет написать более читаемый код.

В листинге 10.23 реализована та же цепочка функций для асинхронного совмещения двух изображений, которая была создана на C# в листинге 10.21, но только на этот раз на F# с использованием applicative-функции. Здесь функция `blendImagesFromBlobStorage` F# вместо типа `Task` возвращает тип `Async` (выделена жирным шрифтом).

Листинг 10.23. Параллельная цепочка операций с асинхронным applicative-функцией F#

```
let blendImages (imageOne:Image) (imageTwo:Image) (size:Size) : Image =
    let bitmap = new Bitmap(size.Width, size.Height)
    use graphic = Graphics.FromImage(bitmap)
    graphic.InterpolationMode <- InterpolationMode.HighQualityBicubic
    graphic.DrawImage(imageOne,
                      new Rectangle(0, 0, size.Width, size.Height),
                      new Rectangle(0, 0, imageOne.Width, imageTwo.Height),
                      GraphicsUnit.Pixel)
    graphic.DrawImage(imageTwo,
```

```
        new Rectangle(0, 0, size.Width, size.Height),
        new Rectangle(0, 0, imageTwo.Width, imageTwo.Height),
        GraphicsUnit.Pixel)
graphic.Save() |> ignore
bitmap :> Image

let blendImagesFromBlobStorage (blobReferenceOne:string)
  => (blobReferenceTwo:string) (size:Size) =
    Async.apply(
      Async.apply(
        Async.apply(
          Async.``pure`` blendImages)
          (downloadOptionImage(blobReferenceOne)))
          (downloadOptionImage(blobReferenceTwo)))
          (Async.``pure`` size))
```

Функция `blendImages` переносится в область `Task` (надтип) с помощью функции `Async.pure`. Полученная в результате функция с сигнатурой `Async<Image -> Image -> Image>` применяется к выходным данным функций `downloadOptionImage(blobReferenceOne)` и `downloadOptionImage(blobReferenceTwo)`. Обернутое значение `size` обрабатывается параллельно.

Как уже упоминалось, функции в F# по умолчанию являются каррированными; дополнительный шаблонный код, необходимый в C#, здесь не нужен. Несмотря на то что F# не поддерживает аппликативные функторы как встроенную функцию, легко реализовать оператор `apply` и использовать его преимущества компоновки. Но этот код не особенно элегантен, поскольку операторы функции `apply` являются вложенными, вместо того чтобы образовывать цепочку. Лучше создать собственный инфиксный оператор.

10.6.2. Семантика аппликативных функторов и инфиксных операторов в F#

Более декларативный и удобный подход к функциональной компоновке в F# заключается в создании собственных инфиксных операторов. К сожалению, в C# это свойство не поддерживается. Возможность создания собственных инфиксных операторов означает, что вы можете определить свои операторы для достижения желаемого уровня приоритета при работе с передаваемыми аргументами.

Инфиксный оператор в F# — это оператор, который выражается посредством математической нотации, называемой *инфиксной нотацией*. Например, оператор умножения принимает два числа, после чего одно умножается на другое. В данном случае, используя инфиксную нотацию, оператор умножения можно поставить между двумя числами, с которыми он работает. Большинство операторов — это функции с двумя аргументами, и в данном случае, вместо того чтобы записывать функцию умножения как `multiply x y`, инфиксный оператор располагается между двумя аргументами: `x Multiply y`.

Вы уже знакомы с несколькими инфиксными операторами F#: это конвейерный оператор `|>` и оператор компоновки `>>`. Но, согласно разделу 3.7 спецификации языка F#, можно создавать и собственные операторы. Следующие инфиксные операторы (выделены жирным шрифтом) определены для асинхронных функций `apply` и `map`:

```
let (<*>) = Async.apply  
let (<!>) = Async.map
```

ПРИМЕЧАНИЕ

Оператор `<!>` — инфиксная версия `Async.map`, а оператор `<*>` — инфиксная версия `Async.apply`. Эти инфиксные операторы обычно используются в других языках программирования, таких как Haskell, поэтому они стали стандартом. Оператор `<!>` в других языках программирования определен как `<$>`; но в F# оператор `<$>` зарезервирован на будущее, в связи с чем мы задействовали `<!>`.

Применяя эти операторы, можно переписать предыдущий код в более компактной форме:

```
let blendImagesFromBlobStorage (blobReferenceOne:string)  
  => (blobReferenceTwo:string) (size:Size) =  
    blendImages  
    <!> downloadOptionImage(blobReferenceOne)  
    <*> downloadOptionImage(blobReferenceOne)  
    <*> Async.``pure`` size
```

Обычно я рекомендую не злоупотреблять использованием инфиксных операторов, найти разумный баланс. Но, как видим, в случае обычных и applicативных функторов инфиксный оператор оказался очень кстати.

10.6.3. Использование applicативных функторов в гетерогенных параллельных вычислениях

Applicативные функторы приводят нас к мощной технологии, которая позволяет создавать код для гетерогенных параллельных вычислений. «Гетерогенные» означает, что объект состоит из нескольких частей разной природы (в отличие от *гомогенных*, то есть однородных). В контексте параллельного программирования это означает совместное выполнение нескольких операций, даже если у них разные типы выходных данных.

Например, функции `Async.Parallel` из текущей реализации F# и `Task.WhenAll` из TPL принимают в качестве аргументов последовательность асинхронных вычислений с одинаковым типом результата. Эта технология основана на сочетании applicативных функторов и концепции обертывания; ее целью является обертывание любого типа в другой контекст. Подобная идея применима к значениям и функциям; в данном случае целью выступают функции с произвольным количеством

аргументов разных типов. Чтобы можно было выполнять гетерогенные параллельные вычисления, используя это свойство, применяется оператор аппликативного функтора `apply` в сочетании с методикой обертывания функции. Затем с помощью этой комбинации создается несколько полезных функций, обычно называемых `Lift2`, `Lift3` и т. д. Операторы `Lift` и `Lift1` не определены, потому что являются функциями функтора `map`.

В листинге 10.24 показана реализация функций `Lift2` и `Lift3` на C#. Это прозрачное решение для выполнения параллельной функции `Async`, возвращающей гетерогенные типы. Данные функции будут использованы в следующих примерах.

Листинг 10.24. Асинхронные lift-функции на C#

```
lift-функции применяют эту функцию
к выходным данным последовательности задач

static Task<R> Lift2<T1, T2, R>(Func<T1, T2, R> selector, Task<T1> item1,
➥ Task<T2> item2)                                Каррирование функции для частичного использования
{
    Func<T1, Func<T2, R>> curry = x => y => selector(x, y); ←
    var lifted1 = Pure(curry);
    var lifted2 = Apply(lifted1, item1);               Создание обертки частично
    return Apply(lifted2, item2);                      примененной функции
}

static Task<R> Lift3<T1, T2, T3, R>(Func<T1, T2, T3, R> selector,
➥ Task<T1> item1, Task<T2> item2, Task<T3> item3)
{
    Func<T1, Func<T2, Func<T3, R>>> curry = x => y => z =>
        selector(x, y, z);                           ←
    var lifted1 = Pure(curry);
    var lifted2 = Apply(lifted1, item1);               Создание обертки частично
    var lifted3 = Apply(lifted2, item2);               примененной функции
    return Apply(lifted3, item3);
}

Создание надтипа с помощью оператора Apply
```

Реализация функций `Lift2` и `Lift3` основана на аппликативных функторах, которые каррируют функцию-селектор и создают ее обертку, обеспечивая возможность применения этой функции к надтипам.

Те же концепции реализации функций `Lift2` и `Lift3` определяют и структуру программы на F#. Но из-за внутренней функциональной природы этого языка программирования и компактности, обеспечиваемой инфиксными операторами, реализация lift-функций (выделены жирным шрифтом) на F# получается более лаконичной:

```
let lift2 (func:'a -> 'b -> 'c) (asyncA:Async<'a>) (asyncB:Async<'b>) =
    func <!> asyncA <*> asyncB

let lift3 (func:'a -> 'b -> 'c -> 'd) (asyncA:Async<'a>)
➥ (asyncB:Async<'b>) (asyncC:Async<'c>) =
    func <!> asyncA <*> asyncB <*> asyncC
```

Благодаря системе вывода типов F# входные значения обертываются в тип `Async` и компилятор может интерпретировать инфиксные операторы `<*>` и `<!>` как функторы и аппликативные функторы в контексте надтипа `Async`. Также обратите внимание, что в F# принято присваивать функциям уровня модуля имена, начинающиеся со строчной буквы.

10.6.4. Компоновка и выполнение гетерогенных параллельных вычислений

Какую пользу мы можем извлечь из этих функций? Проанализируем пример, в котором применяются данные операторы.

Представьте, что вам поручено написать простую программу для проверки правильности решения о покупке опционов на акции в зависимости от условий, полученных на основе анализа рыночных тенденций и истории изменения курса акций. Программа должна быть разделена на три операции.

1. Проверить общую сумму, выделенную для покупки акций, на основе остатка на банковском счету и текущей цены акции:
 - 1) получить остаток на банковском счету;
 - 2) получить цену акции с фондового рынка.
2. Проверить, является ли данное биржевое название рекомендованным для покупки:
 - 1) проанализировать рыночные индексы;
 - 2) проанализировать тенденцию изменения курса этой акции на основании ее истории.
3. Принять решение для данного биржевого кода: покупать или не покупать определенное количество опционов на акции в зависимости от доступной суммы денег, вычисленной на шаге 1.

В листинге 10.25 представлены асинхронные функции для реализации программы, которые в идеале должны быть объединены (выделены жирным шрифтом). Отдельные детали реализации кода опущены, поскольку они не имеют отношения к данному примеру.

Листинг 10.25. Асинхронные операции для компоновки и параллельного выполнения

```
let calcTransactionAmount amount (price:float) = ←
  let readyToInvest = amount * 0.75
  let cnt = Math.Floor(readyToInvest / price)
  if (cnt < 1e-5) && (price < amount)
    then 1 else int(cnt)

  Вычисление суммы транзакции,
  включая условные сборы

let rnd = Random()
let mutable bankAccount = 500.0 + float(rnd.Next(1000))
let getAmountOfMoney() = async {
```

```

    return bankAccount
}

} ← Имитация обращения асинхронного веб-сервиса к банковской
      учетной записи, которая возвращает случайное значение

→ let getCurrentPrice symbol = async {
    let! (_,data) = processStockHistory symbol ←
        Функция processStockHistory (см. главу 8)
        загружает и анализирует тенденцию
        изменений заданного биржевого кода акций
    return data.[0].open'
}

let getStockIndex index = async {
    let url = sprintf "http://download.finance.yahoo.com/d/quotes.
    → csv?s=%s&f=snl1" index ←
        Асинхронная загрузка с фондового рынка
        и получение цены этого биржевого кода
    Получение последней цены акции
    let req = WebRequest.Create(url)
    let! resp = req.AsyncGetResponse()
    use reader = new StreamReader(resp.GetResponseStream())
    return! reader.ReadToEndAsync()
}

|> Async.map (fun (row:string) ->
    let items = row.Split(',')
    Double.Parse(items.[items.Length-1]))
|> AsyncResult.handler ←
    Отображение данных банковского
    кода акций путем получения цены
    на момент закрытия биржи. Результат
    имеет тип AsyncResult, поскольку
    операция может выдавать исключение

let analyzeHistoricalTrend symbol = asyncResult { ←
    let! data = getStockHistory symbol (365/2)
    let trend = data.[data.Length-1] - data.[0]
    return trend
}

let withdraw amount = async {
    return
        if amount > bankAccount
        then Error(InvalidOperationException("Not enough money"))
        else
            bankAccount <- bankAccount - amount
            Ok(true)
}
} ← Анализ изменений тенденций этого
      биржевого кода. Операция выполняется
      асинхронно в вычислительном
      выражении asyncResult, чтобы
      обрабатывать потенциальные ошибки

Получение текущего результата, доступного асинхронно,
или Error, если на банковском счету недостаточно
средств для совершения торговой операции

```

Все операции выполняются асинхронно и выдают результаты разных типов. Соответственно, функция `calcTransactionAmount` возвращает гипотетическую стоимость сделки (покупки); функция `analysisHistoricalTrend` возвращает результат анализа курса акций, который используется для определения того, можно ли считать данный опцион рекомендаемой покупкой; функция `getStockIndex` возвращает текущее значение цены акции, а функция `getCurrentPrice` — последнюю цену акции.

Как бы вы скомпоновали и выполняли эти вычисления параллельно, используя, например, шаблон `Fork/Join`, с учетом того, что типы результатов различны? Проще всего было бы создать независимую задачу для каждой функции, а затем ожидать завершения всех задач для передачи результатов в итоговую функцию, объединя-

ющую результаты и продолжающую работу. Но было бы гораздо лучше склеить все эти функции, используя более общий комбинатор, который бы обеспечивал много-кратное применение и, конечно же, более удобную компоновку с помощью набора полиморфных инструментов.

В коде из листинга 10.26, написанном на F#, применяется метод параллельного выполнения гетерогенных вычислений с использованием функции `lift2`. После асинхронного выполнения нескольких простых задач диагностики (выделенных жирным шрифтом) можно будет оценить, сколько опционов на акции рекомендуется купить.

Листинг 10.26. Выполнение гетерогенных асинхронных операций

```
Создание обертки гетерогенной функции
для применения операции calcTransactionAmount
к результатам функций getAmountOfMoney
и getCurrentPrice

let howMuchToBuy stockId : AsyncResult<int> =
    Async.lift2 (calcTransactionAmount)
        (getAmountOfMoney())
        (getCurrentPrice stockId)
    |> AsyncResult.handler

let analyze stockId =           ←
    howMuchToBuy stockId          | Аналisis заданного биржевого названия акции, на выходе
                                    | получаем рекомендацию о продолжении покупки
    |> Async.StartCancelable(function
        | Ok (total) -> printfn "I recommend to buy %d unit" total
        | Error (e) -> printfn "I do not recommend to buy now")
```

Запуск вычисления с использованием оператора `Async.StartCancelable`.

Шаблон функции продолжения соответствует входному аргументу `Result`,

чтобы выполнить оставшуюся часть вычислений в зависимости

от того, является результат успешным (OK) или нет (Error)

`HowMuchToBuy` — это двухпараметрическая функция, возвращающая тип `AsyncResult<float>`. Тип результата определяется типом результата внутренней функции `calcTransactionAmount`, в которой `AsyncResult<float>` указывает либо на успешное завершение операции — рекомендуется купить данное количество акций, — либо на то, что покупать акции не следует. Первым аргументом `stockId` является произвольное биржевое название акций для анализа. Функция `howMuchToBuy` использует оператор `lift2` и без блокировок ожидает завершения вычислений двух внутренних асинхронных выражений (`getAmountOfMoney` и `getCurrentPrice`). Функция `analyze` выполняет `howMuchToBuy`, чтобы сформулировать и вывести рекомендуемый результат. В данном случае выполнение является асинхронным с применением функции `Async.StartCancelable`, определенной в подразделе 9.3.5.

Одним из многих преимуществ использования applicативных и обычных функций, монад и комбинаторов являются их воспроизводимость и общие шаблоны (независимо от применяемой технологии). Это облегчает понимание и создание словаря, который можно задействовать для коммуникации с разработчиками и описания назначения кода.

10.6.5. Управление потоком с помощью условных асинхронных комбинаторов

Как правило, разработчики стремятся реализовать комбинаторы посредством «склеивания» других комбинаторов. Если существует набор операторов, с помощью которых можно описать любую асинхронную операцию, то можно легко составлять новые комбинаторы типов, позволяющие компоновать и составлять асинхронные операции множеством различных способов.

Существует безграничное количество возможностей настройки асинхронных комбинаторов в соответствии с потребностями программы. Например, можно реализовать асинхронный комбинатор, который эмулирует оператор `if-else`, эквивалентный императивной условной логике. Но как это сделать?

Решение лежит в области функциональных шаблонов:

- ❑ для создания комбинатора `Or` можно использовать моноиды;
- ❑ для создания комбинатора `And` можно задействовать аппликативные функторы;
- ❑ для составления цепочек асинхронных операций и склеивания комбинаторов можно применять монады.

В данном подразделе мы определим несколько условных асинхронных комбинаторов и используем их, чтобы понять, какие возможности они предоставляют и как мало усилий для этого потребуется. На самом деле, применяя уже известные нам комбинаторы, можно получить разнообразные варианты поведения — это всего лишь вопрос компоновки. Кроме того, в случае инфиксных операторов F# данную функцию легко использовать для обертывания встроенных функций и управления ими, избегая применения промежуточных функций. Например, мы определили функции, такие как `lift2` и `lift3`, с помощью которых можно применять гетерогенные параллельные вычисления.

Можно абстрагироваться от понятия комбинации и сосредоточиться на условных операторах, таких как `IF`, `AND` и `OR`. В листинге 10.27 показаны несколько комбинаторов, которые применяются к асинхронному рабочему процессу F#. Семантически эти операторы лаконичны и легко компонуются благодаря функциональным свойствам данного языка программирования. Но те же концепции можно перенести и на C# — вообще без усилий или максимум с помощью свойства интероперабельности (код, на который следует обратить внимание, выделен жирным шрифтом).

Листинг 10.27. Условные комбинаторы асинхронного рабочего процесса

```
module AsyncCombinators =
    let inline ifAsync (predicate:Async<bool>) (funcA:Async<'a>)
    => (funcB:Async<'a>) =
        async.Bind(predicate, fun p -> if p then funcA else funcB)

    let inline iffAsync (predicate:Async<'a -> bool>) (context:Async<'a>) =
        async {
```

```

let! p = predicate <*> context
return if p then Some context else None }

let inline notAsync (predicate:Async<bool>) =
    async.Bind(predicate, not >> async.Return)

let inline AND (funcA:Async<bool>) (funcB:Async<bool>) =
    ifAsync funcA funcB (async.Return false)

let inline OR (funcA:Async<bool>) (funcB:Async<bool>) =
    ifAsync funcA (async.Return true) funcB

let (&&)(funcA:Async<bool>) (funcB:Async<bool>) = AND funcA funcB
let (|||)(funcA:Async<bool>) (funcB:Async<bool>) = OR funcA funcB

```

Комбинатор `ifAsync` принимает в качестве аргументов асинхронный предикат и две произвольные асинхронные операции, при этом в соответствии с результатом предиката будет выполнено только одно из данных вычислений. Это полезный шаблон, обеспечивающий ветвление логики асинхронной программы без выхода из асинхронного контекста.

Комбинатор `iffAsync` принимает условие HOF, которое верифицирует заданный контекст. Если условие выполняется, то комбинатор асинхронно возвращает контекст; в противном случае он асинхронно возвращает `None`. Комбинаторы из предыдущего кода могут применяться в любом сочетании перед началом выполнения; они играют роль «синтаксического сахара», благодаря которому код выглядит так же, как если бы он был последовательным.

Встроенные функции

Ключевое слово `inline` позволяет встроить тело функции в точку ее вызова. Таким образом, функции, помеченные как `inline`, дословно вставляются в код всякий раз, когда функция вызывается во время компиляции, что повышает производительность при выполнении кода. Обратите внимание, что встраивание — процесс компилятора, при котором размер кода увеличивается в обмен на повышение скорости выполнения, при этом для небольших или простых методов вызовы метода заменяются вставкой его тела.

Проанализируем подробнее такие логические асинхронные комбинаторы, чтобы лучше понять, как они работают. Подобные знания являются ключевым условием для создания собственных комбинаторов.

Логический асинхронный комбинатор AND

Асинхронный комбинатор `AND` возвращает результат после завершения функций `funcA` и `funcB`. Такое поведение аналогично `Task.WhenAll`, но этот оператор сначала запускает первое выражение и ждет результата, а затем вызывает второе и объединяет

результаты. Если выполнение будет отменено, или завершится сбоем, или будет возвращен неверный результат, то вторая функция не будет запущена в соответствии с логикой короткого замыкания.

Концептуально описанный ранее оператор `Task.WhenAll` хорошо подходит для выполнения логического AND между несколькими асинхронными операциями. Этот оператор применяет пару итераторов к контейнеру задач или к переменному числу задач и возвращает одну задачу, которая запускается, когда будут готовы все аргументы.

Оператор AND можно встраивать в цепочки операторов, если все они возвращают один и тот же тип. Разумеется, его можно обобщить и расширить с помощью applicative функторов. Если у функций нет побочных эффектов, то их результат является детерминированным и не зависит от последовательности выполнения, поэтому такие функции могут выполняться параллельно.

Логический асинхронный комбинатор OR

Асинхронный комбинатор OR работает как оператор сложения, имеющий моноидную структуру. Это означает, что операции должны быть ассоциативными. Комбинатор OR запускает две асинхронные операции параллельно, ожидая завершения первой из них. Ему присущи те же свойства, что и комбинатору AND. Комбинатор OR можно включать в цепочки операторов; однако его результат может быть детерминированным только в том случае, если в результате выполнения обеих функций возвращается результат одного и того же типа, а в случае отмены — если будут отменены сразу обе функции.

Комбинатор, действующий как логический OR для двух асинхронных операций, может быть реализован с помощью оператора `Task.WhenAny`, который запускает вычисления параллельно и выбирает то из них, которое завершится первым. Он также является основой для качественных вычислений, когда несколько алгоритмов сравниваются между собой.

Такой же подход для построения асинхронных комбинаторов может быть применен к типу `AsyncResult`, что представляет собой более мощный способ определения параметрических операций, результат которых зависит от успешности завершения внутренних операций. Другими словами, `AsyncResult` играет роль двух флагов состояния, которые соответствуют либо сбою, либо успешному завершению операции, причем второй флаг предоставляет окончательное значение. Ниже приводится несколько примеров комбинаторов `AsyncResult` (в листинге 10.28 выделены жирным шрифтом).

Листинг 10.28. Условные комбинаторы AsyncResult

```
module AsyncResultCombinators =
    let inline AND (funcA:AsyncResult<'a>) (funcB:AsyncResult<'a>)
    => : AsyncResult<_> =
        asyncResult {
```

```

        let! a = funcA
        let! b = funcB
        return (a, b)
    }

let inline OR (funcA:AsyncResult<'a>) (funcB:AsyncResult<'a>)
→ : AsyncResult<'a> =
    asyncResult {
        return! FuncA
        return! FuncB
    }

let (<&&&>) (funcA:AsyncResult<'a>) (funcB:AsyncResult<'a>) =
    AND funcA funcB
let (<|||>) (funcA:AsyncResult<'a>) (funcB:AsyncResult<'a>) =
    OR funcA funcB

let (<||||>) (funcA:AsyncResult<bool>) (funcB:AsyncResult<bool>) =
    asyncResult {
        let! rA = funcA
        match rA with
        | true -> return! FuncB
        | false -> return false
    }

let (<&&&&>) (funcA:AsyncResult<bool>) (funcB:AsyncResult<bool>) =
    asyncResult {
        let! (rA, rB) = funcA &&& funcB
        return rA && rB
    }

```

Комбинаторы `AsyncResult`, по сравнению с комбинаторами `Async`, позволяют создавать логические асинхронные операторы `AND` и `OR`, которые выполняют логическое ветвление кода для параметрических, а не для логических типов. Ниже показано сравнение операторов `AND`, реализованных на основе `Async` и `AsyncResult`:

```

let inline AND (funcA:Async<bool>) (funcB:Async<bool>) =
    ifAsync funcA funcB (async.Return false)

let inline AND (funcA:AsyncResult<'a>) (funcB:AsyncResult<'a>)
→ : AsyncResult<_> =
    asyncResult {
        let! a = funcA
        let! b = funcB
        return (a, b)
    }

```

В операторе `AND`, реализованном на основе `AsyncResult`, используется размеченное объединение `Result`, позволяющее трактовать вариант `Success` как значение `true`, которое переносится в результат внутренней функции.

Советы по реализации специальных асинхронных комбинаторов

Для создания специальных комбинаторов рекомендуется использовать следующую общую стратегию.

1. Опишите проблему исключительно в терминах конкурентности.
2. Упрощайте описание до тех пор, пока оно не сводится к одному имени.
3. Рассмотрите другие варианты упрощения.
4. Напишите и протестируйте (или импортируйте) конкурентную структуру.

10.6.6. Как работают асинхронные комбинаторы

В листинге 10.26 асинхронно анализировался биржевой код акции и генерировались рекомендации о покупке данной акции. Теперь нам нужно добавить туда условную проверку `if-else`, которая ведет себя асинхронно. Мы сделаем это с помощью комбинатора `ifAsync`: если биржевой опцион рекомендуется покупать, то транзакция будет продолжена; в противном случае будет выдано сообщение об ошибке. Код, на который следует обратить внимание, в листинге 10.29 выделен жирным шрифтом.

В этом примере функция `doInvest` анализирует данный биржевой код, тенденции его изменений и текущее состояние фондового рынка и выдает рекомендацию о целесообразности совершения торговой транзакции. Функция `doInvest` объединяет в себе асинхронные функции, которые выполняются как единое целое и позволяют выдать рекомендацию. Функция `shouldIBuy` с помощью асинхронного логического оператора `OR` проверяет, превышает ли индекс `^IXIC` или `^NYA` заданный порог. Результат используется в качестве базового значения для оценки того, является ли текущее состояние фондового рынка благоприятным для операций покупки.

Если результат функции `shouldIBuy` успешен (`true`), то асинхронный логический оператор `AND` продолжает работу, выполняя функцию `analyHistoricalTrend`, которая возвращает результат анализа истории изменений курса у данной акции. Затем функция `buy` проверяет, достаточно ли остатка на банковском счету для покупки желаемых биржевых опционов; если баланс слишком низкий, то функция возвращает альтернативное значение или ноль.

В итоге все эти функции объединяются. Комбинатор `ifAsync` асинхронно выполняет функцию `shouldIBuy`. В зависимости от ее результатов код разветвляется, чтобы либо продолжить транзакцию покупки, либо выдать сообщение об ошибке. Назначение инфиксного оператора `map (<!>)` состоит в том, чтобы обернуть функцию `buy` в надтип `AsyncResult` и затем выполнить ее для того количества акций, которое было рекомендовано для покупки функцией `howMuchToBuy`.

ПРИМЕЧАНИЕ

Функции, представленные в листинге 10.29, выполняются как элементарная операция, но каждый шаг производится асинхронно по требованию.

Листинг 10.29. Условные комбинаторы AsyncResult

```

let gt (value:'a) (ar:AsyncResult<'a>) = asyncResult {
    let! result = ar
    return result > value
}

let doInvest stockId =
    let shouldIBuy =
        ((getStockIndex "^IXIC" |> gt 6200.0)
         <|||>
         (getStockIndex "^NYA" |> gt 11700.0))
        &&&> ((analyzeHistoricalTrend stockId) |> gt 10.0)
    |> AsyncResult.defaultValue false
    let buy amount = async {
        let! price = getCurrentPrice stockId
        let! result = withdraw (price*float(amount))
        return result |> Result.bimap (fun x -> if x then amount else 0)
                           (fun _ -> 0)
    }
    |> AsyncComb.ifAsync shouldIBuy

```

Применение логического асинхронного оператора OR для выполнения заданных функций. Эта функция представляет собой предикат, который сообщает, стоит ли совершать покупку при текущем состоянии рынка

Проверка того, превышает ли заданное значение результат асинхронной операции, возвращающей тип AsyncResult. Параметрический тип 'a должен быть сравниваемым

Вспомогательная функция, которая возвращает значение по умолчанию для заданного типа, оберывая результат в тип AsyncResult. Эта функция используется при возникновении ошибки в процессе вычислений

Выполнение условного оператора If, позволяющего решить, покупать или не покупать данную акцию

Проверка текущего баланса на банковском счету с возвращением количества акций, которое можно приобрести

Проверка успешности транзакции. Затем возвращается либо Async<int>, в который обернуто значение суммы в случае успешной транзакции, либо ноль

Если результат операции shouldIBuy отрицательный, выводится сообщение об ошибке

Обертывание всех комбинаторов функций в асинхронном перехватчике ошибок

Если результат операции shouldIBuy положительный, то функция покупки оберывается (AsyncResult) и выполняется для заданного количества акций, рекомендованных к покупке. Это значение является результатом функции howMuchToBuy

Резюме

- ❑ Хотите повысить читабельность кода? Тогда ясность ваших намерений имеет решающее значение. Класс `Result` помогает показать, закончился метод неудачно или успешно, позволяет избавиться от ненужного стереотипного кода и приводит к понятной структуре кода.
- ❑ Тип `Result` предоставляет способ явной обработки ошибок в функциональном стиле, без введения побочных эффектов (в отличие от генерирования и перехвата исключений), что приводит к созданию выразительного и легко читаемого кода.
- ❑ В отношении семантики выполнения кода типы `Result` и `Option` играют аналогичную роль, позволяя при выполнении кода учитывать не только «счастливый путь», но и другие варианты. `Result` — лучший тип для использования в тех случаях, когда нужно представить и сохранить ошибку, которая может возникнуть во время выполнения. `Option` лучше подходит для тех случаев, когда нужно представить наличие или отсутствие значения или когда желательно, чтобы потребители учитывали возможность ошибки, но вы не хотите сохранять ее.
- ❑ Функциональное программирование снимает маски с шаблонов, позволяя упростить компоновку асинхронных операций посредством поддержки математических шаблонов. Например, аппликативные функторы, которые являются усиленными вариантами обычных функторов, позволяют комбинировать функции с несколькими аргументами, оперирующие непосредственно надтипами.
- ❑ Асинхронные комбинаторы можно использовать для управления асинхронным потоком выполнения программы. Управление выполнением включает в себя условную логику. Можно легко скомпоновать несколько асинхронных комбинаторов для создания более сложных, таких как асинхронные версии операторов `AND` и `OR`.
- ❑ F# поддерживает инфиксные операторы, которые можно адаптировать и получить удобный набор операторов. Последние упрощают стиль программирования, позволяя легко создавать достаточно сложные нестандартные цепочки операций.
- ❑ Аппликативы и функторы можно объединять, чтобы обертывать обычные функции, которые затем можно применять к надтипам, не выходя за пределы контекста. Эта технология позволяет параллельно запускать несколько гетерогенных функций, результат которых может быть вычислен как единое целое.
- ❑ Использование основных функций функционального программирования, таких как `Bind`, `Return`, `Map` и `Apply`, упрощает определение поведения расширенного кода, в котором приложения компонуются, запускаются параллельно и выполняются для надтипов, имитируя условную логику, такую как `if-else`.

Реактивное программирование с использованием агентов

В этой главе:

- использование конкурентной модели передачи сообщений;
- обработка миллионов сообщений в секунду;
- использование агентной модели программирования;
- распараллеливание рабочего процесса и координация агентов.

Веб-приложения играют в нашей жизни важную роль: от крупных социальных сетей и потоковой передачи мультимедиа до систем интернет-банкинга и командных онлайн-игр. Некоторые сайты сейчас обрабатывают больше трафика, чем весь Интернет десять лет назад. Facebook и Twitter, два самых популярных сайта, насчитывают по несколько миллиардов пользователей. Чтобы обеспечить процветание этих приложений, необходимы конкурентные соединения, масштабируемость и распределенные системы. Традиционные архитектуры прошлых лет не способны работать при таком большом количестве запросов.

Высокопроизводительные вычисления становятся необходимостью. Ответом на это требование стала конкурентная программная модель передачи сообщений, о чем свидетельствует растущая поддержка данной модели в основных языках, таких как Java, C# и C++.

Количество конкурентных онлайн-подключений, безусловно, будет продолжать расти. Тенденция смещается в сторону физических устройств, которые, будучи

взаимосвязанными, образуют сложные и разветвленные сети, круглосуточно работают и обмениваются сообщениями. Предполагается, что к 2025 г. *Интернет вещей* (Internet of Things, IoT) расширится до 75 млрд единиц установленного оборудования (<http://mng.bz/wiwP>).

Что такое Интернет вещей

Интернет вещей, как следует из названия, представляет собой гигантскую сеть вещей (холодильников, стиральных машин и т. п.), подключенных к Интернету. По сути, частью IoT может быть любой предмет, у которого есть выключатель и который можно подключить к Сети. По оценкам одной аналитической компании, к 2020 г. к IoT будет подключено 26 млрд устройств (www.forbes.com/companies/gartner/). По другим оценкам, это число превысит 100 млрд. Это огромное количество передаваемых данных. Одной из проблем IoT является необходимость их передачи в реальном времени без задержек и узких мест, а также постоянное уменьшение времени отклика. Еще одна проблема — безопасность: все, что соединено с Интернетом, открыто для взлома.

Постоянная эволюция устройств, подключенных к Сети, создает условия для революции при разработке приложений следующего поколения. Новые приложения должны быть неблокирующими, быстрыми и способными реагировать на большие объемы системных уведомлений. Выполнение реактивных приложений будет управляться событиями. Нам понадобятся высокодоступные приложения, эффективно использующие ресурсы, способные адаптироваться к быстро меняющимся условиям и реагировать на бесконечно растущий объем интернет-запросов. Парадигмы асинхронности и управления событиями являются основными архитектурными требованиями для разработки таких приложений. В этом контексте необходимо параллельное асинхронное программирование.

Данная глава посвящена разработке адаптивных и быстро реагирующих систем, начиная с программной модели передачи уникальных сообщений, универсальной конкурентной модели с особенно широкой областью применения. Программная модель передачи сообщений имеет несколько общих черт с архитектурой микросервисов (<http://microservices.io/>).

Мы будем использовать стиль агентного конкурентного программирования, который основан на передаче сообщений как средстве связи между небольшими блоками вычислений, называемыми *агентами*. Каждый агент может иметь свое внутреннее состояние с однопоточным доступом, а однопоточный доступ гарантирует потокобезопасность без необходимости блокировок (или каких-либо других примитивов синхронизации). Поскольку агенты просты для понимания, программирование с их помощью является эффективным инструментом для создания масштабируемых и адаптивных приложений, что упрощает реализацию сложной асинхронной логики.

К концу этой главы вы научитесь использовать семантику асинхронной передачи сообщений в своих приложениях, что позволит упростить их, повысить скорость реакции и производительность. (Если вы сомневаетесь в своем понимании асинхрон-

ности, пересмотрите главы 8 и 9.) Прежде чем углубиться в технические аспекты архитектуры передачи сообщений и агентной модели, рассмотрим реактивную систему, обращая особое внимание на свойства, благодаря которым приложение представляет ценность в парадигме реактивного программирования.

11.1. Что такое реактивное программирование и чем оно полезно

Реактивное программирование — это набор принципов проектирования, используемых в асинхронном программировании для создания связанных систем, которые своевременно отвечают на команды и запросы. Это способ рассматривать системную архитектуру и разработку в распределенной среде, где технологии реализации, инструментарий и шаблоны проектирования являются компонентами большего целого — системы. В данной среде приложение разделено на несколько самостоятельных этапов, каждый из которых может выполняться асинхронно, не требуя блокировок. Потоки выполнения, конкурирующие за разделяемые ресурсы, могут выполнять другую полезную работу, пока ресурс занят, вместо того чтобы простоять и зря тратить вычислительную мощность.

В 2013 г. реактивное программирование превратилось в устоявшуюся парадигму с набором формализованных правил под эгидой Реактивного манифеста (*Reactive Manifesto*, www.reactivemanifesto.org/), в котором описывается количество составных частей, определяющих реактивную систему. В Реактивном манифесте определены шаблоны для реализации надежных, отказоустойчивых и отзывчивых систем. Причиной создания Реактивного манифеста стали недавние изменения требований, предъявляемых к приложениям (табл. 11.1).

Таблица 11.1. Требования к приложениям: раньше и сейчас

Требования к приложениям в прошлом	Современные требования к приложениям
Одноядерные процессоры	Многоядерные процессоры
Дорогая память	Дешевая память
Дорогие диски	Дешевые диски
Медленные сети	Быстрые сети
Малое количество конкурентных запросов	Большое количество конкурентных запросов
Мало данных	Большие данные
Задержки в несколько секунд	Задержки в несколько миллисекунд

Раньше в приложениях работало малое количество служб с достаточно большим временем отклика и временем, в течение которого считалось допустимым отключить систему для обслуживания. Сегодня приложения развертываются на тысячах сервисов, каждый из которых может работать на нескольких ядрах. Кроме того, пользователи ожидают, что время отклика измеряется в миллисекундах, а не

в секундах и любое время доступности приложения меньше 100 % является недопустимым. Реактивный манифест стремится разрешить эти проблемы, предлагая разработчикам создавать системы, имеющие такие четыре свойства: отзывчивость (способность реагировать на действия пользователей), устойчивость (умение реагировать на сбои), управляемость сообщениями (реагирование на события) и масштабируемость (реагирование на изменение нагрузки). Эти свойства и взаимосвязи между ними показаны на рис. 11.1.



Рис. 11.1. В соответствии с Реактивным манифестом, чтобы систему можно было назвать реактивной, она должна иметь следующие четыре свойства: быть отзывчивой (реагировать на действия пользователей), устойчивой (реагировать на сбои), управляемой сообщениями (реагировать на события) и масштабируемой (реагировать на изменение нагрузки)

Система, построенная в соответствии с требованиями манифеста, будет:

- иметь постоянное время отклика независимо от рабочей нагрузки;
- реагировать своевременно, независимо от объема входящих запросов. Это гарантирует, что пользователь не будет тратить заметное время на ожидание завершения операций; тем самым обеспечивается его положительный опыт.

Такая отзывчивость возможна благодаря тому, что реактивное программирование оптимизирует применение вычислительных ресурсов на многоядерном оборудовании, что приводит к повышению производительности. Асинхронность является одним из ключевых элементов реактивного программирования. В главах 8 и 9 была описана модель АРМ и то, сколь важна ее роль в создании масштабируемых систем. В главе 14 мы создадим законченное серверное приложение, полностью реализующее эту парадигму.

Архитектура, управляемая сообщениями, является основой реактивных приложений. *Управление сообщениями* означает, что реактивные системы построены на принципе асинхронной передачи сообщений. Более того, в архитектуре, управля-

емой сообщениями, компоненты могут быть слабо связаны. Основное преимущество реактивного программирования состоит в том, что оно устраниет необходимость явной координации между активными компонентами системы, упрощая подход к асинхронным вычислениям.

11.2. Программная модель асинхронной передачи сообщений

В типичном синхронном приложении операции с коммуникационной моделью «запрос — ответ» выполняются последовательно, используя вызов процедуры для извлечения данных или изменения состояния. Этот шаблон является ограниченным вследствие блокирующего стиля и структуры, которую нельзя масштабировать или выполнять не по порядку.

Архитектура на основе передачи сообщений — форма асинхронной коммуникации, при которой данные помещаются в очередь для последующей обработки, если это необходимо. В контексте реактивного программирования в архитектуре передачи сообщений используется асинхронная семантика для обмена данными между отдельными частями системы. В результате приложение может обрабатывать миллионы сообщений в секунду, обеспечивая невероятный прирост производительности (рис. 11.2).

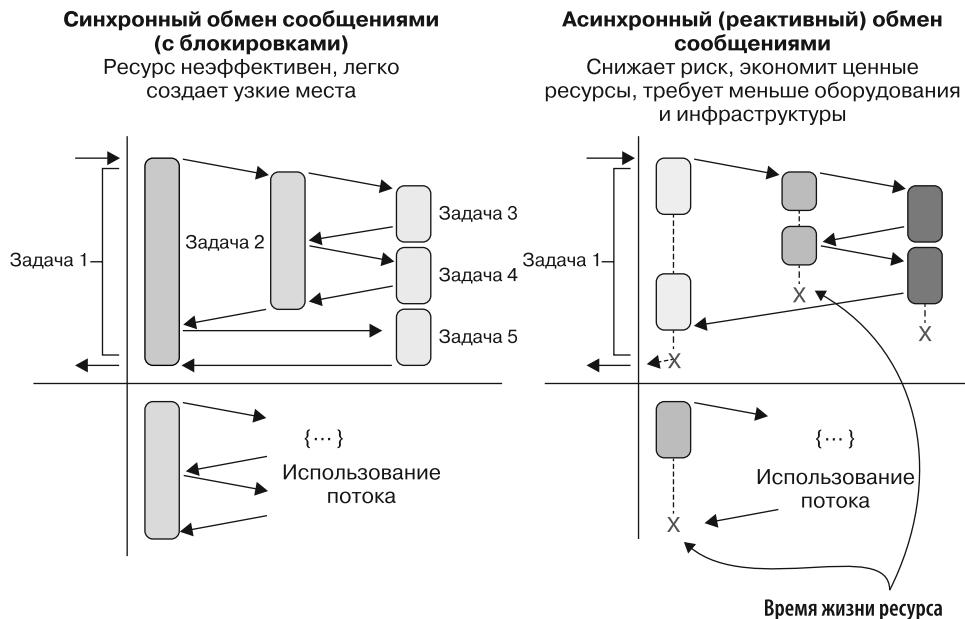


Рис. 11.2. Синхронный (блокирующий) обмен данными неэффективен с точки зрения ресурсов и легко создает узкие места. Подход с асинхронной передачей сообщений (реактивный) снижает риски блокировок, экономит ценные ресурсы, требует меньше оборудования и инфраструктуры

ПРИМЕЧАНИЕ

Модель передачи сообщений становится все более популярной и реализована во многих новых языках программирования, часто в качестве основной концепции. Во многих других языках программирования эта модель доступна посредством сторонних библиотек, основанных на традиционной многопоточности.

Идея конкурентного обмена сообщениями основана на простых вычислительных единицах (или процессах), которые имеют исключительные права владения состоянием. По своему замыслу состояние является защищенным и неразделяемым, то есть может быть изменяемым или неизменяемым, не создавая при этом потенциальных ловушек из-за многопоточной среды (см. главу 1). В архитектуре передачи сообщений две сущности, отправитель и получатель сообщения, работают в отдельных потоках. Преимущество этой модели программирования заключается в том, что все вопросы совместного использования и конкурентного доступа к памяти скрыты внутри канала связи. Никакая сущность, участвующая в обмене данными, не должна применять какие-либо стратегии синхронизации низкого уровня, такие как блокировка. Архитектура передачи сообщений (конкурентная модель передачи сообщений) не предусматривает обмен данными посредством совместного использования памяти; вместо этого обмен ими происходит посредством отправки сообщений.

Асинхронная передача сообщений разделяет коммуникацию между объектами и позволяет отправителям передавать сообщения, не дожидаясь получателей этих сообщений. Для обмена сообщениями не требуется синхронизации между отправителями и получателями; обе сущности могут работать независимо одна от другой. Помните, что отправитель не знает, когда сообщение получено и обработано получателем.

Конкурентная модель передачи сообщений поначалу может показаться более сложной, чем последовательные и даже параллельные системы, как видно на рис. 11.3, где показано сравнение всех трех систем (квадраты соответствуют объектам, а стрелки — вызову методов или передаче сообщений).

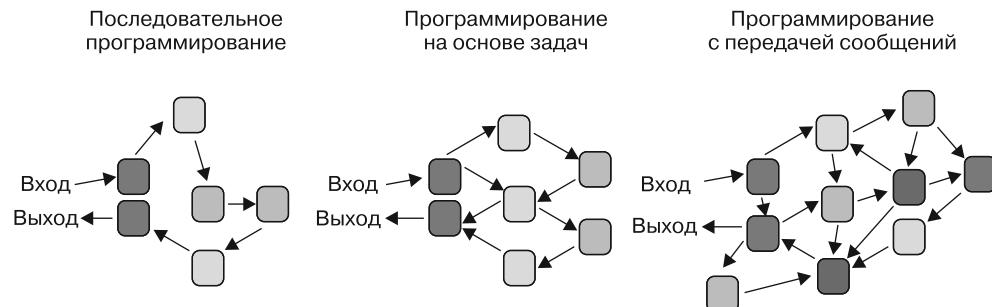


Рис. 11.3. Сравнение программирования на основе задач последовательного и агентного программирования. Каждый блок представляет собой единицу вычислений

На рис. 11.3 каждый блок представляет собой элементарную операцию.

- ❑ Последовательное программирование является самым простым вариантом; у него только один вход, и оно создает только один выход с использованием единственного потока управления, в котором блоки соединяются напрямую, линейно, причем каждая задача зависит от выполнения предыдущей задачи.
- ❑ Программирование на основе задач подобно модели последовательного программирования, но позволяет выполнять шаблоны MapReduce или Fork/Join в потоке управления.
- ❑ Программирование на основе передачи сообщений позволяет управлять потоком выполнения, поскольку одни блоки непрерывно и напрямую связаны с другими блоками. В конечном счете каждый блок отправляет сообщения другим блокам напрямую, нелинейно. На первый взгляд такая структура может показаться сложной и трудной для понимания. Но поскольку блоки инкапсулированы внутри активных объектов, то каждое сообщение передается независимо от других сообщений без блокировок и задержек. Благодаря конкурентной модели передачи сообщений можно создать много строительных блоков, у каждого из которых будет свой независимый вход и выход; посредством этих входов и выходов блоки подключаются друг к другу. Каждый работает изолированно, и благодаря изоляции можно распределить вычисления между разными задачами.

Оставшаяся часть главы посвящена агентам как основному инструменту для построения конкурентных моделей передачи сообщений.

11.2.1. Передача сообщений и неизменяемость

К этому моменту вам уже должно быть ясно, что неизменяемость обеспечивает повышенный уровень конкурентности. (Напомню: неизменяемый объект — это объект, состояние которого нельзя изменить после его создания.) Неизменяемость — это основополагающий инструмент для построения конкурентных, надежных и предсказуемых программ. Но это не единственный инструмент, который имеет значение. Так же критически важна естественная изоляция — возможно, даже еще важна, потому что ее легче достичь в языках программирования, которые по своей природе не поддерживают неизменяемость. Оказывается, что агенты обеспечивают грубую изоляцию посредством передачи сообщений.

11.2.2. Естественная изоляция

Естественная изоляция является критически важной концепцией для написания неблокирующего конкурентного кода. В многопоточной программе изоляция решает проблему разделяемых состояний, предоставляя каждому потоку скопированную часть данных для выполнения локальных вычислений. Благодаря изоляции не возникает состояние гонки, поскольку каждая задача обрабатывает собственную, независимую копию данных.

Построение отказоустойчивых систем посредством изоляции

Изоляция является аспектом построения устойчивых систем. Например, в случае отказа одного из компонентов остальная часть системы, по-видимому, окажется не-восприимчивой к этому отказу. Передача сообщений оказывает огромную помощь в упрощении процесса построения корректных конкурентных систем, и это становится возможным благодаря принципу изоляции, также называемому *подходом без разделения ресурсов*.

Естественную изоляцию, или подход без разделения ресурсов, проще обеспечить, чем неизменяемость, но оба варианта представляют собой независимые подходы и должны использоваться совместно для сокращения издержек выполнения, предотвращения состояния гонки и взаимных блокировок.

11.3. Что такое агент

Агент – это однопоточная вычислительная единица, используемая для разработки конкурентных приложений, основанных на изолированной передаче сообщений (подход без разделения ресурсов). Агенты являются облегченными конструкциями, которые содержат запрос и могут получать и обрабатывать сообщения. В данном случае «облегченный» означает, что агенты имеют небольшой объем памяти по сравнению с порождением новых потоков, поэтому можно без проблем создать 100 000 агентов на одном компьютере.

Агент можно представить как процесс, который обладает исключительным правом владения каким-либо изменяемым состоянием; к этому состоянию невозможно получить доступ иначе, как через агента. Агенты взаимодействуют между собой конкурентно, но *внутри* агента все выполняется последовательно. Изоляция внутреннего состояния агента является ключевой концепцией данной модели; это состояние совершенно недоступно для внешнего мира, что делает его потокобезопасным. Действительно, если состояние изолировано, то его можно изменять как угодно.

Базовая функциональность агента заключается в следующем:

- ❑ в поддержке внутреннего состояния, к которому можно безопасно обращаться в многопоточной среде;
- ❑ разной реакции на сообщения в разных состояниях;
- ❑ отправке уведомлений другим агентам;
- ❑ отправке событий подписчикам;
- ❑ отправке ответов отправителям сообщений.

Одним из самых важных моментов агентного программирования является то, что сообщения отправляются асинхронно и отправитель не инициирует блокировку. Когда сообщение отправляется агенту, оно помещается в почтовый ящик. Агент обрабатывает сообщения по одному, последовательно, в том порядке, в котором

эти сообщения были помещены в почтовый ящик, переходя к следующему только после того, как завершена обработка предыдущего. Пока агент обрабатывает одно сообщение, другие входящие сообщения не теряются, а буферизуются во внутреннем изолированном почтовом ящике. Таким образом, несколько агентов могут без проблем работать параллельно; это означает, что производительность хорошо написанного агентного приложения масштабируется в зависимости от количества ядер или процессоров.

Агент — не актор

На первый взгляд может показаться, что между агентами и актерами есть сходства, что иногда заставляет людей использовать эти термины взаимозаменяющими. Но главное различие заключается в том, что агенты находятся *внутри* процесса, а акторы могут действовать *в другом* процессе. Фактически агент означает конкретную сущность, тогда как актор отличается прозрачностью расположения. *Прозрачность расположения* — идентификация сетевых ресурсов по именам, а не по их фактическому расположению; это означает, что актор может выполняться как в том же, так и в другом процессе и даже на удаленной машине.

Агентная конкурентность построена по принципам модели акторов, но ее строение намного проще. Системы акторов имеют встроенные сложные инструменты для поддержки распределения, в том числе контроль управления исключениями и, возможно, самовосстановление системы, маршрутизацию для настройки распределения работы и многое другое.

В экосистеме .NET модель акторов реализована в виде нескольких библиотек и наборов инструментов, таких как Akka.net (<http://getakka.net/>), Proto.Actor (<http://proto.actor/>) и Microsoft Orleans (<https://dotnet.github.io/orleans/>). Неудивительно, что платформа Microsoft Azure Service-Fabric (<https://azure.microsoft.com/en-us/services/service-fabric>), используемая для создания распределенных, масштабируемых и отказоустойчивых облачных микросервисов, основана именно на модели акторов. Для получения дополнительной информации о модели акторов в .NET я рекомендую книгу Энтони Брауна (Anthony Brown) *Reactive Applications with Akka.Net* (Manning, 2017).

Инструменты и функции, предоставляемые библиотеками акторов, можно легко реализовать и воспроизвести для агентов. Есть несколько библиотек, восполняющих отсутствующую функциональность, такую как контроль и маршрутизация (<http://mbrace.io/> и <http://akka.net>).

11.3.1. Компоненты агента

На рис. 11.4 показаны следующие основные составные части агента.

- ❑ *Почтовый ящик* — внутренняя очередь для буферизации входящих сообщений, реализованная как асинхронная, без состояния гонки и блокировок.
- ❑ *Поведение* — внутренняя функция, последовательно применяемая к каждому входящему сообщению. Поведение является однопоточным.

- *Состояние* — агент может иметь внутреннее состояние. Внутреннее состояние является изолированным и закрыто для совместного использования, поэтому никогда не является предметом конкуренции за доступ и не подвержено блокировкам.
- *Сообщение* — агенты могут общаться только посредством сообщений, которые отправляются асинхронно и буферизуются в почтовом ящике.

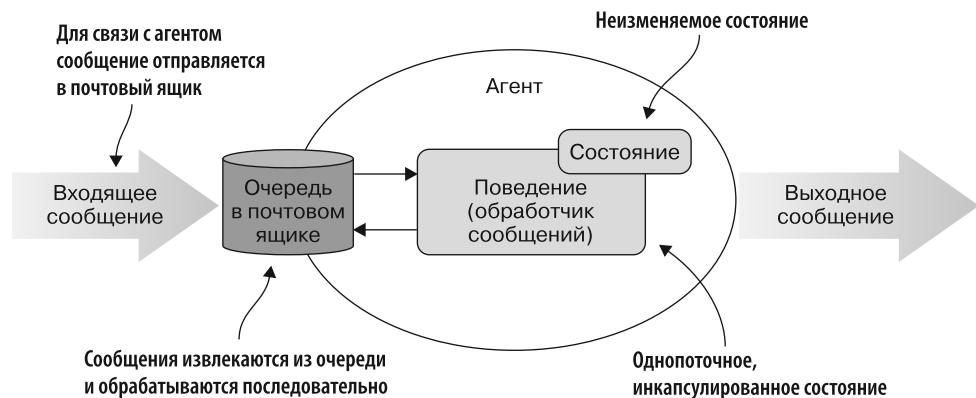


Рис. 11.4. Агент состоит из почтового ящика, в котором содержится очередь входящих сообщений, состояния и поведения. Это поведение в цикле обрабатывает сообщения по одному за раз. Поведение — это функциональность, применяемая к сообщениям

11.3.2. Что может делать агент

Модель агентного программирования обеспечивает отличную поддержку конкурентности и имеет широкий спектр применения. Агенты используются при сборе и анализе данных, уменьшая количество узких мест в приложениях путем буферизации запросов; при анализе в реальном времени с ограниченной и неограниченной реактивной потоковой передачей; при решении числовых задач общего назначения, машинном обучении и моделировании, в шаблонах Master/Worker, Compute Grid, MapReduce, компьютерных играх, обработке звука и видео — и это далеко не все.

11.3.3. Подход без разделения ресурсов для конкурентного программирования без блокировок

Архитектура без разделения ресурсов относится к программированию на базе обмена сообщениями, где каждый агент является независимым и в системе нет ни одной точки разногласий. Такая модель архитектуры отлично подходит для построения конкурентных и безопасных систем. Если в системе нет разделяемых ресурсов, то

нет и возможностей для создания состояния гонки. Изолированные блоки передачи сообщений (агенты) являются мощным и эффективным способом для реализации масштабируемых алгоритмов программирования, включая масштабируемые серверы запросов и масштабируемые алгоритмы распределенного программирования. Простота и интуитивно понятное поведение агентов как строительных блоков позволяют проектировать и реализовывать изящные, высокоэффективные асинхронные и параллельные приложения, в которых отсутствуют разделяемые состояния. Как правило, агенты выполняют вычисления в ответ на полученные сообщения и могут отправлять сообщения другим агентам по принципу «отправил и забыл» или собирать отклики, называемые *ответами* (рис. 11.5).

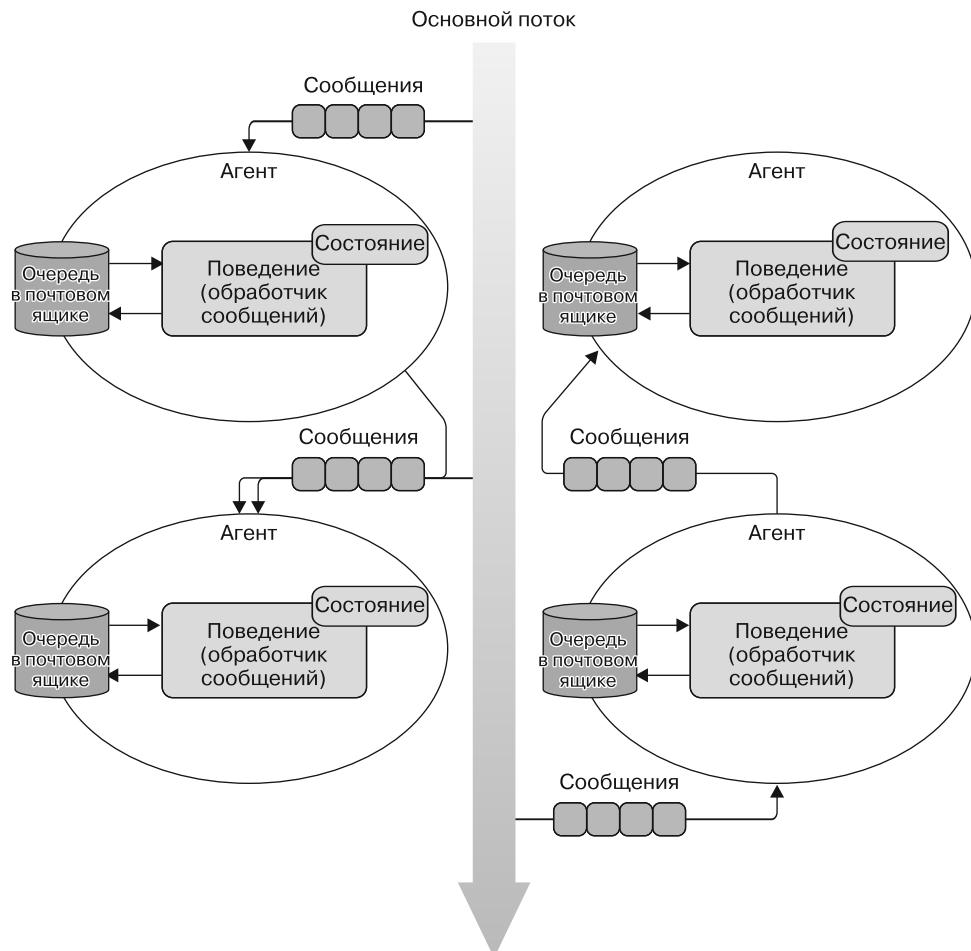


Рис. 11.5. Агенты взаимодействуют друг с другом посредством семантики передачи сообщений, создавая взаимосвязанную систему вычислительных единиц, которые выполняются конкурентно. Каждый агент имеет изолированное состояние и независимое поведение

11.3.4. Как функционирует агентное программирование

Некоторые аспекты агентного программирования не являются функциональными. Несмотря на то что агенты (и акторы) были разработаны в контексте функциональных языков программирования, их цель — генерировать побочные эффекты, что противоречит принципам ФП. Агент часто выполняет побочный эффект или отправляет сообщение другому агенту, который, в свою очередь, выполняет другой побочный эффект.

Менее важно, но тоже стоит отметить, что функциональное программирование в целом отделяет логику от данных. Но агенты содержат и данные, и логику функций обработки. Кроме того, отправка сообщения агенту не накладывает никаких ограничений на тип возвращаемого значения. Поведение агента, которое является операцией, применяемой к каждому сообщению, может либо возвращать результат, либо не возвращать никакого результата. В последнем сценарии структура сообщения, отправляемого в режиме «отправил и забыл», поощряет программных агентов использовать шаблон одностороннего потока, что означает, что сообщения передаются от одного агента следующему. Этот односторонний поток сообщений между агентами может сохранять их композиционно-семантический аспект, достигаемый путем связывания заданного набора агентов. В результате получается конвейер агентов, состоящий из этапов операций по обработке сообщений, каждый из них выполняется независимо и, возможно, параллельно.

Основная причина, по которой модель агента является функциональной, заключается в том, что агенты *могут передавать поведение состоянию, вместо того чтобы передавать состояние поведению*. В агентной модели отправитель, кроме передачи сообщений, может предоставить функцию, реализующую действие по обработке входящих сообщений. Агент — это слот памяти, в который можно поместить структуру данных, такую как сегмент (контейнер). Помимо хранения данных, агенты позволяют отправлять сообщения в виде функции, затем атомарно применяемой к внутреннему сегменту.

ПРИМЕЧАНИЕ

Атомарным называется набор операций, которые, будучи запущенными, должны завершаться до любого прерывания, за один шаг, так что другие параллельные потоки могут видеть одно из двух состояний: либо старое, либо новое.

Функцию можно скомпоновать из других функций и затем отправить агенту в виде сообщения. Преимущество заключается в возможности обновления и изменения поведения во время выполнения, используя функции и их компоновку в соответствии с функциональной парадигмой.

11.3.5. Агент является объектно-ориентированным

Интересно отметить, что оригинальное видение Алана Кэя (Alan Kay, https://ru.wikipedia.org/wiki/Кэй,_Алан_Кёртис) для объектов в Smalltalk гораздо ближе к агентной модели, чем к объектам, встречающимся в большинстве языков программиро-

вания (например, базовая концепция «сообщений»). Кэй считал, что изменения состояний должны быть инкапсулированы и не должны выполняться неограниченно. Его идея передачи сообщений между объектами интуитивно понятна и помогает объяснить границы между объектами.

Ясно, что передача сообщений напоминает ООП, и можно полагаться на передачу сообщений в стиле ООП, при которой только вызывается метод. Здесь агент подобен объекту в объектно-ориентированной программе, поскольку он инкапсулирует состояние и осуществляет коммуникацию с другими агентами посредством обмена сообщениями.

11.4. Агенты в F#: MailboxProcessor

Поддержка АРМ в F# не ограничивается асинхронными рабочими процессами (описанными в главе 9). Язык программирования F#, кроме того, поддерживает и другие средства, в том числе `MailboxProcessor` — примитивный тип, который ведет себя как простой размещаемый в памяти агент передачи сообщений (рис. 11.6).

`MailboxProcessor` работает полностью асинхронно и представляет собой простую конкурентную программную модель, которая позволяет создавать быстрые и надежные конкурентные программы. Я мог бы написать целую книгу о `MailboxProcessor`, множество вариантов его использования и гибкости, которую он обеспечивает для создания разнообразных приложений. Преимущества применения `MailboxProcessor` включают в себя выделенную и изолированную очередь сообщений в сочетании с асинхронным обработчиком, который задействуется для регулирования обработки сообщений и обеспечивает автоматическую и прозрачную оптимизацию использования ресурсов компьютера.

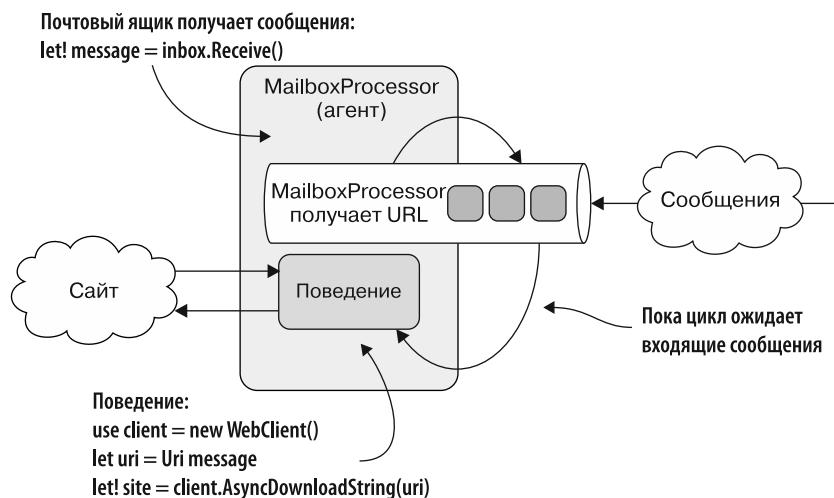


Рис. 11.6. `MailboxProcessor` (агент) асинхронно ожидает входящие сообщения в цикле `while`. Сообщения представляют собой строки, содержащие URL, которые передаются внутреннему поведению для загрузки соответствующего сайта

В листинге 11.1 показан простой пример кода с использованием `MailboxProcessor`, который получает произвольный URL-адрес и выдает размер сайта.

Листинг 11.1. Простой MailboxProcessor с циклом while

```
type Agent<'T> = MailboxProcessor<'T>

let webClientAgent =
    Agent<string>.Start(fun inbox -> async {
        while true do
            let! message = inbox.Receive() ← Метод MailboxProcessor.Start
            use client = new WebClient() ← возвращает работающий агент
            let uri = Uri message
            let! site = client.AsyncDownloadString(uri) ← Асинхронное ожидание
            printfn "Size of %s is %d" uri.Host site.Length ← получения сообщения
            print "Size of %s is %d" uri.Host site.Length ← Загрузка данных посредством
        }) ← асинхронного рабочего процесса

agent.Post "http://www.google.com" | Отправка сообщения в MailboxProcessor
agent.Post "http://www.microsoft.com" | в режиме «отправил и забыл»
```

Рассмотрим, как создать агента на F#. Во-первых, у экземпляра агента должно быть имя. В данном случае `webClientAgent` — это адрес обработчика почтового ящика. Именно туда мы отправляем сообщение для обработки. `MailboxProcessor` обычно инициализируется с помощью ускоренного метода `MailboxProcessor.Start`, хотя можно также создать экземпляр, вызвав конструктор напрямую, а затем запустить агента, используя метод экземпляра класса `Start`. Чтобы упростить имя и использование `MailboxProcessor`, мы назначим агенту псевдоним `Agent`, а затем запустим агента с помощью `Agent.Start`.

Далее, существует лямбда-функция с входящим почтовым ящиком, содержащим асинхронный рабочий процесс. Каждое сообщение, отправленное обработчику почтового ящика, передается асинхронно. Тело агента играет роль обработчика сообщений, который принимает в качестве аргумента почтовый ящик (`inbox:MailboxProcessor`). Этот почтовый ящик имеет работающий логический поток, управляющий выделенной и инкапсулированной очередью сообщений. Такая очередь является потокобезопасной, чтобы использовать обмен данными и координировать его с другими потоками или агентами. Почтовый ящик работает асинхронно, задействуя асинхронный рабочий процесс F#. Этот процесс может содержать длительные операции, которые не блокируют поток.

Как правило, сообщения должны обрабатываться по порядку, поэтому нам нужен цикл. В нашем примере используется нефункциональный цикл в стиле `while-true`. В данном случае можно задействовать как его, так и функциональный, рекурсивный цикл. Агент, представленный в листинге 11.1, начинает получать и обрабатывать сообщения, вызывая асинхронную функцию `agent.Receive()` с помощью конструкции `let!`, расположенной внутри императивного цикла `while`.

Внутри цикла располагается сердце обработчика почтового ящика. Вызов функции почтового ящика `Receive` ожидает входящее сообщение, не блокируя действи-

ющий поток, и возобновляет работу после получения сообщения. Использование оператора `let!` гарантирует, что вычисление начнется немедленно.

Затем первое доступное сообщение удаляется из очереди почтового ящика и привязывается к идентификатору сообщения. В этот момент агент реагирует, обрабатывая сообщение, — в нашем примере загружает и печатает размер данного адреса сайта. Если очередь в почтовом ящике пуста и сообщений для обработки нет, то агент освобождает поток и поток помещается обратно в планировщик пула потоков. Это означает, что никакие потоки не находятся в режиме ожидания, пока функция `Receive` ожидает входящих сообщений, которые отправляются в `MailboxProcessor` методом «отправил и забыл» с помощью метода `agent.Post`.

Асинхронный рекурсивный цикл для обработки почтового ящика. В предыдущем примере почтовый ящик агента асинхронно ожидает получения сообщений, используя императивный цикл `while`. Изменим императивный цикл так, чтобы в нем применялась функциональная рекурсия, можно было бы избежать изменяемости и хранить локальные состояния.

В листинге 11.2 представлена та же версия агента, который подсчитывает сообщения, что и в листинге 11.1, но на этот раз агент использует рекурсивную асинхронную функцию, которая обрабатывает состояние.

Листинг 11.2. Простой MailboxProcessor с рекурсивной функцией

```
let agent = Agent<string>.Start(fun inbox ->
    let rec loop count = async {
        let! message = inbox.Receive()
        use client = new WebClient()
        let uri = Uri message
        let! site = client.AsyncDownloadString(uri)
        printfn "Size of %s is %d - total messages %d" uri.Host
        site.Length (count + 1)
        return! loop (count + 1) } )
    loop 0)
agent.Post "http://www.google.com"
agent.Post "http://www.microsoft.com"
```

Использование асинхронной рекурсивной функции, которая обрабатывает состояние в неизменяемой форме

Рекурсивная функция — хвостовой вызов, асинхронно передающий обновленное состояние

Такой функциональный подход немного сложнее, но он значительно сокращает количество явных изменений в коде и часто носит более общий характер. Фактически, как мы вскоре увидим, эту же стратегию можно использовать для обработки и безопасного повторного применения состояния при кэшировании.

