8 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 время 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 APM, 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 (APM)

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, начинающейся 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 простаивают 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.

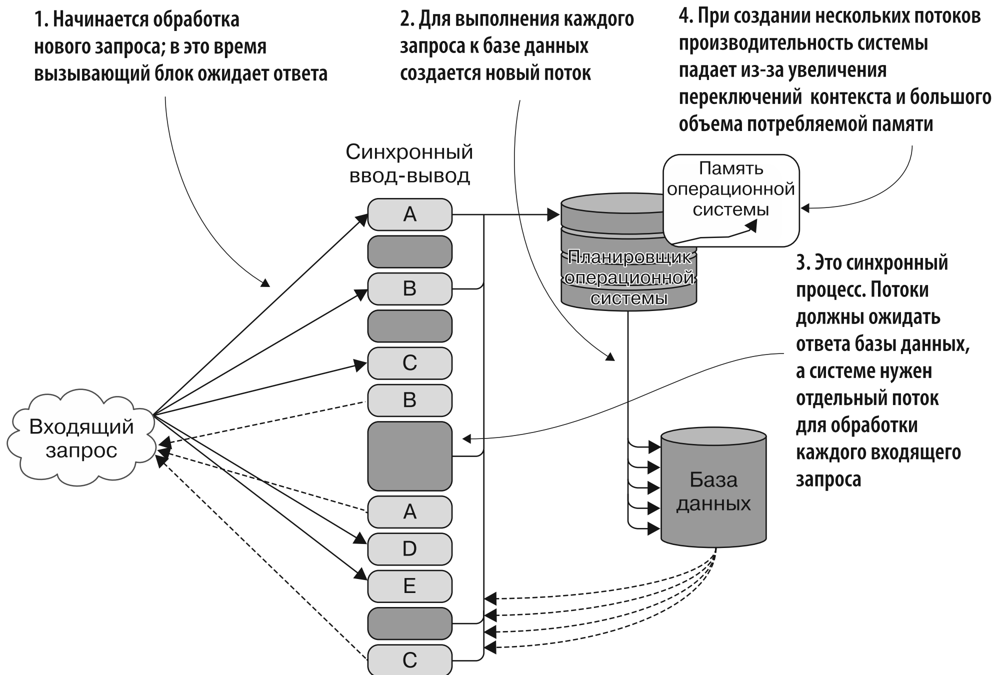

Рис. 8.1. Серверы, которые синхронно обрабатывают входящие запросы, не масштабируются

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.

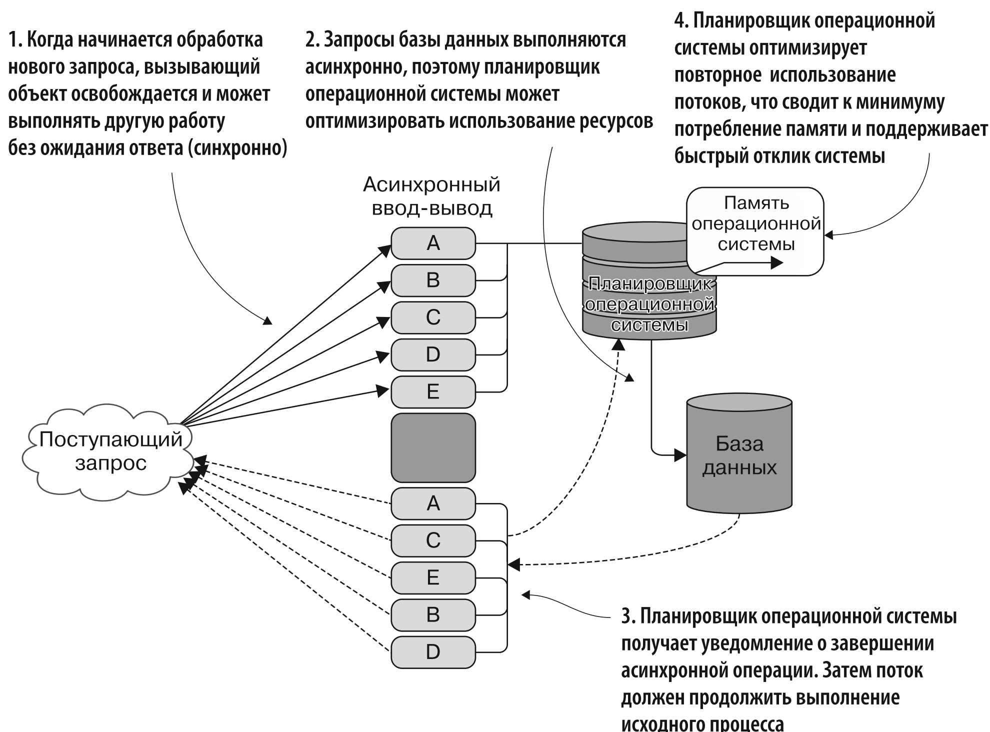

Рис. 8.2. Несколько операций асинхронного ввода-вывода могут запускаться параллельно без ограничений, а после завершения будут возвращаться к вызывающему блоку, что обеспечивает масштабируемость системы

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)

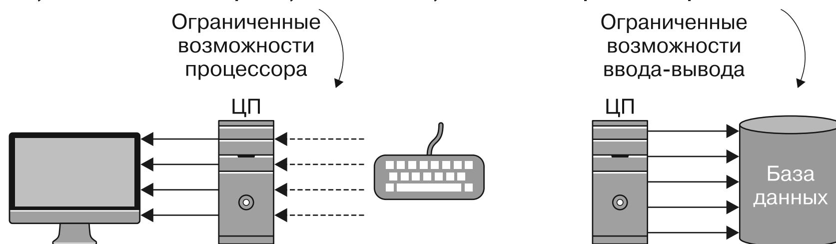

Рис. 8.3. Сравнение операций с ограничениями процессора и ввода-вывода

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 APM 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 четырехъядерной 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 сраз 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 {       ◄── Асинхронное чтение
    let req = WebRequest.Create(url)              содержимого сайта
    let! resp = req.AsyncGetResponse()
    use stream = resp.GetResponseStream()
    use reader = new StreamReader(stream)
    let! text = reader.ReadToEndAsync()
    return text }
                                              Список произвольных
let sites =                          ◄──      сайтов для загрузки
    [ "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"; ]

sites                                 Создание последовательности
|> Seq.map httpAsync     ◄──          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

APM 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, считывающая 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 позволят 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.Asynchronous)
     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);
}
```

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

Обратный вызов восстанавливает состояние в исходной форме для доступа к базовым значениям

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 APM труднее, чем их последовательные версии. APM-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 APM, — 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 APM 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.