

```

select
    new XElement ("customer", new XAttribute ("id", c.ID),
    new XElement ("name", c.Name),
    new XElement ("buys", c.Purchases.Count),
    lastBigBuy == null ? null :
        new XElement ("lastBigBuy",
            new XElement ("description", lastBigBuy.Description),
            new XElement ("price", lastBigBuy.Price))

```

Для заказчиков, не имеющих lastBigBuy, вместо пустого элемента XElement выдается значение null. Это именно то, что нужно, т.к. содержимое null попросту игнорируется.

Потоковая передача проекции

Если проецирование в модель X-DOM осуществляется только с целью его сохранения посредством метода Save (или вызова ToString), то эффективность использования памяти можно повысить, задействовав класс XStreamingElement. Класс XStreamingElement представляет собой усеченную версию XElement, которая применяет к своему дочернему содержимому семантику отложенной загрузки. Для его использования нужно просто заменить внешние элементы XElement элементами XStreamingElement:

```

var customers =
    new XStreamingElement ("customers",
        from c in dbContext.Customers
        select
            new XStreamingElement ("customer", new XAttribute ("id", c.ID),
                new XElement ("name", c.Name),
                new XElement ("buys", c.Purchases.Count)
            )
    );
customers.Save ("data.xml");

```

Запросы, переданные конструктору XStreamingElement, не перечисляются вплоть до вызова метода Save, ToString или WriteTo на элементе, что позволяет избежать загрузки в память сразу целого дерева X-DOM. Обратной стороной такого подхода является то, что запросы оцениваются повторно, требуя сохранения заново. Кроме того, обход дочернего содержимого XStreamingElement невозможен — данный класс не открывает доступ к методам вроде Elements или Attributes.

Класс XStreamingElement не основан на XObject (или на любом другом классе), поэтому он располагает таким ограниченным набором членов. В состав членов помимо Save, ToString и WriteTo входят:

- метод Add, который принимает содержимое подобно конструктору;
- свойство Name.

Класс XStreamingElement не позволяет читать содержимое в потоковой манере — в таком случае придется применять класс XmlReader в сочетании с X-DOM. Мы объясним, как это делать, в разделе “Шаблоны для использования XmlReader/XmlWriter” главы 11.



Другие технологии XML и JSON

В главе 10 был раскрыт API-интерфейс LINQ to XML и язык XML в целом. В настоящей главе мы исследуем низкоуровневые классы `XmlReader/XmlWriter` и типы для работы с форматом JSON (JavaScript Object Notation — запись объектов JavaScript), который стал популярной альтернативой XML.

В дополнительных материалах, доступных на веб-сайте издательства, описаны инструменты для работы со схемой XML и таблицами стилей.

XmlReader

`XmlReader` — высокопроизводительный класс для чтения XML-потока низкоуровневым односторонним способом.

Рассмотрим следующее содержимое XML-файла `customer.xml`:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

Для создания экземпляра `XmlReader` вызывается статический метод `XmlReader.Create`, которому передается объект `Stream`, объект `TextReader` или строка `URI`:

```
using XmlReader reader = XmlReader.Create ("customer.xml");
...
```



Поскольку класс `XmlReader` позволяет читать из потенциально медленных источников (`Stream` и `URI`), он предлагает асинхронные версии большинства своих методов, так что можно легко писать неблокирующий код. Асинхронность подробно обсуждается в главе 14.

Ниже показано, как сконструировать экземпляр XmlReader, который читает из строки:

```
using XmlReader reader = XmlReader.Create (
    new System.IO.StringReader (myString));
```

Для управления настройками разбора и проверки достоверности можно также передавать объект XmlReaderSettings. В частности, следующие три свойства XmlReaderSettings полезны при пропускании избыточного содержимого:

bool IgnoreComments	// Пропускать узлы комментариев?
bool IgnoreProcessingInstructions	// Пропускать инструкции обработки?
bool IgnoreWhitespace	// Пропускать пробельные символы?

В приведенном далее примере средству чтения сообщается о том, что узлы с пробельными символами, которые отвлекают внимание в типовых сценариях, выпускаться не должны:

```
XmlReaderSettings settings = new XmlReaderSettings ();
settings.IgnoreWhitespace = true;

using XmlReader reader = XmlReader.Create ("customer.xml", settings);
...
```

Еще одним полезным свойством XmlReaderSettings является ConformanceLevel. Его стандартное значение Document указывает средству чтения на то, что необходимо предполагать наличие допустимого XML-документа с единственным корневым узлом. Такая проблема возникает, когда нужно прочитать только внутреннюю порцию XML-кода, содержащую несколько узлов:

```
<firstname>Jim</firstname>
<lastname>Bo</lastname>
```

Чтобы прочитать такой XML-код без генерации исключения, потребуется установить ConformanceLevel в Fragment.

Класс XmlReaderSettings также имеет свойство по имени CloseInput, которое указывает на то, должен ли закрываться лежащий в основе поток, когда закрывается средство чтения (в XmlWriterSettings существует аналогичное свойство под названием CloseOutput). Стандартное значение для свойств CloseInput и CloseOutput равно false.

Чтение узлов

Единицами XML-потока являются узлы XML. Средство чтения перемещается по потоку в текстовом порядке (сначала в глубину). Свойство Depth средства чтения возвращает текущую глубину курсора.

Самый простой способ чтения из XmlReader предполагает вызов метода Read. Он осуществляет перемещение на следующий узел в XML-потоке подобно методу MoveNext из интерфейса IE enumerator. Первый вызов Read устанавливает курсор на первый узел. Когда метод Read возвращает false, это означает, что курсор переместился за последний узел, и в данном случае экземпляр XmlReader должен быть закрыт и освобожден.

Два строковых свойства в классе `XmlReader` предоставляют доступ к содержимому узла: `Name` и `Value`. В зависимости от типа узла заполняется либо `Name`, либо `Value` (или оба).

В следующем примере мы читаем каждый узел в XML-потоке, выводя тип узла по мере продвижения:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using XmlReader reader = XmlReader.Create ("customer.xml", settings);
while (reader.Read())
{
    Console.Write (new string (' ', reader.Depth * 2)); // Вывести отступ
    Console.WriteLine (reader.NodeType.ToString ());

    if (reader.NodeType == XmlNodeType.Element ||
        reader.NodeType == XmlNodeType.EndElement)
    {
        Console.WriteLine (" Name=" + reader.Name);
    }
    else if (reader.NodeType == XmlNodeType.Text)
    {
        Console.WriteLine (" Value=" + reader.Value);
    }
    Console.WriteLine ();
}
```

Ниже показан вывод:

```
XmlDeclaration
Element Name=customer
Element Name=firstname
    Text Value=Jim
EndElement Name=firstname
Element Name=lastname
    Text Value=Bo
EndElement Name=lastname
EndElement Name=customer
```



Атрибуты в обход на основе `Read` не включаются (см. раздел “Чтение атрибутов” далее в главе).

Свойство `NodeType` имеет тип `XmlNodeType`, который представляет собой перечисление со следующими членами:

None	Comment	Document
XmlDeclaration	Entity	DocumentType
Element	EndEntity	DocumentFragment
EndElement	EntityReference	Notation
Text	ProcessingInstruction	Whitespace
Attribute	CDATA	SignificantWhitespace

Чтение элементов

Зачастую структура читаемого XML-документа уже известна. Чтобы помочь в этом отношении, класс `XmlReader` предлагает набор методов, которые выполняют чтение, предполагая наличие определенной структуры. Они упрощают код и одновременно предпринимают некоторую проверку достоверности.



Класс `XmlReader` генерирует исключение `XmlException`, если любая проверка достоверности терпит неудачу. Класс `XmlException` имеет свойства `LineNumber` и `LinePosition`, которые указывают, где произошла ошибка — в случае крупных XML-файлов регистрация такой информации в журнале очень важна!

Метод `ReadStartElement` проверяет, что текущий `NodeType` является `Element`, и затем вызывает метод `Read`. Если указано имя, тогда он проверяет, совпадает ли оно с именем текущего элемента.

Метод `ReadEndElement` удостоверяется в том, что текущий `NodeType` — это `EndElement`, и затем вызывает метод `Read`.

Например, мы могли бы прочитать узел:

```
<firstname>Jim</firstname>
```

следующим образом:

```
reader.ReadStartElement ("firstname");
Console.WriteLine (reader.Value);
reader.Read();
reader.ReadEndElement();
```

Метод `ReadElementContentAsString` выполняет сразу все описанные ранее действия. Он читает начальный элемент, текстовый узел и конечный элемент, возвращая содержимое в виде строки:

```
string firstName = reader.ReadElementContentAsString ("firstname", "");
```

Второй аргумент ссылается на пространство имен, которое в приведенном примере оставлено пустым. Доступны также типизированные версии метода, такие как `ReadElementContentAsInt`, разбирающие результат. Вернемся к исходному XML-документу:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
  <creditlimit>500.00</creditlimit> <!--Да, мы не учитываем этот комментарий!-->
</customer>
```

Его можно прочитать следующим образом:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using XmlReader r = XmlReader.Create ("customer.xml", settings);
r.MoveToContent();                                // Пропустить XML-объявление
r.ReadStartElement ("customer");
```

```
string firstName      = r.ReadElementContentAsString ("firstname", "");  
string lastName       = r.ReadElementContentAsString ("lastname", "");  
decimal creditLimit = r.ReadElementContentAsDecimal ("creditlimit", "");  
  
r.MoveToContent();           // Пропустить этот надоедливый комментарий  
r.ReadEndElement();         // Прочитать закрывающий дескриптор customer
```



Метод `MoveToContent` по-настоящему удобен. Он пропускает все малоинтересное: XML-объявления, пробельные символы, комментарии и инструкции обработки. Посредством свойств `XmlReaderSettings` можно заставить средство чтения выполнять большинство таких действий автоматически.

Необязательные элементы

Предположим, что в предыдущем примере элемент `<lastname>` был необязательным. Решение прямолинейно:

```
r.ReadStartElement ("customer");  
string firstName      = r.ReadElementContentAsString ("firstname", "");  
string lastName       = r.Name == "lastname"  
    ? r.ReadElementContentAsString () : null;  
decimal creditLimit = r.ReadElementContentAsDecimal ("creditlimit", "");
```

Случайный порядок элементов

Примеры в текущем разделе полагаются на то, что элементы в XML-файле расположены в установленном порядке. Чтобы справиться с элементами, представленными в другом порядке, проще всего прочитать такой раздел XML-файла в дерево X-DOM. Мы покажем, как это делать, в разделе “Шаблоны для использования `XmlReader/XmlWriter`” далее в главе.

Пустые элементы

Способ, которым класс `XmlReader` обрабатывает пустые элементы, таит в себе серьезную ловушку. Рассмотрим следующий элемент:

```
<customerList></customerList>
```

Вот его эквивалент в XML:

```
<customerList/>
```

Тем не менее, `XmlReader` трактует два варианта по-разному. В первом случае приведенный ниже код работает ожидаемым образом:

```
reader.ReadStartElement ("customerList");  
reader.ReadEndElement();
```

Во втором случае метод `ReadEndElement` генерирует исключение, т.к. отсутствует отдельный “конечный элемент”, на который рассчитывает класс `XmlReader`. Обходной путь предусматривает добавление проверки на предмет пустых элементов:

```
bool isEmpty = reader.IsEmptyElement;  
reader.ReadStartElement ("customerList");  
if (!isEmpty) reader.ReadEndElement();
```

На самом деле такая неприятность возникает, только когда рассматриваемый элемент может содержать дочерние элементы (скажем, список заказчиков). В случае элементов, которые содержат простой текст (вроде `firstname`), проблемы можно избежать путем вызова такого метода, как `ReadElementContentAsString`. Методы `ReadElementXXX` корректно обрабатывают оба вида пустых элементов.

Другие методы `ReadXXX`

В табл. 11.1 приведена сводка по всем методам `ReadXXX` в классе `XmlReader`. Большинство из них предназначено для работы с элементами. Выделенная полужирным часть в примере XML-фрагмента — это раздел, который читает описываемый метод.

Таблица 11.1. Методы чтения

Методы	Типы узлов, на которых методы работают	Пример XML-фрагмента	Входные параметры	Возвращаемые данные
<code>ReadContentAsXXX</code>	Text	<code><a>x</code>		x
<code>ReadElementContentAsXXX</code>	Element	<code><a>x</code>		x
<code>ReadInnerXml</code>	Element	<code><a>x</code>		x
<code>ReadOuterXml</code>	Element	<code><a>x</code>		<code><a>x</code>
<code>ReadStartElement</code>	Element	<code><a>x</code>		
<code>ReadEndElement</code>	Element	<code><a>x</code>		
<code>ReadSubtree</code>	Element	<code><a>x</code>		<code><a>x</code>
<code>ReadToDescendant</code>	Element	<code><a>x</code>	"b"	
<code>ReadToFollowing</code>	Element	<code><a>x</code>	"b"	
<code>ReadToNextSibling</code>	Element	<code><a>x</code>	"b"	
<code>ReadAttributeValue</code>	Attribute	См. раздел “Чтение атрибутов” далее в главе		

Методы `ReadContentAsXXX` разбирают текстовый узел в тип `XXX`. Внутренне класс `XmlConvert` выполняет преобразование из строки в данный тип. Текстовый узел может находиться внутри элемента или атрибута.

Методы `ReadElementContentAsXXX` представляют собой оболочки вокруг соответствующих методов `ReadContentAsXXX`. Они применяются к узлу элемента, а не к текстовому узлу, заключенному в элемент.

Метод `ReadInnerXml` обычно применяется к элементу; он читает и возвращает элемент со всеми его потомками. В случае применения к атрибуту метод `ReadInnerXml` возвращает значение атрибута.

Метод `ReadOuterXml` аналогичен `ReadInnerXml`, но только включает, а не исключает элемент в позиции курсора.

Метод `ReadSubtree` возвращает новый экземпляр `XmlReader`, который обеспечивает представление лишь текущего элемента (и его потомков). Чтобы исходный `XmlReader` мог безопасно продолжить чтение, этот экземпляр должен быть закрыт. Когда новый экземпляр `XmlReader` закрывается, позиция курсора исходного `XmlReader` перемещается в конец поддерева.

Метод `ReadToDescendant` перемещает курсор в начало первого узла-потомка с указанным именем/пространством имен. Метод `ReadToFollowing` перемещает курсор в начало первого узла — независимо от глубины — с указанным именем/пространством имен. Метод `ReadToNextSibling` перемещает курсор в начало первого родственного узла с указанным именем/пространством имен.

Существуют также два унаследованных метода: `ReadString` и `ReadElementString`. Они ведут себя подобно методам `ReadContentAsString` и `ReadElementContentAsString`, но с тем отличием, что генерируют исключение, если внутри элемента обнаружено более одного текстового узла. Вы должны избегать использования унаследованных методов, т.к. они генерируют исключение, если элемент содержит комментарий.

Чтение атрибутов

Класс `XmlReader` предоставляет индексатор, обеспечивающий прямой (произвольный) доступ к атрибутам элемента — по имени или по позиции. Вызов метода `GetAttribute` эквивалентен применению индексатора.

Имея следующий XML-фрагмент:

```
<customer id="123" status="archived"/>
```

вот как мы могли бы прочитать его атрибуты:

```
Console.WriteLine (reader ["id"]);           // 123
Console.WriteLine (reader ["status"]);         // archived
Console.WriteLine (reader ["bogus"] == null);   // True
```



Для того чтобы читать атрибуты, экземпляр `XmlReader` должен располагаться на начальном элементе. После вызова метода `ReadStartElement` атрибуты исчезают навсегда!

Хотя порядок атрибутов семантически несуществен, доступ к атрибутам возможен по их порядковым позициям. Предыдущий пример можно переписать следующим образом:

```
Console.WriteLine (reader [0]);           // 123
Console.WriteLine (reader [1]);           // archived
```

Индексатор также позволяет указывать пространство имен атрибута, если оно имеется.

Свойство `AttributeCount` возвращает количество атрибутов для текущего узла.

Узлы атрибутов

Для явного обхода узлов атрибутов потребуется сделать специальное отклонение от нормального пути, предусматривающего просто вызов метода Read. Веской причиной поступить так является необходимость разбора значений атрибутов в другие типы с помощью методов ReadContentAsXXX.

Отклонение должно начинаться с начального элемента. В целях упрощения работы во время обхода атрибутов правило односторонности ослабляется: можно переходить к любому атрибуту (вперед или назад) за счет вызова метода MoveToAttribute.



Метод MoveToElement возвращает начальный элемент из любого места внутри ответвления узла атрибута.

Вернувшись к предыдущему примеру:

```
<customer id="123" status="archived"/>
```

можно поступить так:

```
reader.MoveToAttribute ("status");
string status = reader.ReadContentAsString();

reader.MoveToAttribute ("id");
int id = reader.ReadContentAsInt();
```

Метод MoveToAttribute возвращает false, если указанный атрибут не существует.

Можно также совершить обход всех атрибутов в последовательности, вызывая метод MoveToFirstAttribute, а затем метод MoveToNextAttribute:

```
if (reader.MoveToFirstAttribute())
    do
    {
        Console.WriteLine (reader.Name + "=" + reader.Value);
    }
    while (reader.MoveToNextAttribute());
```

Вот вывод:

```
id=123
status=archived
```

Пространства имен и префиксы

Класс XmlReader предлагает две параллельные системы для ссылки на имена элементов и атрибутов:

- Name;
- NamespaceURI и LocalName.

Всякий раз, когда читается свойство Name элемента или вызывается метод, принимающий одиночный аргумент name, используется первая система. Такой подход хорошо работает в отсутствие каких-либо пространств имен или префиксов; в противном случае он действует в грубой и буквальной манере.

Пространства имен игнорируются, а префиксы включаются в точности так, как они записаны. Ниже показаны примеры.

Пример фрагмента	Значение Name
<customer ...>	customer
<customer xmlns='blah' ...>	customer
<x:customer ...>	x:customer

Приведенный далее код работает с первыми двумя случаями:

```
reader.ReadStartElement ("customer");
```

Для обработки третьего случая требуется следующий код:

```
reader.ReadStartElement ("x:customer");
```

Вторая система работает через два свойства, осведомленные о пространствах имен: NamespaceURI и LocalName. Указанные свойства принимают во внимание префиксы и стандартные пространства имен, определенные родительскими элементами. Префиксы автоматически расширяются. Это означает, что свойство NamespaceURI всегда отражает семантически корректное пространство имен для текущего элемента, а свойство LocalName всегда свободно от префиксов.

При передаче двух аргументов имен в такой метод, как ReadStartElement, вы применяете ту же самую систему. Например, взгляните на следующий XML-фрагмент:

```
<customer xmlns="DefaultNamespace" xmlns:other="OtherNamespace">
<address>
  <other:city>
  ...

```

Прочитать его можно было бы так:

```
reader.ReadStartElement ("customer", "DefaultNamespace");
reader.ReadStartElement ("address", "DefaultNamespace");
reader.ReadStartElement ("city", "OtherNamespace");
```

Абстрагирование от префиксов обычно является именно тем, что нужно. При необходимости посредством свойства Prefix можно просмотреть, какой префикс использовался, и с помощью метода LookupNamespace преобразовать его в пространство имен.

XmlWriter

Класс XmlWriter — это одностороннее средство записи в XML-поток. Проектное решение, положенное в основу XmlWriter, симметрично таковому в классе XmlReader.

Как и XmlTextReader, экземпляр XmlWriter конструируется вызовом метода Create, которому передается необязательный объект настроек. В приведенном ниже примере мы разрешаем отступы, чтобы сделать вывод удобным для восприятия человеком, и затем записываем его в простой XML-файл:

```
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;

using XmlWriter writer = XmlWriter.Create ("foo.xml", settings);
writer.WriteStartElement ("customer");
writer.WriteString ("firstname", "Jim");
writer.WriteString ("lastname", "Bo");
writer.WriteEndElement();
```

В результате получается следующий документ (тот же самый, что и в файле, который мы читали в первом примере применения класса `XmlReader`):

```
<?xml version="1.0" encoding="utf-8"?>
<customer>
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

Класс `XmlWriter` автоматически записывает объявление в начале, если только в `XmlWriterSettings` не указано обратное за счет установки свойства `OmitXmlDeclaration` в `true` или свойства `ConformanceLevel` в `Fragment`. В последнем случае также разрешена запись нескольких корневых узлов — то, что иначе приводит к генерации исключения.

Метод `WriteValue` записывает одиночный текстовый узел. Он принимает строковые и нестроковые типы, такие как `bool` и `DateTime`, внутренне используя класс `XmlConvert` для выполнения совместимых с XML преобразований строк:

```
writer.WriteStartElement ("birthdate");
writer.WriteLine (DateTime.Now);
writer.WriteEndElement();
```

Напротив, если мы вызовем:

```
WriteElementString ("birthdate", DateTime.Now.ToString());
```

то результат окажется несовместимым с XML и уязвимым к некорректному разбору.

Вызов метода `WriteString` эквивалентен вызову метода `WriteValue` со строкой. Класс `XmlWriter` автоматически защищает символы, которые в противном случае были бы недопустимыми внутри атрибута либо элемента, такие как `&`, `<`, `>`, и расширенные символы Unicode.

Запись атрибутов

Атрибуты можно записывать немедленно после записи начального элемента:

```
writer.WriteStartElement ("customer");
writer.WriteAttributeString ("id", "1");
writer.WriteAttributeString ("status", "archived");
```

Для записи нестроковых значений нужно вызывать методы `WriteStartAttribute`, `WriteValue` и `WriteEndAttribute`.

Запись других типов узлов

В классе `XmlWriter` также определены следующие методы для записи других разновидностей узлов:

```
WriteBase64           // для двоичных данных  
WriteBinHex          // для двоичных данных  
WriteCData  
WriteComment  
WriteDocType  
WriteEntityRef  
WriteProcessingInstruction  
WriteRaw  
WriteWhitespace
```

Метод `WriteRaw` внедряет строку прямо в выходной поток. Имеется также метод `WriteNode`, который принимает экземпляр `XmlReader` и копирует из него все данные.

Пространства имен и префиксы

Перегруженные версии методов `Write*` позволяют ассоциировать элемент или атрибут с пространством имен. Давайте перепишем содержимое XML-файла из предыдущего примера. На этот раз мы будем связывать все элементы с пространством имен `http://oreilly.com`, объявив префикс `o` в элементе `customer`:

```
writer.WriteStartElement ("o", "customer", "http://oreilly.com");  
writer.WriteString ("o", "firstname", "http://oreilly.com", "Jim");  
writer.WriteString ("o", "lastname", "http://oreilly.com", "Bo");  
writer.WriteEndElement();
```

Вывод теперь выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>  
<o:customer xmlns:o='http://oreilly.com'>  
  <o:firstname>Jim</o:firstname>  
  <o:lastname>Bo</o:lastname>  
</o:customer>
```

Обратите внимание, что для краткости класс `XmlWriter` опускает объявления пространств имен в дочерних элементах, если они уже объявлены их родительским элементом.

Шаблоны для использования `XmlReader/XmlWriter`

Работа с иерархическими данными

Рассмотрим следующие классы:

```

public class Contacts
{
    public IList<Customer> Customers = new List<Customer>();
    public IList<Supplier> Suppliers = new List<Supplier>();
}

public class Customer { public string FirstName, LastName; }
public class Supplier { public string Name; }

```

Предположим, что мы хотим применить классы XmlReader и XmlWriter для сериализации объекта Contacts в XML, как в приведенном фрагменте:

```

<?xml version="1.0" encoding="utf-8"?>
<contacts>
    <customer id="1">
        <firstname>Jay</firstname>
        <lastname>Dee</lastname>
    </customer>
    <customer> <!-- мы будем предполагать, что id необязателен -->
        <firstname>Kay</firstname>
        <lastname>Gee</lastname>
    </customer>
    <supplier>
        <name>X Technologies Ltd</name>
    </supplier>
</contacts>

```

Лучший подход заключается в том, чтобы не записывать один крупный метод, а инкапсулировать XML-функциональность в самих типах Customer и Supplier, реализовав для них методы ReadXml и WriteXml. Используемый шаблон довольно прост:

- когда методы ReadXml и WriteXml завершаются, они оставляют средство чтения/записи на той же глубине;
- метод ReadXml читает внешний элемент, тогда как метод WriteXml записывает только его внутреннее содержимое.

Ниже показано, как можно было бы реализовать тип Customer:

```

public class Customer
{
    public const string XmlName = "customer";
    public int? ID;
    public string FirstName, LastName;

    public Customer () { }

    public Customer (XmlReader r) { ReadXml (r); }

    public void ReadXml (XmlReader r)
    {
        if (r.MoveToAttribute ("id")) ID = r.ReadContentAsInt ();
        r.ReadStartElement ();
        FirstName = r.ReadElementContentAsString ("firstname", "");
        LastName = r.ReadElementContentAsString ("lastname", "");
        r.ReadEndElement ();
    }
}

```

```

public void WriteXml (XmlWriter w)
{
    if (ID.HasValue) w.WriteAttributeString ("id", "", ID.ToString ());
    w.WriteElementString ("firstname", FirstName);
    w.WriteElementString ("lastname", LastName);
}
}

```

Обратите внимание, что метод `ReadXml` читает узлы внешнего начального и конечного элементов. Если бы эту работу делал вызывающий компонент, то класс `Customer` мог бы не читать собственные атрибуты. Причина, по которой метод `WriteXml` не сделан симметричным в таком отношении, двойственна:

- вызывающий компонент может нуждаться в выборе способа именования внешнего элемента;
- вызывающему компоненту может быть необходима запись дополнительных XML-атрибутов, таких как подтип элемента (который затем может применяться для принятия решения о том, экземпляр какого класса создавать при чтении данного элемента).

Еще одно преимущество следования описанному шаблону связано с тем, что ваша реализация будет совместимой с интерфейсом `IXmlSerializable` (это раскрывается в разделе “Сериализация” дополнительных материалов, которые доступны на веб-сайте издательства).

Класс `Supplier` аналогичен:

```

public class Supplier
{
    public const string XmlName = "supplier";
    public string Name;

    public Supplier () { }

    public Supplier (XmlReader r) { ReadXml (r); }

    public void ReadXml (XmlReader r)
    {
        r.ReadStartElement ();
        Name = r.ReadElementContentAsString ("name", "");
        r.ReadEndElement ();
    }

    public void WriteXml (XmlWriter w) =>
        w.WriteElementString ("name", Name);
    }
}

```

В классе `Contacts` мы должны выполнять перечисление элемента `customers` в методе `ReadXml` с целью проверки, является ли каждый подэлемент заказчиком или поставщиком. Также понадобится закодировать обработку пустых элементов:

```

public void ReadXml (XmlReader r)
{
    bool isEmpty = r.IsEmptyElement;           // Это обеспечивает корректную
    r.ReadStartElement ();                     // обработку пустого
    if (isEmpty) return;                      // элемента <contacts/>
}

```

```

while (r.NodeType == XmlNodeType.Element)
{
    if (r.Name == Customer.XmlName)      Customers.Add (new Customer (r));
    else if (r.Name == Supplier.XmlName)  Suppliers.Add (new Supplier (r));
    else
        throw new XmlException ("Unexpected node: " + r.Name);
        // Непредвиденный узел
}
r.ReadEndElement ();
}

public void WriteXml (XmlWriter w)
{
    foreach (Customer c in Customers)
    {
        w.WriteStartElement (Customer.XmlName);
        c.WriteXml (w);
        w.WriteEndElement ();
    }
    foreach (Supplier s in Suppliers)
    {
        w.WriteStartElement (Supplier.XmlName);
        s.WriteXml (w);
        w.WriteEndElement ();
    }
}

```

Вот как сериализовать объект Contacts, заполненный экземплярами Customer и Supplier, в XML-файл:

```

var settings = new XmlWriterSettings();
settings.Indent = true; // Для улучшения визуального восприятия
using XmlWriter writer = XmlWriter.Create ("contacts.xml", settings);
var cts = new Contacts();
// Добавить экземпляры Customer и Supplier...
writer.WriteStartElement ("contacts");
cts.WriteXml (writer);
writer.WriteEndElement ();

```

А так выполняется десериализация из того же XML-файла:

```

var settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
settings.IgnoreComments = true;
settings.IgnoreProcessingInstructions = true;
using XmlReader reader = XmlReader.Create ("contacts.xml", settings);
reader.MoveToContent ();
var cts = new Contacts ();
cts.ReadXml (reader);

```

Смешивание XmlReader/XmlWriter с моделью X-DOM

Переключиться на модель X-DOM можно в любой точке XML-дерева, где работа с классами XmlReader или XmlWriter становится слишком громоздкой. Использование X-DOM для обработки внутренних элементов — великолепный способ комбинирования простоты применения X-DOM и низкого расхода памяти классами XmlReader и XmlWriter.

Использование XmlReader с XElement

Чтобы прочитать текущий элемент в модель X-DOM, необходимо вызвать метод XNode.ReadFrom, передав ему экземпляр XmlReader. В отличие от XElement.Load этот метод не является “жадным” в том смысле, что он не ожидает увидеть целый документ. Взамен метод XNode.ReadFrom читает только до конца текущего поддерева.

В качестве примера предположим, что имеется XML-файл журнала со следующей структурой:

```
<log>
  <logentry id="1">
    <date>...</date>
    <source>...</source>
    ...
  </logentry>
  ...
</log>
```

При наличии миллиона элементов logentry чтение целого журнала в модель X-DOM приведет к непроизводительному расходу памяти. Более эффективное решение предусматривает обход всех элементов logentry с помощью класса XmlReader и затем использование XElement для индивидуальной обработки каждого элемента:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using XmlReader r = XmlReader.Create ("logfile.xml", settings);
r.ReadStartElement ("log");
while (r.Name == "logentry")
{
  XElement logEntry = (XElement) XNode.ReadFrom (r);
  int id = (int) logEntry.Attribute ("id");
  DateTime date = (DateTime) logEntry.Element ("date");
  string source = (string) logEntry.Element ("source");
  ...
}
r.ReadEndElement();
```

Если следовать шаблону, описанному в предыдущем разделе, тогда XElement можно поместить внутрь метода ReadXml или WriteXml специального типа так, что вызывающий компонент даже не обнаружит подвоха! Например, метод ReadXml класса Customer можно было бы переписать следующим образом:

```
public void ReadXml (XmlReader r)
{
    XElement x = (XElement) XNode.ReadFrom (r);
    ID = (int) x.Attribute ("id");
    FirstName = (string) x.Element ("firstname");
    LastName = (string) x.Element ("lastname");
}
```

Класс XElement взаимодействует с классом XmlReader, чтобы гарантировать, что пространства имен остались незатронутыми, а префиксы соответствующим образом расширенными — даже если они определены на внешнем уровне. Таким образом, если содержимое XML-файла выглядит, как показано ниже:

```
<log xmlns="http://loggingspace">
<logentry id="1">
  ...

```

то экземпляры XElement, сконструированные на уровне logentry, будут корректно наследовать внешнее пространство имен.

Использование XmlWriter с XElement

Класс XElement можно применять только для записи внутренних элементов в XmlWriter. В приведенном далее коде производится запись миллиона элементов logentry в XML-файл с использованием класса XElement — без помещения всех их в память:

```
using XmlWriter w = XmlWriter.Create ("logfile.xml");
w.WriteStartElement ("log");
for (int i = 0; i < 1000000; i++)
{
    XElement e = new XElement ("logentry",
        new XAttribute ("id", i),
        new XElement ("date", DateTime.Today.AddDays (-1)),
        new XElement ("source", "test"));
    e.WriteTo (w);
}
w.WriteEndElement ();
```

С применением класса XElement связаны минимальные накладные расходы во время выполнения. Если мы изменим пример для повсеместного использования класса XmlWriter, то никакой заметной разницы в скорости выполнения не будет.

Работа с JSON

Формат JSON стал популярной альтернативой XML. Хотя в JSON нет расширенных средств XML (таких как пространства имен, префиксы и схемы), его преимущество связано с простотой и отсутствием перегруженности; он похож на формат, который вы бы получили в результате преобразования объекта JavaScript в строку.

Исторически сложилось так, что платформа .NET не имела встроенной поддержки JSON, поэтому приходилось полагаться на сторонние библиотеки — в первую очередь на Json.NET. Хотя сейчас ситуация изменилась, библиотека Json.NET по-прежнему популярна по ряду причин:

- она существует с 2011 года;
- тот же самый API-интерфейс работает и на старых платформах .NET;
- она считается более функциональной (во всяком случае, так было в прошлом), чем API-интерфейсы Microsoft JSON.

Преимущество API-интерфейсов Microsoft JSON заключается в том, что они с самого начала проектировались как простые и чрезвычайно эффективные. Кроме того, начиная с версии .NET 6, их функциональность стала достаточно близкой к Json.NET.

В этом разделе мы раскроем:

- однонаправленные средства чтения и записи (`Utf8JsonReader` и `Utf8JsonWriter`);
- средство чтения DOM-модели (`JsonDocument`);
- средство чтения/записи DOM-модели (`JsonNode`).

В разделе “Сериализация” дополнительных материалов, доступных на веб-сайте издательства, рассматривается класс `JsonSerializer`, который отвечает за сериализацию и десериализацию JSON.

Utf8JsonReader

Структура `System.Text.Json.Utf8JsonReader`(<https://docs.microsoft.com/en-us/dotnet/api/system.text.json.utf8jsonreader?view=net-8.0>) является оптимизированным однонаправленным средством чтения для текста JSON, закодированного посредством UTF-8. Она концептуально похожа на класс `XmlReader`, представленный ранее в главе, и применяется во многом аналогично.

Возьмем JSON-файл по имени `people.json` со следующим содержимым:

```
{  
    "FirstName": "Sara",  
    "LastName": "Wells",  
    "Age": 35,  
    "Friends": ["Dylan", "Ian"]  
}
```

Фигурными скобками обозначается объект JSON (содержащий свойства, такие как `"FirstName"` и `"LastName"`), а квадратными скобками — массив JSON (который содержит повторяющиеся элементы). В данном случае повторяющимися элементами являются строки, но ими могут быть объекты (или другие массивы).

Приведенный далее код производит разбор содержимого файла, проходя по его маркерам JSON. Маркер — это начало или конец объекта, начало или ко-

нец массива, имя свойства или значение массива либо свойства (строка, число, true, false или null):

```
byte[] data = File.ReadAllBytes ("people.json");
Utf8JsonReader reader = new Utf8JsonReader (data);
while (reader.Read())
{
    switch (reader.TokenType)
    {
        case JsonTokenType.StartObject:
            Console.WriteLine ($"Start of object");
            break;
        case JsonTokenType.EndObject:
            Console.WriteLine ($"End of object");
            break;
        case JsonTokenType.StartArray:
            Console.WriteLine ();
            Console.WriteLine ($"Start of array");
            break;
        case JsonTokenType.EndArray:
            Console.WriteLine ($"End of array");
            break;
        case JsonTokenType.PropertyName:
            Console.Write ($"Property: {reader.GetString()}");
            break;
        case JsonTokenType.String:
            Console.WriteLine ($" Value: {reader.GetString()}");
            break;
        case JsonTokenType.Number:
            Console.WriteLine ($" Value: {reader.GetInt32()}");
            break;
        default:
            Console.WriteLine ($"No support for {reader.TokenType}");
            break;
    }
}
```

Вот как выглядит вывод:

```
Start of object
Property: FirstName Value: Sara
Property: LastName Value: Wells
Property: Age Value: 35
Property: Friends
Start of array
Value: Dylan
Value: Ian
End of array
End of object
```

Поскольку структура Utf8JsonReader работает напрямую с кодировкой UTF-8, она проходит по маркерам, не требуя предварительного преобразования входных данных в UTF-16 (формат строк .NET). Преобразование в UTF-16 происходит только при вызове метода вроде GetString.

Интересно отметить, что конструктор `Utf8JsonReader` принимает не байтовый массив, а объект `ReadOnlySpan<byte>` (по этой причине `Utf8JsonReader` определена как ссылочная структура). Вы можете передавать байтовый массив, т.к. существует неявное преобразование из `T[]` в `ReadOnlySpan<T>`. В главе 23 мы опишем, как работают интервалы, и объясним, каким образом их можно использовать для улучшения показателей производительности за счет минимизации выделений памяти.

JsonReaderOptions

По умолчанию `Utf8JsonReader` требует, чтобы данные JSON строго соответствовали стандарту RFC 8259. Вы можете проинструктировать средство чтения о том, что оно должно быть более терпимым, передав конструктору `Utf8JsonReader` экземпляр класса `JsonReaderOptions`, который определяет описанные ниже параметры.

- **Комментарии в стиле языка С.** По умолчанию комментарии в JSON вызывают генерацию исключения `JsonException`. Установка свойства `CommentHandling` в `JsonCommentHandling.Skip` заставляет средство чтения игнорировать комментарии, а в `JsonCommentHandling.Allow` — распознавать их и выпускать маркеры `JsonTokenType.Comment`, когда они встречаются. Комментарии не могут появляться в середине других маркеров.
- **Завершающие запятые.** Согласно стандарту последнее свойство объекта и последний элемент массива не должны иметь завершающую запятую. Установка свойства `AllowTrailingCommas` ее смягчает это ограничение.
- **Контроль над максимальной глубиной вложения.** По умолчанию объекты и массивы можно вкладывать на глубину до 64 уровней. Установка свойства `MaxDepth` в отличающееся число переопределяет данную настройку.

Utf8JsonWriter

Класс `System.Text.Json.Utf8JsonWriter` (<https://docs.microsoft.com/en-us/dotnet/api/system.text.json.utf8jsonwriter?view=net-8.0>) представляет собой одностороннее средство записи JSON. Он поддерживает следующие типы:

- `String` и `DateTime` (сформатированные как строка в JSON);
- числовые типы `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double` и `Decimal` (сформатированные как числа в JSON);
- `bool` (сформатированный как литералы `true/false` в JSON);
- `null` в JSON;
- массивы.

Вы можете организовать указанные типы данных в виде объектов согласно стандарту JSON. Кроме того, можно создавать компоненты, которые не являются частью стандарта JSON, но на практике часто поддерживаются инструментами разбора JSON.

Ниже демонстрируется применение `Utf8JsonWriter`:

```
var options = new JsonWriterOptions { Indented = true };
using (var stream = File.Create ("MyFile.json"))
using (var writer = new Utf8JsonWriter (stream, options))
{
    writer.WriteStartObject ();
    // Имя и значение свойства указываются в одном вызове
    writer.WriteString ("FirstName", "Dylan");
    writer.WriteString ("LastName", "Lockwood");

    // Имя и значение свойства указываются в разных вызовах
    writer.WritePropertyName ("Age");
    writer.WriteNumberValue (46);
    writer.WriteCommentValue ("This is a (non-standard) comment");
    writer.WriteEndObject ();
}
```

Код приводит к генерации выходного файла со следующим содержимым:

```
{
    "FirstName": "Dylan",
    "LastName": "Lockwood",
    "Age": 46
    /*This is a (non-standard) comment*/
}
```

Начиная с версии .NET 6, класс `Utf8JsonWriter` имеет метод `WriteRawValue`, который выдает строку или байтовый массив напрямую в поток данных JSON. Это полезно в особых случаях — например, если необходимо записывать число так, чтобы оно всегда включало десятичную точку (1.0, а не 1).

В приведенном примере мы устанавливаем свойство `Indented` экземпляра `JsonWriterOptions` в `true` с целью улучшения читабельности. Если бы мы этого не сделали, то результат оказался бы таким:

```
{"FirstName": "Dylan", "LastName": "Lockwood", "Age": 46...}
```

Класс `JsonWriterOptions` также имеет свойство `Encoder` для управления специальными символами в строках и свойство `SkipValidation`, позволяющее игнорировать структурные проверки достоверности (и делающее возможным выпуск недействительных выходных данных JSON).

JsonDocument

Класс `System.Text.Json.JsonDocument` производит разбор данных JSON в допускающую только чтение DOM-модель, которая состоит из экземпляров `JsonElement`, генерируемых по требованию. В отличие от `Utf8JsonReader` класс `JsonDocument` обеспечивает произвольный доступ к элементам.

Класс `JsonDocument` является одним из двух API-интерфейсов на основе DOM для работы с JSON, второй API-интерфейс — `JsonNode` (который рассматривается в следующем разделе). Класс `JsonNode` был введен в версии .NET 6 главным образом для удовлетворения спроса на записываемую DOM-модель. Однако он также подходит для сценариев, предусматривающих только чтение,

и предоставляет несколько более гибкий интерфейс, поддерживаемый традиционной DOM-моделью, который использует классы для значений, массивов и объектов JSON. В отличие от этого API-интерфейс `JsonDocument` чрезвычайно легковесный и включает всего один класс примечаний (`JsonDocument`) и две легковесных структуры (`JsonProperty` и `JsonValue`), которые производят разбор базовых данных по требованию. Разница проиллюстрирована на рис. 11.1.



В большинстве реальных сценариев преимущества `JsonDocument` в плане производительности по сравнению с `JsonNode` незначительны, поэтому вы можете сразу выбрать `JsonNode`, если предпочитаете изучать только один API-интерфейс.

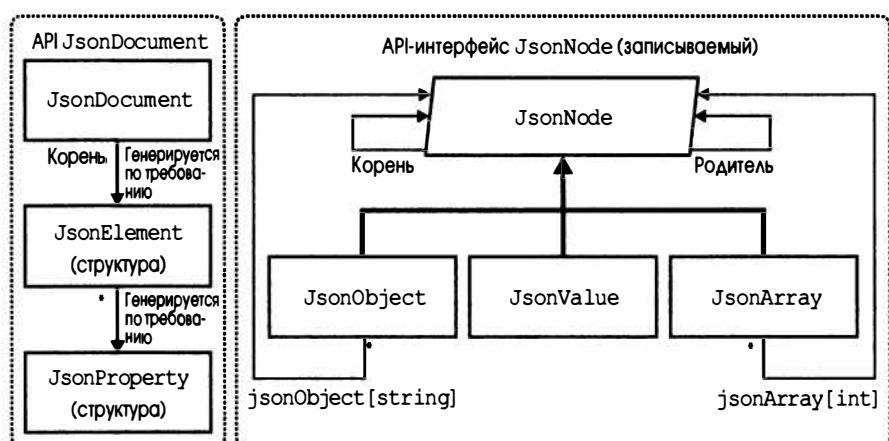


Рис. 11.1. API-интерфейсы DOM-модели JSON



Класс `JsonDocument` дополнительно увеличивает эффективность за счет того, что действует пул в памяти для минимизации сборки мусора. Это означает, что вы обязаны освобождать экземпляр `JsonDocument` после использования, иначе его память не будет возвращена в пул. Следовательно, когда класс сохраняет экземпляр `JsonDocument` в поле, он также должен реализовывать интерфейс `IDisposable`. Если это окажется обременительным, тогда рассмотрите возможность использования `JsonNode`.

Статический метод `Parse` создает экземпляр `JsonDocument` из потока, строки или буфера памяти:

```
using JsonDocument document = JsonDocument.Parse (jsonString);  
...
```

При вызове `Parse` можно дополнительно предоставлять объект `JsonDocumentOptions` для управления обработкой завершающих запятых, комментариев и максимальной глубины вложения (упомянутые параметры обсуждались в разделе “`JsonReaderOptions`” ранее в главе).

Экземпляр `JsonDocument` позволяет получить доступ к модели DOM через свойство `RootElement`:

```
using JsonDocument document = JsonDocument.Parse ("123");
JsonElement root = document.RootElement;
Console.WriteLine (root.ValueKind); // Number
```

Класс `JsonElement` способен представлять значение JSON (строку, число, `true/false`, `null`), массив или объект; свойство `ValueKind` указывает, что именно.



Методы, описанные в последующих разделах, генерируют исключение, если элемент не относится к ожидаемому типу. Если вы не уверены в корректности схемы файла JSON, то во избежание генерации таких исключений можете сначала проверить свойство `ValueKind` (или применять методы `TryGet*`).

Класс `JsonElement` также предлагает два метода, которые работают с элементом любого вида: `GetRawText` возвращает внутренние данные JSON, а `WriteTo` записывает элемент в `Utf8JsonWriter`.

Чтение простых значений

Если элемент представляет значение JSON, то его можно получить с помощью вызова `GetString`, `GetInt32`, `GetBoolean` и т.д.:

```
using JsonDocument document = JsonDocument.Parse ("123");
int number = document.RootElement.GetInt32();
```

Кроме того, в классе `JsonElement` имеются методы для разбора строк JSON в другие распространенные типы CLR, такие как `DateTime` (и даже двоичный тип Base-64). Существуют также версии `TryGet*`, которые предотвращают генерацию исключения в случае неудачи разбора.

Чтение массивов JSON

Если `JsonElement` представляет массив, то вы можете вызывать описанные ниже методы.

- `EnumerateArray()`. Выполняет перечисление всех элементов массива JSON (как экземпляров `JsonElement`).
- `GetArrayLength()`. Возвращает количество элементов в массиве.

Кроме того, вы можете использовать индексатор для возвращения элемента в определенной позиции:

```
using JsonDocument document = JsonDocument.Parse (@"[1, 2, 3, 4, 5]");
int length = document.RootElement.GetArrayLength(); // 5
int value = document.RootElement[3].GetInt32(); // 4
```

Чтение объектов JSON

Если элемент представляет объект JSON, тогда вы можете вызывать следующие методы.

- `EnumerateObject()`. Выполняет перечисление имен и значений всех свойств объекта.
- `GetProperty (string propertyName)`. Получает свойство по имени (возвращая еще один экземпляр `JsonElement`). Генерирует исключение, если свойство с указанным именем не существует.
- `TryGetProperty (string propertyName, out JsonElement value)`. Возвращает свойство объекта, если оно существует.

Например:

```
using JsonDocument document = JsonDocument.Parse(@"{ ""Age"": 32}");
JsonElement root = document.RootElement;
int age = root.GetProperty("Age").GetInt32();
```

Вот как можно было бы “обнаружить” свойство `Age`:

```
JsonProperty ageProp = root.EnumerateObject().First();
string name = ageProp.Name;                                     // Age
JsonElement value = ageProp.Value;
Console.WriteLine (value.ValueKind);                            // Number
Console.WriteLine (value.GetInt32());                           // 32
```

JsonDocument и LINQ

Класс `JsonDocument` хорошо сочетается с LINQ. Имея файл JSON с показанным ниже содержимым:

```
[
  {
    "FirstName": "Sara",
    "LastName": "Wells",
    "Age": 35,
    "Friends": ["Ian"]
  },
  {
    "FirstName": "Ian",
    "LastName": "Weems",
    "Age": 42,
    "Friends": ["Joe", "Eric", "Li"]
  },
  {
    "FirstName": "Dylan",
    "LastName": "Lockwood",
    "Age": 46,
    "Friends": ["Sara", "Ian"]
  }
]
```

мы можем применять `JsonDocument` для его запрашивания с помощью LINQ:

```
using var stream = File.OpenRead(jsonPath);
using JsonDocument document = JsonDocument.Parse(json);
var query =
  from person in document.RootElement.EnumerateArray()
```

```
select new
{
    FirstName = person.GetProperty ("FirstName").GetString(),
    Age = person.GetProperty ("Age").GetInt32(),
    Friends =
        from friend in person.GetProperty ("Friends").EnumerateArray()
        select friend.GetString()
};
```

Поскольку запросы LINQ оцениваются ленивым образом, важно организовать перечисление запроса до того, как документ покинет область видимости и экземпляр JsonDocument неявно освободится благодаря оператору using.

Выполнение обновлений с помощью средства записи JSON

Хотя экземпляр JsonDocument допускает только чтение, посредством метода WriteTo содержимое JsonElement можно отправить экземпляру Utf8JsonWriter, что образует механизм выпуска модифицированной версии данных JSON. Вот как можно взять содержимое файла JSON из предыдущего примера и записать его в новый файл JSON, включая только объекты людей с двумя и более друзьями:

```
using var json = File.OpenRead (jsonPath);
using JsonDocument document = JsonDocument.Parse (json);
var options = new JsonWriterOptions { Indented = true };
using (var outputStream = File.Create ("NewFile.json"))
using (var writer = new Utf8JsonWriter (outputStream, options))
{
    writer.WriteStartArray();
    foreach (var person in document.RootElement.EnumerateArray())
    {
        int friendCount = person.GetProperty ("Friends").GetArrayLength();
        if (friendCount >= 2)
            person.WriteTo (writer);
    }
}
```

Тем не менее, если вам нужна возможность обновления DOM-модели, тогда лучшим решением будет класс JsonNode.

Класс JsonNode

Класс JsonNode (из пространства имен System.Text.Json.Nodes) был введен в .NET 6 в первую очередь для удовлетворения спроса на записываемую DOM-модель. Однако он также подходит для сценариев, предусматривающих только чтение, и предоставляет несколько более гибкий интерфейс, поддерживающий традиционной DOM-моделью, который использует классы для значений, массивов и объектов JSON (см. рис. 11.1). Из-за того, что они являются классами, с ними связаны накладные расходы на сборку мусора, но в большинстве реальных сценариев они, скорее всего, будут незначительными. Класс JsonNode высоко оптимизирован и может работать быстрее, чем JsonDocument, когда одни и те же узлы читаются многократно (поскольку JsonNode, несмотря на ленивое чтение, кеширует результаты разбора).

Статический метод Parse создает экземпляр JsonNode из потока данных, строки, буфера памяти или объекта Utf8JsonReader:

```
JsonNode node = JsonNode.Parse (jsonString);
```

При вызове Parse можно дополнительно предоставить объект JsonDocumentOptions для управления обработкой завершающих запятых, комментариев и максимальной глубины вложенности (эти параметры обсуждались в разделе “JsonReaderOptions” ранее в главе). В отличие от JsonDocument класс JsonNode не требует освобождения.



Вызов ToString() на JsonNode возвращает удобочитаемую (с отступами) строку JSON. Существует также метод ToJsonString(), который возвращает компактную строку JSON.