Обратите особое внимание на строку кода `return! loop (n + 1)`, где цикл организован посредством рекурсивного вызова асинхронных рабочих процессов с передачей увеличенного значения счетчика. Вызов с помощью оператора `return!` представляет собой хвостовую рекурсию. Это означает, что компилятор транслирует рекурсию более эффективно, и это позволяет избежать исключений переполнения стека. Подробнее поддержка рекурсивных функций (в том числе на C#) описана в главе 3.

Важнейшие функции MailboxProcessor

Ниже перечислены самые важные функции `MailboxProcessor`.

- `Start` — эта функция определяет асинхронный обратный вызов, который формирует цикл обработки сообщений.
- `Receive` — асинхронная функция для получения сообщений из внутренней очереди.
- `Post` — отправляет сообщение в `MailboxProcessor` в режиме «отправил и забыл».

11.5. Как избежать узких мест при обращении к базе данных с помощью F#-типа `MailboxProcessor`

Основной функцией большинства приложений является доступ к базе данных, который часто является главной причиной образования узких мест в коде. Простая настройка производительности базы данных может значительно ускорить работу приложений и обеспечить доступность сервера.

Как гарантировать стабильно высокую пропускную способность при обращении к базе данных? Чтобы улучшить к ней доступ, необходимо сделать операцию асинхронной вследствие природы ввода-вывода при обращении к базе данных. Асинхронность гарантирует, что сервер сможет обрабатывать несколько запросов параллельно. Сколько параллельных запросов способен обработать сервер баз данных без снижения производительности (на рис. 11.7 показано снижение производительности при большом количестве запросов)? Точного ответа на такой вопрос не существует. Это зависит от множества различных факторов: например, от размера пула соединений с базой данных.

Критически важным элементом проблемы узких мест являются контроль и ограничение количества входящих запросов для достижения максимальной производительности приложения. `MailboxProcessor` обеспечивает решение этой проблемы путем буферизации входящих сообщений и обработки возможной ситуации с переполнением стека запросов (рис. 11.8). Использование `MailboxProcessor` в качестве механизма ограничения количества операций с базой данных обеспечивает детальный контроль для оптимизации применения пула соединений с базой данных. Например, программа может добавлять или удалять агенты для выполнения операций с базой данных, точно регулируя уровень параллелизма.

В листинге 11.3 показана полностью асинхронная F#-функция. Она обращается к указанной базе данных и инкапсулирует запрос в теле `MailboxProcessor`. Инкапсуляция операции в качестве поведения агента гарантирует, что запросы к базе данных будут обрабатываться последовательно, по одному.

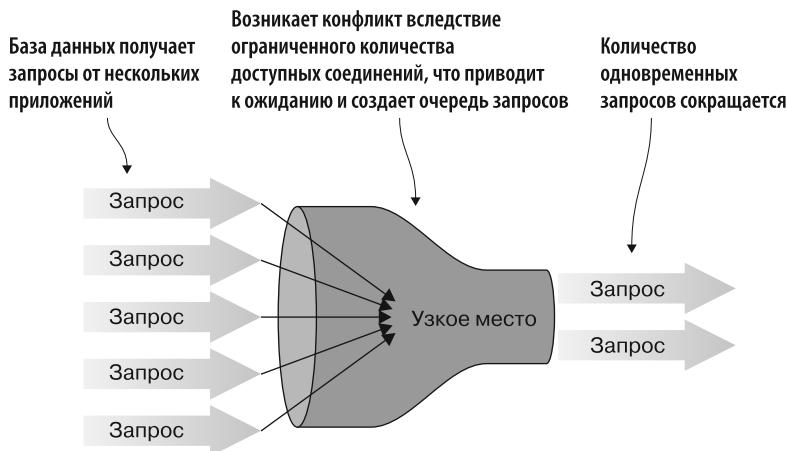


Рис. 11.7. Большое количество конкурентных запросов доступа к базе данных сокращается вследствие ограниченного размера пула соединений

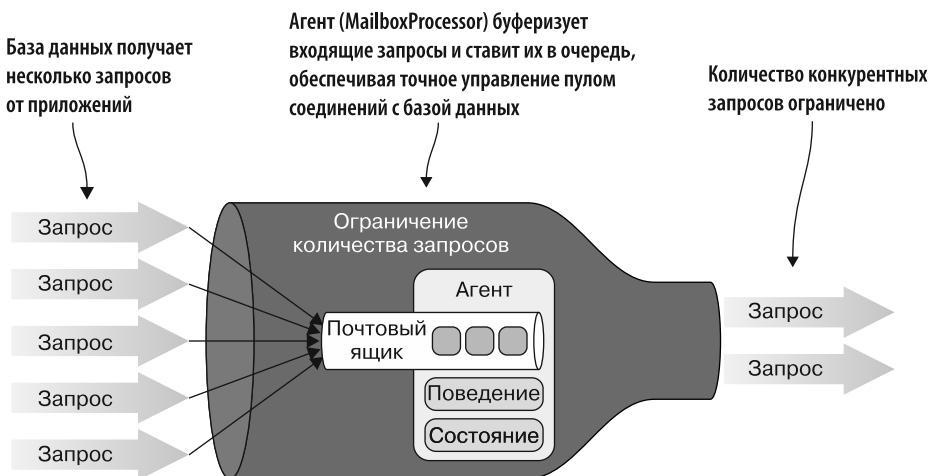


Рис. 11.8. Агент (MailboxProcessor) управляет входящими запросами, оптимизируя использование пула соединений с базой данных

СОВЕТ

Одно из очевидных решений для обработки большего количества запросов — установить максимальный размер пула соединений с базой данных. Но это не очень хорошая практика. Часто ваше приложение является не единственным клиентом, подключенным к базе данных, и если оно займет все соединения, то сервер базы данных не сможет работать так, как ожидалось.

Листинг 11.3. Использование MailboxProcessor для управления обращениями к базе данных

```

Использование размеченного объединения
с одним значением для определения
сообщения MailboxProcessor

Использование типа записи Person

type Person =
    { id:int; firstName:string; lastName:string; age:int }

type SqlMessage =
    | Command of id:int * AsyncReplyChannel<Person option>

let agentSql connectionString =
    fun (inbox: MailboxProcessor<SqlMessage>) ->
        let rec loop() = async {
            let! Command(id, reply) = inbox.Receive()
                use conn = new SqlConnection(connectionString)
                use cmd = new SqlCommand("Select FirstName, LastName, Age
                    from db.People where id = @id")
                cmd.Connection <- conn
                cmd.CommandType <- CommandType.Text
                cmd.Parameters.Add("@id", SqlDbType.Int).Value <- id
                if conn.State <> ConnectionState.Open then
                    do! conn.OpenAsync()
                    use! reader = cmd.ExecuteReaderAsync(
                        CommandBehavior.SingleResult ||| CommandBehavior.CloseConnection)
                    let! canRead = (reader:SqlDataReader).ReadAsync()
                    if canRead then
                        let person =
                            { id = reader.GetInt32(0)
                                firstName = reader.GetString(1)
                                lastName = reader.GetString(2)
                                age = reader.GetInt32(3) }
                        reply.Reply(Some person)
                    else reply.Reply(None)
                    return! Loop()
            }
            loop()
        }

type AgentSql(connectionString:string) =
    let agentSql = new MailboxProcessor<SqlMessage>
        (agentSql connectionString)

    member this.ExecuteAsync (id:int) =
        agentSql.PostAndAsyncReply(fun ch -> Command(id, ch))

    member this.ExecuteTask (id:int) =
        agentSql.PostAndAsyncReply(fun ch -> Command(id, ch))
        |> Async.StartAsTask

```

Асинхронное открытие SQL-соединения с использованием оператора асинхронного рабочего процесса do!

Сопоставление полученного сообщения с образцом и деконструирование его для доступа к внутренним значениям

Если SQL-команда может быть выполнена, то формируется ответ вызывающей стороне и возвращается результат операции Some

Если SQL-команда не может быть выполнена, то формируется ответ вызывающей стороне с результатом None

API для взаимодействия синкапсулированным MailboxProcessor

Для доступа к базе данных используется традиционный Access-Data-Object (ADO) из .NET. Можно также задействовать Microsoft Entity Framework или любое другое средство доступа к данным по вашему выбору. Описание того, как обращаться к компоненту доступа к данным Entity Framework, выходит за рамки этой книги. Для получения дополнительной информации обратитесь к онлайн-документации MSDN по адресу <http://mng.bz/4sdU>.

Первоначально структура данных `Person` определена как тип записи, который можно легко использовать на любом языке программирования .NET как неизменяемый класс. Функция `agentSql` определяет тело `MailboxProcessor`, поведение которого принимает сообщения и асинхронно выполняет запросы к базе данных. Благодаря применению типа `Option` для значения `Person`, которое в противном случае было бы равно `null`, приложение становится более устойчивым. Это помогает предотвратить исключения с нулевой ссылкой.

Тип `AgentSql` инкапсулирует `MailboxProcessor`, создаваемый при выполнении функции `agentSql`. Доступ к внутреннему агенту обеспечивается благодаря методам `ExecuteAsync` и `ExecuteTask`.

Назначением метода `ExecuteTask` является улучшение взаимодействия с C#. Тип `AgentSql` можно скомпилировать в виде библиотеки F# и распространять как многократно используемый компонент. Для того чтобы использовать этот компонент на C#, нужно также предоставить методы, возвращающие тип `Task` или `Task<T>` для функций F#, которые запускают объект асинхронного рабочего процесса (`Async<'T>`). Организация взаимодействия между типами F# `Async` и .NET `Task` описана в приложении B.

11.5.1. Тип сообщения `MailboxProcessor`: размеченные объединения

`type SqlMessage Command` — это размеченное объединение с единственным значением, используемое для отправки сообщений в `MailboxProcessor` с четко определенным типом, который может сопоставляться с образцом:

```
type SqlMessage =
    | Command of id:int * AsyncReplyChannel<Person option>
```

Обычная для F# практика заключается в использовании размеченных объединений для определения разных типов сообщений, которые может принимать `MailboxProcessor`, и сопоставления их с образцами, чтобы деконструировать и извлечь внутреннюю структуру данных (подробнее об F# читайте в приложении B). Сопоставление размеченных объединений с образцом обеспечивает точный и лаконичный способ обработки сообщений. Обычной практикой являются вызов `inbox.Receive()` или `inbox.TryReceive()` и последующая проверка соответствия содержимого сообщения.

Как повысить производительность при использовании размеченного объединения с единственным значением на F#

Использование размеченных объединений с единственным значением (как показано в листинге 11.3) для обертывания примитивных значений считается эффективным способом написания программ. Но, поскольку объединения компилируются как классы, следует ожидать снижения производительности. Это падение производительности является платой за размещение в памяти и последующую утилизацию класса сборщиком мусора. Начиная с F# 4.1, появилось более эффективное решение: декорировать размеченные объединения атрибутом `Struct`, что позволяет компилятору обрабатывать эти типы как значения, избегая излишнего выделения памяти в куче и снижая нагрузку на сборщик мусора.

Использование строго типизированных сообщений позволяет поведению `MailboxProcessor` различать типы сообщений и предоставлять разные коды обработки в зависимости от типа сообщения.

11.5.2. Двусторонний обмен данными посредством `MailboxProcessor`

В листинге 11.3 внутренний `MailboxProcessor` возвращает (отвечает) вызывающему объекту результат обращения к базе данных в форме типа параметра `Person`. В этом обмене данными используется тип `AsyncReplyChannel<'T>`, который определяет механизм, применяемый для ответа на параметр канала, установленный во время инициализации сообщения (рис. 11.9).

Код, который может асинхронно ожидать ответа, использует `AsyncReplyChannel`. После завершения вычислений функция `Reply` позволяет вернуть результаты из почтового ящика:

```
type SqlMessage =
    | Command of id:int * AsyncReplyChannel<Person option>

    member this.ExecuteAsync (id:int) =
        agentSql.PostAndAsyncReply(fun ch -> Command(id, ch))
```

Метод `PostAndAsyncReply` инициализирует канал для логики `Reply`, которая передает агенту канал для ответа как часть сообщения, используя анонимную лямбда-функцию. На этом этапе рабочий процесс приостанавливается (без блокировки) до тех пор, пока операция не завершится и агент не передаст по каналу функцию `Reply`, содержащую результат, назад вызывающему объекту:

```
reply.Reply(Some person)
```

Рекомендуется встроить обработчик `AsyncReplyChannel` в само сообщение, как показано в размеченном объединении `SqlMessage.Command of id: int * AsyncReplyChannel<Person option>`, поскольку процесс ответа на отправленное сообщение может быть легко реализован компилятором.

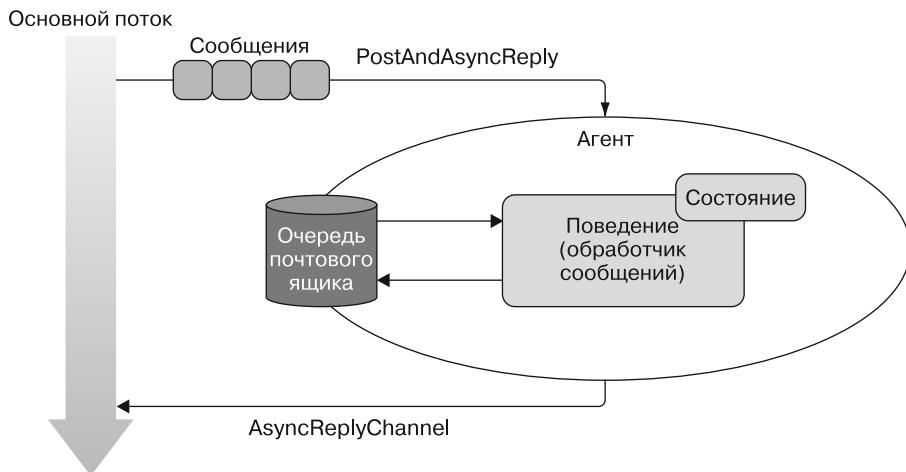


Рис. 11.9. При двусторонней коммуникации с агентом генерируется AsyncReplyChannel, который используется агентом в качестве обратного вызова для уведомления вызывающего объекта о завершении вычислений и — обычно — для предоставления результата

Вы спросите: зачем использовать `MailboxProcessor` для обработки нескольких запросов, если сообщения все равно обрабатываются по одному? Не потеряются ли входящие сообщения, если `MailboxProcessor` будет занят?

Отправка сообщений в `MailboxProcessor` — всегда неблокирующая операция; но с точки зрения агента получение сообщений является блокирующей операцией. Даже при отправке агенту нескольких сообщений ни одно из них не будет потеряно, поскольку сообщения буферизуются и помещаются в очередь почтового ящика.

Также можно реализовать семантику выборочного приема и сканировать входящие сообщения (<http://mng.bz/1lJr>) в поиске точных соответствий заданному типу. В зависимости от поведения агента обработчик может ожидать поступления в почтовый ящик определенного сообщения и временно откладывать другие. Эта технология используется для реализации конечных автоматов с возможностью продолжения после паузы.

11.5.3. Использование AgentSql из C#

Теперь мы воспользуемся `AgentSql`, чтобы код можно было применять в других языках. API, предоставляемые `AgentSql`, подходят и для `Task` в C#, и для асинхронного рабочего процесса F#.

Используя C#, задействовать `AgentSql` легко. После обращения к библиотеке F#, содержащей `AgentSql`, можно создать экземпляр объекта и затем вызвать метод `ExecuteTask`:

```
AgentSql agentSql = new AgentSql("<< ConnectionString Here >>");  
Person person = await agentSql.ExecuteTask(42);  
Console.WriteLine($"Fullname {person.FirstName} {person.LastName}");
```

`ExecuteTask` возвращает `Task<Person>`, поэтому на C# можно использовать модель `async/await` для извлечения внутреннего значения, когда операция завершится как продолжение.

На F# можно задействовать аналогичный подход, который поддерживает модель программирования на основе задач, хотя вследствие внутренней улучшенной поддержки асинхронного рабочего процесса я рекомендую использовать метод `ExecuteAsync`. В этом случае можно вызвать метод либо внутри асинхронного вычислительного выражения, либо с помощью функции `Async.StartWithContinuations`. С помощью этой функции обработчик продолжения может продолжить работу, когда `AgentSql` возвратит результат (см. главу 9). В листинге 11.4 показан пример использования обоих подходов F# (см. код, выделенный жирным шрифтом).

Листинг 11.4. Асинхронное взаимодействие с `AgentSql`

```

let token = CancellationToken()                                ← Остановка MailboxProcessor
let agentSql = AgentSql("< Connection String Here >")   ← маркером отмены
let printPersonName id = async {
    let! (Some person) = agentSql.ExecuteAsync id           ← Отправка сообщения и асинхронное
    printfn "Fullname %s %s" person.firstName person.lastName
}
→ Async.Start(printPersonName 42, token)                      ← ожидание ответа от MailboxProcessor

Асинхронный запуск вычисления
[Async.StartWithContinuations(agentSql.ExecuteAsync 42,
    (fun (Some person) ->
        printfn "Fullname %s %s" person.firstName person.lastName),
    (fun exn -> printfn "Error: %s" exn.Message),
    (fun cnl -> printfn "Operation cancelled"), token)
]
Запуск вычисления
с асинхронным управлением
завершением операции
Запуск функций, соответствующих
успешному завершению операции,
завершению с ошибкой или отмене

```

Функция `Async.StartWithContinuations` содержит код, выполняемый при завершении задания, в качестве продолжения. `Async.StartWithContinuations` принимает три различные функции продолжения, которые запускаются в зависимости от результата операции:

- ❑ код, выполняемый после успешного завершения операции и получения результата;
- ❑ код, выполняемый в случае возникновении исключения;
- ❑ код, выполняемый при отмене операции. Маркер отмены передается в качестве необязательного аргумента при запуске задания.

Для получения дополнительной информации перечитайте главу 9 или обратитесь к документации MSDN, доступной онлайн (<http://mng.bz/teA8>). Функция `Async.StartWithContinuations` несложна, и она обеспечивает удобное управление выбором поведения в случае успешного завершения, ошибки или отмены опе-

рации. Эти передаваемые функции называются *функциями продолжения*. Функции продолжения могут быть определены как лямбда-выражения в аргументах `Async.StartWithContinuations`. Очень эффективно определить код, который будет выполняться как простое лямбда-выражение.

11.5.4. Распараллеливание рабочего процесса и координация групп агентов

Основная причина, по которой агент должен обрабатывать сообщения для доступа к базе данных, заключается в контроле пропускной способности и правильной оптимизации использования пула соединений. Как обеспечить столь точный контроль параллелизма? Как система может выполнять несколько запросов параллельно без снижения производительности? `MailboxProcessor` — это примитивный тип, позволяющий гибко создавать компоненты многократного применения путем инкапсуляции поведения и построения обобщенных или индивидуальных интерфейсов в соответствии с потребностями программы.

В листинге 11.5 показан компонент многократного применения `parallelWorker` (выделен жирным шрифтом), порождающий множество агентов, количество которых задано счетчиком (`workers`). Здесь все агенты реализуют одинаковое поведение и обрабатывают входящие запросы методом циклического перебора. *Циклический перебор* — это алгоритм, который в данном случае используется в почтовом ящике агента для обработки входящих сообщений в порядке поступления, по кругу, обрабатывая все процессы без особого приоритета.

Главный агент (`agentCoordinator`) инициализирует набор субагентов, чтобы координировать работу и предоставлять доступ к дочерним агентам. Когда родительский агент получает сообщение, отправленное `MailboxProcessor parallelWorker`, он отправляет сообщение следующему доступному дочернему агенту (рис. 11.10).

Чтобы связать поведение с типом `MailboxProcessor`, функция `parallelWorker` существует свойство, называемое *расширением типа* (<http://mng.bz/Z5q9>). Расширение типа похоже на расширение метода. Благодаря расширению типа можно вызывать функцию `parallelWorker` посредством записи через точку; в результате можно использовать и вызывать функцию `parallelWorker` на любом другом языке программирования .NET, по-прежнему скрывая ее реализацию.

Эта функция имеет следующие аргументы:

- ❑ `workers` — количество инициализируемых параллельных агентов;
- ❑ `behavior` — функция для идентичной реализации внутренних агентов;
- ❑ `errorHandler` — функция, на которую подписываются все дочерние агенты для обработки возможных ошибок. Это необязательный аргумент, его можно пропустить. В таком случае передается функция игнорирования;
- ❑ `cts` — маркер отмены, используемый для остановки и удаления всех дочерних агентов. Если маркер отмены не передается в качестве аргумента, то инициализируется значение по умолчанию, которое передается в конструктор агента.

Листинг 11.5. Параллельные обработчики MailboxProcessor

```

type MailboxProcessor<'a> with
    static member public parallelWorker (workers:int) ←
        (behavior:MailboxProcessor<'a> -> Async<unit>)
        (?errorHandler:exn -> unit) (?cts:CancellationToken) =
            let cts = defaultArg cts (CancellationToken())
            let errorHandler = defaultArg errorHandler ignore
            let agent = new MailboxProcessor<'a>((fun inbox ->
                let agents = Array.init workers (fun _ ->
                    let child = MailboxProcessor.Start(behavior, cts)
                    child.Error.Subscribe(errorHandler)
                    child)
                cts.Register(fun () -> agents |> Array.iter(
                    fun a -> (a :> IDisposable).Dispose()))
                let rec loop i = async {
                    let! msg = inbox.Receive()
                    agents.[i].Post(msg)
                    return! loop((i+1) % workers)
                }
                loop 0), cts)
            agent.Start()
    
```

Поведение для построения внутренних дочерних агентов

Инициализация дочерних агентов

Значение workers определяет агенты, которые выполняются параллельно

Если нет маркера отмены или обработчик ошибок не найден, создается значение по умолчанию

Регистрация функции маркера отмены, которая останавливает и удаляет все агенты

Отправка сообщения агентам методом циклического перебора

Обработчик ошибок подписывается на каждого агента

Передача сообщений агентам-обработчикам методом циклического перебора

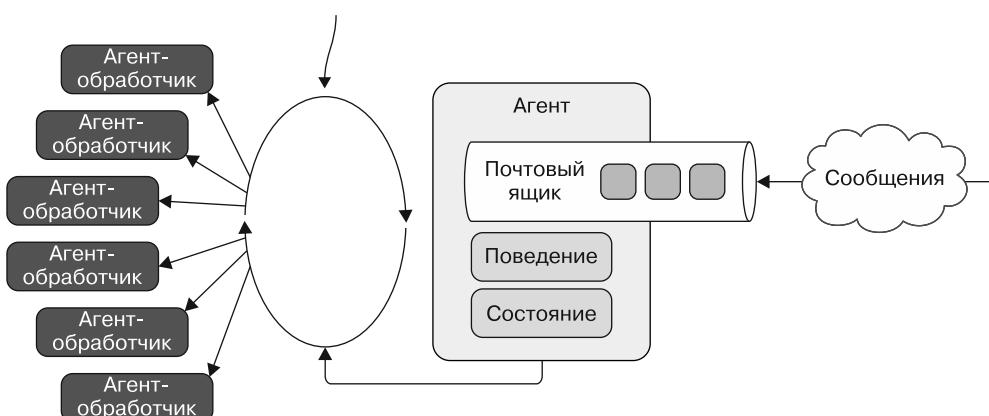


Рис. 11.10. Параллельный агент-обработчик получает сообщения, которые отправляются дочерним агентам в циклическом порядке для выполнения параллельных вычислений

11.5.5. Как обрабатывать ошибки на F# с помощью MailboxProcessor

Внутри функции `parallelWorker` создается экземпляр агента `MailboxProcessor`, который является родительским координатором массива агентов (дочерних), чье число равно значению аргумента `workers`:

```
let agents = Array.init workers (fun _ ->
    let child = MailboxProcessor.Start(behavior, cts)
    child.Error.Subscribe(errorHandler)
    child)
```

В процессе инициализации каждый дочерний агент подписывается на событие ошибки посредством функции `errorHandler`. В случае исключения, генерируемого в теле `MailboxProcessor`, инициируется событие ошибки и применяется подписанная функция.

Обнаружение ошибок и уведомление системы о них имеет решающее значение в агентном программировании, поскольку позволяет применять логику для соответствующего реагирования. В `MailboxProcessor` есть встроенная функциональность для обнаружения и пересылки ошибок.

Когда в агенте `MailboxProcessor` возникает неперехваченная ошибка, агент создает событие ошибки:

```
let child = MailboxProcessor.Start(behavior, cts)
child.Error.Subscribe(errorHandler)
```

Для управления ошибками можно зарегистрировать в обработчике событий функцию обратного вызова. Обычная практика — пересыпать ошибки управляющему агенту. Например, следующий простой агент-супервизор выводит полученную ошибку:

```
let supervisor = Agent<System.Exception>.Start(fun inbox ->
    async { while true do
        let! err = inbox.Receive()
        printfn "an error occurred in an agent: %A" err })
```

Можно определить функцию обработчика ошибок, которая передается в качестве аргумента при инициализации всех дочерних агентов:

```
let handler = fun error -> supervisor.Post error
```

```
let agents = Array.init workers (fun _ ->
    let child = MailboxProcessor.Start(behavior, cts)
    child.Error.Subscribe(errorHandler)
    child)
```

В критически важных компонентах приложения, таких как запросы сервера, представленные в виде агентов, следует планировать использование `MailboxProcessor` для тщательной обработки ошибок и надлежащего перезапуска приложения.

Для того чтобы упростить обработку ошибок путем уведомления главного агента, удобно определить вспомогательную функцию:

```
module Agent =
    let withSupervisor (supervisor: Agent<exn>) (agent: Agent<_>) =
        agent.Error.Subscribe(fun error -> supervisor.Post error); agent
```

Используя эту вспомогательную функцию, можно переписать предыдущую часть кода, в которой регистрируется обработка ошибок для `parallelWorker`, следующим образом:

```
let supervisor = Agent<System.Exception>.Start(fun inbox -> async {
    while true do
        let! error = inbox.Receive()
        errorHandler error })
let agent = new MailboxProcessor<'a>((fun inbox ->
let agents = Array.init workers (fun _ ->
    MailboxProcessor.Start(behavior)
|> withSupervisor supervisor)
```

Функция `parallelWorker` инкапсулирует главный агент, который применяет функцию `errorHandler` в качестве конструктора поведения для обработки сообщений об ошибках от дочерних агентов.

11.5.6. Остановка агентов MailboxProcessor — CancellationToken

Для того чтобы создать экземпляр дочернего агента, используется конструктор `MailboxProcessor`, который принимает в качестве первого аргумента функцию поведения агента, а в качестве второго — объект `CancellationToken`. Объект `CancellationToken` регистрирует функцию для удаления и остановки всех работающих агентов. Эта функция выполняется в случае отмены `CancellationToken`:

```
cts.Register(fun () ->
    agents |> Array.iter(fun a -> (a :> IDisposable).Dispose()))
```

Каждый выполняемый дочерний элемент в части `MailboxProcessor` агента `parallelWorker` представлен асинхронной операцией, связанной с заданным объектом `CancellationToken`. Маркеры отмены удобны в тех случаях, когда есть несколько агентов, зависящих друг от друга, и нужно отменить их все одновременно, как в нашем примере.

Дальнейшая реализация заключается в инкапсуляции агента `MailboxProcessor` в `IDisposable`:

```
type AgentDisposable<'T>(f:MailboxProcessor<'T> -> Async<unit>,
                           ?cancelToken:CancellationTokenSource) =
    let cancelToken = defaultArg cancelToken (new CancellationTokenSource())
```

```
let agent = MailboxProcessor.Start(f, cancellationToken.Token)

member x.Agent = agent
interface IDisposable with
    member x.Dispose() = (agent :> IDisposable).Dispose()
        cancellationToken.Cancel()
```

Таким образом, `AgentDisposable` упрощает отмену и освобождение памяти (`Dispose`) внутреннего `MailboxProcessor`, вызывая метод `Dispose` из интерфейса `IDisposable`.

Используя `AgentDisposable`, можно переписать предыдущую часть кода, которая регистрирует отмену дочернего агента для `parallelWorker`:

```
let agents = Array.init workers (fun _ ->
    new AgentDisposable<'a>(behavior, cancellationToken)
    |> withSupervisor supervisor)

thislet cancellationToken.Register(fun () ->
    agents |> Array.iter(fun agent -> agent.Dispose()))
```

При срабатывании маркера отмены `thislet cancellationToken` вызывается метод `Dispose` всех дочерних агентов, в результате чего они останавливаются. Полную реализацию переработанного `parallelWorker` вы найдете в исходном коде к этой книге.

11.5.7. Распределение работы с помощью `MailboxProcessor`

Остальная часть кода не требует пояснений. Когда сообщение передается `parallelWorker`, родительский агент перехватывает его и пересыпает первому агенту в очередь. Родительский агент применяет рекурсивный цикл для хранения состояния последнего использованного агента по индексу. На каждой итерации индекс увеличивается, чтобы доставить очередное доступное сообщение следующему агенту:

```
let rec loop i = async {
    let! msg = inbox.Receive()
    agents.[i].Post(msg)
    return! loop((i+1) % workers) }
```

Компонент `parallelWorker` можно использовать в самых разных случаях. В предыдущем примере кода с `AgentSql` мы применили расширение `parallelWorker`, чтобы достичь первоначальной цели — контроля (управления) над количеством параллельных запросов, которые могут получить доступ к серверу базы данных, чтобы оптимизировать применение пула соединений (листинг 11.6).

В приведенном примере максимальное количество открытых соединений задается произвольно, но в реальности это значение может варьироваться. В данном коде сначала создается `MailboxProcessor` `agentParallelRequests`, который

выполняется параллельно, с числом агентов, равным `maxOpenConnection`. Функция `fetchPeopleAsync` — последний кусок мозаики, соединяющий все части. Аргумент, передаваемый в эту функцию, представляет собой список идентификаторов людей, данные о которых необходимо извлечь из базы. Внутри функции к каждому идентификатору применяется агент `agentParallelRequests`, чтобы создать коллекцию асинхронных операций, которые будут выполняться параллельно с использованием функции `Async.Parallel`.

Листинг 11.6. Использование `parallelWorker` для распараллеливания операций чтения базы данных

```

let connectionString = ConfigurationManager.ConnectionStrings.[ "DbConnection" ].ConnectionString
    |—————| Извлечение строки
    | подключения из конфигурации

→ let maxOpenConnection = 10
    |—————| Произвольное значение максимального количества соединений
    | с базой данных, которые могут быть открыты одновременно

let agentParallelRequests =
    MailboxProcessor<SqlMessage>.parallelWorker(maxOpenConnection,
                                                ←
                                                |—————| Создание экземпляра parallelWorker
                                                | с агентом для подключения
                                                |—————| Использование массовой операции для получения
                                                | диапазона идентификаторов из базы данных
let fetchPeopleAsync (ids:int list) =
    let asyncOperation =
        ids
        |> Seq.map (fun id -> agentParallelRequests.PostAndAsyncReply(
                    fun ch -> Command(id, ch)))
        |> Async.Parallel
        Async.StartWithContinuations(asyncOperation,
            (fun people -> people |> Array.choose id
             |> Array.iter(fun person ->
                printfn "Fullname %s %s" person.firstName person.lastName),
            (fun exn -> printfn "Error: %s" exn.Message),
            (fun cnl -> printfn "Operation cancelled")))

```

ПРИМЕЧАНИЕ

Для асинхронного параллельного доступа к базе данных лучше всего контролировать и определять приоритеты операций чтения и записи. Базы данных работают наиболее эффективно, если операции записи выполняются последовательно. В предыдущем примере все операции являются операциями чтения, поэтому такой проблемы не существует. В главе 13 среди других практических рекомендаций приводится версия `MailboxProcessor parallelWorker`, ульт отдаётся приоритет режиму, при котором может выполняться одновременно только одна операция записи, но допускается несколько параллельных операций чтения.

В этом примере идентификаторы людей извлекаются параллельно; более эффективный способ — создать функцию `SqlCommand`, которая извлекала бы данные

из базы методом полного обхода. Но цель примера все равно достигается. Уровень параллелизма определяется количеством агентов. Это эффективная технология. В исходном коде к данной книге вы найдете полный, усовершенствованный и готовый к работе компонент `parallelWorker`, который сможете использовать в своей повседневной работе.

11.5.8. Операции кэширования в агентном программировании

В предыдущем разделе мы использовали F#-тип `MailboxProcessor` для реализации агента, обеспечивающего высокопроизводительный асинхронный доступ к базе данных и способного контролировать количество одновременных параллельных операций. Чтобы еще сократить время отклика на входящие запросы (скорость обработки запросов), можно уменьшить фактическое количество запросов к базе данных. Это возможно, если создать в программе кэш базы данных. Нет причин выполнять один и тот же запрос более одного раза за обращение, если его результат не изменяется. Применяя стратегии интеллектуального кэширования при доступе к базе данных, можно значительно увеличить производительность. Реализуем агентный компонент многоразовой кэш-памяти, который затем можно будет связать с агентом `agentParallelRequests`.

Задача агента кэширования — изолировать и сохранить состояние приложения, а также чтение и обновление этого состояния при обработке сообщений. В листинге 11.7 показана реализация такого агента — `MailboxProcessor CacheAgent`.

В этом примере первый тип, `CacheMessage`, определяет сообщение, которое `MailboxProcessor` отправляет в форме размеченного объединения. Последнее определяет валидные сообщения для отправки агенту кэша.

ПРИМЕЧАНИЕ

Размеченные объединения для вас уже не новость, но стоит напомнить, что в сочетании с `MailboxProcessor` они являются мощным инструментом, поскольку позволяют каждому определенному типу содержать свою сигнатуру. Таким образом, размеченные объединения предоставляют возможность определять связанные группы типов и контракты сообщений, которые используются для выбора различных реакций агента.

Ядром реализации `CacheAgent` являются инициализация и немедленный запуск `MailboxProcessor`, который постоянно следит за входящими сообщениями.

Листинг 11.7. Агент кэширования, использующий `MailboxProcessor`

```
type CacheMessage<'Key> =
| GetOrSet of 'Key * AsyncReplyChannel<obj>
| UpdateFactory of Func<'Key,obj>
| Clear
```

Использование размеченного объединения
для определения типа сообщения, которое
обрабатывает `MailboxProcessor`

```

type Cache<'Key when 'Key : comparison>
    (factory : Func<'Key, obj>, ?timeToLive : int) = 
        let timeToLive = defaultArg timeToLive 1000
        let expiry = TimeSpan.FromMilliseconds (float timeToLive)

        let cacheAgent = Agent.Start(fun inbox ->
            let cache = Dictionary<'Key, (obj * DateTime)>(
                HashIdentity.Structural)
                let rec loop (factory:Func<'Key, obj>) = async {
                    let! msg = inbox.TryReceive timeToLive
                    match msg with
                    | Some (GetOrSet (key, channel)) ->
                        match cache.TryGetValue(key) with
                        | true, (v,dt) when DateTime.Now - dt < expiry ->
                            channel.Reply v
                            return! loop factory
                        | _ ->
                            let value = factory.Invoke(key)
                            channel.Reply value
                            cache.Add(key, (value, DateTime.Now))
                            return! loop factory
                    | Some(UpdateFactory newFactory) ->
                        return! loop (newFactory)
                    | Some(Clear) ->
                        cache.Clear()
                        return! loop factory
                    | None ->
                        cache
                        |> Seq.filter(function KeyValue(k,(_, dt)) ->
                                DateTime.Now - dt > expiry)
                        |> Seq.iter(function KeyValue(k, _) ->
                                cache.Remove(k)|> ignore)
                        return! loop factory }
                loop factory )
            member this.TryGet<'a>(key : 'Key) = async {
                let! item = cacheAgent.PostAndAsyncReply(
                    fun channel -> GetOrSet(key, channel))
                match item with
                | :? 'a as v -> return Some v
                | _ -> return None }

            member this.GetOrSetTask (key : 'Key) =
                cacheAgent.PostAndAsyncReply(fun channel -> GetOrSet(key, channel))
                |> Async.StartAsTask

            member this.UpdateFactory(factory:Func<'Key, obj>) =
                cacheAgent.Post(UpdateFactory(factory))

```

Конструктор принимает функцию фабрики для изменения поведения агента во время выполнения программы

Использование состояния внутреннего поиска для кэширования

Попытка получить значение из кэша; если это невозможно, то создание нового значения с применением функции фабрики.

Затем значение отправляется вызывающему объекту

Установка времени жизни кэша, по истечении которого кэш аннулируется

Асинхронное ожидание сообщения, пока не истечет время ожидания. Если время ожидания истекло, то выполняется очистка кэша

Обновление функции фабрики

После того как получено значение от агента кэша, выполняется проверка типов. В случае успеха возвращается Some; иначе — None

Член класса для удобной совместимости с C#

Обновление функции фабрики

Конструкции F# облегчают применение лексической области видимости для изоляции асинхронных агентов. В следующем коде агента с помощью стандартной и изменяемой коллекции словарей .NET поддерживается состояние, полученное из различных сообщений, отправленных агенту:

```
let cache = Dictionary<'Key, (obj * DateTime)>()
```

Внутренний словарь лексически закрыт для асинхронного агента; любые операции чтения и записи в словарь возможны только через него. Изменяемое состояние в словаре является изолированным. Функция агента определяется как рекурсивный функциональный цикл, который принимает единственный параметр — фабрику, как показано ниже:

```
Agent.Start(fun inbox ->
    let rec loop (factory:'Key, obj) = async { ... }
```

Функция фабрики описывает политику инициализации для создания и добавления элемента в том случае, если этот элемент не удалось найти с помощью `cacheAgent` в локальном кэше состояний. Данная функция фабрики постоянно передается в рекурсивный функциональный цикл для управления состоянием, что позволяет менять процедуру инициализации во время выполнения программы. В случае кэширования запросов `AgentSql`, если прервется связь с базой данных или системой, стратегия ответа может измениться. Это легко достигается путем отправки сообщения агенту.

Агент получает сообщение с семантикой `MailboxProcessor`, в котором содержится задержка, определяющая время истечения. Такое особенно удобно для компонентов кэширования, когда нужно запустить сначала аннулирование, а затем обновление данных:

```
let! msg = inbox.TryReceive timeToLive
```

Функция входящих сообщений `TryReceive` возвращает тип параметра сообщения, который может быть либо `Some`, когда сообщение получено до истечения времени `timeToLive`, либо `None`, когда сообщение не получено в течение времени `timeToLive`:

```
| None ->
  cache
|> Seq.filter(function KeyValue(k,(_, dt)) -> DateTime.Now - dt > expiry)
|> Seq.iter(function KeyValue(k, _) -> cache.Remove(k) |> ignore)
```

В этом случае по истечении времени ожидания агент автоматически обновляет кэшированные данные, аннулируя (удаляя) все элементы кэша, время действия которых истекло. Но если сообщение получено, то агент использует сопоставление с образцом, чтобы определить тип сообщения и выполнить соответствующую обработку. У входящих сообщений есть следующие возможности.

- `GetOrSet` — в этом случае агент ищет в словаре кэша запись, содержащую указанный ключ. Если агент находит такой ключ и время аннулирования еще

не истекло, то агент возвращает соответствующее значение. В противном случае, если агент не находит ключ или время аннулирования ключа истекло, агент применяет функцию фабрики для генерирования нового значения, которое сохраняется в локальном кэше вместе с меткой времени его создания. Метка времени используется агентом для проверки времени истечения. В конце агент возвращает результат отправителю сообщения:

```
| Some (GetOrSet (key, channel)) ->
    match cache.TryGetValue(key) with
    | true, (v,dt) when DateTime.Now - dt < expiry ->
        channel.Reply v
        return! loop factory
    | _ ->
        let value = factory.Invoke(key)
        channel.Reply value
        cache.Add(key, (value, DateTime.Now))
        return! loop factory
```

- `UpdateFactory` — этот тип сообщения, как уже объяснялось, позволяет обработчику изменять политику инициализации элемента кэша во время выполнения:

```
| Some(UpdateFactory newFactory) ->
    return! loop (newFactory)
```

- `Clear` — этот тип сообщений очищает кэш для того, чтобы потом заново загрузить все элементы.

В итоге получается следующий код, который связывает предыдущие параллельные запросы `AgentSql agentParallelRequests` с `CacheAgent`:

```
let connectionString =
    ConfigurationManager.ConnectionStrings.[ "DbConnection" ].ConnectionString

let agentParallelRequests =
    MailboxProcessor<SqlMessage>.parallelWorker(8, agentSql
    ↪ connectionString)

let cacheAgentSql =
    let ttl = 60000
    CacheAgent<int>(fun id ->
        agentParallelRequests.PostAndAsyncReply(fun ch -> Command(id, ch)), ttl)

let person = cacheAgentSql.TryGet<Person> 42
```

Когда агент `cacheAgentSql` получает запрос, он проверяет, существует ли в кэше значение 42 и не истекло ли время его существования. В противном случае агент запрашивает внутреннюю функцию `parallelWorker`, которая возвращает ожидаемый элемент и сохраняет его в кэше для ускорения будущих запросов (рис. 11.11).

Каждый входящий запрос обрабатывается асинхронно в цикле CacheAgent. Если значение, связанное с запросом (ключ), существует во внутреннем кэше, то оно возвращается вызывающему объекту. Это гарантирует, что операция вычисления значения не повторяется. Если значение отсутствует в кэше, то операция вычисляет значение, добавляет его в кэш и затем возвращает значение вызывающему объекту

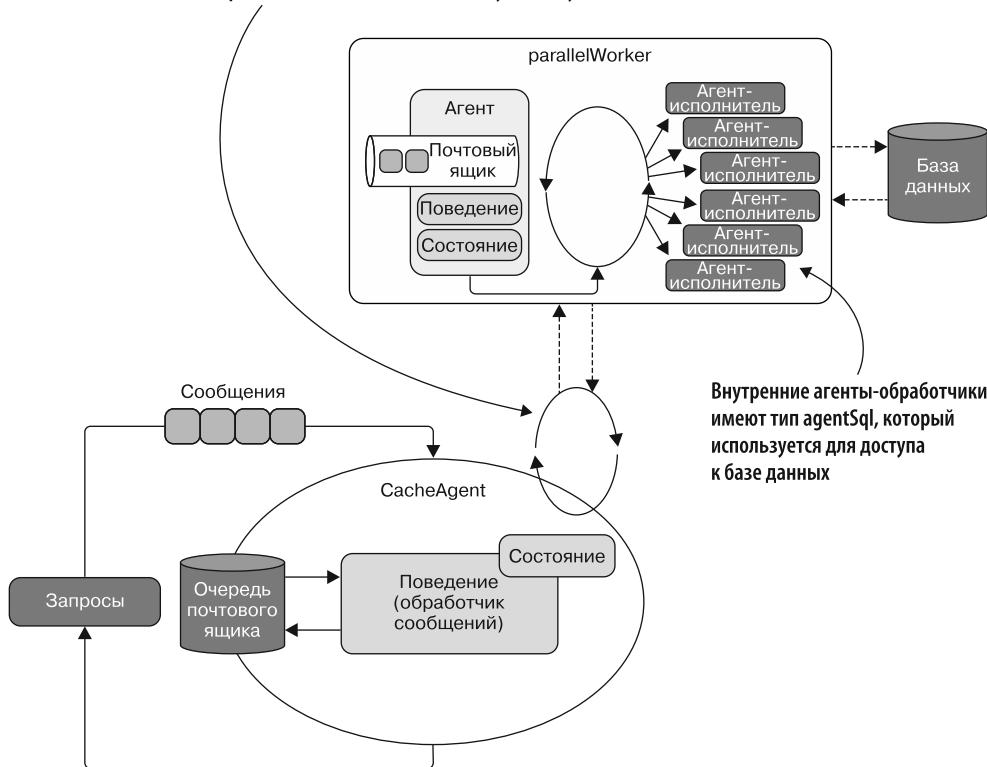


Рис. 11.11. CacheAgent поддерживает локальный кэш, состоящий из пар «ключ — значение», которые связывают входные данные (из запроса) со значением. Когда приходит запрос, CacheAgent проверяет существование входных данных (ключа) и затем либо возвращает значение (если ключ уже существует в локальном кэше) без выполнения каких-либо вычислений, либо вычисляет значение и отправляет его вызывающему объекту. В последнем случае значение также сохраняется в локальном кэше, чтобы избежать повторных вычислений для одних и тех же входных данных

11.5.9. Получение результатов от MailboxProcessor

Иногда MailboxProcessor должен отправить в систему отчет об изменении состояния, чтобы подписанный компонент мог обработать это изменение состояния. Например, для того, чтобы пример с CacheAgent был более полным, можно расширить его, добавив такие свойства, как уведомление об изменении данных или удалении кэша.

Но как `MailboxProcessor` отправляет уведомления во внешнюю систему? Это достигается с помощью событий (листинг 11.8). Мы уже видели, как `MailboxProcessor` сообщает о возникновении внутренней ошибки, рассыпая уведомления всем своим подписчикам. Тот же прием можно применить, чтобы сообщать о любых других событиях агента. Используя описанный ранее агент `CacheAgent`, реализуем отправку отчетов о событиях, которые можно задействовать для уведомления об аннулировании данных. Например, можно изменить агент так, чтобы при изменении данных выполнялось автоматическое обновление и рассыпались уведомления (соответствующий код выделен жирным шрифтом).

ПРИМЕЧАНИЕ

Такой шаблон уведомлений не рекомендуется использовать в ситуациях, когда `CacheAgent` обрабатывает много элементов, поскольку в зависимости от функции фабрики и обновляемых данных процесс автоматического обновления может занять значительное время.

Листинг 11.8. Кэш с уведомлением о событии для обновленных элементов

Использование события, чтобы сообщить об обновлении элемента кэша, что означает изменение состояния

```
type Cache<'Key when 'Key : comparison>
    (factory : Func<'Key, obj>, ?timeToLive : int,
     ?synchContext:SynchronizationContext) =
    let timeToLive = defaultArg timeToLive 1000
    let expiry = TimeSpan.FromMilliseconds (float timeToLive)

    → let cacheItemRefreshed = Event<('Key * 'obj)[]>()

    let reportBatch items = ←
        match synchContext with ←
        | None -> cacheItemRefreshed.Trigger(items) ←
        | Some ctx -> ←
            ctx.Post((fun _ -> cacheItemRefreshed.Trigger(items)),null) ←

    let cacheAgent = Agent.Start(fun inbox ->
        let cache = Dictionary<'Key, (obj * ←
        → DateTime)>(HashIdentity.Structural)
        let rec loop (factory:Func<'Key, obj>) = async {
            let! msg = inbox.TryReceive timeToLive
            match msg with
            | Some (GetOrSet (key, channel)) ->
                match cache.TryGetValue(key) with
                | true, (v,dt) when DateTime.Now - dt < expiry ->
                    channel.Reply v
                    return! loop factory
                | _ ->
```

Запуск события с применением указанного контекста синхронизации или прямую, если контекст синхронизации не указан

Контекст синхронизации не существует, поэтому он запускается как в первом случае

Применение метода `Post` для контекста, чтобы запустить событие

```

let value = factory.Invoke(key)
channel.Reply value
reportBatch [| (key, value) |])
cache.Add(key, (value, DateTime.Now))
return! loop factory
| Some(UpdateFactory newFactory) ->
    return! loop (newFactory)
| Some(Clear) ->
    cache.Clear()
    return! loop factory
| None ->
    cache
    |> Seq.choose(function KeyValue(k,(_, dt)) ->
        if DateTime.Now - dt > expiry then
            let value, dt = factory.Invoke(k), DateTime.Now
            cache.[k] <- (value,dt)
            Some (k, value)
        else None)
    |> Seq.toArray
    |> reportBatch
}
loop factory )
member this.TryGet<'a>(key : 'Key) = async {
    let! item = cacheAgent.PostAndAsyncReply(
        fun channel -> GetOrSet(key, channel))
    match item with
    | :? 'a as v -> return Some v
    | _ -> return None }
member this.DataRefreshed = cacheItemRefreshed.Publish
member this.Clear() = cacheAgent.Post(Clear)

```

В этом коде событие `cacheItemRefreshed` рассыпает сообщения об изменении состояния. По умолчанию обработчики событий F# выполняются в том же потоке, в котором были запущены события. В данном случае применяется текущий поток агента. Но в зависимости от того, из какого потока был создан `MailboxProcessor`, текущий поток может принадлежать `threadPool` либо это может быть поток из пользовательского интерфейса, в частности из `SynchronizationContext` — класса, принадлежащего `System.Threading`, который захватывает текущий контекст синхронизации. Последнее может быть полезно, если уведомление инициируется в ответ на событие, направленное на обновление пользовательского интерфейса. Именно поэтому конструктор агента в данном примере имеет новый параметр `synchContext` — этот тип параметра предоставляет удобный механизм для контроля того, где инициируется событие.

ПРИМЕЧАНИЕ

Дополнительные параметры в F# записываются с использованием вопросительного знака в виде префикса (?). `?synchContext` передает типы в качестве значений параметров.

Команда `Some ctx` означает, что `SynchronizationContext` не равен `null`; `ctx` — это произвольное имя, созданное для доступа к значению. Когда значение контекста синхронизации равно `Some ctx`, механизм создания отчетов использует метод `Post` для уведомления об изменениях состояния в потоке, выбранном контекстом синхронизации. Согласно сигнатуре метод синхронизации контекста `ctx.Post` принимает делегат и аргумент, используемый делегатом.

Второй аргумент необязателен, в качестве замены используется `null`. Функция `reportBatch` запускает событие `cacheItemRefreshed`:

```
this.DataRefreshed.Add(printAgent.Post)
```

В этом примере обработчик уведомлений об изменении состояния отправляет сообщение в `MailboxProcessor` для потокобезопасной печати отчета. Но вы можете использовать ту же идею для реализации более сложных сценариев — например, для автоматического обновления веб-страницы с отображением самых новых данных посредством `SignalR`.

11.5.10. Использование пула потоков для сообщения о событиях `MailboxProcessor`

В большинстве случаев во избежание ненужных издержек предпочтительно инициировать событие, используя текущий поток. Тем не менее бывают обстоятельства, когда лучше выбрать другую модель потоков: например, если инициирование события может вызвать временную блокировку или создать исключение, которое может прервать текущий процесс. Допустимый вариант — инициирование события, управляемого пулем потоков, для запуска уведомления в отдельном потоке. Функция `reportBatch` может быть модифицирована с применением асинхронного рабочего процесса F# и оператора `Async.Start` следующим образом:

```
let reportBatch batch =
    async { batchEvent.Trigger(batch) } |> Async.Start
```

Задействуя такую реализацию, помните, что код, выполняемый в пуле потоков, не может получить доступ к элементам пользовательского интерфейса.

11.6. F# `MailboxProcessor`: 10 000 агентов для `Game of Life`

По сравнению с потоками `MailboxProcessor` в сочетании с асинхронными рабочими процессами представляет собой простой вычислительный блок (примитив). Агенты могут появляться и уничтожаться с минимальными издержками. Можно распределить работу между несколькими объектами `MailboxProcessor` аналогично тому, как

можно использовать потоки, без дополнительных накладных расходов, связанных с созданием нового потока. Благодаря этому вполне возможно создавать приложения, состоящие из сотен тысяч агентов, работающих параллельно, с минимальной нагрузкой на ресурсы компьютера.

ПРИМЕЧАНИЕ

На компьютере с 32-разрядной операционной системой можно создать чуть более 1300 потоков, прежде чем возникнет исключение недостатка памяти. Это ограничение не распространяется на объекты MailboxProcessor, которые резервируются пулом потоков и не связываются с потоком напрямую.

В данном разделе мы воспользуемся несколькими экземплярами MailboxProcessor, чтобы реализовать игру Game of Life (игра «Жизнь») (https://en.wikipedia.org/wiki/Game_of_Life и https://ru.wikipedia.org/wiki/Игра_«Жизнь»). Согласно «Википедии», Game of Life, говоря простыми словами, является клеточным автоматом. Это игра без игроков — другими словами, когда игра начинается со случайной начальной конфигурации, она выполняется без каких-либо других входных данных. Игра состоит из набора клеток, образующих сетку; в каждой клетке выполняется несколько математических правил. Клетки могут жить, умирать и размножаться. Каждая клетка взаимодействует с восемью соседями (соседними клетками). Для перемещения клеток в соответствии с этими правилами необходимо постоянно вычислять новое состояние сетки.

Game of Life имеет следующие правила:

- если у клетки только один сосед или нет соседей, то она умирает «от одиночества»;
- если четверо или более соседей клетки умерло, то она умирает «из-за перенаселения»;
- если у клетки два или три соседа, то она остается жить;
- если у клетки три соседа, то она размножается.

В зависимости от начальных условий клетки образуют характерные структуры на протяжении всей игры. Посредством многократного применения правил создаются следующие поколения клеток, пока клетки не достигнут стабильного состояния (рис. 11.12).

В листинге 11.9 представлена реализация клетки Game of Life AgentCell, основанная на основе F#-типов MailboxProcessor. Каждая клетка-агент взаимодействует с соседними клетками посредством асинхронной передачи сообщений, создавая, таким образом, полностью распараллеленную Game of Life. Для краткости я опустил некоторые части кода, поскольку они не имеют отношения к основной теме примера. Полную реализацию вы найдете в исходном коде к этой книге, выложенном на сайте издательства.

AgentCell подключается и обменивается данными с окружающими соседними клетками, чтобы проверить их состояние и определить, являются ли они мертвыми или живыми. Выживание **AgentCell** зависит от этих окружающих клеток

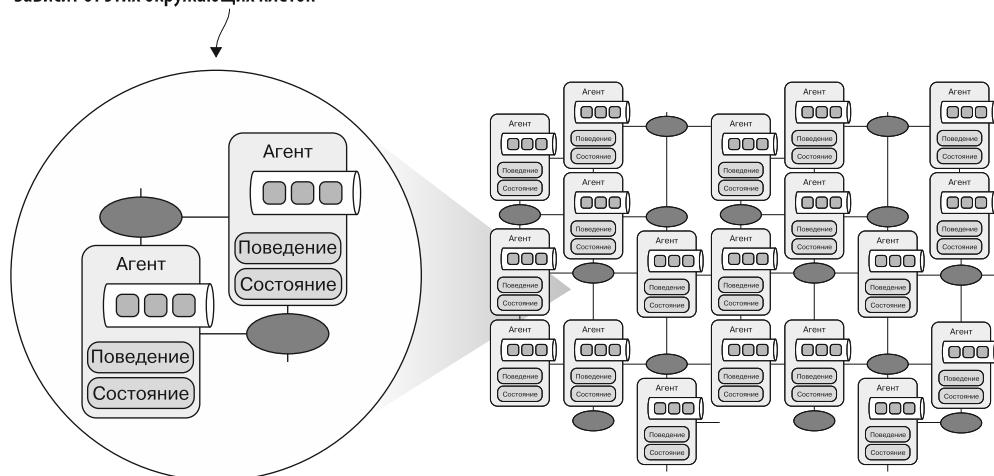


Рис. 11.12. После запуска Game of Life все клетки (в данном примере насчитывается 100 000 клеток) создаются с использованием MailboxProcessor AgentCell. Каждый агент может быть мертвым (отмечен как черный круг) или живым, в зависимости от состояния его соседей

Листинг 11.9. Game of Life с клетками на базе MailboxProcessor

```

type CellMessage =
| NeighborState of cell:AgentCell * isalive:bool
| State of cellstate:AgentCell
| Neighbors of cells:AgentCell list
| ResetCell

and State =
{   neighbors:AgentCell list
    wasAlive:bool
    isAlive:bool }

static member createDefault isAlive =
{ neighbors=[]; isAlive=isAlive; wasAlive=false; }

and AgentCell(location, alive, updateAgent:Agent<_>) as this =
let neighborStates = Dictionary<AgentCell, bool>()

let AgentCell =
    Agent<CellMessage>.Start(fun inbox -
        let rec loop state = async {
            let! msg = inbox.Receive()
            match msg with
            | ResetCell ->
                state.neighbors
                |> Seq.iter(fun cell -> cell.Send(State(this)))
                neighborStates.Clear()
            |> _ -> state
        }
        loop state
    )

```

← Размеченные объединения, определяющие сообщение для клетки-агента

← Тип записи, применяемый для отслеживания состояния каждой клетки-агента

← Внутреннее состояние каждого агента, позволяющее отслеживать состояние соседей каждой клетки-агента

← Уведомление всех соседей о ее текущем состоянии

```

    return! loop { state with wasAlive=state.isAlive } ←
  | Neighbors(neighbors) ->
    return! loop { state with neighbors=neighbors } ←
  | State(c) ->
    c.Send(NeighborState(this, state.wasAlive))
    return! loop state
  | NeighborState(cell, alive) ->
    neighborStates.[cell] <- alive
    if neighborStates.Count = 8 then
      let aliveState =
        let numberOfneighborAlive =
          neighborStates
            |> Seq.filter(fun (KeyValue(_,v)) -> v)
            |> Seq.length
        match numberOfneighborAlive with
        | a when a > 3 || a < 2 -> false
        | 3 -> true
        | _ -> state.isAlive
        updateAgent.Post(Update(aliveState, location))
        return! loop { state with isAlive = aliveState } ←
      else return! loop state
    loop (State.createDefault alive) ←
  member this.Send(msg) = AgentCell.Post msg

```

Использование алгоритма, который обновляет текущее состояние клетки в соответствии с состоянием соседей

Выполнение правил Game of Life

Рекурсивная поддержка локального состояния

Обновление агента, который обновляет пользовательский интерфейс

AgentCell описывает клетку в сетке Game of Life. Основная концепция заключается в том, что каждый агент обменивается информацией с соседними ячейками о своем текущем состоянии посредством асинхронной передачи сообщений. Этот шаблон создает цепочку взаимосвязанных параллельных коммуникаций, которая действует все клетки, отправляющие свое обновленное состояние агенту MailboxProcessor updateAgent. Получив эти данные, updateAgent обновляет графику в пользовательском интерфейсе (листинг 11.10).

Листинг 11.10. updateAgent обновляет пользовательский WPF-интерфейс в реальном времени

```

Конструктор updateAgent принимает записи, SynchronizationContext
использует их для обновления WPF-контроллера в соответствующем потоке
→ let updateAgent grid (ctx: SynchronizationContext) =
  let gridProduct = grid.Width * grid.Height
  let pixels = Array.zeroCreate<byte> (gridProduct) ←
    Массив пикселов, применяемый для визуализации состояния
    Game of Life. Каждый пиксел соответствует состоянию ячейки
Agent<UpdateView>.Start(fun inbox ->
  let gridState = Dictionary<Location, bool>(HashIdentity.Structural) ←
  let rec loop () = async {
    let! msg = inbox.Receive() ←
      Перечисление сообщений Update,
      каждое из которых обновляет
      состояние соответствующей клетки
      и обнуляет состояние клетки
    match msg with
    | Update(alive, location, agent) ->
      agentStates.[location] <- alive
      agent.Send(ResetCell)
    loop () ←
  }
)

```

Массив пикселов, применяемый для визуализации состояния Game of Life. Каждый пикセル соответствует состоянию ячейки

Общедоступное состояние сетки, соответствующее состоянию текущего поколения клеток

```

→ if agentStates.Count = gridProduct then
    agentStates.AsParallel().ForAll(fun s ->
        pixels.[s.Key.x+s.Key.y*grid.Width]
        <- if s.Value then 128uy else 0uy
    )
    do! Async.SwitchToContext ctx
    image.Source <- createImage pixels
    do! Async.SwitchToThreadPool()
    agentStates.Clear()
    return! Loop()
}
loop()

```

Пиксель, представляющий клетку, обновляется в соответствии с ее состоянием: живая (цветной) или мертвая (белый)

Обновление пользовательского интерфейса с применением соответствующего потока, переданного из конструктора

Когда все клетки сообщают о том, что они обновились, генерируется новое изображение, представляющее обновленную сетку, и обновляется приложение пользовательского WPF-интерфейса

`updateAgent`, как следует из названия, обновляет состояние каждого пикселя в соответствии со значением клетки, полученным в сообщении `Update`. Агент поддерживает состояние пикселов и задействует его для создания нового изображения, когда все клетки передадут свое новое состояние. Затем `updateAgent` обновляет графический пользовательский WPF-интерфейс, применяя это новое изображение, которое соответствует текущей сетке Game of Life:

```

do! Async.SwitchToContext ctx
image.Source <- createImage pixels
do! Async.SwitchToThreadPool()

```

Важно отметить, что агент `updateAgent` задействует текущий контекст синхронизации для корректного обновления WPF-контроллера. Текущий поток переключается на поток пользовательского интерфейса с помощью функции `Async.SwitchToContext` (описана в главе 9).

Последний фрагмент кода для выполнения Game of Life генерирует сетку, которая служит игровой площадкой для клеток, а затем таймер уведомляет клетки о необходимости выполнить обновление (листинг 11.11). В этом примере сетка представляет собой квадрат 100×100 клеток, всего 10 000 клеток (объектов `MailboxProcessor`), которые вычисляются параллельно по таймеру каждые 50 мс, как показано на рис. 11.13. Десять тысяч объектов `MailboxProcessor` взаимодействуют и обновляют пользовательский интерфейс 20 раз в секунду (код, на который следует обратить внимание, выделен жирным шрифтом).

Листинг 11.11. Создание сетки Game of Life и запуск таймера обновления

Задание размера стороны сетки

```

let run(ctx:SynchronizationContext) =
    let size = 100
    let grid = { Width= size; Height=size}
    let updateAgent = updateAgent grid ctx

```

Определение сетки с использованием типа записи с доступными свойствами `Width` и `Height`

```

let cells = seq { for x = 0 to grid.Width - 1 do
                  for y = 0 to grid.Height - 1 do
                      let agent = AgentCell({x=x;y=y}, ←
                        alive=getRandomBool(),
                        updateAgent=updateAgent)
                      yield (x,y), agent } |> dict
let neighbours (x', y') =
    seq {
        for x = x' - 1 to x' + 1 do
            for y = y' - 1 to y' + 1 do
                if x <> x' || y <> y' then
                    yield cells.[(x + grid.Width) % grid.Width,
                                (y + grid.Height) % grid.Height]
    } |> Seq.toList
}

cells.AsParallel().ForAll(fun pair ->
    let cell = pair.Value
    let neighbours = neighbours pair.Key
    cell.Send(Neighbors(neighbours)) ←
    cell.Send(ResetCell) ←
)

```

Создание сетки 100 × 100,
по одному MailboxProcessor
на клетку (всего 10 000 агентов)

Параллельное уведомление
всех клеток об их соседях
и сброс их состояния

Уведомления всем клеткам (агентам) рассылаются параллельно, с использованием PLINQ. Клетки представляют собой F#-последовательности, которые рассматриваются как .NET `IEnumerable`, что позволяет легко интегрировать LINQ/PLINQ.



Рис. 11.13. Game of Life. GUI является WPF-приложением

При выполнении кода программа генерирует 10 000 F#-объектов типа `MailboxProcessor` менее чем за 1 мс, при этом агенты занимают в памяти менее 25 Мбайт. Впечатляет!

Резюме

- ❑ Агентная модель программирования естественным образом обеспечивает неизменяемость и изоляцию при написании конкурентных систем, благодаря чему становится проще обсуждать даже сложные системы, поскольку агенты инкапсулированы внутри активных объектов.
- ❑ Реактивный манифест определяет свойства для реализации реактивной системы, которая является гибкой, слабосвязанной и масштабируемой.
- ❑ Естественная изоляция важна для написания конкурентного кода без блокировок. В многопоточной программе изоляция решает проблему разделяемых состояний, предоставляя каждому потоку скопированную часть данных для выполнения локальных вычислений. При использовании изоляции отсутствует состояние гонки.
- ❑ Будучи асинхронными, агенты являются простыми, поскольку не блокируют потоки, ожидая сообщений. В результате можно задействовать сотни тысяч агентов в одном приложении без особого влияния на объем памяти.
- ❑ F#-объект `MailboxProcessor` предусматривает двустороннюю коммуникацию: агент может использовать асинхронный канал, чтобы возвратить (ответить) вызывающему объекту результат вычислений.
- ❑ Модель агентного программирования в F# посредством `MailboxProcessor` является отличным инструментом для решения проблем узких мест в приложениях, таких как множественный конкурентный доступ к базе данных. Фактически с помощью агентов можно значительно ускорить работу приложений, сохраняя отзывчивость сервера.
- ❑ Другие языки программирования .NET позволяют использовать F#-тип `MailboxProcessor`, предоставляя методы с применением удобной TPL-модели программирования на основе задач.

12 Параллельный рабочий процесс и агентное программирование с помощью TPL Dataflow

В этой главе:

- использование блоков TPL Dataflow;
- построение высококонкурентного рабочего процесса;
- реализация сложной модели «поставщик — потребитель»;
- интеграция реактивных расширений с TPL Dataflow.

Сегодняшний глобальный рынок требует от предприятий и отраслей достаточной гибкости, чтобы реагировать на непрерывный поток меняющихся данных. Эти рабочие процессы часто бывают большими, а иногда даже бесконечными или неизвестного размера. Данные зачастую требуют сложной обработки, что приводит к требованиям высокой пропускной способности и потенциально огромным вычислительным нагрузкам. Определяющим условием, позволяющим справиться с этими требованиями, является использование параллелизма при эксплуатации системных ресурсов и нескольких ядер.

Но современные модели конкурентного программирования в .NET Framework не были рассчитаны на потоки данных. При разработке реактивного приложения крайне важно создавать и рассматривать компоненты системы как элементарные операции. Последние реагируют на сообщения, которые распространяются другими компонентами в цепочке обработки. Такие реактивные модели усиливают важность push-модели для работы приложений вместо pull-модели (см. главу 6).

Такая push-стратегия гарантирует, что отдельные компоненты будет легко тестировать и соединять, а главное, их будет легко понять.

Этот новый акцент на push-конструкциях меняет способ проектирования приложений. Единственная задача может быстро обрасти сложностями, и даже простые, на первый взгляд, требования могут привести к сложному коду.

Из данной главы вы узнаете, как библиотека .NET Task Parallel Library Dataflow (TPL Dataflow, или TDF) помогает справиться со сложностями разработки современных систем с помощью API, основанного на ТАР. TDF полностью поддерживает асинхронную обработку в сочетании с мощной семантикой компоновки и улучшенным, по сравнению с TPL, механизмом конфигурации. TDF упрощает конкурентную обработку, реализует специализированный асинхронный параллельный рабочий процесс и пакетную обработку. Кроме того, это упрощает реализацию сложных шаблонов, основанных на объединении нескольких компонентов, которые обмениваются данными между собой путем передачи сообщений.

12.1. Преимущества TPL Dataflow

Предположим, вы строите сложный шаблон «поставщик — потребитель», который должен параллельно поддерживать несколько поставщиков и/или несколько потребителей. Или, возможно, этот шаблон должен поддерживать рабочие процессы, различные этапы которых могут масштабироваться независимо друг от друга. Одним из решений задачи является использование Microsoft TPL Dataflow. После выпуска .NET 4.5 библиотека Microsoft TPL Dataflow стала частью инструментария для написания конкурентных приложений. В состав TDF входят высокогорловневые конструкции, необходимые для решения простых параллельных проблем и обеспечивающие при этом простой в использовании и мощный фреймворк для построения асинхронных конвейеров обработки данных. TDF не распространяется в составе .NET 4.5 Framework, поэтому для доступа к API и классам этой библиотеки необходимо импортировать официальный Microsoft NuGet Package (`install-Package Microsoft.Tpl.DataFlow`).