Начиная с версии .NET 8, класс JsonNode имеет статический метод DeepEquals, так что можно сравнивать два объекта JsonNode без предварительного их расширения в строки JSON. Кроме того, есть метод DeepCopy из .NET 8.

Метод Parse возвращает подтип JsonNode, которым будет JsonValue, JsonObject или JSONArray. В классе JsonNode предусмотрены вспомогательные методы AsValue(), AsObject() и AsArray(), позволяющие избежать громоздких приведений вниз:

```
var node = JsonNode.Parse ("123"); // Выполняет разбор объекта JsonValue
int number = node.AsValue<int>().GetValue<int>();
// Сокращение для ((JsonValue)node).GetValue<int>();
```

Тем не менее, обычно вызывать эти методы не придется, поскольку наиболее часто используемые члены представлены в самом классе JsonNode:

```
var node = JsonNode.Parse ("123");
int number = node.GetValue<int>();
// Сокращение для node.AsValue().GetValue<int>();
```

Чтение простых значений

Только что было показано, что выполнить извлечение или разбор простого значения можно с помощью вызова GetValue с параметром типа. Для еще большего упрощения явные операции приведения C# в классе JsonNode переопределены, делая возможным применение следующей сокращенной версии кода:

```
var node = JsonNode.Parse ("123");
int number = (int) node;
```

Прием работает для стандартных числовых типов: char, bool, DateTime, DateTimeOffset и Guid (и их версий, допускающих значение null), а также string.

Если нет уверенности, что разбор будет успешным, тогда придется использовать такой код:

```
if (node.AsValue().TryGetValue<int> (out var number))
    Console.WriteLine (number);
```

В .NET 8 вызов `node.GetValueKind()` сообщает, чем является узел: строкой, числом, массивом, объектом или значением `true/false`.

Узлы, которые были получены в результате разбора текста JSON, внутренне поддерживаются `JsonElement` (часть API-интерфейса `JsonDocument`, допускающего только чтение). Извлечь лежащий в основе объект `JsonElement` можно следующим образом:

```
JsonElement je = node.GetValue<JsonElement>();
```

Однако прием не работает, когда экземпляр узла создается явно (как будет в случае обновления DOM-модели). Такие узлы поддерживаются не `JsonElement`, а фактическим значением после разбора (см. раздел “Обновление с помощью `JsonNode`” далее в главе).

Чтение массивов JSON

Экземпляр `JsonNode`, представляющий массив JSON, будет иметь тип `JsonArray`, который реализует `IList<JsonNode>`, поэтому можно выполнять его перечисление и получать доступ к элементам подобно обычному массиву или списку:

```
var node = JsonNode.Parse(@"[1, 2, 3, 4, 5]");
Console.WriteLine(node.AsArray().Count); // 5
foreach (JsonNode child in node.AsArray())
{ ... }
```

Если обратиться к индексатору напрямую из класса `JsonNode`, тогда код удастся сократить:

```
Console.WriteLine((int)node[0]); // 1
```

Начиная с версии .NET 8, можно также вызывать метод `GetValues<T>` для возвращения данных в виде экземпляра реализации `IEnumerable<T>`:

```
int[] values = node.AsArray().GetValues<int>().ToArray();
```

Чтение объектов JSON

Экземпляр `JsonNode`, который представляет объект JSON, будет иметь тип `JsonObject`.

Класс `JsonObject` реализует интерфейс `IDictionary<string, JsonNode>`, поэтому можно получать доступ к члену через индексатор, а также выполнять перечисление пар “ключ/значение” словаря.

Как и в случае с `JsonArray`, обращаться к индексатору можно прямо из класса `JsonNode`:

```
var node = JsonNode.Parse(@"{ ""Name"": ""Alice"" , ""Age"": 32}");
string name = (string) node["Name"]; // Alice
int age = (int) node["Age"]; // 32
```

Вот как можно было бы “обнаружить” свойства `Name` и `Age`:

```
// Выполнить перечисление пар ключ/значение словаря:
foreach (KeyValuePair<string, JsonNode> keyValuePair in node.AsObject())
{
    string propertyName = keyValuePair.Key; // "Name" (затем "Age")
    JsonNode value = keyValuePair.Value;
}
```

Если вы не уверены, определено ли свойство, то подойдет следующий прием:

```
if (node.AsObject().TryGetProperty("Name", out JsonNode nameNode))  
{ ... }
```

Гибкий обход и LINQ

Вы можете глубоко проникнуть в иерархию с помощью только индексаторов. Например, имея следующий файл JSON:

```
[  
 {  
     "FirstName": "Sara",  
     "LastName": "Wells",  
     "Age": 35,  
     "Friends": ["Ian"]  
 },  
 {  
     "FirstName": "Ian",  
     "LastName": "Weems",  
     "Age": 42,  
     "Friends": ["Joe", "Eric", "Li"]  
 },  
 {  
     "FirstName": "Dylan",  
     "LastName": "Lockwood",  
     "Age": 46,  
     "Friends": ["Sara", "Ian"]  
 }  
 ]
```

Вот как можно извлечь третьего друга второго человека:

```
string li = (string) node[1]["Friends"][2];
```

Выполнять запросы в таком файле также легко посредством LINQ:

```
JsonNode node = JsonNode.Parse(File.ReadAllText(jsonPath));  
  
var query =  
    from person in node.AsArray()  
    select new  
    {  
        FirstName = (string) person["FirstName"],  
        Age = (int) person["Age"],  
        Friends =  
            from friend in person["Friends"].AsArray()  
            select (string) friend  
    };
```

В отличие от `JsonDocument` класс `JsonNode` не является освобождаемым, так что беспокоиться о возможности освобождения во время ленивого перечисления не нужно.

Обновление с помощью `JsonNode`

Классы `JsonObject` и `JsonArray` являются изменяемыми, поэтому можно обновлять их содержимое.

Самый простой способ замены или добавления свойств в `JsonObject` предусматривает применение индексатора. В следующем примере мы изменяем значение свойства `Color` с `"Red"` на `"White"` и добавляем новое свойство по имени `Valid`:

```
var node = JsonNode.Parse ("{ \"Color\": \"Red\" }");
node ["Color"] = "White";
node ["Valid"] = true;
Console.WriteLine (node.ToString()); // {"Color":"White","Valid":true}
```

Вторая строка в приведенном примере является сокращенной версией следующей строки:

```
node ["Color"] = JsonValue.Create ("White");
```

Вместо присваивания свойству простого значения можно присвоить ему экземпляр `JsonArray` или `JsonObject`. (В следующем разделе будет показано, каким образом создавать экземпляры `JsonArray` и `JsonObject`.)

Чтобы удалить свойство, сначала необходимо привести его к `JsonObject` (или вызвать `AsObject`), а затем вызвать метод `Remove`:

```
node.AsObject().Remove ("Valid");
```

(Класс `JsonObject` вдобавок предоставляет метод `Add`, который генерирует исключение, если свойство уже существует.)

Класс `JsonArray` также позволяет использовать индексатор для замены элементов:

```
var node = JsonNode.Parse ("[1, 2, 3]");
node[0] = 10;
```

Вызов `AsArray` открывает доступ к методам `Add/Insert/Remove/RemoveAt`. В следующем примере мы удаляем первый элемент массива и добавляем его в конец:

```
var arrayNode = JsonNode.Parse ("[1, 2, 3]");
arrayNode.AsArray().RemoveAt (0);
arrayNode.AsArray().Add (4);
Console.WriteLine (arrayNode.ToString()); // [2,3,4]
```

Начиная с версии .NET 8, обновлять экземпляр `JsonNode` можно с помощью вызова `ReplaceWith`:

```
var node = JsonNode.Parse ("{ \"Color\": \"Red\" }");
var color = node["Color"];
color.ReplaceWith ("Blue");
```

Конструирование DOM-модели `JsonNode` в коде

Классы `JsonArray` и `JsonObject` имеют конструкторы, поддерживающие синтаксис инициализации объекта, что позволяет построить всю DOM-модель `JsonNode` в одном выражении:

```
var node = new JSONArray
{
    new JSONObject {
        ["Name"] = "Tracy",
        ["Age"] = 30,
        ["Friends"] = new JSONArray ("Lisa", "Joe")
    },
    new JSONObject {
        ["Name"] = "Jordyn",
        ["Age"] = 25,
        ["Friends"] = new JSONArray ("Tracy", "Li")
    }
};
```

В результате получаются представленные ниже данные JSON:

```
[
    {
        "Name": "Tracy",
        "Age": 30,
        "Friends": ["Lisa", "Joe"]
    },
    {
        "Name": "Jordyn",
        "Age": 25,
        "Friends": ["Tracy", "Li"]
    }
]
```




Освобождение и сборка мусора

Некоторые объекты требуют написания явного кода для возврата таких ресурсов, как открытые файлы, блокировки, дескрипторы операционной системы и неуправляемые объекты. В терминологии .NET процедура называется освобождением и поддерживается через интерфейс `IDisposable`. Управляемая память, занятая неиспользуемыми объектами, тоже в какой-то момент должна быть возвращена; такая функция называется сборкой мусора и выполняется средой CLR.

Освобождение отличается от сборки мусора тем, что обычно оно инициируется явно, в то время как сборка мусора является полностью автоматической. Другими словами, программист заботится об освобождении файловых дескрипторов, блокировок и ресурсов операционной системы, а среда CLR занимается освобождением памяти.

В настоящей главе обсуждаются темы освобождения и сборки мусора, а также рассматриваются финализаторы C# и шаблон, согласно которому они могут предоставить страховку для освобождения. Наконец, здесь раскрываются тонкости сборщика мусора и другие варианты управления памятью.

`IDisposable`, `Dispose` и `Close`

В .NET определен специальный интерфейс для типов, требующих метода освобождения:

```
public interface IDisposable
{
    void Dispose();
}
```

Оператор `using` языка C# предлагает синтаксическое сокращение для вызова метода `Dispose` на объектах, которые реализуют интерфейс `IDisposable`, используя блок `try/finally`. Например:

```
using (FileStream fs = new FileStream ("myFile.txt", FileMode.Open))
{
    // ...Запись в файл...
}
```

Компилятор преобразует такой код следующим образом:

```
FileStream fs = new FileStream ("myFile.txt", FileMode.Open);
try
{
    // ...Запись в файл...
}
finally
{
    if (fs != null) ((IDisposable)fs).Dispose();
}
```

Блок `finally` гарантирует, что метод `Dispose` вызывается даже в случае генерации исключения или принудительного раннего выхода из блока.

Аналогичным образом показанный ниже синтаксис обеспечивает освобождение, когда `fs` покидает область видимости:

```
using FileStream fs = new FileStream ("myFile.txt", FileMode.Open);
// ...Запись в файл...
```

В простых сценариях создание собственного освобождаемого типа сводится просто к реализации интерфейса `IDisposable` и написанию метода `Dispose`:

```
sealed class Demo : IDisposable
{
    public void Dispose()
    {
        // Выполнить очистку/освобождение
        ...
    }
}
```



Такой шаблон хорошо работает в простых случаях и подходит для запечатанных классов. В разделе “Вызов метода `Dispose` из финализатора” далее в главе мы представим более продуманный шаблон, который может обеспечить страховку для потребителей, забывших вызвать `Dispose`. В случае незапечатанных типов есть веские основания следовать такому шаблону с самого начала — иначе наступит крупная путаница, когда подтипы сами пожелают добавить функциональность подобного рода.

Стандартная семантика освобождения

Логика освобождения в .NET подчиняется фактическому набору правил. Эти правила не являются жестко привязанными к платформе .NET или к языку C#; их назначение заключается в том, чтобы определить согласованный протокол для потребителей. Правила описаны ниже.

1. После того, как объект был освобожден, он находится в “подвешенном” состоянии. Его нельзя реактивировать, а обращение к его методам или свойствам (отличающимся от `Dispose`) становится причиной генерации исключения `ObjectDisposedException`.
2. Многократный вызов метода `Dispose` объекта не приводит к ошибкам.
3. Если освобождаемый объект х “владеет” освобождаемым объектом у, то метод `Dispose` объекта х автоматически вызывает метод `Dispose` объекта у — при условии, что не указано иначе.
4. Перечисленные правила также полезны при написании собственных типов, хотя они не являются обязательными. Ничто не способно остановить вас от написания метода вроде “`Undispose`”, разве только перспектива получить взбучку от коллег по разработке!

Согласно правилу 3 объект контейнера автоматически освобождает свои дочерние объекты. Хорошим примером может служить контейнерный элемент управления Windows Forms, такой как `Form` или `Panel`. Контейнер может размещать множество дочерних элементов управления, однако вы не должны освобождать каждый из них явно: обо всех них позаботится закрываемый или освобождаемый родительский элемент управления либо форма. Еще один пример — помещение типа `FileStream` в оболочку `DeflateStream`. Освобождение `DeflateStream` также приводит к освобождению `FileStream` — если только в конструкторе не указано иное.

Close и **Stop**

Некоторые типы в дополнение к `Dispose` определяют метод по имени `Close`. Библиотека .NET BCL не полностью согласована относительно семантики метода `Close`, хотя почти во всех случаях он обладает одной из следующих характеристик:

- является функционально идентичным методу `Dispose`;
- реализует подмножество функциональности метода `Dispose`.

Второй характеристикой обладает, например, интерфейс `IDbConnection`: подключение с состоянием `Closed` (закрыто) можно повторно открыть посредством метода `Open`, но сделать это для освобожденного (по причине вызова `Dispose`) подключения нельзя. Другим примером может служить элемент `Form` из Windows Forms, активизированный с помощью метода `ShowDialog`: вызов `Close` скрывает этот элемент, а вызов `Dispose` освобождает его ресурсы.

В некоторых классах определен метод `Stop` (скажем, в `Timer` и `HttpListener`). Метод `Stop` может освобождать неуправляемые ресурсы подобно `Dispose`, но в отличие от `Dispose` он разрешает последующий перезапуск (посредством `Start`).

Когда выполнять освобождение

Безопасное правило, которому необходимо следовать (почти во всех случаях), формулируется так: если есть сомнения, тогда нужно освобождать.

Объекты, содержащие неуправляемый дескриптор ресурса, почти всегда требуют освобождения, чтобы освободить такой дескриптор. Примеры включают файловые или сетевые потоки, сетевые сокеты, элементы управления Windows Forms, перья, кисти и растровые изображения GDI+. И наоборот, если тип является освобождаемым, тогда он часто (но не всегда) ссылается на неуправляемый дескриптор, прямо или косвенно. Причина в том, что неуправляемые дескрипторы предоставляют шлюз во “внешний мир” ресурсам операционной системы, сетевым подключениям, блокировкам базы данных — основным средствам, из-за некорректного отбрасывания которых объекты могут создавать проблемы за своими пределами. Тем не менее, существуют три сценария, когда в освобождении нет необходимости:

- когда вы не “владеете” объектом, например, при получении общего объекта через статическое поле или свойство;
- когда метод Dispose объекта выполняет какое-то нежелательное действие;
- когда метод Dispose объекта является лишним согласно проекту, и освобождение такого объекта добавляет сложности программе.

Первый сценарий встречается редко. Основные случаи отражены в пространстве имен System.Drawing: объекты GDI+, получаемые через статические поля или свойства (такие как Brushes.Blue), никогда не должны освобождаться, поскольку один и тот же экземпляр задействован на протяжении всего времени жизни приложения. Однако экземпляры, которые получаются с помощью конструкторов (скажем, через new SolidBrush), должны быть освобождены, как и должны освобождаться экземпляры, полученные посредством статических методов (вроде Font.FromHdc).

Второй сценарий более распространен. В пространствах имен System.IO и System.Data можно найти ряд удачных примеров.

Тип	Что делает функция освобождения	Когда освобождение выполнять не нужно
MemoryStream	Предотвращает дальнейший ввод-вывод	Когда позже необходимо читать/записывать в поток
StreamReader, StreamWriter	Сбрасывает средство чтения/записи и закрывает лежащий в основе поток	Когда лежащий в основе поток должен быть сохранен открытый (по окончании потребуется вызвать метод Flush на объекте StreamWriter)
IDbConnection	Освобождает подключение к базе данных и очищает строку подключения	Если необходимо повторно открыть его с помощью Open, то должен быть вызван метод Close, а не Dispose
DbContext (EF Core)	Предотвращает дальнейшее использование	Когда могут существовать лениво оцениваемые запросы, подключенные к данному контексту

Метод `Dispose` класса `MemoryStream` делает недоступным только сам объект; он не выполняет никакой критически важной очистки, потому что `MemoryStream` не удерживает неуправляемые дескрипторы или другие ресурсы подобного рода.

Третий сценарий охватывает такие классы, как `StringReader` и `StringWriter`. Упомянутые типы являются освобождаемыми по принуждению их базового класса, а не по причине реальной потребности в выполнении необходимой очистки. Если приходится создавать и работать с таким объектом полностью внутри одного метода, тогда помещение его в блок `using` привносит лишь небольшое неудобство. Но если объект является более долговечным, то процедура выяснения, когда он больше не применяется и может быть освобожден, добавляет излишнюю сложность. В таких случаях можно просто проигнорировать освобождение объекта.



Игнорирование освобождения может иногда повлечь за собой снижение производительности (см. раздел “Вызов метода `Dispose` из финализатора” далее в главе).

Очистка полей при освобождении

В общем случае очищать поля объекта в его методе `Dispose` вовсе не обязательно. Тем не менее, рекомендуемой практикой является отмена подписки на события, на которые объект был подписан внутренне во время своего существования (пример приведен в разделе “Утечки управляемой памяти” далее в главе). Отмена подписки на события подобного рода позволяет избежать получения нежелательных уведомлений о событиях, а также непреднамеренного сохранения объекта в активном состоянии с точки зрения сборщика мусора (*garbage collector — GC*).



Сам по себе метод `Dispose` не вызывает освобождения (управляемой) памяти — это может произойти только при сборке мусора.

Стоит также установить какое-то поле, указывающее на факт освобождения объекта, чтобы можно было сгенерировать исключение `ObjectDisposedException`, если потребитель позже попытается обратиться к членам освобожденного объекта. Хороший шаблон предусматривает использование свойства, открытого для чтения:

```
public bool IsDisposed { get; private set; }
```

Хотя формально и необязательно, но неплохо также очистить собственные обработчики событий объекта (устанавливая их в `null`) в методе `Dispose`. Тем самым исключается возможность возникновения таких событий во время или после освобождения.

Иногда объект хранит ценную секретную информацию вроде ключей шифрования. В подобных случаях имеет смысл во время освобождения очистить такие данные в полях (во избежание их обнаружения другими процессами на машине, когда память позже передается операционной системе). Именно это делает класс

`SymmetricAlgorithm` из пространства имен `System.Security.Cryptography`, вызывая метод `Array.Clear` на байтовом массиве, который хранит ключ шифрования.

Анонимное освобождение

Иногда бывает полезно реализовать интерфейс `IDisposable` без написания класса. Например, предположим, что вы хотите предоставить доступ к методам класса, которые приостанавливают и возобновляют обработку событий:

```
class Foo
{
    int _suspendCount;

    public void SuspendEvents() => _suspendCount++;
    public void ResumeEvents() => _suspendCount--;

    void FireSomeEvent()
    {
        if (_suspendCount == 0)
            ...Запустить событие...
    }
    ...
}
```

Пользоваться таким API-интерфейсом неудобно. Потребители обязаны помнить о необходимости вызова метода `ResumeEvents`. И для надежности они должны вызывать его в блоке `finally` (на случай генерации исключения):

```
var foo = new Foo();
foo.SuspendEvents();
try
{
    ...Выполнение работы... //Из-за того, что здесь может возникнуть исключение, ...
}
finally
{
    foo.ResumeEvents();    //...мы должны вызывать ResumeEvents в блоке finally.
}
```

Лучше избавиться от `ResumeEvents` и заставить метод `SuspendEvents` возвращать реализацию `IDisposable`. Тогда потребители смогут поступать так:

```
using (foo.SuspendEvents())
{
    ...Выполнение работы...
}
```

Проблема в том, что это увеличивает объем работы у того, кто должен реализовывать метод `SuspendEvents`. Даже приложив немалые усилия по сокращению, мы получаем добавочный беспорядок:

```
public IDisposable SuspendEvents()
{
    _suspendCount++;
    return new SuspendToken (this);
}
```

```

class SuspendToken : IDisposable
{
    Foo _foo;
    public SuspendToken (Foo foo) => _foo = foo;
    public void Dispose()
    {
        if (_foo != null) _foo._suspendCount--;
        _foo = null;           // Предотвратить двойное освобождение потребителем
    }
}

```

Проблему решает шаблон анонимного освобождения. С помощью следующего многократно используемого класса:

```

public class Disposable : IDisposable
{
    public static Disposable Create (Action onDispose)
        => new Disposable (onDispose);
    Action _onDispose;
    Disposable (Action onDispose) => _onDispose = onDispose;
    public void Dispose()
    {
        _onDispose?.Invoke(); // Выполнить действие освобождения, если не null.
        _onDispose = null;   // Обеспечить, чтобы его нельзя было выполнить еще раз
    }
}

```

мы можем сократить метод SuspendEvents:

```

public IDisposable SuspendEvents()
{
    _suspendCount++;
    return Disposable.Create (() => _suspendCount--);
}

```

Автоматическая сборка мусора

Независимо от того, требует ли объект метода `Dispose` для специальной логики освобождения, в какой-то момент память, занимаемая им в куче, должна быть освобождена. Среда CLR обрабатывает данный аспект полностью автоматически посредством автоматического сборщика мусора. Вы никогда не освобождаете управляемую память самостоятельно. Например, взгляните на следующий метод:

```

public void Test()
{
    byte[] myArray = new byte[1000];
    ...
}

```

Когда метод `Test` выполняется, в куче выделяется память под массив для хранения 1000 байтов. Ссылка на массив осуществляется через локальную переменную `myArray`, хранящуюся в стеке. Когда метод завершается, локальная переменная `myArray` покидает область видимости, а это значит, что ничего не

остается для ссылки на массив в куче. Висячий массив затем может быть утилизирован при сборке мусора.



В режиме отладки с отключенной оптимизацией время жизни объекта, на который производится ссылка с помощью локальной переменной, расширяется до конца блока кода, чтобы упростить процесс отладки. В противном случае объект становится пригодным для сборки мусора в самой ранней точке, после которой им перестали пользоваться.

Сборка мусора не происходит немедленно после того, как объект становится висячим. Почти как уборка мусора на улицах, она выполняется периодически, хотя (в отличие от уборки улиц) и не по фиксированному графику. Среда CLR основывает свое решение о том, когда инициировать сборку мусора, на ряде факторов, таких как доступная память, объем выделенной памяти и время, прошедшее с момента последней сборки (сборщик мусора самостоятельно подстраивается для оптимизации под конкретные шаблоны доступа в память). Это значит, что между моментом, когда объект становится висячим, и моментом, когда занятая объектом память будет освобождена, имеется неопределенная задержка. Такая задержка может варьироваться в пределах от наносекунд до дней.



Сборщик мусора не собирает весь мусор при каждой сборке. Взамен диспетчер памяти разделяет объекты на *поколения*, и сборщик мусора выполняет сборку новых поколений (недавно распределенных объектов) чаще, чем старых поколений (объектов, существующих на протяжении длительного времени). Мы обсудим принцип более подробно в разделе “Как работает сборщик мусора” далее в главе.

Сборка мусора и потребление памяти

Сборщик мусора старается соблюдать баланс между временем, затрачиваемым на сборку мусора, и потреблением памяти со стороны приложения (рабочий набор). Следовательно, приложения могут расходовать больше памяти, чем им необходимо, особенно если конструируются крупные временные массивы. Отслеживать потребление памяти процессом можно с помощью диспетчера задач Windows или монитора ресурсов — либо программно запрашивая счетчик производительности:

```
// Эти типы находятся в пространстве имен System.Diagnostics:  
string procName = Process.GetCurrentProcess().ProcessName;  
using PerformanceCounter pc = new PerformanceCounter  
    ("Process", "Private Bytes", procName);  
Console.WriteLine (pc.NextValue());
```

В данном коде запрашивается *закрытый рабочий набор*, который дает наилучшую общую картину потребления памяти программой. В частности, он исключает память, которую среда CLR освободила внутренне и готова возвратить операционной системе, если в данной памяти нуждается другой процесс.

Корневые объекты

Корневой объект — это то, что сохраняет определенный объект в активном состоянии. Если на какой-то объект нет прямой или косвенной ссылки со стороны корневого объекта, то он будет пригоден для сборки мусора.

Корневым объектом может выступать одна из следующих сущностей:

- локальная переменная или параметр в выполняющемся методе (или в любом методе внутри его стека вызовов);
- статическая переменная;
- объект в очереди, которая хранит объекты, готовые к финализации (см. следующий раздел).

Код в удаленном объекте не может выполняться, а потому если есть хоть какая-то вероятность выполнения некоторого метода (экземпляра), то на его объект должна существовать ссылка одним из указанных выше способов.

Обратите внимание, что группа объектов, которые циклически ссылаются друг на друга, считается висячей, если отсутствует ссылка из корневого объекта (рис. 12.1). Выражаясь по-другому, объекты, которые не могут быть доступны за счет следования по ссылкам (обозначенным стрелками) из корневого объекта, являются недостижимыми и таким образом подпадают под сборку мусора.

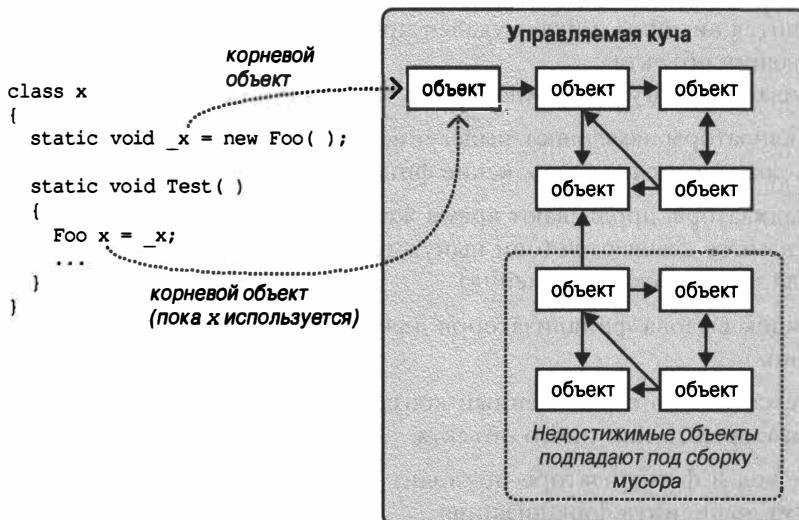


Рис. 12.1. Пример корневых объектов

Финализаторы

Перед освобождением объекта из памяти запускается его финализатор, если он предусмотрен. Финализатор объявляется подобно конструктору, но его имя снабжается префиксом ~:

```
class Test
{
    ~Test()
    {
        // Логика финализатора...
    }
}
```

(Несмотря на сходство в объявлении с конструктором, финализаторы не могут быть объявлены как `public` или `static`, не могут принимать параметры и не могут обращаться к базовому классу.)

Финализаторы возможны потому, что работа сборки мусора организована в виде отличающихся фаз. Первым делом сборщик мусора идентифицирует неиспользуемые объекты, готовые к удалению. Те из них, которые не имеют финализаторов, удаляются немедленно. Те из них, которые располагают отложенными (незапущенными) финализаторами, сохраняются в активном состоянии (на текущий момент) и помещаются в специальную очередь.

В данной точке сборка мусора завершена, и программа продолжает выполнение. Затем параллельно программе начинает выполняться поток финализаторов, который выбирает объекты из этой специальной очереди и запускает их методы финализации. Перед запуском финализатора каждый объект по-прежнему активен — специальная очередь действует в качестве корневого объекта. После того, как объект извлечен из очереди, а его финализатор выполнен, объект становится висячим и будет удален при следующей сборке мусора (для данного поколения объекта).

Финализаторы могут быть удобными, но с рядом оговорок.

- Финализаторы замедляют выделение и утилизацию памяти (сборщик мусора должен отслеживать, какие финализаторы были запущены).
- Финализаторы продлевают время жизни объекта и любых объектов, которые на него ссылаются (они вынуждены ожидать очередной сборки мусора для фактического удаления).
- Порядок вызова финализаторов для набора объектов предсказать невозможно.
- Имеется только ограниченный контроль над тем, когда будет вызван финализатор того или иного объекта.
- Если код в финализаторе приводит к блокировке, то другие объекты не смогут выполнить финализацию.
- Финализаторы могут вообще не запуститься, если приложение не смогло выгрузиться чисто.

В целом финализаторы кое в чем похожи на юристов — хотя и бывают ситуации, когда они действительно нужны, обычно прибегать к их услугам желания нет, разве что в случае крайней необходимости. К тому же, если вы решили воспользоваться услугами юристов, то должны быть абсолютно уверены в понимании того, что они делают для вас.

Ниже приведены некоторые руководящие принципы, применяемые при реализации финализаторов.

- Удостоверьтесь, что финализатор выполняется быстро.
- Никогда не блокируйте финализатор (см. раздел “Блокирование” главы 14).
- Не ссылайтесь на другие финализируемые объекты.
- Не генерируйте исключения.



Среда CLR может вызвать финализатор объекта, даже если во время конструирования сгенерировано исключение. По этой причине при написании финализатора лучше не предполагать, что все поля были корректно инициализированы.

Вызов метода `Dispose` из финализатора

Популярный шаблон предусматривает вызов в финализаторе метода `Dispose`. Подобное действие имеет смысл, когда очистка не является срочной, и ее ускорение вызовом метода `Dispose` является больше оптимизацией, нежели необходимости.



Имейте в виду, что в данном шаблоне вы объединяете вместе освобождение памяти и освобождение ресурсов — две вещи с потенциально несовпадающими интересами (если только сам ресурс не является памятью). Вы также увеличиваете нагрузку на поток финализации.

Такой шаблон также служит в качестве страховки для случаев, когда потребитель попросту забывает вызывать метод `Dispose`. Однако затем подобную небрежность неплохо бы зарегистрировать в журнале, чтобы впоследствии исправить ошибку.

Ниже показан стандартный шаблон реализации:

```
class Test : IDisposable
{
    public void Dispose()           //НЕ virtual
    {
        Dispose (true);
        GC.SuppressFinalize (this); //Препятствует запуску финализатора
    }
    protected virtual void Dispose (bool disposing)
    {
        if (disposing)
        {
            // Вызвать метод Dispose на других объектах, которыми владеет данный
            // экземпляр. Здесь можно ссылаться на другие финализируемые объекты.
            // ...
        }
        //Освободить неуправляемые ресурсы, которыми владеет (только) этот объект
        // ...
    }
    ~Test () => Dispose (false);
}
```

Метод `Dispose` перегружен для приема флага `disposing` типа `bool`. Версия без параметров не объявлена как `virtual` и просто вызывает расширенную версию `Dispose` с передачей ей значения `true`.

Расширенная версия содержит действительную логику освобождения и помечена как `protected` и `virtual`, предоставляя подклассам безопасную точку для добавления собственной логики освобождения. Флаг `disposing` означает, что он вызывается “надлежащим образом” из метода `Dispose`, а не в “режиме крайнего средства” из финализатора. Идея заключается в том, что при вызове с флагом `disposing`, установленным в `false`, этот метод в общем случае не должен ссылаться на другие объекты с финализаторами (поскольку такие объекты сами могут быть финализированными и потому находиться в непредсказуемом состоянии). Как видите, правила исключают довольно многое! Существует пара задач, которые метод `Dispose` по-прежнему может выполнять в режиме крайнего средства, когда `disposing` равно `false`:

- освобождение любых прямых ссылок на ресурсы операционной системы (полученных возможно через обращение `P/Invoke` к Win32 API);
- удаление временного файла, созданного при конструировании.

Чтобы сделать описанный подход надежным, любой код, способный генерировать исключение, должен быть помещен в блок `try/catch`, а исключение в идеальном случае необходимо регистрировать в журнале. Любая регистрация в журнале должна быть насколько возможно простой и надежной.

Обратите внимание, что внутри метода `Dispose` без параметров мы вызываем метод `GC.SuppressFinalize` — это предотвращает запуск финализатора, когда сборщик мусора позже доберется до него. Формально подобное не обязательно, т.к. методы `Dispose` должны допускать многократные вызовы. Тем не менее, такой прием повышает производительность, потому что позволяет подвергнуть данный объект (и объекты, которые на него ссылаются) процедуре сборки мусора в единственном цикле.

Восстановление

Предположим, что финализатор модифицирует активный объект так, что он снова ссылается на неактивный объект. Во время очередной сборки мусора (для поколения объекта) среда CLR выяснит, что ранее неактивный объект больше не является висячим, поэтому он должен избежать сборки мусора. Такой расширенный сценарий называется восстановлением.

В целях иллюстрации предположим, что нужно написать класс, который управляет временным файлом. Во время сборки мусора для экземпляра данного класса финализатор класса должен удалить временный файл. Решение задачи кажется простым:

```
public class TempFileRef
{
    public readonly string FilePath;
    public TempFileRef (string filePath) { FilePath = filePath; }
    ~TempFileRef() { File.Delete (FilePath); }
}
```

К сожалению, здесь присутствует ошибка: вызов метода `File.Delete` может генерировать исключение (возможно, из-за нехватки разрешений, по причине того, что файл в текущий момент используется, или файл уже был удален). Такое исключение привело бы к нарушению работы всего приложения (и воспрепятствовало бы запуску других финализаторов). Мы могли бы просто “поглотить” исключение с помощью пустого блока перехвата, но тогда не было бы известно, что именно пошло не так. Обращение к какому-то хорошо продуманному API-интерфейсу сообщения об ошибках тоже нежелательно, поскольку это привнесет накладные расходы в поток финализаторов, затрудняя проведение сборки мусора для других объектов. Мы хотим, чтобы действия финализации были простыми, надежными и быстрыми.

Более удачное решение предполагает запись информации об отказе в статическую коллекцию:

```
public class TempFileRef
{
    static internal readonly ConcurrentQueue<TempFileRef> FailedDeletions
        = new ConcurrentQueue<TempFileRef>();
    public readonly string FilePath;
    public Exception DeletionError { get; private set; }
    public TempFileRef (string filePath) { FilePath = filePath; }
    ~TempFileRef()
    {
        try { File.Delete (FilePath); }
        catch (Exception ex)
        {
            DeletionError = ex;
            FailedDeletions.Enqueue (this); // Восстановление
        }
    }
}
```

Занесение объекта в статическую коллекцию `FailedDeletions` предоставляет ему еще одну ссылку, гарантуя тем самым, что он останется активным до тех пор, пока со временем не будет изъят из этой коллекции.



Класс `ConcurrentQueue<T>` является безопасной к потокам версией класса `Queue<T>` и определен в пространстве имен `System.Collections.Concurrent` (см. главу 22). Коллекция, безопасная к потокам, применяется по двум причинам. Во-первых, среда CLR резервирует право на выполнение финализаторов более чем одному потоку параллельно. Это значит, что при доступе к общему состоянию, такому как статическая коллекция, мы должны учитывать возможность одновременной финализации двух объектов. Во-вторых, в определенный момент понадобится изъять элементы из `FailedDeletions`, чтобы предпринять в отношении них какие-то действия. Действия должны выполняться также в безопасной к потокам манере, потому что они могут происходить одновременно с занесением в коллекцию другого объекта финализатором.

GC.ReRegisterForFinalize

Финализатор восстановленного объекта не запустится во второй раз, если только не вызвать метод `GC.ReRegisterForFinalize`.

В следующем примере мы пытаемся удалить временный файл внутри финализатора (как в последнем примере). Но если удаление терпит неудачу, то мы повторно регистрируем объект, чтобы предпринять новую попытку при следующей сборке мусора:

```
public class TempFileRef
{
    public readonly string FilePath;
    int _deleteAttempt;

    public TempFileRef (string filePath) { FilePath = filePath; }

    ~TempFileRef()
    {
        try { File.Delete (FilePath); }
        catch
        {
            if (_deleteAttempt++ < 3) GC.ReRegisterForFinalize (this);
        }
    }
}
```

После третьей неудавшейся попытки финализатор молча отказывается от удаления файла. Мы могли бы расширить такое поведение, скомбинировав его с предыдущим примером — другими словами, после третьего отказа добавить объект в очередь `FailedDeletions`.



Позаботьтесь о том, чтобы метод `ReRegisterForFinalize` вызывался в финализаторе только один раз. Двукратный вызов приведет к тому, что объект будет перерегистрирован дважды и ему придется пройти две дополнительных финализации!

Как работает сборщик мусора

Стандартная среда CLR использует сборщик мусора с поддержкой поколений, пометки и сжатия, который осуществляет автоматическое управление памятью для объектов, хранящихся в управляемой куче. Сборщик мусора считается отслеживающим в том, что он не вмешивается в каждый доступ к объекту, а взамен активизируется периодически и отслеживает граф объектов, хранящихся в управляемой куче, с целью определения объектов, которые могут расцениваться как мусор и потому подвергаться сборке.

Сборщик мусора инициирует процесс сборки при выделении памяти (через ключевое слово `new`) либо после того, как выделенный объем памяти превысил определенный порог, либо в другие моменты, чтобы уменьшить объем памяти, занимаемой приложением. Процесс сборки можно также активизировать вручную, вызвав метод `System.GC.Collect`. Во время сборки мусора все потоки могут быть заморожены (более подробно об этом рассказывается в следующем разделе).

Сборщик мусора начинает со ссылок на корневые объекты и проходит по графу объектов, помечая все затрагиваемые им объекты как достижимые. После завершения процесса все объекты, которые не были помечены, считаются неиспользуемыми и пригодными к сборке мусора.

Неиспользуемые объекты без финализаторов отбрасываются немедленно, а неиспользуемые объекты с финализаторами помещаются в очередь для обработки потоком финализаторов после завершения сборщика мусора. Эти объекты затем становятся пригодными к сборке при следующем запуске сборщика мусора для данного поколения объектов (если только они не будут восстановлены).

Оставшиеся активные объекты затем смещаются в начало кучи (сжимаются), освобождая пространство под дополнительные объекты. Сжатие служит двум целям: оно устраниет фрагментацию памяти и позволяет сборщику мусора при выделении памяти под новые объекты применять очень простую стратегию — всегда выделять память в конце кучи. Подобный подход позволяет избежать выполнения потенциально длительной задачи по ведению списка сегментов свободной памяти.

Если оказывается, что после сборки мусора памяти для размещения нового объекта недостаточно, и операционная система не может выделить дополнительную память, тогда генерируется исключение `OutOfMemoryException`.



Вы можете получать информацию о текущем состоянии управляемой кучи с помощью вызова `GC.GetGCMemoryInfo()`. Начиная с версии .NET 5, этот метод был расширен для возвращения данных, связанных с производительностью.

Приемы оптимизации

Для сокращения времени сборки мусора в сборщике предпринимаются разнообразные приемы оптимизации.

Сборка с учетом поколений

Самая важная оптимизация связана с тем, что сборщик мусора поддерживает поколения. Концепция поколений извлекает преимущества из того факта, что хотя многие объекты распределяются и быстро отбрасываются, некоторые объекты существуют длительное время, поэтому не должны отслеживаться в течение каждой сборки.

По существу сборщик мусора разделяет управляемую кучу на три поколения. Объекты, которые были только что распределены, относятся к поколению Gen0, объекты, выдержавшие один цикл сборки — к поколению Gen1, а все остальные объекты — к поколению Gen2. Поколения Gen0 и Gen1 называются недолговечными.

Среда CLR сохраняет раздел Gen0 относительно небольшим (с типичным размером от сотен килобайтов до нескольких мегабайтов). Когда раздел Gen0 заполняется, сборщик мусора GC вызывает сборку Gen0 — что происходит относительно часто. Сборщик мусора применяет похожий порог памяти к разделу

Gen1 (который действует в качестве буфера для Gen2), поэтому сборки Gen1 тоже являются относительно быстрыми и частыми. Однако полные сборки мусора, включающие Gen2, занимают намного больше времени и в итоге происходят нечасто. Результат полной сборки мусора показан на рис. 12.2.

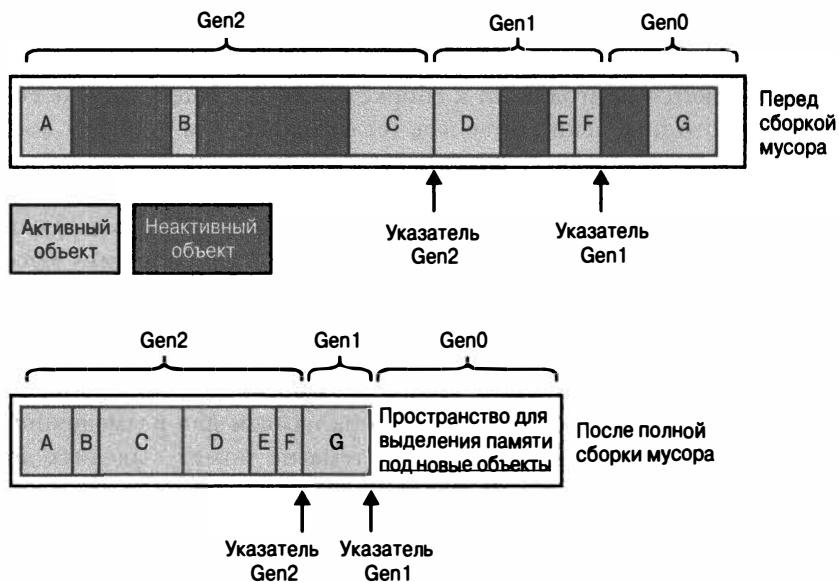


Рис. 12.2. Поколения кучи

Вот очень приблизительные цифры: сборка Gen0 может занимать менее одной миллисекунды, так что заметить ее в типовом приложении нереально. Но полная сборка мусора в программе с крупными графами объектов может длиться примерно 100 мс. Цифры зависят от множества факторов и могут значительно варьироваться — особенно в случае раздела Gen2, размер которого не ограничен (в отличие от разделов Gen0 и Gen1).

В результате кратко живущие объекты очень эффективны в своем использовании сборщика мусора. Экземпляры `StringBuilder`, создаваемые в следующем методе, почти наверняка будут собраны при быстрой сборке Gen0:

```
string Foo()
{
    var sb1 = new StringBuilder ("test");
    sb1.Append (...);
    var sb2 = new StringBuilder ("test");
    sb2.Append (sb1.ToString ());
    return sb2.ToString ();
}
```

Куча для массивных объектов

Для объектов, размеры которых превышают определенный порог (в настоящее время составляющий 85 000 байтов), сборщик мусора применяет отдельную область, которая называется кучей для массивных объектов (Large Object

Heap — LOH). Она позволяет избежать накладных расходов по сжатию крупных объектов, а также избыточных сборок Gen0 — без области LOH распределение последовательности объектов размером 16 Мбайт каждый могло бы приводить к запуску сборки Gen0 после каждого распределения.

По умолчанию область LOH не сжимается, поскольку перемещение крупных блоков памяти во время сборки мусора будет чрезмерно затратным. Отсюда два последствия.

- Выделение памяти может стать медленнее, т.к. сборщик мусора не всегда способен просто распределять объекты в конце кучи — он должен также искать промежутки в середине, что требует поддержки связного списка свободных блоков памяти¹.
- Область LOH подвержена фрагментации. Это значит, что освобождение объекта может привести к возникновению дыры в LOH, которую впоследствии трудно заполнить. Например, дыра, оставленная 86000-байтовым объектом, может быть заполнена только объектом с размером между 85 000 и 86 000 байтов (если только рядом не примыкает еще одна дыра).

Если вы ожидаете проблем с фрагментацией, то можете проинструктировать сборщик мусора о необходимости сжатия области LOH при следующей сборке:

```
GCSettings.LargeObjectHeapCompactionMode =  
    GCLargeObjectHeapCompactionMode.CompactOnce;
```

Еще один обходной путь на тот случай, когда ваша программа часто выделяет память под крупные массивы, предусматривает использование API-интерфейса для работы с пулем массивов (см. раздел “Организация пула массивов” далее в главе).

Куча для массивных объектов не поддерживает концепцию поколений: все объекты трактуются как относящиеся к поколению Gen2.

Сравнение сборки мусора в режиме рабочей станции и в режиме сервера

.NET предлагает два режима сборки мусора: режим рабочей станции и режим сервера. По умолчанию выбирается режим рабочей станции; вы можете переключиться на режим сервера, добавив в файл .csproj приложения следующие строки:

```
<PropertyGroup>  
  <ServerGarbageCollection>true</ServerGarbageCollection>  
</PropertyGroup>
```

При компиляции проекта эта настройка записывается в файл runtimeconfig.json приложения, откуда она читается средой CLR:

```
"runtimeOptions": {  
  "configProperties": {  
    "System.GC.Server": true  
  ...
```

¹ То же самое может иногда возникать в куче с поколениями из-за закрепления (см. раздел “Оператор fixed” в главе 4).

Когда включена сборка мусора в режиме сервера, среда CLR выделяет отдельную кучу и сборщик мусора для каждого ядра. В итоге сборка мусора ускоряется, но потребляет дополнительную память и ресурсы центрального процессора (поскольку каждое ядро требует собственного потока). Если на машине функционирует много других процессов и включена сборка мусора в режиме сервера, то это может привести к “избыточному использованию” центрального процессора, что особенно вредно на рабочих станциях, т.к. из-за него операционная система практически перестает реагировать на запросы.

Сборка мусора в режиме сервера доступна только в многоядерных системах: на одноядерных устройствах (или одноядерных виртуальных машинах) данная настройка игнорируется.

Фоновая сборка мусора

И в режиме рабочей станции, и в режиме сервера среда CLR по умолчанию включает фоновую сборку мусора. Вы можете отключить ее, добавив в файл .csproj приложения следующие строки:

```
<PropertyGroup>
    <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
</PropertyGroup>
```

При компиляции проекта эта настройка записывается в файл .runtimeconfig.json приложения:

```
"runtimeOptions": {
    "configProperties": {
        "System.GC.Concurrent": false,
        ...
    }
}
```

Сборщик мусора обязан замораживать (блокировать) ваши потоки выполнения на период проведения сборки мусора. Фоновая сборка мусора сводит к минимуму такие периоды ожидания, делая ваше приложение более отзывчивым. Это происходит за счет потребления чуть большего объема ресурсов центрального процессора и памяти. Таким образом, отключая фоновую сборку мусора, вы достигаете следующих целей:

- слегка уменьшаете степень использования центрального процессора и памяти;
- увеличиваете паузы (или время ожидания), когда происходит сборка мусора.

Фоновая сборка мусора позволяет вашему прикладному коду выполняться параллельно со сборкой поколения Gen2. (Сборки поколений Gen0 и Gen1 считаются достаточно быстрыми, чтобы иметь возможность извлечь преимущество от такого параллелизма.)

Фоновая сборка мусора является улучшенной версией того, что ранее называлось параллельной сборкой мусора: в ней устранено ограничение, согласно которому параллельная сборка мусора перестает быть параллельной, когда во время выполнения сборки Gen2 заполнился раздел Gen0. В результате приложения, которые постоянно выделяют память, становятся более отзывчивыми.

Уведомления от сборщика мусора

В случае отключения фоновой сборки мусора вы можете запросить у сборщика мусора выдачу уведомления непосредственно перед началом полной (блокирующей) сборки мусора. Это предназначено для конфигураций ферм серверов: идея состоит в переадресации запросов другим серверам прямо перед началом сборки мусора. Затем немедленно инициируется сборка мусора и производится ожидание ее завершения перед переадресацией запросов обратно данному серверу.

Чтобы начать выдачу уведомлений, необходимо вызвать метод `GC.RegisterForFullGCNotification`. Далее потребуется запустить другой поток (см. главу 14), в котором сначала вызывается метод `GC.WaitForFullGCApproach`. Когда этот метод возвратит значение перечисления `GCNotificationStatus`, указывающее на то, что сборка мусора уже близко, можно переадресовывать запросы другим серверам и принудительно запускать сборку мусора вручную (см. следующий раздел). Затем нужно вызвать метод `GC.WaitForFullGCCComplete`: по возвращению управления из данного метода сборка мусора завершена, и можно снова принимать запросы. После этого весь цикл повторяется.

Принудительный запуск сборки мусора

Сборку мусора можно запустить принудительно в любой момент, вызвав метод `GC.Collect`. Вызов `GC.Collect` без аргумента инициирует полную сборку мусора. Если передать целочисленное значение, тогда сборка выполнится для поколений, начиная с `Gen0` и заканчивая поколением, номер которого соответствует указанному значению; таким образом, `GC.Collect(0)` выполняет только быструю сборку `Gen0`.

Обычно наилучшие показатели производительности можно получить, позволив сборщику мусора самостоятельно решать, что именно собирать: принудительная сборка мусора может нанести ущерб производительности за счет излишнего перевода объектов поколения `Gen0` в поколение `Gen1` (и объектов поколения `Gen1` в поколение `Gen2`). Принудительная сборка также нарушит возможность самонастройки сборщика мусора, посредством которой сборщик динамически регулирует пороги для каждого поколения, чтобы добиться максимальной производительности во время работы приложения.

Тем не менее, существуют два исключения. Наиболее распространенным сценарием для вмешательства является ситуация, когда приложение собирается перейти в режим сна на некоторое время: хорошим примером может быть Windows-служба, которая выполняет ежесуточное действие (скажем, проверку обновлений). Такое приложение может использовать объект `System.Timers.Timer` для запуска действия каждые 24 часа. После завершения действия никакой другой код в течение 24 часов выполняться не будет, т.е. в данный период выделение памяти не делается и сборщик мусора не имеет шансов быть активизированным. Сколько бы памяти служба не потребила во время выполнения своего действия, память продолжит быть занятой в течение следующих 24 часов даже при пустом графе объектов! Решение предусматривает вызов метода `GC.Collect` сразу после завершения ежесуточного действия.