TDF предоставляет широкий набор компонентов (также называемых *блоками*) для компоновки инфраструктуры потоков данных и конвейеров на основе внутренней семантики передачи сообщений (рис. 12.1). Такая модель потока данных способствует программированию на основе акторов, обеспечивая внутреннюю передачу сообщений для крупномодульных потоков данных и задач конвейерной обработки.

В TDF используется планировщик задач (`TaskScheduler`, <http://mng.bz/4N8F>) из TPL для эффективного управления внутренними потоками и поддержки модели ТАР (`async/await`), что позволяет оптимизировать использование ресурсов. TDF повышает надежность приложений с высокой степенью конкурентности и обеспечивает более высокую производительность при распараллеливании вычислений между несколькими процессорами и интенсивных операциях ввода-вывода, требующих высокой пропускной способности и малого времени ожидания.

ПРИМЕЧАНИЕ

TPL Dataflow обеспечивает эффективные технологии для выполнения естественно параллельных задач, описанных в главе 3, то есть таких задач, где существует много независимых вычислений, для которых очевидно, что они могут выполняться параллельно.

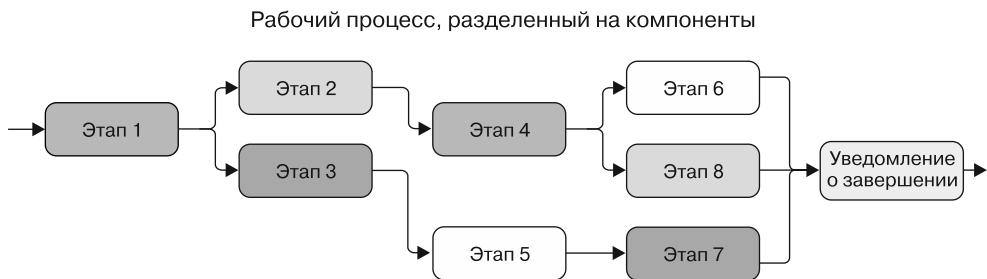


Рис. 12.1. Рабочий процесс, состоящий из нескольких этапов. Каждая операция может рассматриваться как независимое вычисление

Концепция библиотеки TPL Dataflow состоит в том, чтобы упростить создание нескольких шаблонов, например, с конвейерами пакетной обработки, параллельной обработкой потоков, буферизацией данных или объединением и обработкой пакетных данных, поступающих из одного или нескольких источников. Каждый из этих шаблонов можно использовать отдельно или в сочетании с другими шаблонами, что позволяет разработчикам легко описывать сложные потоки данных.

12.2. Блоки TPL Dataflow: созданы для компоновки

Предположим, что нам надо реализовать сложный рабочий процесс, состоящий из множества различных этапов, такой как конвейер анализа ценных бумаг. Было бы идеально разбить вычисления на блоки, разработать каждый блок независимо от других, а затем склеить все блоки вместе. Будет еще удобнее, если мы сделаем эти блоки многоразовыми и взаимозаменяемыми. Такая компонуемая структура упростила бы реализацию сложных и запутанных систем.

Возможность компоновки является главным преимуществом TPL Dataflow, поскольку набор независимых контейнеров этой библиотеки, называемых блоками, рассчитан на компоновку. Данные блоки могут представлять собой цепочку различных задач, образующих параллельный рабочий процесс, и могут быть легко заменены, переупорядочены, использованы повторно или даже удалены. В TDF уделяется особое внимание архитектурному подходу компонентов для упрощения реструктуризации систем. Эти компоненты потока данных полезны в тех случаях, когда имеется несколько операций, которые должны асинхронно взаимодействовать

друг с другом, или когда нужно обрабатывать данные по мере поступления, как показано на рис. 12.2.

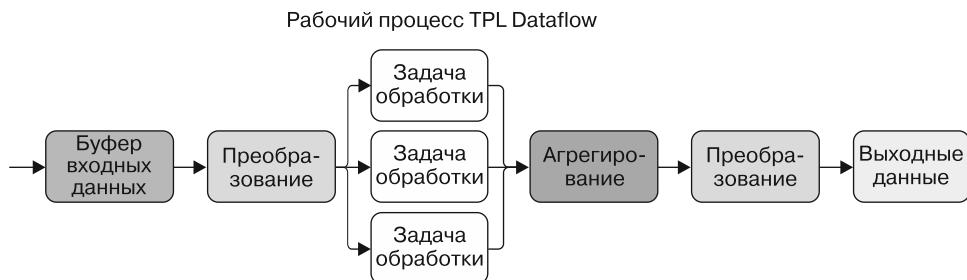


Рис. 12.2. В TDF уделяется особое внимание концепции многократного использования компонентов. На этом рисунке каждый шаг рабочего процесса представляет собой многократно используемый компонент. В состав TDF входит несколько основных примитивов, позволяющих выражать вычисления на основе графов Dataflow

В целом блоки TDF работают следующим образом.

- Каждый блок получает и буферизует данные из одного или нескольких источников, включая другие блоки, в виде сообщений. Получив сообщение, блок реагирует на него, применяя свое поведение к входным данным, которые затем могут быть преобразованы и/или использованы для выполнения побочных эффектов.
- Затем выходные данные компонента (*блока*) передаются следующему, связанному с ним, блоку, а потом следующему, если таковой имеется, и так далее, образуя структуру конвейера.

ПРИМЕЧАНИЕ

Термин «реактивное программирование» долгое время использовался для описания потока данных, поскольку реакция генерируется в ответ на получение фрагмента данных.

В TDF входит отличный набор настраиваемых свойств, с помощью которых с минимальными изменениями можно контролировать уровень параллелизма, управлять размером буфера почтового ящика, обрабатывать данные и передавать выходные данные.

Существует три основных типа блоков потока данных.

- *Источник (source)* — действует как поставщик данных. Из источника также можно прочитать данные.
- *Приемник (target)* — действует как потребитель, который получает данные и в который они могут быть записаны.
- *Распространитель (propagator)* — может действовать и как блок Source, и как блок Target.

Для каждого из этих блоков потока данных TDF предоставляет набор субблоков, каждый из которых имеет свое назначение. Описать все блоки в одной главе невозможно. В следующих разделах мы сосредоточимся на наиболее распространенных и универсальных блоках, применяемых в типичных приложениях с конвейерной компоновкой.

СОВЕТ

Наиболее часто используемые блоки TPL Dataflow — это стандартные BufferBlock, ActionBlock и TransformBlock. Каждый из них основан на делегате, который может быть реализован в форме анонимной функции, описывающей вычисления. Я рекомендую делать эти анонимные методы короткими, простыми для понимания и удобными в обслуживании.

Для получения дополнительной информации о библиотеке Dataflow читайте онлайн-документацию MSDN (<http://mng.bz/GDbF>).

12.2.1. Использование BufferBlock<TInput> в качестве буфера FIFO

TDF-блок `BufferBlock<T>` действует как неограниченный буфер для данных, которые хранятся в порядке FIFO — «первым пришел — первым вышел» (рис. 12.3). В общем случае `BufferBlock` — отличный инструмент для реализации и использования асинхронных шаблонов «поставщик — потребитель», в которых запись во внутреннюю очередь сообщений может производиться из нескольких источников, а чтение — несколькими потребителями.

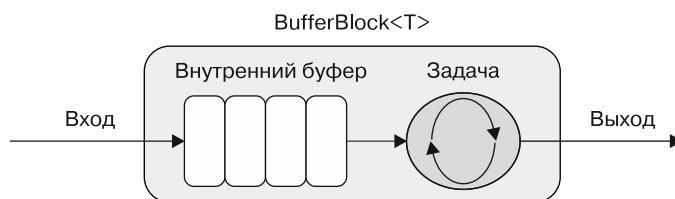


Рис. 12.3. TDF-блок `BufferBlock` имеет внутренний буфер, в котором сообщения помещаются в очередь, ожидая обработки задачей. Входные и выходные данные имеют одинаковый тип, этот блок не выполняет преобразование данных

В листинге 12.1 показана простая реализация шаблона «поставщик — потребитель» с использованием TDF-блока `BufferBlock`.

Элементы со значениями `IEnumerable` отправляются посредством метода `buffer.Post` в буфер `BufferBlock`, который асинхронно извлекает их, используя метод `buffer.ReceiveAsync`. Метод `OutputAvailableAsync` знает, когда следующий элемент готов и его можно получить, и отправляет уведомление. Это важно для защиты кода от исключений; если буфер попытается вызвать метод `Receive` после

того, как блок завершил обработку, выдается ошибка. Главное назначение блока `BufferBlock`, по существу, — принимать и хранить данные, чтобы их можно было передать одному или нескольким другим блокам-приемникам для обработки.

Листинг 12.1. Шаблон «поставщик — потребитель» на основе TDF BufferBlock

```
BufferBlock<int> buffer = new BufferBlock<int>(); ← Доступ возможен только
async Task Producer(IEnumerable<int> values) ← через BufferBlock<T>
{
    foreach (var value in values)
        buffer.Post(value); ← Отправка сообщения в BufferBlock
    buffer.Complete(); ← Уведомление BufferBlock о том, что элементов
} ← для обработки больше нет, и завершение работы
async Task Consumer(Action<int> process)
{
    while (await buffer.OutputAvailableAsync()) ← Сигналы о том, что доступен
        process(await buffer.ReceiveAsync()); ← новый элемент
    } ← и его можно получить
    ← Асинхронное
async Task Run() ← получение сообщения
{
    IEnumerable<int> range = Enumerable.Range(0,100);
    await Task.WhenAll(Producer(range), Consumer(n =>
        Console.WriteLine($"value {n}")));
}
}
```

12.2.2. Преобразование данных с помощью `TransformBlock<TInput, TOutput>`

TDF-блок `TransformBlock<TInput, TOutput>` работает как функция отображения, которая применяет функцию проекции к входному значению и формирует на выходе соответствующие данные (рис. 12.4). Функция преобразования передается в качестве аргумента в виде делегата `Func<TInput, TOutput>`, который обычно представлен как лямбда-выражение. Назначение этого блока по умолчанию — обрабатывать сообщения по одному, строго соблюдая порядок FIFO.

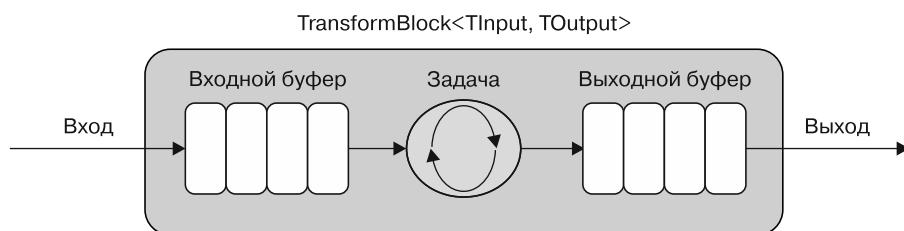


Рис. 12.4. TDF `TransformBlock` имеет внутренний буфер для входных и выходных значений; у блоков этого типа такие же возможности буфера, что и у `BufferBlock`. Назначение такого блока — применить к данным функцию преобразования; `Input` и `Output`, скорее всего, имеют разные типы

Обратите внимание на то, что `TransformBlock<TInput, TOutput>` работает как `BufferBlock<TOutput>`, который буферизует как входные, так и выходные значения. Внутренний делегат может работать синхронно или асинхронно. Асинхронная версия имеет сигнатуру типа `Func<TInput, Task<TOutput>>`, назначение которой — асинхронное выполнение внутренней функции. Блок считает обработку элемента завершенной, когда прекратилось выполнение возвращаемой задачи. Использование типа `TransformBlock` показано в листинге 12.2 (код, на который следует обратить внимание, выделен жирным шрифтом).

Листинг 12.2. Загрузка изображений с использованием TDF `TransformBlock`

**Использование лямбда-выражения
для асинхронной обработки `urlImage`**

```
var fetchImageFlag = new TransformBlock<string, (string, byte[])>(
    async urlImage => {
        using (var webClient = new WebClient()) {
            byte[] data = await webClient.DownloadDataTaskAsync(urlImage); ←
            return (urlImage, data); ←
        }
    });
    ↑ Выходные данные состоят из кортежа с URL-адресом
    ↑ изображения и соответствующим байтовым массивом
List<string> urlFlags = new List<string>{
    "Italy#/media/File:Flag_of_Italy.svg",
    "Spain#/media/File:Flag_of_Spain.svg",
    "United_States#/media/File:Flag_of_the_United_States.svg"
};

foreach (var urlFlag in urlFlags)
    fetchImageFlag.Post($"https://en.wikipedia.org/wiki/{urlFlag}");
```

**Загрузка изображения
флага и возвращение
соответствующего
байтового массива**

В этом примере блок `TransformBlock<string, (string, byte [])>` `fetchImageFlag` извлекает изображение флага в виде кортежа, состоящего из строки и байтового массива. В таком случае выходные данные нигде не используются, поэтому код не особенно полезен. Для осмысленной обработки результата нам нужен еще один блок.

12.2.3. Законченный пример с `ActionBlock<TInput>`

TDF-блок `ActionBlock` выполняет заданный обратный вызов для любого переданного ему элемента. Логически этот блок можно рассматривать как буфер для данных в сочетании с задачей по их обработке. `ActionBlock<TInput>` — блок-приемник, который вызывает делегат при получении данных аналогично циклу `for-each` (рис. 12.5).

`ActionBlock<TInput>` обычно является последним шагом в TDF-конвейере, так как не производит никаких выходных данных. Подобная конструкция не позволяет добавлять после `ActionBlock` другие блоки, если только он не публикует или не отправляет данные в другой блок, что делает его идеальным кандидатом для завершения

рабочего процесса. По этой причине `ActionBlock` обычно вызывает побочные эффекты в качестве последнего этапа для завершения конвейерной обработки.

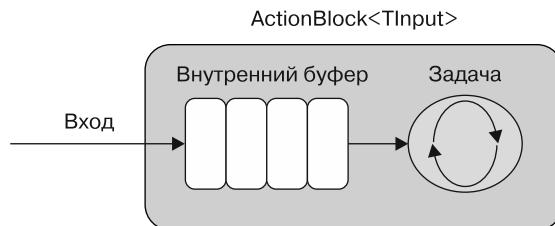


Рис. 12.5. TDF-блок `ActionBlock` имеет внутренний буфер для входящих сообщений, которые помещаются в очередь, если задача занята обработкой другого сообщения. Блоки этого типа имеют такие же возможности буфера, что и `BufferBlock`. Назначение блока — применить действие, которое завершает рабочий процесс без вывода данных, что, вероятно, вызывает побочные эффекты. В целом, поскольку у `ActionBlock` нет выходных данных, его нельзя скомпоновать со следующим блоком, поэтому указанный блок используется для завершения рабочего процесса

В следующем коде (листинг 12.3) показано, как блок `TransformBlock` из предыдущего листинга передает свои выходные данные в `ActionBlock`, чтобы сохранить изображения флагов в локальной файловой системе (выделены жирным шрифтом).

Листинг 12.3. Сохранение данных с использованием TDF-блока `ActionBlock`

Использование лямбда-выражения для асинхронной обработки данных	Деконструирование кортежа для доступа к внутренним элементам
<pre> → var saveData = new ActionBlock<(string, byte[])>(async data => { (string urlImage, byte[] image) = data; string filePath = urlImage.Substring(urlImage.IndexOf("File:") + 5); await File.WriteAllBytesAsync(filePath, image); }); → fetchImageFlag.LinkTo(saveData); </pre>	
Асинхронная запись данных в локальную файловую систему	
Соединение выхода <code>TransformBlock</code> <code>fetchImageFlag</code> с <code>ActionBlock</code> <code>saveData</code>	

Аргумент, передаваемый в конструктор при создании экземпляра блока `ActionBlock`, может быть либо делегатом `Action<TInput>`, либо `Func<TInput, Task>`. Последняя функция асинхронно выполняет внутреннее действие (поведение) для каждого входящего (полученного) сообщения. Обратите внимание, что у `ActionBlock` есть внутренний буфер для обработки входных данных, который работает точно так же, как буфер `BufferBlock`.

Важно помнить, что блок `saveData` `ActionBlock` связан с предшествующим ему блоком `TransformBlock` `fetchImageFlag` посредством метода расширения `LinkTo`. Таким образом, выходные данные, генерируемые `TransformBlock`, передаются в `ActionBlock` сразу же, как только становятся доступными.

12.2.4. Связывание блоков в потоке данных

TDF-блоки можно соединить с помощью метода расширения `LinkTo`. Связывание блоков потока данных является мощной технологией для автоматической передачи результата каждого вычисления между связанными блоками методом передачи сообщений. Ключевым компонентом для построения сложных конвейеров в декларативном стиле является использование взаимосвязанных блоков. Если посмотреть на сигнатуру метода расширения `LinkTo` с концептуальной точки зрения, то увидим, что она выглядит как функциональная компоновка:

`LinkTo: (a -> b) -> (b -> c)`

12.3. Реализация сложных шаблонов «поставщик — потребитель» с помощью TDF

Модель программирования TDF можно рассматривать как сложный шаблон «поставщик — потребитель», поскольку блоки поддерживают конвейерную модель программирования, при которой поставщики отправляют сообщения отделенным от них потребителям. Эти сообщения передаются асинхронно, обеспечивая максимальную пропускную способность. Преимуществом такой структуры является отсутствие блокировок со стороны поставщиков, поскольку TDF-блоки (очередь) играют роль буфера, сводя к нулю время ожидания. Синхронизация доступа между поставщиками и потребителями может показаться абстрактной проблемой, но это обычная задача при конкурентном программировании. Она даже встречается в виде шаблона проектирования для синхронизации двух компонентов.

12.3.1. Реализация шаблона «несколько поставщиков — один потребитель» с помощью TDF

Шаблон «поставщик — потребитель» является одним из наиболее широко распространенных шаблонов параллельного программирования. Разработчики используют его, чтобы изолировать выполняемые действия от обработки результатов этих действий. В типичном шаблоне «поставщик — потребитель» одновременно выполняются как минимум два отдельных потока: один создает данные для обработки и помещает их в очередь, а второй проверяет наличие новой порции входных данных и обрабатывает их. Очередь, в которой содержатся задачи, доступна обоим этим потокам, что требует предосторожностей для обеспечения безопасного доступа к задачам. Библиотека TDF является отличным инструментом для реализации указанного шаблона, поскольку имеет встроенную поддержку конкурентного доступа для нескольких объектов чтения и нескольких объектов записи и поощряет

конвейерный шаблон программирования, в котором поставщики отправляют сообщения отделенным от них потребителям (рис. 12.6).

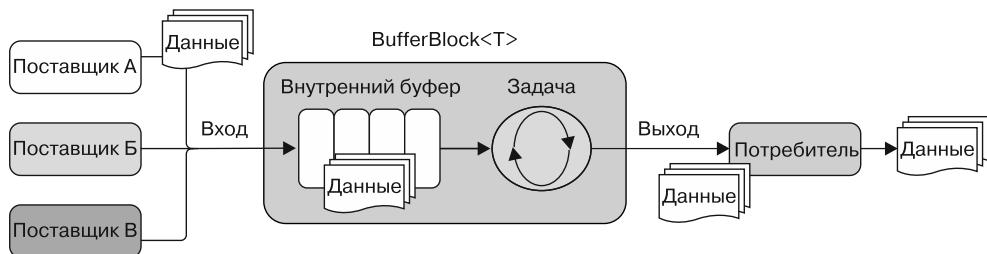


Рис. 12.6. Шаблон со многими поставщиками и одним потребителем, построенный с использованием TDF-блока BufferBlock, который позволяет управлять данными и ограничивать нагрузку, созданную несколькими производителями

В случае шаблона с несколькими поставщиками и одним потребителем важно ввести ограничение на количество генерируемых и потребляемых элементов. Цель такого ограничения — сбалансировать работу поставщиков, пока потребитель не в состоянии справиться с нагрузкой. Эта технология называется *ограничением*. Ограничение защищает программу от исчерпания памяти, если поставщики будут работать быстрее, чем потребители. К счастью, в TDF есть встроенная поддержка ограничения, нужно лишь указать максимальный размер в свойстве `BoundedCapacity`, которое является частью `DataFlowBlockOptions`. В листинге 12.4 это свойство гарантирует, что длина очереди `BufferBlock` никогда не превысит десяти элементов. Также, помимо принудительного ограничения размера буфера, важно использовать функцию `SendAsync`, которая ожидает, пока в буфере не освободится место для размещения нового элемента, не создавая блокировок.

Листинг 12.4. Реализация асинхронного шаблона «поставщик — потребитель» с использованием TDF

Асинхронная отправка сообщения в блок буфера. Метод `SendAsync` помогает ограничивать количество отправляемых сообщений

```

BufferBlock<int> buffer = new BufferBlock<int>(
    new DataFlowBlockOptions { BoundedCapacity = 10 });

async Task Produce(IEnumerable<int> values)
{
    foreach (var value in values)
        await buffer.SendAsync(value);
}

async Task MultipleProducers(params IEnumerable<int>[] producers)
{
    await Task.WhenAll(
        from values in producers select Produce(values).ToArray());
}
  
```

Значение `BoundedCapacity` позволяет управлять и ограничивать нагрузку, созданную несколькими поставщиками

Параллельная работа нескольких поставщиков; ожидание завершения всех поставщиков, после чего блок буфера получает уведомление о завершении

```

        .ContinueWith(_ => buffer.Complete()); ← Когдa все поставщики завершат
    }                                                 работу, блок буфера получает
                                                    уведомление о завершении

    async Task Consumer(Action<int> process)
    {
        while (await buffer.OutputAvailableAsync()) ← Защитa блока буфера от получения
            process(await buffer.ReceiveAsync());     сообщений, пока в очереди
        }                                              есть какие-либо элементы

    async Task Run()
    {
        IEnumerable<int> range = Enumerable.Range(0, 100);

        await Task.WhenAll(MultipleProducers(range, range, range),
                           Consumer(n => Console.WriteLine($"value {n} – ThreadId
                           {Thread.CurrentThread.ManagedThreadId}")));
    }
}

```

По умолчанию у всех TDF-блоков значение `DataFlowBlockOptions.Unbounded` равно `-1`. Это означает, что очередь бесконечна (не ограничена) и может содержать любое количество сообщений. Но такое значение можно изменить и присвоить ему определенное значение, ограничивающее количество сообщений, которые могут находиться в очереди блока. Когда очередь достигнет максимальной длины, любые новые входящие сообщения будут отложены для последующей обработки, что заставит поставщика приостановить работу и ожидать. Очевидно, замедление работы (или ожидание) поставщика не является проблемой, поскольку сообщения отправляются асинхронно.

12.3.2. Шаблон «один поставщик — несколько потребителей»

TDF-блок `BufferBlock` изначально поддерживает шаблон «один поставщик — несколько потребителей». Это удобно, если поставщик работает быстрее, чем несколько потребителей, например, когда потребители выполняют интенсивные операции.

К счастью, такой шаблон выполняется на многоядерном компьютере, поэтому может использовать несколько ядер для выполнения нескольких блоков обработки (потребителей), каждый из которых может конкурентно обрабатывать данные от поставщиков.

Удастся ли обеспечить поведение с несколькими потребителями — зависит от конфигурации. Для этого значение свойства `MaxDegreeOfParallelism` должно быть равно количеству потребителей, которые нужно запустить параллельно. Для того чтобы листинг 12.4 обеспечивал максимальную степень параллелизма в зависимости от количества доступных логических процессоров, его нужно изменить следующим образом:

```
BufferBlock<int> buffer = new BufferBlock<int>(new DataFlowBlockOptions {
    BoundedCapacity = 10,
    MaxDegreeOfParallelism = Environment.ProcessorCount });

```

ПРИМЕЧАНИЕ

Логическими ядрами называют количество физических ядер, умноженное на количество потоков, способных выполняться на каждом ядре. Так, для восьмиядерного процессора с двумя потоками на каждое ядро количество логических процессоров равно 16.

По умолчанию TDF-блок настроен так, чтобы обрабатывать только одно сообщение в единицу времени, в то время как остальные входящие сообщения буферизуются до завершения обработки предыдущего сообщения. Блоки не зависят друг от друга, поэтому в то время, как один блок обрабатывает один элемент, другой блок может обрабатывать другой элемент. Но при создании блока данное поведение можно изменить, назначив свойству `MaxDegreeOfParallelism` в `DataFlowBlockOptions` значение, большее 1. Можно использовать TDF-блоки для ускорения вычислений, изменяя количество сообщений, которые могут обрабатываться параллельно. Остальное сделают внутренние элементы класса, включая порядок обработки последовательности данных.

12.4. Использование агентной модели в C# с помощью TPL Dataflow

По умолчанию TDF-блоки работают без сохранения состояния, что идеально подходит для большинства сценариев. Но в приложениях встречаются ситуации, когда важно поддерживать состояние: например, глобальный счетчик, централизованный кэш в памяти или контекст общедоступной базы данных для транзакционных операций.

В таких ситуациях существует высокая вероятность того, что разделяемое состояние является изменяемым вследствие постоянного отслеживания определенных значений. Проблема всегда заключалась в сложности обработки асинхронных вычислений в сочетании с изменяемым состоянием. Как уже отмечалось ранее, изменяемость разделяемого состояния в многопоточной среде становится опасной и приводит нас в ловушку конкурентных проблем (<http://curtclifton.net/papers/MoseleyMarks06a.pdf>). К счастью, в библиотеке TDF состояние инкапсулируется внутри блоков, а единственными зависимостями являются каналы между блоками. По замыслу это позволяет безопасно изолировать изменяемость.

Как показано в главе 11, F#-объект `MailboxProcessor` позволяет решить эти проблемы, поскольку соответствует философии агентной модели, которая может поддерживать внутреннее состояние, обеспечивая защиту доступа и безопасную конкурентность (в каждый момент времени только один поток может получить доступ к агенту). Наконец, F#-объект `MailboxProcessor` имеет набор API для кода C#, позволяющий без особых усилий организовать доступ потребителей. Кроме того, можно достичь той же производительности, используя TDF для реализации объекта-агента на C#, и тогда этот объект-агент сможет выступать в качестве F# `MailboxProcessor`.

С сохранением или без сохранения состояния?

Сохранение состояния означает, что программа отслеживает состояние взаимодействия. Обычно подобное достигается путем записи значений в поле хранения, предназначенное для этой цели.

Отсутствие сохранения состояния означает отсутствие записи о предыдущих взаимодействиях, так что каждый запрос взаимодействия должен обрабатываться исключительно на основе новой информации, которая поступает вместе с ним.

Реализация `StatefulDataFlowAgent` опирается на экземпляр `actionBlock` для получения, буферизации и обработки входящих сообщений с неограниченным лимитом (рис. 12.7). Обратите внимание: максимальная степень параллелизма равна значению по умолчанию — 1, как и должно быть в соответствии с однопоточным характером агентной модели. Состояние агента инициализируется в конструкторе и поддерживается с помощью полиморфного изменяемого значения `TState`, которое переопределяется при обработке каждого сообщения. (Напомню: агентная модель подразумевает возможность доступа только одного потока в единицу времени, гарантируя, таким образом, последовательную обработку сообщений и отсутствие каких-либо конкурентных проблем.) Рекомендуется использовать неизменяемое состояние, независимо от степени безопасности, обеспечиваемой реализацией агента.

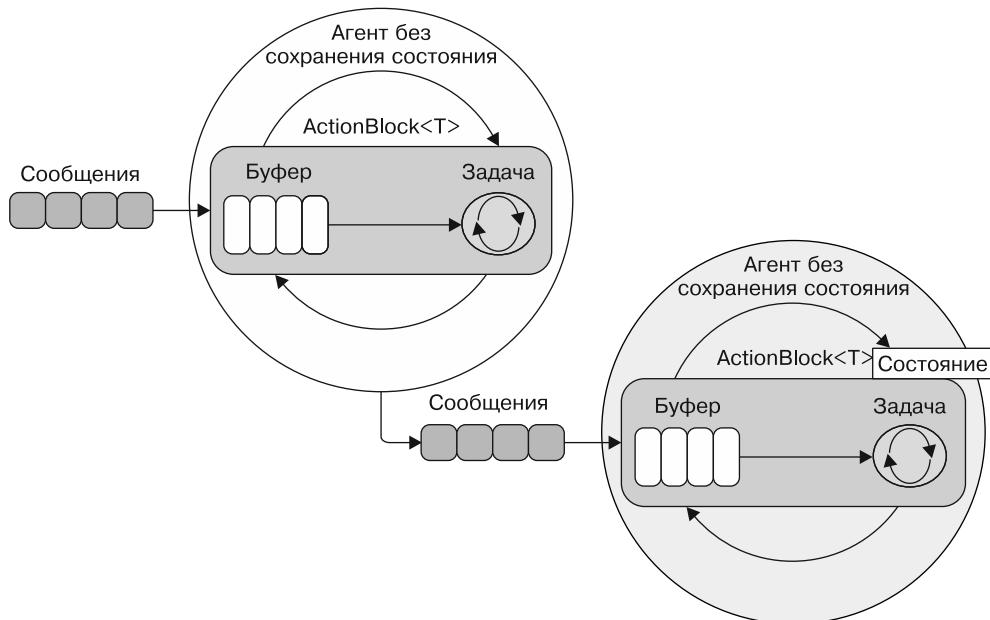


Рис. 12.7. Агенты с сохранением и без сохранения состояния, реализованные с использованием TDF-блока `ActionBlock`. У агента с сохранением состояния есть внутреннее изолированное произвольное значение, позволяющее хранить в памяти состояние, которое может изменяться

В листинге 12.5 показана реализация класса `StatefulDataFlowAgent`, который описывает универсальный агент с сохранением состояния, инкапсулирующего TDF-блок `AgentBlock` для обработки и хранения значений типа (выделены жирным шрифтом).

Листинг 12.5. Реализация агента с сохранением состояния на C# с использованием TDF

```
class StatefulDataFlowAgent<TState, TMessage> : IAgent<TMessage>
{
    private TState state;
    private readonly ActionBlock<TMessage> actionBlock;

    public StatefulDataFlowAgent(
        TState initialState,
        Func<TState, TMessage, Task<TState>> action,
        CancellationTokenSource cts = null)
    {
        state = initialState;
        var options = new ExecutionDataFlowBlockOptions {
            CancellationToken = cts != null ?
                cts.Token : CancellationToken.None
        };
        actionBlock = new ActionBlock<TMessage>(
            async msg => state = await action(state, msg), options);
    }

    public Task Send(TMessage message) => actionBlock.SendAsync(message);
    public void Post(TMessage message) => actionBlock.Post(message);
}
```

Создание внутреннего ActionBlock, который действует как инкапсулированный агент

Использование асинхронной функции для определения поведения агента

Если в конструкторе не предоставлен маркер отмены, то предоставляется новый маркер

`CancellationToken` может остановить агент в любой момент, и это единственный необязательный параметр, передаваемый в конструктор. Функция `Func<TState, TMessage, Task<TState>>` применяется к каждому сообщению в сочетании с текущим состоянием. Когда операция завершается, текущее состояние обновляется и агент переходит к обработке следующего доступного сообщения. Эта функция ожидает асинхронную операцию, что видно по типу ее возвращаемого значения: `Task<TState>`.

ПРИМЕЧАНИЕ

В исходном коде к этой книге вы найдете несколько полезных вспомогательных функций и реализацию агентов с использованием TDF и конструкторов, которая поддерживает асинхронные и синхронные операции, — в листинге 12.5 эта реализация опущена по соображениям краткости.

Агент реализует наследование от интерфейса `IAgent<TMessage>`, который определяет два члена, `Post` и `Send`, используемые для передачи сообщений агенту синхронно или асинхронно соответственно:

```
public interface IAgent<TMessage>
{
    Task Send(TMessage message);
    void Post(TMessage message);
}
```

Для инициализации нового агента, представленного реализованным интерфейсом `IAgent<TMessage>`, используйте вспомогательную функцию фабрики `Start`, как в F#-объекте `MailboxProcessor`:

```
IAgent<TMessage> Start<TState, TMessage>(TState initialState,
➥ Func<TState, TMessage, Task<TState>> action,
➥ CancellationTokenSource cts = null) =>
    new StatefulDataFlowAgent<TState, TMessage>(initialState, action, cts);
```

Поскольку взаимодействие с агентом происходит только посредством передачи сообщения (`Post` или `Send`), основное назначение интерфейса `IAgent<TMessage>` состоит в том, чтобы избежать предоставления параметра типа для состояния, относящегося к деталям внутренней реализации агента.

В листинге 12.6 `agentStateful` — это экземпляр агента `StatefulDataFlowAgent`, получающего сообщение с веб-адресом, с которого агент должен асинхронно загрузить содержимое. Затем результат операции кэшируется в локальное состояние `ImmutableDictionary<string, string>` во избежание повторения идентичных операций. Например, сайт Google упоминается в коллекциях URL-адресов дважды, но загружается только один раз. В конце данного примера содержимое каждого сайта сохраняется в локальной файловой системе. Обратите внимание на то, что, кроме различных побочных эффектов, возникающих при загрузке и сохранении данных, в данной реализации нет других побочных эффектов. Изменения состояния фиксируются благодаря тому, что состояние всегда передается функции действия (или функции `Loop`) в качестве аргумента.

Листинг 12.6. Применение агента, основанного на TDF

```
List<string> urls = new List<string> {
    @"http://www.google.com",
    @"http://www.microsoft.com",
    @"http://www.bing.com",
    @"http://www.google.com"
};

var agentStateful = Agent.Start(ImmutableDictionary<string, string>.Empty,
    async (ImmutableDictionary<string, string> state, string url) => {
        if (!state.TryGetValue(url, out string content))
            using (var webClient = new WebClient())
                content = await webClient.DownloadStringTaskAsync(url);
            await File.WriteAllTextAsync(createFileNameFromUrl(url), content);
            return state.Add(url, content);
    }
);
return state;
});

urls.ForEach(url => agentStateful.Post(url));
```

Использование асинхронной анонимной функции для создания агента. Эта функция обрабатывает текущее состояние и полученное входное сообщение

Функция, которая играет роль поведения агента, возвращает обновленное состояние, чтобы отслеживать любые изменения, доступные при обработке следующего сообщения

12.4.1. Свертка состояний и сообщений агента: Aggregate

Текущее состояние агента является результатом сокращения всех сообщений, которые он получил до сих пор, с учетом начального состояния в качестве значения аккумулятора, и последующей обработки функцией сжатия. Этот агент можно представить как свертку (агрегатор) потока полученных сообщений во времени. Интересно, что конструктор `StatefulDataFlowAgent` имеет такую же сигнатуру и поведение, что и метод расширения `LINQ Enumerable.Aggregate`. В демонстрационных целях в следующем коде конструкция агента из предыдущей реализации заменена ее аналогом — `LINQ`-оператором `Aggregate`:

```
urls.Aggregate(ImmutableDictionary<string, string>.Empty,
    async (state, url) => {
    if (!state.TryGetValue(url, out string content))
        using (var webClient = new WebClient())
    {
        content = await webClient.DownloadStringTaskAsync(url);
        await File.WriteAllTextAsync(createFileNameFromUrl(url), content);
        return state.Add(url, content);
    }
    return state;
});
```

Как видим, основная логика не изменилась. Используя конструктор `StatefulDataFlowAgent`, который работает не с коллекцией, а с передачей сообщений, мы реализовали асинхронный блок сжатия, аналогичный `LINQ`-оператору `Aggregate`.

12.4.2. Агентное взаимодействие: параллельный счетчик слов

Согласно определению *актора*, данного Карлом Хьюиттом¹, одним из родоначальников модели актора: «Один актор — это не актор. Акторы существуют в виде систем». Это означает, что акторы образуют системы и коммуницируют друг с другом. То же правило относится и к агентам. Рассмотрим пример использования агентов, которые взаимодействуют между собой, чтобы подсчитать, сколько раз данное слово присутствует в наборе текстовых файлов (рис. 12.8).

Начнем с простого агента без сохранения состояния, который принимает сообщение в виде строки и печатает его. Этот агент можно использовать для ведения журнала состояний приложения, которое обслуживает последовательность сообщений:

```
IAgent<string> printer = Agent.Start((string msg) =>
    WriteLine($"{msg} on thread {Thread.CurrentThread.
    ManagedThreadId}"));
```

¹ Подробнее о Карле Эдди Хьюитте (Carl Eddie Hewett) читайте здесь: https://en.wikipedia.org/wiki/Carl_Hewitt.

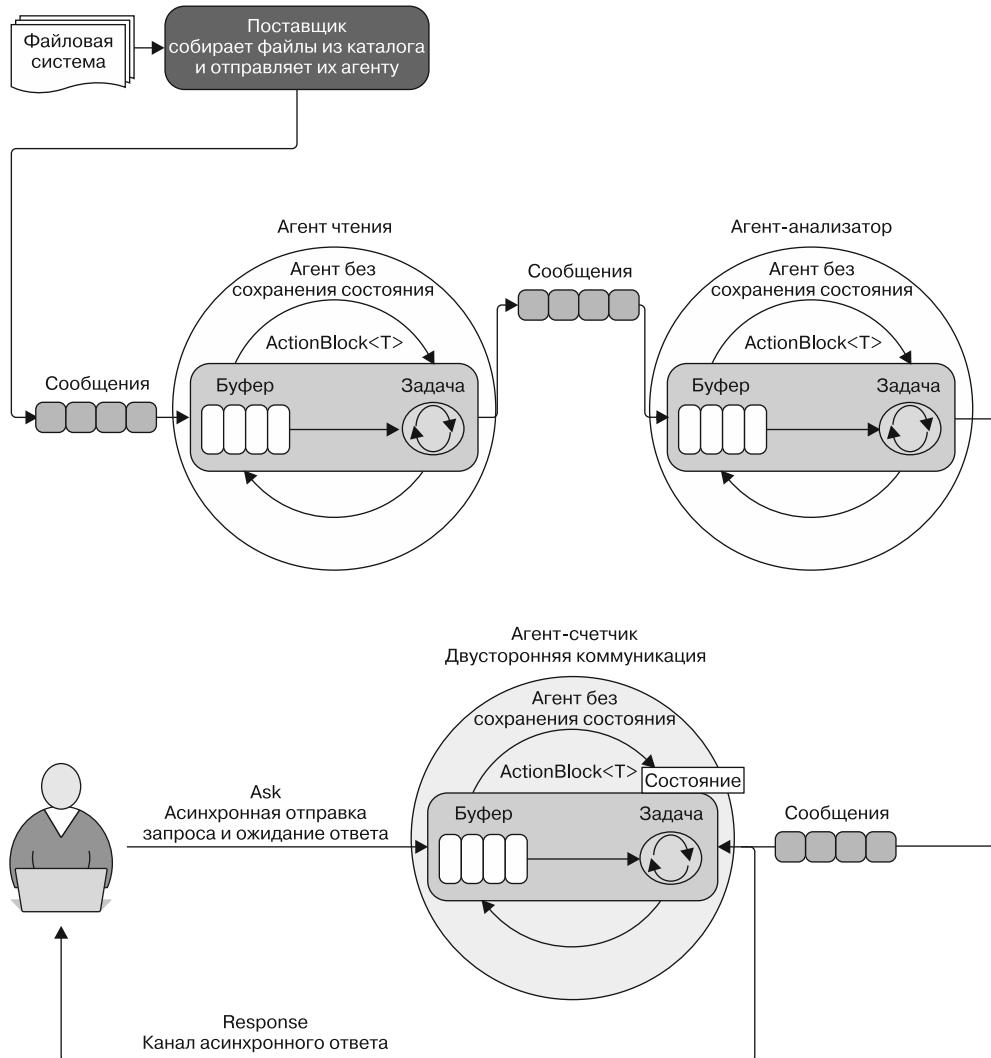


Рис. 12.8. Простое взаимодействие между агентами путем обмена сообщениями. Модель агентного программирования продвигает принцип единственной ответственности при написании кода. Обратите внимание, что агент-счетчик поддерживает двустороннюю коммуникацию, поэтому пользователь может в любой момент запросить (опросить) агента, отправив сообщение, и получить ответ в виде канала, который работает как асинхронный обратный вызов. После завершения операции обратный вызов предоставляет результат

Выходные данные также включают в себя идентификатор текущего потока для проверки нескольких используемых потоков. В листинге 12.7 показана реализация агентной системы для подсчета количества вхождений слов.

Листинг 12.7. Конвейер-счетчик слов с использованием агентов

```

Агент отправляет
журнал агенту печати
IAgent<string> reader = Agent.Start(async (string filePath) => {
    → await printer.Send("reader received message");

    var lines = await File.ReadAllLinesAsync(filePath); ←

    lines.ForEach(async line => await parser.Send(line));
});

→ Отправка всех строк из заданного текстового файла
агенту-анализатору. ForEach — метод расширения,
его реализацию вы найдете в исходном коде, выложенном онлайн
char[] punctuation = Enumerable.Range(0, 256).Select(c => (char)c)
    .Where(c => Char.IsWhiteSpace(c) || Char.IsPunctuation(c)).ToArray();

IAgent<string> parser = Agent.Start(async (string line) => {
    → await printer.Send("parser received message");
    foreach (var word in line.Split(punctuation))
        await counter.Send(word.ToUpper()); ←
});

→ Агент-анализатор разбивает
текст на отдельные слова
IReplyAgent<string, (string, int)> counter =
    Agent.Start(ImmutableDictionary<string, int>.Empty,
    (state, word) => {
        → printer.Post("counter received message");
        int count;
        if (state.TryGetValue(word, out count))
            return state.Add(word, count++); ←
        else return state.Add(word, 1);
    }, (state, word) => (state, (word, state[word])));

foreach (var filePath in Directory.EnumerateFiles(@"myFolder", "*.txt"))
    reader.Post(filePath);

→ Агент-счетчик проверяет,
существует ли данное слово
var wordCount_This = await counter.Ask("this");
var wordCount_Wind = await counter.Ask("wind");

```

Агент-счетчик допускает двустороннюю коммуникацию,
поэтому можно отправить сообщение с запросом
и асинхронно получить результат (ответ)

Система состоит из трех агентов, которые взаимодействуют между собой, образуя цепочку операций:

- агента чтения reader;
- агента-анализатора parser;
- агента-счетчика counter.

Процесс подсчета слов начинается с цикла `for-each` для отправки путей к файлам заданной папки первому агенту `reader`. Этот агент читает текст из файла, а затем отправляет каждую строку из данного текста агенту `parser`:

```
var lines = await File.ReadAllLinesAsync(filePath);
lines.ForEach(async line => await parser.Send(line));
```

Агент `parser` разбивает текстовое сообщение на отдельные слова, а затем передает каждое из этих слов последнему агенту — `counter`:

```
lines.Split(punctuation).ForEach(async word =>
    await counter.Send(word.ToUpper()));
```

Агент `counter` — агент с сохранением состояния, который выполняет работу по подсчету слов по мере их поступления.

Коллекция `ImmutableDictionary` определяет состояние агента `counter`, в котором хранятся слова, а также счетчик количества вхождений каждого слова. Для каждого полученного сообщения агент `counter` проверяет, существует ли данное слово во внутреннем состоянии `ImmutableDictionary<string, int>`, и либо увеличивает существующий счетчик, либо создает новый.

ПРИМЕЧАНИЕ

Преимущество использования агентной модели программирования для реализации подсчета слов состоит в том, что агент является потокобезопасным и может без проблем совместно задействоваться потоками, работающими со связанными текстами. Более того, применение неизменяемого `ImmutableDictionary` для хранения состояния позволяет передать состояние за пределы агента и продолжить обработку, не беспокоясь о том, что внутреннее состояние может оказаться непоследовательным или поврежденным.

Интересной особенностью агента `counter` является возможность асинхронно отвечать вызывающему объекту с помощью метода `Ask`. Можно в любое время запросить у агента результаты подсчета для определенного слова.

Интерфейс `IReplyAgent` — это уже знакомый нам интерфейс `IAgent`, функциональность которого дополнена методом `Ask`:

```
interface IReplyAgent<TMessage, TReply> : Iagent<TMessage>
{
    Task<TReply> Ask(TMessage message);
}
```

В листинге 12.8 показана реализация агента `StatefulReplyDataFlowAgent` с возможностью двусторонней коммуникации, в котором внутреннее состояние представлено одной полиморфной изменяемой переменной.

Данный агент имеет два разных поведения:

- для обработки метода отправки сообщения `Send`;
- для обработки метода `Ask`. Метод `Ask` отправляет сообщение и затем асинхронно ожидает ответа.

Эти поведения передаются в конструктор агента в форме параметрических делегатов Func. Первая функция (Func<TState, TMessage, Task<TState>>) обрабатывает каждое сообщение с учетом текущего состояния и соответствующим образом обновляет данное состояние. Такая логика идентична логике агента StatefulDataFlowAgent.

Листинг 12.8. Реализация агента без сохранения состояния на C# с использованием библиотеки TDF

```

Интерфейс IReplyAgent определяет метод Ask,
чтобы гарантировать способность агента
осуществлять двустороннюю коммуникацию

    Тип сообщения ActionBlock — это кортеж, внутри которого
    передается параметр TaskCompletionSource, чтобы обеспечить
    канал асинхронной обратной связи с вызывающим объектом

→ class StatefulReplyDataFlowAgent<TState, TMessage, TReply> :
    IReplyAgent<TMessage, TReply>
{
    private TState state;
    private readonly ActionBlock<(TMessage,
        Option<TaskCompletionSource<TReply>>)> actionBlock;

    public StatefulReplyDataFlowAgent(TState initialState,
        Func<TState, TMessage, Task<TState>> projection,
        Func<TState, TMessage, Task<(TState, TReply)>> ask,
        CancellationTokenSource cts = null)
        ← Конструкция агента принимает две функции,
        определяющие связь типа «отправить и забыть»,
        и двустороннюю связь соответственно

    {
        state = initialState;
        var options = new ExecutionDataFlowBlockOptions {
            CancellationToken = cts?.Token ?? CancellationToken.None };

        actionBlock = new ActionBlock<(TMessage,
            Option<TaskCompletionSource<TReply>>)>(
            async message => {
                (TMessage msg, Option<TaskCompletionSource<TReply>> replyOpt) =
                → message;                                Если TaskCompletionSource имеет значение None,
                                                    то применяется функция проекции

                → await replyOpt.Match(
                    None: async () => state = await projection(state, msg), ←
                    Метод расширения
                    Some: async reply => {                            Если TaskCompletionSource имеет
                        (TState newState, TReply replyresult) = await ask(state, msg);
                        state = newState;
                        reply.SetResult(replyresult);                  значение Some, то для ответа вызывающему
                                                                объекту используется функция Ask
                    });
                    }, options);
                }

    public Task<TReply> Ask(TMessage message)
    {

```

```

var tcs = new TaskCompletionSource<TReply>();
actionBlock.Post((message, Option.Some(tcs)));
return tcs.Task;
}

public Task Send(TMessage message) =>
    actionBlock.SendAsync((message, Option.None));
}

```

Функция-член Ask создает TaskCompletionSource, применяемый в качестве канала для обратной связи с вызывающим объектом после завершения операции, выполняемой агентом

Вторая функция (`Func<TState, TMessage, Task<(TState, TReply)>>`), наоборот, обрабатывает входящие сообщения, вычисляет новое состояние агента и в итоге отвечает отправителю. Тип выходных данных этой функции — кортеж, который содержит состояние агента, включая указатель (обратный вызов), действующий как отклик (ответ). Кортеж обернут в тип `Task`, чтобы можно было ожидать его без блокировки, как любую асинхронную функцию.

При создании сообщения `Ask` для опроса агента отправитель передает вместе с сообщением экземпляр `TaskCompletionSource<TReply>`, и функция `Ask` возвращает вызывающему объекту ссылку. Объект `TaskCompletionSource` является основой при предоставлении канала для асинхронной обратной связи с отправителем посредством обратного вызова; когда результат вычисления готов, обратный вызов получает уведомление от агента. Эта модель эффективно генерирует двустороннюю коммуникацию.

ПРИМЕЧАНИЕ

TDF не гарантирует встроенную изоляцию. Следовательно, неизменяемое состояние может быть общедоступным в рамках всей функции обработки и может изменяться за пределами области действия агента, что приводит к нежелательному поведению. Настоятельно рекомендуется стараться ограничивать и контролировать доступ к общедоступному изменяемому состоянию.

Чтобы `StatefulReplyDataFlowAgent` мог поддерживать оба типа коммуникации — одностороннюю `Send` и двустороннюю `Ask`, — в состав сообщения включается тип параметра `TaskCompletionSource`. Таким образом агент определяет, получено сообщение из метода `Post` (`TaskCompletionSource` имеет значение `None`) или с помощью метода `Ask` (`TaskCompletionSource` имеет значение `Some`). Для выбора соответствующего поведения агента используется метод расширения `Match` типа `Option` с сигнатурой `Match<T, R>(None: Action<T>, Some(item): Func<T, R>(item))`.

12.5. Параллельный рабочий процесс для сжатия и шифрования больших потоков

В этом разделе мы построим полностью асинхронный и распараллеленный рабочий процесс в сочетании с агентной моделью программирования, чтобы продемонстрировать возможности библиотеки TDF. В данном примере используется сочетание TDF-блоков и агента `StatefulDataFlowAgent`, связанных для работы в качестве параллельного конвейера. Цель данного примера — проанализировать и спроектировать реальное приложение. Затем мы определим проблемы, возникшие при разработке программы, и исследуем, как можно их решить, применив при разработке библиотеку TDF.

Библиотека TDF обрабатывает блоки, составляющие рабочий процесс, параллельно, с разными скоростями. Что еще более важно, она эффективно распределяет работу между несколькими ядрами процессора, чтобы обеспечить максимальную скорость вычислений и общую масштабируемость. Это особенно полезно, когда нужно обработать большой поток байтов, который может генерировать сотни или даже тысячи блоков данных.

12.5.1. Контекст: проблема обработки большого потока данных

Допустим, нам нужно сжать большой файл, чтобы упростить его хранение или передачу по сети, или нужно зашифровать содержимое файла для защиты этой информации. Часто необходимо применить и сжатие, и шифрование. Такие операции могут занять много времени, если обрабатывать весь файл одновременно. Более того, сложно перемещать файл или потоковые данные по сети, и сложность задачи увеличивается вместе с размером файла вследствие таких внешних факторов, как задержки и непредсказуемая пропускная способность. Кроме того, если файл передается за одну транзакцию и что-то пойдет не так, то операция будет пытаться отправить весь файл заново, что может потребовать много времени и ресурсов. В следующих разделах мы будем решать эту проблему шаг за шагом.

В .NET сжать файл размером более 4 Гбайт нелегко вследствие ограничений фреймворка, накладываемых на объем данных, подлежащих сжатию. Из-за максимального адресуемого размера, определяемого 32-разрядным указателем, если создать массив размером свыше 4 Гбайт, то возникнет исключение `OutOfMemoryArray`. Начиная с .NET 4.5, на 64-разрядных платформах для обработки массивов размером более 4 Гбайт доступен параметр `gcAllowVeryLargeObjects` (<http://mng.bz/x0c4>). Он позволяет 64-битным приложениям обрабатывать многомерные массивы размером `UInt32.MaxValue` (4 294 967 295) элементов. С технической точки зрения можно применить стандартное сжатие GZip, которое используется для сжатия потоков

байтов, к массиву данных размером более 4 Гбайт; но дистрибутив GZip по умолчанию не поддерживает такую возможность. Соответствующий класс .NET `GZipStream` также унаследовал это ограничение 4 Гбайт.

Как сжать и зашифровать большой файл, не ограничиваясь пределом 4 Гбайт, установленным классами инфраструктуры? Практическое решение требует создать подпрограмму разбиения на порции, чтобы потом иметь дело с потоками данных меньшего размера. Разделение потока данных позволяет сжать и/или зашифровать каждую порцию по отдельности, а потом записать содержимое порции в выходной поток. Метод разбиения на порции подразумевает разделение данных, как правило, на фрагменты одинакового размера, применение к каждому фрагменту соответствующего преобразования (сначала сжатие, потом шифрование), соединение фрагментов в правильном порядке и сжатие данных. Крайне важно гарантировать правильный порядок порций при повторной сборке в конце рабочего процесса. Вследствие интенсивных асинхронных операций ввода-вывода пакеты могут поступать в неправильной последовательности, особенно если данные передаются по сети. Необходимо проверить последовательность пакетов во время повторной сборки (рис. 12.9).

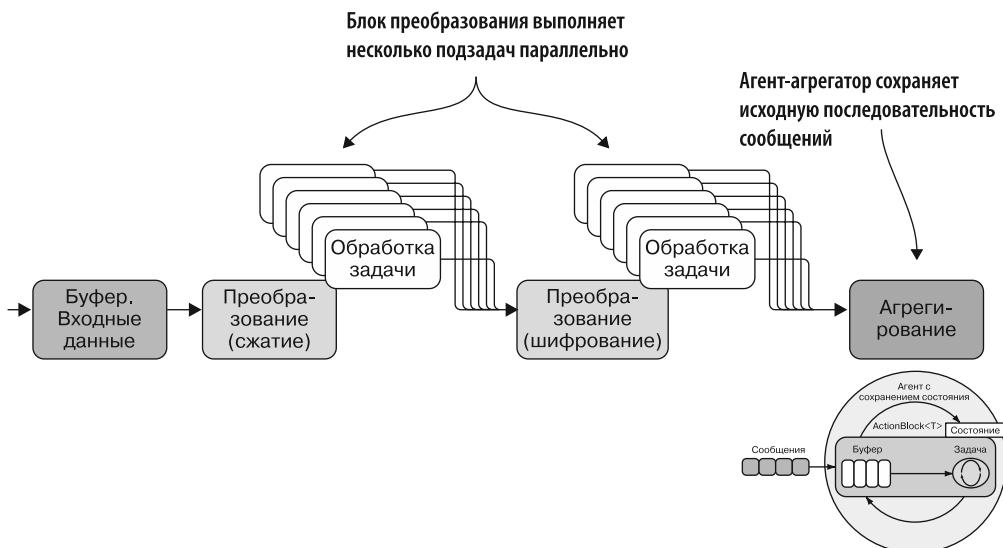


Рис. 12.9. Блоки преобразования обрабатывают сообщения параллельно. Когда операция завершается, результат отправляется в следующий блок. Назначение агента-агрегатора — сохранить исходную последовательность сообщений аналогично методу расширения PLINQ `AsOrdered`

Возможность распараллеливания естественным образом вписывается в эту структуру, поскольку порции данных могут обрабатываться независимо друг от друга.

Шифрование и сжатие: последовательность имеет значение

Может показаться, что, поскольку операции сжатия и шифрования не зависят одна от другой, не имеет значения, в каком порядке они применяются к файлу. Однако это не так. *Порядок, в котором выполняются операции сжатия и шифрования, критически важен.*

Шифрование приводит к тому, что входные данные превращаются в данные с высокой энтропией, которая является мерой непредсказуемости информационного содержимого. Таким образом, зашифрованные данные выглядят как случайный массив байтов, что затрудняет поиск общих закономерностей. И наоборот, алгоритмы сжатия работают лучше всего, когда в данных встречаются похожие фрагменты, которые можно выразить меньшим количеством байтов.

Когда данные нужно и сжать, и зашифровать, то, чтобы достичь наилучших результатов, нужно сначала их сжать, а потом зашифровать. Таким образом алгоритм сжатия сможет найти похожие фрагменты, которые можно сократить, а затем алгоритм шифрования создаст фрагменты данных почти такого же размера. Кроме того, если порядок операций предусматривает сначала сжатие, а потом шифрование, то не только в результате получим файл меньшего размера, но и шифрование, скорее всего, займет меньше времени, поскольку будет работать с меньшим количеством данных.

В листинге 12.9 показана полная реализация параллельного рабочего процесса сжатия-шифрования. Обратите внимание, что в исходном коде к книге вы найдете пример обратного рабочего процесса для дешифрования и распаковки данных, а также примеры использования асинхронных вспомогательных функций для сжатия и шифрования массива байтов.

Листинг 12.9. Параллельное сжатие и шифрование в потоке с использованием TDF

```
async Task CompressAndEncrypt(
    Stream streamSource, Stream streamDestination,
    long chunkSize = 1048576, CancellationTokenSource cts = null)
{
    cts = cts ?? new CancellationTokenSource(); ← Если в конструкторе не указан маркер
                                                отмены, то предоставляется новый

    var compressorOptions = new ExecutionDataflowBlockOptions {
        MaxDegreeOfParallelism = Environment.ProcessorCount,
        BoundedCapacity = 20, ← Установка значения BoundedCapacity, чтобы ограничить
        CancellationToken = cts.Token ← количество сообщений и сократить потребление памяти,
    }; ← ограничивая число одновременно создаваемых
          экземпляров MemoryStream

    var inputBuffer = new BufferBlock<CompressingDetails>(
        new DataflowBlockOptions {
            CancellationToken = cts.Token, BoundedCapacity = 20 }); ← Асинхронное сжатие данных
                                                               (метод приведен в исходном коде,
                                                               выложенном онлайн)

    var compressor = new TransformBlock<CompressingDetails,
        CompressedDetails>(async details => {
            var compressedData = await IOUtils.Compress(details.Bytes); ←

```

```
    return details.ToCompressedDetails(compressedData);
}, compressorOptions);
```

Объединение данных и метаданных в шаблон
байтового массива, который будет деконструирован
и проанализирован во время операций дешифрования и распаковки

```
var encryptor = new TransformBlock<CompressedDetails, EncryptDetails>(
    async details => {
        byte[] data = IOUtils.CombineByteArrays(details.CompressedContentSize,
            details.ChunkSize, details.Bytes);
        var encryptedData = await IOUtils.Encrypt(data);
        return details.ToEncryptDetails(encryptedData);
    }, compressorOptions);
```

Асинхронное шифрование данных (метод приведен
в исходном коде, выложенном онлайн)

Преобразование текущей
структуре данных в форму
сообщения для отправки в следующий блок

```
var asOrderedAgent = Agent.Start((new Dictionary<int, EncryptDetails>(), 0),
    async((Dictionary<int, EncryptDetails>, int) state, EncryptDetails msg)=>{
```

Dictionary<int, EncryptDetails> details, int lastIndexProc) = state;
details.Add(msg.Sequence, msg);

Поведение агента asOrderedAgent
отслеживает порядок получения
сообщений, чтобы сохранить его
(и обеспечить целостность данных)

```
while (details.ContainsKey(lastIndexProc + 1)) {
    msg = details[lastIndexProc + 1];
    await streamDestination.WriteAsync(msg.EncryptedContentSize, 0,
        msg.EncryptedContentSize.Length);
    await streamDestination.WriteAsync(msg.Bytes, 0,
        msg.Bytes.Length);
```

Асинхронное сохранение данных;
файловый поток можно заменить сетевым
потоком для отправки данных по сети

```
    lastIndexProc = msg.Sequence;
    details.Remove(lastIndexProc);
}
return (details, lastIndexProc);
}, cts);
```

Обработанная порция данных удаляется
из локального состояния, одновременно
отслеживаются элементы, подлежащие обработке

Блок чтения ActionBlock отправляет агенту asOrdered порцию
данных, обернутую в структуру, содержащую
также дополнительную информацию об этих данных

```
var writer = new ActionBlock<EncryptDetails>(async details => await
    asOrderedAgent.Send(details), compressorOptions);
```

```
var linkOptions = new DataflowLinkOptions { PropagateCompletion = true };
inputBuffer.LinkTo(compressor, linkOptions);
compressor.LinkTo(encryptor, linkOptions);
encryptor.LinkTo(writer, linkOptions);
```

Соединенные блоки потока данных
образуют рабочий процесс

```
long sourceLength = streamSource.Length;
byte[] size = BitConverter.GetBytes(sourceLength);
await streamDestination.WriteAsync(size, 0, size.Length);
```

Общий размер файлового потока сохраняется как первая
порция данных; таким образом, алгоритм распаковки знает,
как извлечь информацию и когда нужно прекратить работу

```
chunkSize = Math.Min(chunkSize, sourceLength);
```

Определение размера порции
данных при фрагментации

```

int indexSequence = 0;
while (sourceLength > 0) {
    byte[] data = new byte[chunkSize];
    int readCount = await streamSource.ReadAsync(data, 0, data.Length); ←
    byte[] bytes = new byte[readCount];
    Buffer.BlockCopy(data, 0, bytes, 0, readCount);
    var compressingDetails = new CompressingDetails {
        Bytes = bytes,
        ChunkSize = BitConverter.GetBytes(readCount),
        Sequence = ++indexSequence
    };
    await inputBuffer.SendAsync(compressingDetails); ←
    sourceLength -= readCount; ←
    if (sourceLength < chunkSize)
        chunkSize = sourceLength; ←
    if (sourceLength == 0)
        → inputBuffer.Complete();
}

await inputBuffer.Completion.ContinueWith(task => compressor.Complete());
await compressor.Completion.ContinueWith(task => encryptor.Complete());
await encryptor.Completion.ContinueWith(task => writer.Complete());
await writer.Completion;
await streamDestination.FlushAsync();
}

```

Чтение исходного потока и формирование очередной порции данных, и так до конца потока

Отправка порции данных, прочитанной из исходного потока, в inputBuffer

Проверка текущей позиции исходного потока после каждой операции чтения, чтобы решить, когда следует завершить операцию

Уведомление входного буфера о том, что достигнут конец исходного потока

Функция `CompressAndEncrypt` принимает в качестве аргумента исходный поток, подлежащий обработке, и поток-приемник; аргумент `chunkSize` определяет размер блоков данных, на которые будет разделен исходный массив (по умолчанию, если не указано иное значение, — 1 Мбайт), а `CancellationTokenSource` останавливает выполнение потока данных в любой точке. Если значение `CancellationTokenSource` не задано, то создается новый маркер, который распространяется по операциям потока данных.

Ядро функции состоит из трех TDF-блоков и агента с сохранением состояния, завершающего рабочий процесс. Переменная `inputBuffer` имеет тип `BufferBlock`. Он, как следует из названия, буферизует входящие порции байтов, считанные из исходного потока, и хранит эти элементы для передачи следующему блоку в потоке, представляющем собой связанный компрессор `TransformBlock` (код, на который стоит обратить внимание, выделен жирным шрифтом).

Считанные из потока байты отправляются в блок буфера посредством метода `SendAsync`:

```

var compressingDetails = new CompressingDetails {
    Bytes = bytes,
    ChunkSize = BitConverter.GetBytes(chunkSize),
}

```

```
Sequence = ++indexSequence
};

await buffer.SendAsync(compressingDetails);
```

Каждая порция байтов, считанная из исходного потока, обертыивается в структуру данных `CompressingDetails`, которая содержит дополнительную информацию о размере байтового массива. Это монотонное значение затем используется для сохранения порядка в последовательности генерируемых порций данных. *Монотонное значение* — функция упорядоченных наборов данных, которая сохраняет или обращает заданное значение, и это значение всегда либо уменьшается, либо увеличивается. Порядок блоков важен как для корректного выполнения операций сжатия и шифрования, так и для корректного дешифрования и распаковки в исходную форму.

В общем случае, если назначением блока является только передача операций с элементами из одного блока в несколько других, то `BufferBlock` не нужен. Но при чтении большого или непрерывного потока данных такой блок полезен, так как позволяет справиться с обратным давлением, генерируемым огромным количеством данных, разделенных для обработки, путем установки соответствующего значения `BoundedCapacity`. В этом примере емкость `BoundedCapacity` ограничена 20 элементами. Если в блоке уже есть 20 элементов, то он перестает принимать новые, пока один из существующих элементов не будет передан в следующий блок. Поскольку источником потока данных является асинхронная операция ввода-вывода, существует риск обработки больших объемов данных. Рекомендуется ограничить внутреннюю буферизацию, чтобы иметь возможность регулировать количество данных, присвоив параметру `BoundedCapacity` соответствующее значение при создании `BufferBlock`.

Следующими двумя типами блоков являются преобразование сжатия и преобразование шифрования. На первом этапе (сжатие) `TransformBlock` применяет сжатие к порции байтов и добавляет в полученное сообщение `CompressingDetails` соответствующую информацию о данных, которая включает в себя сжатый байтовый массив и его размер. Эта информация сохраняется как часть выходного потока, доступного во время распаковки.

На втором этапе (шифрование) порция сжатого байтового массива шифруется и создается последовательность байтов, являющаяся результатом компоновки трех массивов: `CompressedDataSize`, `ChunkSize` и массива данных. Эта структура содержит инструкции для алгоритмов распаковки и дешифрования, согласно которым данные алгоритмы извлекают из потока правильную порцию байтов.

ПРИМЕЧАНИЕ

Помните, что при наличии нескольких TDF-блоков некоторые TDF-задачи могут простоять, пока выполняются другие задачи, поэтому необходимо настроить параметр выполнения блока, чтобы избежать потенциальных простоев. Детали этой оптимизации будут описаны в следующем подразделе.

12.5.2. Сохранение порядка в потоке сообщений

Документация TDF гарантирует, что `TransformBlock` будет распространять сообщения в том же порядке, в каком они поступили. Внутри `TransformBlock` использует буфер переупорядочения, позволяющий исправить любые нарушения порядка, которые могут возникнуть из-за конкурентной обработки сообщений. К сожалению, вследствие большого количества интенсивных асинхронных операций ввода-вывода, выполняемых параллельно, сохранение порядка сообщений в данном случае неприменимо. Вот почему мы предусмотрели дополнительное сохранение порядка в последовательности, используя монотонные значения.

Если вы решите *отправлять* или передавать данные *в потоке* по сети, то гарантия доставки пакетов в правильной последовательности теряется вследствие таких факторов, как непредсказуемая пропускная способность и ненадежное сетевое соединение. Для того чтобы сохранить порядок при обработке порций данных, последним этапом рабочего процесса является агент `asOrderedAgent` с отслеживанием состояния. Этот агент ведет себя как *мультиплексор*, который собирает элементы и сохраняет их в локальной файловой системе, восстанавливая правильную последовательность. Значение порядкового номера последовательности хранится в свойстве структуры данных `EncryptDetails`, которая принимается агентом в виде сообщения.

Шаблон мультиплексора

Мультиплексор — шаблон, обычно используемый в сочетании с конструкцией «поставщик — потребитель». Этот шаблон позволяет потребителю, который в предыдущем примере является последним этапом конвейера, получать порции данных в правильной последовательности. Порции данных не нужно сортировать или переупорядочивать. Тот факт, что очередь каждого поставщика (TDF-блока) локально упорядочена, позволяет мультиплексору искать следующее значение (сообщение) в последовательности. Мультиплексор ожидает сообщения от блока-поставщика потока данных. Когда приходит порция данных, мультиплексор проверяет, является ли порядковый номер указанной порции следующим в ожидаемой последовательности. Если это так, то мультиплексор сохраняет данные в локальной файловой системе. Если же порция данных не представляет собой очередную ожидаемую порцию в последовательности, то мультиплексор сохраняет ее значение во внутреннем буфере и повторяет операцию анализа для следующего полученного сообщения. Подобный алгоритм позволяет мультиплексору собирать входные данные из входящих сообщений от поставщика таким образом, чтобы гарантировать последовательный порядок без сортировки значений.

Точность всех вычислений требует сохранения порядка исходной последовательности и разделов, чтобы обеспечить согласованность порядка во время объединения.

ПРИМЕЧАНИЕ

В случае отправки порций данных по сети применяется та же стратегия сохранения данных, что и в локальной файловой системе, только агент-приемник находится на другом конце сетевого кабеля.

Состояние этого агента сохраняется с помощью кортежа. Первым элементом кортежа является коллекция `Dictionary<int, EncryptDetails>`, где ключами выступают номера в исходной последовательности, в которой были отправлены данные. Второй элемент, `lastIndexProc`, представляет собой индекс последнего обработанного элемента, предотвращающий повторную обработку одних и тех же порций данных. Тело `asOrderedAgent` выполняет цикл `while`, который использует это значение `lastIndexProc` и гарантирует, что обработка порций данных начинается с последнего необработанного элемента. Перебор в цикле продолжается до тех пор, пока не будет продолжен порядок элементов; в противном случае цикл прерывается и ожидает следующего сообщения, которое может заполнить пробел в последовательности.

Агент `asOrderedAgent` подключается к рабочему процессу через TDF-блок записи `ActionBlock`, который отправляет его в структуру данных `EncryptDetails` для завершения работы.

GZipStream или DeflateStream: что выбрать?

.NET Framework предоставляет несколько классов с разными вариантами сжатия потока байтов. В листинге 12.9 для модуля сжатия используется `System.IO.Compression.GZipStream`: вместо него можно было бы с тем же успехом задействовать `System.IO.Compression.DeflateStream`. Начиная с .NET Framework 4.5, в потоке сжатия `DeflateStream` используется библиотека zlib, что обеспечивает лучший алгоритм сжатия и в большинстве случаев меньший размер сжатых данных по сравнению с более ранними версиями. Оптимизация алгоритма сжатия `DeflateStream` обеспечивает обратную совместимость с данными, сжатыми по алгоритмам более ранних версий. Одна из причин для выбора класса `GZipStream` состоит в том, что он добавляет к сжатым данным циклический контроль избыточности (Cyclic Redundancy Check, CRC), что помогает определить, не были ли данные повреждены. Подробнее об этих потоках читайте в онлайн-документации MSDN (<http://mng.bz/h082>).

12.5.3. Связывание, распространение и завершение

TDF-блоки в рабочем потоке сжатия-шифрования связаны посредством метода расширения `LinkTo`, который по умолчанию распространяет только данные (сообщения). Но если рабочий процесс является линейным, как в настоящем примере, то хорошей практикой будет обмен информацией между блоками посредством автоматического уведомления, — допустим, в тех случаях, когда работа прекращается или когда случаются всевозможные ошибки. Такое поведение достигается путем построения метода `LinkTo` с необязательным аргументом `DataFlowLinkOptions`

и свойством `PropagateCompletion`, значение которого равно `true`. Вот код из предыдущего примера, куда встроен этот параметр:

```
var linkOptions = new DataFlowLinkOptions { PropagateCompletion = true };

inputBuffer.LinkTo(compressor, linkOptions);
compressor.LinkTo(encryptor, linkOptions);
encryptor.LinkTo(writer, linkOptions);
```

Необязательное свойство `PropagateCompletion` информирует блок потока данных об автоматической передаче его результатов и исключений на следующий этап после того, как этот блок завершит работу. Подобное достигается путем вызова метода `Complete`, когда блок буфера сформирует уведомление о завершении при достижении конца потока:

```
if (sourceLength < chunkSize)
    chunkSize = sourceLength;
if (sourceLength == 0)
    buffer.Complete();
```

Затем все блоки потока данных объявляются в виде каскада как цепочка, которую завершил процесс:

```
await inputBuffer.Completion.ContinueWith(task => compressor.Complete());
await compressor.Completion.ContinueWith(task => encryptor.Complete());
await encryptor.Completion.ContinueWith(task => writer.Complete());
await writer.Completion;
```

В итоге можно выполнить код следующим образом:

```
using (var streamSource = new FileStream(sourceFile, FileMode.OpenOrCreate,
                                         FileAccess.Read, FileShare.None, useAsync:
                                         ➔ true))
using (var streamDestination = new FileStream(destinationFile,
                                              FileMode.Create, FileAccess.Write, FileShare.None, useAsync: true))
    await CompressAndEncrypt(streamSource, streamDestination)
```

В табл. 12.1 показаны тесты производительности для сжатия и шифрования файлов разных размеров, а также для обратной операции дешифрования и распаковки. Результатом тестирования является средняя длительность каждой операции, выполненной три раза.

Таблица 12.1. Тестирование производительности для сжатия и шифрования файлов разных размеров

Размер файла, Гбайт	Степень параллелизма	Время сжатия-шифрования, с	Время дешифрования-распаковки, с
3	1	524,56	398,52
3	4	123,64	88,25
3	8	69,20	45,93
12	1	2249,12	1417,07
12	4	524,60	341,94
12	8	287,81	163,72

12.5.4. Правила построения рабочего процесса TDF

Ниже приведены несколько хороших правил и рекомендаций для успешного применения TDF в вашем рабочем процессе.

- ❑ *Делайте что-то одно и делайте это хорошо.* Так звучит принцип современного ООП – *принцип единственной ответственности* (https://ru.wikipedia.org/wiki/Принцип_единственной_ответственности). Идея состоит в том, чтобы блок выполнял только одно действие и имел только одну причину для изменения.
- ❑ *Рассчитывайте на компоновку.* В мире ООП этот принцип известен как *принцип открытости/закрытости* (https://ru.wikipedia.org/wiki/Принцип_открытости/закрытости), согласно которому строительные блоки потока данных разрабатываются так, чтобы быть открытыми для расширения, но закрытыми для модификации.
- ❑ *DRY.* Этот принцип (Don't Repeat Yourself – «не повторяйся») требует писать многократно используемый код и многократно используемые компоненты строительных блоков потока данных.

Как повысить производительность: переработанный **MemoryStream**

Языки программирования .NET полагаются на сборку мусора методом маркировки и очистки, что может отрицательно сказываться на производительности программы, которая под давлением сборщика мусора вынуждена генерировать большое количество операций выделения памяти. Приходится расплачиваться снижением производительности за то, что в коде (например, в листинге 12.9) создается отдельный экземпляр `System.IO.MemoryStream` для каждой операции сжатия и шифрования, включая внутренний байтовый массив.

Количество экземпляров `MemoryStream` увеличивается по мере увеличения количества обрабатываемых порций данных, число которых в большом потоке или файле может достигать нескольких сотен. По мере роста байтового массива `MemoryStream` изменяет его размер, выделяя в памяти все больше места, а затем копируя туда исходные байты. Это неэффективно не только потому, что создаются новые объекты и выбрасываются старые, но и потому, что при каждом изменении размера приходится заново выполнять всю работу по копированию контента.

Один из способов уменьшить нагрузку на память, вызываемую частым созданием и уничтожением крупных объектов, – дать указание сборщику мусора .NET уплотнить кучу больших объектов (Large Object Heap, LOH), используя следующий параметр:

```
GCSettings.LargeObjectHeapCompactionMode =  
GCLargeObjectHeapCompactionMode.CompactOnce
```

Такое решение может уменьшить объем памяти приложения, однако оно никак не решает первоначальную проблему выделения всей этой памяти. Более эффективным решением является создание пула объектов, также называемого пулом буферов, для предварительного выделения произвольного числа объектов `MemoryStream`, которые

можно использовать многократно (пул многократно используемых объектов общего назначения будет описан в главе 13).

Microsoft представила новый объект под названием `RecyclableMemoryStream`, который абстрагируется от реализации пула объектов, оптимизированного для `MemoryStream`; он сводит к минимуму количество выделений памяти в куче больших объектов и фрагментацию памяти. Обсуждение `RecyclableMemoryStream` выходит за рамки этой книги. Для получения дополнительной информации о нем обратитесь к онлайн-документации MSDN.

12.5.5. Реактивные расширения генерации сетки (Rx) и TDF

TDF и реактивные расширения Rx (описанные в главе 6) имеют важные общие черты, несмотря на наличие независимых характеристик и сильных сторон. Эти библиотеки дополняют друг друга, что упрощает их интеграцию. TDF ближе к агентной модели программирования, ориентированной на предоставление строительных блоков для передачи сообщений, что упрощает реализацию параллельных приложений с интенсивной нагрузкой на процессор и средства ввода-вывода, с высокой пропускной способностью и низкой задержкой, а также предоставляет разработчикам возможность явного контроля над буферизацией данных. Реактивные расширения Rx ближе к функциональной парадигме; они предоставляют широкий набор операторов, предназначенных главным образом для координации и компоновки потоков событий с помощью API на основе LINQ.

В TDF встроена поддержка интеграции с Rx, что позволяет представлять блоки потока исходных данных как наблюдаемые и как наблюдателей. Метод расширения `AsObservable` преобразует TDF-блоки в последовательность наблюдаемых объектов, что позволяет эффективно передавать выходные данные цепочки потока данных в произвольный набор реактивных методов расширения для дальнейшей обработки. В частности, метод расширения `AsObservable` создает `IEnumerable<T>` для `ISourceBlock<T>`.

ПРИМЕЧАНИЕ

TDF-блоки также могут выступать в качестве наблюдателей. Метод расширения `AsObserver` создает `IObserver<T>` для `ITargetBlock<T>`, где `OnNext` вызывает наблюдатель, что приводит к отправке данных приемнику. Вызовы `OnError` создают исключение ошибки приемника, а вызовы `OnCompleted` приводят к вызову `Complete` в приемнике.

Рассмотрим интеграцию Rx и TDF на практике. В листинге 12.9 последний блок в параллельном потоке сжатия-шифрования — это `asOrderedAgent` с сохранением со-

стояния. Особенностью данного компонента является наличие внутреннего состояния, отслеживающего полученные сообщения и их порядок. Как уже упоминалось, сигнатура конструкции агента с сохранением состояния подобна LINQ-оператору **Aggregate**, который в терминах Rx можно заменить RX-оператором. Этот оператор был рассмотрен в главе 6.

В листинге 12.10 показана интеграция Rx и TDF путем замены агента **asOrderedAgent** из последнего блока параллельного рабочего потока сжатия-шифрования.

Листинг 12.10. Интеграция реактивных расширений с TDF

```
inputBuffer.LinkTo(compressor, linkOptions);
compressor.LinkTo(encryptor, linkOptions);

encrptor.AsObservable()                                ← Запуск интеграции Rx с TDF
    .Scan((new Dictionary<int, EncryptDetails>(), 0),
          (state, msg) => Observable.FromAsync(async() => {
            (Dictionary<int,EncryptDetails> details, int lastIndexProc) = state;
            details.Add(msg.Sequence, msg);
            while (details.ContainsKey(lastIndexProc + 1)) {           ← Запуск асинхронной
              msg = details[lastIndexProc + 1];                         Rx-операции Scan
              await streamDestination.WriteAsync(msg.EncryptedDataSize, 0,
                msg.EncryptedDataSize.Length);
              await streamDestination.WriteAsync(msg.Bytes, 0, msg.Bytes.Length);
              lastIndexProc = msg.Sequence;
              details.Remove(lastIndexProc);
            }
            return (details, lastIndexProc);                            ← Подписка Rx на TaskPoolScheduler
          }) .SingleAsync().Wait()
    .SubscribeOn(TaskPoolScheduler.Default).Subscribe();      ←
```

Как видим, мы заменили агент **asOrderedAgent** на Rx-оператор **Observable.Scan** без изменения внутренней функциональности. TDF-блоки и наблюдаемые Rx-потоки могут быть завершены успешно или с ошибками, а метод **AsObservable** преобразует завершение блока (успешное или с ошибкой) в завершение наблюдаемого потока. Но если в блоке возникает ошибка с исключением, то это исключение при передаче наблюдаемому потоку будет обернуто в **AggregateException** — подобно распространению ошибок между связанными блоками.

Резюме

- ❑ Система, написанная с применением TPL Dataflow, использует преимущества многоядерности, поскольку все блоки, составляющие рабочий процесс, могут работать параллельно.
- ❑ TDF обеспечивает эффективные технологии для выполнения естественных параллельных задач, когда очевидно, что многие независимые вычисления могут выполняться параллельно.
- ❑ Библиотека TDF имеет встроенную поддержку регулирования и асинхронности, что позволяет улучшить операции с ограничениями как ввода-вывода, так и процессора. В частности, эта библиотека позволяет создавать адаптивные клиентские приложения, сохраняя преимущества массовой параллельной обработки.
- ❑ Библиотека TDF может использоваться для распараллеливания рабочего процесса при сжатии и шифровании большого потока данных путем обработки блоков с различной скоростью.
- ❑ Комбинация и интеграция Rx и TDF упрощает реализацию параллельных приложений с интенсивной нагрузкой на процессор и средства ввода/вывода, а также предоставляет разработчикам возможность явного контроля над буферизацией данных.

Часть III

*Современные
шаблоны
конкурентного
программирования*

Третья, заключительная часть книги позволит вам применить на практике все технологии функционального конкурентного программирования, которые вы уже изучили. Следующие главы станут вашим справочником для ответов на все вопросы о конкурентности.

Глава 13 посвящена рецептам решения как обычных, так и сложных проблем, с которыми вы можете столкнуться в конкурентных приложениях, использующих функциональную парадигму. В главе 14 мы поэтапно рассмотрим полную реализацию масштабируемого и высокопроизводительного серверного приложения для фондового рынка, которое включает в себя версии iOS и WPF для клиентской стороны.

Изученные в этой книге принципы функциональной парадигмы будут применяться при принятии решений по проектированию и архитектуре, а также при разработке кода, чтобы получить высокопроизводительное и масштабируемое решение. В этом разделе вы познакомитесь с положительными побочными эффектами, возникающими в результате использования принципов функционального программирования, позволяющими уменьшить количество ошибок и сделать обслуживание более удобным.

Рецепты и шаблоны для успешного конкурентного программирования

В этой главе:

- двенадцать рецептов кода для решения типичных проблем в параллельном программировании.

Двенадцать рецептов, представленных в этой главе, имеют широкое применение. Вы можете использовать основные идеи как справочный материал, когда столкнетесь с подобной проблемой и будете нуждаться в быстром решении. Этот материал демонстрирует, как функциональные конкурентные абстракции, описанные в данной книге, позволяют решать сложные проблемы путем разработки сложных и мощных функций с относительно небольшим количеством строк кода. Я постарался сделать реализации этих рецептов как можно проще, поэтому вам время от времени придется иметь дело с отменой и обработкой исключений.

В настоящей главе показано, как синтезировать все, что вы уже узнали, чтобы объединить модели конкурентного программирования, применяя в качестве связующего вещества абстракцию функционального программирования с целью писать эффективные и высокопроизводительные программы. К концу этой главы в вашем распоряжении будет набор полезных инструментов (которые можно многократно использовать) для решения типичных проблем конкурентного кодирования.

Каждый рецепт построен на C# или F#; большую часть примеров кода вы найдете в обеих версиях в коде к книге. Имейте также в виду, что F# и C# являются взаимосогласованными языками программирования .NET, которые поддерживают взаимодействие друг с другом. Вы можете легко использовать программу на C# в F# и наоборот.

13.1. Освобождение памяти, занимаемой объектами, для сокращения потребления памяти

В этом разделе мы реализуем многократно используемый пул асинхронных объектов. Его следует применять в тех случаях, когда освобождение занимаемой объектами памяти способствует снижению ее потребления. Сведение к минимуму количества этапов сборки мусора значительно ускоряет работу программы. На рис. 13.1, повторенном из главы 12, показано, как применять конкурентные шаблоны «поставщик – потребитель» из листинга 12.9 для параллельного сжатия и шифрования большого файла.

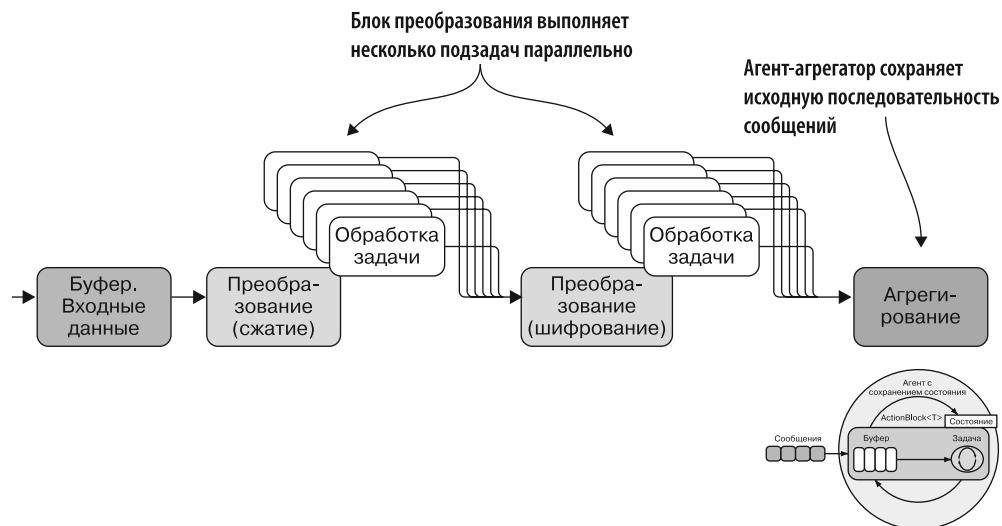


Рис. 13.1. Блоки Transform обрабатывают сообщения параллельно. Когда операция завершается, результат отправляется в следующий блок. Назначение агента-агрегатора — сохранить порядок сообщений аналогично методу расширения PLINQ AsOrdered

Функция `CompressAndEncrypt` из листинга 12.9 разбивает большой файл на набор порций в виде байтовых массивов, что приводит к отрицательному эффекту — многократной сборке мусора вследствие высокого потребления памяти. Каждая порция памяти создается, обрабатывается и утилизируется сборщиком мусора, когда нагрузка на память достигает точки, в которой возникает запрос на выделение дополнительных ресурсов.

Эта ресурсоемкая операция создания и уничтожения байтового массива вызывает многократную сборку мусора, что негативно сказывается на общей производительности приложения. Фактически программа выделяет значительное количество буферов памяти (байтовых массивов), чтобы работать в полностью многопоточном

режиме, то есть в режиме, когда несколько потоков могут одновременно выделять один и тот же объем памяти. Учитывая, что каждый буфер занимает 4096 байт памяти и одновременно работают 25 потоков, получим, что в куче одновременно выделяются около 102 400 байт. Кроме того, когда поток завершает выполнение, многие буфера выходят из области видимости, что приводит к очередной сборке мусора. Это плохо сказывается на производительности, поскольку приложение находится в режиме жесткого управления памятью.

Решение: асинхронное освобождение памяти в пуле объектов. Чтобы оптимизировать производительность конкурентного приложения с интенсивным потреблением памяти, нужно взять на себя функции системного сборщика мусора и самостоятельно освобождать память, занимаемую объектами. В примере параллельного сжатия и шифрования потока нам желательно использовать одни и те же однажды сгенерированные байтовые буфера (байтовые массивы), вместо того чтобы создавать новые. Это возможно с помощью `ObjectPool` — класса, предназначенного для предоставления кэшированного пула объектов. Данный класс освобождает память, занимаемую элементами, которые более не используются. Такое многократное применение объектов позволяет избежать дорогостоящего сбора и освобождения ресурсов, свести к минимуму потенциальное число операций по выделению памяти. В частности, в высококонкурентном примере нам нужен потокобезопасный и неблокирующий (основанный на задачах) пул конкурентных объектов (рис. 13.2).

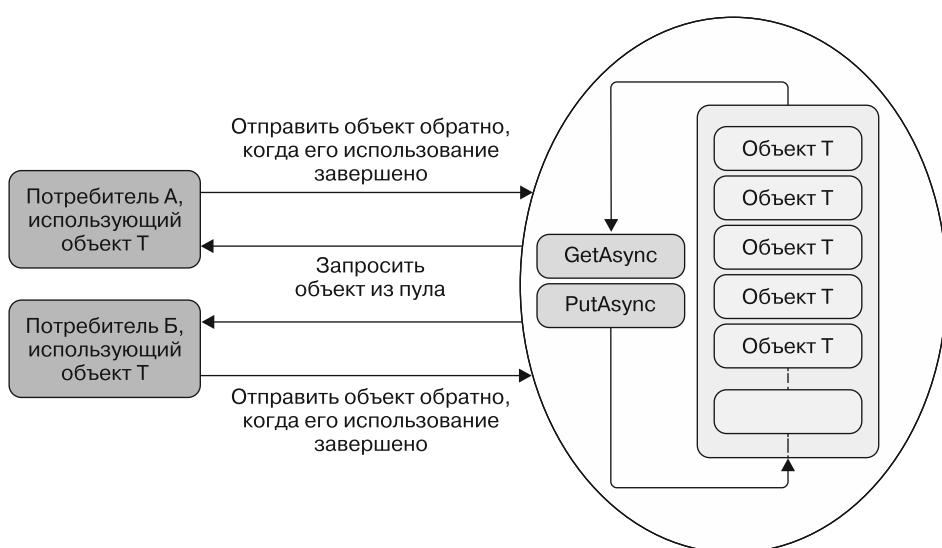


Рис. 13.2. Пул объектов может асинхронно обрабатывать несколько конкурентных запросов для многократно используемых объектов от нескольких потребителей. Затем, после завершения работы, потребитель возвращает объект назад в пул объектов. Внутри пул объектов генерирует очередь объектов, применяя заданный делегат-фабрику. Затем память, занимаемая этими объектами, освобождается, чтобы сократить потребление памяти и затраты на создание новых экземпляров

В листинге 13.1 реализация `ObjectPoolAsync` основана на TDF с применением `BufferBlock` в качестве строительного блока. `ObjectPoolAsync` предварительно инициализирует набор объектов для многократного использования приложением по мере необходимости. Кроме того, библиотека TDF по своей сути потокобезопасна, что обеспечивает асинхронную семантику без блокировок.

Листинг 13.1. Реализация асинхронного пула объектов с использованием TDF

```

public class ObjectPoolAsync<T> : IDisposable
{
    private readonly BufferBlock<T> buffer;
    private readonly Func<T> factory;
    private readonly int msecTimeout;
    public ObjectPoolAsync(int initialCount, Func<T> factory,
    CancellationToken cts, int msecTimeout = 0)
    {
        this.msecTimeout = msecTimeout;
        buffer = new BufferBlock<T>(
            new DataflowBlockOptions { CancellationToken = cts });
        this.factory = () => factory();
        for (int i = 0; i < initialCount; i++)
            buffer.Post(this.factory());
    }
    public Task<bool> PutAsync(T item) => buffer.SendAsync(item);
    public Task<T> GetAsync(int timeout = 0)
    {
        var tcs = new TaskCompletionSource<T>();
        buffer.ReceiveAsync(TimeSpan.FromMilliseconds(msecTimeout))
            .ContinueWith(task =>
    }
    public void Dispose() => buffer.Complete();
}

```

Во время инициализации пула объектов буфер заполняется экземплярами типа T, чтобы объекты были доступны с самого начала

Применение BufferBlock для асинхронной координации внутреннего набора типов T

Использование делегата фабрики для генерации нового экземпляра типа T

Когда потребитель готов, объект типа T возвращается в пул объектов для повторного использования

Когда потребитель делает запрос, пул объектов отправляет объект типа T

`ObjectPoolAsync` принимает в качестве аргументов начальное количество объектов, которые нужно создать, и конструктор делегата фабрики. `ObjectPoolAsync`

предоставляет две функции для управления многократным использованием объекта:

- ❑ `PutAsync` — асинхронное помещение элемента в пул;
- ❑ `GetAsync` — асинхронное извлечение элемента из пула.

В исходном коде к книге вы найдете полную реализацию программы `CompressAndEncrypt`, обновленную для применения `ObjectPoolAsync`. На рис. 13.3 приведено графическое сравнение нескольких этапов сборки мусора в файлах разных размеров для исходной версии программы и новой версии с использованием `ObjectPoolAsync`.

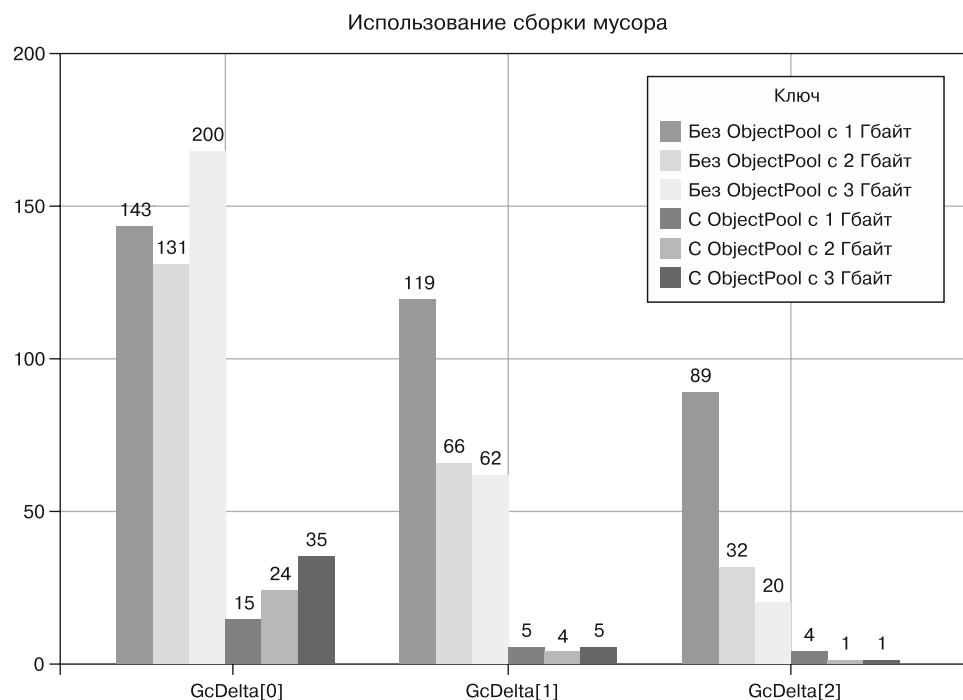


Рис. 13.3. Сравнение программы `CompressAndEncrypt` из главы 12, которая обрабатывает большие файлы разного размера (1, 2 и 3 Гбайт), реализованной с использованием и без использования `AsyncObjectPool`. Реализация, задействующая пул объектов, имеет небольшое количество этапов сборки мусора по сравнению с исходным вариантом. Сведение к минимуму этих этапов приводит к повышению производительности

Результаты, показанные на диаграмме, демонстрируют, что программа `CompressAndEncrypt`, реализованная с использованием `ObjectPoolAsync`, значительно сокращает количество этапов сборки мусора, повышая общую производительность приложения. На восьмиядерном компьютере новая версия `CompressAndEncrypt` работает примерно на 8 % быстрее.

13.2. Нестандартный параллельный оператор Fork/Join

В этом разделе мы реализуем многократно используемый метод расширения для распараллеливания операций Fork/Join. Предположим, что вы обнаружили в программе фрагмент кода, который лучше было бы выполнить параллельно с применением шаблона «разделяй и властвуй», повысив таким образом производительность. Вы решили реорганизовать код, чтобы использовать конкурентный шаблон Fork/Join (рис. 13.4). И чем больше вы будете проверять программу, тем больше будет появляться подобных шаблонов.

ПРИМЕЧАНИЕ

Как мы помним из раздела 4.2, шаблон Fork/Join, как и «разделяй и властвуй», разбивает работу на небольшие задачи до тех пор, пока каждая маленькая задача не станет достаточно простой, чтобы ее можно было выполнить без дальнейших разделений, а затем координирует параллельные обработчики.

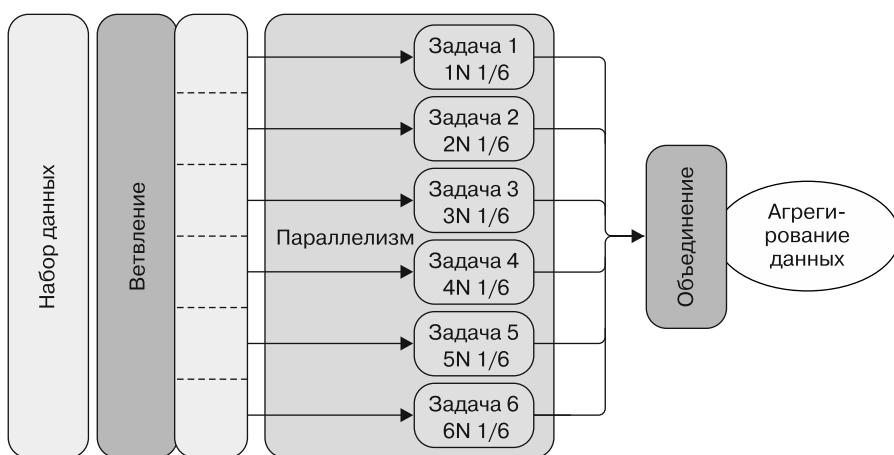


Рис. 13.4. Шаблон Fork/Join разбивает задачу на подзадачи, которые могут выполняться независимо и параллельно. После завершения операций подзадачи снова объединяются. Не случайно этот шаблон часто используется для распараллеливания данных. В сущности, здесь есть явное сходство

К сожалению, в .NET нет встроенной поддержки параллельных методов расширения Fork/Join, которые можно было бы задействовать по мере необходимости. Но эти методы, как и многое другое, можно создать, чтобы иметь многоразовый и гибкий оператор, выполняющий следующие действия:

- ❑ разбиение данных;
- ❑ параллельное применение шаблона Fork/Join;
- ❑ при необходимости — настройка степени параллелизма;
- ❑ объединение результатов с использованием функции сжатия.

Операторы .NET `Task.WhenAll` и F# `Async.Parallel` позволяют составить параллельный набор из заданных задач; но эти операторы не предоставляют функциональность агрегирования (или сжатия) для объединения результатов. Более того, им не хватает возможностей настройки, чтобы контролировать степень параллелизма. Для того чтобы получить желаемый оператор, необходимо индивидуальное решение.

Решение: составление конвейера из шагов, формирующих шаблон Fork/Join. Благодаря TDF можно компоновать различные строительные блоки, образуя конвейер. Его можно использовать для определения шагов шаблона Fork/Join (рис. 13.5), где на шаге Fork параллельно выполняется набор задач, затем, на следующем шаге, результаты объединяются, а на последнем применяется блок сжатия для получения конечного результата. На последующем шаге рабочего процесса, на котором агрегируются результаты, нужен объект, обрабатывающий состояния предыдущих шагов. В данном случае мы воспользуемся агентным блоком, построенным в главе 12 с применением TDF.

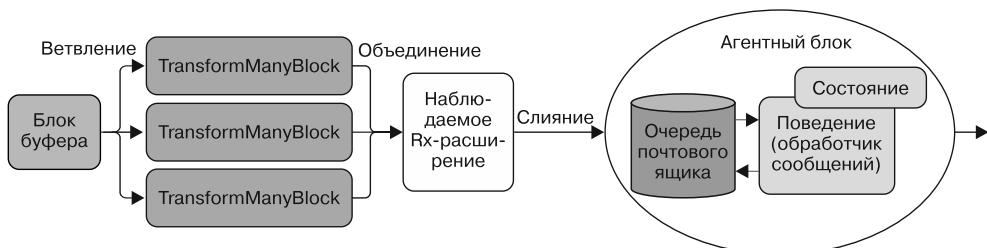


Рис. 13.5. Шаблон Fork/Join, реализованный с использованием TDF, где каждый шаг вычислений определяется отдельным блоком потока данных

Шаблон Fork/Join реализован как метод расширения поверх параметризованного `IEnumerable` — к нему удобно обращаться в свободном стиле из кода, как показано в листинге 13.2 (код, на который следует обратить внимание, выделен жирным шрифтом).

Метод расширения `ForkJoin` принимает в качестве аргумента источник `IEnumerable`, чтобы обработать функцию отображения и преобразовать ее элементы, а также функцию агрегирования (сжатия) для объединения всех результатов, полученных в процессе вычислений отображения. Аргумент `initialState` — это начальное число, которое требуется для функции-агрегатора, чтобы задать значение начального состояния. Но если результаты типа `T2` можно объединить (поскольку соблюдаются моноидальные законы), то можно изменить метод, чтобы использовать функцию сжатия с нулевым начальным состоянием, как показано в листинге 5.10.

Внутренние блоки потока данных связаны, образуя конвейер. Интересно, что `mapperBlock` преобразуется в `Observable` с использованием метода расширения `AsObservable`, который затем подписывается на отправку сообщений агенту `reducerAgent` при материализации вывода. Значения `partitionLevel` и `boundCapacity` применяются соответственно для определения степени параллелизма и ограничения емкости.

Листинг 13.2. Параллельный ForkJoin с использованием TDF

```

Функции map и aggregate возвращают тип Task
для обеспечения конкурентного поведения
    public static async Task<R> ForkJoin<T1, T2, R>(
        this IEnumerable<T1> source,
        Func<T1, Task<IEnumerable<T2>>> map,
        Func<R, T2, Task<R>> aggregate,
        R initialState, CancellationTokenSource cts = null,
        int partitionLevel = 8, int boundCapacity = 20) ←
    {
        cts = cts ?? new CancellationTokenSource();
        var blockOptions = new ExecutionDataflowBlockOptions {
            MaxDegreeOfParallelism = partitionLevel,
            BoundedCapacity = boundCapacity,
            CancellationToken = cts.Token
        }; ←
        Экземпляры строительных блоков,
        образующие конвейер Fork/Join

        var inputBuffer = new BufferBlock<T1>( ←
            new DataflowBlockOptions {
                CancellationToken = cts.Token,
                BoundedCapacity = boundCapacity
            });
        Упаковка свойств деталей
        выполнения для настройки
        BufferBlock inputBuffer

        var mapperBlock = new TransformManyBlock<T1, T2> ←
        (map, blockOptions);
        var reducerAgent = Agent.Start(initialState, aggregate, cts); ←
        var linkOptions = new DataflowLinkOptions{PropagateCompletion=true};
        inputBuffer.LinkTo(mapperBlock, linkOptions); ←

Соединение строительных блоков для формирования
и связывания шагов, образующих шаблон Fork/Join

IDisposable disposable = mapperBlock.AsObservable()
    .Subscribe(async item => await reducerAgent.Send(item)); ←

foreach (var item in source) ←
    await inputBuffer.SendAsync(item);
    inputBuffer.Complete(); ←
    TransformManyBlock преобразуется в объект
    Observable, который используется
    для передачи выходных данных
    агенту reducerAgent в виде сообщений

    var tcs = new TaskCompletionSource<R>();

    await inputBuffer.Completion.ContinueWith(task =>
        mapperBlock.Complete());
    await mapperBlock.Completion.ContinueWith(task => { ←
        var agent = reducerAgent as StatefulDataflowAgent<R, T2>;
        disposable.Dispose();
        tcs.SetResult(agent.State);
    });
    Когда mapperBlock завершен, задача
    продолжения Task назначает reducerAgent
    результатом для задачи tcs, который передается
    вызывающему объекту в качестве выходных данных
    return await tcs.Task;
}

Запуск процесса Fork/Join путем передачи элементов
коллекции входных данных на первый шаг конвейера

```

Вот простой пример использования оператора ForkJoin:

```
Task<long> sum = Enumerable.Range(1, 100000)
    .ForkJoin<int, long, long>(
        async x => new[] { (long)x * x },
        async (state, x) => state + x, 0L);
```

В этом коде суммируются квадраты всех чисел от 1 до 100 000 с использованием шаблона Fork/Join.

13.3. Распараллеливание задач с зависимостями: разработка кода для оптимизации производительности

Предположим, что нам нужно написать инструмент, который бы выполнял серию асинхронных задач — каждая со своим набором зависимостей, влияющих на порядок операций. Эту проблему можно решить методом последовательного и императивного выполнения; но если мы хотим получить максимальную производительность, то последовательные операции не подходят. Вместо этого нам нужно построить задачи для параллельного выполнения. Многие конкурентные проблемы можно считать статическим набором атомарных операций с зависимостями между входом и выходом. После завершения операции ее выходные данные используются как входные для других зависимых операций. Чтобы оптимизировать производительность, выполнение этих задач должно планироваться на основе зависимостей, а алгоритм должен быть оптимизирован для выполнения зависимых задач последовательно по мере необходимости и параллельно, насколько это возможно.

Нам нужен многократно применяемый компонент, который выполнял бы набор задач параллельно, при этом гарантируя соблюдение всех зависимостей, способных повлиять на порядок операций. Как создать программную модель с внутренним параллелизмом набора операций, выполняющихся эффективно, параллельно или последовательно, с учетом зависимостей от других операций?

Решение: реализация графа зависимостей задач. Решение проблемы называется направленным ациклическим графом (Directed Acyclic Graph, DAG), целью которого является формирование графа путем разбиения операций на набор атомарных задач с определенными зависимостями. Ациклическая природа графа важна, поскольку исключает возможность взаимных блокировок между задачами при условии, что задачи действительно атомарны. При описании графа важно понимать все зависимости между задачами, особенно скрытые, которые могут привести к взаимоблокировкам или состоянию гонки. На рис. 13.6 показан типичный пример структуры данных в виде графа, который можно использовать для представления ограничений планирования между операциями графа. Графы в информатике являются чрезвычайно мощными структурами данных, служащими основой для построения сильных алгоритмов.

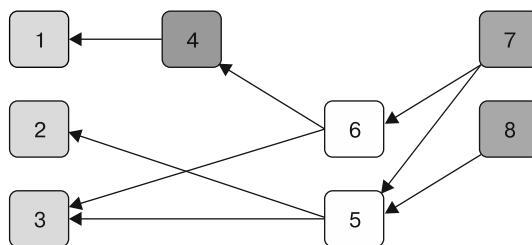


Рис. 13.6. Граф — это набор вершин, соединенных ребрами. В данном направленном ациклическом графе узел 1 зависит от узлов 4 и 5, узел 2 — от узла 5, узел 3 — от узлов 5 и 6 и т. д.

Структуру DAG можно использовать как стратегию для параллельного выполнения задач с учетом порядка зависимостей для повышения производительности. Такую структуру графа можно определить, задействуя F#-объект `MailboxProcessor`, который сохраняет внутреннее состояние задач, зарегистрированных для выполнения, в форме граничных зависимостей.

Валидация направленного ациклического графа

При работе с любой графовой структурой данных, такой как DAG, необходимо решить проблему правильной регистрации ребер. Например, на рис. 13.6: что будет, если зарегистрирован узел 2 с зависимостями от узлов 7 и 8, а узел 8 не существует? Также может оказаться, что некоторые ребра зависят друг от друга, и это приведет к направленному циклу. В случае направленного цикла очень важно выполнять задачи параллельно, иначе некоторые могут ожидать вечно, пока не завершатся другие.

Решение называется *топологической сортировкой* — это означает, что можно упорядочить все вершины графа таким образом, чтобы все направленные ребра шли от вершины с меньшим номером в последовательности к вершине с большим номером. Например, если задача А должна быть завершена раньше, чем задача В, а задача В — раньше, чем задача С, которая, в свою очередь, должна быть выполнена до задачи А, то перед нами цикл, и система сообщит об ошибке, вызвав исключение. Если отношения предшествования образуют прямой цикл, то решения не существует. Такой вид проверки называется *обнаружением прямых циклов*. Если направленный граф удовлетворяет этим правилам, то он считается DAG и подходит для параллельного выполнения нескольких задач с зависимостями.

Полную версию листинга 13.4, включая валидацию DAG, вы найдете в исходном коде к этой книге.

В следующем примере F#-объект `MailboxProcessor` используется в качестве идеального кандидата для реализации DAG с целью выполнения параллельных операций с зависимостями (листинг 13.3). Сначала мы определим размеченное объединение, используемое для управления задачами и выполнения их зависимостей.

Листинг 13.3. Тип сообщений и структура данных для координирования выполнения задач

```

type TaskMessage =
    | AddTask of int * TaskInfo
    | QueueTask of TaskInfo
    | ExecuteTasks
and TaskInfo =
    { Context : System.Threading.ExecutionContext
        Edges : int array; Id : int; Task : Func<Task>
        EdgesLeft : int option; Start : DateTimeOffset option
        End : DateTimeOffset option }

```

Команды отправляются в ParallelTasksDAG внутреннему агенту dagAgent, который отвечает за координацию выполнения задач

Обертка подробностей выполнения каждой задачи

Тип `TaskMessage` описывает варианты сообщений, отправляемых внутреннему агенту `ParallelTasksDAG`, реализованному в листинге 13.4. Эти сообщения используются для координирования задач и синхронизации зависимостей. Тип `TaskInfo` содержит и отслеживает подробности зарегистрированных задач во время выполнения DAG, включая ребра зависимостей. Контекст выполнения (<http://mng.bz/2F9o>) захватывается для доступа во время отложенного выполнения к такой информации, как текущий пользователь, состояние, связанное с логическим потоком выполнения, информация о защите доступа к коду и т. д. При возникновении события публикуется начальное и конечное время выполнения.

Листинг 13.4. F#-агент DAG для распараллеливания выполнения операций

Событие, которое показывает экземпляр `onTaskCompletedEvent`, используемый для уведомления о завершении задачи

Внутреннее состояние агента для отслеживания задач и их зависимостей. Коллекции являются изменяемыми, поскольку состояние изменяется в ходе выполнения `ParallelTasksDAG` и потому что они унаследовали потокобезопасность благодаря размещению внутри агента

```

type ParallelTasksDAG() =
    let onTaskCompleted = new Event<TaskInfo>()

    let dagAgent = new MailboxProcessor<TaskMessage>(fun inbox ->
        let rec loop (tasks : Dictionary<int, TaskInfo>
                    edges : Dictionary<int, int list>) = async {
            let! msg = inbox.Receive()
            match msg with
            | ExecuteTasks ->
                let fromTo = new Dictionary<int, int list>()
                for KeyValue(key, value) in tasks do
                    let operation =
                        { value with EdgesLeft = Some(value.Edges.Length) }
                    for from in operation.Edges do
                        let exists, lstDependencies = fromTo.TryGetValue(from)
                        if not <| exists then
                            fromTo.Add(from, [ operation.Id ])
                        else fromTo.[from] <- (operation.Id :: lstDependencies)
                    ops.Add(key, operation)
            ops |> Seq.iter (fun kv ->
                let task = kv.Value
                let dependencies = fromTo.[task.Id]
                if dependencies.Length > 0 then
                    let edges = task.Edges
                    for dependency in dependencies do
                        let edge = edges.[dependency]
                        edge.EdgesLeft <- edge.EdgesLeft - 1
                        if edge.EdgesLeft = 0 then
                            onTaskCompleted.Trigger(edge)
            )
        }
    )

```

Сообщение, которое запускает выполнение `ParallelTasksDAG`

Асинхронное ожидание сообщения

Коллекция, которая сопоставляет монотонно возрастающий индекс с выполняемой задачей

Процесс перебирает список задач, анализируя зависимости между другими задачами, чтобы создать топологическую структуру, описывающую порядок выполнения задач

```

match kv.Value.EdgesLeft with
| Some(n) when n = 0 -> inbox.Post(QueueTask(kv.Value))
| _ -> ()
return! loop ops fromTo
| QueueTask(op) ->
    Async.Start <| async {
        let start = DateTimeOffset.Now
        match op.Context with
        | null -> op.Task.Invoke() |> Async.AwaitATsk
        | ctx -> ExecutionContext.Run(ctx.CreateCopy(), (fun op -> let opCtx = (op :?> TaskInfo)
                                                    opCtx.Task.Invoke().ConfigureAwait(false)), taskInfo)
    }
    Сообщение для постановки задачи в очередь, запуска и после завершения удаления задачи как активной зависимости из состояния агента
    let end' = DateTimeOffset.Now
    onTaskCompleted.Trigger { op with Start = Some(start)
                                End = Some(end') }
    let exists, deps = edges.TryGetValue(op.Id)
    if exists && deps.Length > 0 then
        let depOps = getDependentOperation deps tasks []
        edges.Remove(op.Id) |> ignore
        depOps |> Seq.iter (fun nestedOp ->
            inbox.Post(QueueTask(nestedOp)))
    return! loop tasks edges
| AddTask(id, op) -> tasks.Add(id, op)
    return! loop tasks edges
loop (new Dictionary<int, TaskInfo>(HashIdentity.Structural))
    (new Dictionary<int, int list>(HashIdentity.Structural)))

```

Если захваченный ExecutionContext равен нулю, то функция задачи выполняется в текущем контексте

Выполнение задачи с использованием захваченного ExecutionContext

Сообщение, добавляющее задачу, которую нужно выполнить, в соответствии с ее зависимостями, если таковые есть

[<CLIEventAttribute>]

```

→ member this.OnTaskCompleted = onTaskCompleted.Publish
member this.ExecuteTasks() = dagAgent.Post ExecuteTasks
member this.AddTask(id, task, [<ParamArray>] edges : int array) =
    let data = { Context = ExecutionContext.Capture()
                Edges = edges; Id = id; Task = task
                NumRemainingEdges = None; Start = None; End = None }
    dagAgent.Post(AddTask(id, data))

```

Начало выполнения зарегистрированных задач

Добавление задачи с ее зависимостями и текущего ExecutionContext для выполнения DAG

Создание и публикация события onTaskCompleted для уведомления о завершении задачи. Событие содержит информацию о задаче

Назначение функции `AddTask` — зарегистрировать задачу, включая произвольные ребра зависимостей. Эта функция принимает уникальный идентификатор, функцию задачи, которая должна быть выполнена, и множество ребер, соответствующих идентификаторам других зарегистрированных задач, все из которых должны быть выполнены, прежде чем можно будет выполнить текущую задачу. Если массив пуст, это означает, что зависимостей нет. `MailboxProcessor` с именем `dagAgent` сохраняет зарегистрированные задачи в текущем состоянии `tasks`, которое представляет собой

отображение (`tasks: Dictionary<int, TaskInfo>`) между идентификатором каждой задачи и ее параметрами. Агент также сохраняет состояние ребер зависимостей для каждого идентификатора задачи (`edges : Dictionary<int, int list>`). Коллекции `Dictionary` являются изменяемыми, поскольку состояние изменяется при выполнении `ParallelTasksDAG` и потому что они унаследовали потокобезопасность, будучи реализованными внутри агента. Когда агент получает уведомление о начале выполнения, часть процесса требует верификации того, что все ребра зависимостей зарегистрированы и что в графе нет циклов. Реализацию этого этапа верификации вы найдете в полной реализации `ParallelTasksDAG` в исходном коде к книге. Следующий код является примером на C#, который ссылается на F#-библиотеку и использует ее для запуска `ParallelTasksDAG`. Зарегистрированные задачи соответствуют зависимостям, показанным на рис. 13.6:

```
Func<int, int, Func<Task>> action = (id, delay) => async () => {
    Console.WriteLine($"Starting operation{id} in Thread Id
{Thread.CurrentThread.ManagedThreadId} . . . ");
    await Task.Delay(delay);
};

var dagAsync = new DAG.ParallelTasksDAG();
dagAsync.OnTaskCompleted.Subscribe(op =>
    Console.WriteLine($"Operation {op.Id} completed in Thread Id
{Thread.CurrentThread.ManagedThreadId}"));

dagAsync.AddTask(1, action(1, 600), 4, 5);
dagAsync.AddTask(2, action(2, 200), 5);
dagAsync.AddTask(3, action(3, 800), 6, 5);
dagAsync.AddTask(4, action(4, 500), 6);
dagAsync.AddTask(5, action(5, 450), 7, 8);
dagAsync.AddTask(6, action(6, 100), 7);
dagAsync.AddTask(7, action(7, 900));
dagAsync.AddTask(8, action(8, 700));
dagAsync.ExecuteTasks();
```

Назначение вспомогательной функции `action` — при запуске задачи напечатать текущий идентификатор потока для доказательства многопоточной функциональности. Событие `OnTaskCompleted` зарегистрировано для того, чтобы отправлять уведомления о завершении печати в консоли идентификатора задачи и текущего идентификатора потока. При вызове метода `ExecuteTasks` будет выведено следующее.

```
Starting operation 8 in Thread Id 23...
Starting operation 7 in Thread Id 24...
Operation 8 Completed in Thread Id 23
Operation 7 Completed in Thread Id 24
Starting operation 5 in Thread Id 23...
Starting operation 6 in Thread Id 25...
Operation 6 Completed in Thread Id 25
Starting operation 4 in Thread Id 24...
Operation 5 Completed in Thread Id 23
Starting operation 2 in Thread Id 27...
```

```
Starting operation 3 in Thread Id 30...
Operation 4 Completed in Thread Id 24
Starting operation 1 in Thread Id 28...
Operation 2 Completed in Thread Id 27
Operation 1 Completed in Thread Id 28
Operation 3 Completed in Thread Id 30
```

Как видим, задачи выполняются параллельно в другом потоке выполнения (с другим идентификатором потока), а порядок зависимостей сохраняется.

13.4. Шлюз для координации конкурентных операций ввода-вывода с разделяемыми ресурсами: одна операция записи, несколько операций чтения

Предположим, что нам нужно реализовать серверное приложение, в которое поступает много конкурентных клиентских запросов. Эти конкурентные запросы приходят в серверное приложение из-за потребности в доступе к разделяемым данным. Иногда приходит запрос, который должен изменить общие данные, что требует их синхронизации.

Когда поступает новый клиентский запрос, пул потоков выделяет поток для обслуживания запроса и запуска обработки. Предположим, что в этот момент запрос хочет обновить данные на сервере потокобезопасным способом. Необходимо решить проблему координации операций чтения и записи таким образом, чтобы обеспечить конкурентный доступ к ресурсам без блокировки. В таком случае блокировка означает координирование доступа к разделяемому ресурсу. При этом операция записи блокирует другие операции, чтобы получить право собственности на ресурс, пока операция не будет завершена.

Возможное решение — использовать примитивную блокировку, такую как `ReaderWriterLockSlim` (<http://mng.bz/FY0J>), которая также управляет доступом к ресурсу, допуская наличие нескольких потоков.

Но, как вы уже узнали из этой книги, следует по возможности избегать использования примитивных блокировок. Они препятствуют параллельному выполнению кода и во многих случаях приводят к перегрузке пула потоков, вынуждая создавать новый поток для каждого запроса. Доступ к ресурсам для других потоков блокируется. Другим недостатком является то, что блокировки могут удерживаться в течение очень долгого времени. И это приводит к тому, что потоки, которые были выделены из пула потоков, обрабатывают запросы на чтение и сразу переводятся в спящий режим, ожидая, пока поток записи завершит свою задачу. Кроме того, такая архитектура не масштабируется.

Наконец, операции чтения и записи должны обрабатываться по-разному, чтобы несколько операций чтения могли выполняться одновременно, поскольку такие

операции не изменяют данных. Это нужно сбалансировать, гарантируя, что операции записи всегда обрабатываются по очереди и операции чтения в тот момент блокируются во избежание извлечения устаревших данных.

Необходимо построить координатор, который бы синхронизировал асинхронные операции чтения и записи без блокировки. Этот координатор должен выполнять операции записи по очереди, последовательно, не блокируя потоки, а операции чтения при этом должны по-прежнему выполняться параллельно.

Решение: применение нескольких операций чтения/записи к разделяемым потокобезопасным ресурсам. `ReaderWriterAgent` предлагает асинхронную семантику для операций чтения/записи без блокирования потоков и поддерживает порядок операций FIFO, за счет чего снижается потребление ресурсов и повышается производительность приложения. Фактически `ReaderWriterAgent` позволяет выполнять огромный объем работы, используя всего несколько потоков. Независимо от количества выполняемых операций, `ReaderWriterAgent` требуется очень незначительное количество ресурсов.

В следующих примерах мы выполним несколько операций чтения и записи для разделяемой базы данных. При выполнении этих операций потокам чтения будет предоставляться более высокий приоритет, чем потокам записи, как показано на рис. 13.7. Аналогичные концепции могут быть применены к любым другим ресурсам, таким как файловая система.

ПРИМЕЧАНИЕ

Вообще говоря, `ReaderWriterAgent` — отличный выбор для программ с конкурентным асинхронным доступом к ресурсам с помощью операций ввода-вывода.

В листинге 13.5 приведена реализация `ReaderWriterAgent` с использованием F#-объекта `MailboxProcessor`. Причина выбора F# `MailboxProcessor` заключается в простоте определения конечных автоматов, которые удобны для реализации асинхронного координатора чтения/записи. Сначала нужно определить типы сообщений для представления операций, с помощью которых `ReaderWriterAgent` координирует и синхронизирует операции чтения и записи.

Листинг 13.5. Типы сообщений, используемые координатором `ReaderWriterAgent`

```
type ReaderWriterMsg <'r, 'w> = 
    | Command of ReadWriteMessages<'r, 'w>
    | CommandCompleted
and ReaderWriterGateState = 
    | SendWrite
    | SendRead of count:int
    | Idle
and ReadWriteMessages<'r, 'w> = 
    | Read of r:'r
    | Write of w:'w
```

Применение размеченного объединения
 с вариантами команд для отправки операций
 чтения/записи для постановки в очередь

Использование типов сообщений для изменения
 состояния и координирования операций
 во внутренней очереди `ReaderWriterAgent`

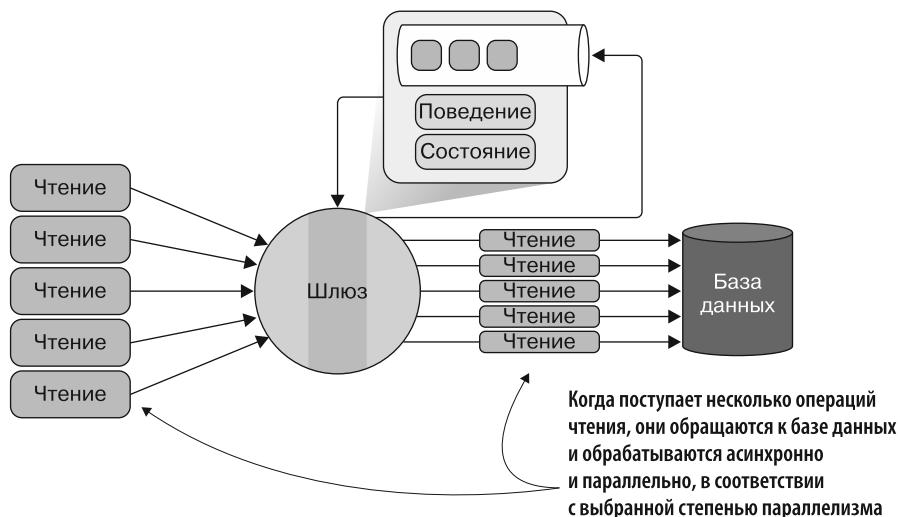
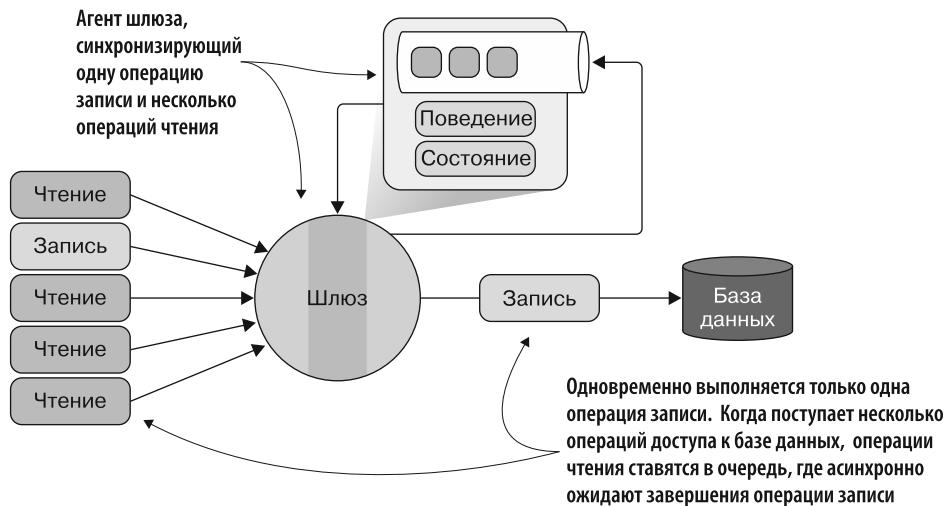


Рис. 13.7. ReaderWriterAgent работает как агент шлюза для асинхронной синхронизации доступа к разделяемым ресурсам. На верхнем изображении за раз выполняется только одна операция записи, в то время как операции чтения ставятся в очередь для асинхронного ожидания завершения операции записи. На нижнем изображении несколько операций чтения обрабатываются асинхронно и параллельно, в соответствии с выбранной степенью параллелизма

Тип сообщения `ReaderWriterMsg` соответствует команде чтения или записи в базу данных либо содержит уведомление о завершении операции. `ReaderWriterGateState` — это размеченное объединение, используемое для постановки операций чтения/записи в очередь `ReaderWriterAgent`. В итоге размеченное объединение

`ReadWriteMessages` идентифицирует варианты для операций чтения/записи, поставленных в очередь во внутреннем `ReaderWriterAgent`.

В листинге 13.6 представлена реализация типа `ReaderWriterAgent`.

Листинг 13.6. `ReaderWriterAgent` координирует асинхронные операции

```

type ReaderWriterAgent<'r,'w>(workers:int,
  behavior: MailboxProcessor<ReadWriteMessages<'r,'w>> ->
  Async<unit>,?errorHandler, ?cts:CancellationTokenSource) =
  let cts = defaultArg cts (new CancellationTokenSource())
  let errorHandler = defaultArg errorHandler ignore
  let supervisor = MailboxProcessor<Exception>.Start(fun inbox -> async
  {
    while true do
      let! error = inbox.Receive(); errorHandler error })
  
```

Если необязательные аргументы не переданы
в конструктор, то они инициализируются
значениями по умолчанию

Агент-супервизор обрабатывает исключения. Для асинхронного
ожидания входящих сообщений используется цикл `while-true`

```

let agent = MailboxProcessor<ReaderWriterMsg<'r,'w>>.Start(fun inbox ->
  let agents = Array.init workers (fun _ ->
    (new AgentDisposable<ReadWriteMsg<'r,'w>>(behavior, cts))
    .withSupervisor supervisor)
  
```

Конструктор принимает
число обработчиков для задания степени параллелизма,
поведение агента, который обращается к базе данных
для выполнения операций чтения/записи, и необязательные
аргументы для обработки ошибок и отмены внутреннего
агента в случае остановки все еще активных операций

Каждый вновь созданный агент
регистрирует обработчик ошибок
для уведомления агента-супервизора

Создание коллекции агентов с заданным поведением,
переданным для распараллеливания операций
чтения/записи базы данных. Доступ синхронизирован

```

  cts.Token.Register(fun () ->
    agents |> Array.iter(fun agent -> (agent:>IDisposable).Dispose()))
  
```

Регистрация стратегии отмены, чтобы
остановить обработчики внутренних агентов

```

  let writeQueue = Queue<_>()
  let readQueue = Queue<_>()
  let rec loop i state = async {
    let! msg = inbox.Receive()
    let next = (i+1) % workers
    match msg with
    | Command(Read(req)) ->
        match state with
        | Idle -> agents.[i].Agent.Post(Read(req))
          return! loop next (SendRead 1)
        | SendRead(n) when writeQueue.Count = 0 ->
          agents.[i].Agent.Post(Read(req))
          return! loop next (SendRead(n+1))
        | _ -> readQueue.Enqueue(req)
          return! loop i state
    | Command(Write(req)) ->
        match state with
        | Idle -> agents.[i].Agent.Post(Write(req))
          return! loop next (SendWrite 1)
        | SendWrite(n) when readQueue.Count = 0 ->
          agents.[i].Agent.Post(Write(req))
          return! loop next (SendWrite(n+1))
        | _ -> writeQueue.Enqueue(req)
          return! loop i state
  }
  
```

Использование внутренних очередей для управления
доступом и выполнением операций чтения/записи

Вариант Command Read, основанный на текущем
состоянии агента, может поставить в очередь
новую операцию чтения, запустить операцию чтения,
если очередь записи пуста, или же ничего не делать

Вариант Command Write, основанный на текущем состоянии агента, может
либо ничего не делать, либо поставить в очередь операцию записи

```

| Idle -> agents.[i].Agent.Post(Write(req))
    return! loop next SendWrite
| SendRead(_) | SendWrite -> writeQueue.Enqueue(req)
    return! loop i state
| CommandCompleted -> ← CommandCompleted уведомляет о завершении операции,
    match state with                         чтобы обновить текущее состояние очередей чтения/записи
    | Idle -> failwith "Operation no possible"
    | SendRead(n) when n > 1 -> return! loop i (SendRead(n-1))
    | SendWrite | SendRead(_) ->
        if writeQueue.Count > 0 then
            let req = writeQueue.Dequeue()
            agents.[i].Agent.Post(Write(req))
            return! loop next SendWrite
        elif readQueue.Count > 0 then
            readQueue |> Seq.iteri (fun j req ->
                agents.[(i+j)%workers].Agent.Post(Read(req)))
            let count = readQueue.Count
            readQueue.Clear()
            return! loop ((i+ count)%workers) (SendRead count)
        else return! loop i Idle }
    loop 0 Idle), cts.Token)                  Эта функция определяет асинхронную
                                                двунаправленную коммуникацию между
                                                агентом и вызывающим объектом для отправки
                                                команды и ожидания ответа без блокировки

let postAndAsyncReply cmd createRequest =
    agent.PostAndAsyncReply(fun ch -> ←
        createRequest(AsyncReplyChannelWithAck(ch, fun () ->
            agent.Post(CommandCompleted))) |> cmd |> ReaderWriterMsg.Command
    member this.Read(readRequest) = postAndAsyncReply Read readRequest
    member thisWrite(writeRequest) = postAndAsyncReply Write writeRequest

```

Реализация внутреннего F# MailboxProcessor в типе ReaderWriterAgent представляет собой конечный автомат с несколькими состояниями, который монопольно координирует доступ к разделяемым ресурсам для чтения и записи. ReaderWriterAgent создает субагенты, которые осуществляют доступ к ресурсам на основе полученного сообщения типа ReaderWriterMsg. Когда координатор агента получает команду Read, его текущее состояние проверяется посредством сопоставления с образцом для применения следующей логики монопольного доступа.

- ❑ В случае состояния Idle команда Read отправляется дочерним агентам для обработки. Если активных операций записи нет, то состояние основного агента меняется на SendRead.
- ❑ В случае состояния SendRead и при отсутствии активных операций записи операция Read отправляется дочернему агенту для выполнения.
- ❑ Во всех остальных случаях операция Read помещается в локальную очередь Read для последующей обработки.

В случае если координатору агента отправлена команда Write, сообщение сопоставляется с образцом и обрабатывается в соответствии с его текущим состоянием.

- ❑ В случае состояния Idle команда Write отправляется в почтовый ящик субагента для обработки, затем состояние основного агента меняется на SendWrite.

- ❑ Во всех остальных случаях операция `Write` помещается в локальную очередь `Write` для последующей обработки.

На рис. 13.8 показан конечный автомат с несколькими состояниями `ReaderWriterAgent`.

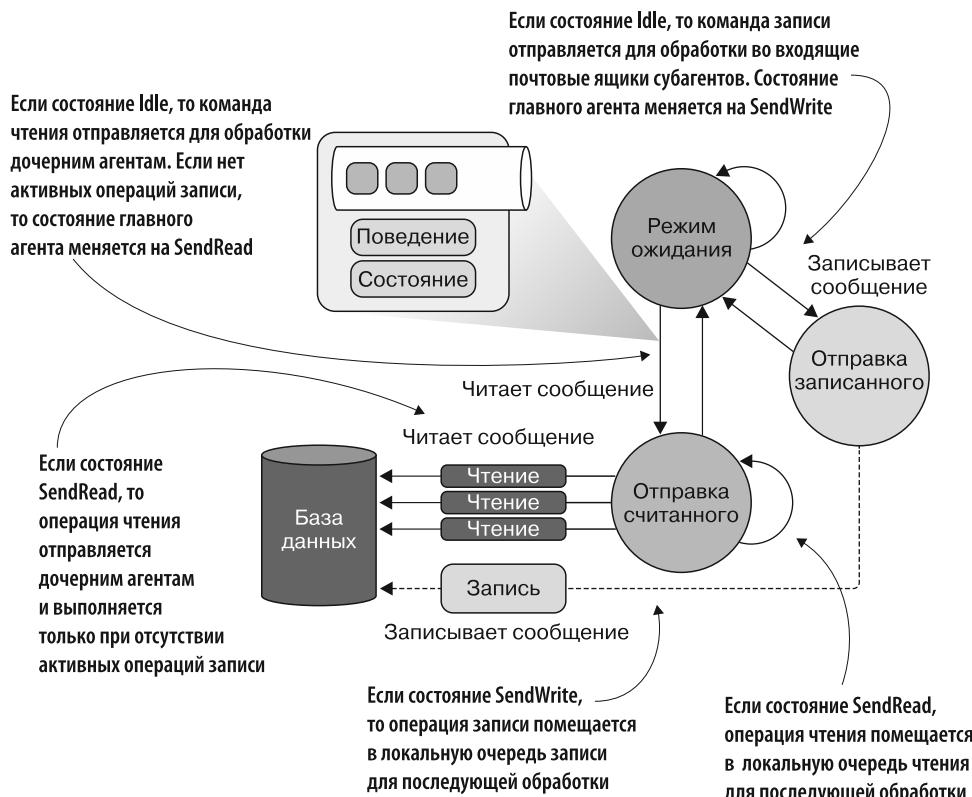


Рис. 13.8. `ReaderWriterAgent` работает как конечный автомат, где каждое состояние стремится синхронизировать доступ к разделяемым ресурсам (в нашем случае к базе данных)

В следующем фрагменте кода представлен простой пример использования `ReaderWriterAgent`. Для простоты вместо конкурентного доступа к базе данных здесь организовано потокобезопасное и неблокирующее обращение к локальному изменяемому словарю:

```
type Person = { id:int; firstName:string; lastName:string; age:int }

let myDB = Dictionary<int, Person>()

let agentSql connectionString =
    fun (inbox: MailboxProcessor<_>) ->
        let rec loop() = async {
            let! msg = inbox.Receive()
```

```

match msg with
| Read(Get(id, reply)) ->
    match myDB.TryGetValue(id) with
    | true, res -> reply.Reply(Some res)
    | _ -> reply.Reply(None)
| Write(Add(person, reply)) ->
    let id = myDB.Count
    myDB.Add(id, {person with id = id})
    reply.Reply(Some id)
return! Loop() }
loop()

```

```

let agent = ReaderWriterAgent(maxOpenConnection, agentSql connectionString)

let write person = async {
    let! id = agent.Write(fun ch -> Add(person, ch))
    do! Async.Sleep(100)
}

let read personId = async {
    let! resp = agent.Read(fun ch -> Get(personId, ch))
    do! Async.Sleep(100)
}

[ for person in people do
    yield write person
    yield read person.Id
    yield write person
    yield read person.Id
    yield read person.Id ]
|> Async.Parallel

```

В примере кода создается объект `agentSql`, чье назначение — эмуляция доступа к базе данных, в роли которой выступает локальный ресурс `myDB`. Экземпляр агента типа `ReaderWriterAgent` координирует параллельные операции чтения и записи, что обеспечивает конкурентный и потокобезопасный доступ к словарю `myDB` без блокировки. В реальном сценарии вместо изменяемой коллекции `myDB` используется база данных, файл или любой другой разделяемый ресурс.