Чтобы обеспечить сборку мусора в отношении объектов, для которых она отложена финализаторами, необходимо предпринять дополнительный шаг — вызвать метод `WaitForPendingFinalizers` и запустить сборщик мусора еще раз:

```
GC.Collect();  
GC.WaitForPendingFinalizers();  
GC.Collect();
```

Часто это делается в цикле: действие по выполнению финализаторов может освободить больше объектов, а не только те, которые имеют финализаторы.

Еще один сценарий для вызова метода `GC.Collect` связан с тестированием класса, располагающего финализатором.

Настройка сборки мусора во время выполнения

Статическое свойство `GCSettings.LatencyMode` определяет способ, которым сборщик мусора балансирует задержку и общую эффективность. Изменение значения данного свойства со стандартного `Interactive` на `LowLatency` или `SustainedLowLatency` указывает среде CLR на необходимость применения более быстрых (но более частых) процедур сборки мусора. Это полезно, если приложение нуждается в очень быстром реагировании на события, возникающие в реальном времени. Изменение режима на `Batch` доводит до максимума производительность за счет потенциально низкой скорости реагирования, что полезно для пакетной обработки.

Режим `SustainedLowLatency` не поддерживается, если вы отключили фоновую сборку мусора в файле `.runtimeconfig.json`.

Кроме того, вы можете сообщить среде CLR о том, что сборка мусора должна быть временно приостановлена, вызвав метод `GC.TryStartNoGCRegion`, и затем возобновить ее с помощью метода `GC.EndNoGCRegion`.

Нагрузка на память

Исполняющая среда решает, когда инициировать сборку мусора, на основе нескольких факторов, в числе которых общая загрузка памяти на машине. Если программа выделяет неуправляемую память (см. главу 24), то исполняющая среда получит нереалистично оптимистическое представление об использовании памяти программой, потому что среде CLR известно только об управляемой памяти. Чтобы ослабить такое влияние, можно проинструктировать среду CLR о необходимости учесть выделение указанного объема неуправляемой памяти, вызвав метод `GC.AddMemoryPressure`. Чтобы отменить это (когда неуправляемая память освобождена), потребуется вызвать метод `GC.RemoveMemoryPressure`.

Организация пула массивов

Если ваше приложение часто создает экземпляры массивов, то вы можете избежать большей части накладных расходов, связанных со сборкой мусора, за счет организации пула массивов. Средство пула массивов появилось в .NET Core 3 и работает путем “аренды” массива, который позже возвращается в пул для повторного использования.

Чтобы создать экземпляр массива, вызовите метод Rent класса ArrayPool из пространства имен System.Buffers, указав желаемый размер массива:

```
int[] pooledArray = ArrayPool<int>.Shared.Rent (100); // 100 байтов
```

Такой вызов приведет к выделению массива размером (по крайней мере) 100 байтов из глобального общего пула массивов. Диспетчер пула может предоставить вам массив, размер которого больше запрошенного вами (обычно размер является степенью 2).

Завершив работу с массивом, вызовите метод Return, который вернет массив в пул, позволяя арендовать его снова:

```
ArrayPool<int>.Shared.Return (pooledArray);
```

Вы можете дополнительно передать булевское значение, указывающее диспетчеру пула на необходимость очистки массива перед его возвращением в пул.



Ограничение организации пула массивов связано с отсутствием возможности воспрепятствовать дальнейшему (незаконному) использованию массива после его возвращения в пул, поэтому вы должны внимательно писать код, чтобы избежать такого сценария. Имейте в виду, что вы способны нарушить работу не только своего кода, но и других API-интерфейсов, эксплуатирующих пулы массивов, наподобие ASP.NET Core.

Вместо применения общего пула массивов вы можете создать специальный пул и арендовать массивы из него. Такой подход устраниет риск нарушения работы других API-интерфейсов, но увеличивает общий расход памяти (как и снижает возможности для многократного использования):

```
var myPool = ArrayPool<int>.Create();
int[] array = myPool.Rent (100);
...
```

Утечки управляемой памяти

В неуправляемых языках вроде C++ вы должны не забывать об освобождении памяти вручную, когда объект больше не требуется; в противном случае возникнет утечка памяти. В мире управляемых языков такая ошибка невозможна, поскольку в среде CLR существует система автоматической сборки мусора.

Несмотря на это, крупные и сложные приложения .NET могут демонстрировать аналогичный синдром в легкой форме с тем же самым конечным результатом: с течением времени жизни приложение потребляет все больше и больше памяти до тех пор, пока его не придется перезапустить. Хорошая новость в том, что утечки управляемой памяти обычно легче диагностировать и предотвращать.

Утечки управляемой памяти связаны с неиспользуемыми объектами, которые остаются активными по причине существования неиспользуемых или забытых ссылок на них. Распространенным кандидатом являются обработчики событий — онидерживают ссылку на целевой объект (если только он не является статическим методом). Например, взгляните на следующие классы:

```

class Host
{
    public event EventHandler Click;
}

class Client
{
    Host _host;
    public Client (Host host)
    {
        _host = host;
        _host.Click += HostClicked;
    }
    void HostClicked (object sender, EventArgs e) { ... }
}

```

Приведенный ниже тестовый класс содержит метод, который создает 1000 экземпляров класса Client:

```

class Test
{
    static Host _host = new Host();

    public static void CreateClients()
    {
        Client[] clients = Enumerable.Range (0, 1000)
            .Select (i => new Client (_host))
            .ToArray();
        // Делать что-нибудь с экземплярами класса Client...
    }
}

```

Может показаться, что после того, как метод CreateClients завершит выполнение, тысяча объектов Client станут пригодными для сборки мусора. К сожалению, на каждый объект Client имеется еще одна ссылка: объект _host, событие Click которого теперь ссылается на экземпляр Client. Данный факт может остаться незамеченным, если событие Click не возникает — или если метод HostClicked не делает ничего такого, что привлекало бы внимание.

Один из способов решения проблемы — обеспечить, чтобы класс Client реализовывал интерфейс IDisposable, и в методе Dispose отсоединиться от обработчика событий:

```
public void Dispose() { _host.Click -= HostClicked; }
```

Тогда потребители класса Client освободят его экземпляры после завершения работы с ними:

```
Array.ForEach (clients, c => c.Dispose());
```



В разделе “Слабые ссылки” далее в главе мы опишем другое решение этой проблемы, которое может оказаться удобным в средах, где освобождаемые объекты, как правило, не применяются (примером такой среды может служить WPF). В действительности инфраструктура WPF предлагает класс по имени WeakEventManager, который задействует шаблон использования слабых ссылок.

Таймеры

Забытые таймеры тоже приводят к утечкам памяти (таймеры обсуждаются в главе 21). В зависимости от вида таймера существуют два отличающихся сценария. Давайте сначала рассмотрим таймер в пространстве имен `System.Timers`. В следующем примере класс `Foo` (когда создан его экземпляр) вызывает метод `tmr_Elapsed` каждую секунду:

```
using System.Timers;  
class Foo  
{  
    Timer _timer;  
    Foo()  
    {  
        _timer = new System.Timers.Timer { Interval = 1000 };  
        _timer.Elapsed += tmr_Elapsed;  
        _timer.Start();  
    }  
    void tmr_Elapsed (object sender, ElapsedEventArgs e) { ... }  
}
```

К сожалению, экземпляры `Foo` никогда не смогут быть обработаны сборщиком мусора! Проблема в том, что сама исполняющая среда удерживает ссылки на активные таймеры, чтобы они могли запускать свои события `Elapsed`. Таким образом:

- исполняющая среда будет удерживать `_timer` в активном состоянии;
- `_timer` будет удерживать экземпляр `Foo` в активном состоянии через обработчик событий `tmr_Elapsed`.

Решение станет очевидным, как только вы осознаете, что класс `Timer` реализует интерфейс `IDisposable`. Освобождение таймера останавливает его и гарантирует, что исполняющая среда больше не ссылается на данный объект:

```
class Foo : IDisposable  
{  
    ...  
    public void Dispose() { _timer.Dispose(); }  
}
```



Полезный руководящий принцип предусматривает реализацию интерфейса `IDisposable`, если хоть одному полю в классе присваивается объект, который реализует `IDisposable`.

Касательно того, что уже обсуждалось: таймеры WPF и Windows Forms ведут себя аналогично.

Однако таймер из пространства имен `System.Threading` является особым. Ссылки на активные потоковые таймеры .NET не хранит, а взамен напрямую ссылается на делегаты обратного вызова. Таким образом, если вы забудете освободить потоковый таймер, то может запуститься финализатор, который остановит и освободит таймер автоматически:

```
static void Main()
{
    var tmr = new System.Threading.Timer (TimerTick, null, 1000, 1000);
    GC.Collect ();
    System.Threading.Thread.Sleep (10000); // Ждать 10 секунд
}
static void TimerTick (object notUsed) { Console.WriteLine ("tick"); }
```

Если данный пример скомпилирован в режиме выпуска (отладка отключена, а оптимизация включена), тогда таймер будет обработан сборщиком мусора и финализирован еще до того, как у него появится шанс запуститься хотя бы один раз! И снова мы можем исправить положение, освободив таймер по завершении работы с ним:

```
using (var tmr = new System.Threading.Timer (TimerTick, null, 1000, 1000))
{
    GC.Collect ();
    System.Threading.Thread.Sleep (10000); // Ждать 10 секунд
}
```

Явный вызов метода `tmr.Dispose` в конце блока `using` гарантирует, что переменная `tmr` “используется” и потому не рассматривается сборщиком мусора как неактивная вплоть до конца блока. По иронии судьбы этот вызов метода `Dispose` на самом деле приводит к тому, что объект сохраняется активным дальше!

Диагностика утечек памяти

Простейший способ избежать утечек управляемой памяти предполагает проведение упреждающего мониторинга потребления памяти после того, как приложение написано. Получить данные по текущему использованию памяти объектами программы можно следующим образом (аргумент `true` сообщает сборщику мусора о необходимости выполнения сначала процесса сборки):

```
long memoryUsed = GC.GetTotalMemory (true);
```

Если вы практикуете разработку через тесты, то у вас есть возможность применять модульные тесты для утверждения, что память восстановлена должным образом. Если такое утверждение терпит неудачу, тогда придется проверить только изменения, которые были внесены недавно.

Находить утечки управляемой памяти в крупных приложениях помогает инструмент `windbg.exe`. Доступны также средства с дружественным графическим пользовательским интерфейсом наподобие Microsoft CLR Profiler, SciTech Memory Profiler и Red Gate ANTS Memory Profiler.

Среда CLR также открывает доступ к многочисленным счетчикам событий для помощи в мониторинге потребления ресурсов.

Слабые ссылки

Иногда удобно удерживать ссылку на объект, который является “невидимым” сборщику мусора, в том смысле, что объект сохраняется в активном состоянии. Это называется слабой ссылкой и реализовано классом `System.WeakReference`.

Для использования класса `WeakReference` необходимо сконструировать его экземпляр с целевым объектом:

```
var sb = new StringBuilder ("this is a test");
var weak = new WeakReference (sb);
Console.WriteLine (weak.Target);           // Выводит this is a test
```

Если на целевой объект имеется только одна или более слабых ссылок, то сборщик мусора считает его пригодным для сборки. После того, как целевой объект обработан сборщиком мусора, свойство `Target` экземпляра `WeakReference` получает значение `null`:

```
var weak = GetWeakRef();
GC.Collect();
Console.WriteLine (weak.Target);           // (пусто)
WeakReference GetWeakRef () =>
    new WeakReference (new StringBuilder ("weak"));
```

Во избежание обработки сборщиком мусора целевой объект понадобится присвоить локальной переменной в промежутке между его проверкой на предмет `null` и использованием:

```
var sb = (StringBuilder) weak.Target;
if (sb != null) { /* Делать что-нибудь с sb */ }
```

Поскольку целевой объект был присвоен локальной переменной, он получил надежный корневой объект и потому не может быть обработан сборщиком мусора, пока эта переменная используется.

В приведенном ниже классе слабые ссылки применяются для отслеживания всех создаваемых объектов `Widget`, не препятствуя их обработке сборщиком мусора:

```
class Widget
{
    static List<WeakReference> _allWidgets = new List<WeakReference>();
    public readonly string Name;
    public Widget (string name)
    {
        Name = name;
        _allWidgets.Add (new WeakReference (this));
    }
    public static void ListAllWidgets ()
    {
        foreach (WeakReference weak in _allWidgets)
        {
            Widget w = (Widget)weak.Target;
            if (w != null) Console.WriteLine (w.Name);
        }
    }
}
```

Единственное замечание, которое следует сделать относительно такой системы — с течением времени статический список разрастается и накапливает слабые ссылки с целевыми объектами, установленными в `null`. Таким образом, понадобится внедрить определенную стратегию очистки.

Слабые ссылки и кеширование

Один из сценариев применения WeakReference связан с кешированием крупных графов объектов. Они позволяют интенсивно использующим память данным кешироваться без излишнего потребления памяти:

```
_weakCache = new WeakReference (...);           // _weakCache является полем  
...  
var cache = _weakCache.Target;  
if (cache == null) { /* Пересоздать кеш и присвоить его _weakCache */ }
```

На практике такая стратегия может оказаться не особенно эффективной, потому что вы располагаете лишь небольшим контролем над тем, когда запускается сборщик мусора и какое поколение он выберет для проведения сборки. В частности, если ваш кеш останется в поколении Gen0, то он может быть обработан сборщиком в пределах нескольких микросекунд (не забывайте, что сборщик мусора выполняет свою работу не только тогда, когда памяти становится мало — он производит регулярную сборку и при нормальных условиях потребления памяти). В итоге, как минимум, придется организовать двухуровневый кеш, где процесс начинается с хранения сильных ссылок, которые со временем преобразуются в слабые ссылки.

Слабые ссылки и события

Ранее уже было показано, каким образом события могут вызывать утечки управляемой памяти. Простейшее решение заключается в том, чтобы избегать подписки в таких условиях или реализовать метод Dispose для отмены подписки. Слабые ссылки предлагают еще одно решение.

Предположим, что есть делегат, который удерживает только слабые ссылки на свои целевые объекты. Такой делегат не будет сохранять свои целевые объекты в активном состоянии — если только не существуют независимые ссылки на них. Конечно, при этом нельзя предотвратить ситуацию, когда запущенный делегат сталкивается с висячей ссылкой на целевой объект — в период времени между моментом, когда целевой объект пригоден для сборки мусора, и моментом, когда сборщик мусора подхватит его. Чтобы такое решение было эффективным, код должен быть надежным в указанном сценарии. С учетом данного случая класс слабого делегата может быть реализован так, как показано ниже:

```
public class WeakDelegate<TDelegate> where TDelegate : Delegate  
{  
    class MethodTarget  
    {  
        public readonly WeakReference Reference;  
        public readonly MethodInfo Method;  
        public MethodTarget (Delegate d)  
        {  
            // d.Target будет null для целей в виде статических методов:  
            if (d.Target != null) Reference = new WeakReference (d.Target);  
            Method = d.Method;  
        }  
    }  
}
```

```

List<MethodTarget> _targets = new List<MethodTarget>();
public WeakDelegate()
{
    if (!typeof (TDelegate).IsSubclassOf (typeof (Delegate)))
        throw new InvalidOperationException
            ("TDelegate must be a delegate type");
        // TDelegate должен быть типом делегата
}
public void Combine (TDelegate target)
{
    if (target == null) return;
    foreach (Delegate d in (target as Delegate).GetInvocationList())
        _targets.Add (new MethodTarget (d));
}
public void Remove (TDelegate target)
{
    if (target == null) return;
    foreach (Delegate d in (target as Delegate).GetInvocationList())
    {
        MethodTarget mt = _targets.Find (w =>
            Equals (d.Target, w.Reference?.Target) &&
            Equals (d.Method.MethodHandle, w.Method.MethodHandle));
        if (mt != null) _targets.Remove (mt);
    }
}
public TDelegate Target
{
    get
    {
        Delegate combinedTarget = null;
        foreach (MethodTarget mt in _targets.ToArray())
        {
            WeakReference wr = mt.Reference;
            // Статический целевой объект или активный целевой объект экземпляра
            if (wr == null || wr.Target != null)
            {
                var newDelegate = Delegate.CreateDelegate (
                    typeof(TDelegate), wr?.Target, mt.Method);
                combinedTarget = Delegate.Combine (combinedTarget, newDelegate);
            }
            else
                _targets.Remove (mt);
        }
        return combinedTarget as TDelegate;
    }
    set
    {
        _targets.Clear();
        Combine (value);
    }
}

```

В методах `Combine` и `Remove` мы осуществляли ссылочное преобразование `target` в `Delegate` с помощью операции `as`, а не более привычной операции приведения. Причина в том, что C# запрещает использовать операцию приведения с таким параметром типа, поскольку существует потенциальная неоднозначность между специальным преобразованием и ссылочным преобразованием.

Затем мы вызываем метод `GetInvocationList`, т.к. эти методы могут быть вызваны групповыми делегатами, т.е. делегатами с более чем одним методом для вызова.

В свойстве `Target` мы строим групповой делегат, комбинирующий все делегаты, на которые имеются слабые ссылки с активными целевыми объектами, удаляя оставшиеся (висячие) ссылки из списка `_targets` во избежание его разрастания до бесконечности. (Мы могли бы усовершенствовать наш класс, делая то же самое в методе `Combine`; еще одним улучшением было бы добавление блокировок для обеспечения безопасности в отношении потоков (см. раздел “Блокировка и безопасность потоков” в главе 14).) Мы также разрешаем иметь делегаты вообще без слабой ссылки; они представляют делегаты, целевой метод которых является статическим.

В следующем коде показано, как использовать готовый делегат при реализации события.

```
public class Foo
{
    WeakDelegate<EventHandler> _click = new WeakDelegate<EventHandler>();
    public event EventHandler Click
    {
        add { _click.Combine (value); } remove { _click.Remove (value); }
    }
    protected virtual void OnClick (EventArgs e)
        => _click.Target?.Invoke (this, e);
}
```



Диагностика

Когда что-то пошло не так, важно иметь доступ к информации, которая поможет в диагностировании проблемы. Существенную помощь в этом оказывает IDE-среда или отладчик, но он обычно доступен только на этапе разработки. После поставки приложение обязано самостоятельно собирать и фиксировать диагностическую информацию. Для удовлетворения данного требования .NET предлагает набор средств, которые позволяют регистрировать диагностическую информацию, следить за поведением приложений, обнаруживать ошибки времени выполнения и интегрироваться с инструментами отладки в случае их доступности.

Некоторые диагностические инструменты и API-интерфейсы специфичны для Windows, поскольку они полагаются на функциональные средства операционной системы Windows. Чтобы не допустить загромождения библиотеки .NET BCL специфичными для платформ API-интерфейсами, в Microsoft решили поставлять их в виде отдельных пакетов NuGet, на которые можно дополнительно ссылаться. Существует более десятка специфичных для Windows пакетов; сослаться сразу на все пакеты можно с помощью “главного” пакета Microsoft.Windows.Compatibility.

Типы, рассматриваемые в настоящей главе, определены преимущественно в пространстве имен System.Diagnostics.

Условная компиляция

С помощью директив препроцессора любой раздел кода C# можно компилировать условно. Директивы препроцессора представляют собой специальные инструкции для компилятора, которые начинаются с символа # (и в отличие от других конструкций C# должны полностью располагаться в одной строке). Логически они выполняются перед основной компиляцией (хотя на практике компилятор обрабатывает их во время фазы лексического анализа). Директивами препроцессора для условной компиляции являются #if, #else, #endif и #elif.

Директива #if указывает компилятору на необходимость игнорирования раздела кода, если не определен специальный символ. Определить символ можно в исходном коде с использованием директивы #define (в таком случае символ применяется только к этому файлу) или в файле .csproj, используя элемент <DefineConstants> (в данном случае символ применяется к целой сборке):

```

#define TESTMODE //Директивы #define должны находиться в начале файла.
                // По соглашению имена символов записываются в верхнем регистре.
using System;
class Program
{
    static void Main()
    {
#define TESTMODE
        Console.WriteLine ("in test mode!");      // ВЫВОД: in test mode!
#endif
    }
}

```

Если удалить первую строку, то программа скомпилируется без оператора `Console.WriteLine` в исполняемом файле, как если бы он был закомментирован.

Директива `#else` аналогична оператору `else` языка C#, а директива `#elif` эквивалентна директиве `#else`, за которой следует `#if`. Операции `||`, `&&` и `!` могут использоваться для выполнения операций ИЛИ, И и НЕ:

```

#if TESTMODE && !PLAYMODE                                // если TESTMODE и не PLAYMODE
...

```

Однако помните, что вы не строите обычное выражение C#, а символы, над которыми вы оперируете, не имеют никакого отношения к переменным — статическим или любым другим.

Вы можете определять символы, которые применяются к каждому файлу в сборке, редактируя файл `.csproj` (или в Visual Studio на вкладке Build (Сборка) диалогового окна Project Properties (Свойства проекта)). Ниже определены две константы, `TESTMODE` и `PLAYMODE`:

```

<PropertyGroup>
    <DefineConstants>TESTMODE;PLAYMODE</DefineConstants>
</PropertyGroup>

```

Если вы определили символ на уровне сборки и затем хотите отменить его определение для какого-то файла, тогда применяйте директиву `#undef`.

Сравнение условной компиляции и статических переменных-флагов

Предыдущий пример взамен можно было бы реализовать с использованием простого статического поля:

```

static internal bool TestMode = true;

static void Main()
{
    if (TestMode) Console.WriteLine ("in test mode!");
}

```

Преимущество такого подхода связано с возможностью конфигурирования во время выполнения. Итак, почему выбирают условную компиляцию? Причина в том, что условная компиляция способна решать задачи, которые нельзя решить посредством переменных-флагов, такие как:

- условное включение атрибута;
- изменение типа, объявляемого для переменной;
- переключение между разными пространствами имен или псевдонимами типов в директиве `using`; например:

```
using TestType =
    #if V2
        MyCompany.Widgets.GadgetV2;
    #else
        MyCompany.Widgets.Gadget;
    #endif
```

Под директивой условной компиляции можно даже реализовать крупную перестройку кода, так что появится возможность немедленного переключения между старой версией и новой. Вдобавок можно создавать библиотеки, которые компилируются для нескольких версий исполняющей среды, используя в своих интересах самые новые функциональные средства, когда они доступны.

Еще одно преимущество условной компиляции связано с тем, что отладочный код может ссылаться на типы в сборках, которые не включаются при развертывании.

Атрибут `Conditional`

Атрибут `Conditional` указывает компилятору на необходимость игнорирования любых обращений к определенному классу или методу, если заданный символ не был определен.

Чтобы выяснить, насколько это полезно, представим, что мы реализуем метод для регистрации информации о состоянии следующим образом:

```
static void LogStatus (string msg)
{
    string logFilePath = ...
    System.IO.File.AppendAllText (logFilePath, msg + "\r\n");
}
```

Теперь предположим, что его нужно выполнять, только если определен символ `LOGGINGMODE`. Первое решение предусматривает помещение всех вызовов метода `LogStatus` внутрь директивы `#if`:

```
#if LOGGINGMODE
LogStatus ("Message Headers: " + GetMsgHeaders());
#endif
```

Результат получается идеальным, но постоянно писать такой код утомительно. Второе решение заключается в помещении директивы `#if` внутрь самого метода `LogStatus`. Однако это проблематично, поскольку `LogStatus` должен вызываться так:

```
LogStatus ("Message Headers: " + GetComplexMessageHeaders());
```

Метод `GetComplexMessageHeaders` будет вызываться всегда, что приведет к снижению производительности.

Мы можем скомбинировать функциональность первого решения с удобством второго, присоединив к методу LogStatus атрибут Conditional (который определен в пространстве имен System.Diagnostics):

```
[Conditional ("LOGGINGMODE")]
static void LogStatus (string msg)
{
    ...
}
```

В результате компилятор трактует вызовы LogStatus, как если бы они были помещены внутрь директивы #if LOGGINGMODE. Когда символ не определен, любые обращения к методу LogStatus полностью исключаются из процесса компиляции, в том числе и выражения оценки его аргумента. (Следовательно, будут пропускаться любые выражения, дающие побочные эффекты.) Такой прием работает, даже если метод LogStatus и вызывающий класс находятся в разных сборках.



Еще одно преимущество конструкции [Conditional] в том, что условная проверка выполняется, когда компилируется *вызывающий класс*, а не *вызываемый метод*. Это удобно, т.к. позволяет написать библиотеку, которая содержит методы вроде LogStatus, и построить только одну версию данной библиотеки.

Во время выполнения атрибут Conditional игнорируется — он представляет собой исключительно инструкцию для компилятора.

Альтернативы атрибуту Conditional

Атрибут Conditional бесполезен, когда во время выполнения необходима возможность динамического включения или отключения функциональности: вместо него должен применяться подход на основе переменных. Остается открытый вопрос о том, как элегантно обойти оценку аргументов при вызове условных методов регистрации. Проблема решается с помощью функционального подхода:

```
using System;
using System.Linq;
class Program
{
    public static bool EnableLogging;
    static void LogStatus (Func<string> message)
    {
        string filePath = ...
        if (EnableLogging)
            System.IO.File.AppendAllText (filePath, message() + "\r\n");
    }
}
```

Лямбда-выражение позволяет вызывать данный метод без разбужения синтаксиса:

```
LogStatus ( () => "Message Headers: " + GetComplexMessageHeaders() );
```

Если значение `EnableLogging` равно `false`, тогда вызов метода `GetComplexMessageHeaders` никогда не выполняется.

Классы `Debug` и `Trace`

`Debug` и `Trace` — статические классы, которые предлагают базовые возможности регистрации и утверждений. Указанные два класса очень похожи; основное отличие связано с тем, для чего они предназначены. Класс `Debug` предназначен для отладочных сборок, а класс `Trace` — для отладочных и окончательных сборок. Чтобы достичь таких целей:

- все методы класса `Debug` определены с атрибутом `[Conditional ("DEBUG")]`;
- все методы класса `Trace` определены с атрибутом `[Conditional ("TRACE")]`.

Это означает, что если не определен символ `DEBUG` или `TRACE`, то компилятор исключает все обращения к `Debug` или `Trace`. (Среда Visual Studio на вкладке `Build` диалогового окна `Project Properties` предлагает флагги для определения этих символов и в новых проектах по умолчанию включает символ `TRACE`.)

Классы `Debug` и `Trace` предоставляют методы `Write`, `WriteLine` и `WriteIf`. По умолчанию они отправляют сообщения в окно вывода отладчика:

```
Debug.Write      ("Data");
Debug.WriteLine (23 * 34);
int x = 5, y = 3;
Debug.WriteLineIf (x > y, "x is greater than y");
```

Класс `Trace` также предлагает методы `TraceInformation`, `TraceWarning` и `TraceError`. Отличия в поведении между ними и методами `Write` зависят от активных прослушивателей `TraceListener` (мы рассмотрим их в разделе “`TraceListener`” далее в главе).

`Fail` и `Assert`

Классы `Debug` и `Trace` предоставляют методы `Fail` и `Assert`. Метод `Fail` отправляет сообщение всем экземплярам `TraceListener` из коллекции `Listeners` внутри класса `Debug` или `Trace` (как будет показано в следующем разделе), которые по умолчанию записывают переданное сообщение в вывод отладки, а также отображают его в диалоговом окне:

```
Debug.Fail ("File data.txt does not exist!"); // Файл data.txt не существует!
```

Метод `Assert` просто вызывает метод `Fail`, если аргумент типа `bool` равен `false`; это называется созданием утверждения и указывает на ошибку в коде, если оно нарушено. Можно также задать необязательное сообщение об ошибке:

```
Debug.Assert (File.Exists ("data.txt"), "File data.txt does not exist!");
var result = ...
Debug.Assert (result != null);
```

Методы `Write`, `Fail` и `Assert` также перегружены, чтобы в добавок к сообщению принимать строковую категорию, которая может быть полезна при обработке вывода.

Альтернативой утверждению будет генерация исключения, если противоположное условие равно `true`. Это общепринятый подход при проверке достоверности аргументов метода:

```
public void ShowMessage (string message)
{
    if (message == null) throw new ArgumentNullException ("message");
    ...
}
```

Такие “утверждения” компилируются безусловным образом и обладают меньшей гибкостью в том, что не позволяют управлять результатом отказанного утверждения через экземпляры `TraceListener`. К тому же формально они не являются утверждениями. Утверждение представляет собой то, что в случае нарушения говорит об ошибке в коде текущего метода. Генерация исключения на основе проверки достоверности аргумента указывает на ошибку в коде вызывающего компонента.

TraceListener

Класс `Trace` имеет статическое свойство `Listeners`, которое возвращает коллекцию экземпляров `TraceListener`. Они отвечают за обработку содержимого, выпускаемого методами `Write`, `Fail` и `Trace`.

По умолчанию коллекция `Listeners` включает единственный прослушиватель (`DefaultTraceListener`). Стандартный прослушиватель обладает двумя основными возможностями.

- В случае подключения к отладчику наподобие встроенного в Visual Studio сообщения записываются в окно вывода отладки; иначе содержимое сообщения игнорируется.
- Когда вызывается метод `Fail` (или утверждение не выполняется), приложение прекращает работу.

Вы можете изменить такое поведение, (необязательно) удалив стандартный прослушиватель и затем добавив один или большее число собственных прослушивателей. Прослушиватели трассировок можно написать с нуля (создавая подкласс класса `TraceListener`) или воспользоваться одним из предопределенных типов:

- `TextWriterTraceListener` записывает в `Stream` или `TextWriter` либо добавляет в файл;
- `EventLogTraceListener` записывает в журнал событий Windows (только Windows);
- `EventProviderTraceListener` записывает в подсистему трассировки событий для Windows (Event Tracing for Windows — ETW; межплатформенная поддержка).

Класс `TextWriterTraceListener` имеет подклассы `ConsoleTraceListener`, `DelimitedListTraceListener`, `XmlWriterTraceListener` и `EventSchemaTraceListener`.

В следующем примере очищается стандартный прослушиватель `Trace`, после чего добавляются три прослушивателя — первый дописывает в файл, второй выводит на консоль и третий записывает в журнал событий Windows:

```
// Очистить стандартный прослушиватель:  
Trace.Listeners.Clear();  
  
// Добавить средство записи, дописывающее в файл trace.txt:  
Trace.Listeners.Add (new TextWriterTraceListener ("trace.txt"));  
  
// Получить выходной поток Console и добавить его в качестве прослушивателя:  
System.IO.TextWriter tw = Console.Out;  
Trace.Listeners.Add (new TextWriterTraceListener (tw));  
  
// Настроить исходный файл журнала событий и создать/добавить прослушиватель.  
// Метод CreateEventSource требует повышения полномочий до уровня администратора,  
// так что это обычно будет делаться при установке приложения.  
if (!EventLog.SourceExists ("DemoApp"))  
    EventLog.CreateEventSource ("DemoApp", "Application");  
  
Trace.Listeners.Add (new EventLogTraceListener ("DemoApp"));
```

В случае журнала событий Windows сообщения, записываемые с помощью метода `Write`, `Fail` или `Assert`, всегда отображаются в программе “Просмотр событий” как сообщения уровня сведений. Однако сообщения, которые записываются посредством методов `TraceWarning` и `TraceError`, отображаются как предупреждения или ошибки.

Класс `TraceListener` также имеет свойство `Filter` типа `TraceFilter`, которое можно устанавливать для управления тем, будет ли сообщение записано данным прослушивателем. Чтобы сделать это, нужно либо создать экземпляр одного из предопределенных подклассов (`EventTypeFilter` или `SourceFilter`), либо создать подкласс класса `TraceFilter` и переопределить метод `ShouldTrace`. Подобный прием можно использовать, скажем, для фильтрации по категории.

В классе `TraceListener` также определены свойства `IndentLevel` и `IndentSize` для управления отступами и свойство `TraceOutputOptions` для записи дополнительных данных:

```
TextWriterTraceListener tl = new TextWriterTraceListener (Console.Out);  
tl.TraceOutputOptions = TraceOptions.DateTime | TraceOptions.Callstack;
```

Свойство `TraceOutputOptions` применяется при использовании методов `Trace`:

```
Trace.TraceWarning ("Orange alert");
```

Бот вывод:

```
DiagTest.vshost.exe Warning: 0 : Orange alert  
DateTime=2018-03-08T05:57:13.625000Z  
Callstack= at System.Environment.GetStackTrace(Exception e,  
Boolean needFileInfo)  
at System.Environment.get_StackTrace() at ...
```

Сброс и закрытие прослушивателей

Некоторые прослушиватели, такие как `TextWriterTraceListener`, в итоге производят запись в поток, подлежащий кешированию. Результатом будут два последствия.

- Сообщение может не появиться в выходном потоке или файле немедленно.
- Перед завершением приложения прослушиватель потребуется закрыть (или, по крайней мере, сбросить); в противном случае потеряется все то, что находится в кеше (по умолчанию до 4 Кбайт данных, если осуществляется запись в файл).

Классы `Trace` и `Debug` предлагают статические методы `Close` и `Flush`, которые вызывают `Close` или `Flush` на всех прослушивателях (а эти методы в свою очередь вызывают `Close` или `Flush` на любых лежащих в основе средствах записи и потоках). Метод `Close` неявно вызывает `Flush`, закрывает файловые дескрипторы и предотвращает дальнейшую запись данных.

В качестве общего правила: метод `Close` должен вызываться перед завершением приложения, а метод `Flush` — каждый раз, когда нужно удостовериться, что текущие данные сообщений записаны. Такое правило применяется при использовании прослушивателей, основанных на потоках или файлах.

Классы `Trace` и `Debug` также предоставляют свойство `AutoFlush`, которое в случае равенства `true` приводит к вызову метода `Flush` после каждого сообщения.



Если применяются прослушиватели, основанные на потоках или файлах, то эффективной политикой будет установка в `true` свойства `AutoFlush` для экземпляров `Debug` и `Trace`. Иначе при возникновении исключения или критической ошибки последние 4 Кбайт диагностической информации могут быть утеряны.

Интеграция с отладчиком

Иногда для приложения удобно взаимодействовать с каким-нибудь отладчиком, если он доступен. На этапе разработки отладчик обычно предоставляется IDE-средой (например, `Visual Studio`), а после развертывания отладчиком, вероятно, будет один из низкоуровневых инструментов отладки, такой как `WinDbg`, `Cordbg` или `MDbg`.

Присоединение и останов

Статический класс `Debugger` из пространства имен `System.Diagnostics` предлагает базовые функции для взаимодействия с отладчиком, а именно — `Break`, `Launch`, `Log` и `IsAttached`.

Для отладки к приложению сначала потребуется присоединить отладчик. В случае запуска приложения из IDE-среды отладчик присоединяется автоматически, если только не запрошено противоположное (выбором пункта меню `Start without debugging` (Запустить без отладки)). Однако иногда запускать приложение

в режиме отладки внутри IDE-среды неудобно или невозможно. Примером может быть приложение Windows-службы или (по иронии судьбы) визуальный редактор Visual Studio. Одно из решений предполагает запуск приложения обычным образом с последующим выбором пункта меню Debug Process (Отладить процесс) в IDE-среде. Тем не менее, при таком подходе нет возможности поместить точку останова в самое начало процесса выполнения программы.

Обходной путь предусматривает вызов метода Debugger.Break внутри приложения. Данный метод запускает отладчик, присоединяется к нему и приостанавливает выполнение в точке вызова. (Метод Launch делает то же самое, но не приостанавливает выполнение.) После присоединения с помощью метода Log сообщения можно отправлять прямо в окно вывода отладчика. Состояние присоединения к отладчику выясняется через свойство IsAttached.

Атрибуты отладчика

Атрибуты DebuggerStepThrough и DebuggerHidden предоставляют указания отладчику о том, как обрабатывать пошаговое выполнение для конкретного метода, конструктора или класса.

Атрибут DebuggerStepThrough требует, чтобы отладчик прошел через функцию без взаимодействия с пользователем. Данный атрибут полезен для автоматически сгенерированных методов и прокси-методов, которые переключают выполнение реальной работы на какие-то другие методы. В последнем случае отладчик будет отображать прокси-метод в стеке вызовов, даже когда точка останова находится внутри “реального” метода — если только не добавить также атрибут DebuggerHidden. Упомянутые атрибуты можно комбинировать на прокси-методах, чтобы помочь пользователю сосредоточить внимание на отладке прикладной логики, а не связующего вспомогательного кода:

```
[DebuggerStepThrough, DebuggerHidden]
void DoWorkProxy()
{
    // Настройка...
    DoWork();
    // Освобождение...
}
void DoWork() {...} // Реальный метод...
```

Процессы и потоки процессов

В последнем разделе главы 6 приводились объяснения, как запустить новый процесс с помощью метода Process.Start. Класс Process также позволяет запрашивать и взаимодействовать с другими процессами, выполняющимися на том же самом или другом компьютере. Класс Process является частью .NET Standard 2.0, хотя для платформы UWP его возможности ограничены.

Исследование выполняющихся процессов

Методы Process.GetProcessXXX извлекают специфический процесс по имени либо идентификатору или все процессы, выполняющиеся на текущей

либо указанной машине. Сюда входят как управляемые, так и неуправляемые процессы. Каждый экземпляр Process имеет множество свойств, отражающих статистические сведения, такие как имя, идентификатор, приоритет, потребление памяти и процессора, оконные дескрипторы и т.д. В следующем примере производится перечисление всех процессов, функционирующих на текущем компьютере:

```
foreach (Process p in Process.GetProcesses())
using (p)
{
    Console.WriteLine (p.ProcessName);
    Console.WriteLine ("    PID:      " + p.Id);           // Идентификатор процесса
    Console.WriteLine ("    Memory:   " + p.WorkingSet64); // Память
    Console.WriteLine ("    Threads:  " + p.Threads.Count); // Количество потоков
}
```

Метод Process.GetCurrentProcess возвращает текущий процесс.
Завершить процесс можно вызовом его метода Kill.

Исследование потоков в процессе

С помощью свойства Process.Threads можно также реализовать перечисление потоков других процессов. Однако вместо объектов System.Threading.Thread будут получены объекты ProcessThread, которые предназначены для решения административных задач, а не задач, касающихся синхронизации. Объект ProcessThread предоставляет диагностическую информацию о лежащем в основе потоке и позволяет управлять некоторыми связанными с ним аспектами, такими как приоритет и родство:

```
public void EnumerateThreads (Process p)
{
    foreach (ProcessThread pt in p.Threads)
    {
        Console.WriteLine (pt.Id);
        Console.WriteLine ("    State:     " + pt.ThreadState);       // Состояние
        Console.WriteLine ("    Priority:  " + pt.PriorityLevel); // Приоритет
        Console.WriteLine ("    Started:   " + pt.StartTime);        // Запущен
        Console.WriteLine ("    CPU time: " + pt.TotalProcessorTime); // Время ЦП
    }
}
```

StackTrace и StackFrame

Классы StackTrace и StackFrame выдают допускающее только чтение представление стека вызовов. Трассировки стека можно получать для текущего потока или для объекта Exception. Такая информация полезна в основном для диагностических целей, хотя ее также можно использовать и в программировании (как ловкий прием). Экземпляр StackTrace представляет полный стек вызовов, а StackFrame — одиночный вызов метода внутри стека.



Если вам просто нужно узнать имя и номер строки вызывающего метода, то более легкой и быстрой альтернативой будут атрибуты информации о вызывающем компоненте. Эта тема раскрывалась в разделе “Атрибуты информации о вызывающем компоненте” главы 4.

Если экземпляр `StackTrace` создается без аргументов (или с аргументом типа `bool`), тогда будет получен снимок стека вызовов текущего потока. Когда аргумент типа `bool` равен `true`, он инструктирует `StackTrace` о необходимости чтения файлов .pdb (project debug — отладка проекта) сборки, если они существуют, предоставляя доступ к данным об именах файлов, номерах строк и позициях в строках. Файлы отладки проекта генерируются в случае компиляции с ключом `/debug`. (Среда Visual Studio компилирует с этим ключом, если не затребовано построение окончательной сборки через дополнительные параметры построения (в диалоговом окне `Advanced Build Settings` (Дополнительные настройки отладки)).) После получения экземпляра `StackTrace` можно исследовать любой отдельный фрейм с помощью вызова метода `GetFrame` или же все фреймы посредством вызова `GetFrames`:

```
static void Main() { A (); }
static void A()    { B (); }
static void B()    { C (); }
static void C()

{
    StackTrace s = new StackTrace (true);
    Console.WriteLine ("Total frames: " + s.FrameCount);
        // Всего фреймов
    Console.WriteLine ("Current method: " + s.GetFrame(0).GetMethod().Name);
        // Текущий метод
    Console.WriteLine ("Calling method: " + s.GetFrame(1).GetMethod().Name);
        // Вызывающий метод
    Console.WriteLine ("Entry method: " + s.GetFrame
        (s.FrameCount-1).GetMethod().Name); // Входной метод
    Console.WriteLine ("Call Stack:");
        // Стек вызовов
    foreach (StackFrame f in s.GetFrames())
        Console.WriteLine (
            " File: " + f.GetFileName() + /* Файл */
            " Line: " + f.GetFileLineNumber() + /* Страна */
            " Col: " + f.GetFileColumnNumber() + /* Колонка */
            " Offset: " + f.GetILOffset() + /* Смещение */
            " Method: " + f.GetMethod().Name); /* Метод */
}
```

Ниже показан вывод:

```
Total frames: 4
Current method: C
Calling method: B
Entry method: Main
Call stack:
File: C:\Test\Program.cs Line: 15 Col: 4 Offset: 7 Method: C
File: C:\Test\Program.cs Line: 12 Col: 22 Offset: 6 Method: B
File: C:\Test\Program.cs Line: 11 Col: 22 Offset: 6 Method: A
File: C:\Test\Program.cs Line: 10 Col: 25 Offset: 6 Method: Main
```



Смещение IL указывает смещение инструкции, которая будет выполнена *следующей*, а не той, что выполняется в текущий момент. Тем не менее, номера строк и колонок (при наличии файла .pdb) обычно отражают действительную точку выполнения.

Так происходит оттого, что среда CLR делает все возможное для выведения фактической точки выполнения при вычислении строки и колонки из смещения IL. Компилятор генерирует код IL так, чтобы сделать это реальным, при необходимости вставляя в поток IL инструкции пор (no-operation — нет операции).

Однако компиляция с включенной оптимизацией запрещает вставку инструкций пор, а потому трассировка стека может отражать номера строки и колонки, где расположен оператор, который будет выполняться следующим. Получение удобной трассировки стека еще более затрудняется тем фактом, что оптимизация может быть связана и с применением других трюков, таких как устранение целых методов.

Сокращенный способ получения важной информации для полного экземпляра StackTrace предусматривает вызов на нем метода `ToString`. Вот как могут выглядеть результаты:

```
at DebugTest.Program.C() in C:\Test\Program.cs:line 16
at DebugTest.Program.B() in C:\Test\Program.cs:line 12
at DebugTest.Program.A() in C:\Test\Program.cs:line 11
at DebugTest.Program.Main() in C:\Test\Program.cs:line 10
```

Трассировку стека можно также получить для объекта `Exception`, передав его конструктору `StackTrace` (она покажет, что именно привело к генерации исключения).



Класс `Exception` уже имеет свойство `StackTrace`; тем не менее, оно возвращает простую строку, а не объект `StackTrace`. Объект `StackTrace` намного более полезен при регистрации исключений, возникающих после развертывания (когда файлы .pdb уже не доступны), поскольку вместо номеров строк и колонок в журнале можно регистрировать *смещение IL*. С помощью смещения IL и утилиты `ildasm` несложно выяснить, внутри какого метода возникла ошибка.

Журналы событий Windows

Платформа Win32 предоставляет централизованный механизм регистрации в форме журналов событий Windows.

Применяемые ранее классы `Debug` и `Trace` осуществляли запись в журнал событий Windows, если был зарегистрирован прослушиватель `EventLogTraceListener`. Тем не менее, посредством класса `EventLog` можно записывать напрямую в журнал событий Windows, не используя классы `Trace` или `Debug`. Класс `EventLog` можно также применять для чтения и мониторинга данных, связанных с событиями.



Выполнять запись в журнал событий Windows имеет смысл в приложении Windows-службы, поскольку если что-то идет не так, то нет никакой возможности отобразить пользовательский интерфейс, который направил бы пользователя на специфический файл, куда была занесена диагностическая информация. Кроме того, запись в журнал событий Windows является общепринятой практикой для служб, так что данный журнал будет первым местом, где администратор начнет выяснять причины отказа той или иной службы.

Существуют три стандартных журнала событий Windows со следующими именами:

- Application (приложение)
- System (система)
- Security (безопасность)

Большинство приложений обычно производят запись в журнал Application.

Запись в журнал событий

Ниже перечислены шаги, которые понадобится выполнить для записи в журнал событий Windows.

1. Выберите один из трех журналов событий (обычно Application).
2. Примите решение относительно имени источника и при необходимости создайте его (создание требует наличия прав администратора).
3. Вызовите метод EventLog.WriteEntry с именем журнала, именем источника и данными сообщения.

Имя источника — это просто идентифицируемое имя вашего приложения. Имя источника перед использованием должно быть зарегистрировано; такую функцию выполняет метод CreateEventSource. Затем можно вызывать метод WriteEntry:

```
const string SourceName = "MyCompany.WidgetServer";
// Метод CreateEventSource требует наличия прав администратора,
// поэтому данный код обычно выполняется при установке приложения
if (!EventLog.SourceExists (SourceName))
    EventLog.CreateEventSource (SourceName, "Application");
EventLog.WriteEntry (SourceName,
    "Service started; using configuration file=...",
    EventLogEntryType.Information);
```

Перечисление EventLogEntryType содержит следующие значения: Information, Warning, Error, SuccessAudit и FailureAudit. Каждое значение обеспечивает отображение отличающегося значка в программе просмотра событий Windows. Можно также указать необязательные категорию и идентификатор события (произвольные числа по вашему выбору) и предоставить дополнительные двоичные данные.

Метод CreateEventSource также позволяет задавать имя машины, что приведет к записи в журнал событий на другом компьютере при наличии достаточных полномочий.

Чтение журнала событий

Для чтения журнала событий необходимо создать экземпляр класса EventLog с именем нужного журнала и дополнительно именем компьютера, если журнал находится на другом компьютере. Впоследствии любая запись журнала может быть прочитана с помощью свойства Entries типа коллекции:

```
EventLog log = new EventLog ("Application");
Console.WriteLine ("Total entries: " + log.Entries.Count); // Всего записей
EventLogEntry last = log.Entries [log.Entries.Count - 1];
Console.WriteLine ("Index: " + last.Index); // Индекс
Console.WriteLine ("Source: " + last.Source); // Источник
Console.WriteLine ("Type: " + last.EntryType); // Тип
Console.WriteLine ("Time: " + last.TimeWritten); // Время
Console.WriteLine ("Message: " + last.Message); // Сообщение
```

С помощью статического метода EventLog.GetEventLogs можно перечислить все журналы на текущем (или другом) компьютере (для полного доступа требуются права администратора):

```
foreach (EventLog log in EventLog.GetEventLogs())
    Console.WriteLine (log.LogDisplayName);
```

Обычно данный код выводит минимум Application, Security и System.

Мониторинг журнала событий

Организовать оповещение о появлении любой записи в журнале событий Windows можно посредством события EntryWritten. Прием работает для журналов событий на локальном компьютере независимо от того, какое приложение записало событие.

Чтобы включить мониторинг журнала событий, необходимо выполнить следующие действия.

1. Создайте экземпляр EventLog и установите его свойство EnableRaisingEvents в true.
2. Обработайте событие EntryWritten.

Вот пример:

```
using (var log = new EventLog ("Application"))
{
    log.EnableRaisingEvents = true;
    log.EntryWritten += DisplayEntry;
    Console.ReadLine();
}
void DisplayEntry (object sender, EntryWrittenEventArgs e)
{
    EventLogEntry entry = e.Entry;
    Console.WriteLine (entry.Message);
}
```

Счетчики производительности



Счетчики производительности являются средством, предназначенным только для Windows, и требуют загрузки NuGet-пакета `System.Diagnostics.PerformanceCounter`. Если в качестве целевой платформы выбрана система Linux или macOS, тогда ознакомьтесь с альтернативами в разделе “Межплатформенные инструменты диагностики” далее в главе.

Обсуждаемые ранее механизмы регистрации удобны для накопления информации, которая будет анализироваться в будущем. Однако чтобы получить представление о текущем состоянии приложения (или системы в целом), необходим какой-то подход реального времени. Решением такой потребности в Win32 является инфраструктура для мониторинга производительности, которая состоит из набора счетчиков производительности, открываемых системой и приложениями, и оснасток консоли управления Microsoft (Microsoft Management Console — MMC), используемых для отслеживания этих счетчиков в реальном времени.

Счетчики производительности сгруппированы в категории, такие как “Система”, “Процессор”, “Память .NET CLR” и т.д. В инструментах с графическим пользовательским интерфейсом такие категории иногда называются “объектами производительности”. Каждая категория группирует связанный набор счетчиков производительности, отслеживающих один аспект системы или приложения. Примерами счетчиков производительности в категории “Память .NET CLR” могут быть “% времени сборки мусора”, “# байтов во всех кучах” и “Выделено байтов/с”.

Каждая категория может дополнительно иметь один или более экземпляров, допускающих независимый мониторинг. Это полезно, например, для счетчика производительности “% процессорного времени” из категории “Процессор”, который позволяет отслеживать использование центрального процессора. На многопроцессорной машине данный счетчик поддерживает экземпляры для всех процессоров, позволяя независимо проводить мониторинг загрузки каждого процессора.

В последующих разделах будет показано, как решать часто встречающиеся задачи, такие как определение открытых счетчиков, отслеживание счетчиков и создание собственных счетчиков для отображения информации о состоянии приложения.



В зависимости от того, к чему производится доступ, чтение счетчиков производительности или категорий может требовать наличия прав администратора на локальном или целевом компьютере.