13.5. Потокобезопасный генератор случайных чисел

Часто при работе с многопоточным кодом нужно генерировать случайные числа для той или иной операции в программе. Например, предположим, что вы пишете приложение веб-сервера, который в ответ на запрос пользователя должен отправлять случайный аудиоклип. По соображениям производительности набор аудиоклипов загружен в память сервера, конкурентно получающего большое количество запросов. При каждом запросе аудиоклип должен выбираться случайным образом и отправляться пользователю для воспроизведения.

В большинстве случаев класс `System.Random` является достаточно быстрым решением для получения случайных числовых значений. Но эффективное применение экземпляра `Random`, к которому осуществляется параллельный доступ в высокопроизводительных решениях, становится сложной проблемой. Когда экземпляр класса `Random` задействуется несколькими потоками, его внутреннее состояние может быть нарушено и он всегда будет возвращать ноль.

ПРИМЕЧАНИЕ

Класс `System.Random` не может быть случайным в криптографическом смысле. Если нужно обеспечить качественную генерацию случайных чисел, то следует применять `RNGCryptoServiceProvider`, который генерирует криптографически сильные случайные числа.

Решение: использование объекта `ThreadLocal`. `ThreadLocal<T>` гарантирует, что каждый поток получит собственный экземпляр класса `Random`, и, следовательно, обеспечивает потокобезопасный доступ даже в многопоточной программе. В листинге 13.7 показана реализация потокобезопасного генератора случайных чисел с использованием класса `ThreadLocal<T>`, который предоставляет строго типизированный и локально ограниченный тип для создания экземпляров объектов, хранящихся отдельно для каждого потока.

Листинг 13.7. Потокобезопасный генератор случайных чисел

```
public class ThreadSafeRandom : Random      Создание потокобезопасного генератора случайных
{
    private ThreadLocal<Random> random =           чисел с использованием класса ThreadLocal<T>
        new ThreadLocal<Random>(() => new Random(MakeRandomSeed()));

    → public override int Next() => random.Value.Next();
    → public override int Next(int maxValue) =>           ↗
        random.Value.Next(maxValue);
    → public override int Next(int minValue, int maxValue) =>
        random.Value.Next(minValue, maxValue);
    → public override double NextDouble() => random.Value.NextDouble();
    → public override void NextBytes(byte[] buffer) =>
        random.Value.NextBytes(buffer);

    static int MakeRandomSeed() =>
        Guid.NewGuid().ToString().GetHashCode();
}
```

Список методов класса Random

Создание начального числа, которое не зависит от системных часов. При каждом вызове создается уникальное значение

`ThreadSafeRandom` — потокобезопасный генератор псевдослучайных чисел. Этот класс является подклассом `Random` и переопределяет методы `Next`, `NextDouble` и `NextBytes`. Метод `MakeRandomSeed` предоставляет уникальное значение для каждого экземпляра базового класса `Random`, которое не зависит от системных часов.

Конструктор `ThreadLocal<T>` принимает делегата `Func<T>` для создания локального для потока экземпляра класса `Random`. `ThreadLocal<T>.Value` используется для доступа к базовому значению. Здесь мы получаем доступ к экземпляру `ThreadSafeRandom` из параллельного цикла, чтобы имитировать конкурентную среду.

В этом примере параллельный цикл конкурентно вызывает `ThreadSafeRandom`, чтобы получить случайное число для доступа к массиву `clips`:

```
var safeRandom = new ThreadSafeRandom();

string[] clips = new string[] { "1.mp3", "2.mp3", "3.mp3", "4.mp3" };

Parallel.For(0, 1000, (i) =>
{
    var clipIndex = safeRandom.Next(4);
    var clip = clips[clipIndex];

    Console.WriteLine($"clip to play {clip} - Thread Id
                      {Thread.CurrentThread.ManagedThreadId}");
});
```

Вот результат, выведенный в консоль:

```
clip to play 2.mp3 - Thread Id 11
clip to play 2.mp3 - Thread Id 8
clip to play 1.mp3 - Thread Id 20
clip to play 2.mp3 - Thread Id 20
clip to play 4.mp3 - Thread Id 13
clip to play 1.mp3 - Thread Id 8
clip to play 4.mp3 - Thread Id 11
clip to play 3.mp3 - Thread Id 11
clip to play 2.mp3 - Thread Id 20
clip to play 3.mp3 - Thread Id 13
```

ПРИМЕЧАНИЕ

Один экземпляр `ThreadLocal<T>` занимает в памяти несколько сотен байт, поэтому важно учитывать, сколько таких активных экземпляров необходимо в каждый момент. Если программе требуется много параллельных операций, то рекомендуется работать с локальной копией, чтобы как можно реже обращаться к локальному хранилищу потока.

13.6. Полиморфный агрегатор событий

В этом разделе мы создадим инструмент для работы в программе, которая требует возникновения нескольких событий разных типов и имеет систему публикации и подписки, способную получить доступ к этим событиям.

Решение: реализация полиморфного шаблона «издатель — подписчик». На рис. 13.9 показано, как управлять событиями разных типов. В листинге 13.8 представлена реализация `EventAggregator` с использованием Rx (выделена жирным шрифтом).

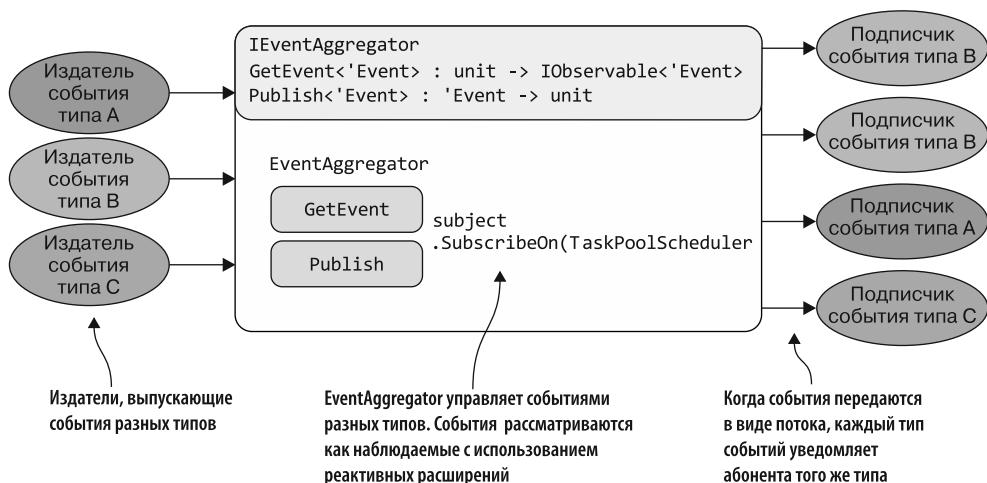


Рис. 13.9. EventAggregator управляет событиями разных типов. При публикации события EventAggregator производит сопоставление и уведомляет подписчиков и события того же типа

Листинг 13.8. EventAggregator с использованием Rx

Интерфейсы для определения контракта EventAggregator, который также реализует интерфейс IDisposable, чтобы гарантировать очистку субъекта ресурса

```

type IEventAggregator =
    inherit IDisposable
    abstract GetEvent<'Event> : unit -> IObservable<'Event>
    abstract Publish<'Event> : eventToPublish:'Event -> unit

```

Экземпляр типа Subject (Rx), который координирует регистрацию событий и уведомления о событиях

```

type internal EventAggregator() =
    let disposedErrorMessage = "The EventAggregator is already disposed."
    let subject = new Subject<obj>()

```

Извлечение события как observable в зависимости от типа

```

    interface IEventAggregator with
        member this.GetEvent<'Event>(): IObservable<'Event> =
            if (subject.IsDisposed) then failwith disposedErrorMessage

```

Подписка Observable в планировщике TaskPool, чтобы обеспечить конкурентное поведение

```

            subject.OfType<'Event>().AsObservable<'Event>()
                .SubscribeOn(TaskPoolScheduler.Default)

```

Публикация уведомлений о событиях для всех подписчиков на события данного типа

```

        member this.Publish(eventToPublish: 'Event): unit =
            if (subject.IsDisposed) then failwith disposedErrorMessage

```

member this.OnNext(eventToPublish)

```

        member this.Dispose(): unit = subject.Dispose()

```

static member Create() = new EventAggregator():>IEventAggregator

Интерфейс `IEventAggregator` способствует слабому связыванию реализации `EventAggregator`. Это означает, что потребляющий код не должен изменяться (поскольку не меняется интерфейс), даже если изменится внутренняя работа класса. Обратите внимание, что `IEventAggregator` наследуется от `IDisposable` для освобождения любых ресурсов, которые были выделены при создании экземпляра `EventAggregator`.

Методы `GetEvent` и `Publish` инкапсулируют экземпляр типа Rx `Subject`, который ведет себя как концентратор событий. `GetEvent` предоставляет `IObservable` из экземпляра `Subject`, что обеспечивает простой способ обработки подписки на события. По умолчанию тип Rx `Subject` является однопоточным, поэтому, чтобы обеспечить одновременное выполнение `EventAggregator` и использование `TaskPoolScheduler`, применяется метод расширения `SubscribeOn`. Метод `Publish` конкурентно уведомляет всех подписчиков `EventAggregator`.

Статический член `Create` создает экземпляр `EventAggregator` и предоставляет только один интерфейс `IEventAggregator`. В следующем примере кода показано, как подписаться на события и публиковать их с помощью `EventAggregator`, а также приводятся результаты выполнения программы:

```
let evtAggregator = EventAggregator.Create()

type IncrementEvent = { Value: int }
type ResetEvent = { ResetTime: DateTime }

evtAggregator
    .GetEvent<ResetEvent>()
    .ObserveOn(Scheduler.CurrentThread)
    .Subscribe(fun evt -> printfn "Counter Reset at: %A - Thread Id %d"
    ↪ evt.ResetTime Thread.CurrentThread.ManagedThreadId)

evtAggregator
    .GetEvent<IncrementEvent>()
    .ObserveOn(Scheduler.CurrentThread)
    .Subscribe(fun evt -> printfn "Counter Incremented. Value: %d - Thread
    ↪ Id %d" evt.Value Thread.CurrentThread.ManagedThreadId)

for i in [0..10] do
    evtAggregator.Publish({ Value = i })
evtAggregator.Publish({ ResetTime = DateTime(2015, 10, 21) })
```

Получается следующий результат:

```
Counter Incremented. Value: 0 - Thread Id 1
Counter Incremented. Value: 1 - Thread Id 1
Counter Incremented. Value: 2 - Thread Id 1
Counter Incremented. Value: 3 - Thread Id 1
Counter Incremented. Value: 4 - Thread Id 1
Counter Incremented. Value: 5 - Thread Id 1
Counter Incremented. Value: 6 - Thread Id 1
Counter Incremented. Value: 7 - Thread Id 1
```

```
Counter Incremented. Value: 8 - Thread Id 1
Counter Incremented. Value: 9 - Thread Id 1
Counter Incremented. Value: 10 - Thread Id 1
Counter Reset at: 10/21/2015 00:00:00 AM - Thread Id 1
```

Интересная идея `EventAggregator` заключается в способе обработки событий разных типов. В этом примере экземпляр `EventAggregator` регистрирует два разных типа событий (`IncrementEvent` и `ResetEvent`), а функция `Subscribe` отправляет уведомление только подписчикам определенного типа события.

13.7. Нестандартный Rx-планировщик для управления степенью параллелизма

Предположим, что вам нужно реализовать систему для асинхронных запросов больших потоков событий, и это требует контроля за уровнем параллелизма. Правильное решение для составления асинхронных, событийно-ориентированных программ — реактивные расширения Rx, основанные на наблюдаемых объектах и конкурентной генерации последовательностей данных. Но, как уже отмечалось в главе 6, по умолчанию Rx не являются многопоточными. Чтобы можно было использовать модель конкурентности, необходимо настроить Rx на задействование планировщика, который поддерживает многопоточность, вызвав расширение `SubscribeOn`. Например, Rx предоставляет несколько параметров настройки планировщика, включая типы `TaskPool` и `ThreadPool`, позволяющие планировать все действия, которые могут выполняться с использованием другого потока.

Но здесь есть проблема, поскольку оба планировщика по умолчанию запускаются в одном потоке и затем только после задержки примерно 500 мс могут увеличить количество потоков, если это потребуется. Такое поведение может иметь критические последствия для производительности.

Например, рассмотрим компьютер с четырьмя ядрами, на котором запланировано восемь действий. Пул потоков Rx по умолчанию начинается с одного потока. Если каждое действие занимает 2000 мс, то три действия будут поставлены в очередь и ожидать 500 мс, прежде чем размер пула потоков планировщика Rx увеличится. Следовательно, вместо того чтобы правильно выполнить четыре действия параллельно, что в общей сложности заняло бы 4 с для всех восьми действий, работа не будет завершена и за 5,5 с, поскольку три задачи будут простаивать в очереди в течение 500 мс. К счастью, стоимость расширения пула потоков — это лишь единовременные затраты. В данном случае нужно разработать собственный планировщик Rx, который будет поддерживать конкурентность с точным контролем уровня параллелизма. Он должен инициализировать внутренний пул потоков во время запуска, а не при необходимости, чтобы избежать затрат при вычислении предельного времени.

Если использовать конкурентность с Rx посредством одного из доступных планировщиков, то нет возможности настроить максимальную степень параллелизма.

Это важное ограничение, поскольку в определенных обстоятельствах желательно иметь несколько потоков, одновременно обрабатывающих поток событий.

Решение: реализация планировщика с несколькими конкурентными агентами.

Метод расширения Rx `SubscribeOn` требует передачи в качестве аргумента объекта, реализующего интерфейс `IScheduler`. Этот интерфейс определяет методы, ответственные за планирование действия, которое будет выполнено как можно скорее или же в какой-то момент в будущем. Можно построить собственный планировщик для Rx, который поддерживал бы модель конкурентности с возможностью выбора степени параллелизма, как показано на рис. 13.10.

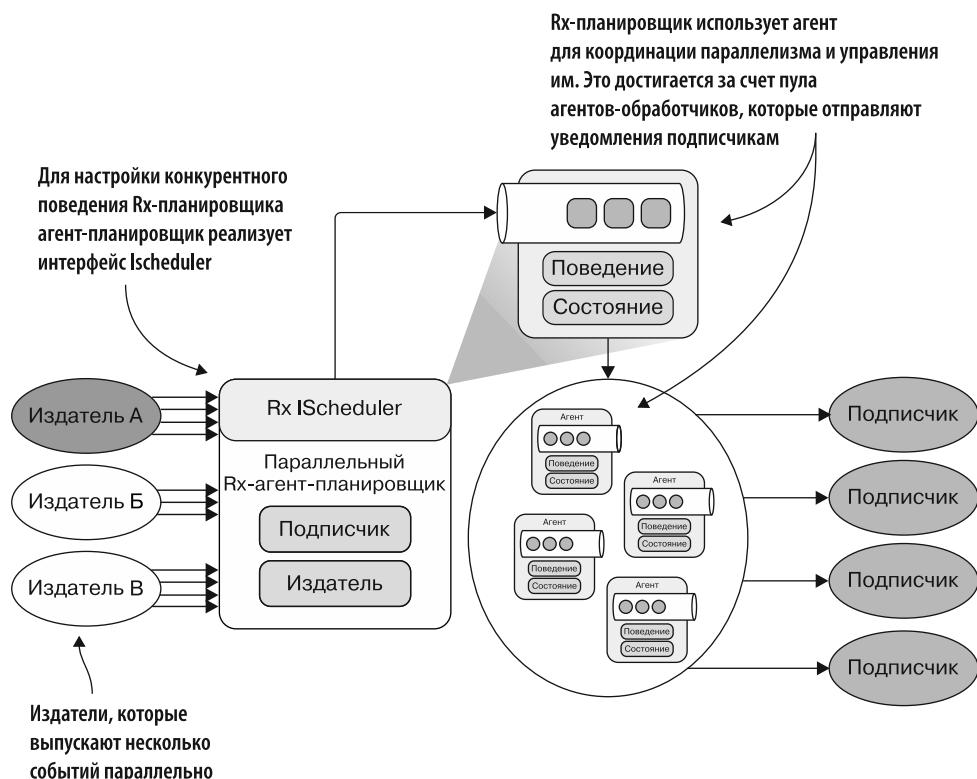


Рис. 13.10. `ParallelAgentScheduler` — это специальный планировщик, назначением которого является настройка конкурентного поведения Rx. Rx-планировщик использует агент для координации и управления параллелизмом. Это достигается за счет пула агентов-обрабочиков, которые отправляют уведомления подписчикам

В листинге 13.9 показана реализация Rx-планировщика `ParallelAgentScheduler`, где для управления степенью параллелизма используется агент `parallelWorker` (показанный в листинге 11.5). Код, на который стоит обратить внимание, выделен жирным шрифтом.

Листинг 13.9. Специальный Rx-планировщик для управления степенью параллелизма

Использование типа сообщения, чтобы запланировать работу. Ответ на сообщение представляет собой IDisposable, обернутый в канал ответа. Объект IDisposable применяется для уведомления об отмене операции/отмене подписки

Функция schedulerAgent создает экземпляр MailboxProcessor, который устанавливает приоритеты и координирует запросы на выполнение заданий

```

type ScheduleMsg = ScheduleRequest * AsyncReplyChannel<IDisposable>

let schedulerAgent (inbox:MailboxProcessor<ScheduleMsg>) =
    let rec execute (queue:IPriorityQueue<ScheduleRequest>) = async {
        → match queue |> PriorityQueue.tryPop with
        | None -> return! idle queue -1
        | Some(req, tail) ->
            let timeout =
                int <| (req.Due - DateTimeOffset.Now).TotalMilliseconds
            if timeout > 0 && (not req.IsCanceled)
            then return! idle queue timeout
            else
                if not req.IsCanceled then req.Action.Invoke()
                return! execute tail }

Когда агент находится в процессе выполнения и получает запрос на задание, он пытается извлечь задание для выполнения из внутренней очереди приоритетов. Если заданий для выполнения нет, то агент переключается в состояние ожидания

        and idle (queue:IPriorityQueue<_>) timeout = async {
            let! msg = inbox.TryReceive(timeout)
            let queue =
                match msg with
                | None -> queue
                | Some(request, replyChannel)->
                    replyChannel.Reply(Disposable.Create(fun () ->
                        request.IsCanceled <- true))
                    queue |> PriorityQueue.insert request
            return! execute queue }

Возвращая одноразовый объект, используемый для отмены запланированного действия
        idle (PriorityQueue.empty(false)) -1

Когда агент находится в состоянии ожидания и приходит запрос на выполнение задания, задание помещается в локальную очередь и назначается для выполнения

type ParallelAgentScheduler(workers:int) =
    let agent = MailboxProcessor<ScheduleMsg>
        .parallelWorker(workers, schedulerAgent)

Создание экземпляра агента parallelWorker (из главы 11) путем создания коллекции субагентов-обработчиков, передающих поведение schedulerAgent

interface IScheduler with
    member this.Schedule(state:'a, due:DateTimeOffset,
    ↳ action:ScheduledAction<'a>) =
        agent.PostAndReply(fun repl ->
            let action () = action.Invoke(this :> IScheduler, state)
            let req = ScheduleRequest(due, Func<_>(action))
            req, repl)

Реализация IScheduler, определяющего планировщик Reactive Extensions

member this.Now = DateTimeOffset.Now
member this.Schedule(state:'a, action) =
    let scheduler = this :> IScheduler
    let due = scheduler.Now
    scheduler.Schedule(state, due, action)

Отправка и планирование запроса задания для экземпляра ParallelWorker, который отправляет задания для параллельного выполнения посредством своих внутренних агентов-обработчиков

```

```
member this.Schedule(state:'a, due:TimeSpan,
                     action:ScheduledAction<'a>) =
    let scheduler = this :> IScheduler
    let due = scheduler.Now.Add(due)
    scheduler.Schedule(state, due, action)
```

ParallelAgentScheduler задает уровень конкурентности для планирования и выполнения задач, помещаемых в распределенный пул работающих агентов (F#-объектов MailboxProcessor). Обратите внимание, что все действия, отправляемые ParallelAgentScheduler, потенциально могут выполняться не по порядку. ParallelAgentScheduler можно применять в качестве Rx-планировщика, внедрив новый экземпляр в метод расширения SubscribeOn. В следующем фрагменте кода показан простой пример использования этого специального планировщика.

```
let scheduler = ParallelAgentScheduler(4)

Observable.Interval(TimeSpan.FromSeconds(0.4))
    .SubscribeOn(scheduler)
    .Subscribe(fun _ ->
        printfn "ThreadId: %A" Thread.CurrentThread.ManagedThreadId)
```

Экземпляр планировщика — объект ParallelAgentScheduler — настроен так, чтобы четыре конкурентных агента выполнялись и были готовы реагировать на отправку нового уведомления. В этом примере наблюдаемый оператор Interval отправляет уведомление каждые 0,4 с; эти уведомления обрабатываются конкурентно внутренними агентами ParallelWorker. Преимущество использования такого специального планировщика ParallelAgentScheduler состоит в отсутствии простоеев и задержек при создании новых потоков, в результате чего обеспечивается точный контроль над степенью параллелизма. Например, бывают случаи, когда нужно ограничить уровень параллелизма для анализа потока событий, когда события, ожидающие обработки, буферизуются во внутренней очереди внутренних агентов и, следовательно, не теряются.

13.8. Конкурентное реактивное масштабируемое приложение «клиент-сервер»

Задача: вам нужно создать сервер, который бы асинхронно прослушивал заданный порт на предмет входящих запросов от нескольких TCP-клиентов. Кроме того, сервер должен быть:

- реактивным;
- иметь возможность управлять большим количеством конкурентных подключений;
- масштабируемым;
- отзывчивым;
- управляемым событиями.

Эти требования гарантируют, что можно использовать функциональные операции высокого порядка для компоновки операций потока событий через соединения с TCP-сокетом в декларативном и неблокирующем стиле.

Затем клиентские запросы должны конкурентно обрабатываться сервером, а полученные ответы — отправляться обратно клиенту. Соединение с сервером по протоколу TCP (Transmission Control Protocol — протокол управления передачей) может быть как защищенным, так и незащищенным. TCP является наиболее распространенным современным интернет-протоколом, который используется для точной доставки с сохранением порядка пакетов данных при передаче от одной конечной точки до другой. TCP способен обнаружить ошибочные и отсутствующие пакеты и управляет действиями, необходимыми для их повторной отправки. Обеспечение связи в приложениях необыкновенно важно, и .NET Framework предоставляет множество различных способов, помогающих удовлетворить эту потребность.

Нам также нужна долговременная клиентская программа, которая использует TCP-сокеты для подключения к серверу. После установки соединения конечные точки клиента и сервера могут отправлять и получать байты асинхронно, иногда корректно закрывать соединение и затем снова его открывать.

Клиентская программа, которая пытается подключиться к TCP-серверу, является асинхронной, неблокирующей и способна поддерживать отзывчивость приложения даже под нагрузкой (от поддержки большого количества передаваемых данных). В этом примере клиент-серверное приложение на основе сокетов после установки соединения непрерывно передает большие объемы пакетов с высокой скоростью. Данные передаются с сервера клиенту в потоковом режиме, порциями, каждая из которых содержит историю изменений цен на акции на определенную дату. В подобном решении поток данных создается путем чтения и анализа файлов в формате CSV (Comma-Separated Values — значения, разделенные запятыми). Когда клиент получает данные, он начинает обновлять диаграмму в режиме реального времени.

Такой сценарий применим к любым операциям, использующим реактивное программирование на основе потоков. Примерами, с которыми вы можете столкнуться, являются удаленные двоичные перехватчики, программирование сокетов и другие приложения с непредсказуемым событийно-управляемым поведением — например, когда видео необходимо передавать по сети.

Решение: объединение Rx и асинхронного программирования. Для построения клиент-серверной программы, показанной в листинге 13.10, CLR-классы `TcpListener` и `TcpClient` предоставляют удобную модель для создания сервера сокетов, состоящую всего из нескольких строк кода. В сочетании с ТАР и Rx они повышают степень масштабируемости и надежности программы. Но, чтобы работать в реактивном стиле, необходимо изменить традиционную структуру приложения.

В частности, чтобы удовлетворить требования высокопроизводительной клиент-серверной TCP-программы, необходимо реализовать TCP-сокеты в асинхронном стиле. Для этого можно использовать сочетание Rx и ТАР. Реактивное

программирование особенно подходит для такого сценария, поскольку позволяет обрабатывать исходные события из любого потока, независимо от его типа (сеть, файл, память и др.). Microsoft дает следующее определение Rx.

*Реактивные расширения (Reactive Extensions, Rx) — это библиотека для со-
ставления асинхронных событийно-ориентированных программ с исполь-
зованием наблюдаемых последовательностей и операторов запросов в стиле
LINQ, а также параметризации конкурентности в асинхронных потоках дан-
ных с применением планировщиков.*

Листинг 13.10. Реактивная серверная программа TcpListener

Сбор файлов биржевых символов (csv)

```
static void ConnectServer(int port, string sslName = null)
{
    var cts = new CancellationTokenSource();
    string[] stockFiles = new string[] { "aapl.csv", "amzn.csv", "fb.csv",
    ➔ ➔ ➔ "goog.csv", "msft.csv" };

    var formatter = new BinaryFormatter(); ← .NET Binary Formatter применяется
                                            из соображений удобства. Вместо этого
                                            средства форматирования можно
                                            использовать любой другой сериализатор

    TcpListener.Create(port) ← Подписка на поток событий
        ➔ .ToAcceptTcpClientObservable() ← из ToAcceptTcpClientObservable для выполнения
        .ObserveOn(TaskPoolScheduler.Default) ← в текущем планировщике TaskPool, чтобы
        .Subscribe(client => { ← гарантировать конкурентное поведение
            using (var stream = GetServerStream(client, sslName))
            {
                stockFiles
                    ➔ .ObservablesStreams(StockData.Parse)
                    .Subscribe(async stock => {
                        var data = Serialize(formatter, stock);
                        await stream.WriteAsync(data, 0, data.Length, cts.Token);
                    });
                }
            },
            error => Console.WriteLine("Error: " + error.Message),
            () => Console.WriteLine("OnCompleted"),
            cts.Token);
    } ← Создание сетевого потока для установки
          соединения и передачи данных. Если указано
          значение sslName, то возвращаемый
          сетевой поток использует безопасный
          базовый сокет SSL

    ➔ ➔ ➔ Возвращение наблюдаемого объекта, который
          передает и анализирует историю изменений
          курса акций из каждого файла в коллекцию типа StockData
    } ← Реализация методов Observer —
          OnError и OnCompleted
```

Преобразование TcpListener в наблюдаемую последовательность на заданном порте

Подписка на уведомления о событиях от ObservablesStreams для сериализации StockData, получаемых в виде байтового массива, а затем для записи данных в сетевой поток

Для реализации масштабируемого сервера экземпляр класса `TcpListener` прослушивает входящие соединения. Когда соединение установлено, оно маршрутизируется как `TcpClient` из обработчика прослушивателя для управления `NetworkStream`. Затем этот поток применяется для чтения и записи байтов при совместном использовании данных клиентом и сервером. На рис. 13.11 показана логика подключения серверной программы.

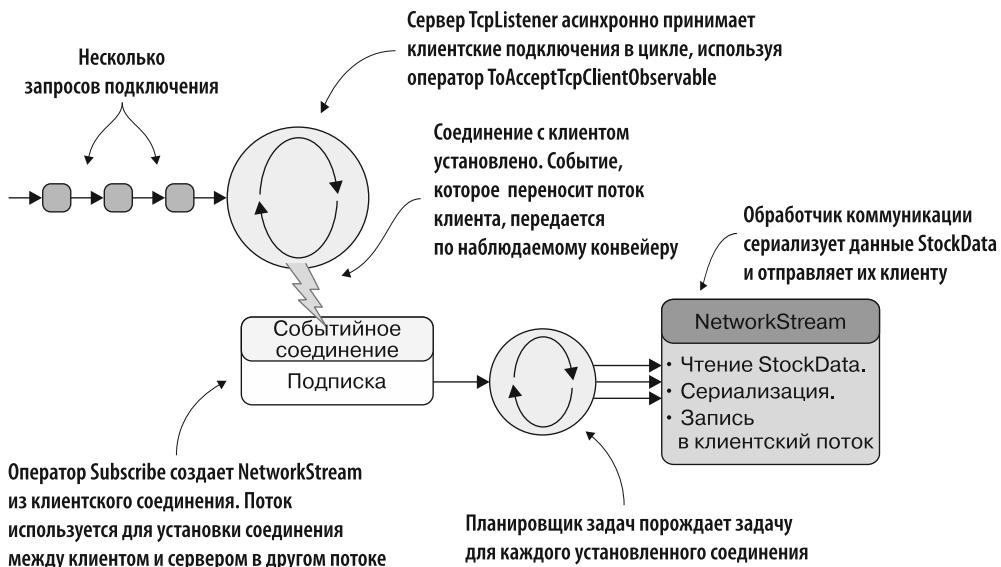


Рис. 13.11. Сервер `TcpListener` асинхронно в цикле принимает клиентские соединения. Когда соединение установлено, событие, которое переносит клиентский поток, передается по наблюдаемому конвейеру для обработки. Затем обработчики соединений начинают читать историю изменений биржевого кода, сериализуют и записывают данные в клиентский поток `NetworkStream`

В этом примере показана серверная реализация реактивного TCP-прослушивателя, который работает как наблюдаемый объект биржевого кода. Естественным подходом для прослушивателя является подписка на конечную точку и получение клиентов по мере их подключения. Это достигается с помощью метода расширения `ToAcceptTcpClientObservable`, который создает наблюдаемый объект `IObservable<TcpClient>`. Метод `ConnectServer` использует конструкцию `TcpListener.Create` для генерации `TcpListener`, применяя заданный номер порта, по которому сервер выполняет асинхронное прослушивание, и необязательное имя Secure Sockets Layer (уровень защищенных сокетов, SSL) для установления защищенного или обычного соединения.

Специальный наблюдаемый метод расширения `ToAcceptTcpClientObservable` использует заданный экземпляр `TcpListener` для предоставления сетевых служб среднего уровня через внутренний объект сокета. Когда удаленный клиент становится доступным и соединение устанавливается, для обработки новой коммуникации

создается объект `TcpClient`, который затем с помощью объекта `Task` отправляется в другой долговременный поток.

После этого, чтобы гарантировать конкурентное поведение обработчика сокета, с помощью оператора `ObserveOn` выполняется настройка планировщика, чтобы создать подписку и переместить работу в другой планировщик, `TaskPoolScheduler`. Таким образом, оператор `ToAcceptTcpClientObservable` может конкурентно управлять большим количеством элементов `TcpClient`, представленных в виде последовательности.

Затем внутренние компоненты наблюдаемого объекта `ToAcceptTcpClientObservable` извлекают из задачи ссылку `TcpClient` и создают сетевой поток, используемый как канал для отправки пакетов данных, генерируемых специальным наблюдаемым оператором `ObservableStreams`. Метод `GetServerStream` извлекает защищенный или обычный поток в соответствии со значением, переданным в `nameSsl`. Этот метод определяет, было ли задано значение `nameSsl` для SSL-соединения, и если да, то создает `SslStream`, используя `TcpClient.GetStream` и выбранное имя сервера для получения сертификата сервера.

Если же SSL не применяется, то `GetServerStream` получает `NetworkStream` от клиента с помощью метода `TcpClient.GetStream`. Реализацию метода `GetServerStream` вы найдете в исходном коде к книге. При материализации `ObservableStreams` генерируемый поток событий направляется к оператору `Subscribe`. Затем оператор асинхронно сериализует входящие данные, преобразуя их в порции байтовых массивов, которые отправляются по сети через поток клиента. Для простоты в качестве сериализатора использовано двоичное средство форматирования .NET, но вы можете заменить его на тот, какой лучше соответствует вашим потребностям.

Данные передаются по сети в виде байтовых массивов, поскольку это единственный многоразовый тип сообщений с данными, который может содержать объект любого вида. В листинге 13.11 показана реализация основного наблюдаемого оператора `ToAcceptTcpClientObservable`, используемого внутренним `TcpListener` для прослушивания удаленных соединений и соответствующей реакции.

`ToAcceptTcpClientObservable` принимает экземпляр `TcpListener`, который асинхронно в цикле `while` прослушивает сеть в поиске новых входящих запросов на соединение, пока эта операция не будет отменена посредством маркера отмены. После того как клиент успешно установит соединение, ссылка `TcpClient` передается как сообщение в последовательности. Данное сообщение выполняется в асинхронной задаче (`Task`) для обслуживания клиент-серверного взаимодействия, позволяя нескольким клиентам конкурентно подключаться к одному и тому же прослушивателю. Когда соединение установлено, другая задача начинает многократно выполнять процедуру прослушивания нового запроса на соединение.

В конце, когда наблюдаемый объект удаляется или маркер отмены запрашивает отмену операции, функция передается в оператор `Disposable.Create`, который запускается, чтобы остановить и закрыть внутренний прослушиватель сервера.

ПРИМЕЧАНИЕ

В общем случае при написании действия для очистки ресурсов и остановки передачи ненужных сообщений, передаваемых уже удаленному наблюдателю, используйте метод `Disposable.Create`.

Листинг 13.11. Асинхронный и реактивный оператор `ToAcceptTcpClientObservable`

```
Начало прослушивания
для заданного клиентского буфера

static IObservable<TcpClient> ToAcceptTcpClientObservable(this TcpListener
    listener, int backlog = 5)
{
    listener.Start(backlog);                                // Создание оператора Observable, который
                                                               // захватывает из контекста маркер отмены

    return Observable.Create<TcpClient>(async (observer, token) =>
    {
        try
        {
            while (!token.IsCancellationRequested)           // Цикл while повторяется до тех пор,
                                                               // пока маркер отмены не запросит отмену
            {
                var client = await listener.AcceptTcpClientAsync(); // Асинхронный прием новых
                                                               // клиентов от прослушивателя
                Task.Factory.StartNew(_ => observer.OnNext(client), token,
                    TaskCreationOptions.LongRunning);
            }
            observer.OnCompleted();                          // Перенаправление клиентских соединений
                                                               // к наблюдателю в асинхронную задачу,
                                                               // чтобы несколько клиентов
                                                               // могли соединиться между собой
        }
        catch (OperationCanceledException)
        {
            observer.OnCompleted();                      // Реализация методов Observer OnCompleted
                                                               // и OnError, чтобы обрабатывать случаи
                                                               // отмены и исключения соответственно
        }
        catch (Exception error)
        {
            observer.OnError(error);                   // Создание функции очистки,
                                                               // которая выполняется при удалении
                                                               // наблюдаемого объекта
        }
        finally
        {
            listener.Stop();                           // listener.Stop();
        }
    return Disposable.Create(() =>                   // listener.Stop();
    {
        listener.Stop();
        listener.Server.Dispose();
    });
});                                                 // listener.Server.Dispose();
});
```

Передаваемые данные генерируются с помощью метода расширения `ObservableStreams`, который читает и анализирует набор CSV-файлов, извлекая оттуда изменения цен на акции. Затем эти данные передаются клиентам, подключенными через `NetworkStream`.

В листинге 13.12 показана реализация ObservableStreams.

ObservableStreams генерирует серию наблюдаемых объектов типа StockData, по одному для каждого переданного filePath. Класс FileLinesStream, реализация которого для простоты опущена, открывает FileStream для каждого заданного пути к файлу. Затем он считывает из потока текстовое содержимое как наблюдаемый объект и применяет проекцию для преобразования каждой строки прочитанного текста в тип StockData. В итоге он передает результаты дальше в виде наблюдаемых объектов.

Листинг 13.12. Специальный Observable для чтения и анализа потока

Специальный метод расширения ObservableStreams принимает
в качестве аргумента список путей к файлам для обработки
и лямбда-функцию для преобразования содержимого файла

```
static IObservable<StockData> ObservableStreams
    (this IEnumerable<string> filePaths,
     Func<string, string, StockData> map, int delay = 50)
{
    return filePaths
        .Select(key =>
            new FileLinesStream<StockData>(key, row => map(key, row)))
        .Select(fsStock => {
            var startDate = new DateTime(2001, 1, 1);
            return Observable.Interval(TimeSpan.FromMilliseconds(delay))
                .Zip(fsStock.ObserveLines(), (tick, stock) => {
                    stock.Date = startDate + TimeSpan.FromDays(tick);
                    return stock;
                });
        })
        .Aggregate((o1, o2) => o1.Merge(o2));
}
```

Для каждого файла создается экземпляр FileLinesStream, который используется для создания наблюдаемого объекта. Этот наблюдаемый объект читает каждую строку файла и применяет функцию отображения для ее преобразования

Оператор Interval используется для создания задержки между уведомлениями. Чтобы отключить задержку, нужно назначить ей нулевое значение

Оператор Zip объединяет элементы из каждой последовательности по очереди. В данном случае последовательность генерируется из оператора Interval и обеспечивает задержку для каждого уведомления

Оператор Aggregate объединяет (сокращает) все наблюдаемые объекты в один

Наиболее интересной частью кода является применение двух операторов Observable, Interval и Zip, которые используются совместно для создания произвольной задержки между сообщениями, если она указана. Оператор Zip объединяет элемент из каждой последовательности по очереди. Это означает, что все записи StockData связаны с элементами, создаваемыми через заданные временные интервалы. В таком случае сочетание StockData с заданным интервалом времени обеспечивает задержку для каждого уведомления.

В конце комбинация операторов Aggregate и Merge используется для объединения наблюдаемых объектов, сгенерированных из каждого файла:

```
.Aggregate((o1, o2) => o1.Merge(o2));
```

Далее, чтобы завершить клиент-серверную программу, мы реализуем класс реактивного клиента, показанный на рис. 13.12. Реализация клиентской части показана в листинге 13.13.

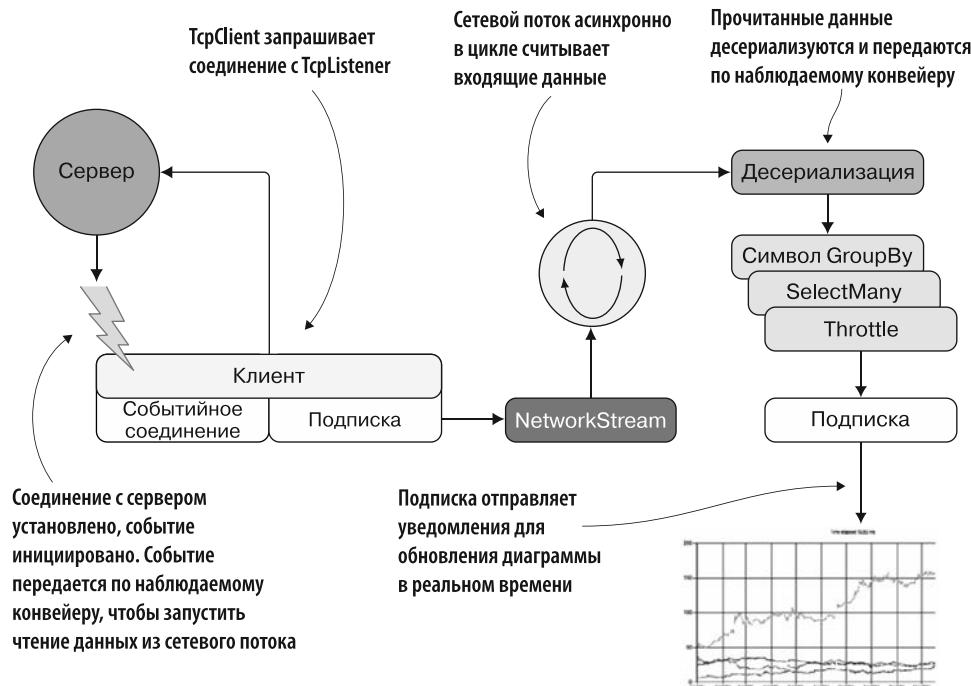


Рис. 13.12. TcpClient запрашивает соединение с сервером TcpListener. После того как соединение установлено, событие инициировано. Событие передается по наблюдаемому конвейеру, чтобы запустить чтение данных из сетевого потока

Листинг 13.13. Реактивная программа TcpClient

Создание наблюдаемого объекта из экземпляра объекта TcpClient, чтобы инициировать и рассыпать уведомления, когда установлено соединение с сервером

```

var endpoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8080);
var cts = new CancellationTokenSource();
var formatter = new BinaryFormatter();
    
```

Непрерывная доставка порций байтов из внутреннего потока до тех пор, пока он не будет прочитан до конца или не будет запрошена отмена. Байтовый массив передается по наблюдаемому конвейеру асинхронно

```

→ endpoint.ToConnectClientObservable()
    .Subscribe(client => {
        GetClientStream(client, sslName) ←
            Создание потока из сетевых потоков,
            используемых для коммуникации
            между сервером и клиентом
        .ReadObservable(0x1000, cts.Token) ←
    })
}
    
```

```

    .Select(rawData => Deserialize<StockData>(formatter, rawData))
    .GroupBy(item => item.Symbol) ← Группировка входящих данных с общим биржевым кодом, так что
        .SelectMany(group =>           для каждого биржевого кода создается свой наблюдаемый объект
            group.Throttle(TimeSpan.FromMilliseconds(20))
            .ObserveOn(TaskPoolScheduler.Default)) ←
    .ObserveOn(ctx)          Последний этап наблюдаемого конвейера — подписка
    .Subscribe(stock => ← на уведомления для обновления диаграммы в реальном времени
        UpdateChart(chart, stock, sw.ElapsedMilliseconds));
    },
    error => Console.WriteLine("Error: " + error.Message),
    () => Console.WriteLine("OnCompleted"),
    cts.Token);           Такое разделение потока по биржевому коду
                        запускает новый поток для каждого раздела

Ограничение входящих уведомлений во избежание
перегрузки потребителя. Ограничение может быть
выполнено на основе самого потока данных
(а не просто промежутка времени)

```

Код начинается с экземпляра `IPEndPoint`, который предназначен для подключения к конечной точке удаленного сервера. Наблюдаемый оператор `ToConnectClientObservable` создает экземпляр объекта `TcpClient` для установки соединения. Теперь можно использовать операторы `Observable` для подписки на подключение удаленного клиента. Когда соединение с сервером установлено, экземпляр `TcpClient` передается как наблюдаемый объект, чтобы начать получать поток данных для обработки. В этой реализации доступ к удаленному `NetworkStream` осуществляется посредством вызова метода `GetClientStream`. Поток данных поступает в наблюдаемый конвейер через оператор `ReadObservable`, который направляет входящие сообщения из внутренней последовательности `TcpClient` в другую наблюдаемую последовательность байтов типа `ArraySegment`.

Как часть кода по обработке потока, после преобразования полученных от сервера порций данных `rawData` в `StockData` оператор `GroupBy` группирует биржевые символы акций по символам в несколько наблюдаемых объектов. На этом этапе каждый наблюдаемый объект может иметь свои уникальные операции. Группировка позволяет ограничивать поток данных независимо для каждого биржевого кода, так что только акции с одинаковыми кодами будут фильтроваться с заданным временным ограничением.

Общая проблема при написании реактивного кода — это ситуация, когда события приходят слишком быстро. Быстро движущийся поток событий может перегрузить программу. В листинге 13.13, поскольку предусматривается интенсивное обновление пользовательского интерфейса, применение оператора ограничения может помочь справиться с интенсивным поступлением потоковых данных, не перегружая обновление в реальном времени.

Оператор, выполняемый после регулирования, `ObserveOn(TaskPoolScheduler.Default)`, запускает новый поток для каждого раздела, созданного `GroupBy`.

В итоге метод `Subscribe` обновляет диаграмму в реальном времени, добавляя новые значения акций. В листинге 13.14 представлена реализация оператора `ToConnectClientObservable`.

Листинг 13.14. Специальный оператор Observable `ToConnectClientObservable`

Создание наблюдаемого объекта путем передачи маркера отмены из текущего контекста

```
static IObservable<TcpClient> ToConnectClientObservable(this IPEndPoint
    ↪ endpoint)
{
    return Observable.Create<TcpClient>(async (observer, token) => {
        var client = new TcpClient();
        try
        {
            await client.ConnectAsync(endpoint.Address, endpoint.Port); ← Начало асинхронного ожидания установки соединения с сервером

            token.ThrowIfCancellationRequested(); ← Проверка, был ли отправлен маркер отмены для прекращения наблюдения за соединением

            observer.OnNext(client); ← Соединение установлено, и наблюдателям рассылаются уведомления
        }
        catch (Exception error)
        {
            observer.OnError(error); ← Когда наблюдаемый объект удаляется, TcpClient и его соединение закрываются
        }
        return Disposable.Create(() => client.Dispose()); ←
    });
}
```

`ToConnectClientObservable` создает экземпляр `TcpClient` из заданной конечной точки `IPEndPoint`, а затем пытается асинхронно подключиться к удаленному серверу. Когда соединение успешно установлено, клиентская ссылка `TcpClient` передается через наблюдаемый объект.

Последним этапом кода этой программы является наблюдаемый оператор `ReadObservable`, который предназначен для асинхронного непрерывного чтения порций данных из потока. В указанной программе поток представляет собой `NetworkStream`, созданный в результате соединения между сервером и клиентом (листинг 13.15).

При реализации этого `ReadObservable` следует учитывать важное замечание: чтобы поток был реактивным, он должен считываться порциями. Именно поэтому оператор `ReadObservable` принимает размер буфера в качестве аргумента для определения размера порций.

Назначение оператора `ReadObservable` — читать поток порциями, чтобы облегчить работу с данными, размер которых превышает объем доступной памяти, или может быть бесконечным, или иметь неизвестный размер, как бывает при потоковой передаче по сети. Кроме того, он поддерживает компонуемую природу Rx, позволяя применять к самому потоку несколько преобразований, поскольку одновременное

чтение порций данных дает возможность выполнять их преобразования в то время, пока поток все еще находится в движении. В этот момент у нас есть метод расширения, который перебирает в цикле байты из потока.

Листинг 13.15. Наблюдаемая операция чтения потока

```
Считывание из потока размера порции данных (буфера) для настройки
длины операции чтения; чтение буфера заданного размера

public static IObservable<ArraySegment<byte>> ReadObservable(this Stream
    stream, int bufferSize, CancellationToken token =
    default(CancellationToken))
{
    var buffer = new byte[bufferSize];
    var asyncRead = Observable.FromAsync<int>(async ct => {
        await stream.ReadAsync(buffer, 0, sizeof(int), ct);
        var size = BitConverter.ToInt32(buffer, 0);
        await stream.ReadAsync(buffer, 0, size, ct);
        return size});
    while (пока есть данные для чтения
        Продолжение
        чтения в цикле
        чтения в потоке)
        Считывание из потока размера порции
        данных (буфера) для настройки длины операции
        чтения; чтение буфера заданного размера
        → return Observable.While(
            () => !token.IsCancellationRequested && stream.CanRead,
            Observable.Defer(() =>
                !token.IsCancellationRequested && stream.CanRead
                ? asyncRead
                : Observable.Empty<int>())
                .Catch((Func<Exception, IObservable<int>>) ex =>
                    Observable.Empty<int>())
                    .TakeWhile(returnBuffer => returnBuffer > 0)
                    .Select(readBytes =>
                        new ArraySegment<byte>(buffer, 0, readBytes)))
                    .Finally(stream.Dispose);
    }
}

Итерации вызова наблюдаемой фабрики, которая
начинается с текущей позиции потока. Наблюдаемый
оператор Defer запускает процесс только при наличии подписчиков
```

Преобразование асинхронной
операции в наблюдаемую

Тихая обработка
ошибки с передачей
пустого результата

После того как порция данных прочитана, создается
экземпляр ArraySegment, чтобы обернуть буфер,
который затем передается наблюдателям

В коде метод расширения `FromAsync` позволяет преобразовать `Task<T>`, в нашем случае `stream.ReadAsync`, в `IObservable<T>`, чтобы рассматривать данные как поток событий и сделать возможным программирование с использованием Rx. Далее `Observable.FromAsync` создает наблюдаемый объект, который запускает операцию независимо каждый раз, когда на нее оформляется подписка.

Затем внутренний поток считывается как `Observable` в цикле `while`, пока данные не станут доступными или операция не будет отменена. Оператор `Observable Defer` ожидает, пока наблюдатель подпишется на него, а затем начинает передавать данные в виде потока. Затем на каждой итерации из потока считывается порция данных. Они передаются в буфер, который принимает форму `ArraySegment<byte>` и разбивает полезную нагрузку на фрагменты нужной длины. `ReadObservable` возвращает `IObservable`, состоящий из объектов `ArraySegment<byte>`, что является эффективным

способом управления байтовыми массивами в пуле. Размер буфера может превышать полезную нагрузку полученных байтов, поэтому `ArraySegment<byte>` содержит байтовый массив и длину полезной нагрузки.

В заключение надо отметить, что при получении и обработке данных .NET Rx позволяет создать более короткий и ясный код, чем традиционные решения. Кроме того, сложность построения реактивной клиент-серверной программы на основе TCP значительно меньше по сравнению с традиционной моделью. На самом деле вам не приходится иметь дело с объектами `TcpClient` и `TcpListener` низкого уровня, а поток байтов обрабатывается посредством абстракции высокого уровня, предлагаемой наблюдаемыми операторами.

13.9. Многоразовый специальный высокопроизводительный параллельный оператор фильтрации-отображения

Представьте, что у вас есть набор данных и вам нужно выполнить одну и ту же операцию для каждого их элемента, чтобы удовлетворить заданному условию. Это операция с ограничениями процессора, и ее выполнение может занять некоторое время. Вы решили создать специальный многократно используемый высокопроизводительный оператор для фильтрации и отображения элементов заданной коллекции. Сочетание фильтрации и преобразования элементов коллекции является обычной операцией при анализе структур данных. Ее можно решить, используя LINQ или PLINQ параллельно с операторами `Where` и `Select`; но есть и более эффективное решение с точки зрения производительности. Как мы видели в подразделе 5.2.1, для каждого вызова и повторного использования операторов высшего порядка, таких как отображение (`Select`), фильтрация (`Where`) и другие аналогичные функции запросов PLINQ (и LINQ), как показано на рис. 13.13, создаются промежуточные последовательности, которые чрезсчур увеличивают размеры выделяемой памяти. Это связано с внутренней функциональной природой LINQ и PLINQ, где коллекции трансформируются, вместо того чтобы изменяться. В случае трансформации больших последовательностей плата в виде затрат на сборку мусора и освобождение памяти становится все более высокой, что негативно сказывается на производительности программы.

В данном примере мы хотим получить сумму всех простых чисел для 100 млн цифр.

Решение: объединение параллельных операций фильтрации и отображения. Реализация специального параллельного оператора фильтрации и отображения с максимальной производительностью требует уделить внимание тому, чтобы свести к минимуму (или устраниТЬ) ненужное временное размещение данных в памяти, как показано на рис. 13.14. Такая технология сокращения выделения памяти для данных во время манипулирования ими для повышения производительности программы называется *усечением*.

```
[numbers].Where(IsPrime).Select(ToPow)
```

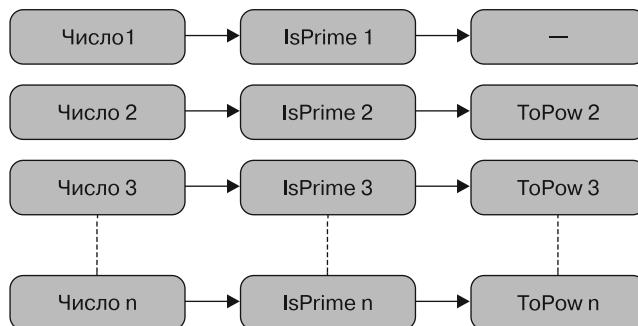
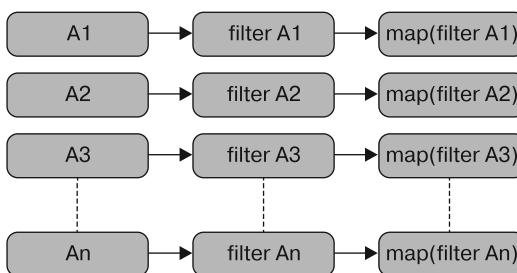


Рис. 13.13. На этой схеме каждое число (первый столбец) сначала фильтруется с помощью функции IsPrime (второй столбец), чтобы проверить, является ли оно простым. Затем простые числа передаются в функцию ToPow (третий столбец). Например, первое значение, число 1, не является простым, поэтому функция ToPow для него не выполняется

```
[source].Where(filter).Select(map)
```



```
[source].FilterMap(filter, map)
```

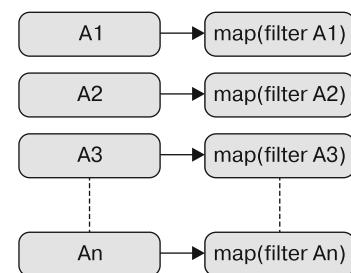


Рис. 13.14. На левой схеме показаны операции Where и Select для заданного источника, выполняемые за несколько шагов, что приводит к дополнительным операциям выделения памяти и, следовательно, к дополнительным этапам сборки мусора. На правой схеме показано, что применение операций Where и Select (фильтрация и отображение) совместно, за один шаг позволяет избежать дополнительного выделения памяти и уменьшает количество этапов сборки мусора, что увеличивает скорость работы программы

В листинге 13.16 показан код функции `ParallelFilterMap`, в которой используется цикл `Parallel.ForEach` для устранения промежуточных операций выделения памяти путем обработки только одного массива вместо создания временной коллекции для каждого оператора.

Параллельный цикл `ForEach` применяет функции `predicate` и `map` к каждому элементу входной коллекции. В общем случае, если тело параллельного цикла выполняет небольшой объем работы, лучшая производительность достигается при условии разбиения итераций на более крупные единицы работы. Причиной этого являются издержки на обработку цикла, которые включают в себя затраты на управление рабочими потоками и вызов метода-делегата. Соответственно, рекомендуется разделять пространство параллельных итераций на определенную константу с помо-

щью конструктора `Partitioner.Create`. Затем каждое тело цикла вызывает функции фильтрации и отображения для определенного диапазона элементов, амортизируя вызовы делегата тела цикла.

ПРИМЕЧАНИЕ

Из-за параллелизма порядок обработки значений не гарантирует, что результат будет получен в том же порядке.

Листинг 13.16. Оператор ParallelFilterMap

Метод расширения принимает лямбда-функции фильтрации (`predicate`) и отображения (`map`) в качестве входных значений из источника

```
static TOutput[] ParallelFilterMap<TInput, TOutput>(this IList<TInput>
    input, Func<TInput, Boolean> predicate,
    Func<TInput, TOutput> transform,
    ParallelOptions parallelOptions = null)
{
    parallelOptions = parallelOptions ?? new ParallelOptions();
    var atomResult = new Atom<ImmutableList<List<TOutput>>> ← Создание экземпляра объекта Atom (описан в главе 3) для применения операций обновления путем сравнения с обменом к внутреннему ImmutableList в поточно-ориентированном стиле
    Каждый поток использует
    экземпляр Local-Thread из List<TOutput> На каждой итерации выполняется независимый
    для выполнения изолированных, поток (задача) из пула потоков, который
    поточно-ориентированных операций выполняет операции фильтрации
    Parallel.ForEach(Partitioner.Create(0, input.Count), parallelOptions, и отображения для разбитого
        () => new List<TOutput>(), на порции набора данных
        delegate (Tuple<int, int> range, ParallelLoopState state, из входного источника
            List<TOutput> localList)
    {
        for (int j = range.Item1; j < range.Item2; j++) ←
        {
            var item = input[j];
            if (predicate(item)) ← Применение функции фильтрации
                localList.Add(transform(item)); ← и отображения к каждому элементу
            } текущего набора данных,
        return localList;
    }, localList => atomResult.Swap(r => r.Add(localList))); разбитого на порции
    return atomResult.Value.SelectMany(id => id).ToArray(); В конце результат
} преобразуется в массив
После каждой итерации общий объект Atom atomResult
обновляет внутренний ImmutableList
```

На каждой итерации цикла `ForEach` выполняется анонимный вызов делегата, который приводит к излишним затратам на выделение памяти и, следовательно, к падению производительности. Один вызов выполняется для функции фильтрации, второй — для функции отображения, и еще один — для делегата, переданного в параллельный цикл. Решение состоит в том, чтобы настроить параллельный цикл,

специфичный для операций фильтрации и отображения, таким образом, чтобы избежать дополнительных вызовов тела делегата.

Параллельный оператор `ForEach` формирует набор потоков, каждый из которых вычисляет промежуточный результат, выполняя функции фильтрации и отображения для своего раздела данных и помещая значение в свой выделенный слот промежуточного массива.

Каждый поток (задача), управляемый параллельным циклом, захватывает изолированный экземпляр локального `List<TOutput>` посредством концепции локальных значений. Локальные значения — это переменные, которые существуют локально в параллельном цикле. Тело цикла может обращаться к данным значениям напрямую, не беспокоясь о синхронизации.

ПРИМЕЧАНИЕ

Причина использования локального, изолированного экземпляра `List<TOutput>` состоит в том, чтобы избежать чрезмерной конкуренции, которая возникает, когда слишком много потоков пытаются одновременно получить доступ к общему ресурсу, что приводит к снижению производительности.

Каждый раздел вычисляет свое промежуточное значение; затем данные значения объединяются в общее конечное значение.

После завершения цикла, когда программа готова агрегировать все локальные результаты, это делается с помощью делегата `localFinally`. Но делегату требуется синхронизированный доступ к переменной, которая содержит итоговый результат. Для преодоления такого ограничения используется экземпляр коллекции `ImmutableList`; он позволяет объединить конечные результаты в поточно-ориентированном стиле.

ПРИМЕЧАНИЕ

Операции `Write` (такие как добавление элемента) в неизменяемые коллекции вместо того, чтобы изменять существующий экземпляр, возвращают новый неизменяемый экземпляр. Это не так расточительно, как может показаться на первый взгляд, потому что неизменяемые коллекции имеют общую память.

Обратите внимание: `ImmutableList` инкапсулирован в объект `Atom`, описанный в главе 3. Объект `Atom` использует стратегию сравнения с обменом (`compare-and-swap`, CAS) для применения потокобезопасных операций записи и обновления объектов без необходимости блокировок и других форм примитивной синхронизации. В следующем примере класс `Atom` содержит ссылку на неизменяемый список и обновляет его автоматически.

В данном фрагменте кода проверяется параллельная сумма простых чисел из 100 млн цифр:

```
bool IsPrime(int n)
{
    if (n == 1) return false;
    if (n == 2) return true;
    var boundary = (int) Math.Floor(Math.Sqrt(n));
```

```

for (int i = 2; i <= boundary; ++i)
    if (n % i == 0) return false;
return true;
}

BigInteger ToPow(int n) => (BigInteger) Math.BigMul(n, n);
var nums = Enumerable.Range(0, 100000000).ToList();

BigInteger SeqOperation() =>
    nums.Where(IsPrime).Select(ToPow).Aggregate(BigInteger.Add);

BigInteger ParallelLinqOperation() =>
    nums.AsParallel().Where(IsPrime).Select(ToPow).Aggregate(BigInteger.Add);

BigInteger ParallelFilterMapInline() =>
    nums.ParallelFilterMap(IsPrime, ToPow).Aggregate(BigInteger.Add);

```

На рис. 13.15 приводится сравнение последовательного кода (в качестве образца), PLINQ-версии и специального оператора `ParallelFilterMap`. На рисунке показан результат тестового кода, вычисляющего сумму простых чисел для 100 млн цифр. Тестируирование проводилось на четырехъядерном компьютере с 6 Гбайт оперативной памяти. Среднее время выполнения последовательного кода составило в среднем 196 482 с; этот результат был использован в качестве базового. PLINQ-версия кода выполнилась за 74 926 с — почти в три раза быстрее, что и следовало ожидать на четырехъядерном компьютере. Специальный оператор `ParallelFilterMap` оказался самым быстрым — код выполнился примерно за 52 566 с.

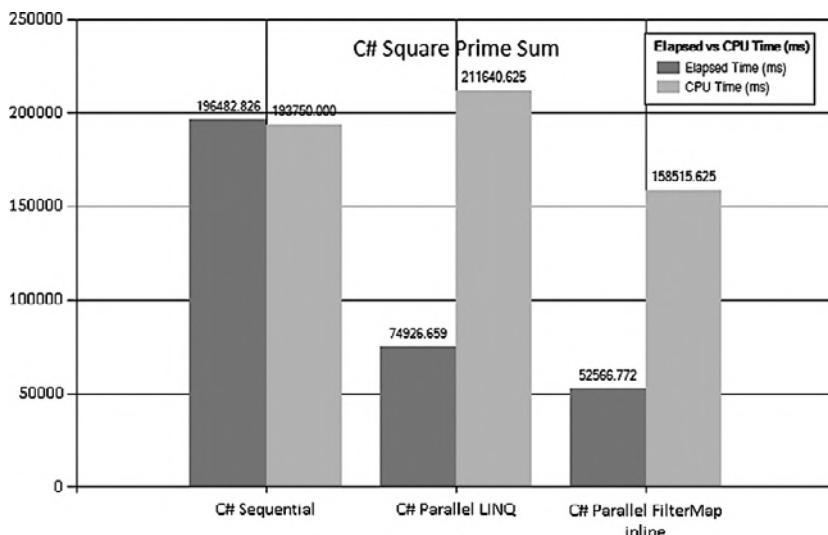


Рис. 13.15. Схема результатов сравнительных тестов для последовательной версии кода, Parallel LINQ и специального оператора `ParallelFilterMap`. На четырехъядерном компьютере специальный оператор `ParallelFilterMap` выполнился примерно на 80 % быстрее, чем последовательная версия кода, и на 30 % быстрее, чем версия PLINQ

13.10. Неблокирующая синхронная модель передачи сообщений

Предположим, что вам нужно построить масштабируемую программу, способную выполнять большое количество операций без блокировки каких-либо потоков. Вам нужна программа, которая загружает, обрабатывает и сохраняет, например, большое количество изображений. Эти операции выполняются несколькими потоками совместно, что позволяет оптимизировать ресурсы без блокировки потоков и без угрозы для производительности программы.

Подобно шаблону «поставщик – потребитель», существует два потока данных. Один – входной, с него начинается обработка, затем следуют промежуточные этапы для преобразования данных, а затем – выход с конечным результатом операций. Эти процессы, поставщик и потребитель, разделяют общий буфер фиксированного размера, применяемый как очередь. Очередь буферизуется для увеличения общей скорости и повышения пропускной способности, что обеспечивает доступ к ней нескольким потребителям и поставщикам. Фактически, когда очередь может безопасно использоваться несколькими потребителями и поставщиками, можно легко изменить уровень конкурентности для разных частей конвейера во время выполнения программы. Однако поставщик может выполнять запись в очередь, когда она не заполнена, или, наоборот, блокировать ее, когда очередь заполнена. С другой стороны, потребитель может читать данные из очереди, когда она не пуста, но будет заблокирован, если очередь пуста. Мы хотим реализовать шаблон «поставщик – потребитель» на основе передачи сообщений, чтобы избежать блокировки потоков и получить максимальную масштабируемость приложения.

Решение: координация полезной нагрузки между операциями с использованием агентной модели программирования. Есть две разновидности моделей передачи сообщений для конкурентных систем: синхронная и асинхронная. Вы уже знакомы с асинхронными моделями, такими как агентная модель (модель акторов), объясненная в главах 11 и 12, – моделями, основанными на асинхронной передаче сообщений. В этом рецепте мы воспользуемся синхронной версией передачи сообщений, которая также известна как передача последовательных процессов (Communicating Sequential Processes, CSP).

CSP имеет много общего с моделью акторов – обе основаны на передаче сообщений. Но в CSP упор делается не на объекты, между которыми происходит связь, а на каналы, применяемые для коммуникации.

Синхронная передача сообщений CSP в моделях конкурентного программирования используется для обмена данными между каналами, который может быть рассчитан на несколько потоков и выполняться параллельно. Каналы аналогичны рабочим потокам, которые обмениваются данными между собой напрямую, путем публикации сообщений, а другие каналы могут прослушивать эти сообщения, так что отправитель не знает, кто его слушает.

Канал можно представить как потокобезопасную очередь; любая задача со ссылкой на канал может добавлять сообщения в один конец канала, и любая задача со

ссылкой на канал может удалять сообщения с другого конца канала. Модель канала показана на рис. 13.16.

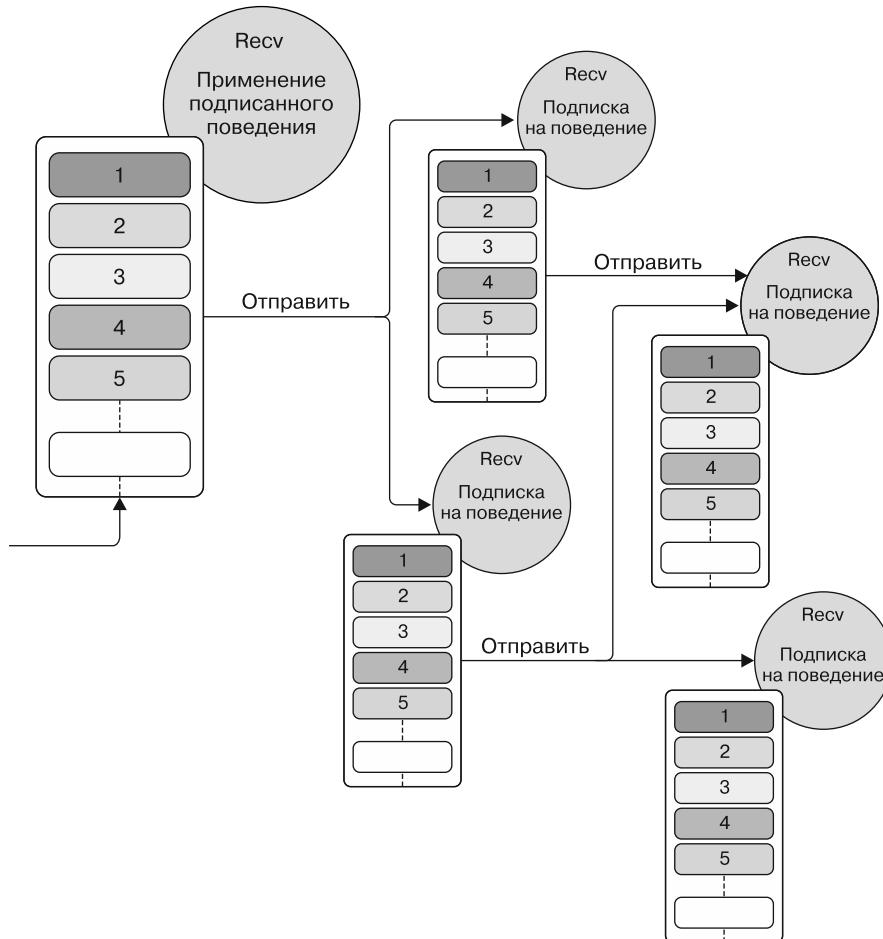


Рис. 13.16. Канал получает (Recv) сообщение и применяет подписанное поведение.

Каналы взаимодействуют между собой путем отправки (Send) сообщений, часто создавая взаимосвязанную систему, похожую на модель акторов. Каждый канал содержит локальную очередь сообщений, используемую для синхронизации связи с другими каналами без блокировок

Каналу не нужно знать о том, какой канал будет обрабатывать сообщение далее по конвейеру. Ему нужно только знать, кому каналу пересыпать сообщения. С другой стороны, прослушиватели каналов могут подписываться и отписываться, и это никак не влияет на каналы, отправляющие сообщения. Такая конструкция способствует слабому связыванию между каналами.

Основным преимуществом CSP является гибкость: каналы — объекты первого класса и могут создаваться, записывать, считывать и передавать данные между задачами независимо друг от друга. В листинге 13.17 показана реализация канала на F#, где можно использовать `MailboxProcessor` для внутренней синхронизации сообщений благодаря близкому сходству с агентной моделью программирования. Те же концепции применимы и на C#. Полную реализацию на C# с использованием TDF вы найдете в исходном коде к книге.

Листинг 13.17. ChannelAgent для реализации CSP с использованием MailboxProcessor

```

Использование размеченного объединения
для определения типа сообщения, которое отправляется
ChannelAgent для координации операций канала

→ type internal ChannelMsg<'a> =
| Recv of ('a -> unit) * AsyncReplyChannel<unit>
| Send of 'a * (unit -> unit) * AsyncReplyChannel<unit>

type [<Sealed>] ChannelAgent<'a>() =
    let agent = MailboxProcessor<ChannelMsg<'a>>.Start(fun inbox ->
        let readers = Queue<'a -> unit>()
        let writers = Queue<'a * (unit -> unit)>()

        let rec loop() = async {
            let! msg = inbox.Receive()
            match msg with
            | Recv(ok, reply) -> ←
                if writers.Count = 0 then
                    readers.Enqueue ok
                    reply.Reply( () )
                else
                    let (value, cont) = writers.Dequeue()
                    TaskPool.Spawn cont
                    reply.Reply( (ok value) )
                    return! loop()
            | Send(x, ok, reply) -> ←
                if readers.Count = 0 then
                    writers.Enqueue(x, ok)
                    reply.Reply( () )
                else
                    let cont = readers.Dequeue()
                    TaskPool.Spawn ok
                    reply.Reply( (cont x) )
                    return! loop()
        }
        loop()
    )

    member this.Recv(ok: 'a -> unit) = ←
        agent.PostAndAsyncReply(fun ch -> Recv(ok, ch)) |> Async.Ignore

    member this.Send(value: 'a, ok:unit -> unit) = ←

```

Применение внутренних очередей для отслеживания
операций чтения и записи канала

При получении сообщения `Recv`,
если текущая очередь записи
пуста, функция чтения ставится
в очередь, ожидая, пока функция
записи уравновесит работу

При получении сообщения `Send`,
если текущая очередь чтения
пуста, функция записи ставится
в очередь, ожидая, пока функция
чтения уравновесит работу

При получении сообщения `Recv`,
если в очереди
есть хотя бы
одна функция
записи, создается
задача
для выполнения
функции чтения

```

agent.PostAndAsyncReply(fun ch -> Send(value, ok, ch)) |> Async.
Ignore

member this.Recv() =
    Async.FromContinuations(fun (ok, _,_) ->
        agent.PostAndAsyncReply(fun ch -> Recv(ok, ch))
    |> Async.RunSynchronously)

member this.Send (value:'a) =
    Async.FromContinuations(fun (ok, _,_) ->
        agent.PostAndAsyncReply(fun ch -> Send(value, ok, ch))
    |> Async.RunSynchronously )

let run (action:Async<_>) = action |> Async.Ignore |> Async.Start
    ↗ let rec subscribe (chan:ChannelAgent<_>) (handler:'a -> unit) =
        chan.Recv(fun value -> handler value
                    subscribe chan handler) |> run      Запуск асинхронной
                                                операции в отдельном
                                                потоке с выводом результата
    Эта вспомогательная функция регистрирует
    обработчик, применяемый к следующему
    доступному сообщению в канале. Функция
    рекурсивно и асинхронно (без блокировки)
    ожидает сообщений из канала (без блокировки)

```

Размеченное объединение `ChannelMsg` содержит тип сообщения, которое обрабатывает `ChannelAgent`. При поступлении сообщения вариант `Recv` используется для выполнения поведения, применяемого к переданной полезной нагрузке. Вариант `Send` задействуется для передачи сообщения в канал.

Внутренний `MailboxProcessor` содержит две параметрические очереди, по одной для каждой операции, `Recv` или `Send`. Как видим, когда сообщение принимается либо отправляется, поведение агента в функции `loop()` проверяет количество доступных сообщений для балансировки нагрузки и синхронизирует связь без блокировки потоков. `ChannelAgent` принимает функции продолжения с помощью операций `Recv` и `Send`. Если обнаружено совпадение, то продолжение вызывается немедленно; иначе оно ставится в очередь. Имейте в виду, что синхронный канал в итоге выдает результат, поэтому вызов логически блокируется. Но при использовании асинхронных рабочих процессов F# никакие фактические потоки не блокируются во время ожидания.

Последние две функции в коде помогают запустить операцию канала (обычно `Send`), тогда как функция `subscribe` задействуется для регистрации и применения обработчика к полученным сообщениям. Эта функция выполняется рекурсивно и асинхронно, ожидая сообщений из канала.

Функция `TaskPool.Spawn` предполагает функцию с сигнатурой `(unit -> unit) -> unit`, которая выполняет ветвление вычислений в текущем планировщике потоков. В листинге 13.18 показана реализация `TaskPool` с использованием концепций, описанных в главе 7.

Тип записи `Context` используется для захвата `ExecutionContext` в тот момент, когда функция продолжения `cont` передается в пул. `TaskPool` инициализирует тип `MailboxProcessor parallelWorker` для обработки нескольких конкурентных потребителей и поставщиков (описание реализации и подробную информацию об агенте `ParallelWorker` вы найдете в главе 11).

Листинг 13.18. Выделенный агент TaskPool (MailboxProcessor)

```

    Конструктор TaskPool принимает количество
    обработчиков, чтобы установить уровень параллелизма
    Использование типа записи, чтобы обернуть
    текущий ExecutionContext, захваченный
    при добавлении операции cont в TaskPool

    type private Context = {cont:unit -> unit; context:ExecutionContext} ←

    type TaskPool private (numWorkers) =
        let worker (inbox: MailboxProcessor<Context>) = ← Установка поведения
            let rec loop() = async {
                let! ctx = inbox.Receive()
                let ec = ctx.context.CreateCopy() ← каждого агента-обрабатчика
                ExecutionContext.Run(ec,
                    (fun _ -> ctx.cont()), null)
                return! loop() }
            loop()
        let agent = MailboxProcessor<Context>.parallelWorker(numWorkers,
        ↵ worker) ← Обработка типа Context, полученного
        ↵ одним из агентов-обрабатчиков,
        ↵ с применением захваченного
        ↵ ExecutionContext

        static let self = TaskPool(2) ← Создание экземпляра F# MailboxProcessor parallelWorker
        ↵ для конкурентного выполнения нескольких операций,
        ↵ числом которых ограничено степенью параллелизма

    member private this.Add (continutaion:unit -> unit) =
        let ctx = { cont = continuaion;
                    context = ExecutionContext.Capture() } ←
        agent.Post(ctx)
    static member Spawn (continuation:unit -> unit) = ← Добавление в TaskPool
        self.Add continuation
        действия продолжения. Текущий
        ExecutionContext захватывается
        и отправляется агенту ParallelWorker
        в виде типа записи Context

Когда задача продолжения отправляется внутреннему агенту,
текущий контекст Execution захватывается и передается
как часть полезной нагрузки сообщения

```

Назначение `TaskPool` состоит в том, чтобы контролировать, сколько задач можно запланировать и выделить для выполнения функции продолжения в коротком цикле. В этом примере выполняется только одна задача, но их может быть любое количество.

`Add` ставит в очередь заданную функцию продолжения, которая будет выполняться, когда поток в канале предложит одну коммуникацию и другой поток предложит соответствующую ей другую коммуникацию. Пока такая компенсация между каналами не будет достигнута, поток будет ожидать в асинхронном режиме.

В следующем фрагменте кода `ChannelAgent` реализует конвейер CSP, который загружает изображение, преобразует его, а затем сохраняет вновь созданное изображение в локальной папке `MyPicture`:

```
let rec subscribe (chan:ChannelAgent<_>) (handler:'a -> unit) =
    chan.Recv(fun value -> handler value
                subscribe chan handler) |> run

let chanLoadImage = ChannelAgent<string>()
let chanApply3DEffect = ChannelAgent<ImageInfo>()
let chanSaveImage = ChannelAgent<ImageInfo>()

subscribe chanLoadImage (fun image ->
    let bitmap = new Bitmap(image)
    let imageInfo = { Path = Environment.GetFolderPath(Environment.
        SpecialFolder.MyPictures)
                    Name = Path.GetFileName(image)
                    Image = bitmap }
    chanApply3DEffect.Send imageInfo |> run)

subscribe chanApply3DEffect (fun imageInfo ->
    let bitmap = convertImageTo3D imageInfo.Image
    let imageInfo = { imageInfo with Image = bitmap }
    chanSaveImage.Send imageInfo |> run)

subscribe chanSaveImage (fun imageInfo ->
    printfn "Saving image %s" imageInfo.Name
    let destination = Path.Combine(imageInfo.Path, imageInfo.Name)
    imageInfo.Image.Save(destination))

let loadImages() =
    let images = Directory.GetFiles(@"..\Images")
    for image in images do
        chanLoadImage.Send image |> run

loadImages()
```

Как видим, реализовать конвейер на основе CSP легко. После того как будут определены каналы `chanLoadImage`, `chanApply3DEffect` и `chanSaveImage`, нужно зарегистрировать их поведение, используя функцию `subscribe`. Когда сообщение доступно для обработки, применяется поведение.

13.11. Координация конкурентных заданий с использованием агентной модели программирования

Концепции параллелизма и асинхронности широко освещались в предыдущих главах этой книги. В главе 9 было показано, насколько мощным и удобным является оператор `Async.Parallel` для параллельного выполнения большого количества асинхронных операций. Однако часто требуется выполнить отображение для

последовательности асинхронных операций и выполнить эти функции для элементов параллельно. В таком случае возможное решение может быть реализовано так:

```
let inline asyncFor(operations: #seq<'a> Async, map:'a -> 'b) =
    Async.map (Seq.map map) operations
```

Как бы вы ограничили и задали степень параллелизма для обработки элементов, чтобы сбалансировать потребление ресурсов? Эта проблема возникает на удивление часто, когда программа выполняет операции с большой нагрузкой на процессор и нет причин запускать больше потоков, чем число процессоров в машине. Если запустить слишком много конкурентных потоков, то конкуренция и переключение контекста сделают программу чрезвычайно неэффективной даже для нескольких сотен задач. Это проблема ограничения. Как ограничить асинхронные вычисления и вычисления с ограничениями процессора, чтобы ожидать результатов без блокировки? Такую задачу еще усложняет тот факт, что данные асинхронные операции порождаются во время выполнения, так что общее число асинхронных заданий для выполнения неизвестно.

Решение: реализация агента, выполняющего задания с настроенной степенью параллелизма. Решение заключается в использовании агентной модели для реализации координатора заданий, который позволяет регулировать степень параллелизма, ограничивая количество параллельных задач, как показано на рис. 13.17. В этом случае единственной задачей агента является ограничение количества конкурентных задач и возвращение результата выполнения каждой операции без блокировки. Кроме того, агент должен удобным образом предоставлять наблюдаемый канал, где можно зарегистрироваться, чтобы получать уведомления о поступлении новых вычисленных результатов.

Определим агент, который может выполнять параллельные операции с ограничениями. Агент должен получать сообщение, но также должен возвращать вызывающему объекту или подписчику ответ с вычисленным результатом.

В листинге 13.19 реализация `TamingAgent` выполняет асинхронные операции, эффективно ограничивая степень параллелизма. Когда количество конкурентных операций превышает этот предел, данные операции ставятся в очередь и обрабатываются позднее.

Размеченнное объединение `JobRequest` описывает тип сообщения для агента `tamingAgent`. Это сообщение имеет вариант `Job`, который обрабатывает значение для отправки с целью вычислений и возвращает канал с результатом. Вариант `Completed` используется агентом для уведомления о прекращении вычислений и готовности к обработке следующего доступного задания. Наконец, вариант `Quit` (сообщение) отправляется для остановки агента, когда это необходимо.

Конструктор `TamingAgent` принимает два аргумента: ограничение конкурентного выполнения и асинхронную операцию для каждого задания. Тело типа `TamingAgent` опирается на две взаимно рекурсивные функции для отслеживания количества конкурентно выполняемых операций. Если агент запускается с нулевым количеством операций или количество выполняемых заданий не превышает заданную

степень параллелизма, выполняемая функция будет ожидать новых входящих сообщений для обработки. И наоборот, если число выполняемых заданий достигает заданного предела, поток выполнения агента переключает функцию в режим ожидания. Агент использует оператор Scan, чтобы дождаться сообщения для выгрузки остальных.

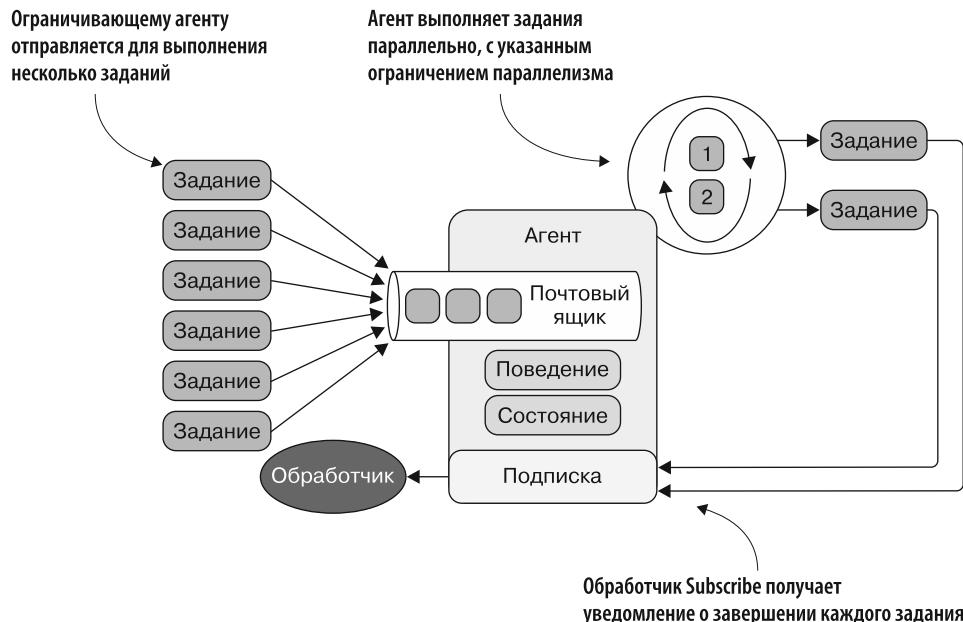


Рис. 13.17. TamingAgent выполняет задания параллельно, будучи ограниченным заданной степенью параллелизма. Когда операция завершается, оператор Subscribe уведомляет зарегистрированных обработчиков о результатах задания

Листинг 13.19. TamingAgent

```
type JobRequest<'T, 'R> = 
    | Ask of 'T * AsyncReplyChannel<'R>
    | Completed
    | Quit
```

Использование размеченного объединения, представляющего сообщение для отправки TamingAgent, чтобы начать новое задание и выдать уведомление о его завершении

```
type TamingAgent<'T, 'R>(limit, operation: 'T -> Async<'R>) =
    let jobCompleted = new Event<'R>() ←
        | Объект события применяется для уведомления подписчика о завершении задания
    let tamingAgent = Agent<JobRequest<'T, 'R>>.Start(fun agent ->
        let dispose() = (agent :> IDisposable).Dispose() ←
            | Вспомогательные функции удаляют и останавливают TamingAgent
        let rec running jobCount = async {
            let! msg = agent.Receive() ←
                | Представление состояния, когда агент работает
            match msg with
            | msg: JobRequest<'T, 'R> -> jobCompleted.Trigger(msg.Result)
            | Completed -> jobCompleted.Trigger()
            | Quit -> dispose()
        }
    )
```

```

Completed -> return! running (jobCount - 1)
Ask(job, reply) ->
do!
    async { try
        Когда задание завершено,
        запускается событие jobCompleted
        для уведомления подписчиков
        Возвращение результата
        выполненного задания
        вызывающему объекту
    let! result = operation job
    jobCompleted.Trigger result
    reply.Reply(result)
    finally agent.Post(Completed) }

Представление состояния простого,
когда агент заблокирован, поскольку
достигнут предел конкурентных заданий
Постановка в очередь указанного асинхронного
рабочего процесса для обработки в отдельном потоке,
чтобы гарантировать конкурентное поведение
    |> Async.StartChild |> Async.Ignore
    if jobCount <= limit - 1 then return! running (jobCount + 1)
    else return! Idle ()

and idle () =
    agent.Scan(function
        | Completed -> Some(running (limit - 1))
        | _ -> None)
    running 0

Предоставление поддержки для подписки
на событие jobCompleted как Observable
member this.Ask(value) = tamingAgent
    .PostAndAsyncReply(fun ch -> Ask(value, ch))
member this.Stop() = tamingAgent.Post(Quit)
member x.Subscribe(action) = jobCompleted.Publish |>
    Observable.subscribe(action)

```

Запуск элемента задания и продолжение работы

Уменьшение количества рабочих элементов после завершения задания

Асинхронное выполнение задания для получения результата

Когда работа завершается, папка входящих сообщений отправляет сама себе уведомление, чтобы уменьшить количество заданий

Использование функции Scan для ожидания завершения работы и изменения состояния агента

Постановка операции в очередь и асинхронное ожидание ответа

Оператор `Scan` применяется в F#-агенте `MailboxProcessor` для обработки подмножества сообщений заданного типа. Оператор `Scan` принимает лямбда-функцию, возвращающую тип `Option`. Сообщения, которые требуется найти в процессе сканирования, должны возвращать `Some`, а сообщения, которые в данный момент следует проигнорировать, должны возвращать `None`.

Операция, передаваемая в конструктор, имеет сигнатуру '`T -> Async<'R>`', которая напоминает функцию `Async.map`. Эта функция применяется к каждому заданию, отправляемому агенту через метод-член `Ask`, который принимает тип значения, переданный агенту, чтобы инициализировать или поставить в очередь новое задание. Когда вычисление завершается, подписанные на внутреннее событие `jobCompleted` получают уведомление о новом результате, который также асинхронно возвращается вызывающему объекту, отправившему сообщение по каналу `AsyncReplyChannel`.

Как уже отмечалось, назначение события `jobCompleted` состоит в том, чтобы уведомить подписчиков, зарегистрировавших функцию обратного вызова через метод-член `Subscribe`, который использует модуль `Observable` для удобства и гибкости.

В листинге 13.20 показано, как применять TamingAgent для преобразования набора изображений. Этот пример подобен примеру с CPS Channel, что позволяет сравнивать разные варианты кода.

Листинг 13.20. Использование TamingAgent для преобразования изображения

```

let loadImage = (fun (imagePath:string) -> async {
    let bitmap = new Bitmap(imagePath)
    return { Path = Environment.GetFolderPath(Environment.SpecialFolder.
        MyPictures)
        Name = Path.GetFileName(imagePath)
        Image = bitmap } }) ← Загрузка изображения
                                                по заданному пути к файлу,
                                                возвращаем тип записи с загруженным
                                                изображением и информацией о нем

let apply3D = (fun (imageInfo:ImageInfo) -> async {
    let bitmap = convertImageTo3D imageInfo.Image
    return { imageInfo with Image = bitmap } }) ← Функции для применения
                                                3D-эффекта к изображению

let saveImage = (fun (imageInfo:ImageInfo) -> async {
    printfn "Saving image %s" imageInfo.Name
    let destination = Path.Combine(imageInfo.Path, imageInfo.Name)
    imageInfo.Image.Save(destination) ← Сохранение изображения в локальной папке MyPicture
    return imageInfo.Name}) ← Компоновка ранее определенных асинхронных
                            функций с использованием монадических
                            асинхронных операторов return и bind

let loadandApply3dImage (imagePath:string) =
    Async.retn imagePath >>= loadImage >>= apply3D >>= saveImage ←

let loadandApply3dImageAgent = TamingAgent<string, string>(2,
    → loadandApply3dImage) ← Создание экземпляра TamingAgent, способного
                            выполнять конкурентно два задания, применяя
                            к каждому из них скомпонованную функцию loadandApply3dImage

loadandApply3dImageAgent.Subscribe(fun imageName -> printfn "Saved image %s
    → from subscriber" imageName) ← Подписка обработчика, который выполняется
                                    при получении уведомления о завершении задания

let transformImages() = ←
    let images = Directory.GetFiles(@".\Images")
    for image in images do
        loadandApply3dImageAgent.Ask(image)
        |> run (fun imageName ->
            printfn "Saved image %s - from reply back" imageName) ← Запуск процесса чтения файлов изображений
                                                и помещения нового задания в экземпляр
                                                TamingAgent loadandApply3dImageAgent

```

Три асинхронные функции — loadImage, apply3D и saveImage, — будучи скомпонованными с помощью асинхронного инфиксного F#-оператора связывания `>>=`, определенного в главе 9, образуют функцию loadandApply3dImage. Вот пример ее реализации:

```

let bind (operation:'a -> Async<'b>) (xAsync:Async<'a>) = async {
    let! x = xAsync
    return! operation x }

let (>>=) (item:Async<'a>) (operation:'a -> Async<'b>) =
    bind operation item

```

Затем экземпляр `loadAndApply3dImageAgent` объекта `TamingAgent` определяется путем передачи в конструктор аргумента со значением предела. Этот предел устанавливает степень параллелизма агента и передается в функцию `loadAndApply3dImage`, которая определяет поведение в вычислительных заданиях. Функция `Subscribe` регистрирует обратный вызов, выполняемый после завершения каждого задания. В этом примере он только выводит имя изображения для выполненного задания.

ПРИМЕЧАНИЕ

Пути к изображениям отправляются последовательно. `TamingAgent` является потокобезопасным, поэтому несколько потоков могут отправлять сообщения одновременно без каких-либо проблем.

Функция `loadImages()` считывает пути к изображениям из каталога `Images` и в цикле `for-each` отправляет значения в `loadAndApply3dImageAgent` `TamingAgent`. Функция `run` использует CPS для выполнения обратного вызова, когда результат уже вычислен и возвращается обратно.

13.12. Компоновка монадических функций

Предположим, что у нас есть функции, которые принимают простой тип и возвращают надтип, такие как `Task` или `Async`, и нужно скомпоновать данные функции. Вы можете подумать, что для этого достаточно получить первый результат, применить его ко второй функции, а затем повторить для всех функций. Но подобный процесс может оказаться довольно громоздким. Именно в таких случаях используется концепция компоновки функций. Напоминаю, что вы можете создать новую функцию из двух меньших. Обычно это работает, если типы входных и выходных данных у этих функций совпадают.

Указанное правило неприменимо к монадическим функциям, поскольку у них нет совпадающих типов входных и выходных данных. Например, монадические функции `Async` и `Task` нельзя скомпоновать, потому что `Task<T>` — не то же самое, что `T`.

Сигнатура монадического оператора `Bind` выглядит следующим образом:

```
Bind : (T -> Async<R>) -> Async<T> -> Async<R>
Bind : (T -> Task<R>) -> Task <T> -> Task <R>
```

Оператор `Bind` может передавать надтипы в функции, которые обрабатывают обернутые внутренние значения. Есть ли способ легко компоновать монадические функции?

Решение: объединение асинхронных операций с использованием оператора компоновки Клейсли. Компоновка монадических функций называется *компоновкой Клейсли*. В функциональном программировании она обычно представляется

с помощью инфиксного оператора `>=>`, который может быть создан с помощью монадического оператора `Bind`. Оператор `Kleisli`, по существу, предоставляет конструкцию компоновки монадических функций, которая вместо компоновки обычных функций, таких как `a -> b` и `b -> c`, используется для компоновки по `a -> M b` и `b -> M c`, где `M` — надтип.

Сигнатура оператора компоновки `Kleisli` для надтипов, таких как типы `Async` и `Task`, имеет вид:

```
Kleisli (>=>) : ('T -> Async<TR>) -> (TR -> Async<R>) -> T -> Async<R>
Kleisli (>=>) : ('T -> Task<TR>) -> (TR -> Task <R>) -> T -> Task <R>
```

С применением этого оператора две монадические функции могут быть скомпонованы непосредственно следующим образом:

```
(T -> Task<TR>) >=> (TR -> Task<R>)
(T -> Async<TR>) >=> (TR -> Async<R>)
```

Результатом является новая монадическая функция:

```
T -> Task<R>
T -> Async<R>
```

В следующем фрагменте кода показана реализация оператора `Kleisli` на C# на примере монадического оператора `Bind`. Оператор `Bind` (или `SelectMany`) для типа `Task` был описан в главе 7:

```
static Func<T, Task<U>> Kleisli<T, R, U>(Func<T, Task<R>> task1,
Func<R, Task<U>> task2) => async value => await task1(value).Bind(task2);
```

Эквивалентная функция на F# также может быть определена с использованием обычного инфиксного оператора `kleisli >=>`, в данном случае для типа `Async`:

```
let kleisli (f:'a -> Async<'b>) (g:'b -> Async<'c>) (x:'a) = (f x) >=> g
let (>=>) (f:'a -> Async<'b>) (g:'b -> Async<'c>) (x:'a) = (f x) >=> g
```

`Async bind` и инфиксный оператор `>=>` были описаны в главе 9. Напомню их реализацию:

```
let bind (operation:'a -> Async<'b>) (xAsync:Async<'a>) = async {
    let! x = xAsync
    return! operation x }

let (>=>)(item:Async<'a>) (operation:'a -> Async<'b>) = bind operation item
```

Где и как здесь может помочь оператор `Kleisli`? Рассмотрим случай с несколькими асинхронными операциями, которые мы хотим легко скомпоновать. Эти функции имеют следующую сигнатуру:

```
operationOne      : ('a -> Async<'b>)
operationTwo      : ('b -> Async<'c>)
operationThree    : ('c -> Async<'d>)
```

Концептуально скомпонованная функция должна выглядеть так:

```
('a -> Async<'b>) -> ('b -> Async<'c>) -> ('c -> Async<'d>)
```

На высоком уровне можно рассматривать подобную компоновку монадических функций как конвейер, где результат первой функции передается в следующую и так далее до последнего шага. В общем случае, когда речь идет о конвейере, можно представить два подхода: applicативный (`<*>`) и монадический (`>>=`). Поскольку в каждом следующем вызове нам нужен результат предыдущего вызова, лучше выбрать монадический стиль (`>>=`).

В этом примере мы будем применять `TamingAgent` из предыдущего примера. В `TamingAgent` есть метод-член `Ask`, сигнатура которого соответствует сценарию, где он принимает параметризованный аргумент '`T`' и возвращает тип `Async<'R>`. На этом этапе мы воспользуемся оператором `Kleisli` для компоновки множества типов `TamingAgent`, чтобы сформировать конвейер агентов, как показано на рис. 13.18. Результат каждого агента вычисляется независимо и передается в качестве входных данных в виде сообщения следующему агенту, и так до тех пор, пока последний узел цепочки не выполнит последний побочный эффект. Технология связывания и компоновки агентов может привести к созданию надежных конструкций и конкурентных систем. Когда агент возвращает (отправляет обратно) результат вызывающему объекту, он может быть скомпонован в конвейер агентов.

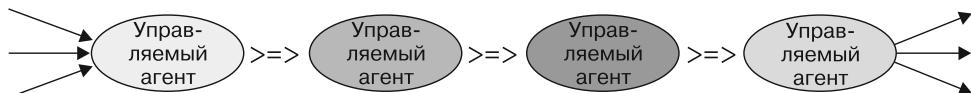


Рис. 13.18. Шаблон конвейерной обработки полезен, когда нужно обрабатывать данные в несколько этапов. Идея шаблона заключается в том, что входные данные отправляются первому агенту в конвейере. Основное преимущество шаблона конвейерной обработки заключается в том, что он обеспечивает простой способ обеспечить компромисс между чрезмерно последовательной обработкой (что может снизить производительность) и чрезмерно параллельной обработкой (что может иметь большие издержки)

В листинге 13.21 показан результат компоновки `TamingAgent`. Этот пример представляет собой доработанный листинг 13.20, в котором используется та же функция для загрузки, преобразования и сохранения изображения.

В данном примере программа использует `TamingAgent` для преобразования изображения, отличного от того, как это сделано в листинге 13.20. В более ранних примерах три функции, которые загружают, преобразуют и сохраняют изображение в локальной файловой системе, именно в таком порядке были скомпонованы, образуя новую функцию. Эта функция обрабатывается и применяется ко всем входящим сообщениям одним экземпляром типа `TamingAgent`. В данном приложении (см. листинг 13.21) для каждой выполняемой функции создается экземпляр `TamingAgent`, а затем агенты компонуются посредством внутреннего метода `Ask`, образуя конвейер. Асинхронная функция `Ask` обеспечивает ответ вызывающему объекту через `AsyncResult` после завершения задания. Компоновка агентов упрощается благодаря оператору `Kleisli`.

Листинг 13.21. TamingAgent с оператором Kleisli

```
let pipe limit operation job : Async<_> = ←  
    let agent = TamingAgent(limit, operation)  
    agent.Ask(job)  
  
    let loadImageAgent = pipe 2 loadImage  
    let apply3DEffectAgent = pipe 2 apply3D  
    let saveImageAgent = pipe 2 saveImage  
  
Создание экземпляра канала агента TamingAgent  
для каждой функции обработки изображений  
let pipeline = ←  
    loadImageAgent >=> apply3DEffectAgent >=> saveImageAgent ←  
  
let transformImages() = ←  
    let images = Directory.GetFiles(@"..\Images")  
    for image in images do  
        pipeline image  
    |> run (fun imageName -> printfn "Saved image %s" imageName)
```

Создание экземпляра типа TamingAgent и предоставление доступа к его асинхронному методу Ask, который обеспечивает ответ вызывающему объекту через AsyncReplyChannel после завершения выполнения задания

Объединение асинхронных операций, сгенерированных из функции канала, с использованием оператора Kleisli

Запуск процесса путем чтения файлов изображений и отправки нового задания в конвейер

Назначение функции pipe состоит в том, чтобы помочь создать экземпляр TamingAgent и предоставить функцию Ask, сигнатура которой 'a -> Async<'b> подобна монадическому оператору Bind, применяемому для компоновки с другими агентами.

После определения этих трех агентов, loadImageAgent, apply3DEffectAgent и saveImageAgent, с использованием вспомогательной функции pipe можно легко создать конвейер путем компоновки данных агентов посредством оператора Kleisli.

Резюме

- ❑ Конкурентный пул объектов следует использовать для освобождения памяти экземпляров одних и тех же объектов без блокировки, чтобы оптимизировать производительность программы. Количество этапов сборки мусора можно значительно сократить за счет применения пула объектов, что повышает скорость выполнения программы.
- ❑ Можно распараллелить набор зависимых задач с ограничениями по порядку выполнения. Этот процесс полезен, потому что он максимально увеличивает параллелизм при выполнении нескольких задач без учета зависимостей между ними.
- ❑ Можно координировать доступ нескольких потоков к разделяемым ресурсам для операций чтения/записи без блокировки, сохраняя порядок FIFO. Такая координация позволяет выполнять операции чтения одновременно, асинхронно (без блокировок) ожидая возможных операций записи. Подобный шаблон повышает производительность приложения благодаря применению параллелизма и снижению потребления ресурсов.
- ❑ Агрегатор событий действует подобно шаблону «посредник», где все события проходят через центральный агрегатор и могут потребляться в любом месте приложения. Rx позволяет реализовать агрегатор событий, который поддерживает многопоточность для конкурентной обработки нескольких событий.
- ❑ С помощью интерфейса `IScheduler` можно реализовать специальный Rx-планировщик, чтобы ограничить количество входящих событий и обеспечить точный контроль степени параллелизма. Кроме того, благодаря возможности явно задать уровень параллелизма внутренний пул потоков Rx-планировщика не страдает от простоев из-за расширения размера потоков, когда это необходимо.
- ❑ Даже без встроенной поддержки модели программирования CSP в .NET можно использовать либо F# `MailboxProcessor`, либо TDF для координации и балансировки полезной нагрузки между асинхронными операциями в неблокирующем синхронном стиле передачи сообщений.

Построение масштабируемого мобильного приложения методом конкурентного функционального программирования

В этой главе:

- разработка масштабируемых высокопроизводительных приложений;
- использование шаблона CQRS с уведомлениями WebSocket;
- отключение от контроллера ASP.NET Web API с помощью Rx;
- реализация шины сообщений.

Дочитав до этой главы, вы уже изучили и освоили методы конкурентного функционального программирования и шаблоны для построения высокопроизводительных и масштабируемых приложений. Данная глава является кульминацией книги, где рассматривается практическое применение этих методов. Здесь вы используете свои знания TPL-задач, асинхронного рабочего процесса, программирования методом передачи сообщений, а также реактивного программирования с реактивными расширениями для разработки полностью конкурентного приложения.

Приложение, которое мы создадим в этой главе, основано на мобильном интерфейсе, взаимодействующем с конечной точкой Web API для мониторинга фондового рынка в реальном времени. Приложение включает в себя возможность отправлять команды покупки и продажи акций и обслуживать эти команды на стороне сервера посредством долговременной асинхронной операции. Данная операция реактивно применяет команды относительно торговли, когда цены на акции достигают желаемой отметки.

Мы рассмотрим такие темы, как выбор архитектуры, и обсудим, почему функциональная парадигма хорошо подходит для реализации и серверной, и клиентской части при разработке масштабируемого и отзывчивого приложения. К концу главы вы узнаете, как создавать оптимальные конкурентные функциональные шаблоны и выбирать наиболее эффективные модели конкурентного программирования.

14.1. Практическое применение функционального программирования для серверной части приложения

Серверное приложение должно быть рассчитано на конкурентную обработку нескольких запросов. В общем случае обычные веб-приложения можно рассматривать как интенсивно параллельные, поскольку запросы полностью изолированы и легко выполняются независимо друг от друга. Чем мощнее сервер, на котором выполняется приложение, тем больше запросов он может обработать.

Программная логика современных крупномасштабных веб-приложений является конкурентной по своей природе. Кроме того, современные высокointерактивные веб-приложения и приложения реального времени, такие как многопользовательские браузерные игры, платформы для совместной работы и мобильные сервисы, представляют собой огромную проблему с точки зрения конкурентного программирования. В качестве строительных блоков для координации различных операций и обмена данными между различными конкурентными запросами, которые, скорее всего, выполняются параллельно, такие приложения используют мгновенные уведомления и асинхронный обмен сообщениями. В подобных случаях уже невозможно написать простое приложение с единственным последовательным потоком управления; вместо этого приходится планировать целостную синхронизацию между независимыми компонентами. Чем же может помочь функциональное программирование при создании серверного приложения?

В сентябре 2013 г. в Twitter была опубликована статья Мариуса Эриксена (Marius Eriksen) «Сервер как функция» (<https://monkey.org/~marius/funsrv.pdf>). Целью статьи была валидация архитектуры и модели программирования, принятой в Twitter для разработки высокомасштабируемого серверного программного обе-

спечения, когда системы демонстрируют высокую степень конкурентности и изменчивости среды. Вот цитата из этой статьи.

Мы представляем три абстракции, вокруг которых структурируем наше серверное программное обеспечение в Twitter. Они относятся к стилю функционального программирования — с акцентом на неизменяемости, компоновке функций первого класса и изоляции побочных эффектов — и их сочетание обеспечивает большой выигрыш в гибкости, простоте, удобстве обсуждения и надежности.

Поддержка конкурентного функционального программирования в .NET является ключевым свойством, благодаря которому .NET представляет собой отличный инструмент для программирования серверной части приложений. Это касается поддержки асинхронного выполнения операций в декларативном стиле с семантикой компоновки; кроме того, для разработки поточно-ориентированных компонентов можно использовать агенты. Основные технологии можно объединять, получая возможность декларативной обработки событий и эффективный параллелизм с применением TPL.

Функциональное программирование облегчает реализацию сервера без сохранения состояния (рис. 14.1), который очень ценен для обеспечения масштабируемости при построении больших веб-приложений, требующих конкурентной обработки огромного количества запросов, таких как социальные сети или сайты электронной коммерции. Считается, что программа работает без сохранения состояния, если операции (например, функции, методы и процедуры) не чувствительны к состоянию вычислений. Таким образом, все данные, задействуемые в операции, передаются другим операциям в качестве *входных данных* и все данные, задействуемые вызванными операциями, передаются ими в качестве *выходных данных*. В архитектуре без сохранения состояния данные приложения или пользователя никогда не сохраняются для последующих вычислительных нужд. Архитектура без сохранения состояния упрощает реализацию конкурентности, поскольку каждый этап приложения легко выполнить в другом потоке. Архитектура без сохранения состояния — это ключ, который позволяет идеально масштабировать приложение в соответствии с законом Амдала.

На практике программу без сохранения состояния можно легко распараллелить и распределить между компьютерами и процессами для масштабирования производительности. Вам не нужно знать, где выполняются вычисления, поскольку никакая часть программы не изменит какую-либо структуру данных, что позволяет избежать состояния гонки. Кроме того, вычисления могут выполняться в разных процессах или на разных компьютерах без ограничений для выполнения в конкретной среде.

Используя методы функционального программирования, можно создавать сложные, полностью асинхронные и адаптивные системы, которые автоматически масштабируются с применением одного и того же уровня абстракций, с одинаковой семантикой по всем измерениям масштабирования — от ядер процессора до центров обработки данных.

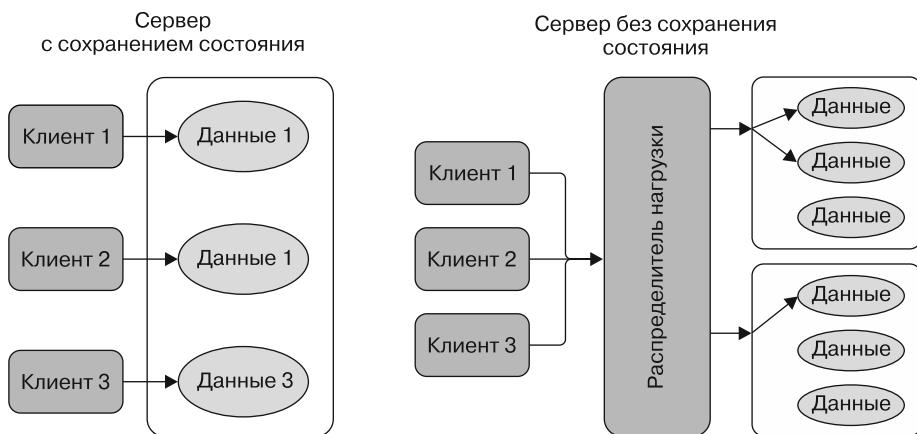


Рис. 14.1. Сравнение сервера с состоянием (с сохранением состояния) с сервером без состояния (без сохранения состояния). Сервер с сохранением состояния должен сохранять состояние между запросами, что ограничивает масштабируемость системы и требует больше ресурсов для выполнения. Сервер без сохранения состояния может автоматически масштабироваться, поскольку у него нет разделяемого состояния. Перед сервером без сохранения состояния может быть установлен распределитель нагрузки, распределяющий входящие запросы, которые могут быть направлены на любой компьютер, и не приходится беспокоиться о том, чтобы эти запросы были направлены на конкретный сервер

14.2. Как разработать высокопроизводительное приложение

При одновременной обработке сотен тысяч запросов в секунду в крупномасштабных системах требуется высокая степень конкурентности и эффективности при обработке операций ввода-вывода, а также синхронизация, чтобы обеспечить максимальную пропускную способность и использование процессоров в серверном программном обеспечении. Эффективность, безопасность и надежность являются первостепенными целями, которые традиционно входят в конфликт с модульностью, возможностью многократного применения и гибкостью кода. В функциональной парадигме уделяется особое внимание декларативному стилю программирования, который требует, чтобы асинхронные программы по своей структуре представляли собой набор компонентов, чьи зависимости от данных обеспечивались бы различными асинхронными комбинаторами.

ПРИМЕЧАНИЕ

Как обсуждалось в главе 8, асинхронные операции ввода-вывода должны выполняться параллельно, поскольку их масштабируемость может на порядок превосходить число доступных процессоров. Кроме того, чтобы корректно получить такую неограниченную возможность ресурса, асинхронные операции должны быть написаны в функциональном стиле, чтобы вместо изменений состояния в памяти оперировать неизменяемыми значениями.

При реализации программы необходимо заранее заложить в проект требования производительности. Производительность — это вопрос разработки программного обеспечения, который нельзя решать в последнюю очередь; она должна быть определена как явная цель с самого начала. Нет ничего *принципиально невозможного* в перепроектировании существующего приложения с нуля, но это намного дороже, чем правильно запроектировать все с самого начала.

14.2.1. Ноу-хэй: ACD

Итак, нам нужна система, способная гибко приспосабливаться к увеличению (или уменьшению) количества запросов с соответствующим ростом скорости при подключении новых ресурсов. Секретными компонентами для разработки и реализации такой системы являются асинхронность, кэширование и распределение (Asynchronicity, Caching, Distribution — ACD).

- ❑ *Асинхронность* означает операцию, которая завершится не сейчас, а в будущем. Асинхронность можно интерпретировать как элемент архитектуры — например, постановка задач в очередь, чтобы выполнить их позже и таким образом, например, выровнять нагрузку на процессор. Важно разделять операции, чтобы в критически важные для производительности моменты выполнялся минимальный объем работы. Аналогично можно использовать асинхронное программирование, чтобы запланировать выполнение запросов вочные часы.
- ❑ *Кэширование* имеет целью избежать повторного выполнения работы. Например, кэширование позволяет сохранить результаты ранее выполненной работы и использовать их впоследствии, не повторяя эту работу для получения тех же самых результатов. Обычно кэширование применяется для трудоемких операций, которые часто повторяются и результаты которых меняются достаточно редко.
- ❑ *Распределение* направлено на разделение запросов между несколькими системами, чтобы масштабировать обработку. Распределение проще реализовать в системе без сохранения состояния: чем меньше состояний хранится на сервере, тем легче распределять работу.

ПРИМЕЧАНИЕ

При разработке высокопроизводительных, масштабируемых и отказоустойчивых веб-приложений важно учитывать замечания, изложенные в работе «Ошибки распределенных вычислений» (www.rgoarchitects.com/Files/fallacies.pdf). Ее автор, Арnon Ротем-Гал-Оз (Arnon Rotem-Gal-Oz), исходит из предположения, что распределенные системы работают в безопасной, надежной, однородной сети с нулевой задержкой, бесконечной пропускной способностью, нулевыми транспортными издержками и неизменной топологией.

ACD является основным компонентом для написания масштабируемых и отзывчивых приложений, способных поддерживать высокую пропускную способность в условиях высокой рабочей нагрузки. Эта задача становится все более актуальной.

14.2.2. Другой асинхронный шаблон: постановка в очередь для отложенного выполнения

На этом этапе у вас уже должно составиться четкое представление о том, что собой представляет асинхронное программирование. Как вы помните, асинхронность означает, что отправленное на выполнение задание будет выполнено *в будущем*. Подобное может достигаться посредством одного из двух шаблонов. Первый из них основан на стиле продолжения (Continuation Passing Style, CPS) или обратных вызовах, описанных в главах 8 и 9. Второй шаблон основан на асинхронной передаче сообщений, описанных в главах 11 и 12. Как отмечалось в предыдущем разделе, асинхронность также может быть результатом (поведением) архитектуры приложения.

Шаблон, показанный на рис. 14.2, реализует асинхронные системы на уровне проектирования и предназначен для выравнивания рабочей нагрузки программы. Он отправляет операции или запросы на выполнение в службу, которая ставит задачи в очередь для выполнения в будущем. Этот сервис может работать на удаленном аппаратном устройстве, удаленном сервере в облаке или быть другим процессом на локальной машине. В последнем случае поток выполнения отправляет запрос в режиме «отправил и забыл» и освобождается для дальнейшей работы. Примером задачи, для которой используется такая архитектура, является планирование отправки сообщения по списку рассылки.

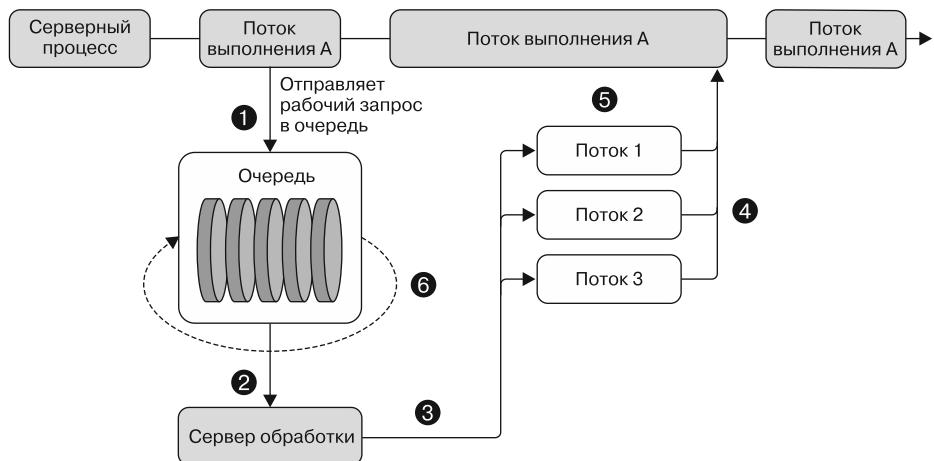


Рис. 14.2. Работа передается в очередь, удаленный обработчик получает сообщение и выполняет запрошенное действие в будущем

Когда операция завершается, сервис может отправить уведомление источнику (отправителю) запроса с подробной информацией о результате. На рис. 14.2 показаны следующие шесть этапов.

1. Поток выполнения отправляет задание или запрос сервису, который ставит его в очередь. Сервис принимает задачу и сохраняет ее для выполнения в будущем.

2. В какой-то момент сервис извлекает задачу из очереди и отправляет ее на обработку. Сервер обработки отвечает за планирование потока для выполнения операции.
3. Запланированный поток выполняет операцию; как правило, для каждой задачи используется свой поток.
4. В идеале, когда работа завершена, сервис уведомляет источник (отправителя) о том, что работа завершена.
5. Пока запрос обрабатывается в фоновом режиме, поток выполнения свободен и может выполнять другую работу.
6. Если что-то пойдет не так, задача будет перенесена (повторно поставлена в очередь) для последующего выполнения.

Сначала онлайн-компании, чтобы справиться с растущим объемом запросов, инвестировали в более мощное оборудование. Этот подход оказался дорогостоящим, учитывая сопутствующие расходы. В последние годы Twitter, Facebook, StackOverflow.com и другие компании доказали, что можно создать быструю, отзывчивую систему с меньшим количеством компьютеров, если использовать хорошую архитектуру программного обеспечения и правильные шаблоны, такие как ACD.

14.3. Правильный выбор конкурентной модели программирования

Повышение производительности программы с использованием конкурентности и параллелизма было в центре дискуссий и исследований на протяжении многих лет. Результатом этих исследований стало появление нескольких конкурентных моделей программирования, каждая из которых имеет свои сильные и слабые стороны. Общей чертой данных моделей является стремление обеспечить такой стиль работы и предложить такие характеристики, чтобы получить более быстрый код. Кроме этих конкурентных моделей программирования, компании разработали инструментарий для содействия подобному программированию: Microsoft создала TPL, а Intel внедрила Threading Building Blocks (TBB) для производства высококачественных и эффективных библиотек, помогающих профессиональным разработчикам создавать параллельные программы. Существует множество моделей конкурентного программирования, которые различаются по механизмам взаимодействия задач, степени детализации задач, гибкости, масштабируемости и модульности.

Получив многолетний опыт в создании масштабируемых систем, я убежден, что правильная модель программирования представляет собой сочетание моделей программирования, адаптированных для каждой части вашей системы. Вы можете выбрать модель актора для систем передачи сообщений и PLINQ для распараллеливания данных в вычислительных задачах в каждом из узлов, а при загрузке данных для анализа перед вычислением — неблокирующую асинхронную обработку ввода-вывода. Главное — подобрать правильный инструмент или сочетание инструментов для работы.

В следующем списке представлен мой выбор конкурентной технологии для типичных случаев.

- ❑ Если есть чистые функции и операции с четко определенными управляющими зависимостями, где данные могут быть разделены или могут обрабатываться рекурсивно, то стоит рассмотреть возможность использования TPL для построения динамических параллельных вычислений в форме шаблона Fork/Join или «разделяй и властвуй».
- ❑ Если параллельные вычисления требуют сохранения порядка операций или алгоритм зависит от логического потока, то рассмотрите возможность использования DAG либо с примитивами TPL-задач, либо с агентной моделью (см. главу 13).
- ❑ В случае последовательного цикла с независимыми итерациями и отсутствием зависимостей между шагами параллельный цикл TPL может повысить производительность за счет вычисления данных в одновременных операциях, выполняемых в отдельных задачах.
- ❑ При обработке данных в виде оператора комбинирования, например, путем фильтрации и агрегирования входных элементов, хорошим решением для ускорения вычислений представляется Parallel LINQ (PLINQ). Попробуйте использовать параллельный редуктор (также называемый сверткой или агрегатором), такой как параллельная функция `Aggregator`, для объединения результатов и применения шаблона `MapReduce`.
- ❑ Если приложение предназначено для выполнения последовательности операций в виде рабочего процесса и если порядок выполнения набора задач важен и должен соблюдаться, используйте шаблон «конвейер» или «поставщик — потребитель»: это отличные решения, позволяющие без особых усилий распараллелить операции. Такие шаблоны легко реализовать, используя TPL Dataflow или `MailboxProcessor` в F#.

При создании детерминированных параллельных программ имейте в виду, что их можно строить снизу вверх, составляя детерминированные параллельные шаблоны для вычислений и доступа к данным. Рекомендуется выбирать параллельные шаблоны так, чтобы они обеспечивали контроль за детализацией их выполнения, расширяя и сокращая степень параллелизма в зависимости от доступных ресурсов.

В этом разделе мы создадим приложение, имитирующее онлайн-сервис фондового рынка (рис. 14.3). Данный сервис периодически обновляет цены на акции и передает обновления всем подключенными клиентам в реальном времени. Это высокопроизводительное приложение способно обрабатывать огромное количество одновременных подключений внутри веб-сервера.

Клиент представляет собой мобильное приложение iOS для iPhone, созданное с применением Xamarin и Xamarin.Forms. В мобильном клиенте значения изменяются в реальном времени в ответ на уведомления, поступающие от сервера. Пользователи приложения могут управлять своим портфелем акций, размещая заказы на покупку и/или продажу определенной акции, когда она достигает заранее определенной цены. Кроме мобильного приложения, в исходном коде к книге представлена версия клиентской программы WPF.

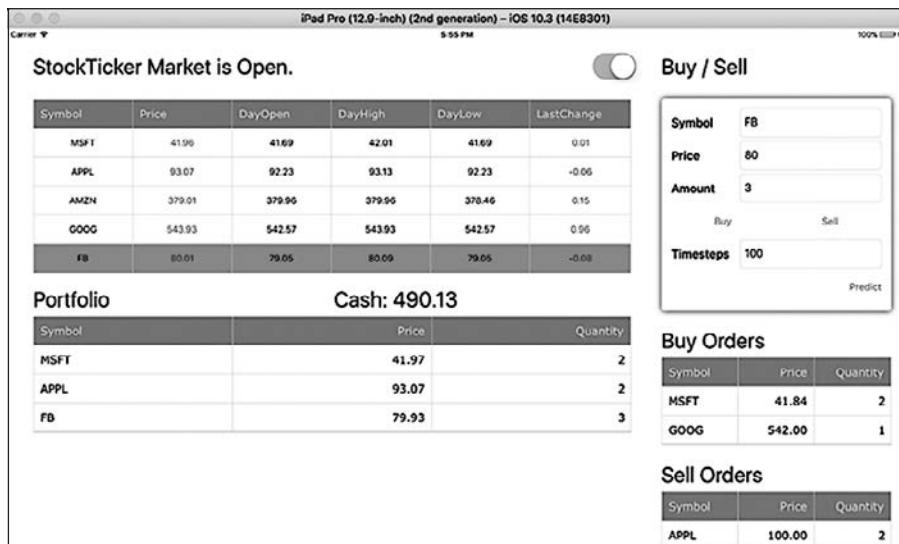


Рис. 14.3. Пример пользовательского интерфейса мобильного приложения для работы с фондовым рынком (Apple iPad). На левой панели представлены обновления цен на акции в режиме реального времени. Правая панель используется для управления портфелем и размещения заказов на покупку и продажу акций

ПРИМЕЧАНИЕ

Для запуска мобильного проекта установите Xamarin (www.xamarin.com). Подробные инструкции вы найдете в онлайн-документации.

Xamarin и Xamarin.Forms

Xamarin — это фреймворк, предназначенный для быстрой разработки кроссплатформенных пользовательских интерфейсов. Xamarin предоставляет абстракцию пользовательского интерфейса, который отображается с применением внутренних элементов управления на iOS и Android, Windows и Windows Phone. Это означает, что приложения получают доступ к значительной части кода пользовательского интерфейса данных ОС, сохраняя внешний вид и поведение соответствующей платформы.

Xamarin.Forms — кроссплатформенная абстракция инструментария пользовательского интерфейса с собственной поддержкой, с помощью которой можно легко разрабатывать пользовательские интерфейсы для Android, iOS, Windows и Windows Phone. Пользовательские интерфейсы визуализируются с применением собственных элементов управления операционной платформы, что позволяет приложениям Xamarin.Forms сохранять внешний вид, соответствующий данной платформе.

Xamarin и Xamarin.Forms — это две отдельные большие темы, которые не имеют отношения к контексту настоящей книги. Для получения дополнительной информации о них обратитесь на сайт www.xamarin.com/forms.

Создавая свое приложение, мы внимательно изучим, как применить к такому приложению функциональную конкурентность. Мы объединим свои знания с приемами и шаблонами конкурентного функционального программирования, представленными в предыдущих главах. Для обработки параллельных запросов мы будем использовать шаблон разделения команд и запросов (Command and Query Responsibility Segregation, CQRS), Rx и асинхронное программирование. Мы задействуем порождение событий, основанное на функциональном постоянстве (хранилище событий, задействующее модель агентного программирования), и многое другое. Я опишу данные шаблоны позже, на примере соответствующей части приложения.

Веб-серверное приложение — это ASP.NET Web API с использованием Rx для передачи сообщений от входящих запросов контроллера к другим компонентам приложения. Указанные компоненты реализованы с применением агентов (`MailboxProcessor`), которые порождают новые агенты для каждого установленного и активного пользовательского соединения. Таким образом, приложение может поддерживаться в изолированном состоянии для каждого пользователя и обеспечивать простое масштабирование.

Мобильное приложение написано на C#, что в общем случае в сочетании с моделью ТАР и Rx является хорошим выбором для разработки клиентской части. Для кода веб-сервера вместо C# мы будем применять F#; но вы найдете версию программы на C# в исходном коде к этой книге. Основной причиной выбора F# для серверной части кода является неизменяемость в качестве конструкции по умолчанию, которая идеально подходит для архитектуры без сохранения состояния, используемой в данном примере приложения для фондового рынка. Кроме того, встроенная поддержка модели агентного программирования с помощью `MailboxProcessor` позволяет легко инкапсулировать и поддерживать состояние потокобезопасным способом. Более того, как вы вскоре увидите, F# представляет более лаконичное решение, по сравнению с C#, для реализации шаблона CQRS, делая код более явным и фиксируя то, что происходит в функции, без скрытых побочных эффектов.

В приложении используется ASP.NET SignalR для реализации функции широковещательной рассылки сервера для обновлений данных в реальном времени. *Широковещательная рассылка сервера* — это коммуникация, инициируемая сервером, с отправкой данных клиентам.

Коммуникация в реальном времени посредством SignalR. Библиотека Microsoft SignalR предоставляет абстракцию некоторых транспортных протоколов, необходимых для передачи содержимого с сервера подключенным клиентам в режиме реального времени. Это означает, что серверы и их клиенты могут передавать данные в реальном времени в обоих направлениях, устанавливая канал двунаправленной коммуникации. В SignalR применяется несколько транспортных протоколов с автоматическим выбором наилучшего из доступных для данного клиента и сервера.

Соединение начинается как HTTP, а затем, если есть такая возможность, преобразуется в WebSocket. WebSocket является идеальным транспортным протоколом для SignalR, поскольку наиболее эффективно использует память сервера, имеет самую низкую задержку и наибольшее количество внутренних функций. Если эти требования не выполняются, SignalR откатывается, пытаясь применять для установки соединений другие транспортные протоколы, такие как длинный опрос Ajax. SignalR всегда пытается использовать наиболее эффективный транспортный протокол и продолжает попытки до тех пор, пока не выберет наилучший из совместимых с контекстом. Решение принимается автоматически на начальном этапе коммуникации между клиентом и сервером, который называется *согласованием*.

14.4. Торговля акциями в реальном времени: архитектура высокого уровня для фондового рынка

Прежде чем углубиться в реализацию кода приложения для работы с фондовым рынком, кратко рассмотрим архитектуру приложения, чтобы лучше понять, что мы разрабатываем. Архитектура основана на шаблоне CQRS, который обеспечивает разделение между уровнями предметной области и применение моделей чтения и записи.

ПРИМЕЧАНИЕ

Код, используемый для реализации серверной части приложения, описанного в этой главе, написан на языке F#, но его полную версию на C# вы найдете в исходном коде к книге. Принципы, описанные в последующих разделах, применимы как к C#, так и к F#.

Ключевым принципом CQRS выступает отделение *команд*, которые являются операциями, вызывающими изменение состояния (побочные эффекты в системе), от запросов, которые предоставляют данные только для чтения, без изменения состояния какого-либо объекта, как показано на рис. 14.4. Шаблоны CQRS также основаны на *разделении ответственности*, что важно во всех аспектах разработки программного обеспечения и особенно для решений, основанных на архитектуре на базе сообщений.

В число преимуществ использования шаблона CQRS входит возможность управлять более сложными бизнес-функциями, упрощая уменьшение масштабирования системы, возможность писать оптимизированные запросы и внедрение механизма кэширования за счет возможностей API чтения. Применение CQRS для систем с большим несоответствием между нагрузкой записи и чтения позволяет резко

масштабировать часть приложения, относящуюся к чтению. На рис. 14.5 показана схема веб-серверного приложения для работы с фондовым рынком, построенного на основе шаблона CQRS.

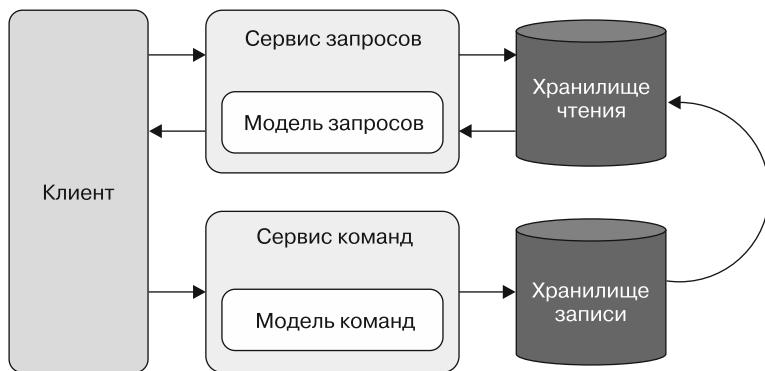


Рис. 14.4. Шаблон CQRS обеспечивает разделение между уровнями предметной области и использование моделей чтения и записи. Чтобы получить максимальную производительность операций чтения, в приложении можно создать отдельное хранилище данных, специально оптимизированное для запросов. Часто в роли такого хранилища может выступать база данных NoSQL. Синхронизация между экземплярами хранилища для чтения и записи выполняется асинхронно в фоновом режиме и может занять некоторое время. Считается, что такие хранилища данных в итоге являются согласованными

Эту функциональную архитектуру можно рассматривать как архитектуру потока данных. Внутри приложения данные проходят через различные стадии. На каждом этапе данные фильтруются, обогащаются, преобразуются, буферизуются, рассылаются, сохраняются или обрабатываются любым другим способом. На рис. 14.5 показаны следующие этапы потока.

- Пользователь отправляет запрос на сервер. Запрос имеет форму команды для размещения заказа на покупку или продажу данной акции. Контроллер ASP.NET Web API реализует интерфейс `IObservable` с его методом `Subscribe`, который регистрирует наблюдатели, прослушивающие входящие запросы. Благодаря такой архитектуре контроллер превращается в издателя сообщений, отправляющего команды подписчикам. В данном примере существует только один подписчик — агент (`MailboxProcessor`), который действует как шина сообщений. Но может быть любое количество подписчиков, например, для регистрации и измерения показателей производительности.
- Входящие запросы действий Web API проверяются и преобразуются в системные команды. Эти команды заключаются в оболочку вместе с дополнительными метаданными, такими как метка времени и уникальный идентификатор. Последний, обычно представленный в виде идентификатора соединения SignalR, впоследствии применяется для хранения событий, агрегированных по уникальному идентификатору, определяющему пользователя. Это упрощает

направление и выполнение потенциальных запросов и воспроизведение истории событий.

- Команда передается в обработчик команд, который передает сообщение подписчикам по шине сообщений. Подписчиком обработчика команд является **StockTicker** — объект для поддержания состояния, реализованный на основе агента. Как следует из названия, это объект биржевого кода.

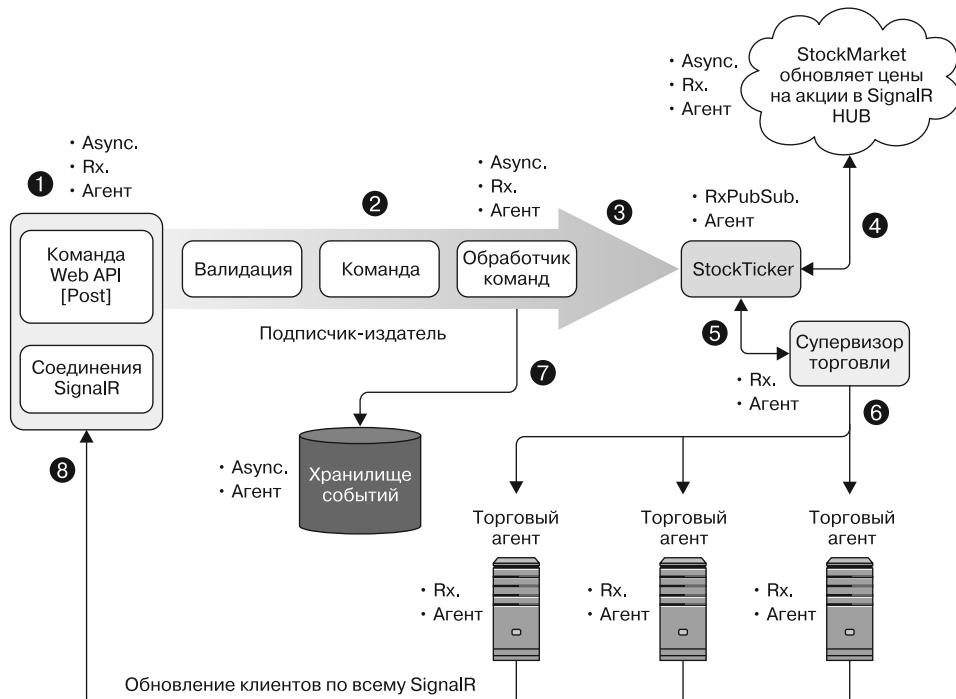


Рис. 14.5. Типичная модель веб-серверного приложения для работы с фондовым рынком, основанная на шаблоне CQRS. Команды (Write) передаются по конвейеру приложения для выполнения торговых операций в одном канале, а запросы (Read) выполняются в другом канале. В этой схеме запросы (Read) автоматически выполняются системой в форме серверных уведомлений, рассылаемых клиентам через соединения SignalR. SignalR можно представить как канал, который позволяет клиенту получать уведомления, генерируемые сервером. На рисунке в выносках указаны технологии, используемые для реализации указанного компонента

- Типы **StockTicker** и **StockMarket** установили двунаправленную коммуникацию, которая используется для уведомления об изменении курсов акций. В данном случае Rx применяется для постоянного обновления цен на акции случайным образом. Эти цены отправляются в **StockMarket** и потом передаются в **StockTicker**. Затем концентратор SignalR рассыпает обновления всем активным клиентским соединениям.

5. `StockTicker` отправляет уведомление объекту `TradingCoordinator`. Это агент, который ведет список активных пользователей. Когда пользователь регистрируется в приложении, `TradingCoordinator` получает уведомление и, если это новый пользователь, порождает новый агент. Сервер приложений создает новый экземпляр агента для каждого входящего запроса, который соответствует новому клиентскому соединению. Объект `TradingCoordinator` реализует интерфейс `IObservable`, применяемый для установления реактивного шаблона «издатель — подписчик» с Rx для отправки сообщений зарегистрированным наблюдателям — объектам `TradingAgent`.
6. `TradingCoordinator` получает команды для торговых операций и отправляет их ассоциированному агенту (пользователю), проверяя уникальный идентификатор клиентского соединения. Тип `TradingAgent` — это агент, который реализует интерфейс `IObserver`, зарегистрированный для получения уведомлений от `IObservable TradingCoordinator`. Для каждого пользователя существует свой `TradingAgent`, главная цель которого — поддерживать состояние портфеля и торговых заказов на покупку и продажу акций. Этот объект постоянно получает обновления данных фондового рынка, чтобы проверить, удовлетворяет ли состояние рынка условиям, определенным в каком-либо из заказов в качестве критерии выполнения торговой операции.
7. Для хранения торговых событий в приложении реализована система порождения событий. События объединяются в группы по пользователю и упорядочены по метке времени. Это при необходимости позволяет воспроизвести историю торговых операций каждого пользователя.
8. При запуске сделки `TradingAgent` уведомляет клиентское мобильное приложение через `SignalR`. Назначение приложения — дать клиенту возможность размещать заказы на торговые операции и асинхронно ожидать уведомления о завершении каждой операции.

Схема приложения, представленная на рис. 14.5, основана на шаблоне CQRS с четким разделением чтения и записи. Интересно отметить, что уведомления, рассылаемые в реальном времени, реализованы на стороне запроса (чтения), поэтому пользователю не нужно отправлять запрос на получение обновлений.