Перечисление доступных счетчиков производительности

В следующем примере осуществляется перечисление всех доступных счетчиков производительности на компьютере. В случае если счетчик имеет экземпляры, тогда перечисляются счетчики для каждого экземпляра:

```

PerformanceCounterCategory[] cats =
    PerformanceCounterCategory.GetCategories();
foreach (PerformanceCounterCategory cat in cats)
{
    Console.WriteLine ("Category: " + cat.CategoryName);           // Категория
    string[] instances = cat.GetInstanceNames();
    if (instances.Length == 0)
    {
        foreach (PerformanceCounter ctr in cat.GetCounters())
            Console.WriteLine (" Counter: " + ctr.CounterName);      // Счетчик
    }
    else // Вывести счетчики, имеющие экземпляры
    {
        foreach (string instance in instances)
        {
            Console.WriteLine (" Instance: " + instance);          // Экземпляр
            if (cat.InstanceExists (instance))
                foreach (PerformanceCounter ctr in cat.GetCounters (instance))
                    Console.WriteLine ("     Counter: " + ctr.CounterName); // Счетчик
        }
    }
}

```



Результат содержит свыше 10 000 строк! К тому же его получение занимает некоторое время, поскольку реализация метода `PerformanceCounterCategory.InstanceExists` неэффективна. В реальной системе настолько детальная информация извлекается только по требованию.

В приведенном далее примере с помощью LINQ извлекаются лишь счетчики производительности, связанные с .NET, а результат помещается в XML-файл:

```

var x =
    new XElement ("counters",
        from PerformanceCounterCategory cat in
            PerformanceCounterCategory.GetCategories()
        where cat.CategoryName.StartsWith (".".NET")
        let instances = cat.GetInstanceNames()
        select new XElement ("category",
            new XAttribute ("name", cat.CategoryName),
            instances.Length == 0
            ?
            from c in cat.GetCounters()
            select new XElement ("counter",
                new XAttribute ("name", c.CounterName))
            :
            from i in instances
            select new XElement ("instance", new XAttribute ("name", i),
                !cat.InstanceExists (i)
                ?
                null
                :
            )
    )

```

```

        from c in cat.GetCounters (i)
        select new XElement ("counter",
            new XAttribute ("name", c.CounterName))
    )
);
x.Save ("counters.xml");

```

Чтение данных счетчика производительности

Чтобы извлечь значение счетчика производительности, необходимо создать объект `PerformanceCounter` и затем вызвать его метод `NextValue` или `NextSample`. Метод `NextValue` возвращает простое значение типа `float`, а метод `NextSample` — объект `CounterSample`, который открывает доступ к более широкому набору свойств наподобие `CounterFrequency`, `TimeStamp`, `BaseValue` и `RawValue`.

Конструктор `PerformanceCounter` принимает имя категории, имя счетчика и необязательный экземпляр. Таким образом, чтобы отобразить сведения о текущем использовании всех процессоров, потребуется написать следующий код:

```

using PerformanceCounter pc = new PerformanceCounter ("Processor",
    "% Processor Time",
    "_Total");

Console.WriteLine (pc.NextValue());

```

А вот как отобразить данные о потреблении “реальной” (т.е. закрытой) памяти текущим процессом:

```

string procName = Process.GetCurrentProcess ().ProcessName;
using PerformanceCounter pc = new PerformanceCounter ("Process",
    "Private Bytes",
    procName);

Console.WriteLine (pc.NextValue());

```

Класс `PerformanceCounter` не открывает доступ к событию `ValueChanged`, поэтому для отслеживания изменений потребуется реализовать опрос. В следующем примере опрос производится каждые 200 мс — пока не поступит сигнал завершения от `EventWaitHandle`:

```

// Необходимо импортировать пространства имен System.Threading
и System.Diagnostics

static void Monitor (string category, string counter, string instance,
    EventWaitHandle stopper)
{
    if (!PerformanceCounterCategory.Exists (category))
        throw new InvalidOperationException ("Category does not exist");
        // Категория не существует
    if (!PerformanceCounterCategory.CounterExists (counter, category))
        throw new InvalidOperationException ("Counter does not exist");
        // Счетчик не существует
    if (instance == null) instance = ""; // == экземпляры отсутствуют (не null!)
    if (instance != "" &&
        !PerformanceCounterCategory.InstanceExists (instance, category))
        throw new InvalidOperationException ("Instance does not exist");
        // Экземпляр не существует

```

```

float lastValue = 0f;
using (PerformanceCounter pc = new PerformanceCounter (category,
                                                       counter, instance))
{
    while (!stopper.WaitOne (200, false))
    {
        float value = pc.NextValue();
        if (value != lastValue)           // Записывать значение, только
        {                                // если оно изменилось
            Console.WriteLine (value);
            lastValue = value;
        }
    }
}

```

Ниже показано, как применять метод `Monitor` для одновременного мониторинга работы процессора и жесткого диска:

```

EventWaitHandle stopper = new ManualResetEvent (false);
new Thread (() =>
    Monitor ("Processor", "% Processor Time", "_Total", stopper)
).Start();
new Thread (() =>
    Monitor ("LogicalDisk", "% Idle Time", "C:", stopper)
).Start();
// Проведение мониторинга; для завершения нужно нажать любую клавишу
Console.WriteLine ("Monitoring - press any key to quit");
Console.ReadKey();
stopper.Set();

```

Создание счетчиков и запись данных о производительности

Перед записью данных счетчика производительности понадобится создать категорию производительности и счетчик. Категория производительности должна быть создана наряду со всеми принадлежащими ей счетчиками за один шаг:

```

string category = "Nutshell Monitoring";
// Мы создадим два счетчика в следующей категории:
string eatenPerMin = "Macadamias eaten so far";
                    // Макадамия, съеденная до сих пор
string tooHard = "Macadamias deemed too hard";
                    // Макадамия считается слишком твердой
if (!PerformanceCounterCategory.Exists (category))
{
    CounterCreationDataCollection cd = new CounterCreationDataCollection();
    // Количество потребленной макадамии, включая время очистки от скорлупы
    cd.Add (new CounterCreationData (eatenPerMin,
                                    "Number of macadamias consumed, including shelling time",
                                    PerformanceCounterType.NumberOfItems32));
    // Количество макадамии, скорлупа которой не треснет, несмотря на все усилия
    cd.Add (new CounterCreationData (tooHard,
                                    "Number of macadamias that will not crack, despite much effort",
                                    PerformanceCounterType.NumberOfItems32));
}

```

```
    PerformanceCounterCategory.Create (category, "Test Category",
        PerformanceCounterCategoryType.SingleInstance, cd);
}
```

Новые счетчики появятся в инструменте мониторинга производительности Windows в окне Add Counters (Добавить счетчики).

Если позже понадобится определить дополнительные счетчики в той же самой категории, то старая категория должна быть сначала удалена вызовом метода `PerformanceCounterCategory.Delete`.



Создание и удаление счетчиков производительности требует наличия прав администратора. По этой причине такие действия выполняются как часть процесса установки приложения.

После того, как счетчик создан, его значение можно обновить, создав экземпляр `PerformanceCounter`, установив его свойство `ReadOnly` в `false` и затем установив его свойство `RawValue`. Для обновления существующего значения можно также применять методы `Increment` и `IncrementBy`:

```
string category = "Nutshell Monitoring";
string eatenPerMin = "Macadamias eaten so far";
using (PerformanceCounter pc = new PerformanceCounter (category,
    eatenPerMin, ""))
{
    pc.ReadOnly = false;
    pc.RawValue = 1000;
    pc.Increment ();
    pc.IncrementBy (10);
    Console.WriteLine (pc.NextValue());           // 1011
}
```

Класс Stopwatch

Класс `Stopwatch` предлагает удобный механизм для измерения времени выполнения. Класс `Stopwatch` использует механизм с самым высоким разрешением, которое только обеспечивается операционной системой и оборудованием; обычно разрешение составляет меньше одной микросекунды. (По контрасту с ним свойства `DateTime.Now` и `Environment.TickCount` поддерживают разрешение около 15 мс.)

Для работы с классом `Stopwatch` необходимо вызвать метод `StartNew` — в результате создается новый экземпляр `Stopwatch` и запускается измерение времени. (В качестве альтернативы экземпляр `Stopwatch` можно создать вручную и затем вызвать метод `Start`.) Свойство `Elapsed` возвращает интервал прошедшего времени в виде структуры `TimeSpan`:

```
Stopwatch s = Stopwatch.StartNew();
System.IO.File.WriteAllText ("test.txt", new string ('*', 30000000));
Console.WriteLine (s.Elapsed);                  // 00:00:01.4322661
```

Класс `Stopwatch` также открывает доступ к свойству `ElapsedTicks`, которое возвращает количество пройденных “тиков” как значение `long`. Чтобы преобразовать тики в секунды, нужно разделить полученное значение на `StopWatch.Frequency`. Есть также свойство `ElapsedMilliseconds`, которое часто оказывается наиболее удобным.

Вызов метода `Stop` фиксирует значения свойств `Elapsed` и `ElapsedTicks`. Никакого фонового действия, связанного с “выполнением” `Stopwatch`, не предусмотрено, а потому вызов метода `Stop` является необязательным.

Межплатформенные инструменты диагностики

В этом разделе будут кратко описаны межплатформенные инструменты диагностики, доступные в .NET.

- `dotnet-counters`. Предоставляет общее представление состояния работающего приложения.
- `dotnet-trace`. Предназначен для более детального мониторинга производительности и событий.
- `dotnet-dump`. Предназначен для создания дампа памяти по запросу или после аварийного отказа.

Указанные инструменты не требуют повышения прав до уровня администратора и подходят как для среды разработки, так и для производственной среды.

`dotnet-counters`

Инструмент `dotnet-counters` осуществляет мониторинг использования памяти и центрального процессора процессом .NET и выводит данные на консоль (или в файл).

Чтобы установить этот инструмент, введите следующую команду в окне командной строки или терминала при наличии каталога с `dotnet` в пути поиска:

```
dotnet tool install --global dotnet-counters
```

Затем вы сможете запустить мониторинг процесса:

```
dotnet-counters monitor System.Runtime --process-id <<ИдентификаторПроцесса>>
```

`System.Runtime` означает то, что нужно выполнять мониторинг всех счетчиков из категории `System.Runtime`. Можно указывать либо название категории, либо имя счетчика (команда `dotnet-counters list` выводит список всех доступных категорий и счетчиков).

Вывод постоянно обновляется и выглядит примерно так:

```
Press p to pause, r to resume, q to quit.
```

```
Status: Running
```

```
[System.Runtime]
```

# of Assemblies Loaded	63
------------------------	----

% Time in GC (since last GC)	0
------------------------------	---

Allocation Rate (Bytes / sec)	244,864
-------------------------------	---------

CPU Usage (%)	6
Exceptions / sec	0
GC Heap Size (MB)	8
Gen 0 GC / sec	0
Gen 0 Size (B)	265,176
Gen 1 GC / sec	0
Gen 1 Size (B)	451,552
Gen 2 GC / sec	0
Gen 2 Size (B)	24
LOH Size (B)	3,200,296
Monitor Lock Contention Count / sec	0
Number of Active Timers	0
ThreadPool Completed Work Items / sec	15
ThreadPool Queue Length	0
ThreadPool Threads Count	9
Working Set (MB)	52

Ниже описаны все доступные команды.

Команда	Предназначение
list	Отображает список имен счетчиков вместе с описанием каждого из них
ps	Отображает список процессов dotnet, пригодных для мониторинга
monitor	Отображает значения выбранных счетчиков (обновляемые через определенные промежутки времени)
collect	Сохраняет информацию о счетчике в файл

Поддерживаются следующие параметры.

Параметры/аргументы	Предназначение
--version	Отображает версию инструмента dotnet-counters
-h, --help	Отображает справочную информацию о программе
-p, --process-id	Устанавливает идентификатор процесса dotnet, подлежащего мониторингу. Применяется к командам monitor и collect
--refresh-interval	Устанавливает желаемый интервал обновления в секундах. Применяется к командам monitor и collect
-o, --output	Устанавливает имя выходного файла. Применяется к команде collect
--format	Устанавливает выходной формат. Допускается csv или json. Применяется к команде collect

dotnet-trace

Трассировки — это записи с отметками времени возникновения событий в вашей программе, таких как вызов метода или запрос к базе данных.

Трассировки могут также включать метрики производительности и специальные события, равно как содержать локальный контекст, подобный значениям локальных переменных. Традиционно .NET Framework и инфраструктуры вроде ASP.NET использовали ETW. В .NET 5 трассировки приложения записываются с помощью ETW в случае функционирования в среде Windows и LTTracing при работе в среде Linux.

Для установки инструмента введите следующую команду:

```
dotnet tool install --global dotnet-trace
```

Чтобы начать запись событий программы, введите такую команду:

```
dotnet-trace collect --process-id <<ИдентификаторПроцесса>>
```

Команда запускает инструмент `dotnet-trace` со стандартным профилем, который собирает события центрального процессора и исполняющей среды .NET и записывает их в файл по имени `trace.nettrace`. Указывать другие профили можно посредством переключателя `--profile: gc-verbose` отслеживает сборку мусора и выборочно выделение памяти под объекты, а `gc-collect` — сборку мусора с низкими накладными расходами. Переключатель `-o` позволяет задавать другое имя выходного файла.

По умолчанию вывод производится в файл `.netperf`, который можно анализировать прямо на машине Windows с помощью инструмента `PerfView`. В качестве альтернативы инструмент `dotnet-trace` можно проинструктировать о необходимости создания файла, совместимого с бесплатной онлайновой службой анализа `Speedscope` (<https://speedscope.app>). Для создания файла `Speedscope` (`.speedscope.json`) используйте параметр `--format speedscope`.



Самая последняя версия инструмента `PerfView` доступна для загрузки по ссылке <https://github.com/microsoft/perfview>. Версия, поставляемая в составе Windows 10, может не поддерживать файлы `.netperf`.

Поддерживаются перечисленные ниже команды.

Команда	Предназначение
<code>collect</code>	Начинает запись информации о счетчике в файл
<code>ps</code>	Отображает список процессов <code>dotnet</code> , пригодных для мониторинга
<code>list-profiles</code>	Выводит список заранее построенных профилей трассировки с описанием поставщиков и фильтров в каждом
<code>convert <file></code>	Выполняет преобразование из формата <code>.nettrace</code> (<code>.netperf</code>) в альтернативный формат. В настоящее время единственным вариантом является <code>speedscope</code>

Специальные события трассировки

Ваше приложение может выпускать специальные события, определив специальный класс `EventSource`:

```

[EventSource (Name = "MyTestSource")]
public sealed class MyEventSource : EventSource
{
    public static MyEventSource Instance = new MyEventSource ();
    MyEventSource() : base (EventSourceSettings.EtwSelfDescribingEventFormat)
    {
    }

    public void Log (string message, int someNumber)
    {
        WriteEvent (1, message, someNumber);
    }
}

```

Метод `WriteEvent` перегружен для приема различных комбинаций простых типов (в основном строк и целых чисел). Затем вы можете вызывать его следующим образом:

```
MyEventSource.Instance.Log ("Something", 123);
```

При запуске `dotnet-trace` вы должны указать имя либо имена любых источников специальных событий, которые хотите записывать:

```
dotnet-trace collect --process-id <<ИдентификаторПроцесса>>
--providers MyTestSource
```

dotnet-dump

Дамп, иногда называемый дампом ядра, представляет собой моментальный снимок состояния виртуальной памяти процесса. Можно по запросу создавать дамп работающего процесса или сконфигурировать операционную систему для генерации дампа в случае аварийного отказа приложения.

Показанная ниже команда включает создание дампа ядра при аварийном отказе приложения в среде Ubuntu Linux (необходимые шаги могут отличаться между разновидностями Linux):

```
ulimit -c unlimited
```

В среде Windows с помощью редактора реестра (`regedit.exe`) понадобится создать или отредактировать следующий раздел в реестре локальной машины:

```
SOFTWARE\Microsoft\Windows\Windows Error Reporting\LocalDumps
```

Внутрь этого раздела нужно добавить ключ с именем, совпадающим с именем исполняемого файла (скажем, `foo.exe`), и поместить в него перечисленные далее ключи:

- `DumpFolder` (типа `REG_EXPAND_SZ`) со значением, указывающим путь, куда желательно сохранять файлы дампов;
- `DumpType` (типа `REG_DWORD`) со значением 2 для запрашивания полного дампа;
- (необязательно) `DumpCount` (типа `REG_DWORD`) со значением, указывающим максимальное количество файлов дампов, по достижении которого более старые файлы начнут удаляться.

Чтобы установить инструмент dotnet-dump, введите такую команду:

```
dotnet tool install --global dotnet-dump
```

После его установки вот как можно инициировать создание дампа по запросу (не заканчивая процесс):

```
dotnet-dump collect --process-id <<ИдентификаторПроцесса>>
```

Следующая команда запускает интерактивную оболочку для анализа файла дампа:

```
dotnet-dump analyze <<ФайлДампа>>
```

Если исключение привело к прекращению работы приложения, тогда с применением команды printexceptions (сокращенно pe) можно отобразить детали сгенерированного исключения. Оболочка dotnet-dump поддерживает множество дополнительных команд, список которых можно вывести с использованием команды help.



Параллелизм и асинхронность

Большинству приложений приходится иметь дело сразу с несколькими действиями, происходящими одновременно (*параллелизм*). Настоящую главу мы начнем с рассмотрения важнейших предпосылок, а именно — основ многопоточности и задач, после чего подробно обсудим принципы асинхронности и асинхронные функции C#.

В главе 21 мы продолжим более детальный анализ многопоточности, а в главе 22 раскроем связанную тему параллельного программирования.

Введение

Ниже приведены самые распространенные сценарии применения параллелизма.

- **Написание отзывчивых пользовательских интерфейсов.** Для обеспечения приемлемого времени отклика в приложениях WPF, мобильных приложениях и приложениях Windows Forms длительно выполняющиеся задачи должны запускаться параллельно с кодом, реализующим пользовательский интерфейс.
- **Обеспечение одновременной обработки запросов.** Клиентские запросы могут поступать на сервер одновременно, а потому они должны обрабатываться параллельно для обеспечения масштабируемости. В случае использования инфраструктуры ASP.NET Core или Web API исполняющая среда делает это автоматически. Тем не менее, вы по-прежнему должны заботиться о совместно используемом состоянии (например, учитывать последствия применения статических переменных для кеширования).
- **Параллельное программирование.** Код, в котором присутствуют интенсивные вычисления, может выполняться быстрее на многоядерных/много-процессорных компьютерах, если рабочая нагрузка распределяется между ядрами (данной теме посвящена глава 22).

- **Упреждающее выполнение.** На многоядерных машинах иногда удается улучшить производительность, предсказывая то, что возможно понадобится сделать, и выполняя это действие заранее. В LINQPad такой прием используется для ускорения создания новых запросов. Вариацией может быть запуск нескольких алгоритмов параллельно для решения одной и той же задачи. Тот из них, который завершится первым, “выиграет” — прием эффективен, когда нельзя узнать заранее, какой алгоритм будет выполняться быстрее всех.

Общий механизм, с помощью которого программа может выполнять код одновременно, называется *многопоточностью*. Многопоточность поддерживается как средой CLR, так и операционной системой (ОС), и в рамках параллелизма является фундаментальной концепцией. Таким образом, крайне важно четко понимать основы многопоточной обработки и в особенности влияние потоков на *совместно используемое состояние*.

Многопоточная обработка

Поток — это путь выполнения, который может проходить независимо от других таких путей.

Каждый поток запускается внутри процесса ОС, который предоставляет изолированную среду для выполнения программы. В однопоточной программе внутри изолированной среды процесса функционирует только один поток, поэтому он получает монопольный доступ к среде. В многопоточной программе внутри единственного процесса запускается множество потоков, совместно используя одну и ту же среду выполнения (скажем, память). Отчасти это одна из причин, почему полезна многопоточность: например, один поток может извлекать данные в фоновом режиме, в то время как другой поток — отображать их по мере поступления. Такие данные называются *совместно используемым состоянием*.

Создание потока

Клиентская программа (консольная, WPF, UWP или Windows Forms) запускается в единственном потоке, который создается автоматически операционной системой (“главный” поток). Здесь он и будет существовать как однопоточное приложение, если только вы не создадите дополнительные потоки (прямо или косвенно)¹.

Создать и запустить новый поток можно за счет создания объекта Thread и вызова его метода Start. Простейший конструктор Thread принимает делегат ThreadStart: метод без параметров, который указывает, где должно начинаться выполнение. Вот пример:

¹ “За кулисами” среда CLR создает другие потоки, предназначенные для сборки мусора и финализации.

```

// Напоминание. Во всех примерах главы предполагается
// импортирование следующих пространств имен:
using System;
using System.Threading;
Thread t = new Thread (WriteY);      // Начать новый поток,
t.Start();                          // выполняющий WriteY()

// Одновременно делать что-то в главном потоке
for (int i = 0; i < 1000; i++) Console.Write ("x");
void WriteY()
{
    for (int i = 0; i < 1000; i++) Console.Write ("y");
}

```

Ниже показан типичный вывод:

```

xxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
...
```

Главный поток создает новый поток *t*, в котором запускает метод, многократно выводящий символ “у”. В то же самое время главный поток многократно выводит символ “х” (рис. 14.1). На компьютере с одноядерным процессором ОС должна выделять каждому потоку кванты времени (обычно размером 20 миллисекунд в среде Windows) для эмуляции параллелизма, что дает в результате повторяющиеся блоки вывода “х” и “у”. На многоядерной или многопроцессорной машине два потока могут выполняться по-настоящему параллельно (конкурируя с другими активными процессами в системе), хотя в рассматриваемом примере все равно будут получаться повторяющиеся блоки вывода “х” и “у” из-за тонкостей работы механизма, которым класс *Console* обрабатывает параллельные запросы.

Главный поток

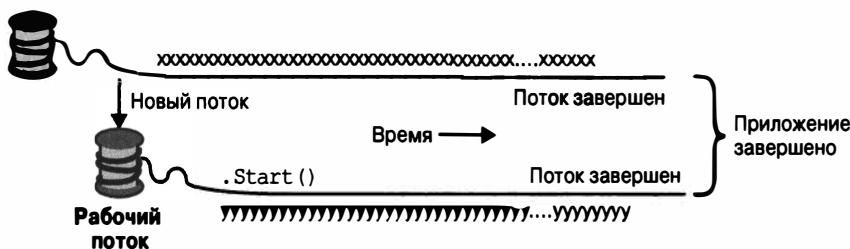


Рис. 14.1. Начало нового потока



Говорят, что поток **вытесняется** в точках, где его выполнение пересекается с выполнением кода в другом потоке. К этому термину часто прибегают при объяснении, почему что-то пошло не так, как было задумано!

После запуска свойство IsAlive потока возвращает true до тех пор, пока не будет достигнута точка, где поток завершается. Поток заканчивается, когда завершает выполнение делегат, переданный конструктору класса Thread. После завершения поток не может быть запущен повторно.

Каждый поток имеет свойство Name, которое можно установить для сопровождения отладки. Это особенно полезно в Visual Studio, т.к. имя потока отображается в окне Threads (Потоки) и в панели инструментов Debug Location (Местоположение отладки). Установить имя потока можно только один раз; попытки изменить его позже приведут к генерации исключения.

Статическое свойство Thread.CurrentThread возвращает поток, выполняющийся в текущее время:

```
Console.WriteLine (Thread.CurrentThread.Name);
```

Join и Sleep

С помощью метода Join можно организовать ожидание окончания другого потока:

```
Thread t = new Thread (Go);
t.Start();
t.Join();
Console.WriteLine ("Thread t has ended!");           // Поток t завершен!
void Go() { for (int i = 0; i < 1000; i++) Console.Write ("y"); }
```

Код выводит на консоль символ "у" тысячу раз и затем сразу же строку "Thread t has ended!". При вызове метода Join можно указывать тайм-аут, выраженный в миллисекундах или в виде структуры TimeSpan. Тогда метод будет возвращать true, если поток был завершен, или false, если истекло время тайм-аута.

Метод Thread.Sleep приостанавливает текущий поток на заданный период:

```
Thread.Sleep (TimeSpan.FromHours (1));           // Ожидать 1 час
Thread.Sleep (500);                            // Ожидать 500 мс
```

Вызов Thread.Sleep(0) немедленно прекращает текущий квант времени потока, добровольно передавая контроль над центральным процессором (ЦП) другим потокам. Метод Thread.Yield() делает то же самое, но уступает контроль только потокам, функционирующими на том же самом процессоре.



Вызов Sleep(0) или Yield() в производственном коде иногда полезен для расширенной настройки производительности. Это также великолепный диагностический инструмент для поиска проблем, связанных с безопасностью к потокам: если вставка вызова Thread.Yield() в любое место кода нарушает работу программы, то в ней почти наверняка присутствует ошибка.

На период ожидания Sleep или Join поток блокируется.

Блокирование

Поток считается заблокированным, если его выполнение приостановлено по некоторой причине, такой как вызов метода `Sleep` или ожидание завершения другого потока через вызов `Join`. Заблокированный поток немедленно *уступает* свой квант процессорного времени и далее не потребляет процессорное время, пока удовлетворяется условие блокировки. Проверить, заблокирован ли поток, можно с помощью его свойства `ThreadState`:

```
bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;
```



Свойство `ThreadState` является перечислением флагов, комбинирующим три “уровня” данных в побитовой манере. Однако большинство значений являются избыточными, неиспользуемыми или устаревшими. Следующий расширяющий метод ограничивает `ThreadState` одним из четырех полезных значений: `Unstarted`, `Running`, `WaitSleepJoin` и `Stopped`:

```
public static ThreadState Simplify (this ThreadState ts)
{
    return ts & (ThreadState.Unstarted |
                  ThreadState.WaitSleepJoin |
                  ThreadState.Stopped);
}
```

Свойство `ThreadState` удобно для диагностических целей, но непригодно для синхронизации, т.к. состояние потока может измениться в промежутке между проверкой `ThreadState` и обработкой данной информации.

Когда поток блокируется или деблокируется, ОС производит *переключение контекста*. С ним связаны небольшие накладные расходы, обычно составляющие одну или две микросекунды.

Интенсивный ввод-вывод или интенсивные вычисления

Операция, которая большую часть своего времени тратит на ожидание, пока что-то произойдет, называется операцией с *интенсивным вводом-выводом*; примером может служить загрузка веб-страницы или вызов метода `Console.ReadLine`. (Операции с интенсивным вводом-выводом обычно включают в себя ввод или вывод, но это не жесткое требование: вызов метода `Thread.Sleep` также считается операцией с интенсивным вводом-выводом.) И напротив, операция, которая большую часть своего времени затрачивает на выполнение вычислений с привлечением ЦП, называется операцией с *интенсивными вычислениями*.

Блокирование или зацикливание

Операция с интенсивным вводом-выводом работает одним из двух способов. Она либо *синхронно* ожидает завершения определенной операции в текущем потоке (такой как `Console.ReadLine`, `Thread.Sleep` или `Thread.Join`), либо

работает асинхронно, инициируя обратный вызов, когда интересующая операция завершается спустя какое-то время (более подробно об этом позже).

Операции с интенсивным вводом-выводом, которые ожидают синхронным образом, большую часть своего времени тратят на блокирование потока. Они также могут периодически “прокручиваться” в цикле:

```
while (DateTime.Now < nextStartTime)  
    Thread.Sleep (100);
```

Оставляя в стороне тот факт, что существуют более эффективные средства (вроде таймеров и сигнализирующих конструкций), еще одна возможность предусматривает зацикливание потока:

```
while (DateTime.Now < nextStartTime);
```

В общем случае это крайне неэкономное расходование процессорного времени: среда CLR и ОС предполагают, что поток выполняет важные вычисления, и надлежащим образом выделяют ресурсы. В сущности, мы превращаем код, который должен быть операцией с интенсивным вводом-выводом, в операцию с интенсивными вычислениями.



Относительно вопроса зацикливания или блокирования следует отметить пару нюансов. Во-первых, очень *кратковременное* зацикливание может быть эффективным, когда ожидается скорое (возможно в пределах нескольких микросекунд) удовлетворение некоторого условия, поскольку оно избегает накладных расходов и задержки, связанной с переключением контекста. Платформа .NET предлагает специальные методы и классы для содействия зацикливанию (см. информацию по ссылке [SpinLock and SpinWait](http://albahari.com/threading/) на странице <http://albahari.com/threading/>).

Во-вторых, затраты на блокирование не являются *нулевыми*. Дело в том, что за время своего существования каждый поток связывает около 1 Мбайт памяти и служит источником текущих накладных расходов на администрирование со стороны среды CLR и ОС. По этой причине блокирование может быть ненадежным в контексте программ с интенсивным вводом-выводом, которые нуждаются в поддержке сотен или тысяч параллельных операций. Взамен такие программы должны использовать подход, основанный на обратных вызовах, что полностью освободит поток на время ожидания. Таково (отчасти) целевое назначение асинхронных шаблонов, которые мы обсудим позже.

Локальное или совместно используемое состояние

Среда CLR назначает каждому потоку собственный стек в памяти, так что локальные переменные хранятся отдельно. В следующем примере мы определяем метод с локальной переменной, после чего вызываем его одновременно в главном потоке и во вновь созданном потоке:

```
new Thread (Go).Start();           // Вызвать Go в новом потоке
Go();                            // Вызвать Go в главном потоке
void Go()
{
    // Объявить и использовать локальную переменную cycles
    for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}
```

В стеке каждого потока создается отдельная копия переменной `cycles`, так что вывод вполне предсказуемо содержит десять знаков вопроса.

Потоки совместно используют данные, если они имеют общую ссылку на один и тот же экземпляр:

```
bool _done = false;
new Thread (Go).Start();
Go();
void Go()
{
    if (!_done) { _done = true; Console.WriteLine ("Done"); }
}
```

Оба потока совместно используют переменную `_done`, поэтому слово “Done” выводится один раз, а не два.

Локальные переменные, захваченные лямбда-выражением, тоже могут быть совместно используемыми:

```
bool done = false;
ThreadStart action = () =>
{
    if (!done) { done = true; Console.WriteLine ("Done"); }
};
new Thread (action).Start();
action();
```

Тем не менее, для совместного использования данных между потоками чаще применяются поля. В следующем примере в обоих потоках метод `Go` вызывается на том же самом экземпляре `ThreadTest`, так что они совместно используют поле `_done`:

```
var tt = new ThreadTest();
new Thread (tt.Go).Start();
tt.Go();
class ThreadTest
{
    bool _done;
    public void Go()
    {
        if (!_done) { _done = true; Console.WriteLine ("Done"); }
    }
}
```

Статические поля предлагают еще один способ совместного использования данных между потоками:

```
class ThreadTest
{
    static bool _done; // Статические поля совместно используются между всеми
                      // потоками в том же самом домене приложения
    static void Main()
    {
        new Thread (Go).Start();
        Go();
    }
    static void Go()
    {
        if (!_done) { _done = true; Console.WriteLine ("Done"); }
    }
}
```

Все четыре примера иллюстрируют еще одну ключевую концепцию: безопасность в отношении потоков (или наоборот — ее отсутствие). Вывод в действительности не определен: возможно (хотя и маловероятно), что слово “Done” будет выведено дважды. Однако если мы поменяем местами порядок следования операторов в методе Go, то вероятность двукратного вывода слова “Done” значительно возрастет:

```
static void Go()
{
    if (!_done) { Console.WriteLine ("Done"); _done = true; }
```

Проблема в том, что один поток может оценивать оператор if точно в то же самое время, когда второй поток выполняет оператор WriteLine — до того, как он получит шанс установить поле _done в true.



Приведенный пример демонстрирует одну из многочисленных ситуаций, в которых *совместно используемое записываемое состояние* может привести к возникновению определенной разновидности несистематических ошибок, характерных для многопоточности. В следующем разделе мы покажем, как с помощью блокировки устраниТЬ проблемы; тем не менее, по возможности лучше вообще избегать применения совместно используемого состояния. Позже мы объясним, как в этом могут помочь шаблоны асинхронного программирования.

Блокировка и безопасность потоков



Блокировка и безопасность в отношении потоков являются обширными темами. Полное их обсуждение приведено в разделах “Монопольное блокирование” и “Блокирование и безопасность к потокам” главы 21.

Исправить предыдущий пример можно, получив *монопольную блокировку* на период чтения и записи совместно используемого поля. Для этой цели в языке C# предусмотрен оператор lock:

```

class ThreadSafe
{
    static bool _done;
    static readonly object _locker = new object();
    static void Main()
    {
        new Thread (Go).Start();
        Go();
    }
    static void Go()
    {
        lock (_locker)
        {
            if (!_done) { Console.WriteLine ("Done"); _done = true; }
        }
    }
}

```

Когда два потока одновременно соперничают за блокировку (что может возникать с любым объектом ссылочного типа; `_locker` в рассматриваемом случае), один из потоков ожидает, или блокируется, до тех пор, пока блокировка не станет доступной. В таком случае гарантируется, что только один поток может войти в данный блок кода за раз, и строка `Done` будет выведена лишь однократно. Код, защищенный подобным образом (от неопределенности в многопоточном контексте), называется *безопасным в отношении потоков*.



Даже действие автоинкрементирования переменной не является безопасным к потокам: выражение `x++` выполняется на лежащем в основе процессоре как отдельные операции чтения, инкремента и записи. Таким образом, если два потока выполняют `x++` одновременно за пределами блокировки, то переменная `x` в итоге может быть инкрементирована один раз, а не два (или, что еще хуже, в определенных обстоятельствах переменная `x` может быть вообще *разрушена*, получив смесь битов старого и нового содержимого).

Блокировка не является панацеей для обеспечения безопасности потоков — довольно легко забыть заблокировать доступ к полю и тогда блокировка сама может создать проблемы (наподобие состояния взаимоблокировки).

Хорошим примером применения блокировки может служить доступ к совместно используемому кешу внутри памяти для часто эксплуатируемых объектов базы данных в приложении ASP.NET. Приложение такого вида очень просто заставить работать правильно без возникновения взаимоблокировки. Пример будет приведен в разделе “Безопасность к потокам в серверах приложений” главы 21.

Передача данных потоку

Иногда требуется передать аргументы начальному методу потока. Проще всего это сделать с использованием лямбда-выражения, которое вызывает данный метод с желаемыми аргументами:

```
Thread t = new Thread ( () => Print ("Hello from t!") );
t.Start();
void Print (string message) => Console.WriteLine (message);
```

Такой подход позволяет передавать методу любое количество аргументов. Можно даже поместить всю реализацию в лямбда-функцию с множеством операторов:

```
new Thread ( () =>
{
    Console.WriteLine ("I'm running on another thread!");
    Console.WriteLine ("This is so easy!");
}).Start();
```

Альтернативный (менее гибкий) прием предусматривает передачу аргумента методу Start класса Thread:

```
Thread t = new Thread (Print);
t.Start ("Hello from t!");
void Print (object messageObj)
{
    string message = (string) messageObj; // Здесь необходимо приведение
    Console.WriteLine (message);
}
```

Код работает из-за того, что конструктор класса Thread перегружен для приема одного из двух делегатов:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart (object obj);
```

Лямбда-выражения и захваченные переменные

Как уже должно быть понятно, лямбда-выражение является наиболее удобным и мощным способом передачи данных потоку. Однако следует соблюдать осторожность, чтобы случайно не изменить *захваченные переменные* после запуска потока. Например, рассмотрим следующий код:

```
for (int i = 0; i < 10; i++)
    new Thread ( () => Console.Write (i)).Start();
```

Вывод будет недетерминированным! Вот типичный результат:

```
0223557799
```

Проблема в том, что на протяжении всего времени жизни цикла переменная i ссылается на *ту же самую ячейку* в памяти. Следовательно, каждый поток вызывает метод Console.Write с переменной, значение которой может измениться в ходе его выполнения! Решение заключается в применении временной переменной, как показано ниже:

```
for (int i = 0; i < 10; i++)
{
    int temp = i;
    new Thread ( () => Console.Write (temp)).Start();
}
```

Теперь все цифры от 0 до 9 будут выводиться в точности по одному разу.
(Порядок вывода по-прежнему не определен, т.к. потоки могут запускаться в непредсказуемые моменты времени.)



Данная проблема аналогична проблеме, описанной в разделе “Захваченные переменные” главы 8. Она в основном обусловлена правилами языка C# для захвата переменных внутри циклов `for` в многопоточном сценарии.

Переменная `temp` является локальной по отношению к каждой итерации цикла. Таким образом, каждый поток захватывает отличающуюся ячейку памяти, и проблемы не возникают. Проблему в приведенном ранее коде проще проиллюстрировать с помощью показанного далее примера:

```
string text = "t1";
Thread t1 = new Thread ( () => Console.WriteLine (text) );
text = "t2";
Thread t2 = new Thread ( () => Console.WriteLine (text) );
t1.Start(); t2.Start();
```

Поскольку оба лямбда-выражения захватывают одну и ту же переменную `text`, строка `t2` выводится дважды.

Обработка исключений

Любые блоки `try/catch/finally`, действующие во время создания потока, не играют никакой роли в потоке, когда он начинает свое выполнение. Взгляните на следующую программу:

```
try
{
    new Thread (Go).Start();
}
catch (Exception ex)
{
    // Сюда мы никогда не попадем!
    Console.WriteLine ("Exception!"); // Исключение!
}

void Go() { throw null; } // Генерирует исключение NullReferenceException
```

Оператор `try/catch` здесь безрезультатен, и вновь созданный поток будет обременен необработанным исключением `NullReferenceException`. Такое поведение имеет смысл, если принять во внимание тот факт, что каждый поток обладает независимым путем выполнения.

Чтобы исправить ситуацию, обработчик событий потребуется переместить внутрь метода `Go`:

```
new Thread (Go).Start();

void Go()
{
```

```
try
{
    ...
    throw null; // Исключение NullReferenceException будет перехвачено ниже
    ...
}
catch (Exception ex)
{
    // Обычно необходимо зарегистрировать исключение в журнале
    // и/или сигнализировать другому потоку об отсоединении
    ...
}
```

В производственных приложениях необходимо предусмотреть обработчики исключений для всех методов входа в потоки — в частности как это делается в главном потоке (обычно на более высоком уровне в стеке выполнения). Необработанное исключение приведет к прекращению работы всего приложения, да еще и с отображением безобразного диалогового окна!



При написании таких блоков обработки исключений вы редко будете *игнорировать* ошибку: обычно вы предусмотрите регистрацию деталей исключения в журнале. Для клиентского приложения, возможно, вы отобразите диалоговое окно, позволяющее пользователю автоматически отправить подробные сведения веб-серверу. Затем, по всей видимости, вы решите перезапустить приложение, поскольку существует возможность того, что непредвиденное исключение оставило приложение в недопустимом состоянии.

Централизованная обработка исключений

В приложениях WPF, UWP и Windows Forms можно подписываться на “глобальные” события обработки исключений — `Application.DispatcherUnhandledException` и `Application.ThreadException`. Они инициируются после возникновения необработанного исключения в любой части программы, которая вызвана в цикле сообщений (сказанное относится ко всему коду, выполняющемуся в главном потоке, пока активен экземпляр `Application`). Прием полезен в качестве резервного средства для регистрации и сообщения об ошибках (хотя он неприменим для необработанных исключений, которые возникают в созданных вами рабочих потоках). Обработка упомянутых событий предотвращает аварийное завершение программы, хотя впоследствии может быть принято решение о ее перезапуске во избежание потенциального разрушения состояния, к которому может привести необработанное исключение.

Потоки переднего плана или фоновые потоки

По умолчанию потоки, создаваемые явно, являются *потоками переднего плана*. Потоки переднего плана удерживают приложение в активном состоянии до тех пор, пока хотя бы один из них выполняется, но *фоновые потоки* этого не делают. После того, как все потоки переднего плана прекратят свою работу, заканчивается и приложение, а любые все еще выполняющиеся фоновые потоки будут принудительно завершены.



Состояние переднего плана или фоновое состояние потока не имеет никакого отношения к его *приоритету* (выделению времени на выполнение).

Выяснить либо изменить фоновое состояние потока можно с использованием его свойства `IsBackground`:

```
static void Main (string[] args)
{
    Thread worker = new Thread ( () => Console.ReadLine() );
    if (args.Length > 0) worker.IsBackground = true;
    worker.Start();
}
```

Если запустить такую программу без аргументов, тогда рабочий поток предполагает, что она находится в фоновом состоянии, и будет ожидать в операторе `ReadLine` нажатия пользователем клавиши `<Enter>`. Тем временем главный поток завершится, но приложение останется запущенным, потому что поток переднего плана все еще активен. С другой стороны, если методу `Main` передается аргумент, то рабочему потоку назначается фоновое состояние, и программа завершается почти сразу после завершения главного потока (прекращая выполнение метода `ReadLine`).

Когда процесс прекращает работу подобным образом, любые блоки `finally` в стеке выполнения фоновых потоков пропускаются. Если в программе задействованы блоки `finally` (или `using`) для проведения очистки вроде удаления временных файлов, то вы можете избежать этого, явно ожидая окончание таких фоновых потоков вплоть до завершения приложения, либо за счет присоединения к потоку, либо с помощью сигнализирующей конструкции (см. раздел “Передача сигналов” далее в главе). В любом случае должен быть указан тайм-аут, чтобы можно было уничтожить поток, который отказывается завершаться, иначе приложение не сможет быть нормально закрыто без привлечения пользователем диспетчера задач (или команды `kill` в Unix).

Потоки переднего плана не требуют такой обработки, но вы должны позаботиться о том, чтобы избежать ошибок, которые могут привести к отказу завершения потока. Обычной причиной отказа в корректном завершении приложений является наличие активных фоновых потоков.

Приоритет потока

Свойство `Priority` потока определяет, сколько времени на выполнение получит данный поток относительно других активных потоков в ОС, со следующей шкалой значений:

```
enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }
```

Это становится важным, когда одновременно активно несколько потоков. Увеличение приоритета потока должно производиться осторожно, т.к. может привести к торможению других потоков. Если нужно, чтобы поток имел больший приоритет, чем потоки в других процессах, тогда потребуется также увели-

чить приоритет процесса с применением класса `Process` из пространства имен `System.Diagnostics`:

```
using Process p = Process.GetCurrentProcess();
p.PriorityClass = ProcessPriorityClass.High;
```

Прием может нормально работать для потоков, не относящихся к пользовательскому интерфейсу, которые выполняют минимальную работу и нуждаются в низкой задержке (т.е. возможности реагировать очень быстро). В приложениях с интенсивными вычислениями (особенно в тех, которые имеют пользовательский интерфейс) увеличение приоритета процесса может приводить к торможению других процессов и замедлению работы всего компьютера.

Передача сигналов

Иногда нужно, чтобы поток ожидал получения уведомления (либо уведомлений) от другого потока (потоков). Это называется *передачей сигналов*. Простейшей сигнализирующей конструкцией является класс `ManualResetEvent`. Вызов метода `WaitOne` класса `ManualResetEvent` блокирует текущий поток до тех пор, пока другой поток не “откроет” сигнал, вызвав метод `Set`. В приведенном ниже примере мы запускаем поток, который ожидает события `ManualResetEvent`. Он остается заблокированным в течение двух секунд до тех пор, пока главный поток не выдаст *сигнал*:

```
var signal = new ManualResetEvent (false);
new Thread (() =>
{
    Console.WriteLine ("Waiting for signal..."); // Ожидание сигнала...
    signal.WaitOne(); // Сигнал получен!
    signal.Dispose();
    Console.WriteLine ("Got signal!"); // "Открыть" сигнал
}).Start();
Thread.Sleep(2000);
signal.Set(); // "Открыть" сигнал
```

После вызова метода `Set` сигнал остается “открытым”; для его “закрытия” понадобится вызвать метод `Reset`.

Класс `ManualResetEvent` — одна из нескольких сигнализирующих конструкций, предоставляемых средой CLR; все они подробно рассматриваются в главе 21.

Многопоточность в обогащенных клиентских приложениях

В приложениях WPF, UWP и Windows Forms выполнение длительных по времени операций в главном потоке снижает отзывчивость приложения, потому что главный поток обрабатывает также цикл сообщений, который отвечает за визуализацию и поддержку событий клавиатуры и мыши.

Популярный подход предусматривает настройку “рабочих” потоков для выполнения длительных по времени операций. Код в рабочем потоке запускает длительную операцию и по ее завершении обновляет пользовательский интер-

фейс. Тем не менее, все обогащенные клиентские приложения поддерживают потоковую модель, в которой элементы управления пользовательского интерфейса могут быть доступны только из создавшего их потока (обычно главного потока пользовательского интерфейса). Нарушение данного правила приводит либо к непредсказуемому поведению, либо к генерации исключения.

Следовательно, когда нужно обновить пользовательский интерфейс из рабочего потока, запрос должен быть перенаправлен потоку пользовательского интерфейса (формально это называется *маршиализацией*). Вот как выглядит низкоуровневый способ реализации такого действия (позже мы обсудим другие решения, которые на нем основаны):

- в приложении WPF вызовите метод `BeginInvoke` или `Invoke` на объекте `Dispatcher` элемента;
- в приложении UWP вызовите метод `RunAsync` или `Invoke` на объекте `Dispatcher`;
- в приложении Windows Forms вызовите метод `BeginInvoke` или `Invoke` на элементе управления.

Все упомянутые методы принимают делегат, ссылающийся на метод, который требуется запустить. Методы `BeginInvoke`/`RunAsync` работают путем постановки этого делегата в очередь сообщений потока пользовательского интерфейса (та же очередь, которая обрабатывает события, поступающие от клавиатуры, мыши и таймера). Метод `Invoke` делает то же самое, но затем блокируется до тех пор, пока сообщение не будет прочитано и обработано потоком пользовательского интерфейса. По указанной причине метод `Invoke` позволяет получить возвращаемое значение из метода. Если возвращаемое значение не требуется, то методы `BeginInvoke`/`RunAsync` предпочтительнее из-за того, что они не блокируют вызывающий компонент и не привносят возможность возникновения взаимоблокировки (см. раздел “Взаимоблокировки” в главе 21).



Вы можете представлять себе, что при вызове метода `Application.Run` выполняется следующий псевдокод:

```
while (приложение не завершено)
{
    Ожидать появления чего-нибудь в очереди сообщений
    Что-то получено: к какому виду сообщений оно относится?
    Сообщение клавиатуры/мыши -> запустить обработчик событий
    Пользовательское сообщение BeginInvoke -> выполнить делегат
    Пользовательское сообщение Invoke -> выполнить делегат
    и отправить результат
}
```

Цикл такого вида позволяет рабочему потоку подготовить делегат для выполнения в потоке пользовательского интерфейса.

В целях демонстрации предположим, что имеется окно WPF с текстовым полем по имени `txtMessage`, содержимое которого должно быть обновлено рабочим потоком после выполнения длительной задачи (эмулируемой с помощью вызова метода `Thread.Sleep`).

Ниже приведен необходимый код:

```
partial class MyWindow : Window
{
    public MyWindow()
    {
        InitializeComponent();
        new Thread(Work).Start();
    }
    void Work()
    {
        Thread.Sleep(5000); // Эмулировать длительно выполняющуюся задачу
        UpdateMessage("The answer");
    }
    void UpdateMessage(string message)
    {
        Action action = () => txtMessage.Text = message;
        Dispatcher.BeginInvoke(action);
    }
}
```

После запуска показанного кода немедленно появляется окно. Спустя пять секунд текстовое поле обновляется. Для случая Windows Forms код будет похож, но только в нем вызывается метод BeginInvoke объекта Form:

```
void UpdateMessage(string message)
{
    Action action = () => txtMessage.Text = message;
    this.BeginInvoke(action);
}
```

Множество потоков пользовательского интерфейса

Допускается иметь множество потоков пользовательского интерфейса, если каждый из них владеет своим окном. Основным сценарием может служить приложение с несколькими высокоуровневыми окнами, которое часто называют приложением с *однодокументным интерфейсом* (Single Document Interface — SDI), например, Microsoft Word. Каждое окно SDI обычно отображает себя как отдельное “приложение” в панели задач и по большей части оно функционально изолировано от других окон SDI. За счет предоставления каждому такому окну собственного потока пользовательского интерфейса окна становятся более отзывчивыми.

Контексты синхронизации

В пространстве имен System.ComponentModel определен абстрактный класс SynchronizationContext, который делает возможным обобщение маршализации потоков.

В обогащенных API-интерфейсах для мобильных и настольных приложений (UWP, WPF и Windows Forms) определены и созданы экземпляры подклассов SynchronizationContext, которые можно получить через статическое свой-

ство `SynchronizationContext.Current` (при выполнении в потоке пользовательского интерфейса). Захват этого свойства позволяет позже “отправлять” сообщения элементам управления пользовательского интерфейса из рабочего потока:

```
partial class MyWindow : Window
{
    SynchronizationContext _uiSyncContext;
    public MyWindow()
    {
        InitializeComponent();
        // Захватить контекст синхронизации для текущего потока
        // пользовательского интерфейса:
        _uiSyncContext = SynchronizationContext.Current;
        new Thread(Work).Start();
    }
    void Work()
    {
        Thread.Sleep(5000); // Эмулировать длительно выполняющуюся задачу
        UpdateMessage("The answer");
    }
    void UpdateMessage(string message)
    {
        // Маршализовать делегат потоку пользовательского интерфейса:
        _uiSyncContext.Post(_ => txtMessage.Text = message, null);
    }
}
```

Удобство заключается в том, что один и тот же подход работает со всеми обогащенными API-интерфейсами.

Вызов метода `Post` эквивалентен вызову `BeginInvoke` на объекте `Dispatcher` или `Control`; есть также метод `Send`, который является эквивалентом `Invoke`.

Пул потоков

Всякий раз, когда запускается поток, несколько сотен микросекунд тратится на организацию таких элементов, как новый стек локальных переменных. Снизить эти накладные расходы позволяет *пул потоков*, предлагая накопитель заранее созданных многократно применяемых потоков. Организация пула потоков жизненно важна для эффективного параллельного программирования и реализации мелкомодульного параллелизма; пул потоков позволяет запускать короткие операции без накладных расходов, связанных с начальной настройкой потока.

При использовании потоков из пула следует учитывать несколько моментов.

- Невозможность установки свойства `Name` потока из пула затрудняет отладку (хотя при отладке в окне `Threads` среды Visual Studio к потоку можно присоединять описание).
- Потоки из пула всегда являются *фоновыми*.
- Блокирование потоков из пула может привести к снижению производительности (см. раздел “Чистота пула потоков” далее в главе).

Приоритет потока из пула можно свободно изменять — когда поток возвращается обратно в пул, будет восстановлен его первоначальный приоритет.

Для выяснения, является ли текущий поток из пула, предназначено свойство `Thread.CurrentThread.IsThreadPoolThread`.

Вход в пул потоков

Простейший способ явного запуска какого-то кода в потоке из пула предполагает применение метода `Task.Run` (мы рассмотрим этот прием более подробно в следующем разделе):