Оболочка

Как правило, рекомендуется заключать сообщения в оболочку, которая может нести дополнительную информацию о сообщении, что удобно для реализации системы передачи сообщений. Обычно наиболее важными элементами этой дополнительной информации являются уникальный идентификатор и метка времени, показывающая, когда было создано сообщение. Идентификаторы сообщений ценные тем, что позволяют обнаруживать повторы и сделать систему идемпотентной.

Возвращаясь к схеме шаблона CQRS, показанной на рис. 14.4, которая повторяет изображение на рис. 14.6, можно заметить, что существуют два отдельных хранилища: для чтения и для записи. Разрабатывая подобные разделенные хранилища с использованием шаблона CQRS, рекомендуется обеспечить максимальную производительность операций чтения. В случае двух отдельных хранилищ сторона записи должна обновлять сторону чтения. Эта синхронизация выполняется асинхронно в фоновом режиме и может занять некоторое время, поэтому считается, что хранилище считанных данных в итоге будет согласованным.

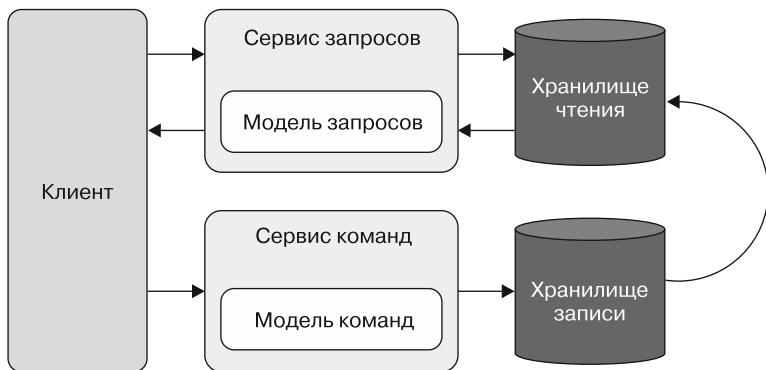


Рис. 14.6. Шаблон CQRS

Итоговая согласованность и теорема о согласованности, доступности и разделении (CAP)

Теорема CAP (consistency, availability and partition — согласованность, доступность и разделение) утверждает, что постоянное состояние в распределенных системах трудно реализовать корректно. Она говорит о том, что для распределенных систем с внутренней корреляцией существует три различных и желательных свойства, но любая реальная система может иметь не более двух из этих свойств для любых общих данных.

- *Согласованность* — свойство, которое означает согласованное представление данных во всех узлах распределенной системы; система гарантирует, что операции записи имеют атомарную природу, а обновления распространяются одновременно на все узлы и приводят к одинаковым результатам.
- *Доступность* — это свойство запроса, при котором система рано или поздно, даже в случае сбоев, отвечает на каждый запрос за разумное время.
- Устойчивость к *разделению* означает, что система устойчива к потерям сообщений между узлами. Разделение — это произвольное разбиение между узлами системы, приводящее к полной потере сообщений между узлами.

Итоговая согласованность — это модель согласованности, используемая в распределенных вычислениях для достижения высокой доступности. Такая модель гарантирует, что в итоге все обращения, адресованные данному элементу, вернут последнее обновленное значение. В приложении для фондового рынка итоговая согласованность предоставляется системой автоматически. Пользователи будут получать обновления и последние значения посредством уведомлений в реальном времени в те моменты, когда данные будут изменяться. Подобное возможно благодаря двунаправленной коммуникации SignalR между сервером и клиентами. Это удобный механизм, позволяющий пользователям не запрашивать обновления — сервер предоставляет их автоматически.

14.5. Главные элементы приложения для фондового рынка

Мы все еще не изучили несколько важных элементов для приложения для фондового рынка, поскольку предполагается, что вы уже сталкивались с этими темами. Мы кратко рассмотрим данные вопросы, и я дам ссылки на источники, с помощью которых вы сможете продолжить свое обучение по мере необходимости.

Первым существенным элементом является F#. Если вы не очень хорошо знакомы с F#, то обратитесь к приложению Б за более подробной информацией и справочными материалами, которые могут вам пригодиться.

Серверное приложение основано на интерфейсе ASP.NET Web API, что потребует от вас знания этой технологии. Для клиентской стороны в мобильном приложении использованы Xamarin и Xamarin.Forms с шаблоном «*модель — представление — модель представления*» (Model — View — ViewModel, MVVM) для связывания данных; но какие-либо особые знания об этих фреймворках вам не потребуются.

Шаблон MVVM

Шаблон MVVM применяется на всех платформах XAML. Его назначение — обеспечить четкое разделение областей ответственности между элементами управления пользовательского интерфейса и их логикой. Существует три основных компонента шаблона MVVM: *модель* (Model, бизнес-правило, доступ к данным, классы модели), *представление* (View, расширяемый язык разметки пользовательского интерфейса приложения — UI Extensible Application Markup Language, XAML) и *модель представления* (ViewModel, агент, или посредник, между представлением и моделью). Каждый элемент выполняет свою отдельную роль. Модель представления действует как интерфейс между моделью и представлением. Модель представления обеспечивает связывание данных между данными представления и модели, а также обрабатывает все действия пользовательского интерфейса посредством команд.

Представление связывает свое управляющее значение со свойствами модели представления, которая, в свою очередь, предоставляет данные, содержащиеся в объектах модели.

В оставшейся части этой главы мы будем применять следующие средства:

- реактивные расширения .NET;
- Task Parallel Library;
- MailboxProcessor в F#;
- асинхронные рабочие процессы.

Понятия, применяемые в следующих примерах кода, актуальны для всех языков программирования .NET.

14.6. Пишем код приложения для торговли на фондовом рынке

В этом разделе рассматриваются примеры кода, позволяющие реализовать мобильное приложение для работы на фондовом рынке в реальном времени с возможностями торговли. Части программы, которые не имеют отношения или не представляют особой важности для целей данной главы, намеренно опущены. Но вы найдете полную функциональную реализацию приложения в исходном коде к книге.

Начнем с серверного контроллера Web API, куда клиентское мобильное приложение отправляет запросы на выполнение торговых операций.

ПРИМЕЧАНИЕ

В исходный код, прилагаемый к этой книге, также входит реализация клиентской части приложения на WPF.

Обратите внимание, что контроллер реализует часть записи шаблона CQRS; на самом деле действия выполняются только через HTTP POST, как показано в листинге 14.1 (код, на который следует обратить внимание, выделен жирным шрифтом).

Контроллер Web API `TradingController` реализует действия продажи (`PostSell`) и покупки (`PostBuy`). Оба эти действия одинаково реализованы в коде, но имеют разное назначение. В листинге представлено только одно из них, чтобы избежать повторов.

Каждый элемент управления действием построен на основе двух основных функций — валидации и публикации. Функция `tradingValidation` отвечает за валидацию сообщений для каждого соединения, поскольку сообщения поступают от клиента. Функция `publish` отвечает за публикацию сообщений для элементов управления подписчиками с целью основной обработки.

Листинг 14.1. Контроллер для работы с фондовым рынком на основе Web API

```
[<RoutePrefix("api/trading")>]
type TradingController() =
    inherit ApiController()

    let subject = new Subject<CommandWrapper>()

    let publish connectionId cmd =
        match cmd with
        | Result.Ok(cmd) ->
            CommandWrapper.Create connectionId cmd
        | Result.Error(e) -> subject.OnError(exn e)

    member this.PostSell([<FromBody>] tr : TradingRequest) = async {
        let connectionId = tr.ConnectionID
        return
            { Symbol = tr.Symbol.ToUpper()
              Quantity = tr.Quantity
              Price = tr.Price
              Trading = TradingType.Sell }
        |> tradingValidation
        |> publish connectionId
        |> toResponse this.Request
    }|> Async.StartAsTask

    interface IObservable<CommandWrapper> with
        member this.Subscribe observer = subject.Subscribe observer

    override this.Dispose disposing =
        if disposing then subject.Dispose()
        base.Dispose disposing
```

Контроллер использует экземпляр Subject, чтобы реализовать поведение наблюдаемого объекта для публикации команд, адресованных зарегистрированному наблюдателю

Валидация
команды с применением subject.OnNext типа Result

Публикация команды с использованием Rx

Создание оболочки для заданной команды, чтобы добавить к типу метаданные

Применение вспомогательной функции для действий контроллера по доставке ответа HTTP

Показывает текущий идентификатор соединения из контекста SignalR

Валидация с использованием функциональной компоновки

Публикация команды с применением Rx

Запуск действия как Task, чтобы действие выполнялось асинхронно

Удаление Subject; это важно, так как позволяет освободить ресурсы

Контроллер использует экземпляр Subject, чтобы реализовать поведение наблюдаемого объекта для публикации команд, адресованных зарегистрированному наблюдателю

Действие `PostSell` выполняет валидацию входящего запроса посредством функции `tradingValidation`, которая возвращает `Result.OK` или `Result.Error`, в зависимости от валидности входных данных. Затем результат функции валидации обертывается в объект команды с помощью функции `CommandWrapper.Create` и публикуется у подписанных наблюдателей `subject.OnNext`.

`TradingController` использует экземпляр типа `Subject` из библиотеки Rx в качестве наблюдаемого объекта путем реализации интерфейса `IObservable`. Таким образом, этот контроллер слабо связан и ведет себя как шаблон «издатель — подписчик», отправляя команды зарегистрированным наблюдателям. Регистрация этого контроллера как `Observable` подключается к фреймворку Web API с помощью класса, реализующего `IHttpControllerActivator`, как показано в листинге 14.2 (код, на который следует обратить внимание, выделен жирным шрифтом).

Листинг 14.2. Регистрация контроллера Web API как Observable

```
type ControlActivatorPublisher(requestObserver:IObserver<CommandWrapper>) =
    interface IHttpControllerActivator with
        member this.Create(request, controllerDescriptor, controllerType) =
            if controllerType = typeof<TradingController> then
                let obsController =
                    let tradingCtrl = new TradingController()
                    tradingCtrl
                |> Observable.subscribeObserver requestObserver
                |> request.RegisterForDispose
                tradingCtrl
            obsController :> IHttpController
            else raise (ArgumentException("Unknown controller type requested"))
```

Если запрашиваемый тип контроллера
соответствует `TradingController`, то создается
новый экземпляр и регистрируется как `Observable`

Интерфейсы для подключения
конструктора или активатора
нового контроллера
к платформе Web API

Тип `ControlActivatorPublisher` реализует интерфейс `IHttpControllerActivator`, который внедряет в фреймворк Web API специальный активатор контроллера. В данном случае, если запрос соответствует типу `TradingController`, `ControlActivatorPublisher` преобразует контроллер в издатель `Observable`, а затем регистрирует контроллер в диспетчере команд. Наблюдатель `tradingRequestObserver`, переданный в конструктор `CompositionRoot`, используется как подписка для контроллера `TradingController`, который теперь может отправлять сообщения о действиях подписчикам в реактивном, разделенном стиле.

В итоге субзначение подписанного наблюдателя `requestObserver` представляет собой подписку и должно быть зарегистрировано для удаления вместе с экземпляром `TradingController` `tradingCtrl`, используя метод `request.RegisterForDispose`.

В листинге 14.3 представлен следующий этап — подписчик наблюдаемого контроллера `TradingController`.

Функция `Startup` выполняется, когда веб-приложение начинает применять параметры конфигурации. Именно здесь класс `CompositionRoot` (определенный

в листинге 14.2) заменяет стандартный `IHttpControllerActivator` новым экземпляром. Тип подписчика, передаваемый в конструктор `ControlActivatorPublisher`, является наблюдателем, который отправляет сообщения, поступающие от действий `TradingController`, в экземпляр агента `MailboxProcessor`. Издатель `TradingController` отправляет сообщения через метод `OnNext` интерфейса наблюдателя всем подписчикам, в данном случае агенту, который зависит только от реализации `IObserver` и, следовательно, уменьшает количество зависимостей.

Листинг 14.3. Настройка концентратора SignalR и шины агентских сообщений

```

type Startup() =
    let agent = new Agent<CommandWrapper>(fun inbox ->
        let rec loop () = async {
            let! (cmd:CommandWrapper) = inbox.Receive()
            do! cmd |> AsyncHandle
            return! loop() }
        loop())
    do agent.Start()

    Агент асинхронно обрабатывает
    полученные команды, публикуя их
    через обработчик команд AsyncHandle

    member this.Configuration(builder : IAppBuilder) =
        let config =
            let config = new HttpConfiguration()
            config.MapHttpAttributeRoutes()
            config.Services.Replace(typeof< IHttpControllerActivator >,
                ControlActivatorPublisher(Observer.Create(fun x ->
                    agent.Post(x))))
        configSignalR =
            new HubConfiguration(EnableDetailedErrors = true)

        Owin.CorsExtensions.UseCors(builder, Cors.CorsOptions.AllowAll)
        builder.MapSignalR(configSignalR) |> ignore
        builder.UseWebApi(config) |> ignore
    
```

Метод `agent.Post` из `MailboxProcessor` публикует сообщение, обернутое в тип `Command`, используя Rx. Обратите внимание, что сам контроллер реализует интерфейс `IObservable`, поэтому его можно представить как конечную точку сообщения, оболочку команды и издателя.

Агент-подписчик `MailboxProcessor` асинхронно обрабатывает входящие сообщения, подобно шине сообщений, но на меньшем и более специализированном уровне (рис. 14.7). Шина сообщений обеспечивает ряд преимуществ, от масштабируемости до естественной разделенности системы и межплатформенного взаимодействия. Архитектуры на основе сообщений, использующие шину сообщений, концен-

трируются на общих контрактах и передаче сообщений. Остальная часть метода конфигурации включает в себя концентраторы SignalR в приложении по всему предоставляемому `IAppBuilder`.



Рис. 14.7. Команда и обработчик команд, реализованные в листинге 14.4

В листинге 14.4 показана реализация функции `AsyncHandle`, которая обрабатывает сообщения агента в виде команд CQRS.

`RetryPublish` является экземпляром специализированного вычислительного выражения `RetryAsyncBuilder`, описанного в листинге 9.4. Это вычислительное выражение предназначено для асинхронного выполнения операций; если что-то пойдет не так, оно повторяет вычисления с определенной задержкой. `AsyncHandle` — обработчик команд, отвечающий за выполнение поведения команды из предметной области. Команды представлены в виде торговых операций покупки или продажи акций. В общем случае команды — это директивы для выполнения действий (поведений) из предметной области.

Назначение `AsyncHandle` — публиковать команды, полученные от экземпляра `TradingCoordinator`, выполняя следующий этап конвейера приложения в стиле передачи сообщений. Команда — это сообщение, полученное агентом `MailboxProcessor`, определенным при запуске приложения (см. листинг 14.3).

Модель программирования на основе сообщений приводит к архитектуре, управляемой событиями. В системе, управляемой сообщениями, получатели ожидают прибытия сообщений и реагируют на них; в противном случае они неактивны. В системе уведомлений в случае управления событиями слушатели подключаются к источникам событий и вызываются при отправке события.

Листинг 14.4. Обработчик команд с логикой асинхронных повторов

```

Экземпляр специализированного
вычислительного выражения RetryAsyncBuilder,
описанного в листинге 9.4.

module CommandHandler =
    let retryPublish = RetryAsyncBuilder(10, 250)

    let tradingCoordinator = TradingCoordinator.Instance()
    let Storage = new EventStorage()

    let AsyncHandle (commandWrapper:CommandWrapper) =
        let connectionId = commandWrapper.ConnectionId

        retryPublish {
            tradingCoordinator.PublishCommand(
                PublishCommand(connectionId, commandWrapper))
            let event =
                let cmd = commandWrapper.Command
                match cmd with
                | BuyStockCommand(connId, trading) ->
                    StocksBuyedEvent(commandWrapper.Id, trading)
                | SellStockCommand(connId, trading) ->
                    StocksSoldEvent(commandWrapper.Id, trading)
            let eventDescriptor = Event.Create(commandWrapper.Id, event)
            Storage.SaveEvent(Guid(connectionId)) eventDescriptor
        }
    }

Публикация команды
и указанного
пользователем ID
на фондовом рынке.
ID определяется
уникальным
идентификатором
соединения SignalR

```

УстановкаRetryAsyncBuilder для публикации сообщений, адресованных TradingAgent, который представляет активных клиентов

Экземпляр EventStorage для сохранения событий с целью реализации EventSourcing

Обработчик команд, который выполняет поведение предметной области. В данном случае он асинхронно публикует команду с семантикой повторения

Преобразование команды в тип события методом сопоставления с шаблоном

Сохранение события в хранилище событий

Архитектура, управляемая событиями

Управляемая событиями архитектура (Event-Driven Architecture, EDA) — это стиль разработки приложений, основанный на фундаментальных аспектах уведомлений о событиях и призванный упростить мгновенное распространение информации и реактивное выполнение бизнес-процессов. В приложении, основанном на EDA, информация распространяется в реальном времени в сильно распределенной среде, что позволяет различным компонентам приложения, получающим уведомления, проактивно реагировать на бизнес-действия. EDA обеспечивает низкую задержку и высокую реактивность системы. Различие между системами, управляемыми событиями и управляемыми сообщениями, состоит в том, что системы, управляемые событиями, сконцентрированы на адресуемых источниках событий, а системы, управляемые сообщениями, — на адресуемых получателях.

Обработчик `AsyncHandle` также отвечает за преобразование каждой полученной команды в тип `Event`, который затем сохраняется в хранилище событий (рис. 14.8). Хранение событий является частью реализации стратегии порождения событий для сохранения текущего состояния приложения (листинг 14.5).

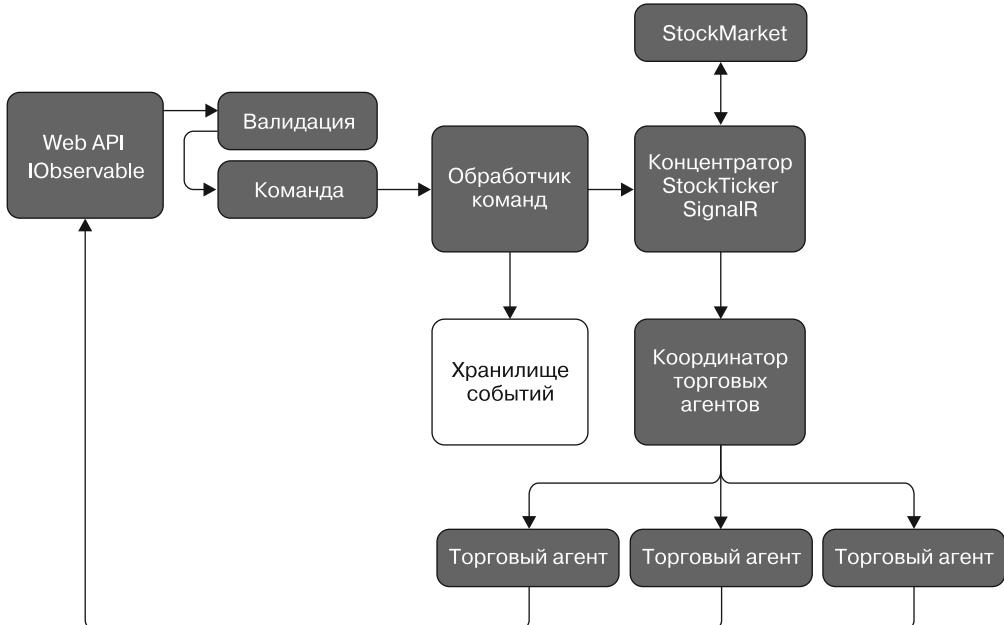


Рис. 14.8. Хранение событий, реализованное в листинге 14.5

Листинг 14.5. Реализация EventBus с использованием агента

```

module EventBus =
    let public EventPublisher = new Event<Event>() ← Брокер событий для коммуникации
                                                основан на шаблоне
                                                «издатель — подписчик»
    let public Subscribe (eventHandle: Events.Event -> unit) =
        EventPublisher.Publish |> Observable.subscribe(eventHandle) ←
    let public Notify (event:Event) = EventPublisher.Trigger event ←

module EventStorage =
    type EventStorageMessage = ← Использование типов сообщений в хранилище
                                событий для сохранения событий или получения истории
        | SaveEvent of id:Guid * event:EventDescriptor
        | GetEventsHistory of Guid * AsyncReplyChannel<Event list option>
    type EventStorage() = ← Реализация хранилища событий в памяти
                            с применением MailboxProcessor для потокобезопасности
        let eventstorage = MailboxProcessor.Start(fun inbox ->
  
```

```

let rec loop (history:Dictionary<Guid, EventDescriptor list>) =
    async { ←
        let! msg = inbox.Receive() ←
        match msg with
        | SaveEvent(id, event) -> ←
            EventBus.Notify event.EventData ←
            match history.TryGetValue(id) with
            | true, events -> history.[id] <- (event :: events)
            | false, _ -> history.Add(id, [event])
        | GetEventsHistory(id, reply) -> ←
            match history.TryGetValue(id) with
            | true, events ->
                events |> List.map (fun i -> i.EventData) |> Some
                |> reply.Reply
            | false, _ -> reply.Reply(None)
            return! loop history } ←
    loop (Dictionary<Guid, EventDescriptor list>()) ←

```

Сохранение события с применением
в качестве ключа уникального идентификатора
SignalR-подключения пользователя; если запись
с таким ключом уже существует, то событие добавляется к ней

member this.SaveEvent(id:Guid) (event:EventDescriptor) =
eventstorage.Post(SaveEvent(id, event))

member this.GetEventsHistory(id:Guid) = ←
 eventstorage.PostAndReply(fun rep -> GetEventsHistory(id, rep)) ←
 |> Option.map(List.iter) ←
 Переупорядочение истории событий

Получение истории событий

Тип `EventBus` — это простая реализация шаблона «издатель — подписчик» для событий. Внутри функции `Subscribe` для регистрации события используется Rx; функция получает уведомление при активации `EventPublisher` посредством функции `Notify`. Тип `EventBus` — это удобный способ дать сигнал различным частям приложения о том, что компонент отправляет уведомление при достижении заданного состояния.

Событие является результатом действия, которое уже произошло, — скорее всего, результатом выполнения команды. Тип `EventStorage` — это размещаемое в памяти хранилище для поддержки концепции порождения событий, которая в своей основе заключается в сохранении последовательности событий изменения состояния приложения, а не в сохранении текущего состояния объекта. Таким образом,

приложение способно в любой момент восстановить текущее состояние объекта путем воспроизведения событий. Поскольку сохранение события — это отдельная операция, оно по своей сути является атомарным.

Реализация `EventStorage` основана на F#-агенте `MailboxProcessor`, который гарантирует потокобезопасность при доступе к внутреннему словарю истории структуры событий `Dictionary<Guid, EventDescriptor list>`. Размеченнное объединение `EventStorageMessage` определяет следующие две операции, выполняемые с хранилищем событий.

- ❑ `SaveEvent` добавляет `EventDescriptor` во внутреннее состояние агента хранилища событий по заданному уникальному идентификатору. Если такой идентификатор уже существует, то событие добавляется к нему.
- ❑ `GetEventsHistory` извлекает историю событий в упорядоченной временной последовательности по заданному уникальному идентификатору. В общем случае история событий воспроизводится с использованием заданной функции действия, как показано в листинге 14.5.

В реализации использован агент, так как это удобный способ абстрагироваться от основ хранилища событий. Благодаря этому можно легко создавать различные типы хранилищ событий, изменяя только две функции — `SaveEvent` и `GetEventsHistory`.

Рассмотрим объект `StockMarket`, показанный на рис. 14.9. В листинге 14.6 показана внутренняя реализация приложения — объект `StockMarket`.

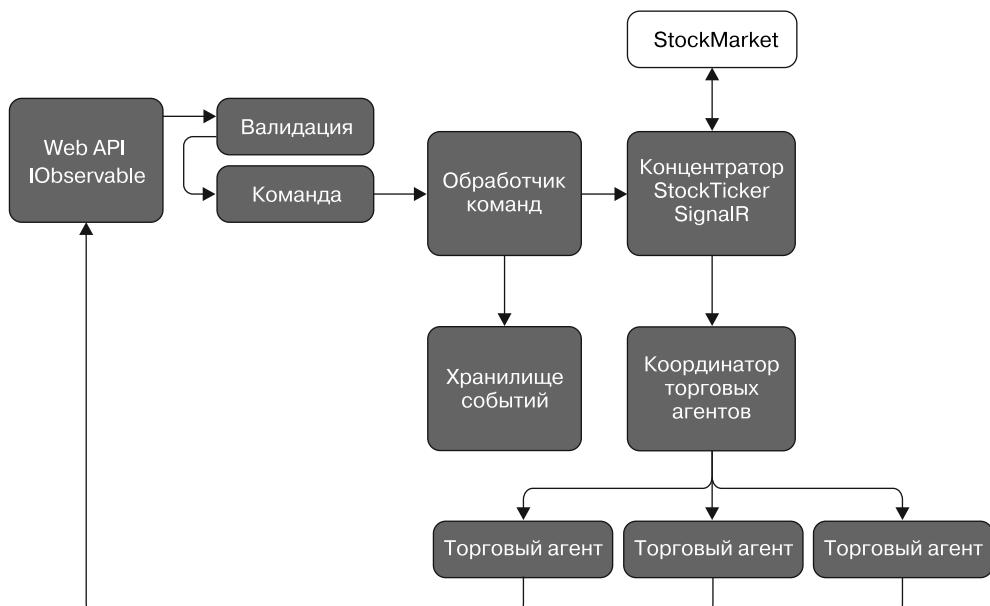


Рис. 14.9. Объект `StockMarket`, реализованный в листинге 14.6

Листинг 14.6. Тип StockMarket для координации пользовательских соединений

```

type StockMarket (initStocks : Stock array) =
    let subject = new Subject<Trading>()

    Экземпляр Rx Subject, который
    реализует шаблон «издатель — подписчик»
    ↓
    static let instanceStockMarket =
        Lazy.Create(fun () -> StockMarket(Stock.InitialStocks()))

    let stockMarketAgent =
        Agent<StockTickerMessage>.Start(fun inbox ->
            let rec marketIsOpen (stocks : Stock array)
                → (stockTicker : IDisposable) = async {
                    Состояние
                    MailboxProcessor
                    для хранения
                    обновленных
                    значений
                    курсов акций
                    ↓
                    Обновление курсов
                    акций, отправка
                    уведомлений
                    подписчикам
                    ↓
                    Использование
                    двух состояний агента
                    для изменения
                    состояния
                    рынка
                    на Open или Close
                    ↓
                    let! msg = inbox.Receive()
                    match msg with
                    | GetMarketState(c, reply) ->
                        reply.Reply(MarketState.Open)
                        return! marketIsOpen stocks stockTicker
                    | GetAllStocks(c, reply) ->
                        reply.Reply(stocks |> Seq.toList)
                        return! marketIsOpen stocks stockTicker
                    | UpdateStockPrices ->
                        stocks
                        |> PSeq.iter(fun stock ->
                            let isStockChanged = updateStocks stock stocks
                            isStockChanged
                            |> Option.iter(fun _ ->
                                subject.OnNext(Trading.UpdateStock(stock))))
                        return! marketIsOpen stocks stockTicker
                    | CloseMarket(c) ->
                        stockTicker.Dispose()
                        return! marketIsClosed stocks
                    | _ -> return! marketIsOpen stocks stockTicker }

            marketIsClosed (stocks : Stock array) = async {
                let! msg = inbox.Receive()
                match msg with
                | GetMarketState(c, reply) ->
                    reply.Reply(MarketState.Closed)
                    return! marketIsClosed stocks
                | GetAllStocks(c, reply) ->
                    reply.Reply((stocks |> Seq.toList))
                    return! marketIsClosed stocks
                | OpenMarket(c) ->
                    return! marketIsOpen stocks (startStockTicker inbox)
                | _ -> return! marketIsClosed stocks }

            member this.GetAllStocks(connId) =

```

Этот экземпляр имитирует фондовый рынок, обновляя курсы акций

Использование двух состояний агента для изменения состояния объекта Market: Open или Close

Использование сопоставления с шаблоном для отправки сообщений соответствующему поведению

Применение параллельных итераций с PSeq для максимально быстрой отправки обновлений

Приложение сопоставления с шаблоном для отправки сообщений соответствующему поведению

```
stockMarketAgent.PostAndReply(fun ch -> GetAllStocks(connId, ch))  
  
member this.GetMarketState(connId) =  
    stockMarketAgent.PostAndReply(fun ch -> GetMarketState(connId, ch))  
  
member this.OpenMarket(connId) =  
    stockMarketAgent.Post(OpenMarket(connId))  
  
member this.CloseMarket(connId) =  
    stockMarketAgent.Post(CloseMarket(connId))  
  
member this.AsObservable() = subject.AsObservable()  
    .SubscribeOn(TaskPoolScheduler.Default) ← Представление типа StockMarket  
    как наблюдаемого объекта,  
    чтобы можно было подписаться  
на его внутренние изменения  
  
static member Instance() = instanceStockMarket.Value
```

Представление типа StockMarket как наблюдаемого объекта, чтобы можно было подписаться на его внутренние изменения

Тип **StockMarket** отвечает за имитацию фондового рынка в приложении. В нем применяются операции **OpenMarket** и **CloseMarket**, чтобы запускать или останавливать рассылку уведомлений об обновлении курса акций, а функция **GetAllStocks** извлекает биржевые сводки для мониторинга и управления пользователями. Реализация типа **StockMarket** основана на агентной модели с применением **MailboxProcessor**, чтобы прибегнуть к преимуществам внутренней потокобезопасности и удобной семантики конкурентной асинхронной передачи сообщений, которая лежит в основе построения высокопроизводительных и реактивных (управляемых событиями) систем.

Обновления цены StockTicker имитируются путем отправки в StockMarketAgent MailboxProcessor большого количества случайных запросов посредством UpdateStockPrices, который затем уведомляет всех активных клиентов-подписчиков.

Член `AsObservable` представляет тип `StockMarket` как поток событий в интерфейсе `Iobservable`. Таким образом, тип `StockMarket` может уведомлять `Iobserver`, подписанный на интерфейс `Iobservable`, об обновлениях курсов акций, которые генерируются при получении сообщения `UpdateStock`.

Функция, обновляющая курс акций, использует Rx-таймер, чтобы выдавать случайные значения для каждого из зарегистрированных биржевых кодов, увеличивая или уменьшая цены на небольшой процент, как показано в листинге 14.7.

Листинг 14.7. Функция обновления биржевых котировок через заданные интервалы

```
let startStockTicker (stockAgent : Agent<StockTickerMessage>) =
    Observable.Interval(TimeSpan.FromMilliseconds 50.0)
        |> Observable.subscribe(fun _ -> stockAgent.Post UpdateStockPrices)
```

`startStockTicker` — это имитатор поставщика услуг, который каждые 50 мс сообщает `StockTicker`, что пора обновить цены.

ПРИМЕЧАНИЕ

Отправка сообщений F#-агенту MailboxProcessor (или блоку TPL Dataflow) едва ли создаст узкое место в системе, поскольку MailboxProcessor способен обрабатывать 30 млн сообщений в секунду на машине с частотой ядра 3,3 ГГц.

Назначение типа `TradingCoordinator` (рис. 14.10) состоит в том, чтобы управлять внутренними активными SignalR-соединениями и подписчиками `TradingAgent`, которые выступают в качестве наблюдателей, через `MailboxProcessor coordinatorAgent`. Реализация показана в листинге 14.8.

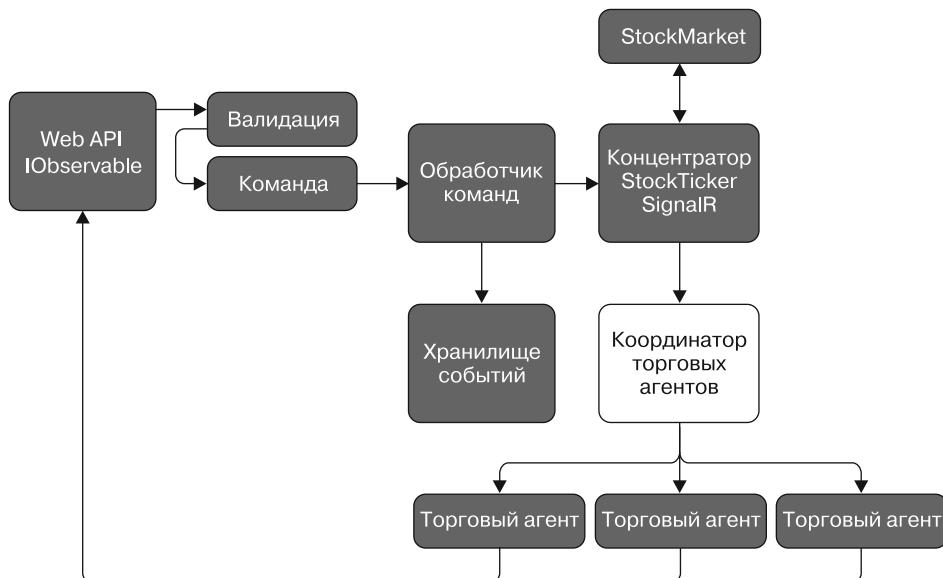


Рис. 14.10. Координатор торговых агентов, реализованный в листинге 14.8

Листинг 14.8. Агент `TradingCoordinator` для обработки активного дочернего торгового агента

Использование размеченного объединения,
чтобы определить тип сообщения для `TradingCoordinator`

```

type CoordinatorMessage =
    | Subscribe of id : string * initialAmount : float *
    | caller:IhubCallerConnectionContext<IStockTickerHubClient>
    | Unsubscribe of id : string
    | PublishCommand of connId : string * CommandWrapper

```

Реактивный шаблон «издатель —
подписчик», описанный в листинге 6.6

```

type TradingCoordinator() = 
    let subject = new RxPubSub<Trading>()
    static let tradingCoordinator =
        Lazy.Create(fun () -> new TradingCoordinator())

```

Использование экземпляра
одиночки типа `TradingCoordinator`

```

let coordinatorAgent =
    Agent<CoordinatorMessage>.Start(fun inbox ->

```

```

let rec loop (agents : Map<string,
  ↳ (IObserver<Trading> * IDisposable)) = async {
  → Использование экземпляра TradingAgent,
    который играет роль наблюдателя и получает
    уведомления в реактивном стиле

  let! msg = inbox.Receive()
  match msg with
  | Subscribe(id, amount, caller) -> ←
    let observer = TradingAgent(id, amount, caller)
    let dispObsrever = subject.Subscribe(observer)
    observer.Agent
    |> reportErrorsTo id supervisor |> startAgent
    caller.Client(id).SetInitialAsset(amount) ←
    return! loop (Map.add id (observer :>
      IObserver<Trading>, dispObsrever) agents) ←
    Уведомление клиента
    | Unsubscribe(id) -> ←
      match Map.tryFind id agents with
      | Some(_, disposable) -> ←
        Disposable.Dispose()
        return! loop (Map.remove id agents) ←
        Публикация команд
        | None -> return! loop agents ←
        с использованием реактивного
        | PublishCommand(id, command) -> ←
        шаблона «издатель —
        подписчик» для размещения
        match command.Command with
        | TradingCommand.BuyStockCommand(id, trading) -> ←
          match Map.tryFind id agents with
          | Some(a, _) -> ←
            let tradingInfo = {Quantity=trading.Quantity;
              Price=trading.Price;
              TradingType = TradingType.Buy}
            a.OnNext(Trading.Buy(trading.Symbol, tradingInfo))
            return! loop agents ←
            Публикация команд
            | None -> return! loop agents ←
            | TradingCommand.SellStockCommand(id, trading) -> ←
              match Map.tryFind id agents with
              | Some(a, _) -> ←
                let tradingInfo = {Quantity=trading.Quantity;
                  Price=trading.Price;
                  TradingType = TradingType.Sell}
                a.OnNext(Trading.Sell(trading.Symbol, tradingInfo))
                return! loop agents ←
                Подписка нового TradingAgent,
                который будет получать
                | None -> return! loop agents } ←
                уведомления при изменении
                цен на акции
                loop (Map.empty())
  member this.Subscribe(id : string, initialAmount : float,
  ↳ caller:IHubCallerConnectionContext<IStockTickerHubClient>) = ←
    coordinatorAgent.Post(Subscribe(id, initialAmount, caller)) ←
  
```

Применение логики управления ко вновь созданным агентам типа TradingAgent, описанным в главе 11

Отмена подписки существующего TradingAgent на заданный уникальный идентификатор и закрытие канала для получения уведомлений. Отмена подписки осуществляется путем удаления наблюдателя

Подписка нового TradingAgent, который будет получать уведомления при изменении цен на акции

```

member this.Unsubscribe(id : string) =
    coordinatorAgent.Post(Unsubscribe(id))

member this.PublishCommand(command) =
    coordinatorAgent.Post(command) ← TradingCoordinator предоставляет
                                    как наблюдаемый объект через
                                    экземпляр типа RxPubSub
                                    реактивного шаблона
                                    «издатель — подписчик»

member this.AddPublisher(observable : IObservable<Trading>) =
    subject.AddPublisher(observable) ← Использование члена класса,
                                    которому разрешено добавлять
                                    издатели, чтобы инициировать
                                    рассылку уведомлений для RxPubSub

static member Instance() = tradingCoordinator.Value ← Применение
                                         экземпляра одиночки
                                         типа TradingCoordinator

interface IDisposable with ← Удаление внутреннего
    member x.Dispose() = subject.Dispose()          типа субъекта
                                                    RxPubSub (важно)

```

Размеченное объединение `CoordinatorMessage` определяет сообщения для `coordinatorAgent`. Эти типы сообщений используются для координации операций с внутренними агентами `TradingAgent`, подписанными на уведомления об обновлениях.

`coordinatorAgent` можно представить как агент, который отвечает за поддержку активных клиентов. Он подписывает или отписывает их в зависимости от того, подключаются клиенты к приложению или отключаются от него, а затем отправляет активным клиентам операционные команды. В этом случае концентратор SignalR уведомляет `TradingCoordinator` об установке нового или удалении существующего соединения, чтобы он мог зарегистрировать или отменить регистрацию соответствующего клиента.

Агентурная модель используется в приложении для генерации нового агента для каждого входящего запроса. При распараллеливании операций запроса агент `TradingCoordinator` порождает новые агенты и назначает работу посредством сообщений. Это позволяет выполнять параллельные операции с ограничениями ввода-вывода, а также параллельные вычисления. `TradingCoordinator` предоставляет интерфейс `IObservable` через экземпляр типа `RxPubSub`, описанный в листинге 6.6. `RxPubSub` используется здесь для реализации высокопроизводительного реактивного шаблона «издатель — подписчик», где наблюдатели `TradingAgent` могут регистрироваться для получения уведомлений при обновлении цен на данный биржевой индекс. Другими словами, `TradingCoordinator` — это объект `Observable`, на который может подписаться обозреватель `TradingAgent`, реализуя реактивный шаблон «издатель — подписчик» для получения уведомлений.

Член-метод `AddPublisher` регистрирует любой тип, который реализует интерфейс `IObservable` и отвечает за обновление всех подписчиков `TradingAgent`. В данной реализации тип `IObservable`, зарегистрированный как `Publisher` в `TradingCoordinator`, является типом `StockMarket`.

Методы-члены `StockMarket Subscribe` и `Unsubscribe` используются для регистрации и отмены регистрации клиентских подключений, полученных из гелио-концентратора `StockTicker`. Запросы на подписку или отмену подписки передаются непосредственно во внутренний наблюдаемый тип `coordinatorAgent`.

Операция подписки, инициируемая сообщением `Subscribe`, проверяет, существует ли тип `TradingAgent` (рис. 14.11) в состоянии локального наблюдателя, проверяя уникальный идентификатор соединения. Если `TradingAgent` не существует, то создается новый экземпляр, и он подписывается на экземпляр субъекта путем реализации интерфейса `IObserver`. Затем ко вновь созданному наблюдателю `TradingAgent` применяется стратегия управления `reportErrorsTo` (для сообщения об ошибках и обработки ошибок). Эта стратегия управления обсуждалась в подразделе 11.5.5.

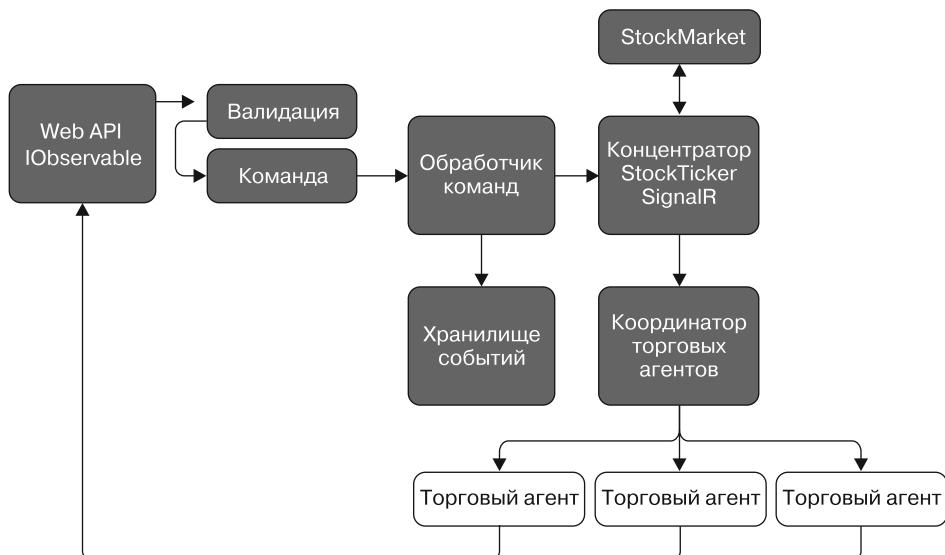


Рис. 14.11. `TradingAgent` представляет собой портфель на базе агентов для каждого пользователя, подключенного к системе. Этот агент обновляет пользовательский портфель и координирует операции покупки и продажи акций. `TradingAgent` реализован в листинге 14.9

Обратите внимание, что в конструкции `TradingAgent` использована ссылка на внутренний канал `SignalR`, который применяется для обеспечения прямой коммуникации с клиентом, в данном случае с мобильным устройством для уведомлений в реальном времени. Торговые операции `Buy` и `Sell` отправляются соответствующему `TradingAgent`, который идентифицируется по уникальному идентификатору из локального состояния наблюдателя. Операция отправки выполняется с использованием семантики `OnNext` типа `Observer`. Как уже отмечалось, `TradingCoordinator` отвечает за координацию операций `TradingAgent`, реализация которых показана в листинге 14.9.

Листинг 14.9. Агент TradingAgent, представляющий активного пользователя

```

type TradingAgent(connId : string, initialAmount : float, caller :
  IHubCallerConnectionContext<IStockTickerHubClient>) =
  let agent = new Agent<Trading>(fun inbox ->
    let rec loop cash (portfolio : Portfolio)
      (buyOrders : Treads) (sellOrders : Treads) = async {
        let! msg = inbox.Receive()
        match msg with
          | Kill(reply) -> reply.Reply()
          | Error(exn) -> raise exn
          | Trading.Buy(symbol, trading) ->
              let items = setOrder buyOrders symbol trading
              let buyOrders =
                createOrder symbol trading TradingType.Buy
                caller.Client(connId).UpdateOrderBuy(buyOrders)
              return! loop cash portfolio items sellOrders
          | Trading.Sell(symbol, trading) ->
              let items = setOrder sellOrders symbol trading
              let sellOrder =
                createOrder symbol trading TradingType.Sell
                caller.Client(connId).UpdateOrderSell(sellOrder)
              return! loop cash portfolio buyOrders items
          | Trading.UpdateStock(stock) ->
              caller.Client(connId).UpdateStockPrice stock
              let cash, portfolio, sellOrders = updatePortfolio cash
              stock portfolio sellOrders TradingType.Sell
              let cash, portfolio, buyOrders = updatePortfolio cash
              stock portfolio buyOrders TradingType.Buy
      }
    member this.Agent = agent
    interface IObserver<Trading> with
      member this.OnNext value = agent.Post value
      member this.OnError exn = agent.Post(Choice1Of2 exn)
      member this.OnCompleted() = agent.Post(Choice2Of2())
  }

  let asset = getUpdatedAsset portfolio sellOrders
  buyOrders cash
  caller.Client(connId).UpdateAsset(asset)
  return! loop cash portfolio buyOrders sellOrders }

loop initialAmount (Portfolio(HashIdentity.Structural))
  (Treads(HashIdentity.Structural)) (Treads(HashIdentity.Structural)))

```

Конструктор принимает ссылку на соединение SignalR для уведомлений в реальном времени

Использование специального сообщения для реализации метода наблюдателя, позволяющего завершать (прерывать) рассылку уведомлений и обрабатывать ошибки

TradingAgent поддерживает в памяти локальное состояние клиентского портфеля и обрабатываемых торговых операций

Показ сообщений о торговых операциях

Обновление значения курса акции с уведомлением клиента и обновлением портфеля, если новое значение удовлетворяет требованиям любой из текущих торговых операций

Проверка текущего портфеля на возможные обновления в соответствии с новыми значениями курсов акций

Клиент получает уведомления через внутренний канал SignalR

TradingAgent реализует интерфейс IObserver, чтобы действовать как подписчик для наблюдаемого типа TradingCoordinator

```
member this.OnNext(msg) = agent.Post(msg:Trading)
member this.OnError(exn) = agent.Post(Error exn)
member this.OnCompleted() = agent.PostAndReply(Kill)
```

Использование специального сообщения для реализации метода наблюдателя, позволяющего завершать (прерывать) рассылку уведомлений и обрабатывать ошибки

Тип **TradingAgent** — это объект на основе агента, который реализует интерфейс **IObserver**, позволяющий отправлять сообщения внутреннему агенту, используя реактивную семантику. Более того, поскольку тип **TradingAgent** относится к **Observer**, он может быть подписан на **TradingCoordinator** и, следовательно, автоматически получать уведомления в виде передаваемых сообщений. Это удобная конструкция, позволяющая разделить части приложения, которые могут взаимодействовать между собой реактивным и независимым образом, посредством передачи сообщений. **TradingAgent** представляет один активный клиент, что означает наличие отдельного экземпляра этого агента для каждого подключенного пользователя. Как уже отмечалось в главе 11, наличие нескольких тысяч работающих агентов (типа **MailboxProcessor**) не вызывает чрезмерной нагрузки на систему.

Локальное состояние **TradingAgent** отражает текущий портфель клиента и управляет им, включая торговые операции покупки и продажи акций. При получении сообщения **TradingMessage.Buy** или **TradingMessage.Sell** **TradingAgent** проверяет торговый запрос, добавляет операцию в локальное состояние, а затем отправляет клиенту уведомление, которое обновляет локальное состояние транзакции и вносит изменения в соответствующий пользовательский интерфейс.

Самым важным является сообщение **TradingMessage.UpdateStock**. **TradingAgent** может получать большое количество сообщений, целью которых будет обновление портфелей с новыми курсами акций. Что еще более важно, поскольку цены на акции могут изменяться в процессе обновления, функциональность, инициируемая сообщением **UpdateStock**, проверяет, удовлетворяет ли какая-либо из существующих (находящихся в работе) торговых операций **buyOrders** и **sellOrders** новому значению. Если какая-либо из совершаемых сделок выполняется, портфель обновляется соответствующим образом и клиент получает уведомление для каждого обновления.

Как уже упоминалось, для передачи возможных обновлений сущность **TradingAgent** сохраняет ссылку на канал соединения с клиентом, который был установлен во время события **OnConnected** в концентраторе **SignalR** (листинг 14.10).

Класс **StockTickerHub** является производным от класса **SignalR Hub**, который предназначен для обработки соединений, двунаправленного взаимодействия и обработки клиентских вызовов. Экземпляр класса **SignalR Hub** создается для каждой операции концентратора, такой как установка соединения и вызовы от клиентов серверу. Если вместо этого поместить состояние в класс **SignalR Hub**, то оно будет потеряно, так как экземпляры концентратора являются временными. Именно поэтому мы используем экземпляры **TradingAgent** для управления механизмом хранения данных о курсах акций, обновления цен и рассылки информации об обновлении цен.

Листинг 14.10. Гелиоконцентратор StockTicker

Использование атрибута `SignalR`

для определения имени концентратора,
на которое ссылается клиент для получения доступа

```
[<HubName("stockTicker")>]
type StockTickerHub() as this =
    inherit Hub<IStockTickerHubClient>()
```

StockTickerHub реализует строго
типовизированный класс `Hub<IStockTickerHubClient>`
для обеспечения коммуникации с `SignalR`

Применение по одному экземпляру типов `StockMarket`
и `TradingCoordinator`, который основан на принципе
агента и может использоваться в качестве экземпляра
одиночки в многопоточном режиме

```
let stockMarket : StockMarket = StockMarket.Instance()
let tradingCoordinator : TradingCoordinator = TradingCoordinator.
    Instance()
```

Для каждого соединения создается событие,
агент либо подписывается на него,
либо отписывается соответственно

```
override x.OnConnected() =
    let connId = x.Context.ConnectionId
    stockMarket.Subscribe(connId, 1000., this.Clients)
    base.OnConnected()
```

Использование базовых событий `SignalR`
для управления новыми
и разорванными соединениями

```
override x.OnDisconnected(stopCalled) =
    let connId = x.Context.ConnectionId
    stockMarket.Unsubscribe(connId)
    base.OnDisconnected(stopCalled)
```

Методы управления
событиями фондового рынка

```
member x.GetAllStocks() =
    let connId = x.Context.ConnectionId
    let stocks = stockMarket.GetAllStocks(connId)
    for stock in stocks do
        this.Clients.Caller.SetStock stock
```

```
member x.OpenMarket() =
    let connId = x.Context.ConnectionId
    stockMarket.OpenMarket(connId)
    this.Clients.All.SetMarketState(MarketState.Open.ToString())
```

```
member x.CloseMarket() =
    let connId = x.Context.ConnectionId
    stockMarket.CloseMarket(connId)
    this.Clients.All.SetMarketState(MarketState.Closed.ToString())
```

```
member x.GetMarketState() =
    let connId = x.Context.ConnectionId
    stockMarket.GetMarketState(connId).ToString()
```

Использование шаблона «Одиночка» — распространенный способ сохранить объект экземпляра внутри концентратора `SignalR`. В данном случае мы создаем экземпляр одиночки типа `StockMarket`; поскольку его реализация основана на

агентах, не возникает проблем с гонкой потоков и снижением производительности, описанных в разделе 3.1.

Базовые методы SignalR `OnConnected` и `OnDisconnected` вызываются каждый раз, когда устанавливается новое или разрывается существующее соединение; при этом либо создается и регистрируется экземпляр `TradingAgent`, либо регистрация данного экземпляра отменяется и он уничтожается.

Другие методы обрабатывают операции фондового рынка, такие как его открытие и закрытие. Базовый канал SignalR немедленно уведомляет активных клиентов о каждой из этих операций, как показано в листинге 14.11.

Листинг 14.11. Клиентский интерфейс `StockTicker` для получения уведомлений с использованием SignalR

```
interface IstockTickerHub
{
    Task Init(string serverUrl, IStockTickerHubClient client);
    string ConnectionId { get; }
    Task GetAllStocks();
    Task<string> GetMarketState();
    Task OpenMarket();
    Task CloseMarket();
}
```

Интерфейс `IStockTickerHub` используется на стороне клиента для определения методов, которые могут вызывать клиенты, в классе SignalR `Hub`. Чтобы сделать метод, который вы хотите вызывать из клиента, доступным в концентраторе, нужно объявить этот метод как открытый. Обратите внимание, что методы, определенные в интерфейсе, могут долго выполнятся, поэтому они возвращают тип `Task` (или `Task<T>`), предназначенный для асинхронной работы, чтобы избежать блокировки соединения при использовании транспортного протокола `WebSocket`. Если метод возвращает объект `Task`, SignalR ожидает завершения задачи, а затем отправляет распакованный результат обратно клиенту.

Мы воспользуемся Portable Class Library (P2P), чтобы одна и та же функциональность была доступна на разных платформах. Назначение интерфейса `IStockTickerHub` — установка специального платформенно-ориентированного контракта для реализации концентратора SignalR. Таким образом, каждая платформа должна соответствовать точному определению этого интерфейса, внедряемому во время выполнения приложения посредством поставщика классов `DependencyService` (<http://mng.bz/vFc3>):

```
IStockTickerHub stockTickerHub = DependencyService.Get<IStockTickerHub>();
```

После того как контракт `IStockTickerHub` определен, для установления способа взаимодействия клиента и сервера в листинге 14.12 показана реализация мобильного приложения, а именно класса `ViewModel`, содержащего основную функциональность. Некоторые из его свойств удалены из первоначального исходного кода, поскольку повторяющаяся логика может отвлечь от основной цели этого примера.

Листинг 14.12. Клиентское мобильное приложение с использованием Xamarin.Forms

```

public class MainPageViewModel : ModelObject, IstockTickerHubClient
{
    public MainPageViewModel(Page page)
    {
        Stocks = new ObservableCollection<StockModelObject>();
        Portfolio = new ObservableCollection<Models.OrderRecord>();
        BuyOrders = new ObservableCollection<Models.OrderRecord>();
        SellOrders = new ObservableCollection<Models.OrderRecord>(); ← Использование коллекции наблюдаемых
                                                                объектов для рассылки уведомлений
                                                                об автоматических обновлениях свойств ViewModel

        Stocks = new ObservableCollection<StockModelObject>();
        Portfolio = new ObservableCollection<Models.OrderRecord>();
        BuyOrders = new ObservableCollection<Models.OrderRecord>();
        SellOrders = new ObservableCollection<Models.OrderRecord>(); ←

        SendBuyRequestCommand =
            new Command(async () => await SendBuyRequest());
        SendSellRequestCommand =
            new Command(async () => await SendSellRequest()); → Применение асинхронных команд
                                                                для отправки заказов на торговые
                                                                операции покупки или продажи

        stockTickerHub = DependencyService.Get<IStockTickerHub>(); ← Инициализация stockTickerHub для установки
                                                                соединения с сервером. Во время инициализации
                                                                и клиент-серверного соединения пользовательский
                                                                интерфейс обновляется соответствующим образом

        hostPage = page;

        var hostBase = "http://localhost:8735/"; ←
        stockTickerHub
            .Init(hostBase, this)
            .ContinueWith(async x =>
        {
            var state = await stockTickerHub.GetMarketState();
            isMarketOpen = state == "Open";
            OnPropertyChanged(nameof(IsMarketOpen));
            OnPropertyChanged(nameof(MarketStatusMessage)); → Инициализация stockTickerHub выполняется в контексте
                                                                синхронизации пользовательского интерфейса,
                                                                чтобы легко обновлять элементы управления им

            await stockTickerHub.GetAllStocks(); →
            , TaskScheduler.FromCurrentSynchronizationContext()); →

            client = new HttpClient(); ← Инициализация HttpClient,
                client.BaseAddress = new Uri(hostBase); ← который применяется для отправки
                client.DefaultRequestHeaders.Accept.Clear(); ← запросов к Web Server API
                client.DefaultRequestHeaders.Accept.Add(
                    new MediaTypeWithQualityHeaderValue("application/json")); ←

            }
        private IStockTickerHub stockTickerHub;
        private HttpClient client;
        private Page hostPage;

        public Command SendBuyRequestCommand {get; } →
        public Command SendSellRequestCommand {get; } ←
    }
}

```

```

private double price;
public double Price {
    get => price; set
    {
        if (price == value)
            return;
        price = value;
        OnPropertyChanged();
    }
}
private async Task SendTradingRequest(string url)
{
    if (await Validate()) {
        var request = new
        ➔ TradingRequest(stockTickerHub.ConnectionId, Symbol, Price, Amount);
        var response = await client.PostAsJsonAsync(url, request);
        response.EnsureSuccessStatusCode();
    }
}
private async Task SendBuyRequest() =>
    await SendTradingRequest("/api/trading/buy");
private async Task SendSellRequest() =>
    await SendTradingRequest("/api/trading/sell");

public ObservableCollection<Models.OrderRecord> Portfolio {get; }
public ObservableCollection<Models.OrderRecord> BuyOrders {get; }
public ObservableCollection<Models.OrderRecord> SellOrders {get; }
➔ public ObservableCollection<StockModelObject> Stocks {get; }

Применение коллекции наблюдаемых объектов
для рассылки уведомлений об автоматических
обновлениях свойств ViewModel
    Задействование функций, запускаемых каналом SignalR
    при отправке клиенту уведомлений от веб-серверного
    приложения. Эти функции обновляют пользовательский
    интерфейс
    public void UpdateOrderBuy(Models.OrderRecord value) =>
        BuyOrders.Add(value);
    public void UpdateOrderSell(Models.OrderRecord value) =>
        SellOrders.Add(value);
}

```

Класс `MainPageViewModel` является компонентом `ViewModel` мобильного клиентского приложения. Этот класс основан на шаблоне MVVM (<http://mng.bz/qfbR>) и предназначен для обеспечения коммуникации и связывания данных между пользовательским интерфейсом (представлением) и моделью представления. Таким образом, пользовательский интерфейс и логика представления имеют отдельные обязанности, что обеспечивает четкое разделение областей ответственности в приложении.

Обратите внимание, что класс `MainPageViewModel` реализует интерфейс `IStockTickerHubClient`, который допускает рассылку уведомлений из канала SignalR после установления соединения. Интерфейс `IStockTickerHubClient` определен в проекте

`StockTicker.Core`; он представляет собой контракт для клиента, на который полагается сервер. Реализация этого интерфейса показана в следующем фрагменте кода:

```
type IStockTickerHubClient =
    abstract SetMarketState : string -> unit
    abstract UpdateStockPrice : Stock -> unit
    abstract SetStock : Stock -> unit
    abstract UpdateOrderBuy : OrderRecord -> unit
    abstract UpdateOrderSell : OrderRecord -> unit
    abstract UpdateAsset : Asset -> unit
    abstract SetInitialAsset : float -> unit
```

Эти уведомления будут автоматически поступать от сервера в мобильное приложение, обновляя элементы управления пользовательского интерфейса в реальном времени. В листинге 14.12 коллекции наблюдаемых объектов, определенные в начале описания класса, применяются для двусторонней коммуникации с пользовательским интерфейсом. Когда одна из этих коллекций обновляется, изменения распространяются на контроллеры пользовательского интерфейса, чтобы они отражали новое состояние (<http://mng.bz/nvma>).

`Command` из `ViewModel` применяется для определения пользовательской операции, данные которой привязаны к кнопке, чтобы асинхронно отправить на веб-сервер запрос для выполнения торговой операции с акциями, определенной в пользовательском интерфейсе¹. Для выполнения запроса запускается метод `SendTradingRequest`, который применяется для покупки или продажи акций в соответствии с выбранной конечной точкой API.

Для установки SignalR-соединения инициализируется интерфейс `stockTickerHub` и создается его экземпляр путем вызова метода `DependencyService.Get<IStockTickerHub>`. После создания экземпляра `stockTickerHub` инициализация приложения выполняется путем вызова метода `Init`, который вызывает удаленный сервер для локальной загрузки курсов акций с помощью метода `stockTickerHub.GetAllStocks` и получения текущего состояния рынка с помощью метода `stockTickerHub.GetMarketState` для обновления пользовательского интерфейса.

Инициализация приложения выполняется асинхронно с использованием метода `FromCurrentSynchronizationContext TaskScheduler`, который предоставляет функциональные возможности для распространения обновлений на контроллеры пользовательского интерфейса из его главного потока без необходимости применения каких-либо операций регулирования потока.

В итоге приложение получает уведомления от SignalR-канала, подключенного к серверу фондового рынка, путем вызова методов, определенных в интерфейсе `IStockTickerHubClient`. Это методы `UpdateOrderBuy`, `UpdatePortfolio` и `UpdateOrderSell`, которые отвечают за обновление контроллеров пользовательского интерфейса путем изменения соответствующих коллекций наблюдаемых объектов.

¹ Cleary S. Async Programming: Patterns for Asynchronous MVVM Applications: Data Binding. <https://msdn.microsoft.com/magazine/dn605875>.

Тесты для измерения масштабируемости приложения для работы с биржевыми данными. Приложение для работы с биржевыми данными было развернуто в Microsoft Azure Cloud со средней конфигурацией (два ядра и 3,5 Гбайт оперативной памяти) и подверглось стресс-тестированию с использованием онлайн-инструментария для моделирования 5000 конкурентных подключений, каждое из которых генерировало сотни HTTP-запросов. Целью теста была проверка производительности веб-сервера при чрезмерной нагрузке, чтобы гарантировать, что критическая информация и сервисы будут доступны с ожидаемыми конечными пользователями скоростями. Результат оказался положительным — тесты подтвердили, что веб-серверное приложение способно поддерживать много конкурентных активных пользователей и справляться с чрезмерной нагрузкой HTTP-запросов.

Резюме

- ❑ Обычные веб-приложения можно рассматривать как естественно параллельные, поскольку их запросы полностью изолированы и легко выполняются независимо друг от друга. Чем мощнее сервер, на котором выполняется приложение, тем больше запросов он может обработать.
- ❑ Программу без сохранения состояния можно легко распараллелить и распределить между компьютерами и процессами для увеличения производительности. Нет необходимости поддерживать какое-либо состояние, в котором выполняются вычисления, поскольку ни одна часть программы не изменяет какую-либо структуру данных, что исключает состояние гонки данных.
- ❑ Асинхронность, кэширование и распределение (Asynchronicity, Caching, and Distribution, ACD) являются секретом успеха при разработке и реализации системы, способной гибко приспосабливаться к увеличению (или уменьшению) количества запросов с соответствующим параллельным ускорением и добавлением ресурсов.
- ❑ Для того чтобы отделить ASP.NET Web API и отправку сообщений, происходящих от входных запросов контроллера, к другим компонентам приложения подписчика, можно использовать Rx. Эти компоненты могут быть реализованы с применением агентной модели программирования, которая порождает новый агент для каждого установленного и активного пользовательского соединения. Таким образом приложение поддерживается в изолированном состоянии для каждого пользователя и обеспечивается простая возможность масштабирования.
- ❑ Поддержка конкурентного функционального программирования в .NET является определяющим условием, благодаря которому .NET представляет собой отличный инструмент для программирования на стороне сервера. .NET поддерживает асинхронное выполнение операций в декларативном стиле с семантикой компоновки; кроме того, для разработки потокобезопасных компонентов могут быть использованы агенты. Эти основные технологии можно объединять для декларативной обработки событий и достижения эффективного параллелизма с TPL.
- ❑ Управляемая событиями архитектура (Event-Driven Architecture, EDA) — это стиль разработки приложений, основанный на фундаментальных аспектах уведомлений о событиях, что обеспечивает мгновенное распространение информации и реактивное выполнение бизнес-процессов. В EDA информация распространяется в реальном времени в сильно распределенной среде, что позволяет различным компонентам приложения, получающим уведомления, проактивно реагировать на бизнес-действия. EDA обеспечивает низкую задержку и высокую отзывчивость системы. Различие между системами, управляемыми событиями, и системами, управляемыми сообщениями, состоит в том, что управляемые событиями системы фокусируются на адресуемых источниках событий, а системы, управляемые сообщениями, — на адресуемых получателях.

Приложения

Функциональное программирование



Можно с уверенностью сказать, что изучение функционального программирования сделает вас лучшим программистом. Действительно, ФП — это альтернативный, часто более простой способ представления проблем. Кроме того, многие технологии ФП могут быть успешно применены в других языках. Независимо от того, на каком языке вы работаете, программирование в функциональном стиле дает свои преимущества.

Функциональное программирование — это скорее образ мыслей, чем набор инструментов или языков. Именно знакомство с разными парадигмами программирования делает вас лучшим программистом, а у программиста, владеющего несколькими парадигмами программирования, больше возможностей, чем у программиста, владеющего несколькими языками. И следовательно...

Это приложение не охватывает аспекты ФП, необходимые для понимания всех технических основ, описанных в предыдущих главах и применяемых в области конкурентного программирования, таких как неизменяемость, ссылочная прозрачность, функции без побочных эффектов и «ленивые» вычисления. Скорее здесь приводится общая информация о том, что такое функциональное программирование, и о причинах, по которым стоит его изучить.

Что такое функциональное программирование

Разные люди вкладывают разный смысл в понятие функционального программирования. Это парадигма программирования, которая рассматривает вычисление как выполнение выражения. В науке *парадигма* описывает различные концепции или шаблоны мышления.

ФП включает в себя использование состояний и изменяемых данных для решения проблем предметной области и основывается на лямбда-вычислениях. Соответственно, функции являются *величинами первого класса*.

Величины первого класса

Величиной первого класса в языке программирования называется сущность, которая поддерживает все операции, доступные для других сущностей. Эти операции обычно включают в себя передачу в качестве параметра, возвращение из функции и присвоение переменной в качестве значения.

Функциональное программирование — это стиль программирования, основанный на вычислении выражений, а не на выполнении оператора. Термин «выражение» происходит из математики; выражение всегда возвращает результат (значение), не изменяя состояние программы. *Оператор* не возвращает ничего и может изменять состояние программы.

- *Выполнение операторов* относится к программе, представленной в виде последовательности команд или операторов. Команды определяют, как достичь конечного результата, создавая объекты и управляя ими.
- *Выполнение выражений* означает, как именно программа определяет свойства объекта, которые вы хотите получить в результате. Вы не описываете этапы, необходимые для построения объекта, и не можете случайно использовать объект прежде, чем он будет создан.

Преимущества функционального программирования

Функциональное программирование имеет следующие преимущества.

- *Модульность и возможность компоновки*. Благодаря внедрению чистых функций можно компоновать функции и создавать из простых функций абстракции более высокого уровня. Используя модули, можно лучше организовать программу. Возможность компоновки — мощнейший инструмент для преодоления сложности; она позволяет описывать и строить решения для сложных проблем.
- *Выразительность*. Сложные идеи можно представить в кратком декларативном формате, делая более ясными свои намерения, повышая способность рассуждать о программе и снижая сложность кода.
- *Надежность и тестирование*. Функции существуют без побочных эффектов; функция только вычисляет и возвращает значение, которое зависит от ее аргументов. Таким образом, можно проверить функцию, сосредоточившись только на ее аргументах, что позволяет лучше тестировать код и легко проверить его правильность.
- *Упрощенная конкурентность*. Конкурентность способствует ссылочной прозрачности и неизменяемости, которые являются основными условиями для

написания корректных конкурентных приложений без блокировок, способных эффективно работать на нескольких ядрах.

- **«Ленивые» вычисления.** Можно получить результат функции по требованию. Предположим, у вас есть большой поток данных, которые надо проанализировать. Благодаря LINQ вы можете использовать отложенное выполнение и «ленивые» вычисления, чтобы провести анализ данных по требованию (только при необходимости).
- **Продуктивность.** Это огромное преимущество: можно написать меньше строк кода и получить такую же реализацию, как и в другой парадигме. Продуктивность позволяет сократить время, необходимое для разработки программ, и, следовательно, повысить прибыль.
- **Корректность.** Можно написать меньше кода, естественным образом уменьшая возможное количество ошибок.
- **Поддерживаемость.** Это преимущество вытекает из других, таких как возможность компоновки, модульность, выразительность и корректность кода.

Освоение функционального программирования приводит к созданию модульного, ориентированного на выражения, концептуально простого кода. Сочетание этих возможностей ФП позволяет лучше понимать, что делает ваш код, независимо от того, в скольких потоках он выполняется.