```
// Класс Task находится в пространство имен System.Threading.Tasks  
Task.Run (() => Console.WriteLine ("Hello from the thread pool"));
```

Поскольку до выхода версии .NET Framework 4.0 задачи не существовали, обшепринятой альтернативой был вызов метода `ThreadPool.QueueUserWorkItem`:

```
ThreadPool.QueueUserWorkItem (notUsed => Console.WriteLine ("Hello"));
```



Перечисленные ниже компоненты неявно используют пул потоков:

- серверы приложений ASP.NET Core и Web API;
- классы `System.Timers.Timer` и `System.Threading.Timer`;
- конструкции параллельного программирования, которые будут описаны в главе 22;
- (унаследованный) класс `BackgroundWorker`.

Чистота пула потоков

Пул потоков содействует еще одной функции, которая гарантирует то, что временный излишек интенсивной вычислительной работы не приведет к *превышению лимита ЦП*. Превышение лимита — это условие, при котором активных потоков имеется больше, чем ядер ЦП, и операционная система вынуждена выделять потокам кванты времени. Превышение лимита наносит ущерб производительности, т.к. выделение квантов времени требует интенсивных переключений контекста и может приводить к недействительности кешей ЦП, которые стали очень важными в обеспечении производительности современных процессоров.

Среда CLR избегает превышения лимита в пуле потоков за счет постановки задач в очередь и настройки их запуска. Она начинает с запуска такого количества параллельных задач, которое соответствует числу аппаратных ядер, и затем регулирует уровень параллелизма по алгоритму поиска экстремума, непрерывно подгоняя рабочую нагрузку в определенном направлении. Если производительность улучшается, тогда среда CLR продолжает двигаться в том же направлении (а иначе — в противоположном). В результате обеспечивается продвижение по оптимальной кривой производительности даже при наличии соперничающих процессов на компьютере.

Стратегия, реализованная в CLR, хорошо функционирует в случае удовлетворения следующих двух условий:

- элементы работы являются в основном кратковременными (менее 250 мс либо в идеале менее 100 мс), так что CLR имеет много возможностей для измерения и корректировки;
- в пуле не доминируют задания, которые большую часть своего времени являются заблокированными.

Блокирование ненадежно, поскольку дает среде CLR ложное представление о том, что оно загружает ЦП. Среда CLR достаточно интеллектуальна, чтобы обнаружить это и скомпенсировать (за счет внедрения дополнительных потоков в пул), хотя такое действие может сделать пул уязвимым к последующему превышению лимита. Также может быть введена задержка, потому что среда CLR регулирует скорость внедрения новых потоков, особенно на раннем этапе времени жизни приложения (тем более в клиентских ОС, где она отдает предпочтение низкому потреблению ресурсов).

Поддержание чистоты пула потоков особенно важно, когда требуется в полной мере задействовать ЦП (например, через API-интерфейсы параллельного программирования, рассматриваемые в главе 22).

Задачи

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

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

Указанные ограничения препятствуют реализации мелкомодульного параллелизма; другими словами они затрудняют формирование более крупных параллельных операций за счет комбинирования мелких операций (как будет показано в последующих разделах, это очень важно при асинхронном программировании). В свою очередь возникает более высокая зависимость от ручной синхронизации (блокировки, выдачи сигналов и т.д.) и проблем, которые ее сопровождают.

Прямое применение потоков также оказывает влияние на производительность, как обсуждалось ранее в разделе “Пул потоков”. И если требуется запустить сотни или тысячи параллельных операций с интенсивным вводом-выводом, то подход на основе потоков повлечет за собой затраты сотен или тысяч мегабайтов памяти исключительно в качестве накладных расходов, связанных с потоками.

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

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



В настоящем разделе мы не затрагиваем функциональные возможности задач, предназначенные для параллельного программирования — они подробно рассматриваются в главе 22.

Запуск задачи

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

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



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

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

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

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

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

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

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

Wait

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

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

Метод `Wait` позволяет дополнительно указывать тайм-аут и признак отмены для раннего завершения ожидания (см. раздел “Отмена” далее в главе).

Длительно выполняющиеся задачи

По умолчанию среда CLR запускает задачи в потоках из пула, что идеально в случае кратковременных задач с интенсивными вычислениями. Для длительно выполняющихся и блокирующих операций (как в предыдущем примере) использованию потоков из пула можно воспрепятствовать, как показано ниже:

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



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

- если задачи являются интенсивными в плане ввода-вывода, то вместо потоков следует применять класс `TaskCompletionSource` и асинхронные функции, которые позволяют реализовать параллельное выполнение с обратными вызовами (продолжениями);
- если задачи являются интенсивными в плане вычислений, то отрегулировать параллелизм для таких задач позволит очередь производителей/потребителей, избегая ограничения других потоков и процессов (см. раздел “Реализация очереди производителей/потребителей” в главе 22).

Возвращение значений

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

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

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

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

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

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

Код выводит строку “`Task running...`” (Задача выполняется...) и спустя несколько секунд выдает ответ 216816.



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

Исключения

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

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

(Среда CLR помещает исключение в оболочку AggregateException для нормальной работы в сценариях параллельного программирования; мы обсудим данный аспект в главе 22.)

Проверить, отказалася ли задача, можно без повторной генерации исключения посредством свойств IsFaulted и IsCanceled класса Task. Если оба свойства возвращают false, то ошибки не возникали; если IsCanceled равно true, то для задачи было сгенерировано исключение OperationCanceledException (см. раздел “Отмена” в главе 22); если IsFaulted равно true, то было сгенерировано исключение другого типа и на ошибку укажет свойство Exception.

Исключения и автономные задачи

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



Игнорировать исключения normally в ситуации, когда исключение только указывает на неудачу при получении результата, который больше не интересует. Например, если пользователь отменяет запрос на загрузку веб-страницы, то мы не должны переживать, если выяснится, что веб-страница не существует.

Игнорировать исключения проблематично, когда исключение указывает на ошибку в программе, по двум причинам:

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

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

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

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

Продолжение

Продолжение сообщает задаче о том, что после завершения она должна продолжиться и делать что-то другое. Продолжение обычно реализуется посредством обратного вызова, который выполняется один раз после завершения

операции. Существуют два способа присоединения признака продолжения к задаче. Первый из них особенно важен, поскольку применяется асинхронными функциями C#, что вскоре будет показано. Мы можем продемонстрировать его на примере с подсчетом простых чисел, который был реализован в разделе “Возвращение значений” ранее в главе:

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

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



Объект ожидания (`awaiter`) — это любой объект, открывающий доступ к двум методам, которые мы только что видели (`OnCompleted` и `GetResult`), и к булевскому свойству по имени `IsCompleted`. Никакого интерфейса или базового класса для унификации указанных членов не предусмотрено (хотя метод `OnCompleted` является частью интерфейса `INotifyCompletion`). Мы объясним важность данного шаблона в разделе “Асинхронные функции в C#” далее в главе.

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

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

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

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

Когда контекст синхронизации отсутствует (или применяется `ConfigureAwait(false)`), продолжение будет (в общем случае) выполняться в потоке из пула.

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

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

Сам метод `ContinueWith` возвращает объект `Task`, который полезен, если планируется присоединение дальнейших признаков продолжения. Однако если задача отказывает, тогда в приложениях с пользовательским интерфейсом придется иметь дело напрямую с исключением `AggregateException` и предусмотреть дополнительный код для маршализации продолжения (см. раздел “Планировщики задач” в главе 22). В контекстах, не связанных с пользовательским интерфейсом, потребуется указывать `TaskContinuationOptions.ExecuteSynchronously`, если продолжение должно выполняться в том же потоке, иначе произойдет возврат в пул потоков. Метод `ContinueWith` особенно удобен в сценариях параллельного программирования; мы рассмотрим это подробно в разделе “Продолжение” главы 22.

TaskCompletionSource

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

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

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

```
public class TaskCompletionSource<TResult>
{
    public void SetResult (TResult result);
    public void SetException (Exception exception);
    public void SetCanceled();
    public bool TrySetResult (TResult result);
    public bool TrySetException (Exception exception);
```

```

public bool TrySetCanceled();
public bool TrysetCanceled (CancellationToken cancellationToken);
...
}

```

Вызов одного из перечисленных методов *передает сигнал* задаче, помещая ее в состояние завершения, отказа или отмены (последнее состояние мы рассмотрим в разделе “Отмена” далее в главе). Предполагается, что вы будете вызывать любой из этих методов в точности один раз: в случае повторного вызова методы SetResult, SetException и SetCanceled генерируют исключение, а методы Try* возвратят false.

В следующем примере после пятисекундного ожидания выводится число 42:

```

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

```

Можно реализовать собственный метод Run с использованием класса TaskCompletionSource:

```

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

```

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

Реальная мощь класса TaskCompletionSource заключается в возможности создания задач, не связывающих потоки. Например, рассмотрим задачу, которая ожидает пять секунд и затем возвращает число 42. Мы можем реализовать ее без потока с применением класса Timer, который с помощью CLR (и в свою очередь ОС) инициирует событие каждые x миллисекунд (таймеры еще будут рассматриваться в главе 21):

```

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

```

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

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

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

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

Теперь можно реализовать универсальный метод Delay:

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



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

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

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

Такое применение класса TaskCompletionSource без потока означает, что поток будет занят, только когда запускается продолжение, т.е. спустя пять секунд. Мы можем продемонстрировать это, запустив 10 000 таких операций одновременно и не столкнувшись с ошибкой или чрезмерным потреблением ресурсов:

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



Таймеры инициируют свои обратные вызовы на потоках из пула, так что через пять секунд пул потоков получит 10 000 запросов вызова `SetResult(null)` на `TaskCompletionSource`. Если запросы поступают быстрее, чем они могут быть обработаны, тогда пул потоков отреагирует постановкой их в очередь и последующей обработкой на оптимальном уровне параллелизма для ЦП. Это идеально в ситуации, когда привязанные к потокам задания являются кратковременными, что в данном случае справедливо: привязанное к потоку задание просто вызывает метод `SetResult` и либо осуществляет отправку признака продолжения контексту синхронизации (в приложении с пользовательским интерфейсом), либо выполняет само продолжение (`Console.WriteLine(42)`).

Task.Delay

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

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

или:

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

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

Принципы асинхронности

Мы завершили демонстрацию `TaskCompletionSource` написанием *асинхронных* методов. В данном разделе мы объясним, что собой представляют асинхронные операции, и покажем, как они приводят к асинхронному программированию.

Сравнение синхронных и асинхронных операций

Синхронная операция выполняет свою работу *перед* возвратом управления вызывающему коду.

Асинхронная операция может выполнять большую часть или всю свою работу *после* возврата управления вызывающему коду.

Большинство создаваемых и вызываемых вами методов будут синхронными. Примерами могут служить `List<T>.Add`, `Console.WriteLine` и `Thread.Sleep`. Асинхронные методы менее распространены и инициируют *параллелизм*, т.к. они продолжают работать параллельно с вызывающим кодом. Асинхронные методы обычно быстро (или немедленно) возвращают управление вызывающему компоненту, потому их также называют *неблокирующими методами*.

Большинство асинхронных методов, которые мы видели до сих пор, могут быть описаны как универсальные методы:

- `Thread.Start`;
- `Task.Run`;
- методы, которые присоединяют признаки продолжения к задачам.

В добавок некоторые методы из числа рассмотренных в разделе “Контексты синхронизации” ранее в главе (`Dispatcher.BeginInvoke`, `Control.BeginInvoke` и `SynchronizationContext.Post`) являются асинхронными, как и методы, которые были написаны в разделе “`TaskCompletionSource`”, включая `Delay`.

Что собой представляет асинхронное программирование

Принцип асинхронного программирования состоит в том, что длительно выполняющиеся (или потенциально длительно выполняющиеся) функции реализуются асинхронным образом. Он отличается от традиционного подхода синхронной реализации длительно выполняющихся функций с последующим их вызовом в новом потоке или в задаче для введения параллелизма по мере необходимости.

Отличие от синхронного подхода заключается в том, что параллелизм инициируется *внутри* длительно выполняющейся функции, а не *за ее пределами*. В результате появляются два преимущества.

- Параллельное выполнение с интенсивным вводом-выводом может быть реализовано без связывания потоков (как было продемонстрировано в разделе “`TaskCompletionSource`” ранее в главе), улучшая показатели масштабируемости и эффективности.
- Обогащенные клиентские приложения в итоге содержат меньше кода в рабочих потоках, что упрощает достижение безопасности в отношении потоков.

В свою очередь это приводит к двум различающимся сценариям использования асинхронного программирования. Первый из них связан с написанием (обычно серверных) приложений, которые эффективно обрабатывают большой объем параллельных операций ввода-вывода. Проблемой здесь является не обеспечение безопасности к потокам (т.к. совместно используемое состояние обычно минимально), а достижение эффективности потоков; в частности, отсутствие потребления одного потока на сетевой запрос. Следовательно, в таком контексте выигрыш от асинхронности получают только операции с интенсивным вводом-выводом.

Второй сценарий применения касается упрощения поддержки безопасности в отношении потоков внутри обогащенных клиентских приложений. Он особенно актуален с ростом размера программы, поскольку для борьбы со сложностью мы обычно проводим рефакторинг крупных методов в методы меньших размеров, получая в результате цепочки методов, которые вызывают друг друга (*графы вызовов*).

Если любая операция внутри традиционного графа *синхронных* вызовов является длительно выполняющейся, тогда мы должны запускать целый график вызовов в рабочем потоке, чтобы обеспечить отзывчивость пользовательского

интерфейса. Таким образом, мы в конечном итоге получаем единственную параллельную операцию, которая охватывает множество методов (*крупномодульный параллелизм*), что требует учета безопасности к потокам для каждого метода в графе.

В случае графа асинхронных вызовов мы не должны запускать поток до тех пор, пока это не станет действительно необходимым — как правило, в нижней части графа (или вообще не запускать поток для операций с интенсивным вводом-выводом). Все остальные методы могут выполняться полностью в потоке пользовательского интерфейса со значительно упрощенной поддержкой безопасности в отношении потоков. В результате получается *мелкомодульный параллелизм* — последовательность небольших параллельных операций, между которыми выполнение возвращается в поток пользовательского интерфейса.



Чтобы извлечь из этого выгоду, операции с интенсивным вводом-выводом и интенсивными вычислениями должны быть реализованы асинхронным образом; хорошее эмпирическое правило предусматривает асинхронную реализацию любой операции, выполнение которой может занять более 50 мс.

(Оборотная сторона заключается в том, что чрезмерно мелкомодульная асинхронность может нанести ущерб производительности, потому что с асинхронными операциями связаны определенные накладные расходы, как будет показано в разделе “Оптимизация” далее в главе.)

В настоящей главе мы сосредоточим внимание главным образом на более сложном сценарии с обогащенным клиентом. В разделах “Параллелизм и TCP” и “Реализация HTTP-сервера” главы 16 будут приведены два примера, иллюстрирующие сценарий с интенсивным вводом-выводом.



Инфраструктура UWP поддерживает асинхронное программирование до момента, когда синхронные версии некоторых длительно выполняющихся методов либо не доступны, либо генерируют исключения. Взамен потребуется вызывать асинхронные методы, возвращающие объекты задач (или объекты, которые могут быть преобразованы в задачи посредством расширяющего метода `AsTask`).

Асинхронное программирование и продолжение

Задачи идеально подходят для асинхронного программирования, т.к. они поддерживают признаки продолжения, которые являются жизненно важными в реализации асинхронности (взгляните на метод `Delay`, реализованный в разделе “`TaskCompletionSource`” ранее в главе). При написании метода `Delay` мы использовали класс `TaskCompletionSource`, который предлагает стандартный способ реализации асинхронных методов с интенсивным вводом-выводом “нижнего уровня”.

В случае методов с интенсивными вычислениями для инициирования параллелизма, связанного с потоками, мы применяем метод `Task.Run`. Асинхронный

метод создается просто за счет возвращения вызывающему компоненту объекта задачи. Асинхронное программирование отличается тем, что мы стремимся поступать подобным образом на как можно более низком уровне графа вызовов. Тогда высокоуровневые методы в обогащенных клиентских приложениях могут быть оставлены в потоке пользовательского интерфейса и получать доступ к элементам управления и совместно используемому состоянию, не порождая проблем с безопасностью к потокам. В целях иллюстрации рассмотрим показанный ниже метод, который вычисляет и подсчитывает простые числа, используя все доступные ядра (класс `ParallelEnumerable` обсуждается в главе 22):

```
int GetPrimesCount (int start, int count)
{
    return
        ParallelEnumerable.Range (start, count).Count (n =>
            Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0));
}
```

Детали того, как работает алгоритм, не особенно важны; имеет значение лишь то, что метод требует некоторого времени на выполнение. Мы можем продемонстрировать это, написав другой метод, который вызывает `GetPrimesCount`:

```
void DisplayPrimeCounts()
{
    for (int i = 0; i < 10; i++)
        Console.WriteLine (GetPrimesCount (i*1000000 + 2, 1000000) +
            " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1));
    // " простые числа между " + (i*1000000) + " и " + ((i+1)*1000000-1));
    Console.WriteLine ("Done!");
}
```

Вот как выглядит вывод:

```
78498 primes between 0 and 999999
70435 primes between 1000000 and 1999999
67883 primes between 2000000 and 2999999
66330 primes between 3000000 and 3999999
65367 primes between 4000000 and 4999999
64336 primes between 5000000 and 5999999
63799 primes between 6000000 and 6999999
63129 primes between 7000000 and 7999999
62712 primes between 8000000 and 8999999
62090 primes between 9000000 and 9999999
```

Теперь у нас есть *граф вызовов* с методом `DisplayPrimeCounts`, обращающимся к методу `GetPrimesCount`. Для простоты внутри `DisplayPrimeCounts` применяется метод `Console.WriteLine`, хотя в реальности, скорее всего, будут обновляться элементы управления пользовательского интерфейса в обогащенном клиентском приложении, что демонстрируется позже. Крупномодульный параллелизм для такого графа вызовов можно инициировать следующим образом:

```
Task.Run (() => DisplayPrimeCounts());
```

В случае асинхронного подхода с мелкомодульным параллелизмом мы начинаем с написания асинхронной версии метода `GetPrimesCount`:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
    return Task.Run (() =>
        ParallelEnumerable.Range (start, count).Count (n =>
            Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

Важность языковой поддержки

Теперь мы должны модифицировать метод `DisplayPrimeCounts` так, чтобы он вызывал `GetPrimesCountAsync`. Именно здесь в игру вступают ключевые слова `await` и `async` языка C#, поскольку поступить по-другому намного сложнее, чем может показаться. Если мы просто изменим цикл, как показано ниже:

```
for (int i = 0; i < 10; i++)
{
    var awaiter = GetPrimesCountAsync (i*1000000 + 2, 1000000).GetAwaiter();
    awaiter.OnCompleted (() =>
        Console.WriteLine (awaiter.GetResult() + " primes between... "));
}
Console.WriteLine ("Done");
```

то цикл быстро пройдет через 10 итераций (методы не являются блокирующими) и все 10 операций будут выполняться параллельно (с ранним выводом строки "Done").



Выполнять приведенные задачи параллельно в данном случае нежелательно, т.к. их внутренние реализации уже распараллелены; это приведет лишь к более длительному ожиданию первых результатов (и нарушению упорядочения).

Однако существует намного более распространенная причина для *последовательного выполнения задач* — ситуация, когда задача Б зависит от результатов выполнения задачи А. Например, при выборке веб-страницы DNS-поиск должен предшествовать HTTP-запросу.

Для обеспечения последовательного выполнения следующую итерацию цикла нужно запускать из самого продолжения, что означает устранение цикла `for` и реализацию рекурсивного вызова в продолжении:

```
void DisplayPrimeCounts()
{
    DisplayPrimeCountsFrom (0);
}

void DisplayPrimeCountsFrom (int i)
{
    var awaiter = GetPrimesCountAsync (i*1000000 + 2, 1000000).GetAwaiter();
    awaiter.OnCompleted (() =>
    {
```

```

        Console.WriteLine (awaiter.GetResult() + " primes between...");  

        if (i++ < 10) DisplayPrimeCountsFrom (i);  

        else Console.WriteLine ("Done");  

    } );  

}

```

Все становится еще хуже, если необходимо сделать асинхронным *сам* метод `DisplayPrimesCount`, возвращая объект задачи, которая отправляет сигнал о своем завершении. Достижение такой цели требует создания объекта `TaskCompletionSource`:

```

Task DisplayPrimeCountsAsync()  

{  

    var machine = new PrimesStateMachine();  

    machine.DisplayPrimeCountsFrom (0);  

    return machine.Task;  

}  

class PrimesStateMachine  

{  

    TaskCompletionSource<object> _tcs =  

        new TaskCompletionSource<object>();  

    public Task Task { get { return _tcs.Task; } }  

    public void DisplayPrimeCountsFrom (int i)  

    {  

        var awariter = GetPrimesCountAsync (i*1000000+2, 1000000).GetAwaiter();  

        awariter.OnCompleted (() =>  

        {  

            Console.WriteLine (awariter.GetResult());  

            if (i++ < 10) DisplayPrimeCountsFrom (i);  

            else { Console.WriteLine ("Done"); _tcs.SetResult (null); }  

        });
    }
}

```

К счастью, всю работу подобного рода делают *асинхронные функции C#*. Благодаря новым ключевым словам `async` и `await` нам придется написать только следующий код:

```

async Task DisplayPrimeCountsAsync()  

{  

    for (int i = 0; i < 10; i++)  

        Console.WriteLine (await GetPrimesCountAsync (i*1000000 + 2, 1000000) +  

            " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1));  

    Console.WriteLine ("Done!");
}

```

Таким образом, ключевые слова `async` и `await` очень важны для реализации асинхронности без чрезмерной сложности. Давайте посмотрим, как они работают.



Взглянуть на данную проблему можно и по-другому: императивные конструкции циклов (`for`, `foreach` и т.д.) не очень хорошо сочетаются с признаками продолжения, поскольку они полагаются на *текущее локальное состояние* метода (т.е. сколько раз цикл планирует выполняться).

Хотя ключевые слова `async` и `await` предлагают одно решение, иногда решить проблему удается другим способом, заменяя императивные конструкции циклов их функциональными эквивалентами (другими словами, запросами LINQ). Это является основой библиотеки *Reactive Extensions* (Rx) и может оказаться удачным вариантом, когда в отношении результата нужно выполнить операции запросов или скомбинировать несколько последовательностей. Недостаток связан с тем, что во избежание блокировки инфраструктура Rx оперирует на последовательностях с *активным* источником, которые могут оказаться концептуально сложными.

Асинхронные функции в C#

Ключевые слова `async` и `await` позволяют писать асинхронный код, который обладает той же самой структурой и простотой, что и синхронный код, а также устранять необходимость во вспомогательном коде, присущем асинхронному программированию.

Ожидание

Ключевое слово `await` упрощает присоединение признаков продолжения. Рассмотрим базовый сценарий. Приведенные ниже строки:

```
var результат = await выражение;
оператор(ы);
```

компилятор развернет в следующий функциональный эквивалент:

```
var awaiter = выражение.GetAwaiter();
awaiter.OnCompleted(() =>
{
    var результат = awaiter.GetResult();
    оператор(ы);
});
```



Компилятор также выпускает код для замыкания продолжения в случае синхронного завершения (см. раздел “Оптимизация” далее в главе) и для обработки разнообразных нюансов, которые мы затронем в последующих разделах.

В целях демонстрации вернемся к ранее написанному асинхронному методу, который вычисляет и подсчитывает простые числа:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
    return Task.Run(() =>
        ParallelEnumerable.Range (start, count).Count (n =>
            Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

Используя ключевое слово `await`, его можно вызвать следующим образом:

```
int result = await GetPrimesCountAsync (2, 1000000);
Console.WriteLine (result);
```

Чтобы код скомпилировался, к содержащему такой вызов методу понадобится добавить модификатор `async`:

```
async void DisplayPrimesCount ()
{
    int result = await GetPrimesCountAsync (2, 1000000);
    Console.WriteLine (result);
}
```

Модификатор `async` сообщает компилятору о необходимости трактовать `await` как ключевое слово, а не идентификатор, что привело бы к неоднозначности внутри данного метода (это гарантирует успешную компиляцию кода, написанного до выхода версии C# 5, где слово `await` использовалось в качестве идентификатора). Модификатор `async` может применяться только к методам (и лямбда-выражениям), которые возвращают `void` либо (как будет показано позже) тип `Task` или `Task<TResult>`.



Модификатор `async` подобен модификатору `unsafe` в том, что он не дает никакого эффекта на сигнатуре или открытых метаданных метода, а воздействует, только когда находится *внутри* метода. По этой причине не имеет смысла использовать `async` в интерфейсе. Однако вполне законно, например, вводить `async` при переопределении виртуального метода, не являющегося асинхронным, при условии сохранения сигнатуры метода в неизменном виде.

Методы с модификатором `async` называются *асинхронными функциями*, т.к. сами они обычно асинхронны. Чтобы увидеть почему, давайте посмотрим, каким образом процесс выполнения проходит через асинхронную функцию.

Встретив выражение `await`, процесс выполнения (обычно) производит возврат в вызывающий код — почти как оператор `yield return` в итераторе. Но перед возвратом исполняющая среда присоединяет к ожидающей задаче признак продолжения, который гарантирует, что когда задача завершится, управление перейдет обратно в метод и продолжит с места, где оно его оставило. Если задача отказывает, тогда ее исключение генерируется повторно, а в противном случае выражению `await` присваивается возвращаемое значение задачи. Все сказанное можно резюмировать, просмотрев логическое расширение только что рассмотренного асинхронного метода:

```
void DisplayPrimesCount ()
{
    var awaier = GetPrimesCountAsync (2, 1000000).GetAwaiter();
    awaier.OnCompleted (() =>
    {
        int result = awaier.GetResult();
        Console.WriteLine (result);
    });
}
```

Выражение, к которому применяется `await`, обычно является задачей; тем не менее, компилятор устроит любой объект с методом `GetAwaiter`, который возвращает *объект ожидания* (реализующий метод `INotifyCompletion.OnCompleted` и имеющий должным образом типизированный метод `GetResult` и свойство `bool IsCompleted`).

Обратите внимание, что выражение `await` оценивается как относящееся к типу `int`; причина в том, что ожидаемым выражением было `Task<int>` (метод `GetAwaiter().GetResult` которого возвращает тип `int`). Ожидание необобщенной задачи вполне законно и генерирует выражение `void`:

```
await Task.Delay (5000);
Console.WriteLine ("Five seconds passed!");
```

Захват локального состояния

Реальная мощь выражений `await` заключается в том, что они могут находиться практически в любом месте кода. В частности, выражение `await` может присутствовать на месте любого выражения (внутри асинхронной функции) кроме оператора `lock` или контекста `unsafe`.

В следующем примере `await` располагается в цикле:

```
async void DisplayPrimeCounts()
{
    for (int i = 0; i < 10; i++)
        Console.WriteLine (await GetPrimesCountAsync (i*1000000+2, 1000000));
```

При первом выполнении метода `GetPrimesCountAsync` управление возвращается вызывающему коду из-за выражения `await`. Когда метод завершается (или отказывает), выполнение возобновляется с места, где оно его покинуло, с сохраненными значениями локальных переменных и счетчиков циклов.

В отсутствие ключевого слова `await` простейшим эквивалентом мог бы служить пример, реализованный в разделе “Важность языковой поддержки” ранее в главе. Однако компилятор реализует более общую стратегию преобразования таких методов в конечные автоматы (очень похоже на то, как он поступает с итераторами).

В возобновлении выполнения после выражения `await` компилятор полагается на признаки продолжения (согласно шаблону объектов ожидания). Это значит, что в случае запуска в потоке пользовательского интерфейса контекст синхронизации гарантирует, что выполнение будет возобновлено в том же самом потоке. В противном случае выполнение возобновляется в любом потоке, где задача была завершена. Смена потока не оказывает влияния на порядок выполнения и несущественна, если только вы каким-то образом не зависите от родства потоков, возможно, из-за использования локального хранилища потока (см. раздел “Локальное хранилище потока” в главе 21). Здесь уместна аналогия с ситуацией, когда вы ловите такси, чтобы добраться из одного места в другое. При наличии контекста синхронизации в вашем распоряжении всегда будет один и тот же таксомотор, а без контекста синхронизации таксомоторы каждый раз, возможно, окажутся разными. Хотя путешествие в любом случае будет в основном тем же самым.

Ожидание в пользовательском интерфейсе

Мы можем продемонстрировать асинхронные функции в более практическом контексте, реализовав простой пользовательский интерфейс, который остается отзывчивым во время вызова метода с интенсивными вычислениями. Давайте начнем с синхронного решения:

```
class TestUI : Window
{
    Button _button = new Button { Content = "Go" };
    TextBlock _results = new TextBlock();
    public TestUI()
    {
        var panel = new StackPanel();
        panel.Children.Add (_button);
        panel.Children.Add (_results);
        Content = panel;
        _button.Click += (sender, args) => Go();
    }
    void Go()
    {
        for (int i = 1; i < 5; i++)
            _results.Text += GetPrimesCount (i * 1000000, 1000000) +
                " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1) +
                Environment.NewLine;
    }
    int GetPrimesCount (int start, int count)
    {
        return ParallelEnumerable.Range (start, count).Count (n =>
            Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0));
    }
}
```

После щелчка на кнопке Go (Запуск) приложение перестает быть отзывчивым на время, необходимое для выполнения кода с интенсивными вычислениями. Решение превращается в асинхронное за два шага. Первый шаг связан с переключением на асинхронную версию метода GetPrimesCount, который применялся в предыдущем примере:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
    return Task.Run (() =>
        ParallelEnumerable.Range (start, count).Count (n =>
            Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

Второй шаг предусматривает изменение метода Go для вызова метода GetPrimesCountAsync:

```
async void Go()
{
    _button.IsEnabled = false;
    for (int i = 1; i < 5; i++)
        _results.Text += await GetPrimesCountAsync (i * 1000000, 1000000) +
            " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1) +
            Environment.NewLine;
    _button.IsEnabled = true;
}
```

Приведенный выше код демонстрирует простоту программирования с использованием асинхронных функций: все делается как при синхронном подходе, но вместо блокирования функций и их ожидания посредством `await` производится вызов асинхронных функций. В рабочем потоке запускается только код внутри метода `GetPrimesCountAsync`; код в методе Go “арендуется” время у потока пользовательского интерфейса. Можно было бы сказать, что метод Go выполняется *псевдопараллельно* с циклом сообщений (т.е. его выполнение пересекается с другими событиями, которые обрабатывает поток пользовательского интерфейса). Благодаря такому псевдопараллизму единственной точкой, где может возникнуть вытеснение, является выполнение `await`. В итоге обеспечение безопасности к потокам упрощается: в данном случае может возникнуть только одна проблема — *реентерабельность* (из-за повторного щелчка на кнопке во время выполнения метода Go, чего мы избегаем, делая кнопку недоступной). Подлинный параллизм происходит ниже в стеке вызовов, внутри кода, вызываемого методом `Task.Run`. Чтобы извлечь преимущества из такой модели, по-настоящему параллельный код избегает доступа к совместно используемому состоянию или элементам управления пользовательского интерфейса.

Рассмотрим еще один пример, в котором вместо вычисления простых чисел загружается несколько веб-страниц с суммированием их длин. В .NET доступно множество асинхронных методов, возвращающих задачи, один из которых определен в классе `WebClient` внутри пространства имен `System.Net`. Метод `DownloadDataTaskAsync` асинхронно загружает URI в байтовый массив, возвращая объект `Task<byte[]>`, так что в результате ожидания можно получить массив `byte[]`. Давайте перепишем метод Go:

```
async void Go()
{
    _button.IsEnabled = false;
    string[] urls = "www.albahari.com www.oreilly.com www.linqpad.net".Split();
    int totalLength = 0;
    try
    {
        foreach (string url in urls)
        {
            var uri = new Uri ("http://" + url);
            byte[] data = await new WebClient().DownloadDataTaskAsync (uri);
            _results.Text += "Length of " + url + " is " + data.Length +
                Environment.NewLine; // длина загруженных данных
            totalLength += data.Length;
        }
        _results.Text += "Total length: " + totalLength; // общая длина
    }
    catch (WebException ex)
    {
        _results.Text += "Error: " + ex.Message; // ошибка
    }
    finally { _button.IsEnabled = true; }
}
```

И снова код отражает то, как он был бы реализован синхронным образом, включая применение блоков `catch` и `finally`. Хотя после первого `await` управление возвращается вызывающему коду, блок `finally` не выполняется вплоть до логического завершения метода (после выполнения всего его кода либо по причине раннего возвращения из-за оператора `return` или возникновения необработанного исключения).

Полезно посмотреть, что в точности происходит. Для начала необходимо вернуться к псевдокоду, который выполняет цикл сообщений в потоке пользовательского интерфейса:

```
Установить для этого потока контекст синхронизации WPF
while (приложение не завершено)
{
    Ожидать появления чего-нибудь в очереди сообщений
    Что-то получено: к какому виду сообщений оно относится?
        Сообщение клавиатуры/мыши -> запустить обработчик событий
        Пользовательское сообщение BeginInvoke/Invoke -> выполнить делегат
}
```

Обработчики событий, присоединяемые к элементам пользовательского интерфейса, выполняются через такой цикл сообщений. Когда запускается наш метод `Go`, выполнение продолжается до выражения `await`, после чего управление возвращается в цикл сообщений (освобождая пользовательский интерфейс для реагирования на дальнейшие события). Однако расширение компилятором выражения `await` гарантирует, что перед возвращением продолжение настроено так, чтобы выполнение возобновлялось там, где оно было прекращено до завершения задачи. И поскольку ожидание с помощью `await` происходит в потоке пользовательского интерфейса, то признак продолжения отправляется контексту синхронизации, который выполняет его через цикл сообщений, сохраняя выполнение всего метода `Go` псевдопараллельным с потоком пользовательского интерфейса. Подлинный параллелизм (с интенсивным вводом-выводом) происходит внутри реализации метода `DownloadDataTaskAsync`.

Сравнение с крупномодульным параллелизмом

До выхода версии C# 5 асинхронное программирование было затруднено не только из-за отсутствия языковой поддержки, но и потому, что асинхронная функциональность в .NET Framework открывалась через неуклюжие шаблоны EAP и APM (см. раздел “Устаревшие шаблоны” далее в главе), а не через методы, возвращающие объекты задач.

Популярным обходным путем являлся крупномодульный параллелизм (для чего даже был предусмотрен тип по имени `BackgroundWorker`). Мы можем продемонстрировать крупномодульную асинхронность на исходном синхронном примере с методом `GetPrimesCount`, изменив обработчик событий кнопки, как показано ниже:

```
...
_button.Click += (sender, args) =>
{
    _button.IsEnabled = false;
    Task.Run(() => Go());
};
```

(Мы решили использовать метод Task.Run вместо класса BackgroundWorker из-за того, что BackgroundWorker никак бы не упростил данный конкретный пример.) В любом случае конечный результат состоит в том, что целый граф синхронных вызовов (Go и GetPrimesCount) выполняется в рабочем потоке. И поскольку метод Go обновляет элементы пользовательского интерфейса, в код придется добавить вызовы Dispatcher.BeginInvoke:

```
void Go()
{
    for (int i = 1; i < 5; i++)
    {
        int result = GetPrimesCount (i * 1000000, 1000000);
        Dispatcher.BeginInvoke (new Action () =>
            results.Text += result + " primes between " + (i*1000000) +
            " and " + ((i+1)*1000000-1) + Environment.NewLine));
    }
    Dispatcher.BeginInvoke (new Action () => _button.IsEnabled = true));
}
```

В отличие от асинхронной версии цикл сам выполняется в рабочем потоке. Это может казаться безобидным, но даже в таком простом случае применение многопоточности привело к возникновению условия состязаний. (Смогли его заметить? Если нет, тогда запустите программу: условие состязаний почти наверняка станет очевидным.)

Реализация отмены и сообщения о ходе работ создает больше возможностей для ошибок, связанных с нарушением безопасности к потокам, как делает любой дополнительный код в методе. Например, предположим, что верхний предел для цикла не закодирован жестко, а поступает из вызова метода:

```
for (int i = 1; i < GetUpperBound(); i++)
```

Далее представим, что метод GetUpperBound читает значение из конфигурационного файла, который ленивым образом загружается с диска при первом вызове. Весь этот код теперь выполняется в рабочем потоке — код, который весьма вероятно не является безопасным к потокам. В том и заключается опасность запуска рабочих потоков на высоких уровнях внутри графа вызовов.

Написание асинхронных функций

Что касается любой асинхронной функции, то возвращаемый тип void можно заменить типом Task, чтобы сделать сам метод *пригодным* для асинхронного выполнения (и ожидания с помощью await). Никаких других изменений вносить не придется:

```
async Task PrintAnswerToLife() // Вместо void можно возвращать Task
{
    await Task.Delay (5000);
    int answer = 21 * 2;
    Console.WriteLine (answer);
}
```

Обратите внимание, что в теле метода мы не возвращаем объект задачи явным образом. Компилятор произведет задачу, которая будет отправлять сигнал о завершении данного метода (или о возникновении необработанного исключения). В итоге упрощается создание цепочек асинхронных вызовов:

```
async Task Go()
{
    await PrintAnswerToLife();
    Console.WriteLine ("Done");
}
```

Так как метод Go объявлен с возвращаемым типом Task, сам Go допускает ожидание посредством await.

Компилятор расширяет асинхронные функции, возвращающие задачи, в код, использующий класс TaskCompletionSource для создания задачи, которая затем отправляет сигнал о завершении или отказе.

Оставив в стороне нюансы, мы можем развернуть метод PrintAnswerToLife в такой функциональный эквивалент:

```
Task PrintAnswerToLife()
{
    var tcs = new TaskCompletionSource<object>();
    var awaiter = Task.Delay (5000).GetAwaiter();
    awaiter.OnCompleted (() =>
    {
        try
        {
            awaiter.GetResult();      // Сгенерировать повторно любые исключения
            int answer = 21 * 2;
            Console.WriteLine (answer);
            tcs.SetResult (null);
        }
        catch (Exception ex) { tcs.SetException (ex); }
    });
    return tcs.Task;
}
```

Следовательно, всякий раз, когда возвращающий задачу асинхронный метод завершается, управление переходит обратно к месту его ожидания (благодаря признаку продолжения).



В сценарии с обогащенным клиентом управление перемещается с этой точки обратно в поток пользовательского интерфейса (если оно еще в нем не находится). В противном случае выполнение продолжается в любом потоке, куда был направлен признак продолжения, что означает отсутствие задержки при подъеме по графу асинхронных вызовов кроме первого “прыжка”, если он был инициирован потоком пользовательского интерфейса.

Возвращение Task<TResult>

Возвращать объект `Task<TResult>` можно, если в теле метода возвращается тип `TResult`:

```
async Task<int> GetAnswerToLife()
{
    await Task.Delay (5000);
    int answer = 21 * 2;
    return answer; // Метод имеет возвращаемый тип Task<int>,
                    // поэтому возвратить int
}
```

Внутренне это приводит к тому, что объекту `TaskCompletionSource` отправляется сигнал со значением, отличающимся от `null`. Мы можем продемонстрировать работу метода `GetAnswerToLife`, вызвав его из метода `PrintAnswerToLife` (который в свою очередь вызывается из `Go`):

```
async Task Go()
{
    await PrintAnswerToLife();
    Console.WriteLine ("Done");
}

async Task PrintAnswerToLife()
{
    int answer = await GetAnswerToLife();
    Console.WriteLine (answer);
}

async Task<int> GetAnswerToLife()
{
    await Task.Delay (5000);
    int answer = 21 * 2;
    return answer;
}
```

В сущности, мы преобразовали исходный метод `PrintAnswerToLife` в два метода — с той же легкостью, как если бы программировали синхронным образом. Сходство с синхронным программированием является умышленным; вот синхронный эквивалент нашего графа вызовов, для которого вызов метода `Go` дает тот же самый результат после блокирования на протяжении пяти секунд:

```
void Go()
{
    PrintAnswerToLife();
    Console.WriteLine ("Done");
}

void PrintAnswerToLife()
{
    int answer = GetAnswerToLife();
    Console.WriteLine (answer);
}

int GetAnswerToLife()
{
    Thread.Sleep (5000);
    int answer = 21 * 2;
    return answer;
}
```



Тем самым также иллюстрируется базовый принцип проектирования с применением асинхронных функций в C#.

1. Напишите синхронные версии своих методов.
2. Замените вызовы *синхронных* методов вызовами *асинхронных* методов и примените к ним await.
3. За исключением методов “верхнего уровня” (обычно обработчиков событий для элементов управления пользовательского интерфейса) поменяйте возвращаемые типы асинхронных методов на Task или Task<TResult>, чтобы они поддерживали await.

Способность компилятора производить задачи для асинхронных функций означает, что явно создавать объект TaskCompletionSource придется главным образом только в (относительно редком) случае методов **нижнего уровня**, которые инициируют параллелизм с интенсивным вводом-выводом. (Для методов, инициирующих параллелизм с интенсивными вычислениями, создается задача с помощью метода Task.Run.)

Выполнение графа асинхронных вызовов

Чтобы ясно увидеть, как все выполняется, полезно реорганизовать код следующим образом:

```
async Task Go()
{
    var task = PrintAnswerToLife();
    await task; Console.WriteLine ("Done");
}

async Task PrintAnswerToLife()
{
    var task = GetAnswerToLife();
    int answer = await task; Console.WriteLine (answer);
}

async Task<int> GetAnswerToLife()
{
    var task = Task.Delay (5000);
    await task; int answer = 21 * 2; return answer;
}
```

Метод Go вызывает метод PrintAnswerToLife, который вызывает метод GetAnswerToLife, а тот в свою очередь вызывает метод Delay и затем ожидает. Наличие await приводит к тому, что управление возвращается методу PrintAnswerToLife, который сам ожидает, возвращая управление методу Go, который тоже ожидает и возвращает управление вызывающему коду. Все происходит синхронно в потоке, вызвавшем метод Go; это короткая *синхронная* фаза выполнения.

Спустя пять секунд запускается продолжение на Delay и управление возвращается методу GetAnswerToLife в потоке из пула. (Если мы начинали в потоке пользовательского интерфейса, то управление возвратится в него.) Затем выполняются оставшиеся операторы в методе GetAnswerToLife, после чего за-

дача `Task<int>` данного метода завершается с результатом 42 и инициируется продолжение в методе `PrintAnswerToLife`, что приведет к выполнению оставшихся операторов в этом методе. Процесс продолжается до тех пор, пока задача метода `Go` не выдаст сигнал о своем завершении.

Поток выполнения соответствует показанному ранее графу синхронных вызовов, т.к. мы следуем шаблону, при котором к каждому асинхронному методу сразу после вызова применяется `await`. В результате создается последовательный поток без параллелизма или перекрывающегося выполнения внутри графа вызовов. Каждое выражение `await` образует “брешь” в выполнении, после которой программа возобновляет работу с того места, где она ее оставила.

Параллелизм

Вызов асинхронного метода без его ожидания позволяет коду, который за ним следует, выполняться параллельно. В приведенных ранее примерах вы могли отметить наличие кнопки, обработчик события которой вызывал метод `Go` так, как показано ниже:

```
_button.Click += (sender, args) => Go();
```

Несмотря на то что `Go` является асинхронным методом, мы не можем применить к нему `await`, и это действительно то, что соответствует параллелизму, необходимому для поддержки отзывчивого пользовательского интерфейса.

Тот же самый принцип можно использовать для запуска двух асинхронных операций параллельно:

```
var task1 = PrintAnswerToLife();
var task2 = PrintAnswerToLife();
await task1; await task2;
```

(За счет ожидания обеих операций впоследствии мы “заканчиваем” параллелизм в данной точке. Позже мы покажем, как комбинатор задач `WhenAll` помогает в реализации такого шаблона.)

Параллелизм, организованный подобным образом, получается независимо от того, инициированы ли операции в потоке пользовательского интерфейса, хотя существует отличие в том, как он проявляется. В обоих случаях мы получаем тот же самый “подлинный” параллелизм в операциях нижнего уровня, которые его инициируют (таких как `Task.Delay` или код, предоставленный методу `Task.Run`). Методы, находящиеся выше в стеке вызовов, будут по-настоящему параллельными, только если операция была инициирована без наличия контекста синхронизации; иначе они окажутся псевдопараллельными (и упростят обеспечение безопасности к потокам), согласно чему единственным местом, где может произойти вытеснение, является оператор `await`. Это позволяет, например, определить совместно используемое поле `_x` и инкрементировать его в методе `GetAnswerToLife` без блокирования:

```
async Task<int> GetAnswerToLife()
{
    _x++;
    await Task.Delay (5000);
    return 21 * 2;
}
```

(Тем не менее, мы не можем предполагать, что `_x` имеет одно и то же значение до и после `await`.)

Асинхронные лямбда-выражения

Точно так же как обычные именованные методы могут быть асинхронными:

```
async Task NamedMethod()
{
    await Task.Delay (1000);
    Console.WriteLine ("Foo");
}
```

асинхронными способны быть и *неименованные* методы (лямбда-выражения и анонимные методы), если они предварены ключевым словом `async`:

```
Func<Task> unnamed = async () =>
{
    await Task.Delay (1000);
    Console.WriteLine ("Foo");
};
```

Вызывать и применять `await` к ним можно одинаково:

```
await NamedMethod();
await unnamed();
```

Асинхронные лямбда-выражения можно использовать при подключении обработчиков событий:

```
myButton.Click += async (sender, args) =>
{
    await Task.Delay (1000);
    myButton.Content = "Done";
};
```

Это более лаконично, чем приведенный ниже код, который обеспечивает тот же самый результат:

```
myButton.Click += ButtonHandler;
...
async void ButtonHandler (object sender, EventArgs args)
{
    await Task.Delay (1000);
    myButton.Content = "Done";
};
```

Асинхронные лямбда-выражения могут также возвращать тип `Task<TResult>`:

```
Func<Task<int>> unnamed = async () =>
{
    await Task.Delay (1000);
    return 123;
};
int answer = await unnamed();
```

Асинхронные потоки данных

С помощью `yield return` вы можете писать итератор, а с помощью `await` — асинхронную функцию. *Асинхронные потоки* (появившиеся в C# 8) объединяют эти концепции и позволяют реализовать итератор, который ожидает, выдавая элементы асинхронным образом. Такая поддержка основана на следующей паре интерфейсов, которые представляют собой асинхронные аналоги интерфейсов перечисления, описанных в разделе “Перечисление и итераторы” главы 4:

```
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator (...);
}

public interface IAsyncEnumerator<out T>: IAsyncDisposable
{
    T Current { get; }
    ValueTask<bool> MoveNextAsync ();
}
```

`ValueTask<T>` — это структура, которая является оболочкой для `Task<T>` и по поведению похожа на `Task<T>`, но обеспечивает более эффективное выполнение, когда задача завершается синхронно (что часто может происходить при перечислении последовательности). Обсуждение отличий ищите в разделе “`ValueTask<T>`” далее в главе. Интерфейс `IAsyncDisposable` представляет собой асинхронную версию `IDisposable`; он предоставляет возможность выполнения очистки, если вы решите вручную реализовывать интерфейсы:

```
public interface IAsyncDisposable
{
    ValueTask DisposeAsync ();
}
```



Действие по извлечению каждого элемента из последовательности (`MoveNextAsync`) является асинхронной операцией, поэтому асинхронные потоки подходят, когда элементы поступают постепенно (например, как при обработке данных из видеопотока). Напротив, следующий тип лучше подходит, когда задерживается последовательность *как единое целое*, но когда элементы поступают, то поступают все вместе:

```
Task<IEnumerable<T>>
```

Чтобы сгенерировать асинхронный поток, необходимо написать метод, в котором объединены принципы итераторов и асинхронных методов. Другими словами, метод должен включать и `yield return`, и `await`, а также возвращать `IAsyncEnumerable<T>`:

```
async IAsyncEnumerable<int> RangeAsync (
    int start, int count, int delay)
{
```

```
        for (int i = start; i < start + count; i++)
    {
        await Task.Delay (delay);
        yield return i;
    }
}
```

Для применения асинхронного потока нужно использовать оператор `await foreach`:

```
await foreach (var number in RangeAsync (0, 10, 500))
    Console.WriteLine (number);
```

Обратите внимание, что данные поступают постоянно каждые 500 мс (или в реальности, когда они становятся доступными). Сравните это с похожей конструкцией, использующей `Task<IEnumerable<T>>`, для которой данные не возвращаются до тех пор, пока не будет доступной последняя порция данных:

```
static async Task<IEnumerable<int>> RangeTaskAsync (int start, int count,
                                                       int delay)
{
    List<int> data = new List<int>();
    for (int i = start; i < start + count; i++)
    {
        await Task.Delay (delay);
        data.Add (i);
    }
    return data;
}
```

А вот как использовать асинхронный поток посредством оператора `foreach`:

```
foreach (var data in await RangeTaskAsync(0, 10, 500))
    Console.WriteLine (data);
```

Запрашивание `IAsyncEnumerable<T>`

В NuGet-пакете `System.Linq.Async` определены операции LINQ, работающие с `IAsyncEnumerable<T>`, которые позволяют писать запросы во многом подобно тому, как делалось бы с применением `IEnumerable<T>`.

Например, мы можем написать запрос LINQ для метода `RangeAsync`, определенного в предыдущем разделе, следующим образом:

```
IAsyncEnumerable<int> query =
    from i in RangeAsync (0, 10, 500)
    where i % 2 == 0                                // Только четные числа.
    select i * 10;                                    // Умножить на 10.

await foreach (var number in query)
    Console.WriteLine (number);
```

В результате выводятся числа 0, 20, 40 и т.д.



Если вы знакомы с библиотекой Rx, то можете также извлечь преимущества из ее (более мощных) операций запросов, вызывая расширяющий метод `ToObservable`, который преобразует реализацию `IAsyncEnumerable<T>` в `IObservable<T>`. Кроме того, доступен расширяющий метод `ToAsyncEnumerable`, предназначенный для преобразования в обратном направлении.

`IAsyncEnumerable<T>` в ASP.NET Core

Действия контроллера ASP.NET Core теперь могут возвращать `IAsyncEnumerable<T>`. Такие методы должны быть помечены как `async`. Например:

```
[HttpGet]
public async IAsyncEnumerable<string> Get()
{
    using var dbContext = new BookContext();
    await foreach (var title in dbContext.Books
        .Select(b => b.Title)
        .AsAsyncEnumerable())
    {
        yield return title;
    }
}
```

Асинхронные методы в WinRT

В случае разработки приложений UWP вам придется иметь дело с типами WinRT, определенными в ОС. Эквивалентом `Task` в WinRT является тип `IAsyncAction`, а эквивалентом `Task<TResult>` — тип `IAsyncOperation<TResult>`. Для операций, которые сообщают о ходе работ, эквивалентами будут `IAsyncActionWithProgress<TProgress>` и `IAsyncOperationWithProgress<TResult, TProgress>`. Все они определены в пространстве имен `Windows.Foundation`.

Выполнять преобразование в тип `Task` или `Task<TResult>` либо из него можно с помощью расширяющего метода `AsTask`:

```
Task<StorageFile> fileTask = KnownFolders.DocumentsLibrary.CreateFileAsync
    ("test.txt") .AsTask();
```

Или же можно реализовать ожидание напрямую:

```
StorageFile file = await KnownFolders.DocumentsLibrary.CreateFileAsync
    ("test.txt");
```



Из-за ограничений системы типов COM интерфейсы `IAsyncActionWithProgress<TProgress>` и `IAsyncOperationWithProgress<TResult, TProgress>` не основаны на `IAsyncAction`, как можно было бы ожидать. Взамен оба интерфейса унаследованы от общего базового типа по имени `IAsyncInfo`.

Метод `AsTask` также перегружен для приема признака отмены (см. раздел “Отмена” далее в главе). Он также принимает объект `IProgress<T>`, когда соединяется в цепочку с вариантами `WithProgress` (см. раздел “Сообщение о ходе работ” далее в главе).

Асинхронность и контексты синхронизации

Ранее уже было показано, что наличие контекста синхронизации играет важную роль в смысле отправки признаков продолжения. В случае асинхронных функций, возвращающих `void`, существует пара других, более тонких способов взаимодействия с контекстами синхронизации. Они являются не прямым результатом расширений, производимых компилятором C#, а функцией типов `AsyncMethodBuilder` из пространства имен `System.CompilerServices`, которое компилятор использует при расширении асинхронных функций.

Отправка исключений

В обогащенных клиентских приложениях общепринято полагаться на событие централизованной обработки исключений (`Application.DispatcherUnhandledException` в WPF) для учета необработанных исключений, сгенерированных в потоке пользовательского интерфейса. В приложениях ASP.NET Core похожую работу делает специальный фильтр `ExceptionFilterAttribute` в методе `ConfigureServices` из `Startup.cs`. Внутренне они функционируют, инициируя события пользовательского интерфейса (или конвейера методов обработки страниц в случае ASP.NET Core) в собственном блоке `try/catch`.

Асинхронные функции верхнего уровня затрудняют это. Рассмотрим следующий обработчик события щелчка на кнопке:

```
async void ButtonClick (object sender, RoutedEventArgs args)
{
    await Task.Delay(1000);
    throw new Exception ("Will this be ignored?"); //Будет ли это проигнорировано?
}
```

Когда на кнопке осуществляется щелчок и обработчик событий запускается, после оператора `await` управление обычно возвращается в цикл сообщений, и сгенерированное секунду спустя исключение не может быть перехвачено блоком `catch` в цикле сообщений.

Чтобы устранить проблему, структура `AsyncVoidMethodBuilder` перехватывает необработанные исключения (в асинхронных функциях, возвращающих `void`) и отправляет их контексту синхронизации, если он присутствует, гарантируя то, что события глобальной обработки исключений по-прежнему инициируются.



Компилятор применяет упомянутую логику только к асинхронным функциям, возвращающим `void`. Следовательно, если изменить `ButtonClick` для возвращения типа `Task` вместо `void`, тогда необработанное исключение приведет к отказу результирующей задачи, поскольку ему некуда больше двигаться (в результате давая **необнаруженнное исключение**).

Интересный нюанс связан с тем, что нет никакой разницы, где генерируется исключение — до или после `await`. Таким образом, в следующем примере исключение отправляется контексту синхронизации (если он существует), но не вызывающему коду:

```
async void Foo() { throw null; await Task.Delay(1000); }
```

(При отсутствии контекста синхронизации исключение будет распространяться в пуле потоков и приведет к завершению работы приложения.)

Причина, по которой исключение не возвращается обратно вызывающему компоненту, связана с обеспечением предсказуемости и согласованности. В следующем примере исключение InvalidOperationException всегда будет иметь один и тот эффект отказа результирующей задачи независимо от того, как выглядит *какое-то-Условие*:

```
async Task Foo()
{
    if (какое-то-Условие) await Task.Delay (100);
    throw new InvalidOperationException();
}
```

Итераторы работают аналогично:

```
IEnumerable<int> Foo() { throw null; yield return 123; }
```

В приведенном примере исключение никогда не возвращается прямо вызывающему коду: с исключением имеет дело только перечисляемая последовательность.

OperationStarted и OperationCompleted

Если контекст синхронизации присутствует, то асинхронные функции, возвращающие void, также вызывают его метод OperationStarted при входе в функцию и метод OperationCompleted, когда функция завершается.

Переопределение таких методов удобно при написании специального контекста синхронизации для проведения модульного тестирования асинхронных методов, возвращающих void. Данная тема обсуждается в блоге, посвященном параллельному программированию в .NET (<https://devblogs.microsoft.com/pfxteam>).

Оптимизация

Синхронное завершение

Возврат из асинхронной функции может произойти *перед* организацией ожидания. Рассмотрим следующий метод, который обеспечивает кеширование в процессе загрузки веб-страниц:

```
static Dictionary<string, string> _cache = new Dictionary<string, string>();
async Task<string> GetWebPageAsync (string uri)
{
    string html;
    if (_cache.TryGetValue (uri, out html)) return html;
    return _cache [uri] =
        await new WebClient().DownloadStringTaskAsync (uri);
}
```

Если URI присутствует в кеше, тогда управление возвращается вызывающему коду безо всякого ожидания, и метод возвращает объект задачи, которая уже сигнализирована. Это называется *синхронным завершением*.

Когда производится ожидание синхронно завершенной задачи, управление не возвращается вызывающему коду и не переходит обратно через признак продолжения — взамен выполнение продолжается со следующего оператора. Компилятор реализует такую оптимизацию, проверяя свойство `IsCompleted` объекта ожидания; другими словами, всякий раз, когда производится ожидание:

```
Console.WriteLine (await GetWebPageAsync ("http://oreilly.com"));
```

компилятор выпускает код для короткого замыкания продолжения в случае синхронного завершения:

```
var awaiter = GetWebPageAsync().GetAwaiter();
if (awaiter.IsCompleted)
    Console.WriteLine (awaiter.GetResult());
else
    awaiter.OnCompleted (() => Console.WriteLine (awaiter.GetResult()));
```



Ожидание асинхронной функции, которая завершается синхронно, все равно связано с (крайне) небольшими накладными расходами — примерно 20 нс на современных компьютерах.

Напротив, переход в пул потоков вызывает переключение контекста — возможно одну или две микросекунды, а переход в цикл обработки сообщений пользовательского интерфейса — минимум в десять раз больше (и еще больше, если пользовательский интерфейс занят).

Вполне законно даже писать асинхронные методы, для которых никогда не производится ожидание, хотя компилятор генерирует предупреждение:

```
async Task<string> Foo() { return "abc"; }
```

Такие методы могут быть полезны при переопределении виртуальных/абстрактных методов, если случится так, что ваша реализация не потребует асинхронности. (Примером могут служить методы `ReadAsync`/`WriteAsync` класса `MemoryStream`, которые рассматриваются в главе 15.) Другой способ достичь того же результата предусматривает применение метода `Task.FromResult`, который возвращает уже сигнализированную задачу:

```
Task<string> Foo() { return Task.FromResult ("abc"); }
```

Если наш метод `GetWebPageAsync` вызывается из потока пользовательского интерфейса, то он является неявно безопасным к потокам в смысле том, что его можно было бы вызвать несколько раз подряд (инициируя тем самым множество параллельных загрузок) без необходимости в каком-либо блокировании с целью защиты кеша. Однако если бы последовательность обращений относились к одному и тому же URI, то инициировалось бы множество избыточных загрузок, которые все в конечном итоге обновляли бы одну и ту же запись в кеше (в выигрыше окажется последняя загрузка). Хоть это и не ошибка, но более эффективно было бы сделать так, чтобы последующие обращения к тому же самому URI взамен (асинхронно) ожидали результата выполняющегося запроса.

Существует простой способ достичь указанной цели, не прибегая к блокировкам или сигнализирующими конструкциям.

Вместо кеша строк мы создаем кеш “будущего” (`Task<string>`):

```
static Dictionary<string, Task<string>> _cache =
    new Dictionary<string, Task<string>>();
Task<string> GetWebPageAsync (string uri)
{
    if (_cache.TryGetValue (uri, out var downloadTask)) return downloadTask;
    return _cache [uri] = new WebClient ().DownloadStringTaskAsync (uri);
}
```

(Обратите внимание, что мы не помечаем метод как `async`, поскольку напрямую возвращаем объект задачи, полученный в результате вызова метода класса `WebClient`.)

Теперь при повторяющихся вызовах метода `GetWebPageAsync` с тем же самым URI мы гарантируем получение одного и того же объекта `Task<string>`. (Это обеспечивает и дополнительное преимущество минимизации нагрузки на сборщик мусора.) И если задача завершена, то ожидание не требует больших затрат благодаря описанной выше оптимизации, которую предпринимает компилятор.

Мы могли бы и дальше расширять пример, чтобы сделать его безопасным к потокам без защиты со стороны контекста синхронизации, для чего необходимо блокировать все тело метода:

```
lock (_cache)
    if (_cache.TryGetValue (uri, out var downloadTask))
        return downloadTask;
    else
        return _cache [uri] = new WebClient ().DownloadStringTaskAsync (uri);
}
```

Код работает из-за того, что мы производим блокировку не на время загрузки страницы (это нанесло бы ущерб параллелизму), а в течение небольшого промежутка времени, пока проверяется кеш и при необходимости запускается новая задача, которая обновляет кеш.

ValueTask<T>



Тип `ValueTask<T>` предназначен для сценариев микрооптимизации и вам, возможно, никогда не придется писать методы, которые возвращают этот тип. Тем не менее, все равно нужно знать об изложенных в следующем разделе мерах предосторожности, поскольку некоторые методы .NET возвращают `ValueTask<T>`, а `IAsyncEnumerable<T>` тоже его использует.

Мы только что описали, каким образом компилятор оптимизирует выражение `await` для синхронно завершаемой задачи — путем замыкания накоротко продолжения и немедленного перехода к следующему оператору. Если синхронное завершение происходит из-за кеширования, то мы выяснили, что кеширование самой задачи может дать элегантное и эффективное решение.

Однако кешировать задачу во всех сценариях синхронного завершения нецелесообразно. Иногда должна создаваться новая задача, что порождает (кро-

шечную) потенциальную неэффективность. Дело в том, что Task и Task<T> являются ссылочными типами, а потому создание их экземпляров требует выделения памяти в куче и последующей сборки мусора. Экстремальная форма оптимизации предусматривает написание кода без выделения памяти; другими словами, код не создает экземпляры каких-либо ссылочных типов, не увеличивая нагрузку на сборщик мусора. Для поддержки такого шаблона были введены структуры ValueTask и ValueTask<T>, которые компилятор разрешает помещать вместо Task и Task<T>:

```
async ValueTask<int> Foo() { ... }
```

Ожидание ValueTask<T> свободно от выделения памяти, если операция завершается синхронно:

```
int answer = await Foo(); // (Потенциально) свободно от выделения памяти
```

Если операция не завершается синхронно, тогда ValueTask<T> создает “за кулисами” обычновенный экземпляр Task<T> (которому передает ожидание) и никакого преимущества не достигается.

Экземпляр ValueTask<T> можно преобразовать в обычновенный экземпляр Task<T>, вызвав метод AsTask.

Кроме того, существует необобщенная версия — ValueTask, которая похожа на Task.

Меры предосторожности при использовании ValueTask<T>

Тип ValueTask<T> относительно необычен в том, что он определен как структура исключительно по соображениям производительности. Таким образом, он обременен неподходящей семантикой типа значения, что может приводить к неожиданностям. Чтобы избежать некорректного поведения, вы должны не допускать следующие действия:

- организовывать ожидание одного и того же экземпляра ValueTask<T> много раз;
- вызывать .GetAwaiter().GetResult(), когда операция не завершена.

Если вам необходимо выполнить указанные действия, тогда вызовите .AsTask() и работайте с результирующим экземпляром Task.



Избежать описанных ловушек проще всего, применяя await напрямую к вызову метода, например:

```
await Foo(); // Безопасно
```

Дверь к ошибочному поведению открывается, когда вы присваиваете значение задачи ValueTask какой-то переменной:

```
ValueTask<int> valueTask = Foo(); // Осторожно!  
// Использование переменной valueTask теперь может приводить к ошибкам  
что можно смягчить, преобразуя непосредственно в обычновенную  
задачу:
```

```
Task<int> task = Foo().AsTask(); // Безопасно  
// С переменной task работать безопасно.
```

Избегание чрезмерных возвратов

В методах, которые многократно вызываются в цикле, можно избежать накладных расходов, связанных с повторяющимся возвратом в цикл сообщений пользовательского интерфейса, за счет вызова метода `ConfigureAwait`. В результате задача не передает признаки продолжения контексту синхронизации, сокращая накладные расходы до затрат на переключение контекста (или намного меньших, если метод, для которого осуществляется ожидание, завершается синхронно):

```
async void A() { ... await B(); ... }

async Task B()
{
    for (int i = 0; i < 1000; i++)
        await C().ConfigureAwait (false);
}

async Task C() { ... }
```

Это означает, что для методов `B` и `C` мы аннулируем простую модель безопасности к потокам в приложениях с пользовательским интерфейсом, согласно которой код выполняется в потоке пользовательского интерфейса и может быть вытеснен только во время выполнения оператора `await`. Однако метод `A` не затрагивается и останется в потоке пользовательского интерфейса, если он в нем был запущен.

Такая оптимизация особенно уместна при написании библиотек: преимущество упрощенной безопасности потоков не требуется, потому что код библиотеки обычно не использует совместно состояние с вызывающим кодом и не обращается к элементам управления пользовательского интерфейса. (В приведенном выше примере также имеет смысл реализовать синхронное завершение метода `C`, если известно, что операция, скорее всего, окажется кратковременной.)

Асинхронные шаблоны

Отмена

Часто важно иметь возможность отмены параллельной операции после ее запуска, скажем, в ответ на пользовательский запрос. Реализовать это проще всего с помощью флага отмены, который можно было бы инкапсулировать в классе следующего вида:

```
class CancellationToken
{
    public bool IsCancellationRequested { get; private set; }
    public void Cancel() { IsCancellationRequested = true; }
    public void ThrowIfCancellationRequested()
    {
        if (IsCancellationRequested)
            throw new OperationCanceledException();
    }
}
```

Затем можно было бы написать асинхронный метод с возможностью отмены:

```
async Task Foo (CancellationToken cancellationToken)
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine (i);
        await Task.Delay (1000);
        cancellationToken.ThrowIfCancellationRequested();
    }
}
```

Когда вызывающий код желает отменить операцию, он обращается к методу `Cancel` признака отмены, который передается в метод `Foo`. В результате `IsCancellationRequested` устанавливается в `true`, что через короткий промежуток времени приводит к отказу метода `Foo` с генерацией исключения `OperationCanceledException` (предопределенный в пространстве имен `System` класс, который предназначен для данной цели).

Если оставить в стороне безопасность к потокам (мы должны блокировать чтение/запись в `IsCancellationRequested`), то такой шаблон вполне эффективен, и среда CLR предлагает тип по имени `CancellationToken`, который очень похож на только что рассмотренный тип. Тем не менее, в нем отсутствует метод `Cancel`; этот метод открыт в другом типе — `CancellationTokenSource`. Подобное разделение обеспечивает определенную безопасность: метод, который имеет доступ только к объекту `CancellationToken`, может проверять, но не инициировать отмену.

Чтобы получить признак отмены, сначала необходимо создать экземпляр `CancellationTokenSource`:

```
var cancelSource = new CancellationTokenSource();
```

После этого станет доступным свойство `Token`, которое возвращает объект `CancellationToken`. В итоге вызвать наш метод `Foo` можно было бы следующим образом:

```
var cancelSource = new CancellationTokenSource();
Task foo = Foo (cancelSource.Token);
...
... (в какой-то момент позже)
cancelSource.Cancel();
```

Признак отмены поддерживает большинство асинхронных методов в CLR, включая `Delay`. Если модифицировать метод `Foo` так, чтобы он передавал свой признак отмены методу `Delay`, то задача будет завершаться немедленно по запросу (а не секунду спустя):

```
async Task Foo (CancellationToken cancellationToken)
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine (i);
        await Task.Delay (1000, cancellationToken);
    }
}
```

Обратите внимание, что нам больше не понадобится вызывать метод `ThrowIfCancellationRequested`, поскольку это делает `Task.Delay`. Признаки отмены нормально распространяются вниз по стеку вызовов (так же как запросы отмены каскадным образом продвигаются *вверх* по стеку вызовов посредством исключений).



UWP полагается на типы WinRT, чьи асинхронные методы при отмене следуют низкоуровневому протоколу, согласно которому вместо принятия `CancellationToken` тип `IAsyncInfo` открывает доступ к методу `Cancel`. Однако метод `AsTaskExtension` перегружен для приема признака отмены, ликвидируя данный разрыв.

Синхронные методы также могут поддерживать отмену (как делает метод `Wait` класса `Task`). В таких случаях инструкция для отмены должна будет поступать асинхронно (скажем, из другой задачи). Например:

```
var cancelSource = new CancellationTokenSource();
Task.Delay (5000).ContinueWith (ant => cancelSource.Cancel());
...
```

В действительности при конструировании `CancellationTokenSource` можно указывать временной интервал, чтобы инициировать отмену по его прошествии (как только что было продемонстрировано). Прием удобен для реализации тайм-аутов, как синхронных, так и асинхронных:

```
var cancelSource = new CancellationTokenSource (5000);
try { await Foo (cancelSource.Token); }
catch (OperationCanceledException ex) { Console.WriteLine ("Cancelled"); }
```

Структура `CancellationToken` предоставляет метод `Register`, позволяющий зарегистрировать делегат обратного вызова, который будет запущен при отмене; он возвращает объект, который можно освободить с целью отмены регистрации.

Задачи, генерируемые асинхронными функциями компилятора, автоматически входят в состояние отмены при появлении необработанного исключения `OperationCanceledException` (свойство `IsCanceled` возвращает `true`, а свойство `IsFaulted` — `false`). То же самое происходит и в случае задач, созданных с помощью метода `Task.Run`, конструктору которых передается (тот же признак) `CancellationToken`. Отличие между отказавшей и отмененной задачей в асинхронных сценариях не является важным, т.к. обе они генерируют исключение `OperationCanceledException` во время ожидания; это играет роль в расширенных сценариях параллельного программирования (особенно при условном продолжении). Мы обсудим данную тему в разделе “Отмена задач” главы 22.

Сообщение о ходе работ

Временами желательно, чтобы асинхронная операция во время выполнения сообщала о ходе работ. Простое решение заключается в передаче асинхронному методу делегата `Action`, который запускается всякий раз, когда состояние хода работ меняется:

```

Task Foo (Action<int> onProgressPercentChanged)
{
    return Task.Run (() =>
    {
        for (int i = 0; i < 1000; i++)
        {
            if (i % 10 == 0) onProgressPercentChanged (i / 10);
            // Делать что-нибудь, требующее интенсивных вычислений...
        }
    });
}

```

Вот как его можно вызывать:

```

Action<int> progress = i => Console.WriteLine (i + " %");
await Foo (progress);

```

Хотя такой прием нормально работает в консольном приложении, он не идеален в сценариях обогащенных клиентов, поскольку сообщает о ходе работ из рабочего потока, вызывая потенциальные проблемы с безопасностью к потокам у потребителя. (Фактически мы позволяем побочному эффекту от параллелизма “просочиться” во внешний мир, что нежелательно, т.к. в противном случае метод изолируется, если он вызван в потоке пользовательского интерфейса.)

IProgress<T> и Progress<T>

Для решения описанной выше проблемы среда CLR предлагает пару типов: интерфейс **IProgress<T>** и класс **Progress<T>**, который реализует этот интерфейс. В действительности они предназначены для того, чтобы служить оболочкой делегата, позволяя приложениям с пользовательским интерфейсом безопасно сообщать о ходе работ через контекст синхронизации.

Интерфейс **IProgress<T>** определяет только один метод:

```

public interface IProgress<in T>
{
    void Report (T value);
}

```

Использовать интерфейс **IProgress<T>** легко; наш метод почти не изменяется:

```

Task Foo (IProgress<int> onProgressPercentChanged)
{
    return Task.Run (() =>
    {
        for (int i = 0; i < 1000; i++)
        {
            if (i % 10 == 0) onProgressPercentChanged.Report (i / 10);
            // Делать что-нибудь, требующее интенсивных вычислений...
        }
    });
}

```

Класс **Progress<T>** имеет конструктор, принимающий делегат типа **Action<T>**, который помещается в оболочку:

```

var progress = new Progress<int> (i => Console.WriteLine (i + " %"));
await Foo (progress);

```

(В классе `Progress<T>` также определено событие `ProgressChanged`, на которое можно подписаться вместо передачи делегата `Action` конструктору (или в дополнение к ней).) После создания экземпляра `Progress<int>` захватывается контекст синхронизации, если он существует. Когда метод `Foo` затем обращается к `Report`, делегат вызывается через упомянутый контекст.

Асинхронные методы могут реализовать более сложное сообщение о ходе работ путем замены `int` специальным типом, открывающим доступ к набору свойств.



Если вы знакомы с библиотекой Rx, то заметите, что интерфейс `IProgress<T>` вместе с типом задачи, возвращаемым асинхронной функцией, предоставляют набор средств, который подобен такому набору, предлагаемому интерфейсом `IObserver<T>`. Отличие в том, что тип задачи может открывать доступ к “финальному” возвращаемому значению в дополнение к значениям (других типов), выдаваемым интерфейсом `IProgress<T>`.

Значения, выдаваемые `IProgress<T>`, обычно являются “одноразовыми” (скажем, процент выполненной работы или количество загруженных байтов), тогда как значения, возвращаемые методом `MoveNext` интерфейса `IObserver<T>`, обычно содержат в себе сам результат и поэтому существует веская причина для его вызова.

Асинхронные методы в WinRT также поддерживают возможность сообщения о ходе работ, хотя применяемый протокол сложнее из-за (относительно) слабой системы типов COM. Взамен приема объекта, реализующего `IProgress<T>`, асинхронные методы WinRT, которые сообщают о ходе работ, возвращают вместо `IAsyncAction` и `IAsyncOperation<TResult>` один из следующих интерфейсов:

```
IAsyncResultWithProgress<TProgress>
IAsyncOperationWithProgress<TResult, TProgress>
```

Интересно отметить, что оба интерфейса основаны на `IAsyncInfo` (не на `IAsyncResult` и `IAsyncOperation<TResult>`).

Хорошая новость в том, что расширяющий метод `AsTask` тоже перегружен, чтобы принимать `IProgress<T>` для вышеупомянутых интерфейсов, поэтому потребители .NET могут игнорировать интерфейсы COM и поступать так, как показано ниже:

```
var progress = new Progress<int> (i => Console.WriteLine (i + " %"));
CancellationToken cancelToken = ...
var task = someWinRTobject.FooAsync () .AsTask (cancelToken, progress);
```

Асинхронный шаблон, основанный на задачах

В .NET доступны сотни асинхронных методов, возвращающих задачи, к которым можно применять `await` (они относятся главным образом к вводу-выводу). Большинство таких методов (по крайней мере, частично) следуют шаблону, который называется *асинхронным шаблоном, основанным на задачах* (Task-Based

Asynchronous Pattern — ТАР), и представляет собой практическую формализацию всего того, что было описано до настоящего момента. Метод ТАР обладает следующими характеристиками:

- возвращает “горячий” (выполняющийся) экземпляр Task или Task<TResult>;
- имеет суффикс Async (за исключением специальных случаев, таких как комбинаторы задач);
- перегружен для приема признака отмены и/или IProgress<T>, если он поддерживает отмену и/или сообщение о ходе работ;
- быстро возвращает управление вызывающему коду (имеет только небольшую начальную синхронную фазу);
- не связывает поток, если является интенсивным в плане ввода-вывода.

Как видите, методы ТАР легко писать с использованием асинхронных функций C#.

Комбинаторы задач

Важным последствием наличия согласованного протокола для асинхронных функций (в соответствии с которым они возвращают объекты задач) является возможность применения и написания *комбинаторов задач* — функций, которые удобно объединяют задачи, не принимая во внимание то, что конкретно делает та или иная задача.

Среда CLR включает два комбинатора задач: Task.WhenAny и Task.WhenAll. При их описании мы будем предполагать, что определены следующие методы:

```
async Task<int> Delay1() { await Task.Delay (1000); return 1; }
async Task<int> Delay2() { await Task.Delay (2000); return 2; }
async Task<int> Delay3() { await Task.Delay (3000); return 3; }
```

WhenAny

Метод Task.WhenAny возвращает объект задачи, которая завершается при завершении любой задачи из набора. В следующем примере задача завершается через одну секунду:

```
Task<int> winningTask = await Task.WhenAny (Delay1(), Delay2(), Delay3());
Console.WriteLine ("Done");
Console.WriteLine (winningTask.Result); // 1
```

Поскольку метод Task.WhenAny сам возвращает объект задачи, мы применяем к его вызову await, что дает в итоге задачу, завершающуюся первой. Приведенный пример является полностью неблокирующим — включая последнюю строку, где производится доступ к свойству Result (т.к. задача winningTask уже будет завершена). Несмотря на это, обычно лучше применять await к winningTask:

```
Console.WriteLine (await winningTask); // 1
```

потому что тогда любое исключение генерируется повторно без помещения в оболочку AggregateException.

На самом деле оба `await` могут находиться в одном операторе:

```
int answer = await await Task.WhenAny (Delay1(), Delay2(), Delay3());
```

Если какая-то из задач кроме завершившейся первой впоследствии откажет, то исключение окажется необнаруженным, если только для объекта задачи не будет организовано ожидание посредством `await` (или не будет произведен доступ к его свойству `Exception`).

Метод `WhenAny` удобен для применения тайм-аутов или отмены к операциям, которые иначе подобное не поддерживают:

```
Task<string> task = SomeAsyncFunc();
Task winner = await (Task.WhenAny (task, Task.Delay(5000)));
if (winner != task) throw new TimeoutException();
string result = await task;    // Извлечь результат или повторно
                               // сгенерировать исключение
```

Обратите внимание, что поскольку в данном случае метод `WhenAny` вызывается с задачами разных типов, выигравшая задача возвращается как простой объект типа `Task<string>`.

WhenAll

Метод `Task.WhenAll` возвращает объект задачи, которая завершается, когда завершены *все* переданные ему задачи. В следующем примере задача завершается через три секунды (и демонстрируется шаблон *ветвления/присоединения* (`fork/join`)):

```
await Task.WhenAll (Delay1(), Delay2(), Delay3());
```

Похожий результат можно было бы получить без использования `WhenAll`, организовав ожидание `task1`, `task2` и `task3` по очереди:

```
Task task1 = Delay1(); task2 = Delay2(); task3 = Delay3();
await task1; await task2; await task3;
```

Отличие такого подхода (помимо меньшей эффективности из-за требования трех ожиданий вместо одного) связано с тем, что в случае отказа `task1` мы никогда не перейдем к ожиданию задач `task2/task3`, и любые их исключения останутся необнаруженными.

Напротив, метод `Task.WhenAll` не завершается до тех пор, пока не будут завершены все задачи — даже когда возникает отказ. При появлении нескольких отказов их исключения объединяются в экземпляр `AggregateException` задачи (именно здесь класс `AggregateException` становится действительно полезным, потому что вы должны быть заинтересованы в получении всех исключений). Тем не менее, ожидание комбинированной задачи обеспечивает генерацию только первого исключения, так что для просмотра всех исключений понадобится поступить следующим образом:

```
Task task1 = Task.Run (() => { throw null; });
Task task2 = Task.Run (() => { throw null; });
Task all = Task.WhenAll (task1, task2);
try { await all; }
```

```
catch
{
    Console.WriteLine (all.Exception.InnerExceptions.Count); // 2
}
```

Вызов `WhenAll` с задачами типа `Task<TResult>` возвращает `Task<TResult[]>`, предоставляя объединенные результаты всех задач. При ожидании все сводится к `TResult[]`:

```
Task<int> task1 = Task.Run (() => 1);
Task<int> task2 = Task.Run (() => 2);
int[] results = await Task.WhenAll (task1, task2); // { 1, 2 }
```

В качестве практического примера рассмотрим параллельную загрузку веб-страниц по нескольким URI с подсчетом их суммарной длины:

```
async Task<int> GetTotalSize (string[] uris)
{
    IEnumerable<Task<byte[]>> downloadTasks = uris.Select (uri =>
        new WebClient().DownloadDataTaskAsync (uri));
    byte[][] contents = await Task.WhenAll (downloadTasks);
    return contents.Sum (c => c.Length);
}
```

Однако здесь присутствует некоторая неэффективность, связанная с тем, что во время загрузки мы излишне удерживаем байтовый массив до тех пор, пока не будет завершена каждая задача. Было бы более эффективно сразу же после загрузки сворачивать байтовые массивы в их длины. Для этого очень удобно применять асинхронные лямбда-выражения, потому что нам необходимо передавать выражение `await` в операцию `Select` из LINQ:

```
async Task<int> GetTotalSize (string[] uris)
{
    IEnumerable<Task<int>> downloadTasks = uris.Select (async uri =>
        await new WebClient().DownloadDataTaskAsync (uri)).Length;
    int[] contentLengths = await Task.WhenAll (downloadTasks);
    return contentLengths.Sum();
}
```

Специальные комбинаторы

Временами удобно создавать собственные комбинаторы задач. Простейший “комбинатор” принимает одиночную задачу вроде приведенной ниже, что позволяет организовать ожидание любой задачи с использованием тайм-аута:

```
async static Task<TResult> WithTimeout<TResult> (this Task<TResult> task,
                                                       TimeSpan timeout)
{
    Task winner = await Task.WhenAny (task, Task.Delay (timeout))
        .ConfigureAwait (false);
    if (winner != task) throw new TimeoutException();
    return await task.ConfigureAwait (false); // Извлечь результат или
                                                // повторно генерировать исключение
}
```

Поскольку это в значительной степени “библиотечный метод”, который не имеет доступа к внешнему совместно используемому состоянию, при ожидании мы используем `ConfigureAwait(false)`, чтобы избежать потенциального возврата в контекст синхронизации пользовательского интерфейса. Мы можем еще больше повысить эффективность, отменяя `Task.Delay`, когда задача завершается вовремя (что позволяет устраниТЬ небольшие накладные расходы, связанные с таймером):

```
async static Task<TResult> WithTimeout<TResult> (this Task<TResult> task,
                                                    TimeSpan timeout)
{
    var cancelSource = new CancellationSource();
    var delay = Task.Delay (timeout, cancelSource.Token);
    Task winner = await Task.WhenAny (task, delay).ConfigureAwait (false);
    if (winner == task)
        cancelSource.Cancel();
    else
        throw new TimeoutException();
    return await task.ConfigureAwait (false); // Извлечь результат или
                                                // повторно сгенерировать исключение
}
```

Следующий комбинатор позволяет “отменить” задачу посредством `CancellationToken`:

```
static Task<TResult> WithCancellation<TResult> (this Task<TResult> task,
                                                    CancellationToken cancellationToken)
{
    var tcs = new TaskCompletionSource<TResult>();
    var reg = cancellationToken.Register (() => tcs.TrySetCanceled ());
    task.ContinueWith (ant =>
    {
        reg.Dispose();
        if (ant.IsCanceled)
            tcs.TrySetCanceled();
        else if (ant.IsFaulted)
            tcs.TrySetException (ant.Exception.InnerException);
        else
            tcs.TrySetResult (ant.Result);
    });
    return tcs.Task;
}
```

Комбинаторы задач могут оказаться сложными в написании, иногда требуя применения сигнализирующих конструкций, которые будут раскрыты в главе 21. На самом деле это хорошо, т.к. способствует вынесению сложности, связанной с параллелизмом, за пределы бизнес-логики и ее помещению в многократно используемые методы, которые могут быть протестированы в изоляции.

Следующий комбинатор работает подобно `WhenAll` за исключением того, что если любая из задач отказывает, то результирующая задача откажет незамедлительно:

```

async Task<TResult[]> WhenAllOrError<TResult>
    (params Task<TResult>[] tasks)
{
    var killJoy = new TaskCompletionSource<TResult[]>();
    foreach (var task in tasks)
        task.ContinueWith (ant =>
    {
        if (ant.IsCanceled)
            killJoy.TrySetCanceled();
        else if (ant.IsFaulted)
            killJoy.TrySetException (ant.Exception.InnerException);
    });
    return await await Task.WhenAny (killJoy.Task, Task.WhenAll (tasks))
        .ConfigureAwait (false);
}

```

Мы начинаем с создания экземпляра `TaskCompletionSource`, единственной работой которого является завершение всего в случае, если какая-то задача отказывает. Таким образом, мы никогда не вызываем его метод `SetResult`, а только методы `TrySetCanceled` и `TrySetException`. В данном случае метод `ContinueWith` более удобен, чем `GetAwaiter().OnCompleted`, потому что мы не обращаемся к результатам задач и в этой точке не хотим возврата в поток пользовательского интерфейса.

Асинхронное блокирование

В разделе “Асинхронные семафоры и блокировки” главы 21 будет описано, как использовать класс `SemaphoreSlim` для блокирования или ограничения параллелизма асинхронным образом.

Устаревшие шаблоны

В .NET задействованы и другие шаблоны асинхронности, которые применялись до появления задач и асинхронных функций. Теперь они редко востребованы, поскольку асинхронность на основе задач стала доминирующим шаблоном.

Модель асинхронного программирования

Самый старый шаблон назывался *моделью асинхронного программирования* (*Asynchronous Programming Model* — APM) и использовал пару методов, имена которых начинаются с `Begin` и `End`, а также интерфейс по имени `IAsyncResult`. В целях иллюстрации мы возьмем класс `Stream` из пространства имен `System.IO` и рассмотрим его метод `Read`. Вначале взглянем на синхронную версию:

```
public int Read (byte[] buffer, int offset, int size);
```

Вероятно, вы уже в состоянии предугадать, каким образом выглядит асинхронная версия на основе задач:

```
public Task<int> ReadAsync (byte[] buffer, int offset, int size);
```

Теперь давайте посмотрим на версию APM:

```
public IAsyncResult BeginRead (byte[] buffer, int offset, int size,
                               AsyncCallback callback, object state);
public int EndRead (IAsyncResult asyncResult);
```

Вызов метода BeginXXX инициирует операцию, возвращая объект IAsyncResult, который действует в качестве признака для асинхронной операции. Когда операция завершается (или отказывает), запускается делегат AsyncCallback:

```
public delegate void AsyncCallback (IAsyncResult ar);
```

Компонент, поддерживающий этот делегат, затем вызывает метод EndXXX, который предоставляет возвращаемое значение операции, а также повторно генерирует исключение, если операция потерпела неудачу.

Шаблон APM не только неудобен в применении, но также неожиданно сложен в плане корректной реализации. Проще всего иметь дело с методами APM, вызывая метод адаптера Task.Factory.FromAsync, который преобразует пару методов APM в объект Task. Внутренне он использует TaskCompletionSource, чтобы предоставить объект задачи, которой отправляется сигнал, когда операция APM завершается или отказывает.

Метод FromAsync требует передачи следующих параметров:

- делегат, указывающий метод BeginXXX;
- делегат, указывающий метод EndXXX;
- дополнительные аргументы, которые будут передаваться данным методам.

Метод FromAsync перегружен для приема типов делегатов и аргументов, которые соответствуют практически всем сигнатурам асинхронных методов, определенным в .NET. Например, исходя из предположения, что stream имеет тип Stream, а buffer — тип byte[], мы можем записать так:

```
Task<int> readChunk = Task<int>.Factory.FromAsync (
    stream.BeginRead, stream.EndRead, buffer, 0, 1000, null);
```

Асинхронный шаблон на основе событий

Асинхронный шаблон на основе событий (Event-Based Asynchronous Pattern — EAP) появился в 2005 году с целью предоставления более простой альтернативы шаблону APM, особенно в сценариях с пользовательским интерфейсом. Тем не менее, он был реализован лишь в небольшом количестве типов, наиболее примечательным из которых является WebClient в пространстве имен System.Net. Следует отметить, что EAP — это просто шаблон; никаких специальных типов для его поддержки не предусмотрено. По существу шаблон предусматривает то, что класс предлагает семейство членов, которые внутренне управляют параллелизмом, примерно как в показанном далее коде.

```
// Это члены класса WebClient:  
public byte[] DownloadData (Uri address);           // Синхронная версия  
public void DownloadDataAsync (Uri address);
```

```
public void DownloadDataAsync (Uri address, object userToken);
public event DownloadDataCompletedEventHandler DownloadDataCompleted;
public void CancelAsync (object userState); // Отменяет операцию
public bool IsBusy { get; } // Указывает, выполняется ли операция
```

Методы *Async инициируют выполнение операции асинхронным образом. Когда операция завершается, генерируется событие *Completed (с автоматической отправкой захваченному контексту синхронизации, если он имеется). Такое событие передает объект аргументов события, содержащий перечисленные ниже элементы:

- флаг, который указывает, была ли операция отменена (за счет вызова потребителем метода CancelAsync);
- объект Error, указывающий исключение, которое было сгенерировано (если было);
- объект userToken, если он предоставлялся при вызове метода *Async.

Типы EAP могут также определять событие сообщения о ходе работ, которое инициируется всякий раз, когда состояние хода работ изменяется (и вдобавок отправляется в контекст синхронизации):

```
public event DownloadProgressChangedEventHandler DownloadProgressChanged;
```

Реализация шаблона EAP требует написания большого объема стереотипного кода, делая этот шаблон неудобным с композиционной точки зрения.

BackgroundWorker

Универсальной реализацией шаблона EAP является класс BackgroundWorker из пространства имен System.ComponentModel. Он позволяет обогащенным клиентским приложениям запускать рабочий поток и сообщать о процентае выполненной работы без необходимости в явном захвате контекста синхронизации. Вот пример:

```
var worker = new BackgroundWorker { WorkerSupportsCancellation = true };
worker.DoWork += (sender, args) =>
{ // Выполняется в рабочем потоке
    if (args.Cancel) return;
    Thread.Sleep(1000);
    args.Result = 123;
};
worker.RunWorkerCompleted += (sender, args) =>
{ // Выполняется в потоке пользовательского интерфейса
    // Здесь можно безопасно обновлять элементы управления
    // пользовательского интерфейса...
    if (args.Cancelled)
        Console.WriteLine ("Cancelled"); // Отменено
    else if (args.Error != null)
        Console.WriteLine ("Error: " + args.Error.Message); // Ошибка
    else
        Console.WriteLine ("Result is: " + args.Result); // Результат
};
worker.RunWorkerAsync(); // Захватывает контекст синхронизации
// и запускает операцию
```

Метод RunWorkerAsync запускает операцию, инициируя событие DoWork в рабочем потоке из пула. Он также захватывает контекст синхронизации, и когда операция завершается (или отказывает), через данный контекст генерируется событие RunWorkerCompleted (подобно признаку продолжения).

Класс BackgroundWorker порождает крупномодульный параллелизм, при котором событие DoWork инициируется полностью в рабочем потоке. Если в этом обработчике событий нужно обновлять элементы управления пользовательского интерфейса (помимо отправки сообщения о проценте выполненных работ), тогда придется использовать Dispatcher.BeginInvoke или похожий метод.

Класс BackgroundWorker более подробно описан в статье по ссылке www.albahari.com/threading.



Потоки данных и ввод-вывод

В настоящей главе описаны фундаментальные типы, предназначенные для ввода и вывода в .NET, с акцентированием внимания на следующих темах:

- потоковая архитектура .NET и предоставление ею согласованного программного интерфейса для чтения и записи с применением разнообразных типов ввода-вывода;
- классы для манипулирования файлами и каталогами на диске;
- специализированные потоки для сжатия, именованные каналы и размещенные в памяти файлы.

Внимание в главе сконцентрировано на типах из пространства имен System.IO, где реализована функциональность ввода-вывода самого низкого уровня.

Потоковая архитектура

Потоковая архитектура .NET основана на трех концепциях: опорные хранилища, декораторы и адаптеры (рис. 15.1).

Опорное хранилище представляет собой конечную точку, которая делает ввод и вывод полезными, скажем, файл или сетевое подключение. Точнее, это один или оба следующих компонента:

- источник, с которого могут последовательно читаться байты;
- приемник, куда байты могут последовательно записываться.

Тем не менее, опорное хранилище не может использоваться, если программисту не открыт доступ к нему. Стандартным классом .NET, который предназначен для такой цели, является Stream; он предоставляет стандартный набор методов, позволяющих выполнять чтение, запись и позиционирование.

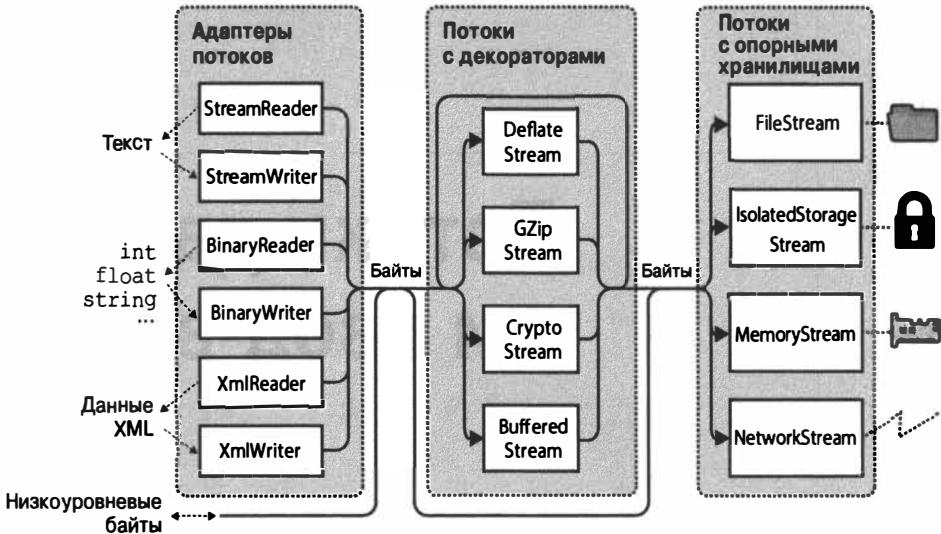


Рис. 15.1. Потоковая архитектура

Опорное хранилище представляет собой конечную точку, которая делает ввод и вывод полезными, скажем, файл или сетевое подключение. Точнее, это один или оба следующих компонента:

- источник, с которого могут последовательно читаться байты;
- приемник, куда байты могут последовательно записываться.

Тем не менее, опорное хранилище не может использоваться, если программисту не открыт доступ к нему. Стандартным классом .NET, который предназначен для такой цели, является `Stream`; он предоставляет стандартный набор методов, позволяющих выполнять чтение, запись и позиционирование. В отличие от массива, где все опорные данные существуют в памяти одновременно, поток имеет дело с данными последовательно — либо по одному байту за раз, либо в блоках управляемого размера. Следовательно, поток может потреблять мало памяти независимо от размера его опорного хранилища.

Потоки делятся на две категории.

- Потоки с опорными хранилищами.** Потоки, которые жестко привязаны к определенному типу опорного хранилища, такие как `FileStream` или `NetworkStream`.
- Потоки с декораторами.** Потоки, которые наполняют другие потоки, каким-то образом трансформируя данные, например, `DeflateStream` или `CryptoStream`.

Потоки с декораторами обладают перечисленными ниже архитектурными преимуществами:

- они освобождают потоки с опорными хранилищами от необходимости самостоятельной реализации таких возможностей, как сжатие и шифрование;
- потоки не страдают от изменения интерфейса, когда они декорированы;
- декораторы можно подключать во время выполнения;

- декораторы можно соединять в цепочки (скажем, декоратор сжатия можно соединить с декоратором шифрования).

Потоки с опорными хранилищами и потоки с декораторами имеют дело исключительно с байтами. Хотя это гибко и эффективно, приложения часто работают на более высоких уровнях, таких как текст или XML. Адаптеры преодолевают такой разрыв, помещая поток в оболочку класса со специализированными методами, которые типизированы для конкретного формата. Например, средство чтения текста открывает доступ к методу `ReadLine`, а средство записи XML — к методу `WriteAttributes`.



Адаптер помещает поток внутрь оболочки в точности как декоратор. Однако в отличие от декоратора адаптер *сам по себе* не является потоком; он обычно полностью скрывает байт-ориентированные методы.

Подведем итоги: потоки с опорными хранилищами предоставляют низкоуровневые данные; потоки с декораторами обеспечивают прозрачные двоичные трансформации вроде шифрования; адаптеры предлагают типизированные методы для работы с типами более высокого уровня, такими как строки и XML. Связи между ними были проиллюстрированы на рис. 15.1. Чтобы сформировать цепочку, необходимо просто передать один объект конструктору другого класса.

Использование потоков

Абстрактный класс `Stream` является базовым для всех потоков. В нем определены методы и свойства для трех фундаментальных операций: *чтение*, *запись* и *поиск*, а также для выполнения административных задач, подобных закрытию, сбросыванию и конфигурированию тайм-аутов (табл. 15.1).

Таблица 15.1. Члены класса Stream

Категория	Члены
Чтение	<code>public abstract bool CanRead { get; }</code> <code>public abstract int Read (byte[] buffer, int offset, int count)</code> <code>public virtual int ReadByte();</code>
Запись	<code>public abstract bool CanWrite { get; }</code> <code>public abstract void Write (byte[] buffer, int offset, int count);</code> <code>public virtual void WriteByte (byte value);</code>
Поиск	<code>public abstract bool CanSeek { get; }</code> <code>public abstract long Position { get; set; }</code> <code>public abstract void SetLength (long value);</code> <code>public abstract long Length { get; }</code> <code>public abstract long Seek (long offset, SeekOrigin origin);</code>
Закрытие/ сбросывание	<code>public virtual void Close();</code> <code>public void Dispose();</code> <code>public abstract void Flush();</code>
Тайм-ауты	<code>public virtual bool CanTimeout { get; }</code> <code>public virtual int ReadTimeout { get; set; }</code> <code>public virtual int WriteTimeout { get; set; }</code>
Другие	<code>public static readonly Stream Null; // Поток null</code> <code>public static Stream Synchronized (Stream stream);</code>

Доступны также асинхронные версии методов Read и Write, возвращающие объекты Task и дополнительно принимающие признак отмены, плюс перегруженные версии, работающие с типами Span<T> и Memory<T>, которые описаны в главе 23.

В следующем примере демонстрируется применение файлового потока для чтения, записи и позиционирования:

```
using System;
using System.IO;

// Создать файл по имени test.txt в текущем каталоге:
using (Stream s = new FileStream ("test.txt", FileMode.Create))
{
    Console.WriteLine (s.CanRead);           // True
    Console.WriteLine (s.CanWrite);          // True
    Console.WriteLine (s.CanSeek);           // True

    s.WriteByte (101);
    s.WriteByte (102);
    byte[] block = { 1, 2, 3, 4, 5 };
    s.Write (block, 0, block.Length);        // Записать блок из 5 байтов
    Console.WriteLine (s.Length);            // 7
    Console.WriteLine (s.Position);          // 7
    s.Position = 0;                         // Переместиться обратно в начало

    Console.WriteLine (s.ReadByte());         // 101
    Console.WriteLine (s.ReadByte());         // 102

    // Читать из потока в массив block:
    Console.WriteLine (s.Read (block, 0, block.Length));           // 5
    // Предполагая, что последний вызов Read возвратил 5,
    // мы находимся в конце файла, и Read теперь возвратит 0:
    Console.WriteLine (s.Read (block, 0, block.Length));           // 0
}
```

Асинхронное чтение или запись предусматривает просто вызов метода ReadAsync/WriteAsync вместо Read/Write и применение к выражению ключевого слова await (как объяснялось в главе 14, к вызываемому методу потребуется также добавить ключевое слово async):