Основные принципы функционального программирования

Существует четыре основных принципа функционального программирования, которые приводят к созданию хорошо скомпонованных декларативных программ:

- функции высшего порядка (Higher-Order Functions, HOF) как величины первого класса;
- неизменяемость;
- чистые функции, также известные как функции без побочных эффектов;
- декларативный стиль программирования.

Противостояние программных парадигм: от императивного к объектно-ориентированному и функциональному программированию

Объектно-ориентированное программирование делает код понятным благодаря инкапсуляции движущихся частей. Функциональное программирование делает код понятным, сводя число движущихся частей к минимуму.

*Майкл Фезерс, автор книги «Работа с устаревшим кодом»,
пост в Twitter*

В этом разделе описаны следующие три парадигмы программирования.

- ❑ *Императивное программирование* описывает вычисления в терминах операторов, которые изменяют состояние программы и определяют последовательность выполняемых команд. Таким образом, императивная парадигма — это стиль программирования, при котором для изменения состояния вычисляется последовательность операторов.
- ❑ *Функциональное программирование* (ФП) подразумевает построение структуры и элементов программы, при этом вычисления рассматриваются как выполнение выражений; таким образом, ФП способствует неизменяемости и избегает хранения состояний.
- ❑ *Объектно-ориентированное программирование* (ООП) оперирует не действиями, а объектами; его структуры данных содержат данные, а не логику. Основные парадигмы программирования можно разделить на императивные и функциональные. ООП лежит в плоскости, перпендикулярной императивному и функциональному программированию, — его можно комбинировать с обоими. Вам не нужно выбирать одну из двух парадигм; можно писать программное обеспечение в стиле ООП, используя функциональные или императивные концепции.

ООП существует уже почти два десятка лет, и его принципы проектирования применяются в таких языках, как Java, C# и VB.Net. ООП достигло большого успеха благодаря способности представлять и моделировать предметную область, повышая уровень абстракции. Основная идея применения языков программирования на основе ООП заключалась в возможности повторного применения кода, но эта идея часто искалась из-за изменений и настроек, необходимых для конкретных сценариев и специальных объектов. ООП-программы, разработанные со слабым связыванием и высокой повторной используемостью кода, напоминали сложный лабиринт с множеством секретных и запутанных ходов, затрудняющих чтение кода.

Чтобы улучшить возможность повторного применения кода, разработчики начали создавать шаблоны проектирования, позволяющие решить присущую ООП проблему громоздкого кода. Шаблоны проектирования побуждали разработчиков строить программное обеспечение в соответствии с шаблонами. Это делало базовый код более сложным, трудным для понимания и в некоторых случаях затрудняло поддержку, но все равно не улучшало возможности повторного использования. В ООП шаблоны проектирования полезны при создании решений повторяющихся проблем проектирования, но с точки зрения самого языка их можно считать дефектом абстракции.

В функциональном программировании шаблоны проектирования имеют другое значение; в сущности, большинство шаблонов проектирования, свойственных ООП, в функциональных языках не нужны вследствие более высокого уровня абстракции и HOF-функций, используемых в качестве строительных блоков. Благодаря более высокому уровню абстракции и уменьшенной рабочей нагрузке на низкоуровневые детали ФП имеет преимущество — позволяет создавать более короткие программы. Небольшую программу проще понимать, улучшать и верифицировать. ФП значительно упрощает многократное повторное применение кода и позволяет значительно сократить его повторяемость, что является наиболее эффективным способом написания кода, менее подверженного ошибкам.

Применение функций высшего порядка для увеличения абстракции

Принцип HOF означает, что функции могут передаваться в другие функции в качестве аргументов и функции могут возвращать другие функции в виде результатов. .NET поддерживает концепцию параметризованных делегатов, таких как `Action<T>` и `Func<T, TResult>`, которые можно использовать как HOF для передачи функций в виде параметров с поддержкой лямбда-выражений. Вот пример применения параметризованного делегата `Func<T, R>` на C#:

```
Func<int, double> fCos = n => Math.Cos( (double)n );
double x = fCos(5);
IEnumerable<double> values = Enumerable.Range(1, 10).Select(fCos);
```

На F# эквивалентный код с функциональной семантикой может быть представлен без необходимости явно использовать делегат `Func<T, TResult>`:

```
let fCos = fun n -> Math.Cos( double n )
let x = fCos 5
let values = [1..10] |> List.map fCos
```

Функции высшего порядка являются основой, определяющей главные возможности ФП. HOF имеют следующие преимущества:

- ❑ возможность компоновки и обеспечение модульности;
- ❑ многократное использование кода;
- ❑ возможность создавать высокодинамичные, адаптируемые системы.

Функции в ФП представляют собой величины первого класса. Это означает, что функции могут именоваться, подобно переменным, могут присваиваться переменным и появляться везде, где могут появляться любые другие конструкции языка. Если вы исходите из опыта ООП, то такая концепция позволяет использовать функции неканоническим способом — например, применять относительные параметрические операции к стандартным структурам данных. Функции высшего порядка позволяют сосредоточиться на результатах, а не на этапах их достижения. Это фундаментальный, мощный сдвиг в подходе к функциональным языкам. Следующие технологии функционального программирования позволяют добиться функциональной компоновки:

- ❑ компоновка;
- ❑ каррирование;
- ❑ частичное применение функций или частичные приложения.

Возможность использования делегатов позволяет описать функциональность не только методов, выполняющих одну задачу, но и поведенческих механизмов, которые можно улучшать, повторно применять и расширять. Этот стиль программирования, лежащий в основе функциональной парадигмы, имеет преимущество уменьшения объема рефакторинга кода: вместо нескольких специализированных и жестких методов программа может быть описана меньшим количеством гораздо

более общих и повторно используемых методов, которые можно усилить для обработки нескольких разных сценариев.

Применение функций высшего порядка и лямбда-выражений для создания многократно используемого кода

Одна из многих важных причин использования лямбда-выражений — это рефакторинг кода, позволяющий уменьшить избыточность кода. В языках с управлением памятью, таких как C#, рекомендуется по возможности размещать ресурсы в памяти. Рассмотрим следующий пример:

```
string text;
using (var stream = new StreamReader(path))
{
    text = stream.ReadToEnd();
}
```

В данном коде ресурс `StreamReader` размещается в памяти с помощью ключевого слова `using`. Это хорошо известный шаблон, но у него есть ограничения — он не может быть повторно использован, поскольку размещаемая в памяти переменная объявляется внутри области применения, что делает невозможным ее повторное применение после удаления, и в случае ее обращения к размещаемым в памяти объектам будет генерироваться исключение. Рефакторинг кода в классическом стиле ООП не является тривиальной задачей. Можно использовать метод шаблона, но это решение также вносит дополнительные сложности из-за необходимости создания нового базового класса и реализации всех производных классов. Лучшее, более элегантное решение — задействовать лямбда-выражение (анонимный делегат). Ниже представлен код для реализации и использования статического вспомогательного метода:

```
R Using<T,R>(this T item, Func<T, R> func) where T : IDisposable {
    using (item)
        return func(item);
}
string text = new StreamReader(path).Using(stream => stream.ReadToEnd());
```

В этом коде реализован гибкий и многократно используемый шаблон для освобождения размещаемых в памяти ресурсов. Его единственным ограничением является то, что параметризованный тип `T` должен быть типом, который реализует `IDisposable`.

Лямбда-выражения и анонимные функции

Термином «*лямбда*» или «*лямбда-выражение*» обычно называют анонимные функции. Назначение лямбда-выражения состоит в том, чтобы описать вычисление в виде функции, используя привязку и подстановку переменных. Проще говоря, лямбда-выражение — это неименованный метод, написанный вместо экземпляра делегата, который вводит понятие *анонимных функций*.

Лямбда-выражения повышают уровень абстракции, что позволяет упростить процесс программирования. Функциональные языки, такие как F#, основаны на лямбда-исчислении, которое применяется для выражения вычислений в виде функциональных абстракций; таким образом, лямбда-выражение является частью ФП-языка. Однако в C# главной мотивацией для введения лямбда-выражений является упрощение потоковых абстракций, позволяющих создавать декларативные потоковые API. Подобные абстракции представляют собой доступный и естественный способ обеспечения многоядерного параллелизма, делая лямбда-выражения ценным инструментом в современных вычислениях.

Лямбда-исчисление или лямбда-выражения?

Лямбда-исчисление (также известное как λ -исчисление) — это формальная система в математической логике и информатике, предназначенная для описания вычислений с использованием привязки и подстановки переменных в функциях, которые являются единственной структурой данных. Лямбда-исчисление ведет себя как небольшой язык программирования, где может быть описана и выполнена любая вычисляемая функция. Например, лямбда-исчисление лежит в основе .NET LINQ.

Лямбда-выражение определяет специальный анонимный метод. Анонимные методы — это экземпляры делегатов, не имеющие имени объявления метода. Термины «*лямбда*» и «*лямбда-выражение*» чаще всего означают анонимные функции.

Лямбда-метод — это «синтаксический сахар» и компактный синтаксис, предназначенный для встраивания в код анонимного метода:

```
Func<int, int> f1 = delegate(int i) { return i + 1; }; ← Анонимный метод
Func<int, int> f2 = i => i+1; ← Лямбда-выражение
```

Для того чтобы создать лямбда-выражение, нужно указать слева от лямбда-оператора `=>` (произносится как «отображает») входные параметры (если они есть), а справа — блок выражений или операторов. Например, лямбда-выражение `(x, y) => x + y` принимает два параметра `x` и `y` и возвращает сумму этих значений.

Любое лямбда-выражение состоит из следующих трех частей:

- (x, y) — набор параметров;
- `=>` — оператор следования `=>`, который отделяет список аргументов от результирующего выражения;
- $x + y$ — набор операторов, которые выполняют действие или возвращают значение. В данном примере лямбда-выражение возвращает сумму `x` и `y`.

Ниже показано, как можно реализовать три лямбда-выражения с одинаковым поведением:

```
Func<int, int, int> add = delegate(int x, int y){ return x + y; };
Func<int, int, int> add = (int x, int y) => { return x + y; };
Func<int, int, int> add = (x, y) => x + y
```

Часть `Func<int, int, int>` определяет функцию, которая принимает два целых числа и возвращает новое целое число.

На F# строго типизированная система может связывать имя или метку с функцией без явного объявления. Функции F# являются примитивами, аналогичными целым числам и строкам. Можно преобразовать предыдущую функцию в эквивалентный F#-синтаксис следующим образом:

```
let add = (fun x y -> x + y)
let add = (+)
```

На F# оператор «плюс» `(+)` — это функция с той же сигнатурой, что и `add`, она принимает два числа и возвращает в качестве результата их сумму.

Лямбда-выражения — простое и эффективное решение для назначения и выполнения блока встроенного кода, особенно в случае, если этот блок имеет одну конкретную цель и нет необходимости определять его как метод. У использования лямбда-выражений в коде есть множество преимуществ. Вот лишь некоторые из них.

- ❑ Нет необходимости явной параметризации типов; компилятор сам определяет типы параметров.
- ❑ Сжатое встроенное кодирование (функции существуют внутри строки) позволяет избежать сбоев, вызванных тем, что разработчикам приходится искать данную функциональность в другом месте кода.
- ❑ Захваченные переменные ограничивают действие переменных уровня класса.
- ❑ Лямбда-выражения делают поток кода более понятным и удобным для чтения.

Каррирование

Термин «каррирование» придумал Хаскелл Карри (Haskell Curry), математик, который оказал значительное влияние на развитие ФП. *Каррирование* — это технология, позволяющая многократно использовать код, представляя функции в виде модулей. Основная идея каррирования состоит в том, чтобы преобразовать выполнение функции, принимающей несколько параметров, в выполнение последовательности функций, каждая из которых имеет только один параметр. Функциональные языки тесно связаны с математическими концепциями, где у функции может быть только один параметр. Этой концепции следует F#, где функции с несколькими параметрами объявляются как последовательность новых функций, каждая из которых имеет только один параметр.

На практике другие языки .NET допускают функции с несколькими аргументами; с точки зрения ООП, если не передать функции все ожидаемые аргументы, компилятор выдаст исключение. И наоборот, в ФП чрезвычайно легко написать каррированную функцию, возвращающую любую функцию, которую ей передали. Но, как уже упоминалось, лямбда-выражения обеспечивают отличный синтаксис для создания анонимных делегатов, что упрощает реализацию каррированных функций. Более того, можно реализовать каррирование на любом языке программирования, который поддерживает замыкания, — интересная концепция,

поскольку эта технология упрощает лямбда-выражения, включая только функции с одним параметром.

Технология каррирования позволяет рассматривать все функции с одним или несколькими аргументами как функции, принимающие только один аргумент, независимо от количества аргументов, необходимых для выполнения функции. Так создается цепочка функций, каждая из которых использует только один параметр.

В конце этой цепочки функций доступны все параметры, что позволяет выполнить исходную функцию. Кроме того, каррирование позволяет создавать специализированные группы функций, генерируемые благодаря фиксации аргументов базовой функции. В частности, если каррировать функцию с двумя аргументами и применить ее к первому аргументу, то функциональность будет ограничена одним измерением. Это не ограничение, а мощная технология, поскольку теперь можно применить новую функцию ко второму аргументу и вычислить конкретное значение.

В математической нотации между этими двумя функциями существует важное различие:

```
Add(x, y, z)  
Add x y z
```

Разница в том, что первая функция принимает один аргумент типа `tuple` (состоящий из трех элементов — `x`, `y` и `z`), а вторая функция принимает входной элемент `x` и возвращает функцию, принимающую входной элемент `y`, который возвращает функцию,ирующую элемент `z` и возвращающую результат окончательного вычисления. Проще говоря, эквивалентная функция может быть переписана в следующей форме:

```
((Add x) y) z
```

Важно отметить, что применение функций остается ассоциативным, принимая по одному аргументу за раз. Предыдущая функция `Add` применяется к `x`, а затем ее результат применяется к `y`. Результат такого применения `((Add x) y)` затем применяется к `z`. Поскольку каждый из этих промежуточных этапов представляет собой функцию, вполне приемлемо определить ее следующим образом:

```
Plus2 = Add 2
```

Эта функция эквивалента `Add x`. В данном случае можно ожидать, что функция `Plus2` принимает два входных аргумента и всегда передает 2 как фиксированный параметр. Для ясности можно переписать предыдущую функцию следующим образом:

```
Plus2 x = Add 2 x
```

Процесс получения промежуточных функций (каждая из которых принимает один входной аргумент) называется *каррированием*. Как же применяется каррирование на практике? Рассмотрим следующую простую C#-функцию, в которой используется лямбда-выражение:

```
Func<int,int,int> add = (x,y) => x + y;  
Func<int,Func<int,int>> curriedAdd = x => y => x + y;
```

В данном коде определяется функция `Func<int, int, int> add`, которая принимает в качестве аргументов два целых числа и возвращает целое число в качестве результата. При вызове этой функции компилятору требуются оба аргумента — и `x`, и `y`. Однако каррированная версия функции `add`, `curriedAdd`, выдает на выходе делегат со специальной сигнатурой `Func<int, Func<int, int>>`.

В общем случае любой делегат типа `Func<A, B, R>` может быть преобразован в делегат типа `Func<A, Func<B, R>>`. Эта каррированная функция принимает только один аргумент и возвращает функцию, которая принимает в качестве аргумента исходную функцию и возвращает значение типа `A`. Каррированная функция `curriedAdd` может применяться для создания мощных специализированных функций. Например, можно определить функцию `increment`, добавив значение 1:

```
Func<int,int> increment = curriedAdd(1)
```

Теперь можно использовать эту функцию для определения других функций, которые выполняют различные формы сложения:

```
int a = curriedAdd(30)
int b = increment(41)

Func<int, int> add30 = curriedAdd(30)
int c = add30(12)
```

Одним из преимуществ каррирования функций является то, что создаваемые специализированные функции проще использовать повторно; но настоящая ценность каррирования функций в том, что оно вводит полезную концепцию, называемую *частичным выполнением функций*, которая будет рассмотрена в следующем разделе. Другими преимуществами технологии каррирования являются уменьшение количества аргументов и простота повторного использования абстрактных функций.

Автоматическое каррирование в C#

Уровень абстракции технологии каррирования в C# можно автоматизировать и повысить, используя методы расширения. В данном примере целью метода расширения каррирования является введение «синтаксического сахара», позволяющего скрыть реализацию каррирования:

```
static Func<A, Func<B, R>> Curry<A, B, R>(this Func<A, B, R> function)
{
    return a => b => function(a, b);
}
```

Вот предыдущий код, реорганизованный с применением вспомогательного метода расширения:

```
Func<int,int,int> add = (x,y) => x + y;
Func<int,Func<int,int>> curriedAdd = add.Curry();
```

Этот синтаксис выглядит более лаконичным. Важно отметить, что компилятор может определять типы, задействуемые во всех функциях, и в этом его основная польза. В сущности, хотя `Curry` является параметризованной функцией, не обязательно

явно передавать ей параметризованные аргументы. Применение такого метода каррирования позволяет использовать другой синтаксис, более удобный для построения библиотеки сложных функций, скомпонованных из более простых функций. В исходном коде, выложенном в Сети в числе других ресурсов этой книги, есть библиотека, которая содержит полную реализацию вспомогательных методов, включая метод расширения для автоматического каррирования.

Раскаррирование

Функцию можно не только каррировать, но так же легко и раскаррировать, используя функции более высокого порядка для отмены каррирования. Раскаррирование — это, как следует из названия, преобразование, обратное каррированию. Раскаррирование можно представить как технологию отмены каррирования путем применения параметризованной функции раскаррирования.

В следующем примере каррированная функция с сигнатурой `Func<A, Func<B, R>>` будет преобразована обратно в функцию с несколькими аргументами:

```
public static Func<A, B, R> Uncurry<A, B, R>(Func<A, Func<B, R>> function)
    => (x, y) => function(x)(y);
```

Главное назначение раскаррирования функции — привести сигнатуру каррированной функции к стилю, более свойственному ООП.

Каррирование в F#

В F# объявления функций являются каррированными по умолчанию. Но, несмотря на то что компилятор делает это автоматически, полезно понимать, как именно F# обрабатывает каррированные функции.

В следующем примере показаны две F#-функции, выполняющие умножение двух значений. Если вы не знакомы с F#, то данные функции могут показаться вам одинаковыми или как минимум похожими, но это не так:

```
let multiplyOne (x,y) = x * y
let multiplyTwo x y = x * y

let resultOne = multiplyOne(7, 8)
let resultTwo = multiplyTwo 7 8
let values = (7,8)
let resultThree = multiplyOne values
```

Не считая синтаксиса, между этими функциями нет явных различий, но они ведут себя по-разному. Первая функция имеет только один параметр, который является кортежем, состоящим из требуемых значений; вторая функция имеет два разных параметра — `x` и `y`.

Разница становится явной, если посмотреть на сигнатуры объявлений данных функций:

```
val multiplyOne : (int * int) -> int
val multiplyTwo : int -> int -> int
```

Теперь очевидно, что это разные функции. Первая принимает в качестве входного аргумента кортеж и возвращает целое число. Вторая принимает в качестве первого аргумента целое число и возвращает функцию, которая принимает на входе целое число и возвращает целое число. Эта вторая функция, принимающая два аргумента, автоматически преобразуется компилятором в цепочку функций, каждая из которых имеет только один входной аргумент.

В следующем примере показаны эквивалентные каррированные функции, показывающие, как именно компилятор выполняет интерпретацию:

```
let multiplyOne x y = x * y
let multiplyTwo = fn x -> fun y -> x * y

let resultOne = multiplyOne 7 8
let resultTwo = multiplyTwo 7 8
let resultThree =
    let tempMultiplyBy7 = multiplyOne 7
    tempMultiplyBy7 8
```

Реализация этих функций на F# аналогична, поскольку, как уже отмечалось, они каррируются по умолчанию. Главное назначение каррирования — оптимизировать функцию, чтобы было удобно использовать частичное применение.

Частично примененные функции

Частично примененная функция (или частичное применение функции) — это технология замены функции с несколькими аргументами и создания другой функции меньшей арности (*арность* функции — это количество ее аргументов). Таким образом, частичная функция представляет собой функцию с меньшим количеством аргументов, чем ожидалось, и создает специализированную функцию для заданных значений. Частично примененные функции, в дополнение к возможности компоновки, делают возможной модульную организацию функций.

Проще говоря, частичное применение функции — процесс привязки значений к параметрам, то есть частично примененные функции — это функции, в которых количество аргументов уменьшено за счет фиксированных значений (значений по умолчанию). Если у нас есть функция с N аргументами, то можно создать функцию с $N - 1$ аргументами, которая вызывает исходную функцию с фиксированным аргументом. Поскольку частичное применение зависит от каррирования, эти два метода применяются совместно. Разница между частичным применением и каррированием заключается в том, что частичное применение привязывает к фиксированному значению более одного параметра, поэтому для выполнения остальной части функции необходимо применить оставшиеся аргументы.

В общем случае при частичном применении параметризованная функция преобразуется в новую, специализированную функцию. Рассмотрим в качестве примера следующую каррированную C#-функцию:

```
Func<int,int,int> add = (x,y) => x + y;
```

Как из нее создать новую функцию с одним аргументом?

Это тот случай, когда полезно использовать частичное применение функции, поскольку можно частично применить функцию к HOF со значением по умолчанию для первого аргумента исходной функции. Ниже показан метод расширения, который можно использовать для частичного применения функции:

```
static Func<B, R> Partial<A, B, R>(this Func<A, B, R> function, A argument)
    => argument2 => function(argument, argument2);
```

И вот пример применения этой технологии:

```
Func<int, int, int> max = Math.Max;
Func<int, int> max5 = max.Partial(5);

int a = max5(8);
int b = max5(2);
int c = max5(12);
```

`Math.Max(int, int)` — пример функции, которая может быть расширена с помощью частично примененных функций. Используя в данном примере частично примененную функцию с зафиксированным по умолчанию значением аргумента, равным 5, мы создали новую специализированную функцию `max5`, которая находит максимальное значение из двух чисел со значением по умолчанию, равным 5. Благодаря частичному применению мы создали новую, более специализированную функцию из ранее существующей.

С точки зрения ООП частичное применение функции можно рассматривать как способ переопределения функций. Можно также использовать эту технологию для расширения на лету функциональности сторонней библиотеки, которая не является расширяемой.

Как уже отмечалось, в F# функции по умолчанию являются каррированными, что обеспечивает более простой способ создания частично примененных функций, чем в C#. У частичного применения функций есть много преимуществ, в первую очередь следующие:

- ❑ такие функции можно легко компоновать;
- ❑ нет необходимости передавать отдельный набор параметров, что позволяет избежать создания ненужных классов, которые содержат переопределенные версии того же метода с другим количеством входных аргументов;
- ❑ разработчик может писать высокообобщенные функции путем параметризации их поведения.

На практике преимущество использования частично применяемых функций состоит в том, что функции, построенные с задействованием лишь части аргументов, удобны для повторного использования кода, расширяемости функций и компоновки. Более того, частично применяемые функции упрощают использование HOF в вашем стиле программирования. Частичное применение функции также может быть отложенным для повышения производительности, как было показано в разделе 2.6.

Преимущества частичного применения и каррирования функций в C#

Рассмотрим более полный пример сценария реального использования частичного применения функции и каррирования. `Retry` в листинге А.1 — это метод расширения для делегата `Func<T>` для любой функции, которая не принимает параметров и возвращает значение типа `T`. Назначение данного метода — выполнение входящей функции в блоке `try-catch`; если во время выполнения выдается исключение, то функция будет повторять попытки выполнить операцию максимум три раза.

Листинг А.1. Метод расширения `Retry` на C#

```
public static T Retry<T>(this Func<T> function) {
    int retry = 0;           ← Установка счетчика
    T result = default(T); ← Присвоение результату значения по умолчанию T
    bool success = false;
    do {
        try {
            result = function(); ← Увеличение счетчика
            success = true;      ← Выполнение функции. В случае успешного завершения
        } catch {               ← цикл while останавливается и возвращается результат;
            retry++;           ← в противном случае выполняется новая итерация
        }
    } while (!success && retry < 3); ← Повторение три раза
    return result;
}
```

Предположим, что этот метод пытается прочитать текст из файла. В следующем коде метод `ReadText` принимает путь к файлу в качестве входного значения и возвращает текст из файла. Чтобы выполнить указанную функциональность с подключенным поведением `Retry` для повторного выполнения и восстановления в случае проблем, можно использовать замыкание, как показано далее:

```
static string ReadText(string filePath) => File.ReadAllText(filePath);

string filePath = "TextFile.txt";
Func<string> readText = () => ReadText(filePath);

string text = readText.Retry();
```

Для захвата локальной переменной `filePath` и передачи ее в метод `ReadText` можно использовать лямбда-выражение. Этот процесс позволяет создать функцию `Func<string>`, соответствующую сигнатуре метода расширения `Retry`, который может быть присоединен. Если файл заблокирован или принадлежит другому процессу, выдается ошибка и выполняется функционал `Retry`, как и ожидалось. Если первый вызов завершится неудачно, метод будет повторен второй и третий раз. В итоге он возвращает значение по умолчанию `T`.

Этот код работает, но возникает вопрос: что произойдет, если понадобится повторить функцию, которая нуждается в строковом параметре? Решение состоит

в частичном применении функции. В следующем коде реализована функция, принимающая в качестве параметра строку, представляющую собой путь к файлу для чтения текста. Затем функция передает этот параметр в метод `ReadText`. Поведение `Retry` работает только с функциями, которые не принимают параметров, поэтому код не компилируется:

```
Func<string, string> readText = (path) => ReadText(path);

string text = readText.Retry();
string text = readText(filePath).Retry();
```

Поведение `Retry` не работает с такой версией `readText`. Одно из возможных решений — написать другую версию метода `Retry`, принимающую дополнительный параметризованный аргумент, определяющий тип параметра, который требуется передать после вызова. Это неидеальное решение, поскольку нужно найти способ, применить данную новую логику `Retry` во всех методах, использующих эту функцию, с разными аргументами или реализациами.

Лучший вариант — задействовать сочетание каррирования и частичного применения функции. В листинге A.2 вспомогательные методы `Curry` и `Partial` определены как методы расширения.

Листинг A.2. Вспомогательные расширения `Retry` на C#

```
static class RetryExtensions
{
    public static Func<R> Partial<T, R>(this Func<T, R> function, T arg){
        return () => function(arg);
    }

    public static Func<T, Func<R>> Curry<T, R>(this Func<T, R> function){
        return arg => () => function(arg);
    }
}

Func<string, string> readText = (path) => ReadText(path);

string text = readText.Partial("TextFile.txt").Retry();

Func<string, Func<string>> curriedReadText = readText.Curry();
string text = curriedReadText("TextFile.txt").Retry();
```

Такой подход позволяет ввести путь к файлу и благополучно использовать функцию `Retry`. Это возможно, потому что обе вспомогательные функции, `Partial` и `Curry`, превращают функцию `readText` в функцию, которая не нуждается в параметре, в итоге соответствующая сигнатуре `Retry`.

Б

Обзор F#

В этом приложении рассматривается базовый синтаксис языка F#, который является общепризнанным первым функциональным языком общего назначения с поддержкой объектно-ориентированного программирования (ООП). Фактически F# охватывает объектную модель общеязыковой инфраструктуры (Common Language Infrastructure, CLI) в .NET, которая позволяет объявлять интерфейсы, обычные и абстрактные классы. Кроме того, F# представляет собой статически и строго типизированный язык, и это означает, что компилятор может определить тип переменных и функций на этапе компиляции. Синтаксис F# отличается от C-образных языков, таких как C#, так как не задействует фигурные скобки для разделения блоков кода. Более того, для разделения аргументов и определения границ тела функции в нем используются пробелы, а не запятые и отступы. Кроме того, F# является кроссплатформенным языком программирования, который может работать как внутри, так и за пределами экосистемы .NET.

let-привязки

let в F# является одним из самых важных ключевых слов, связывающих идентификатор и значение, что означает присвоение значению имени (или связывание значения с именем). Он определяется как `let <идентификатор> = <значение>`.

let-привязки являются неизменяемыми по умолчанию. Вот несколько примеров такого кода:

```
let myInt = 42
let myFloat = 3.14
let myString = "hello functional programming"
let myFunction = fun number -> number * number
```

Как видно из последней строки, можно присвоить функции имя, связав идентификатор `myFunction` с лямбда-выражением `fun number -> number * number`.

Ключевое слово `fun` используется для определения лямбда-выражения (анонимной функции) в синтаксисе вида `fun args -> body`. Примечательно, что вам не нужно определять типы в коде, поскольку вследствие жесткой встроенной системы определения типов компилятор F# распознает их изначально. Например, в предыдущем коде вследствие использования оператора умножения (`*`) компилятор сделал вывод, что аргумент функции `myFunction` является числом.

Сигнатуры функций в F#

В F#, как и в большинстве функциональных языков программирования, сигнатуры функций определяются посредством стрелочной нотации, которая читается слева направо. Функции — это выражения, всегда имеющие выходные данные, поэтому последняя правая стрелка всегда указывает на тип возвращаемого значения. Например, запись `typeA -> typeB` можно интерпретировать как функцию, которая принимает входное значение типа `typeA` и выдает значение типа `typeB`. Тот же принцип применим к функциям, принимающим более двух аргументов. Если функция имеет сигнатуру `typeA -> typeB -> typeC`, то, читая стрелки слева направо, мы получим две функции. Первая из них — это `typeA -> (typeB -> typeC)`, она принимает на входе `typeA` и возвращает функцию `typeB -> typeC`.

Вот сигнатура функции `add`:

```
val add : x:int -> y:int -> int
```

Эта функция принимает один аргумент `x:int` и возвращает функцию, которая принимает на входе `y:int` и возвращает `int` в качестве результата. Стрелочная нотация неразрывно связана с каррированием и анонимными функциями.

Создание изменяемых типов: `mutable` и `ref`

Одна из главных концепций функционального программирования — неизменяемость. F# — изначально функциональный язык программирования; но явное использование ключевых слов `immutable` позволяет создавать изменяемые типы, которые ведут себя как переменные, например:

```
let mutable myNumber = 42
```

Теперь можно изменить значение `myNumber` с помощью оператора перехода (`<-`):

```
myNumber <- 51
```

Другим вариантом определения изменяемого типа является использование *ссылочной ячейки*, которая определяет место хранения, что позволяет создавать изменяемые значения со ссылочной семантикой. Оператор `ref` объявляет новую ссылочную

ячейку, инкапсулирующую значение, которое затем можно изменить с помощью оператора `:=` и получить к нему доступ с помощью оператора `!` (читается «бэнг»):

```
let myRefVar = ref 42
myRefVar := 53
printfn "%d" !myRefVar
```

В первой строке объявляется ссылочная ячейка `myRefVar` со значением 42; во второй строке ее значение меняется на 53. В последней строке кода осуществляются доступ к внутреннему значению и вывод его на печать.

Изменяемые переменные и ссылочные ячейки могут использоваться практически в одинаковых ситуациях; однако предпочтительнее применять изменяемые типы; если же компилятор их не допускает, то вместо них можно задействовать ссылочные ячейки. Например, в выражениях, которые генерируют замыкания, где требуется изменяемое состояние, компилятор сообщает, что изменяемая переменная не может быть использована. В этом случае ссылочная ячейка решает проблему.

Функции как типы первого класса

В F# функции являются типами данных первого порядка; они могут быть объявлены с применением ключевого слова `let` и использоваться точно так же, как любые другие переменные:

```
let square x = x * x
let plusOne x = x + 1
let isEven x = x % 2 = 0
```

Функции всегда возвращают значение, несмотря на отсутствие явного ключевого слова `return`. Возвращаемым значением является значение последнего оператора, выполненного в функции.

Компоновка: операторы конвейера и компоновки

Операторы конвейера (`|>`) и компоновки (`>>`) используются для объединения функций и аргументов в цепочку, чтобы код был более читаемым. Эти операторы позволяют гибко создавать *конвейеры* функций. Определение данных операторов очень простое:

```
let inline (|>) x f = f x
let inline (>>) f g x = g(f x)
```

В следующем примере показано, как воспользоваться преимуществами этих операторов для построения функционального конвейера:

```
let squarePlusOne x = x |> square |> plusOne
let plusOneIsEven = plusOne >> isEven
```

В последней строке кода оператор компоновки (`>>`) позволяет избежать потребности явно указывать входной параметр. Компилятор F# понимает, что функция `plusOneIsEven` ожидает на входе целое число. Функция, которой не нужны определения параметров, называется *бесточечной функцией*.

Основными различиями между операторами конвейера (`|>`) и компоновки (`>>`) являются их сигнатура и использование. Оператор конвейера принимает функции и аргументы, а оператор компоновки — компонует функции.

Делегаты

В .NET *делегат* — это указатель на функцию, переменная, которая хранит ссылку на метод с одинаковой, общей для всех делегатов сигнатурой. В F# вместо функций используются их значения, но в F# реализована поддержка делегатов для взаимодействия с .NET API. Вот синтаксис F# для определения делегата:

```
type delegate-typeName = delegate of typeA -> typeB
```

В следующем коде показан синтаксис для создания делегата с сигнатурой, которая соответствует операции сложения:

```
type MyDelegate = delegate of (int * int) -> int
let add (a, b) = a + b
let addDelegate = MyDelegate(add)
let result = addDelegate.Invoke(33, 9)
```

В этом примере функция `add` F# передается непосредственно в качестве аргумента конструктору делегата `MyDelegate`. Делегаты могут быть присоединены к значениям функций F#, к статическим методам или методам экземпляра. В делегате `addDelegate` метод `Invoke` вызывает внутреннюю функцию `add`.

Комментарии

В F# используются три вида комментариев: блочные комментарии размещаются между символами (* и *); строчные комментарии начинаются с символов // и продолжаются до конца строки; XML-комментарии к документу идут после символов ///, что позволяет применять XML-теги для формирования документации к коду на основе файла, сгенерированного компилятором. Вот как это выглядит:

```
(* Блочный комментарий *)
// Однострочный комментарий с использованием двойной косой черты в начале строки
/// Этот комментарий будет использован при формировании документации
```

Оператор open

Ключевое слово `open` используется для того, чтобы открыть пространство имен или модуль, подобно операторам C#. Следующий код открывает пространство имен `System`: `open System`.

Основные типы данных

В табл. Б.1 представлен список *основных типов* в F#.

Таблица Б.1. Основные типы данных

Тип F#	Тип .NET	Размер в байтах	Диапазон	Пример
sbyte	System.SByte	1	от -128 до 127	42y
byte	System.Byte	1	от 0 до 255	42uy
int16	System.Int16	2	от -32 768 до 32 767	42s
uint16	System.UInt16	2	от 0 до 65 535	42us
int / int32	System.Int32	4	от -2 147 483 648 до 2 147 483 647	42
uint32	System.UInt32	4	от 0 до 4 294 967 295	42u
int64	System.Int64	8	от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	42L
uint64	System.UInt64	8	0 до 18 446 744 073 709 551 615	42UL
float32	System.Single	4	±1,5e-45 до ±3,4e38	42.0F
float	System.Double	8	±5,0e-324 до ±1,7e308	42.0
decimal	System.Decimal	16	±1,0e-28 до ±7,9e28	42.0M
char	System.Char	2	U+0000 до U+ffff	'x'
string	System.String	20+ (2 * длина строки)	0 до примерно 2 млрд символов	"Hello World"
bool	System.Boolean	1	Только два возможных значения: true или false	true

Специальное определение строки

В F# тип `string` является псевдонимом для типа `System.String`; но в придачу к традиционной семантике .NET есть дополнительный способ объявления строк с помощью тройных кавычек. Это дополнительное определение строки позволяет объявить ее без необходимости экранирования специальных символов, как в случае дословного определения. В следующем примере показано, как на F# одна и та же строка определяется стандартным способом и с помощью тройных кавычек для экранирования специальных символов:

```
let verbatimHtml = @"<input type="submit" value="Submit">"  
let tripleHTML = """<input type="submit" value="Submit">"""
```

Кортежи

Кортеж — это группа неименованных упорядоченных значений, которые могут быть разных типов. Кортежи полезны для создания специальных структур данных и являются удобным способом создания функции, возвращающей несколько

значений. Кортеж определяется как набор значений, разделенных запятыми. Вот пример построения кортежей:

```
let tuple = (1, "Hello")
let tripleTuple = ("one", "two", "three")
```

Кортеж также может быть деконструирован. В следующем примере значения кортежей 1 и "Hello" связаны с идентификаторами `a` и `b` соответственно, а функция `swap` меняет местами значения в заданном кортеже (`a`, `b`):

```
let (a, b) = tuple
let swap (a, b) = (b, a)
```

Кортежи являются обычными объектами, но они также могут быть определены как структуры значений, как показано ниже:

```
let tupleStruct = struct (1, "Hello")
```

Обратите внимание, что вывод типа F# может автоматически параметризовать функцию, чтобы получить параметризованный тип. Это означает, что кортежи работают с любыми типами. В следующем примере можно получить доступ и получить первый и второй элементы кортежа с помощью функций `fst` и `snd`:

```
let one = fst tuple
let hello = snd tuple
```

Записи

Запись аналогична кортежу, за исключением того, что ее поля являются именованными и представлены в виде списка, элементы которого разделены точкой с запятой. Поскольку кортежи предоставляют только один метод хранения потенциально разнородных данных в одном контейнере, может оказаться затруднительным интерпретировать назначение элементов, если их будет слишком много. В этом случае тип записи помогает интерпретировать назначение данных, помечая их определения именами. Тип записи явно определяется с помощью ключевого слова `type` и компилируется в неизменяемый, публичный и закрытый класс .NET. Кроме того, компилятор автоматически генерирует функциональность структурного равенства и сравнения, а также предоставляет используемый по умолчанию конструктор, который заполняет все поля, содержащиеся в записи.

ПРИМЕЧАНИЕ

Если запись помечена атрибутом `CLIMutable`, то она будет содержать не имеющий аргументов конструктор для использования по умолчанию в других языках .NET.

В следующем примере показано, как определить и создать новый тип записи:

```
type Person = { FirstName : string; LastName : string; Age : int }
let fred = { FirstName = "Fred"; LastName = "Flintstone"; Age = 42 }
```

К записи можно добавить свойства и методы:

```
type Person with
    member this.FullName = sprintf "%s %s" this.FirstName this.LastName
```

Записи являются неизменяемыми типами. Это означает, что их экземпляры не могут быть изменены. Но существует удобный механизм клонирования записей посредством семантики клонирования `with`:

```
let olderFred = { fred with Age = fred.Age + 1 }
```

Тип записи также может быть представлен в виде структуры посредством атрибута [`<Struct>`]. Это полезно в ситуациях, когда производительность критически важна — важнее, чем гибкость ссылочных типов:

```
[<Struct>]
type Person = { FirstName : string; LastName : string; Age : int }
```

Размеченные объединения

Размеченное объединение (Discriminated Union, DU) — это тип, представляющий собой набор значений, относящийся к одному из нескольких четко определенных вариантов, каждый из которых может иметь различные значения и типы. В объектно-ориентированной парадигме размеченное объединение можно рассматривать как набор классов, унаследованных от общего базового класса. В общем случае DU — это инструмент, используемый для построения сложных структур данных, для создания моделей предметной области и для представления рекурсивных структур, таких как тип данных Tree.

В следующем коде представлены масть и достоинство игральной карты:

```
type Suit = Hearts | Clubs | Diamonds | Spades

type Rank =
    | Value of int
    | Ace
    | King
    | Queen
    | Jack
static member GetAllRanks() =
    [ yield Ace
        for i in 2 .. 10 do yield Value i
        yield Jack
        yield Queen
        yield King ]
```

Как видим, размеченные объединения могут быть расширены посредством свойств и методов. Список, представляющий все карты в колоде, может быть вычислен следующим образом:

```
let fullDeck =
    [ for suit in [ Hearts; Diamonds; Clubs; Spades] do
        for rank in Rank.GetAllRanks() do
            yield { Suit=suit; Rank=rank } ]
```

Кроме того, размеченные объединения также могут быть представлены в виде структур с атрибутом [`<Struct>`].

Сопоставление с образцом

Сопоставление с образцом — это языковая конструкция, которая позволяет компилятору интерпретировать определение типа данных и применять к нему ряд условий. Таким образом компилятор заставляет программиста писать конструкции сопоставления с образцом, покрывая все возможные случаи соответствия заданному значению. Этот прием известен как *полное сопоставление с образцом*. Конструкции сопоставления с образцом используются для управления потоком. Концептуально они похожи на серию операторов `if/then` или `case/switch`, но гораздо более мощные. Такие конструкции позволяют при каждом сопоставлении разбивать структуры данных на составляющие их компоненты, а затем выполнять для этих значений те или иные вычисления. Во всех языках программирования под *управлением потоком* понимают принимаемые в коде решения, которые влияют на последовательность выполнения операторов в приложении.

В общем случае наиболее распространенные образцы включают в себя алгебраические типы данных, такие как размеченные объединения, записи и коллекции. В следующем примере представлены две реализации игры Fizz Buzz (https://en.wikipedia.org/wiki/Fizz_buzz). Первая конструкция сопоставления с образцом содержит набор условий для проверки выполнения функции `divisibleBy`. Если какое-либо условие является истинным либо ложным, вторая реализация использует предложение `when`, называемое `guard`, чтобы описать и интегрировать дополнительные тесты, которые при соответствии образцу должны быть успешными:

```
let fizzBuzz n =
    let divisibleBy m = n % m = 0
    match divisibleBy 3,divisibleBy 5 with
    | true, false -> "Fizz"
    | false, true -> "Buzz"
    | true, true -> "FizzBuzz"
    | false, false -> sprintf "%d" n

let fizzBuzz n =
    match n with
    | _ when (n % 15) = 0 -> "FizzBuzz"
    | _ when (n % 3) = 0 -> "Fizz"
    | _ when (n % 5) = 0 -> "Buzz"
    | _ -> sprintf "%d" n

[1..20] |> List.iter fizzBuzz
```

При выполнении конструкции сопоставления с образцом выражение передается в `match <expression>`, где оно проверяется для каждого образца до обнаружения первого соответствия. Затем выполняется соответствующее тело функции.

Символ `_` (подчеркивание) известен как знак подстановки, который является одним из способов всегда обеспечить соответствие. Часто этот образец используется в качестве заключительного предложения, которому соответствуют все варианты и который описывает стандартное поведение.

Активные образцы

Активные образцы – это конструкции, которые расширяют возможности сопоставления с образцами, позволяя разбивать и деконструировать заданную структуру данных, чем обеспечивают гибкость преобразования и извлечение внутренних значений, делая код более удобным для чтения, а результаты декомпозиции – доступными для дальнейшего сопоставления с образцами.

Кроме того, активные образцы позволяют оберывать произвольные значения в структуру данных DU, что упрощает сопоставление с образцом. Объект, обернутый в активный образец, можно использовать в сопоставлении с образцом так же легко, как и любой другой тип объединения.

Иногда активные образцы не генерируют значение; в этом случае они называются *частичными активными образцами* и приводят к типу, который является вариантическим. Для того чтобы определить частичный активный образец, используют подстановочный символ нижнего подчеркивания (`_`) в конце списка образцов, заключенный в скобки с вертикальной чертой (`| |`). Такие скобки создаются с помощью символов круглых скобок и символов вертикальной черты. Вот как выглядит типичный частичный активный образец:

```
let (|DivisibleBy|_|) divideBy n =
    if n % divideBy = 0 then Some DivisibleBy else None
```

В этом частичном активном образце, если значение `n` кратно значению `DivBy`, возвращается значение типа `Some()`, что указывает на совпадение с активным образцом. В противном случае возвращается тип `None`, который указывает, что сопоставление с образцом было неудачным, и выполняется переход к следующему выражению для сопоставления. Частичные активные образцы используются для разделения и сопоставления только фрагмента пространства *входных данных*. В следующем коде показано, как выполнить сопоставление с частичным активным образцом:

```
let fizzBuzz n =
    match n with
    | DivisibleBy 3 & DivisibleBy 5 -> "FizzBuzz"
    | DivisibleBy 3 -> "Fizz"
    | DivisibleBy 5 -> "Buzz"
    | _ -> sprintf "%d" n

[1..20] |> List.iter fizzBuzz
```

Эта функция использует частичный активный шаблон (`| DivisibleBy | _ |`) для проверки входного значения `n`. Если оно делится на 3 и 5, то выбирается

первый вариант. Если оно делится только на 3, то выбирается второй, и т. д. Обратите внимание, что оператор & позволяет запускать более одного шаблона для одного аргумента.

Другим типом активного образца является *параметризованный активный образец*. Он похож на частичный активный образец, но принимает в качестве входных данных один или несколько дополнительных аргументов.

Более интересным является *многовариантный активный образец*, который разбивает все пространство входных данных на различные структуры данных в формате размеченного объединения. Ниже показан пример FizzBuzz, реализованный с использованием многовариантных активных образцов:

```
let (|Fizz|Buzz|FizzBuzz|Val|) n =
    match n % 3, n % 5 with
    | 0, 0 -> FizzBuzz
    | 0, _ -> Fizz
    | _, 0 -> Buzz
    | _ -> Val n
```

Поскольку активные образцы преобразуют данные из одного типа в другой, они отлично подходят для преобразования и валидации данных. Существует четыре взаимосвязанных варианта активных образцов: одновариантные, частичные, многовариантные и частично параметризованные. Подробнее об активных образцах читайте в документации MSDN (<http://mng.bz/ltmw>) и книге Исаака Абрахама (Isaac Abraham) *Get Programming with F#* (Manning, 2018).

Коллекции

F# поддерживает стандартные коллекции .NET, такие как массивы и последовательности (*IEnumerable*). Кроме того, он предлагает набор неизменяемых функциональных коллекций: списки, множества и словари.

Массивы

Массивы — это изменяемые коллекции с нулевой базой и фиксированным количеством элементов одного типа. Массивы компилируются как непрерывный блок памяти и поэтому обеспечивают быстрый произвольный доступ к элементам. Ниже показаны разные способы создания, фильтрации и проецирования массива:

```
let emptyArray= Array.empty
let emptyArray = [| |]
let arrayOfFiveElements = [| 1; 2; 3; 4; 5 |]
let arrayFromTwoToTen= [| 2..10 |]
let appendTwoArrays = emptyArray |> Array.append arrayFromTwoToTen
let evenNumbers = arrayFromTwoToTen |> Array.filter(fun n -> n % 2 = 0)
let squareNumbers = evenNumbers |> Array.map(fun n -> n * n)
```

Чтение и изменение элементов массива выполняется с помощью оператора точки (.) и скобок []:

```
let arr = Array.init 10 (fun i -> i * i)
arr.[1] <- 42
arr.[7] <- 91
```

Массивы также можно создавать в других синтаксисах, используя функции из модуля `Array`:

```
let arrOfBytes = Array.create 42 0uy
let arrOfSquare = Array.init 42 (fun i -> i * i)
let arrOfIntegers = Array.zeroCreate<int> 42
```

Последовательности (seq)

Последовательности представляют собой наборы элементов одного типа. В отличие от типа `List` последовательности вычисляются ленивым методом. Это означает, что элементы последовательности могут вычисляться по требованию (только когда они необходимы, что обеспечивает лучшую производительность, по сравнению со списками, в тех случаях, когда нужны не все элементы). Ниже показаны разные способы создания, фильтрации и проецирования последовательностей:

```
let emptySeq = Seq.empty
let seqFromTwoToFive = seq { yield 2; yield 3; yield 4; yield 5 }
let seqOfFiveElements = seq { 1 .. 5 }
let concatenateTwoSeqs = emptySeq |> Seq.append seqOfFiveElements
let oddNumbers = seqFromTwoToFive |> Seq.filter(fun n -> n % 2 <> 0)
let doubleNumbers = oddNumbers |> Seq.map(fun n -> n + n)
```

В последовательностях может использоваться ключевое слово `yield`, позволяющее вычислять и возвращать значение, которое становится частью последовательности.

Списки

В F# коллекции `List` представляют собой неизменяемые односвязные списки элементов одного типа. В общем случае списки хорошо подходят для описания перечислений, но их не рекомендуется использовать для произвольного доступа и объединения в тех случаях, когда критична производительность. Списки определяются посредством синтаксиса [...]. Ниже представлено несколько примеров создания, фильтрации и отображения списков:

```
let emptyList = List.empty
let emptyList = []
let listOfFiveElements = [ 1; 2; 3; 4; 5 ]
let listFromTwoToTen = [ 2..10 ]
let appendOneToEmptyList = 1::emptyList
```

```
let concatenateTwoLists = listOfFiveElements @ listFromTwoToTen
let evenNumbers = listOfFiveElements |> List.filter(fun n -> n % 2 = 0)
let squareNumbers = evenNumbers |> List.map(fun n -> n * n)
```

Скобки ([]) и разделители в виде точки с запятой (;) используются для добавления нескольких элементов в список; символ :: — для добавления одного элемента, а оператор «собака» (@) — для объединения двух существующих списков.

Множества

Множества — это коллекции, основанные на бинарных деревьях, элементы которых принадлежат к одному типу. В множествах порядок вставки элементов не сохраняется, а дубликаты не допускаются. Множества являются неизменяемыми, и каждая операция по изменению их элементов создает новое множество. Вот несколько способов создать множество:

```
let emptySet = Set.empty<int>
let setWithOneItem = emptySet.Add 8
let setFromList = [ 1..10 ] |> Set.ofList
```

Словари

Словари — это коллекции неизменяемых пар «ключ — значение», состоящие из элементов одного типа. Такие коллекции связывают значения с ключом и ведут себя как тип *Set*, который не допускает дубликатов и не учитывает порядок вставки элементов. В следующем примере показаны разные способы создания экземпляров словарей:

```
let emptyMap = Map.empty<int, string>
let mapWithOneItem = emptyMap.Add(42, "the answer to the meaning of life")
let mapFromList = [ (1, "Hello"), (2, "World") ] |> Map.ofSeq
```

Циклы

F# поддерживает конструкции циклов для перебора перечисляемых коллекций, таких как списки, массивы, последовательности, словари и т. п. Выражение *while ... do* выполняет итеративное выполнение, пока истинно заданное условие:

```
let mutable a = 10
while (a < 20) do
    printfn "value of a: %d" a
    a <- a + 1
```

Выражение *for ... to* повторяется в цикле для набора значений переменной цикла:

```
for i = 1 to 10 do
    printf "%d " i
```

Выражение `for ... in` повторяется в цикле для каждого элемента из коллекции значений:

```
for i in [1..10] do
    printfn "%d" i
```

Классы и наследование

Как уже отмечалось, F#, как и другие языки программирования .NET, поддерживает конструкции ООП. Фактически на F# можно создавать объекты классов для моделирования реальной предметной области. Ключевое слово `type`, используемое в F# для объявления класса, позволяет описывать свойства, методы и поля. В следующем коде показан пример определения подкласса `Student`, унаследованного от класса `Person`:

```
type Person(firstName, lastName, age) =
    member this.FirstName = firstName
    member this.LastName = lastName
    member this.Age = age

    member this.UpdateAge(n:int) =
        Person(firstName, lastName, age + n)

    override this.ToString() =
        sprintf "%s %s" firstName lastName

type Student(firstName, lastName, age, grade) =
    inherit Person(firstName, lastName, age)

    member this.Grade = grade
```

Свойства `FirstName`, `LastName` и `Age` представлены в виде полей; метод `UpdateAge` возвращает новый объект `Person` с измененным свойством `Age`. Чтобы изменить стандартное поведение методов, унаследованное от базового класса, нужно воспользоваться ключевым словом `override`. В этом примере базовый метод `ToString` переопределен и возвращает полное имя.

Объект `Student` является подклассом, определенным с применением ключевого слова `inherit`, его свойства унаследованы от базового класса `Person`, а также добавлено собственное свойство `Grade`.

Абстрактные классы и наследование

Абстрактный класс – это объект, который предоставляет шаблон для определения классов. Обычно абстрактный класс предоставляет одну или несколько неполных реализаций методов или свойств и требует создания подклассов для дополнения этих реализаций. Но можно определить поведение по умолчанию, которое потом

можно будет переопределить. В следующем примере абстрактный класс `Shape` определяет классы `Rectangle` и `Circle`:

```
[<AbstractClass>]
type Shape(weight :float, height :float) =
    member this.Weight = weight
    member this.Height = height
    abstract member Area : unit -> float
    default this.Area() = weight * height

type Rectangle(weight :float, height :float) =
    inherit Shape(weight, height)

type Circle(radius :float) =
    inherit Shape(radius, radius)
    override this.Area() = radius * radius * Math.PI
```

Атрибут `AbstractClass` уведомляет компилятор о том, что в классе есть абстрактные члены. В классе `Rectangle` применяется реализация метода `Area` по умолчанию, а в классе `Circle` она переопределена и заменена на пользовательское поведение.

Интерфейсы

Интерфейс — это контракт для определения деталей реализации класса. Однако в объявлении интерфейса его элементы не реализованы. Интерфейс представляет собой абстрактный способ обращения к описанным в нем открытым членам и функциям. В F# при определении интерфейса его члены объявляются с использованием ключевого слова `abstract`, после которого следует их сигнатура типа:

```
type IPerson =
    abstract FirstName : string
    abstract LastName : string
    abstract FullName : unit -> string
```

Доступ к методам интерфейса, реализованным в классе, осуществляется не через экземпляр класса, а через определение интерфейса. Таким образом, для вызова метода интерфейса к классу применяется операция приведения типа с использованием оператора `:>` («вверх»):

```
type Person(firstName : string, lastName : string) =
    interface IPerson with
        member this.FirstName = firstName
        member this.LastName = lastName
        member this.FullName() = sprintf "%s %s" firstName lastName

let fred = Person("Fred", "Flintstone")

(fred :> IPerson).FullName()
```

Объектные выражения

Интерфейсы — это полезная реализация кода, который может совместно использоваться разными частями программы. Но написание определения для специализированного интерфейса, реализованного посредством создания новых классов, иногда представляет собой громоздкую и обременительную работу. Решением является применение *объектных выражений*, которые позволяют реализовывать интерфейсы по ходу выполнения программы посредством анонимных классов. Вот пример создания нового объекта, который реализует интерфейс `IDisposable`, позволяющий применить цвет в консоли, а затем вернуться к оригиналу:

```
let print color =
    let current = Console.ForegroundColor
    Console.ForegroundColor <- color
    { new IDisposable with
        member x.Dispose() =
            Console.ForegroundColor <- current
    }

using(print ConsoleColor.Red) (fun _ -> printf "Hello in red!!")
using(print ConsoleColor.Blue) (fun _ -> printf "Hello in blue!!")
```

Приведение типов

Преобразование примитивного значения в тип объекта называется *упаковкой*, применяемой с помощью функции `box`. Эта функция приводит любой тип к типу .NET `System.Object`, название которого в F# сокращено до имени `obj`.

Функция *приведения типа вверх* применяет преобразование «вверх» для иерархии классов и интерфейсов, при котором происходит переход от данного класса к его наследнику. Синтаксис этого преобразования — `expr :> type`. Успешность преобразования проверяется на этапе компиляции.

Функция *приведения типа вниз* применяется для преобразования, которое идет вниз по иерархии классов или интерфейсов: например, от интерфейса к реализованному классу. Синтаксис такого приведения — `expr :?> type`, где вопросительный знак внутри оператора предполагает, что операция может завершиться с ошибкой `InvalidCastException`. Перед использованием приведения вниз можно безопасно сравнивать и тестировать тип с помощью оператора проверки типа `:?`, который эквивалентен оператору `is` в C#. *Выражение сопоставления* истинно, если значение соответствует заданному типу; в противном случае оно ложно:

```
let testPersonType (o:obj) =
    match o with
    | o :? IPerson -> printfn "this object is an IPerson"
    | _ -> printfn "this is not an IPerson"
```

Единицы измерения

Единицы измерения (Units of Measure, UoM) — это уникальная особенность системы типов F#, которая позволяет определять контекст и снабдить числовые константы аннотациями в виде статически типизированных метаданных. Это удобный способ манипулирования числами, которые соответствуют определенным единицам измерения, таким как метры, секунды, фунты и т. п. Система типов F# сразу проверяет корректность использования единиц измерения, чтобы исключить ошибки на стадии выполнения. Например, компилятор F# выдаст ошибку, если вместо типа `float<m/sec>` будет применено `float<mil>`. Кроме того, можно связать функцию с определенными единицами измерения, если она работает не просто с числовыми литералами, а с величинами, измеряемыми в данных единицах. В следующем коде показано, как создать значения, измеряемые в метрах (m) и секундах (sec) UoM, а затем выполнить операцию вычисления скорости:

```
[<Measure>]
type m

[<Measure>]
type sec

let distance = 25.0<m>
let time = 10.0<sec>
let speed = distance / time
```

Краткая справка по API модуля событий

Модуль событий содержит функции управления потоками событий. В табл. Б.2 приводится справочная информация об API из документации MSDN, доступной онлайн (<http://mng.bz/a0hG>).

Таблица Б.2. API. Краткая информация

Функция	Описание
Сложение: ('T -> unit) -> Event<'Del,'T> -> unit	Выполняет функцию каждый раз, когда возникает событие
Выбор: ('T -> 'U option) -> IEvent<'Del,'T> -> IEvent<'U>	Возвращает новое событие, которое возникает при выборе сообщений из исходного события. Функция выбора включает исходное сообщение в необязательное новое сообщение
Фильтрация: ('T -> bool) -> IEvent<'Del,'T> -> IEvent<'T>	Возвращает новое событие, которое прослушивает исходное событие и генерирует результирующее событие только в том случае, если аргумент, переданный событию, содержит заданную функцию

Функция	Описание
Отображение: $('T \rightarrow 'U) \rightarrow IEvent<'Del, 'T> \rightarrow IEvent<'U>$	Возвращает новое событие, которое передает значения, преобразованные заданной функцией
Слияние: $IEvent<'Del1, 'T> \rightarrow IEvent<'Del2, 'T> \rightarrow IEvent<'T>$	Создает выходное событие при возникновении любого из входных событий
Попарное объединение: $IEvent<'Del, 'T> \rightarrow IEvent<'T * 'T>$	Возвращает новое событие, которое срабатывает при втором и последующих срабатываниях входного события. N-е срабатывание входного события передает аргументы N – 1-го и N-го срабатываний в виде пары. Аргумент, переданный для N – 1-го срабатывания, остается в скрытом внутреннем состоянии до тех пор, пока не произойдет N-е срабатывание
Разделение: $('T \rightarrow bool) \rightarrow Ievent<'Del, 'T> \rightarrow IEvent<'T> * IEvent<'T>$	Возвращает пару событий, которые прослушивают исходное событие. Когда срабатывает исходное событие, запускается первое или второе событие пары, в зависимости от логики функции
Сканирование: $('U \rightarrow 'T \rightarrow 'U) \rightarrow 'U \rightarrow IEvent<'Del, 'T> \rightarrow IEvent<'U>$	Возвращает новое событие, состоящее из результатов применения данной аккумулирующей функции к последовательным значениям, сработавшим во входном событии. В переменную внутреннего состояния записывается текущее значение параметра состояния. Внутреннее состояние не блокируется во время выполнения аккумулирующей функции, поэтому следует позаботиться о том, чтобы вход IEvent не запускался несколькими потоками одновременно
Разбиение: $('T \rightarrow Choice<'U1, 'U2>) \rightarrow Ievent<'Del, 'T> \rightarrow IEvent<'U1> * IEvent<'U2>$	Возвращает новое событие, которое прослушивает исходное и генерирует первое результирующее событие, если функция, выполненная для аргументов исходного события, возвратит Choice1Of2, и второе событие, если функция возвратит Choice2Of2

Дополнительная информация

Для получения более подробной информации и более глубокого изучения F# я рекомендую книгу Исаака Абрахама *Get Programming with F#: A Guide for .NET Developers* (Manning, 2018, www.manning.com/books/get-programming-with-f-sharp).

Совместимость асинхронного рабочего процесса F# и задач .NET

Несмотря на сходство моделей асинхронного программирования, предоставляемых языками программирования C# и F#, обеспечить их совместимость было непросто. В F#-программах обычно используется больше асинхронных вычислительных выражений, чем задач .NET. Эти типы похожи, но имеют семантические различия, как было показано в главах 7 и 8. Например, задачи запускаются сразу после их создания, а F# `Async` необходимо запускать явно.

Как организовать взаимодействие между асинхронными вычислительными выражениями F# и .NET Task? Можно применять функции F#, такие как `Async.StartAsTask<T>` и `Async.AwaitTask<T>`, чтобы построить взаимодействие с библиотекой C#, которая возвращает или ожидает тип `Task`.

Обратное неверно: не существует эквивалентных методов для преобразования F# `Async` в тип `Task`. Было бы полезно использовать в C# встроенные вычисления F# `Async.Parallel`. В листинге B.1, продублированном из главы 9, функция F# `downloadMediaAsyncParallel` асинхронно и параллельно загружает изображения из хранилища Azure Blob.

Листинг B.1. Асинхронная параллельная функция загрузки изображений из хранилища Azure Blob

```
let downloadMediaAsyncParallel containerName = async {
    let storageAccount = CloudStorageAccount.Parse(azureConnection)
    let blobClient = storageAccount.CreateCloudBlobClient()
    let container = blobClient.GetContainerReference(containerName)
    let computations =
        container.ListBlobs()
        |> Seq.map(fun blobMedia -> async {
            let blobName = blobMedia.Uri.Segments.
                [blobMedia.Uri.Segments.Length - 1]
            let blockBlob = container.GetBlockBlobReference(blobName)
            use stream = new MemoryStream()
```

```
do! blockBlob.DownloadToStreamAsync(stream)
let image = System.Drawing.Bitmap.FromStream(stream)
return image }
return! Async.Parallel computations }
```

Параллельное выполнение нескольких асинхронных вычислений F#

Функция `downloadMediaAsyncParallel` возвращает значение типа `Async<Image []>`. Как уже отмечалось, как правило, сложно организовать взаимодействие между типом `Async` из F# и задачами (`async/await`) из кода C#; а также трудно заставить этот тип вести себя как задачи C#. В следующем фрагменте код C# запускает F#-функцию `downloadMediaAsyncParallel` как задачу с использованием оператора `Async.Parallel`:

```
var cts = new CancellationToken();
var images = await downloadMediaAsyncParallel("MyMedia").AsTask(cts);
```

Функциональная совместимость легко обеспечивается с помощью метода расширений `AsTask`. Решение, обеспечивающее совместимость, заключается в реализации служебного модуля F#, который предоставляет набор методов расширения, используемых в других языках .NET (листинг B.2).

Листинг B.2. Вспомогательные методы расширения, обеспечивающие совместимость объектов Task и асинхронного рабочего процесса

```
Экземпляр, который позволяет
контролировать выполнение в форме Task
```

Если маркер отмены не передан в качестве аргумента, то по умолчанию назначается контекстный маркер, который автоматически распространяется через асинхронный рабочий процесс

```
module private AsyncInterop =
    let asTask(async: Async<'T>, token: CancellationToken option) =
        let tcs = TaskCompletionSource<'T>()
```

Возвращение
TaskCompletionSource
для представления
поведения
в стиле Task

Начинается выполнение с продолжениями
для передачи контекста завершения
в конкретную функцию продолжения
в зависимости от того, было
выполнение успешным, неудачным
или же было отменено

Начинается выполнение с продолжений, чтобы
захватить текущий результат выполнения (успешное,
исключение или отмена) и чтобы выбрать для дальнейшего
выполнения одну из заданных функций продолжения

```
let asAsync(task: Task, token: CancellationToken option) =
    Async.FromContinuations(
        fun (completed, caught, canceled) ->
            let token = defaultArg token Async.CancellationToken
            Async.StartWithContinuations(async,
                tcs.SetResult, tcs.SetException,
                tcs.SetException, token)
            tcs.Task)
```

Продолжение
выполнения
в стиле
продолжений Task

Уведомление об успешном
завершении вычислений

```
let asAsyncT(task: Task<'T>, token: CancellationToken option) =
    Async.FromContinuations(
        fun (completed, caught, canceled) ->
            let token = defaultArg token Async.CancellationToken
```

```

Продолжение выполнения в стиле продолжений Task    task.ContinueWith(new Action<Task<'T>>(fun _ ->
if task.IsFaulted then caught(task.Exception)
else if task.IsCanceled then
    canceled(OperationCanceledException(token) |> raise)
else completed(task.Result)), token) |> ignore

Предоставление посредством метода расширения вспомогательных функций, которые могут использоваться в других языках программирования .NET
type AsyncInteropExtensions =
[<Extension>]
static member AsAsync (task: Task<'T>) = AsyncInterop.asAsyncT
    (task, None)

[<Extension>]
static member AsAsync (task: Task<'T>, token: CancellationToken) =
    AsyncInterop.asAsyncT (task, Some token)

[<Extension>]
static member AsTask (async: Async<'T>) = AsyncInterop.asTask
    (async, None)

[<Extension>]
static member AsTask (async: Async<'T>, token: CancellationToken) =
    AsyncInterop.asTask (async, Some token)
  
```

Уведомление об успешном завершении вычислений

Модуль `AsyncInterop` является приватным, но основные функции, обеспечивающие взаимодействие между типами `Async` в F# и `Task` в C#, предоставляются посредством типа `AsyncInteropExtensions`. Атрибут `Extension` обновляет методы как расширение, делая их доступными для других языков программирования .NET.

Метод `asTask` преобразует F#-тип `Async` в `Task`, начиная асинхронную операцию с помощью функции `Async.StartWithContinuations`. Внутри этой функции используется `TaskCompletionSource`, что позволяет возвращать экземпляр `Task`, поддерживающий состояние операции. Когда операция завершается, возвращаемое состояние может означать отмену, исключение или содержать фактический результат в случае успешного завершения.

ПРИМЕЧАНИЕ

Эти методы расширения встроены в F# для обеспечения доступа к асинхронному рабочему процессу, но модуль скомпилирован в виде библиотеки, на которую можно ссылаться и использовать ее в C#. Несмотря на то что этот код написан на F#, он предназначен для C#.

Функция `asAsync` служит для преобразования объектов `Task` в F#-тип `Async`. В этой функции применяется `Async.FromContinuations` для создания асинхрон-

ных вычислений, которые обеспечивают обратный вызов, выполняющий одно из заданных продолжений в случае успешного завершения, исключения или отмены.

Все данные функции принимают в качестве второго аргумента необязательный параметр `CancellationToken`, который можно использовать для остановки текущей операции. Если маркер не предоставлен, то по умолчанию в контексте назначается `DefaultCancellationToken`.

Эти функции обеспечивают совместимость между асинхронным шаблоном на основе задач (Task-based Asynchronous Pattern, TAP) из .NET TPL и моделью асинхронного программирования F#.

Рикардо Террелл

**Конкурентность и параллелизм на платформе .NET.
Паттерны эффективного проектирования**

Перевела с английского Е. Сандицкая

Заведующая редакцией	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>С. Давид</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Е. Рафаилок-Бузовская</i>
Художественный редактор	<i>В. Мостисан</i>
Корректоры	<i>О. Андриевич, Е. Павлович</i>
Верстка	<i>Г. Блинов</i>

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: 04.2019. 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 18.04.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 50,310. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87