```
async static void AsyncDemo()
{
    using (Stream s = new FileStream ("test.txt", FileMode.Create))
    {
        byte[] block = { 1, 2, 3, 4, 5 };
        await s.WriteAsync (block, 0, block.Length); // Выполнить запись
                                                    // асинхронно
        s.Position = 0;                          // Переместиться обратно в начало

        // Читать из потока в массив block:
        Console.WriteLine (await s.ReadAsync (block, 0, block.Length)); // 5
    }
}
```

Асинхронные методы упрощают построение отзывчивых и масштабируемых приложений, которые работают с потенциально медленными потоками данных (особенно сетевыми потоками), не связывая поток управления.



Для краткости мы будем использовать синхронные методы почти во всех примерах настоящей главы; тем не менее, в большинстве сценариев, связанных с сетевым вводом-выводом, мы рекомендуем отдавать предпочтение асинхронным операциям Read/Write.

Чтение и запись

Поток может поддерживать чтение, запись, а также то и другое. Если свойство `CanWrite` возвращает `false`, тогда поток предназначен только для чтения; если свойство `CanRead` возвращает `false`, то поток предназначен только для записи.

Метод `Read` получает блок данных из потока и помещает его в массив. Он возвращает количество полученных байтов, которое всегда либо меньше, либо равно значению аргумента `count`. Если оно меньше `count`, то это означает, что достигнут конец потока или поток выдает данные порциями меньшего размера (как часто случается с сетевыми потоками). В любом случае остаток байтов в массиве останется неизменным, сохраняя предыдущие значения.



При работе с методом `Read` можно определенно утверждать, что достигнут конец потока, только когда он возвращает 0. Таким образом, если есть поток из 1000 байтов, тогда следующий код может не прочитать их все в память:

```
// Предполагается, что s является потоком:  
byte[] data = new byte [1000];  
s.Read (data, 0, data.Length);
```

Метод `Read` мог бы прочитать от 1 до 1000 байтов, оставив остаток потока непрочитанным.

Вот корректный способ чтения потока из 1000 байтов:

```
byte[] data = new byte [1000];  
  
// Переменная bytesRead в итоге получит значение 1000,  
// если только сам поток не имеет меньшую длину:  
  
int bytesRead = 0;  
int chunkSize = 1;  
while (bytesRead < data.Length && chunkSize > 0)  
    bytesRead +=  
        chunkSize = s.Read (data, bytesRead, data.Length - bytesRead);
```

Чтобы упростить решение этой задачи, в .NET 7 класс `Stream` включает вспомогательные методы `ReadExactly` и `ReadAtLeast` (а также асинхронные версии каждого из них). Следующий код читает в точности 1000 байтов из потока (генерируя исключение, если поток заканчивается раньше):

```
byte[] data = new byte [1000];  
s.ReadExactly (data); // Прочитать в точности 1000 байтов
```

Последняя строка эквивалентна следующей строке:

```
s.ReadExactly (data, offset:0, count:1000);
```



Тип `BinaryReader` предлагает более простой способ для достижения того же самого результата:

```
byte[] data = new BinaryReader (s).ReadBytes (1000);
```

Если поток имеет длину меньше 1000 байтов, то возвращенный байтовый массив отражает действительный размер потока. Если поток поддерживает поиск, тогда можно прочитать все его содержимое, заменив 1000 выражением `(int)s.Length`.

Мы более подробно опишем тип `BinaryReader` в разделе “АдAPTERЫ ПОТОКОВ” далее в главе.

Метод `ReadByte` проще: он читает одиночный байт, возвращая `-1` для указания на конец массива. На самом деле `ReadByte` возвращает значение типа `int`, а не `byte`, т.к. `-1` типом `byte` не поддерживается.

Методы `Write` и `WriteByte` отправляют данные в поток. Если они не могут отправить указанные байты, то генерируется исключение.



В методах `Read` и `Write` аргумент `offset` ссылается на индекс в массиве `buffer`, с которого начинается чтение или запись, а не на позицию внутри потока.

Поиск

Поток поддерживает возможность позиционирования, если свойство `CanSeek` возвращает `true`. Для потока с возможностью позиционирования (такого как файловый поток) можно запрашивать или модифицировать его свойство `Length` (вызывая метод `SetLength`) и в любой момент изменять свойство `Position`, отражающее позицию, в которой производится чтение или запись. Свойство `Position` принимает значения относительно начала потока; однако метод `Seek` позволяет перемещаться относительно текущей позиции либо относительно конца потока.



Изменение свойства `Position` экземпляра `FileStream` обычно занимает несколько микросекунд. Если вам нужно делать это миллионы раз в цикле, тогда класс `MemoryMappedFile` может оказаться более удачным выбором, чем `FileStream` (как показано в разделе “Размещенные в памяти файлы” далее в главе).

Единственный способ определения длины потока, не поддерживающего возможность позиционирования (вроде потока с шифрованием), заключается в его чтении до самого конца. Более того, если требуется повторно прочитать предшествующую область, то поток придется закрыть и начать работу с новым потоком.

Закрытие и сбрасывание

После применения потоки должны быть освобождены, чтобы освободить лежащие в их основе ресурсы, подобные файловым и сокетным дескрипторам. Самый простой способ предусматривает создание экземпляров потоков внутри блоков `using`. В общем случае потоки поддерживают следующую стандартную семантику освобождения:

- методы `Dispose` и `Close` функционируют идентично;
- многократное освобождение или закрытие потока не вызывает ошибки.

Закрытие потока с декоратором приводит к закрытию и декоратора, и его потока с опорным хранилищем. В случае цепочки декораторов закрытие самого внешнего декоратора (в голове цепочки) закрывает всю цепочку.

Некоторые потоки внутренне буферизируют данные, поступающие в и из опорного хранилища, чтобы снизить количество двухсторонних обменов и тем самым улучшить производительность (хорошим примером служат файловые потоки). Это значит, что данные, записываемые в поток, могут не сразу попасть в опорное хранилище; возможна задержка до тех пор, пока буфер не заполнится. Метод `Flush` обеспечивает принудительную запись любых буферизированных данных. Метод `Flush` вызывается автоматически при закрытии потока, поэтому поступать так, как показано ниже, никогда не придется:

```
s.Flush(); s.Close();
```

Тайм-ауты

Поток поддерживает тайм-ауты чтения и записи, если свойство `CanTimeout` возвращает `true`. Сетевые потоки поддерживают тайм-ауты, а файловые потоки и потоки в памяти — нет. Для потоков, поддерживающих тайм-ауты, свойства `ReadTimeout` и `WriteTimeout` задают желаемый тайм-аут в миллисекундах, причем 0 означает отсутствие тайм-аута. Методы `Read` и `Write` указывают на то, что тайм-аут произошел, генерацией исключения.

Асинхронные методы `ReadAsSync/WriteAsSync` не поддерживают тайм-ауты; взамен этим методам можно передавать признак отмены.

Безопасность в отношении потоков управления

Как правило, потоки данных не являются безопасными в отношении потоков управления, т.е. два потока управления не могут параллельно выполнять чтение или запись в один и тот же поток данных, не создавая возможность для ошибки. Класс `Stream` предлагает простой обходной путь через статический метод `Synchronized`, который принимает поток данных любого типа и возвращает оболочку, безопасную к потокам управления. Такая оболочка работает за счет получения монопольной блокировки на каждой операции чтения, записи или позиционирования, гарантируя, что в любой момент времени заданную операцию может выполнять только один поток управления. На практике это позволяет множеству потоков управления одновременно дописывать данные в один и тот же поток данных — другие разновидности действий (подобные параллельному чтению) требуют дополнительной блокировки, обеспечивающей доступ каж-

дого потока управления к желаемой части потока данных. Вопросы безопасности в отношении потоков управления подробно обсуждаются в главе 21.



Начиная с версии .NET 6, можно использовать класс RandomAccess для выполнения безопасных к потокам файловых операций ввода-вывода. Класс RandomAccess также позволяет передавать несколько буферов для улучшения показателей производительности.

Потоки с опорными хранилищами

На рис. 15.2 показаны основные потоки с опорными хранилищами, предлагаемые .NET. “Поток null” также доступен через статическое поле Null класса Stream. Потоки null могут быть удобны при написании модульных тестов.

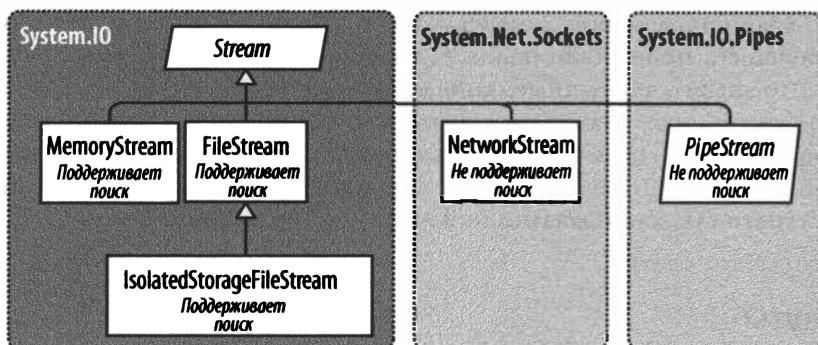


Рис. 15.2. Основные потоки с опорными хранилищами

В последующих разделах мы опишем классы FileStream и MemoryStream, а в финальном разделе настоящей главы — класс IsolatedStorageStream. Класс NetworkStream будет раскрыт в главе 16.

FileStream

Ранее в разделе мы демонстрировали базовое использование класса FileStream для чтения и записи байтов данных. Теперь мы рассмотрим специальные возможности этого класса.



Если вы все еще пользуетесь UWP, тогда файловый ввод-вывод лучше выполнять с помощью типов из пространства имен Windows.Storage (см. дополнительные материалы, доступные на веб-сайте издательства).

Конструирование экземпляра FileStream

Простейший способ создания экземпляра FileStream предполагает использование следующих статических фасадных методов класса File:

```
FileStream fs1 = File.OpenRead ("readme.bin"); // Только для чтения
FileStream fs2 = File.OpenWrite ("writeme.tmp"); // Только для записи
FileStream fs3 = File.Create ("readwrite.tmp"); // Для чтения и записи
```

Поведение методов `OpenWrite` и `Create` отличается в ситуации, когда файл уже существует. Метод `Create` усекает любое имеющееся содержимое, а метод `OpenWrite` оставляет содержимое незатронутым, устанавливая позицию потока в ноль. Если будет записано меньше байтов, чем ранее существовало в файле, то метод `OpenWrite` оставит смесь старого и нового содержимого.

Создавать экземпляры `FileStream` можно также напрямую. Конструкторы класса `FileStream` предоставляют доступ ко всем средствам, позволяя указывать имя файла или низкоуровневый файловый дескриптор, режимы создания и доступа к файлу, а также параметры для совместного использования, буфериизации и безопасности. Приведенный ниже оператор открывает существующий файл для чтения/записи, не перезаписывая его (ключевое слово `using` гарантирует освобождение, когда `fs` покидает область видимости):

```
using var fs = new FileStream ("readwrite.tmp", FileMode.Open);
```

Вскоре мы более подробно рассмотрим перечисление `FileMode`.

Сокращенные методы класса `File`

Следующие статические методы читают целый файл в память за один шаг:

- `File.ReadAllText` (возвращает строку);
- `File.ReadAllLines` (возвращает массив строк);
- `File.ReadAllBytes` (возвращает байтовый массив).

Приведенные ниже статические методы записывают целый файл за один шаг:

- `File.WriteAllText`;
- `File.WriteAllLines`;
- `File.WriteAllBytes`;
- `File.AppendAllText` (удобен для добавления данных в журнальный файл).

Есть также статический метод по имени `File.ReadLines`: он похож на `ReadAllLines` за исключением того, что возвращает лениво оцениваемое перечисление `IEnumerable<string>`. Он более эффективен, т.к. не производит загрузку всего файла в память за один раз. Для потребления результатов идеально подходит LINQ; скажем, следующий код подсчитывает количество строк с длиной, превышающей 80 символов:

```
int longLines = File.ReadLines ("filePath")
    .Count (l => l.Length > 80);
```

Указание имени файла

Имя файла может быть либо абсолютным (скажем, `c:\temp\test.txt` или `/tmp/test.txt` в Unix), либо относительным к текущему каталогу (например, `test.txt` или `temp\test.txt`). Получить доступ либо изменить текущий каталог можно через статическое свойство `Environment.CurrentDirectory`.



Когда программа запускается, текущий каталог может совпадать или не совпадать с каталогом, где находится исполняемый файл программы. По этой причине при поиске во время выполнения дополнительных файлов, упакованных с исполняемым файлом, никогда не следует полагаться на текущий каталог.

Свойство AppDomain.CurrentDomain.BaseDirectory возвращает базовый каталог приложения, которым в нормальных ситуациях является каталог, содержащий исполняемый файл программы. Чтобы указать имя файла относительно базового каталога, можно вызвать метод Path.Combine:

```
string baseFolder = AppDomain.CurrentDomain.BaseDirectory;
string logoPath = Path.Combine (baseFolder, "logo.jpg");
Console.WriteLine (File.Exists (logoPath));
```

Чтение и запись по сети можно выполнять через путь UNC (Universal Naming Convention — соглашение об универсальном назначении имен), такой как \\JoesPC\PicShare\pic.jpg или \\10.1.1.2\PicShare\pic.jpg.

(Чтобы получить доступ к общему файловому ресурсу Windows из macOS или Unix, смонтируйте его в своей файловой системе согласно инструкциям, специфичным для вашей операционной системы (ОС), и откройте с использованием обычного пути в коде C#.)

Указание режима файла

Все конструкторы класса FileStream, которые принимают имя файла, также требуют указания режима файла — аргумента типа перечисления FileMode. На рис. 15.3 показано, как выбрать значение FileMode, и варианты дают результаты сродни вызову статического метода класса File.

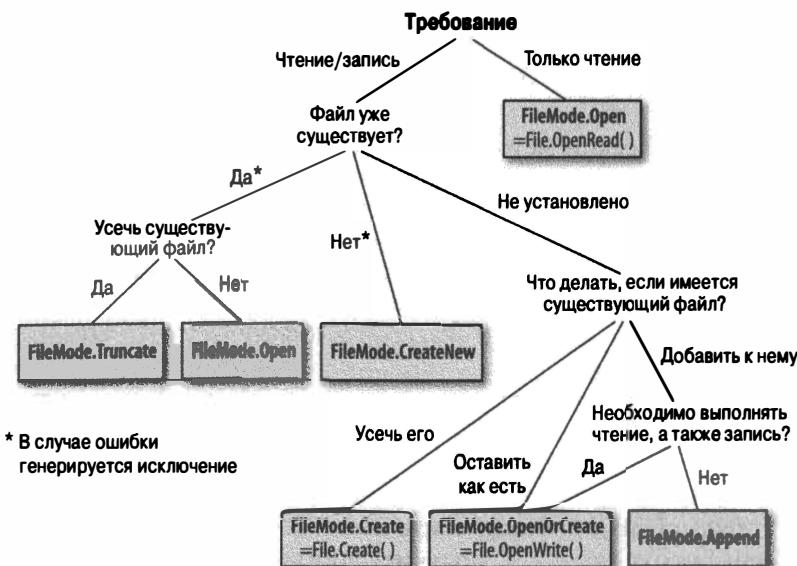


Рис. 15.3. Выбор значения FileMode



Метод `File.Create` и значение `FileMode.Create` приведут к генерации исключения, если используются для скрытых файлов. Чтобы перезаписать скрытый файл, потребуется удалить его и затем создать повторно:

```
File.Delete ("hidden.txt");
using var file = File.Create ("hidden.txt");
...
```

Конструирование экземпляра `FileStream` с указанием имени файла и режима `FileMode` дает (с одним исключением) поток с возможностью чтения/записи. Можно запросить понижение уровня доступа, если также предоставить аргумент `FileAccess`:

```
[Flags]
public enum FileAccess { Read = 1, Write = 2, ReadWrite = 3 }
```

Следующий вызов возвращает поток, предназначенный только для чтения, и он эквивалентен вызову метода `File.OpenRead`:

```
using var fs = new FileStream ("x.bin", FileMode.Open, FileAccess.Read);
...
```

Значение `FileMode.Append` считается особым: в таком режиме будет получен поток, предназначенный *только для записи*. Чтобы можно было добавлять, располагая поддержкой чтения-записи, вместо `FileMode.Append` придется указать `FileMode.Open` или `FileMode.OpenOrCreate` и перейти в конец потока:

```
using var fs = new FileStream ("myFile.bin", FileMode.Open);
fs.Seek (0, SeekOrigin.End);
...
```

Расширенные возможности `FileStream`

Ниже описаны другие необязательные аргументы, которые можно задавать при конструировании экземпляра `FileStream`.

- Значение перечисления `FileShare`, которое описывает, какой уровень доступа должен быть выдан другим процессам, чтобы они могли просматривать файл до того, как вы завершите с ним работу (`None`, `Read` (по умолчанию), `ReadWrite` или `Write`).
- Размер внутреннего буфера в байтах (в настоящее время стандартным является размер, составляющий 4 Кбайт).
- Флаг, который указывает, следует ли возложить асинхронный вывод на ОС.
- Значение перечисления флагов `FileOptions` для запроса шифрования ОС (`Encrypted`), автоматического удаления при закрытии временных файлов (`DeleteOnClose`) и подсказки для оптимизации (`RandomAccess` и `SequentialScan`). Имеется также флаг `WriteThrough`, который запрашивает у ОС отключение кеширования при записи; он предназначен для транзакционных файлов или журналов. Флаги, не поддерживаемые имеющейся ОС, молча игнорируются.

Открытие файла со значением `FileShare.ReadWrite` позволяет другим процессам или пользователям одновременно читать и записывать в один и тот же файл. Во избежание хаоса потребуется блокировать определенные области файла перед чтением или записью с помощью следующих методов:

```
// Определены в классе FileStream:  
public virtual void Lock (long position, long length);  
public virtual void Unlock (long position, long length);
```

Метод `Lock` генерирует исключение, если часть или вся запрошенная область файла уже заблокирована.

MemoryStream

В качестве опорного хранилища класс `MemoryStream` использует массив. Отчасти это противоречит замыслу самого потока, поскольку опорное хранилище должно располагаться в памяти целиком. Класс `MemoryStream` все еще полезен, когда необходим произвольный доступ в поток данных, не поддерживающий позиционирование. Если известно, что исходный поток будет иметь поддающийся управлению размер, то вот как его можно скопировать в `MemoryStream`:

```
var ms = new MemoryStream();  
sourceStream.CopyTo (ms);
```

Вызвав метод `ToByteArray`, поток `MemoryStream` можно преобразовать в байтовый массив. Метод `GetBuffer` делает ту же самую работу более эффективно, возвращая прямую ссылку на лежащий в основе массив хранилища; к сожалению, такой массив обычно превышает реальный размер потока.



Закрывать и сбрасывать `MemoryStream` вовсе не обязательно. После закрытия потока `MemoryStream` производить чтение и запись в него больше не удастся, но по-прежнему можно вызывать метод `ToByteArray` для получения лежащих в основе данных. Метод `Flush` в потоке `MemoryStream` вообще ничего не делает.

Дополнительные примеры использования `MemoryStream` можно найти в разделе “Потоки со сжатием” далее в главе и в разделе “Обзор” главы 20.

PipeStream

Класс `PipeStream` предоставляет простой способ взаимодействия одного процесса с другим через протокол *каналов* ОС. Различают два вида каналов.

- **Анонимный канал.** Делает возможным однонаправленное взаимодействие между родительским и дочерним процессами на одном и том же компьютере.
- **Именованный канал (более гибкий).** Делает возможным двунаправленное взаимодействие между произвольными процессами на одном и том же компьютере или на разных компьютерах по сети Windows.

Канал удобен для организации взаимодействия между процессами (inter-process communication — IPC) на одном компьютере: он не полагается на сетевой транспорт, что означает отсутствие накладных расходов, связанных с протоколами, и проблем с брандмауэрами.



Каналы основаны на потоках, так что один процесс ожидает получения последовательности байтов, в то время как другой процесс их отправляет. Альтернативой является взаимодействие процессов через блок совместно используемой памяти; мы покажем, как это делать, в разделе “Размещенные в памяти файлы” далее в главе.

Тип `PipeStream` представляет собой абстрактный класс с четырьмя конкретными подтипами. Два из них применяются для анонимных каналов и еще два — для именованных каналов.

- **Анонимные каналы.** `AnonymousPipeServerStream` и `AnonymousPipeClientStream`
- **Именованные каналы.** `NamedPipeServerStream` и `NamedPipeClientStream`

Именованные каналы проще в использовании, так что рассмотрим их первыми.

Именованные каналы

В случае именованных каналов участники взаимодействуют через канал с таким же именем. Протокол определяет две отдельные роли: клиент и сервер. Взаимодействие между клиентом и сервером происходит следующим образом.

- Сервер создает экземпляр `NamedPipeServerStream` и вызывает метод `WaitForConnection`.
- Клиент создает экземпляр `NamedPipeClientStream` и вызывает метод `Connect` (необязательно указывая тайм-аут).

Затем для взаимодействия два участника производят чтение и запись в потоки.

В приведенном ниже примере демонстрируется сервер, который отправляет одиночный байт (100) и ожидает получения одиночного байта:

```
using var s = new NamedPipeServerStream ("pipedream");
s.WaitForConnection();
s.WriteByte (100);           // Отправить значение 100
Console.WriteLine (s.ReadByte());
```

А вот соответствующий код клиента:

```
using var s = new NamedPipeClientStream ("pipedream");
s.Connect();
Console.WriteLine (s.ReadByte());
s.WriteByte (200);           // Отправить обратно значение 200
```

Потоки данных именованных каналов по умолчанию являются двунаправленными, так что любой из участников может читать или записывать в свой поток. Это значит, что клиент и сервер должны следовать определенному протоколу для координации своих действий, чтобы оба участника не начали одновременно отправлять или получать данные.

Также должно быть предусмотрено соглашение о длине каждой передачи. В данном смысле приведенный выше пример тривиален, поскольку в каждом направлении передается один байт. Для поддержки сообщений длиннее одного байта каналы предлагают режим передачи *сообщений* (только Windows). Когда он включен, вызывающий метод `Read` участник может узнать о том, что сообщение завершено, проверив свойство `IsMessageComplete`. В целях демонстрации мы начнем с написания вспомогательного метода, который читает целое сообщение из `PipeStream` с включенным режимом передачи сообщений — другими словами, до тех пор, пока свойство `IsMessageComplete` не станет равным `true`:

```
static byte[] ReadMessage (PipeStream s)
{
    MemoryStream ms = new MemoryStream();
    byte[] buffer = new byte [0x1000]; // Читать блоками по 4 Кбайт
    do { ms.Write (buffer, 0, s.Read (buffer, 0, buffer.Length)); }
    while (!s.IsMessageComplete);
    return ms.ToArray();
}
```

(Чтобы сделать код асинхронным, замените `s.Read` конструкцией `await s.ReadAsync()`.)



Просто ожидая возвращения методом `Read` значения 0, нельзя выяснить, завершил ли поток `PipeStream` чтение сообщения. Причина в том, что в отличие от большинства других типов потоков потоки данных каналов и сетевые потоки не имеют четко выраженного окончания. Взамен они временно “опустошаются” между передачами сообщений.

Теперь можно активизировать режим передачи сообщений. На стороне сервера это делается за счет указания `PipeTransmissionMode.Message` во время конструирования потока:

```
using var s = new NamedPipeServerStream ("pipedream", PipeDirection.InOut,
                                         1, PipeTransmissionMode.Message);
s.WaitForConnection();
byte[] msg = Encoding.UTF8.GetBytes ("Hello");
s.Write (msg, 0, msg.Length);
Console.WriteLine (Encoding.UTF8.GetString (ReadMessage (s)));
```

На стороне клиента режим передачи сообщений включается установкой свойства `ReadMode` после вызова метода `Connect`:

```
using var s = new NamedPipeClientStream ("pipedream");
s.Connect();
s.ReadMode = PipeTransmissionMode.Message;
Console.WriteLine (Encoding.UTF8.GetString (ReadMessage (s)));
byte[] msg = Encoding.UTF8.GetBytes ("Hello right back!");
s.Write (msg, 0, msg.Length);
```



Режим передачи сообщений поддерживается только в Windows. На других платформах будет генерироваться исключение PlatformNotSupportedException.

Анонимные каналы

Анонимный канал предоставляет односторонний поток взаимодействия между родительским и дочерним процессами. Вместо использования имени на уровне системы анонимные каналы настраиваются посредством закрытого дескриптора.

Как и именованные каналы, анонимные каналы имеют отдельные роли клиента и сервера. Однако система взаимодействия несколько отличается и происходит следующим образом.

1. Сервер создает экземпляр класса `AnonymousPipeServerStream`, фиксируя направление канала (`PipeDirection`) как `In` или `Out`.
2. Сервер вызывает метод `GetClientHandleAsString`, чтобы получить идентификатор для канала, который затем передается клиенту (обычно в качестве аргумента при запуске дочернего процесса).
3. Дочерний процесс создает экземпляр класса `AnonymousPipeClientStream`, указывая противоположное направление канала (`PipeDirection`).
4. Сервер освобождает локальный дескриптор, который был сгенерирован на шаге 2, вызывая метод `DisposeLocalCopyOfClientHandle`.
5. Родительский и дочерний процессы взаимодействуют, выполняя чтение/запись в поток.

Поскольку анонимные каналы являются односторонними, для двунаправленного взаимодействия сервер должен создать два канала. В приведенной далее консольной программе создаются два канала (ввода и вывода) и запускается дочерний процесс. Затем дочернему процессу отправляется одиничный байт и в ответ принимается тоже одиничный байт:

```
class Program
{
    static void Main (string[] args)
    {
        if (args.Length == 0)
            // Отсутствие аргументов сигнализирует о режиме сервера
            AnonymousPipeServer ();
        else
```

```

    // Для сигнализации о режиме клиента в качестве аргументов
    // мы передаем идентификаторы дескрипторов каналов
    AnonymousPipeClient (args [0], args [1]);
}

static void AnonymousPipeClient (string rxID, string txID)
{
    using var rx = new AnonymousPipeClientStream (PipeDirection.In, rxID);
    using var tx = new AnonymousPipeClientStream (PipeDirection.Out, txID);

    Console.WriteLine ("Client received: " + rx.ReadByte ());
        // Клиент получил:
    tx.WriteByte (200);
}

static void AnonymousPipeServer ()
{
    using var tx = new AnonymousPipeServerStream (
        PipeDirection.Out, HandleInheritability.Inheritable);
    using var rx = new AnonymousPipeServerStream (
        PipeDirection.In, HandleInheritability.Inheritable);

    string txID = tx.GetClientHandleAsString ();
    string rxID = rx.GetClientHandleAsString ();

    // Создать и запустить дочерний процесс.
    // Мы используем тот же самый исполняемый файл консольной программы,
    // но передаем аргументы:
    string thisAssembly = Assembly.GetEntryAssembly ().Location;
    string thisExe = Path.ChangeExtension (thisAssembly, ".exe");
    var args = $"{txID} {rxID}";
    var startInfo = new ProcessStartInfo (thisExe, args);

    startInfo.UseShellExecute = false;    // Требуется для дочернего процесса
    Process p = Process.Start (startInfo);

    tx.DisposeLocalCopyOfClientHandle ();    // Освободить неуправляемые
    rx.DisposeLocalCopyOfClientHandle ();    // ресурсы дескрипторов

    tx.WriteByte (100);        // Отправить байт дочернему процессу
    Console.WriteLine ("Server received: " + rx.ReadByte ());
        // Сервер получил:

    p.WaitForExit ();
}
}

```

Как и в случае именованных каналов, клиент и сервер должны координировать свои отправки и получения и согласовывать длину каждой передачи. К сожалению, анонимные каналы не поддерживают режим передачи сообщений, а потому вам придется реализовать собственный протокол для согласования длины сообщений. Одним из решений может быть отправка в первых четырех байтах каждой передачи целочисленного значения, которое определяет длину сообщения, следующего за этими четырьмя байтами. Класс BitConverter предоставляет методы для преобразования между целочисленным типом и массивом из четырех байтов.

BufferedStream

Класс `BufferedStream` декорирует, или помещает в оболочку, другой поток, добавляя возможность буферизации, и является одним из нескольких типов потоков с декораторами, которые определены в .NET; все типы показаны на рис. 15.4.

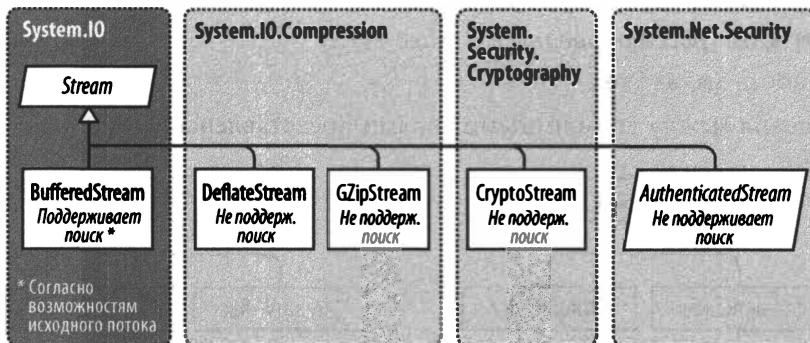


Рис. 15.4. Потоки с декораторами

Буферизация улучшает производительность, сокращая количество двухсторонних обменов с опорным хранилищем. Ниже показано, как поместить поток `FileStream` в `BufferedStream` с буфером 20 000 байтов:

```
// Записать 100 000 байтов в файл:  
File.WriteAllBytes ("myFile.bin", new byte [100000]);  
  
using FileStream fs = File.OpenRead ("myFile.bin");  
using BufferedStream bs = new BufferedStream (fs, 20000); // Буфер размером  
// 20000 байтов  
  
bs.ReadByte();  
Console.WriteLine (fs.Position); // 20000
```

В приведенном примере благодаря буферизации с опережающим чтением внутренний поток перемещает 20 000 байтов после чтения только одного байта. Вызывать метод `ReadByte` можно было бы еще 19 999 раз, и лишь тогда снова произошло бы обращение к `FileStream`.

Связывание `BufferedStream` с `FileStream`, как в предыдущем примере, не особенно ценно, т.к. класс `FileStream` сам поддерживает встроенную буферизацию. Оно могло понадобиться единственно для расширения буфера уже сконструированного потока `FileStream`.

Закрытие `BufferedStream` автоматически закрывает лежащий в основе поток с опорным хранилищем.

АдAPTERЫ ПОТОКОВ

Класс `Stream` имеет дело только с байтами; для чтения и записи таких типов данных, как строки, целые числа или XML-элементы, потребуется подключить адаптер. Ниже описаны виды адаптеров, предлагаемые .NET.

Текстовые адаптеры (для строковых и символьных данных)

TextReader, TextWriter

StreamReader, StreamWriter

StringReader, StringWriter

Двоичные адаптеры (для примитивных типов вроде int, bool, string и float)

BinaryReader, BinaryWriter

Адаптеры XML (рассматривались в главе 11)

XmlReader, XmlWriter

Отношения между упомянутыми типами представлены на рис. 15.5.

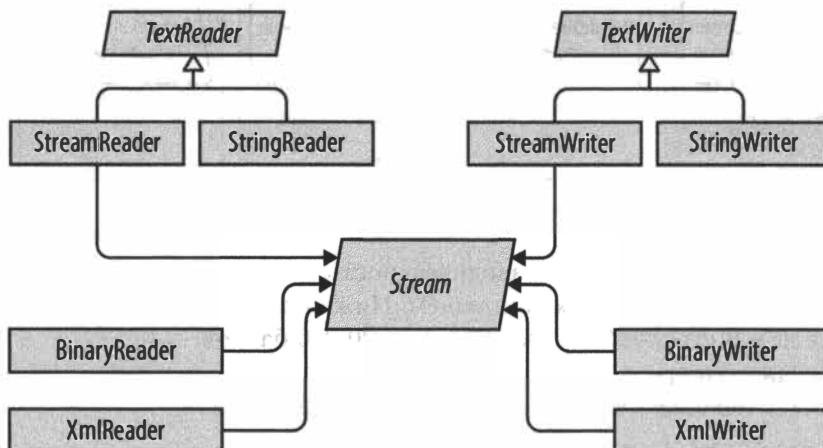


Рис. 15.5. Средства чтения и записи

Текстовые адаптеры

Типы `TextReader` и `TextWriter` являются абстрактными базовыми классами для адаптеров, которые имеют дело исключительно с символами и строками. С каждым из них в .NET связаны две универсальные реализации.

- `StreamReader/StreamWriter`. Применяют для своего хранилища низкоуровневых данных класс `Stream`, транслируя байты потока в символы или строки.
- `StringReader/StringWriter`. Реализуют `TextReader/TextWriter`, используя строки в памяти.

В табл. 15.2 перечислены члены класса `TextReader` по категориям. Метод `Peek` возвращает следующий символ из потока, не перемещая текущую позицию вперед. Метод `Peek` и версия без аргументов метода `Read` возвращают `-1`, если встречается конец потока, и целочисленное значение, которое может быть приведено непосредственно к типу `char`, в противном случае. Перегруженная версия `Read`, принимающая буфер `char[]`, идентична по функциональности методу `ReadBlock`. Метод `ReadLine` производит чтение до тех пор, пока не встретит в последовательности `<CR>` (символ 13), `<LF>` (символ 10) или пару `<CR+LF>`. Затем он возвращает строку с отброшенными символами `<CR>/<LF>`.

Таблица 15.2. Члены класса TextReader

Категория	Члены
Чтение одного символа	public virtual int Peek(); // Результат приводится к char public virtual int Read(); // Результат приводится к char
Чтение множества символов	public virtual int Read (char[] buffer, int index, int count); public virtual int ReadBlock (char[] buffer, int index, int count); public virtual string ReadLine(); public virtual string ReadToEnd();
Закрытие	public virtual void Close(); public void Dispose(); // То же, что и Close
Другие	public static readonly TextReader Null; public static TextReader Synchronized (TextReader reader);



Свойство Environment.NewLine возвращает последовательность новой строки для текущей ОС. В Windows она выглядит как "\r\n" и приближенно моделирует механическую пишущую машинку: возврат каретки (символ 13), за которым следует перевод строки (символ 10). Изменение порядка следования символов на обратный приведет к получению либо двух новых строк, либо вообще ни одной! В Unix и macOS это просто "\n".

Класс TextWriter имеет аналогичные методы для записи (табл. 15.3). Методы Write и WriteLine дополнительно перегружены, чтобы принимать каждый примитивный тип плюс тип object. Такие методы просто вызывают метод ToString на том, что им передается (возможно, через реализацию интерфейса IFormatProvider, указанную или при вызове метода, или при конструировании экземпляра TextWriter).

Таблица 15.3. Члены класса TextWriter

Категория	Члены
Запись одного символа	public virtual void Write (char value);
Запись множества символов	public virtual void Write (string value); public virtual void Write (char[] buffer, int index, int count); public virtual void Write (string format, params object[] arg); public virtual void WriteLine (string value);
Закрытие и сбрасывание	public virtual void Close(); public void Dispose(); // То же, что и Close public virtual void Flush();
Форматирование и кодирование	public virtual IFormatProvider FormatProvider { get; } public virtual string NewLine { get; set; } public abstract Encoding Encoding { get; }
Другие	public static readonly TextWriter Null; public static TextWriter Synchronized (TextWriter writer);

Метод `WriteLine` просто дополняет заданный текст значением `Environment.NewLine`, что можно изменить с помощью свойства `NewLine` (полезно для взаимодействия с файлами в форматах Unix).



Подобно `Stream` классы `TextReader` и `TextWriter` предлагают для своих методов чтения/записи асинхронные версии на основе задач.

StreamReader и StreamWriter

В следующем примере экземпляр `StreamWriter` записывает две строки текста в файл, и затем экземпляр `StreamReader` производит чтение из этого файла:

```
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
{
    writer.WriteLine ("Line1");
    writer.WriteLine ("Line2");
}

using (FileStream fs = File.OpenRead ("test.txt"))
using (TextReader reader = new StreamReader (fs))
{
    Console.WriteLine (reader.ReadLine());           // Line1
    Console.WriteLine (reader.ReadLine());           // Line2
}
```

Поскольку текстовые адAPTERы настолько часто связываются с файлами, для сокращения объема кода класс `File` предоставляет статические методы `CreateText`, `AppendText` и `OpenText`:

```
using (TextWriter writer = File.CreateText ("test.txt"))
{
    writer.WriteLine ("Line1");
    writer.WriteLine ("Line2");
}

using (TextWriter writer = File.AppendText ("test.txt"))
    writer.WriteLine ("Line3");
using (TextReader reader = File.OpenText ("test.txt"))
    while (reader.Peek() > -1)
        Console.WriteLine (reader.ReadLine());      // Line1
                                                // Line2
                                                // Line3
```

Здесь еще иллюстрируется способ осуществления проверки на предмет достижения конца файла (через `reader.Peek()`). Другой способ предполагает чтение до тех пор, пока `reader.ReadLine` не возвратит `null`.

Можно также выполнять чтение и запись других типов данных, подобных целым числам, но из-за того, что `TextWriter` вызывает на них метод `ToString`, при чтении потребуется произвести разбор строки:

```
using (TextWriter w = File.CreateText ("data.txt"))
{
    w.WriteLine (123);                           // Записывает "123"
    w.WriteLine (true);                          // Записывает слово "true"
}
```

```
using (TextReader r = File.OpenText ("data.txt"))
{
    int myInt = int.Parse (r.ReadLine());      // myInt == 123
    bool yes = bool.Parse (r.ReadLine());       // yes == true
}
```

Кодировки символов

Сами по себе `TextReader` и `TextWriter` — всего лишь абстрактные классы, не подключенные ни к потоку, ни к опорному хранилищу. Однако типы `StreamReader` и `StreamWriter` подключены к лежащему в основе байт-ориентированному потоку, поэтому они должны выполнять преобразование между символами и байтами. Они делают это посредством класса `Encoding` из пространства имен `System.Text`, который выбирается при конструировании экземпляра `StreamReader` или `StreamWriter`. Если ничего не выбрано, тогда применяется стандартная кодировка UTF-8.



В случае явного указания кодировки экземпляр `StreamWriter` по умолчанию будет записывать в начале потока префикс для идентификации кодировки. Обычно такое действие нежелательно и предотвратить его можно, конструируя экземпляр класса кодировки следующим образом:

```
var encoding = new UTF8Encoding (
    encoderShouldEmitUTF8Identifier:false,
    throwOnInvalidBytes:true);
```

Второй аргумент сообщает `StreamWriter` (или `StreamReader`) о необходимости генерации исключения, если встречаются байты, которые не имеют допустимой строковой трансляции для их кодировки, что соответствует стандартному поведению, когда кодировка не указана.

Простейшей из всех кодировок является ASCII, т.к. в ней каждый символ представлен одним байтом. Кодировка ASCII отображает первые 127 символов набора Unicode на одиночные байты, охватывая символы, которые находятся на англоязычной клавиатуре. Большинство других символов, включая специализированные и неанглийские символы, не могут быть представлены в ASCII и преобразуются в символ `\u00a0`. Стандартная кодировка UTF-8 может отобразить все выделенные символы Unicode, но она сложнее. Первые 127 символов кодируются в одиночный байт для совместимости с ASCII; остальные символы кодируются в варьирующееся количество байтов (чаще всего в два или три). Взгляните на приведенный ниже код:

```
using (TextWriter w=File.CreateText ("but.txt")) //Использовать стандартную
    w.WriteLine ("but-");                           // кодировку UTF-8

using (Stream s = File.OpenRead ("but.txt"))
    for (int b; (b = s.ReadByte()) > -1;)
        Console.WriteLine (b);
```

За словом “`but`” выводится не стандартный знак переноса, а символ длинного тире (`—`), U+2014. Давайте исследуем вывод:

```
98    // b
117   // u
116   // t
226   // байт 1 длинного тире      Обратите внимание, что значения байтов
128   // байт 2 длинного тире      больше или равны 128 для каждой части
148   // байт 3 длинного тире      многобайтной последовательности
13    // <CR>
10    // <LF>
```

Символ длинного тире находится за пределами первых 127 символов набора Unicode и потому при кодировании в UTF-8 требует более одного байта (трех в данном случае). Кодировка UTF-8 эффективна с западным алфавитом, т.к. большинство популярных символов занимают только один байт. Она также легко понижается до ASCII просто за счет игнорирования всех байтов со значениями больше 127. Недостаток кодировки UTF-8 в том, что поиск внутри потока является ненадежным, поскольку позиция символа не соответствует позиции его байтов в потоке. Альтернативой является кодировка UTF-16 (обозначенная просто как Unicode в классе Encoding). Ниже показано, как записать ту же самую строку с помощью UTF-16:

```
using (Stream s = File.Create ("but.txt"))
using (TextWriter w = new StreamWriter (s, Encoding.Unicode))
    w.WriteLine ("but-");
foreach (byte b in File.ReadAllBytes ("but.txt"))
    Console.WriteLine (b);
```

Вывод будет таким:

```
255   // Маркер порядка байтов 1
254   // Маркер порядка байтов 2
98    // 'b', байт 1
0     // 'b', байт 2
117   // 'u', байт 1
0     // 'u', байт 2
116   // 't', байт 1
0     // 't', байт 2
20    // '--', байт 1
32    // '--', байт 2
13    // <CR>, байт 1
0     // <CR>, байт 2
10    // <LF>, байт 1
0     // <LF>, байт 2
```

Формально кодировка UTF-16 использует два или четыре байта на символ (есть около миллиона выделенных или зарезервированных символов Unicode, поэтому двух байтов не всегда достаточно). Но из-за того, что тип char в C# сам имеет ширину только 16 битов, кодировка UTF-16 всегда будет применять в точности два байта на один символ char. В результате упрощается переход по индексу конкретного символа внутри потока.

Кодировка UTF-16 использует двухбайтный префикс для идентификации записи байтовых пар в порядке “старший байт после младшего” или “старший байт перед младшим” (первым идет менее значащий байт или более значащий байт). Применяемый по умолчанию порядок “старший байт после младшего” считается стандартным для систем на основе Windows.

StringReader и StringWriter

Адаптеры `StringReader` и `StringWriter` вообще не содержат внутри себя поток; взамен в качестве лежащего в основе источника данных они используют строку или экземпляр `StringBuilder`. Это означает, что никакой трансляции байтов не требуется — в действительности классы `StringReader` и `StringWriter` не делают ничего такого, чего нельзя было бы достигнуть с помощью строки или экземпляра `StringBuilder` в паре с индексной переменной. Тем не менее, их преимуществом является совместное использование базового класса с классами `StreamReader/StreamWriter`. Например, пусть имеется строка, содержащая XML-код, и нужно разобрать ее с помощью `XmlReader`. Метод `XmlReader.Create` принимает один из следующих аргументов:

- `URI`
- `Stream`
- `TextReader`

Так каким же образом выполнить XML-разбор строки? Нам повезло, потому что `StringReader` является подклассом `TextReader`. Мы можем создать экземпляр `StringReader` и передать его методу `XmlReader.Create`:

```
XmlReader r = XmlReader.Create (new StringReader (myString));
```

Двоичные адаптеры

Классы `BinaryReader` и `BinaryWriter` осуществляют чтение и запись собственных типов данных: `bool`, `byte`, `char`, `decimal`, `float`, `double`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint` и `ulong`, а также строк и массивов примитивных типов данных.

В отличие от `StreamReader` и `StreamWriter` двоичные адаптеры эффективно сохраняют данные примитивных типов, потому что они представлены в памяти. Таким образом, `int` занимает четыре байта, а `double` — восемь байтов. Строки записываются посредством текстовой кодировки (как в случае `StreamReader` и `StreamWriter`), но с префиксами длины, чтобы сделать возможным чтение последовательности строк без необходимости в наличии специальных разделителей.

Предположим, что есть простой тип со следующим определением:

```
public class Person
{
    public string Name;
    public int Age;
    public double Height;
}
```

Применяя двоичные адаптеры, в класс `Person` можно добавить методы для сохранения его данных в поток и их загрузки из потока:

```
public void SaveData (Stream s)
{
    var w = new BinaryWriter (s);
    w.Write (Name);
```

```

w.Write (Age);
w.Write (Height);
w.Flush();      // Обеспечить очистку буфера BinaryWriter.
                // Мы не будем освобождать/закрывать его, поэтому
}              // в поток можно записывать другие данные
public void LoadData (Stream s)
{
    var r = new BinaryReader (s);
    Name = r.ReadString();
    Age = r.ReadInt32();
    Height = r.ReadDouble();
}

```

Класс `BinaryReader` может также производить чтение в байтовые массивы. Приведенный ниже код читает все содержимое потока, поддерживающего поиск:

```
byte[] data = new BinaryReader (s).ReadBytes ((int) s.Length);
```

Такой прием более удобен, чем чтение напрямую из потока, поскольку он не требует использования цикла для гарантии того, что все данные были прочитаны.

Закрытие и освобождение адаптеров потоков

Доступны четыре способа уничтожения адаптеров потоков.

1. Закрыть только адаптер.
2. Закрыть адаптер и затем закрыть поток.
3. (Для средств записи.) Сбросить адаптер и затем закрыть поток.
4. (Для средств чтения.) Закрыть только поток.



Для адаптеров методы `Close` и `Dispose` являются синонимичными в точности как в случае потоков.

Первый и второй варианты семантически идентичны, т.к. закрытие адаптера приводит к автоматическому закрытию лежащего в основе потока. Всякий раз, когда вы вкладываете операторы `using` друг в друга, то неявно принимаете второй вариант:

```
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
    writer.WriteLine ("Line");
```

Поскольку освобождение при вложении операторов `using` происходит наизнанку, сначала закрывается адаптер, а затем поток. Более того, если внутри конструктора адаптера сгенерировано исключение, то поток все равно закроется. Благодаря вложенным операторам `using` мало что может пойти не так, как было задумано.



Никогда не закрывайте поток перед закрытием или сбросом его средства записи, иначе любые данные, буферизированные в адаптере, будут утеряны.

Третий и четвертый варианты работают из-за того, что адаптеры относятся к необычной категории *необязательно освобождаемых* объектов. Примером, когда может быть принято решение не освобождать адаптер, является ситуация, при которой работа с адаптером закончена, но внутренний поток необходимо оставить открытым для последующего использования:

```
using (FileStream fs = new FileStream ("test.txt", FileMode.Create))
{
    StreamWriter writer = new StreamWriter (fs);
    writer.WriteLine ("Hello");
    writer.Flush();
    fs.Position = 0;
    Console.WriteLine (fs.ReadByte());
}
```

Здесь мы записываем в файл, изменяем позицию в потоке и читаем первый байт перед закрытием потока. Если мы освободим StreamWriter, тогда также закроется лежащий в основе объект FileStream, приводя к неудаче последующего чтения. Обязательное условие состоит в том, что мы вызываем метод Flush для обеспечения записи буфера StreamWriter во внутренний поток.



Адаптеры потоков — со своей семантикой необязательного освобождения — не реализуют расширенный шаблон освобождения, при котором финализатор вызывает метод Dispose. Это позволяет отброшенному адаптеру избежать автоматического освобождения при его подхвате сборщиком мусора.

В классах StreamReader/StreamWriter имеется конструктор, который инструментирует поток о необходимости оставаться открытым после освобождения. Следовательно, вот как можно переписать предыдущий пример:

```
using (var fs = new FileStream ("test.txt", FileMode.Create))
{
    using (var writer = new StreamWriter (fs, new UTF8Encoding (false, true),
                                         0x400, true))
        writer.WriteLine ("Hello");

    fs.Position = 0;
    Console.WriteLine (fs.ReadByte());
    Console.WriteLine (fs.Length);
}
```

ПОТОКИ СО СЖАТИЕМ

В пространстве имён System.IO.Compression доступны два универсальных потока со сжатием: DeflateStream и GZipStream. Оба они применяют популярный алгоритм сжатия, подобный алгоритму, который используется при создании архивов в формате ZIP. Указанные классы отличаются тем, что GZipStream записывает дополнительную информацию в начале и в конце, включая код CRC для обнаружения ошибок. Вдобавок класс GZipStream соответствует стандарту, распознаваемому другим программным обеспечением.

В состав .NET также входит поток `BrotliStream`, который реализует алгоритм сжатия *Brotli*. Класс `BrotliStream` функционирует медленнее классов `DeflateStream` и `GZipStream` более чем в 10 раз, но достигает лучшего коэффициента сжатия. (Падение производительности происходит только при сжатии — распаковка работает очень быстро.)

Все три потока позволяют осуществлять чтение и запись со следующими оговорками:

- при сжатии вы всегда записываете в поток;
- при распаковке вы всегда читаете из потока.

Классы `DeflateStream`, `GZipStream` и `BrotliStream` являются декораторами; они сжимают или распаковывают данные из другого потока, который указывается при конструировании их экземпляров. В следующем примере мы сжимаем и распаковываем последовательность байтов, применяя `FileStream` в качестве опорного хранилища:

```
using (Stream s = File.Create ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Compress))
    for (byte i = 0; i < 100; i++)
        ds.WriteByte (i);
using (Stream s = File.OpenRead ("compressed.bin"))
using (Stream ds = new DeflateStream (s, CompressionMode.Decompress))
    for (byte i = 0; i < 100; i++)
        Console.WriteLine (ds.ReadByte()); // Выводит числа от 0 до 99
```

В случае использования `DeflateStream` сжатый файл имеет длину 102 байта: чуть больше размера исходного файла (класс `BrotliStream` обеспечил бы сжатие до 73 байтов). Дело в том, что сжатие плохо работает с “плотными”, неповторяющимися двоичными данными в файлах (и хуже всего с шифрованными данными, которые лишены закономерности по определению). Сжатие успешно работает с большинством текстовых файлов; в приведенном ниже примере мы с помощью алгоритма *Brotli* сжимаем и распаковываем текстовый поток, состоящий из 1000 слов, которые случайным образом выбраны из короткого предложения. Кроме того, в примере демонстрируется соединение в цепочку потока с опорным хранилищем, потока с декоратором и адаптера (как было показано на рис. 15.1 в начале главы), а также применение асинхронных методов:

```
string[] words = "The quick brown fox jumps over the lazy dog".Split();
Random rand = new Random(0); // Предоставить начальное значение
                            // для согласованности

using (Stream s = File.Create ("compressed.bin"))
using (Stream ds = new BrotliStream (s, CompressionMode.Compress))
using (TextWriter w = new StreamWriter (ds))
    for (int i = 0; i < 1000; i++)
        await w.WriteAsync (words [rand.Next (words.Length)] + " ");
Console.WriteLine (new FileInfo ("compressed.bin").Length); // 808

using (Stream s = File.OpenRead ("compressed.bin"))
using (Stream ds = new BrotliStream (s, CompressionMode.Decompress))
using (TextReader r = new StreamReader (ds))
    Console.Write (await r.ReadToEndAsync()); // Вывод показан ниже:
```

```
lazy lazy the fox the quick The brown fox jumps over fox over fox The  
brown brown brown over brown quick fox brown dog dog lazy fox dog brown  
over fox jumps lazy lazy quick The jumps fox jumps The over jumps dog...
```

Класс `BrotliStream` в этом случае эффективно сжимает текст до 808 байтов — меньше, чем один байт на слово. (Для сравнения класс `DeflateStream` сжимает те же самые данные до 885 байтов.)

Сжатие в памяти

Иногда сжатие нужно выполнять полностью в памяти. Ниже показано, как для такой цели использовать класс `MemoryStream`:

```
byte[] data = new byte[1000];           // Мы можем ожидать хороший коэффициент  
                                         // сжатия для пустого массива!  
  
var ms = new MemoryStream();  
using (Stream ds = new DeflateStream (ms, CompressionMode.Compress))  
    ds.Write (data, 0, data.Length);  
  
byte[] compressed = ms.ToArray();  
Console.WriteLine (compressed.Length);      // 11  
  
// Распаковка обратно в массив data:  
ms = new MemoryStream (compressed);  
using (Stream ds = new DeflateStream (ms, CompressionMode.Decompress))  
    for (int i = 0; i < 1000; i += ds.Read (data, i, 1000 - i));
```

Оператор `using` вокруг `DeflateStream` закрывает его рекомендуемым способом, сбрасывая любые незаписанные буферы. Вдобавок также закрывается внутренний поток `MemoryStream`, т.е. для извлечения данных нам придется вызвать метод `ToArray`.

Ниже представлен альтернативный подход, не закрывающий поток `MemoryStream`, в котором используются асинхронные методы чтения и записи:

```
byte[] data = new byte[1000];  
  
MemoryStream ms = new MemoryStream();  
using (Stream ds = new DeflateStream (ms, CompressionMode.Compress, true))  
    await ds.WriteAsync (data, 0, data.Length);  
  
Console.WriteLine (ms.Length);            // 113  
ms.Position = 0;  
using (Stream ds = new DeflateStream (ms, CompressionMode.Decompress))  
    for (int i = 0; i < 1000; i += await ds.ReadAsync (data, i, 1000 - i));
```

Дополнительный флаг, переданный конструктору `DeflateStream`, сообщает о том, что в отношении освобождения лежащего в основе потока не следует соблюдать обычный протокол. Другими словами, поток `MemoryStream` остается открытым, позволяя устанавливать его в нулевую позицию и читать повторно.

Сжатие файлов с помощью `gzip` в Unix

Алгоритм сжатия класса `GZipStream` популярен в системах Unix в качестве формата сжатых файлов. Каждый исходный файл сжимается в отдельный целевой файл с расширением `.gz`.

Следующие методы работают с утилитами командной строки gzip и gunzip в среде Unix:

```
async Task GZip (string sourcefile, bool deleteSource = true)
{
    var gzip = $"{sourcefile}.gz";
    if (File.Exists (gzip))
        throw new Exception ("Gzip file already exists");
        // Файл gzip уже существует

    // Сжатие
    using (FileStream inStream = File.Open (sourcefile, FileMode.Open))
    using (FileStream outStream = new FileStream (gzip, FileMode.CreateNew))
    using (GZipStream gzipStream =
        new GZipStream (outStream, CompressionMode.Compress))
        await inStream.CopyToAsync (gzipStream);

    if (deleteSource) File.Delete (sourcefile);
}

async Task GUnzip (string gzipfile, bool deleteGzip = true)
{
    if (Path.GetExtension (gzipfile) != ".gz")
        throw new Exception ("Not a gzip file");
        // Не файл gzip

    var uncompressedFile = gzipfile.Substring (0, gzipfile.Length - 3);
    if (File.Exists (uncompressedFile))
        throw new Exception ("Destination file already exists");
        // Целевой файл уже существует

    // Распаковка
    using (FileStream uncompressToStream =
        File.Open (uncompressedFile, FileMode.Create))
    using (FileStream zipfileStream = File.Open (gzipfile, FileMode.Open))
    using (var unzipStream =
        new GZipStream (zipfileStream, CompressionMode.Decompress))
        await unzipStream.CopyToAsync (uncompressToStream);

    if (deleteGzip) File.Delete (gzipfile);
}
```

Вот код, который сжимает файл:

```
await GZip ("/tmp/myfile.txt");           // Создает /tmp/myfile.txt.gz
```

А этот код его распаковывает:

```
await GUnzip ("/tmp/myfile.txt.gz")      // Создает /tmp/myfile.txt
```

Работа с ZIP-файлами

Классы ZipArchive и ZipFile из пространства имен System.IO.Compression поддерживают формат сжатия ZIP. Преимущество формата ZIP над DeflateStream и GZipStream заключается в том, что он также действует в качестве контейнера для множества файлов и совместим с ZIP-файлами, созданными с помощью проводника Windows.

Класс ZipArchive работает с потоками, тогда как ZipFile используется в более распространенном сценарии работы с файлами. (ZipFile является статическим вспомогательным классом для ZipArchive.)

Метод CreateFromDirectory класса ZipFile добавляет все файлы из указанного каталога в ZIP-файл:

```
ZipFile.CreateFromDirectory (@"d:\MyFolder", @"d:\archive.zip");
```

Метод ExtractToDirectory выполняет обратное действие, извлекая содержимое ZIP-файла в заданный каталог:

```
ZipFile.ExtractToDirectory(@"d:\archive.zip", @"d:\MyFolder");
```

(Начиная с версии .NET 8, можно также указывать объект Stream вместо пути к ZIP-файлу.)

При сжатии можно выбирать оптимизацию по размеру файла или по скорости, а также необходимость включения в архив имени исходного каталога. Последний вариант приведет к тому, что в нашем примере внутри архива создается подкаталог по имени MyFolder, куда будут помещены сжатые файлы.

Класс ZipFile имеет метод Open, предназначенный для чтения/записи индивидуальных элементов. Он возвращает объект ZipArchive (который также можно получить, создав экземпляр ZipArchive с объектом Stream). При вызове метода Open потребуется указать имя файла и действие, которое должно быть произведено с архивом — Read (чтение), Create (создание) или Update (обновление). Затем можно выполнить перечисление по существующим элементам через свойство Entries либо искать отдельный файл с помощью метода GetEntry:

```
using (ZipArchive zip = ZipFile.Open(@"d:\zz.zip", ZipArchiveMode.Read))
    foreach (ZipArchiveEntry entry in zip.Entries)
        Console.WriteLine(entry.FullName + " " + entry.Length);
```

В классе ZipArchiveEntry также есть методы Delete, ExtractToFile (на самом деле он представляет собой расширяющий метод из класса ZipFileExtensions) и Open, который возвращает экземпляр Stream с возможностью чтения/записи. Создавать новые элементы можно посредством вызова метода CreateEntry (или расширяющего метода CreateEntryFromFile) на ZipArchive. Приведенный ниже код создает архив d:\zz.zip, к которому добавляется файл foo.dll со структурой каталогов bin\x86 внутри архива:

```
byte[] data = File.ReadAllBytes(@"d:\foo.dll");
using (ZipArchive zip = ZipFile.Open(@"d:\zz.zip", ZipArchiveMode.Update))
    zip.CreateEntry(@"bin\x64\foo.dll").Open().Write(data, 0, data.Length);
```

То же самое можно было бы сделать полностью в памяти, создав экземпляр ZipArchive с потоком MemoryStream.

Работа с файлами Tar

Типы в пространстве имен System.Formats.Tar (начиная с .NET 7) поддерживают формат архива .tar, популярный в системах Unix для объединения нескольких файлов.

Чтобы создать файл .tar (*архив*), вызовите TarFile.CreateFromDirectory:

```
TarFile.CreateFromDirectory ("/tmp/testfolder", "/tmp/test.tar", false);
```

(Третий аргумент указывает, включать ли имя базового каталога в записи архива.)

Для извлечения файлов из архива вызовите TarFile.ExtractToDirectory:

```
TarFile.ExtractToDirectory ("/tmp/test.tar", "/tmp/testfolder", true);
```

(Третий аргумент указывает, перезаписывать ли существующие файлы.)

Оба метода позволяют указывать объект Stream вместо пути к файлу .tar. В следующем примере мы записываем архив в поток памяти, а затем используем GZipStream для сжатия этого потока в файл .tar.gz:

```
var ms = new MemoryStream();
TarFile.CreateFromDirectory ("/tmp/testfolder", ms, false);
ms.Position = 0; // So that we can re-use the stream for reading.
using (var fs = File.Create ("/tmp/test.tar.gz"))
using (var gz = new GZipStream (fs, CompressionMode.Compress))
    ms.CopyTo (gz);
```

(Сжатие файла .tar в .tar.gz полезно, поскольку в отличие от формата .zip формат .tar сам по себе не поддерживает сжатие.) Вот как можно извлечь файл .tar.gz:

```
using (var fs = File.OpenRead ("/tmp/test.tar.gz"))
using (var gz = new GZipStream (fs, CompressionMode.Decompress))
    TarFile.ExtractToDirectory (gz, "/tmp/testfolder", true);
```

Доступ к API-интерфейсу возможен на более низком уровне с помощью классов TarReader и TarWriter. Ниже демонстрируется применение класса TarReader:

```
using (FileStream archiveStream = File.OpenRead ("/tmp/test.tar"))
using (TarReader reader = new (archiveStream))
    while (true)
    {
        TarEntry entry = reader.GetNextEntry();
        if (entry == null) break; // Записей больше нет
        Console.WriteLine (
            $"Entry {entry.Name} is {entry.DataStream.Length} bytes long");
        entry.ExtractToFile (
            Path.Combine ("/tmp/testfolder", entry.Name), true);
    }
```

Операции с файлами и каталогами

Пространство имен System.IO предоставляет набор типов для выполнения в отношении файлов и каталогов “обслуживающих” операций, таких как копирование и перемещение, создание каталогов и установка файловых атрибутов и прав доступа. Для большинства средств можно выбирать один из двух классов: первый предлагает статические методы, а второй — методы экземпляра.

Статические классы

File и Directory

**Классы с методами экземпляра
(сконструированного с указанием имени файла или каталога)**

FileInfo и DirectoryInfo

В добавок имеется статический класс по имени Path. Он ничего не делает с файлами или каталогами, а предоставляет методы строкового манипулирования для имен файлов и путей к каталогам. Класс Path также помогает при работе с временными файлами.

Класс File

File — это статический класс, все методы которого принимают имя файла. Имя файла может или указываться относительно текущего каталога, или быть полностью заданным, включая каталог. Ниже перечислены методы класса File (все они являются public и static):

```
bool Exists (string path); // Возвращает true, если файл существует
void Delete (string path);
void Copy (string sourceFileName, string destFileName);
void Move (string sourceFileName, string destFileName);
void Replace (string sourceFileName, string destinationFileName,
              string destinationBackupFileName);

FileAttributes GetAttributes (string path);
void SetAttributes (string path, FileAttributes fileAttributes);

void Decrypt (string path);
void Encrypt (string path);

DateTime GetCreationTime (string path);           // Так же доступны
DateTime GetLastAccessTime (string path);         // версии UTC.
DateTime GetLastWriteTime (string path);

void SetCreationTime (string path, DateTime creationTime);
void SetLastAccessTime (string path, DateTime lastAccessTime);
void SetLastWriteTime (string path, DateTime lastWriteTime);

FileSecurity GetAccessControl (string path);
FileSecurity GetAccessControl (string path,
                               AccessControlSections includeSections);
void SetAccessControl (string path, FileSecurity fileSecurity);
```

Метод Move генерирует исключение, если файл назначения уже существует; метод Replace этого не делает. Оба метода позволяют переименовывать файл, а также перемещать его в другой каталог.

Метод Delete генерирует исключение UnauthorizedAccessException, если файл помечен как предназначенный только для чтения; ситуацию можно прояснить заранее, вызвав метод GetAttributes. Кроме того, он генерирует исключение, если ОС не выдает вашему процессу разрешение на удаление для этого файла. Метод GetAttributes возвращает значение перечисления FileMode со следующими членами:

Archive, Compressed, Device, Directory, Encrypted, Hidden, IntegritySystem, Normal, NoScrubData, NotContentIndexed, Offline, ReadOnly, ReparsePoint, SparseFile, System, Temporary

Члены перечисления `FileAttribute` допускают комбинирование. Ниже показано, как переключить один атрибут файла, не затрагивая остальные:

```
string filePath = "test.txt";
FileAttributes fa = File.GetAttributes(filePath);
if ((fa & FileAttributes.ReadOnly) != 0)
{
    // Использовать операцию исключающего ИЛИ (^) для переключения флага ReadOnly
    fa ^= FileAttributes.ReadOnly;
    File.SetAttributes(filePath, fa);
}
// Теперь можно удалить файл, например:
File.Delete(filePath);
```



Класс `FileInfo` предлагает более простой способ изменения флага доступности только для чтения, связанного с файлом:

```
new FileInfo("test.txt").IsReadOnly = false;
```

Атрибуты сжатия и шифрования



Данное средство поддерживается только в Windows и требует загрузки NuGet-пакета `System.Management`.

Атрибуты файла `Compressed` и `Encrypted` соответствуют флагам сжатия и шифрования в диалоговом окне *свойств* файла или каталога, которое можно открыть в проводнике Windows. Такой тип сжатия и шифрования прозрачен в том, что ОС делает всю работу “за кулисами”, позволяя читать и записывать простые данные.

Для изменения атрибута `Compressed` или `Encrypted` нельзя применять метод `SetAttributes` — если вы попытаетесь, то он молча откажется! В случае шифрования обойти проблему легко: нужно просто вызывать методы `Encrypt` и `Decrypt` класса `File`. В отношении сжатия ситуация сложнее; одно из решений предполагает использование API-интерфейса WMI (`Windows Management Instrumentation` — инструментарий управления Windows) из пространства имен `System.Management`. Следующий метод сжимает каталог, возвращая 0 в случае успеха (или код ошибки WMI в случае неудачи):

```
static uint CompressFolder(string folder, bool recursive)
{
    string path = "Win32_Directory.Name='\" + folder + '\"";
    using (ManagementObject dir = new ManagementObject(path))
    using (ManagementBaseObject p = dir.GetMethodParameters("CompressEx"))
    {
        p["Recursive"] = recursive;
        using (ManagementBaseObject result = dir.InvokeMethod("CompressEx",
            p, null))
        return (uint) result.Properties["ReturnValue"].Value;
    }
}
```

Для выполнения распаковки имя CompressEx понадобится заменить именем UncompressEx.

Прозрачное шифрование полагается на ключ, построенный на основе пароля пользователя, вошедшего в систему. Система устойчива к изменениям пароля, которые производятся аутентифицированным пользователем, но если пароль сбрасывается администратором, тогда данные в зашифрованных файлах восстановлению не подлежат.



Прозрачное шифрование и сжатие требуют специальной поддержки со стороны файловой системы. Файловая система NTFS (чаще всего применяемая на жестких дисках) такие возможности поддерживает, а CDFS (на компакт-дисках) и FAT (на сменных носителях) — нет.

Определить, поддерживает ли том сжатие и шифрование, можно посредством взаимодействия с Win32:

```
using System;
using System.IO;
using System.Text;
using System.ComponentModel;
using System.Runtime.InteropServices;
class SupportsCompressionEncryption
{
    const int SupportsCompression = 0x10;
    const int SupportsEncryption = 0x20000;

    [DllImport ("Kernel32.dll", SetLastError = true)]
    extern static bool GetVolumeInformation (string vol, StringBuilder name,
        int nameSize, out uint serialNum, out uint maxNameLen, out uint flags,
        StringBuilder fileSysName, int fileSysNameSize);

    static void Main()
    {
        uint serialNum, maxNameLen, flags;
        bool ok = GetVolumeInformation (@"C:\", null, 0, out serialNum,
                                         out maxNameLen, out flags, null, 0);
        if (!ok)
            throw new Win32Exception();
        bool canCompress = (flags & SupportsCompression) != 0;
        bool canEncrypt = (flags & SupportsEncryption) != 0;
    }
}
```

Безопасность файлов в Windows



Данное средство поддерживается только в Windows и требует загрузки NuGet-пакета System.IO.FileSystem.AccessControl.

Класс FileSecurity (из пространства имен System.Security.AccessControl) позволяет запрашивать и изменять права доступа ОС, назначенные пользователям и ролям.

В приведенном ниже примере мы выводим существующие права доступа к файлу, после чего назначаем права на выполнение группе Users:

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;
void ShowSecurity (FileSecurity sec)
{
    AuthorizationRuleCollection rules = sec.GetAccessRules (true, true,
        typeof (NTAccount));
    foreach (FileSystemAccessRule r in rules.Cast<FileSystemAccessRule>()
        .OrderBy (rule => rule.IdentityReference.Value))
    {
        // Например, MyDomain/Joe
        Console.WriteLine ($" {r.IdentityReference.Value}");
        // Allow или Deny: например, FullControl
        Console.WriteLine ($" {r.FileSystemRights}: {r.AccessControlType}");
    }
}
var file = "sectest.txt";
File.WriteAllText (file, "File security test.");
var sid = new SecurityIdentifier (WellKnownSidType.BuiltinUsersSid, null);
string usersAccount = sid.Translate (typeof (NTAccount)).ToString();
Console.WriteLine ($"User: {usersAccount}");
FileSecurity sec = new FileSecurity (file,
    AccessControlSections.Owner |
    AccessControlSections.Group |
    AccessControlSections.Access);
Console.WriteLine ("AFTER CREATE:"); // После создания
ShowSecurity(sec); // Группа BUILTIN\Users не имеет права доступа Write
sec.ModifyAccessRule (AccessControlModification.Add,
    new FileSystemAccessRule (usersAccount, FileSystemRights.Write,
        AccessControlType.Allow),
    out bool modified);
Console.WriteLine ("AFTER MODIFY:"); // После модификации
ShowSecurity (sec); // Группа BUILTIN\Users имеет права доступа Write
```

В разделе “Специальные папки” далее в главе будет представлен еще один пример.

Безопасность файлов в Unix

Начиная с версии .NET 7, в классе `File` определены методы `GetUnix FileMode` и `SetUnix FileMode`, предназначенные для получения и установки разрешений файлов в системах Unix. Метод `Directory.CreateDirectory` теперь также перегружен для принятия файлового режима Unix, и при создании файла можно указывать файловый режим следующим образом:

```
var fs = new FileStream ("test.txt",
    new FileStreamOptions
    {
        Mode = FileMode.Create,
        UnixCreateMode = Unix FileMode.UserRead | Unix FileMode.UserWrite
    });

```

Класс Directory

Статический класс `Directory` предлагает набор методов, аналогичных методам в классе `File` — для проверки существования каталога (`Exists`), для перемещения каталога (`Move`), для удаления каталога (`Delete`), для получения/установки времени создания или времени последнего доступа и для получения/установки разрешений безопасности. Кроме того, класс `Directory` открывает доступ к следующим статическим методам:

```
string GetCurrentDirectory ();
void SetCurrentDirectory (string path);

DirectoryInfo CreateDirectory (string path);
 DirectoryInfo GetParent (string path);
 string GetDirectoryRoot (string path);

string[] GetLogicalDrives(); // Получает точки монтирования в Unix

// Все перечисленные ниже методы возвращают полные пути:
string[] GetFiles (string path);
string[] GetDirectories (string path);
string[] GetFileSystemEntries (string path);

IEnumerable<string> EnumerateFiles (string path);
IEnumerable<string> EnumerateDirectories (string path);
IEnumerable<string> EnumerateFileSystemEntries (string path);
```



Последние три метода потенциально более эффективны, чем варианты `Get*`, т.к. к ним применяется ленивое выполнение — данные извлекаются из файловой системы при перечислении последовательности. Методы `Enumerate*` особенно хорошо подходят для запросов LINQ.

Методы `Enumerate*` и `Get*` перегружены, чтобы также принимать параметры `searchPattern` (строка) и `searchOption` (перечисление). В случае указания `SearchOption.SearchAllSubDirectories` будет выполняться рекурсивный поиск в подкаталогах. Методы `*FileSystemEntries` комбинируют результаты методов `*Files` и `*Directories`.

Вот как создать каталог, если он еще не существует:

```
if (!Directory.Exists(@"d:\test"))
    Directory.CreateDirectory(@"d:\test");
```

FileInfo и DirectoryInfo

Статические методы классов `File` и `Directory` удобны для выполнения одиночной операции над файлом или каталогом. Если необходимо вызвать последовательность методов подряд, то классы `FileInfo` и `DirectoryInfo` предоставляют объектную модель, которая облегчает работу.

Класс `FileInfo` предлагает большинство статических методов класса `File` в форме методов экземпляра — с несколькими дополнительными свойствами вроде `Extension`, `Length`, `IsReadOnly` и `Directory` — для возвращения объекта `DirectoryInfo`. Например:

```

static string TestDirectory =>
    RuntimeInformation.IsOSPlatform (OSPlatform.Windows)
    ? @"C:\Temp"
    : "/tmp";
Directory.CreateDirectory (TestDirectory);
FileInfo fi = new FileInfo (Path.Combine (TestDirectory, "FileInfo.txt"));
Console.WriteLine (fi.Exists);           // False
using (TextWriter w = fi.CreateText())
    w.Write ("Some text");
Console.WriteLine (fi.Exists);           // False (по-прежнему)
fi.Refresh();
Console.WriteLine (fi.Exists);           // True
Console.WriteLine (fi.Name);             // FileInfo.txt
Console.WriteLine (fi.FullName);          // c:\temp\FileInfo.txt (Windows)
                                         // /tmp/FileInfo.txt (Unix)
Console.WriteLine (fi.DirectoryName);    // c:\temp (Windows)
                                         // /tmp (Unix)
Console.WriteLine (fi.Directory.Name);   // temp
Console.WriteLine (fi.Extension);        // .txt
Console.WriteLine (fi.Length);           // 9
fi.Encrypt();
fi.Attributes ^= FileAttributes.Hidden; // (Переключает флаг скрытости)
fi.IsReadOnly = true;
Console.WriteLine (fi.Attributes);        // ReadOnly,Archive,Hidden,Encrypted
Console.WriteLine (fi.CreationTime);     // 3/09/2019 1:24:05 PM
fi.MoveTo (Path.Combine (TestDirectory, "FileInfoX.txt"));
 DirectoryInfo di = fi.Directory;
Console.WriteLine (di.Name);             // temp или tmp
Console.WriteLine (di.FullName);          // c:\temp или /tmp
Console.WriteLine (di.Parent.FullName);   // c:\ или /
di.CreateSubdirectory ("SubFolder");

```

А вот как использовать класс DirectoryInfo для перечисления файлов и подкаталогов:

```

DirectoryInfo di = new DirectoryInfo (@"e:\photos");
foreach (FileInfo fi in di.GetFiles ("*.jpg"))
    Console.WriteLine (fi.Name);
foreach (DirectoryInfo subDir in di.GetDirectories())
    Console.WriteLine (subDir.FullName);

```

Path

В статическом классе Path определены методы и поля для работы с путями и именами файлов.

Предположим, что имеются следующие определения:

```

string dir  = @"c:\mydir";                // или /mydir
string file = "myfile.txt";
string path = @"c:\mydir\myfile.txt";       // или /mydir/myfile.txt
Directory.SetCurrentDirectory (@"k:\demo"); // или /demo

```

Ниже приведены выражения, демонстрирующие применение методов и полей класса Path.

Выражение	Результат (Windows, Unix)
Directory.GetCurrentDirectory()	k:\demo\ или /demo
Path.IsPathRooted (file)	false
Path.IsPathRooted (path)	true
Path.GetPathRoot (path)	c:\ или /
Path.GetDirectoryName (path)	c:\mydir или /mydir
Path.GetFileName (path)	myfile.txt
Path.GetFullPath (file)	k:\demo\myfile.txt или /demo/myfile.txt
Path.Combine (dir, file)	c:\mydir\myfile.txt или /mydir/myfile.txt
Файловые расширения:	
Path.HasExtension (file)	true
Path.GetExtension (file)	.txt
Path.GetFileNameWithoutExtension (file)	myfile
Path.ChangeExtension (file, ".log")	myfile.log
Разделители и символы:	
Path.DirectorySeparatorChar	\ или /
Path.AltDirectorySeparatorChar	/
Path.PathSeparator	; или :
Path.VolumeSeparatorChar	: или /
Path.GetInvalidPathChars()	символы от 0 до 31 и "<> или 0
Path.GetInvalidFileNameChars()	символы от 0 до 31 и "<> _:*?\ или 0 и /
Временные файлы:	
Path.GetTempPath()	<папка локального пользователя>\Temp или /tmp
Path.GetRandomFileName()	d2dwuzjf.dnp
Path.GetTempFileName()	<папка локального пользователя>\Temp\tmp14B.tmp или /tmp/tmpubSUYO.tmp

Метод Combine особенно полезен: он позволяет комбинировать каталог и имя файла (или два каталога) без предварительной проверки, присутствует ли завершающий разделитель пути, и автоматически использует корректный разделитель пути для ОС. Для Combine имеются перегруженные версии, принимающие вплоть до четырех имен каталогов и/или файлов.

Метод `GetFullPath` преобразует путь, указанный относительно текущего каталога, в абсолютный путь. Он принимает значения, подобные `..\..\file.txt`.

Метод `GetRandomFileName` возвращает по-настоящему уникальное символьное имя в формате 8.3, не создавая файла. Метод `GetTempFileName` генерирует временное имя файла с использованием автоинкрементного счетчика, который повторяется для каждого 65 000 файлов. Затем он создает в локальном временном каталоге пустой файл с таким именем.



По завершении работы с файлом, имя которого сгенерировано методом `GetTempFileName`, вы должны его удалить; иначе со временем возникнет исключение (после 65 000 вызовов `GetTempFileName`). Если это проблематично, тогда для результатов выполнения `GetTempPath` и `GetRandomFileName` можно вызвать метод `Combine`. Только будьте осторожны, чтобы не переполнить жесткий диск пользователя!

Специальные папки

В классах `Path` и `Directory` отсутствует средство нахождения таких папок, как `My Documents`, `Program Files`, `Application Data` и т.д. Взамен задача решается с помощью метода `GetFolderPath` класса `System.Environment`:

```
string myDocPath = Environment.GetFolderPath  
    (Environment.SpecialFolder.MyDocuments);
```

Тип `Environment.SpecialFolder` представляет собой перечисление со значениями, охватывающими все специальные каталоги в Windows, такими как `AdminTools`, `ApplicationData`, `Fonts`, `History`, `SendTo`, `StartMenu` и т.д. Они покрывают все кроме каталога исполняющей среды .NET, который можно получить следующим образом:

```
System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory()
```



Большинство специальных папок в системах Unix не имеют назначенных путей. В Ubuntu Linux 18.04 Desktop пути имеют следующие специальные папки: `ApplicationData`, `CommonApplicationData`, `Desktop`, `DesktopDirectory`, `LocalApplicationData`, `MyDocuments`, `MyMusic`, `MyPictures`, `MyVideos`, `Templates` и `UserProfile`.

Особую ценность в системах Windows представляет каталог `ApplicationData`: именно здесь можно хранить настройки, которые перемещаются с пользователем по сети (если бслужащие профили разрешены в домене сети), а также каталог `LocalApplicationData`, предназначенный для неперемещаемых данных (специфичных для зарегистрированного пользователя), и каталог `CommonApplicationData`, который совместно используется всеми пользователями компьютера. Запись данных приложения в указанные папки считается предпочтительнее применения реестра Windows. Стандартный протокол сохранения данных в этих каталогах предусматривает создание подкаталога с именем, которое совпадает с названием приложения:

```

string localAppDataPath = Path.Combine (
    Environment.GetFolderPath (Environment.SpecialFolder.ApplicationData),
    "MyCoolApplication");
if (!Directory.Exists (localAppDataPath))
    Directory.CreateDirectory (localAppDataPath);

```

При работе с каталогом CommonApplicationData можно попасть в одну коварную ловушку: если пользователь запускает программу с повышенными административными полномочиями, после чего она создает папки и файлы в CommonApplicationData, то пользователю может не хватить полномочий для замены этих файлов позже, когда он запустит программу от имени обычной учетной записи. (Похожая проблема возникает при переключении между учетными записями с ограниченными полномочиями.) Такую проблему можно обойти за счет создания желаемой папки (с правами доступа для кого угодно) как части процесса установки.

Еще одним местом для записи конфигурационных и журнальных файлов является базовый каталог приложения, который можно получить с помощью свойства AppDomain.CurrentDomain.BaseDirectory. Однако поступать подобным образом не рекомендуется, потому что ОС, вероятно, не разрешит приложению записывать в этот каталог после первоначальной установки (без повышения прав до администратора).

Запрашивание информации о томе

Запрашивать информацию об устройствах на компьютере можно посредством класса DriveInfo:

```

DriveInfo c = new DriveInfo ("C");           // Запросить устройство C:..
                                                // В Unix: /
long totalSize = c.TotalSize;                // Объем в байтах.
long freeBytes = c.TotalFreeSpace;           //忽роригирует дисковую квоту.
long freeToMe = c.AvailableFreeSpace;        // Учитывает дисковую квоту.

foreach (DriveInfo d in DriveInfo.GetDrives()) // Все определенные устройства
                                                // В Unix: точки монтирования
{
    Console.WriteLine (d.Name);                // C:\ 
    Console.WriteLine (d.DriveType);            // Жесткий диск
    Console.WriteLine (d.RootDirectory);        // C:\ 
    if (d.IsReady)      // Если устройство не готово, то следующие
                        // два свойства сгенерируют исключения:
    {
        Console.WriteLine (d.VolumeLabel);       // The Sea Drive
        Console.WriteLine (d.DriveFormat);        // NTFS
    }
}

```

Статический метод GetDrives возвращает все отображенные устройства, включая приводы компакт-дисков, карты памяти и сетевые устройства. DriveType представляет собой перечисление со следующими значениями:

Unknown, NoRootDirectory, Removable, Fixed, Network, CDRom, Ram

Перехват событий файловой системы

Класс `FileSystemWatcher` позволяет отслеживать действия, производимые над каталогом (и дополнительно над его подкаталогами). Класс `FileSystemWatcher` поддерживает события, которые инициируются при создании, модификации, переименовании и удалении файлов или подкаталогов, а также при изменении их атрибутов. События выдаются независимо от инициатора изменения — пользователя или процесса. Ниже приведен пример:

```
Watch (GetTestDirectory(), "*.txt", true);
void Watch (string path, string filter, bool includeSubDirs)
{
    using (var watcher = new FileSystemWatcher (path, filter))
    {
        watcher.Created += FileCreatedChangedDeleted;
        watcher.Changed += FileCreatedChangedDeleted;
        watcher.Deleted += FileCreatedChangedDeleted;
        watcher.Renamed += FileRenamed;
        watcher.Error += FileError;

        watcher.IncludeSubdirectories = includeSubDirs;
        watcher.EnableRaisingEvents = true;

        // Прослушивание событий; завершение по нажатию <Enter>
        Console.WriteLine ("Listening for events - press <enter> to end");
        Console.ReadLine();
    }
    // Освобождение экземпляра FileSystemWatcher останавливает
    // дальнейшую выдачу событий
}

// Файл создан, изменен или удален
void FileCreatedChangedDeleted (object o, FileSystemEventArgs e)
    => Console.WriteLine ("File {0} has been {1}", e.FullPath, e.ChangeType);

// Файл переименован
void FileRenamed (object o, RenamedEventArgs e)
    => Console.WriteLine ("Renamed: {0}->{1}", e.OldFullPath, e.FullPath);

// Возникла ошибка
void FileError (object o, ErrorEventArgs e)
    => Console.WriteLine ("Error: " + e.GetException().Message);

string GetTestDirectory() =>
    RuntimeInformation.IsOSPlatform (OSPlatform.Windows)
    ? @"C:\Temp"
    : "/tmp";
```



Поскольку `FileSystemWatcher` инициирует события в отдельном потоке, для кода обработки событий должен быть предусмотрен перехват исключений, чтобы предотвратить нарушение работы приложения из-за возникновения ошибки. За дополнительной информацией обращайтесь в раздел “Обработка исключений” главы 14.

Событие `Error` не информирует об ошибках, связанных с файловой системой; взамен оно отражает факт переполнения буфера событий `FileSystemWatcher` событиями `Changed`, `Created`, `Deleted` или `Renamed`. Изменить размер буфера можно с помощью свойства `InternalBufferSize`.

Свойство `IncludeSubdirectories` применяется рекурсивно. Таким образом, если создать экземпляр `FileSystemWatcher` для `C:\` со свойством `IncludeSubdirectories`, установленным в `true`, то события будут инициироваться при изменении любого файла или каталога на всем жестком диске `C:`.



Ловушка, которую можно попасть при использовании `FileSystemWatcher`, связана с открытием и чтением вновь созданных или обновленных файлов до того, как полностью завершится их наполнение или обновление. Если вы работаете совместно с каким-то другим программным обеспечением, создающим файлы, то потребуется предусмотреть некоторую стратегию по смягчению проблемы, например, создание файлов с неотслеживаемым расширением и затем их переименование после завершения записи.

Безопасность, обеспечивающая операционной системой

На все приложения распространяются ограничения ОС, основанные на привилегиях учетной записи пользователя. Такие ограничения влияют на файловый ввод-вывод и другие возможности наподобие доступа к реестру Windows.

В Windows и Unix существуют два типа учетных записей:

- учетная запись администратора/суперпользователя, не накладывающая никаких ограничений в плане доступа на локальном компьютере;
- учетная запись с ограниченными разрешениями, которая сужает административные функции и видимость данных других пользователей.

В среде Windows функциональное средство под названием контроль учетных записей пользователей (`User Account Control — UAC`) означает, что администраторы при входе получают два маркера: административный маркер и маркер рядового пользователя. По умолчанию программы запускаются под маркером рядового пользователя — с ограниченными разрешениями — при условии, что программа не требует *повышения административных полномочий*. Затем пользователь обязан утвердить запрос в открывшемся диалоговом окне.

В Unix пользователи обычно входят в систему с помощью ограниченных учетных записей. Это также справедливо в отношении администраторов, т.к. позволяет уменьшить вероятность неумышленного повреждения системы. Когда пользователю нужно выполнить команду, которая требует повышенных разрешений, он предваряет ее командой `sudo`.

По умолчанию ваше приложение будет запускаться с ограниченными привилегиями пользователя. Таким образом, вы должны придерживаться одного из следующих подходов.

- Писать свое приложение так, чтобы оно могло запускаться без административных привилегий.
- Требовать повышения административных полномочий в манифесте приложения (только Windows) или обнаруживать нехватку обязательных привилегий и предупредить пользователя о необходимости перезапуска приложения от имени администратора/суперпользователя.

Первый подход безопаснее и удобнее для пользователя. Проектировать программу для запуска без административных привилегий в большинстве случаев легко. Вот как можно узнать, работает ли вы под учетной записью администратора:

```
[DllImport("libc")]
public static extern uint getuid();
static bool IsRunningAsAdmin()
{
    if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
    {
        using var identity = WindowsIdentity.GetCurrent();
        var principal = new WindowsPrincipal(identity);
        return principal.IsInRole(WindowsBuiltInRole.Administrator);
    }
    return getuid() == 0;
}
```

При включенном средстве UAC в Windows функция `IsRunningAsAdmin` возвращает `true`, только если текущий процесс имеет повышенные административные полномочия. В Linux она возвращает `true`, только если текущий процесс был запущен от имени суперпользователя (например, `sudo myapp`).

Выполнение под учетной записью стандартного пользователя

Ниже перечислены ключевые действия, которые *не удастся* предпринять под учетной записью стандартного пользователя:

- записывать в следующие каталоги:
- каталог ОС (обычно `\Windows` или `/bin`, `/sbin`, ...) и его подкаталоги;
- каталог файлов программ (`\Program Files` или `/usr/bin`, `/opt`) и его подкаталоги;
- корневой каталог диска с ОС (например, `C:\` или `/`).
- записывать в ветвь `HKEY_LOCAL_MACHINE` реестра (Windows);
- читать данные мониторинга производительности (Windows).

Кроме того, как рядовому пользователю Windows (или даже как администратору), вам может быть отказано в доступе к файлам или ресурсам, которые принадлежат другим пользователям. В Windows для защиты таких ресурсов используется система списков управления доступом (Access Control List — ACL) — вы можете запрашивать и заявлять собственные права в списках ACL через типы

из пространства имен `System.Security.AccessControl`. Списки ACL можно также применять к межпроцессным дескрипторам ожидания, описанным в главе 21.

Если вам отказано в доступе к чему-либо из-за мер безопасности, обеспечивающих ОС, тогда среда CLR обнаруживает отказ и генерирует исключение `UnauthorizedAccessException` (вместо молчаливого сбоя).

В большинстве случаев вы можете иметь дело с ограничениями стандартного пользователя так, как описано далее.

- Записывать файлы в их рекомендуемые места.
- Избегать использования реестра для хранения информации, которая может содержаться в файлах (кроме раздела `HKEY_CURRENT_USER`, к которому вы будете иметь доступ по чтению/записи только в Windows).
- Регистрировать компоненты ActiveX или COM во время установки (только в Windows).

Рекомендованным местом для документов пользователя является `SpecialFolder.MyDocuments`:

```
string docsFolder = Environment.GetFolderPath  
    (Environment.SpecialFolder.MyDocuments);  
  
string path = Path.Combine (docsFolder, "test.txt");
```

Рекомендованным местом для конфигурационных файлов, которые пользователь может пожелать модифицировать за рамками вашего приложения, выглядит как `SpecialFolder.ApplicationData` (только текущий пользователь) или `SpecialFolder.CommonApplicationData` (все пользователи). Обычно вы будете создавать подкаталоги в упомянутых каталогах, основываясь на вашей организации и названии продукта.

Повышение административных полномочий и виртуализация

С помощью манифеста приложения вы можете затребовать, чтобы ОС Windows запрашивала у пользователя повышения административных полномочий всякий раз, когда запускается ваша программа (в ОС Linux такое требование игнорируется):

```
<?xml version="1.0" encoding="utf-8"?>  
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">  
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">  
    <security>  
      <requestedPrivileges>  
        <requestedExecutionLevel level="requireAdministrator" />  
      </requestedPrivileges>  
    </security>  
  </trustInfo>  
</assembly>
```

(Манифести приложений более подробно рассматриваются в главе 17.)

Замена `requireAdministrator` на `asInvoker` сообщает Windows о том, что повышение административных полномочий *не* требуется. Эффект будет почти таким же, как отсутствие манифеста приложения, но только с отключенной *виртуализацией*. Виртуализация — это временная мера, введенная в Windows Vista для корректной работы старых приложений без административных привилегий. Отсутствие манифеста приложения с элементом `requestedExecutionLevel` активизирует такое средство обратной совместимости.

Виртуализация вступает в игру, когда приложение выполняет запись в каталог `Program Files` или `Windows` либо в раздел `HKEY_LOCAL_MACHINE` реестра. Вместо генерации исключения изменения направляются в отдельное место на жестком диске, где они не могут повлиять на первоначальные данные. В итоге приложение не создает помехи ОС или другим нормально функционирующем приложениям.

Размещенные в памяти файлы

Размещенные в памяти файлы поддерживают две основные функции:

- эффективный произвольный доступ к данным файла;
- возможность разделения памяти между различными процессами на одном и том же компьютере.

Типы для размещенных в памяти файлов находятся в пространстве имен `System.IO.MemoryMappedFiles`. Внутренне они работают через API-интерфейс ОС, предназначенный для размещенных в памяти файлов.

Размещенные в памяти файлы и произвольный файловый ввод-вывод

Хотя обычный класс `FileStream` допускает произвольный файловый ввод-вывод (за счет установки свойства `Position` потока), он оптимизирован для последовательного ввода-вывода. Ниже описаны грубые эмпирические правила:

- при последовательном вводе-выводе экземпляры `FileStream` примерно в 10 раз быстрее размещенных в памяти файлов;
- при произвольном вводе-выводе размещенные в памяти файлы примерно в 10 раз быстрее экземпляров `FileStream`.

Изменение свойства `Position` экземпляра `FileStream` может занимать несколько микросекунд — и задержка будет накапливаться, когда это делается в цикле. Класс `FileStream` непригоден для многопоточного доступа, т.к. по мере чтения или записи позиция в нем изменяется.

Чтобы создать размещенный в памяти файл, выполните следующие действия.

1. Получите объект файлового потока (`FileStream`) обычным образом.
2. Создайте экземпляр класса `MemoryMappedFile`, передав его конструктору объект файлового потока.
3. Вызовите метод `CreateViewAccessor` на объекте размещенного в памяти файла.

Выполнение последнего действия приводит к получению объекта `MemoryMappedViewAccessor`, который предоставляет методы для произвольного чтения и записи простых типов, структур и массивов (более подробно об этом речь пойдет в разделе “Работа с аксессорами представлений” далее в главе).

Приведенный ниже код создает файл с одним миллионом байтов и затем использует API-интерфейс размещенных в памяти файлов для чтения и записи байта в позиции 500 000:

```
File.WriteAllBytes ("long.bin", new byte [1000000]);
using (MemoryMappedFile mmf = MemoryMappedFile.CreateFromFile ("long.bin"));
using (MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor ());
accessor.Write (500000, (byte) 77);
Console.WriteLine (accessor.ReadByte (500000)); // 77
```

При вызове метода `CreateFromFile` можно также задавать имя размещенного в памяти файла и емкость. Указание отличающегося от `null` имени позволяет разделять блок памяти с другими процессами (как описано в следующем разделе); указание емкости автоматически увеличивает файл до такого значения. Вот как создать файл из 1000 байтов:

```
File.WriteAllBytes ("short.bin", new byte [1]);
using (var mmf = MemoryMappedFile.CreateFromFile
    ("short.bin", FileMode.Create, null, 1000))
...
```

Размещенные в памяти файлы и совместно используемая память (Windows)

В среде Windows размещенные в памяти файлы можно также применять в качестве средства для совместного использования памяти между процессами, которые функционируют на одном компьютере. Один из процессов создает блок такой памяти, вызывая `MemoryMappedFile.CreateNew`, и затем другие процессы подписываются на этот блок памяти, вызывая метод `MemoryMappedFile.OpenExisting` с тем же именем. Хотя на данный блок по-прежнему ссылаются как на размещенный в памяти “файл”, он располагается полностью в памяти и не имеет никаких представлений на диске.

Следующий код создает размещенный в памяти совместно используемый файл из 500 байтов и записывает целочисленное значение 12345 в позицию 0:

```
using (MemoryMappedFile mmFile = MemoryMappedFile.CreateNew ("Demo", 500))
using (MemoryMappedViewAccessor accessor = mmFile.CreateViewAccessor ())
{
    accessor.Write (0, 12345);
    Console.ReadLine(); // Сохранить совместно используемую память действующей
                        // вплоть до нажатия пользователем <Enter>
}
```

Показанный ниже код открывает тот же самый размещенный в памяти файл и читает из него упомянутое целочисленное значение:

```
// Этот код может быть запущен в отдельном исполняемом файле:
using (MemoryMappedFile mmFile = MemoryMappedFile.OpenExisting ("Demo"))
using (MemoryMappedViewAccessor accessor = mmFile.CreateViewAccessor ())
    Console.WriteLine (accessor.ReadInt32 (0)); // 12345
```

Межплатформенная память, совместно используемая процессами

ОС Windows и Unix позволяют множеству процессов размещать в памяти тот же самый файл. Вы должны позаботиться о соответствующих настройках общего доступа к файлу:

```
static void Writer()
{
    var file = Path.Combine (TestDirectory, "interprocess.bin");
    File.WriteAllBytes (file, new byte [100]);

    using FileStream fs =
        new FileStream (file, FileMode.Open, FileAccess.ReadWrite,
                        FileShare.ReadWrite);

    using MemoryMappedFile mmf = MemoryMappedFile
        .CreateFromFile (fs, null, fs.Length, MemoryMappedFileAccess.ReadWrite,
                         HandleInheritability.None, true);
    using MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor();
    accessor.Write (0, 12345);

    Console.ReadLine(); // Сохранить совместно используемую память действующей
                        // вплоть до нажатия пользователем <Enter>

    File.Delete (file);
}

static void Reader()
{
    // Этот код может быть запущен в отдельном исполняемом файле:
    var file = Path.Combine (TestDirectory, "interprocess.bin");
    using FileStream fs =
        new FileStream (file, FileMode.Open, FileAccess.ReadWrite,
                        FileShare.ReadWrite);
    using MemoryMappedFile mmf = MemoryMappedFile
        .CreateFromFile (fs, null, fs.Length, MemoryMappedFileAccess.ReadWrite,
                         HandleInheritability.None, true);
    using MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor();
    Console.WriteLine (accessor.ReadInt32 (0)); // 12345
}
static string TestDirectory =>
    RuntimeInformation.IsOSPlatform (OSPlatform.Windows)
    ? @"C:\Test"
    : "/tmp";
```

Работа с аксессорами представлений

Вызов метода `CreateViewAccessor` на экземпляре `MemoryMappedFile` дает в результате аксессор представления, который позволяет выполнять чтение и запись в произвольные позиции.

Методы `Read*/Write*` принимают числовые типы, `bool` и `char`, а также массивы и структуры, которые содержат элементы или поля типов значений. Ссыпочные типы — и содержащие ссылочные типы массивы либо структуры — запрещены, поскольку они не могут отображаться на неуправляемую память.

Таким образом, чтобы записать строку, ее потребуется закодировать в массив байтов:

```
byte[] data = Encoding.UTF8.GetBytes ("This is a test");
accessor.Write (0, data.Length);
accessor.WriteArray (4, data, 0, data.Length);
```

Обратите внимание, что первой записывается длина; позже это позволит выяснить, сколько байтов необходимо прочитать:

```
byte[] data = new byte [accessor.ReadInt32 (0)];
accessor.ReadArray (4, data, 0, data.Length);
Console.WriteLine (Encoding.UTF8.GetString (data)); // Выводит This is a test
```

Ниже приведен пример чтения/записи структуры:

```
struct Data { public int X, Y; }
...
var data = new Data { X = 123, Y = 456 };
accessor.Write (0, ref data);
accessor.Read (0, out data);
Console.WriteLine (data.X + " " + data.Y); // Выводит 123 456
```

Методы Read и Write работают на удивление медленно. Намного лучшей производительности можно добиться, напрямую получая доступ к неуправляемой памяти через указатель. Следующий код продолжает предыдущий пример:

```
unsafe
{
    byte* pointer = null;
    try
    {
        accessor.SafeMemoryMappedViewHandle.AcquirePointer (ref pointer);
        int* intPointer = (int*) pointer;
        Console.WriteLine (*intPointer); // 123
    }
    finally
    {
        if (pointer != null)
            accessor.SafeMemoryMappedViewHandle.ReleasePointer();
    }
}
```

Ваш проект должен быть сконфигурирован так, чтобы разрешать небезопасный код. Для этого отредактируйте файл .csproj:

```
<PropertyGroup>
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

Преимущество указателей в плане производительности проявляется еще ярче при работе с крупными структурами, т.к. они позволяют иметь дело непосредственно с низкоуровневыми данными, а не использовать методы Read/Write для копирования данных между управляемой и неуправляемой памятью. Это будет подробно рассматриваться в главе 24.



Взаимодействие с сетью

.NET предлагает в пространствах имен `System.Net.*` множество классов, предназначенных для организации взаимодействия через стандартные сетевые протоколы, такие как HTTP и TCP/IP. Ниже приведен краткий перечень основных компонентов:

- класс `HttpClient` для работы с API-интерфейсами HTTP и веб-службами REST;
- класс `HttpListener` для реализации HTTP-сервера;
- класс `SmtpClient` для формирования и отправки почтовых сообщений через SMTP;
- класс `Dns` для преобразований между доменными именами и адресами;
- классы `TcpClient`, `UdpClient`, `TcpListener` и `Socket` для прямого доступа к транспортному и сетевому уровням.

Типы .NET, рассматриваемые в данной главе, находятся в пространствах имен `System.Net.*` и `System.IO`.



.NET также обеспечивает поддержку FTP на стороне клиента, но только через классы, которые помечены как устаревшие, начиная с версии .NET 6. Если вам нужно использовать FTP, то лучше всего задействовать библиотеку NuGet, подобную FluentFTP.

Сетевая архитектура

На рис. 16.1 показаны типы .NET для работы с сетью и коммуникационные уровни, к которым они относятся. Большинство типов взаимодействуют с *транспортным уровнем* или с *прикладным уровнем*. Транспортный уровень определяет базовые протоколы для отправки и получения байтов (TCP и UDP), а прикладной уровень — высокоуровневые протоколы, предназначенные для конкретных применений, таких как извлечение веб-страниц (HTTP), отправка сообщений электронной почты (SMTP) и преобразование между доменными именами и IP-адресами (DNS).

Прикладной уровень

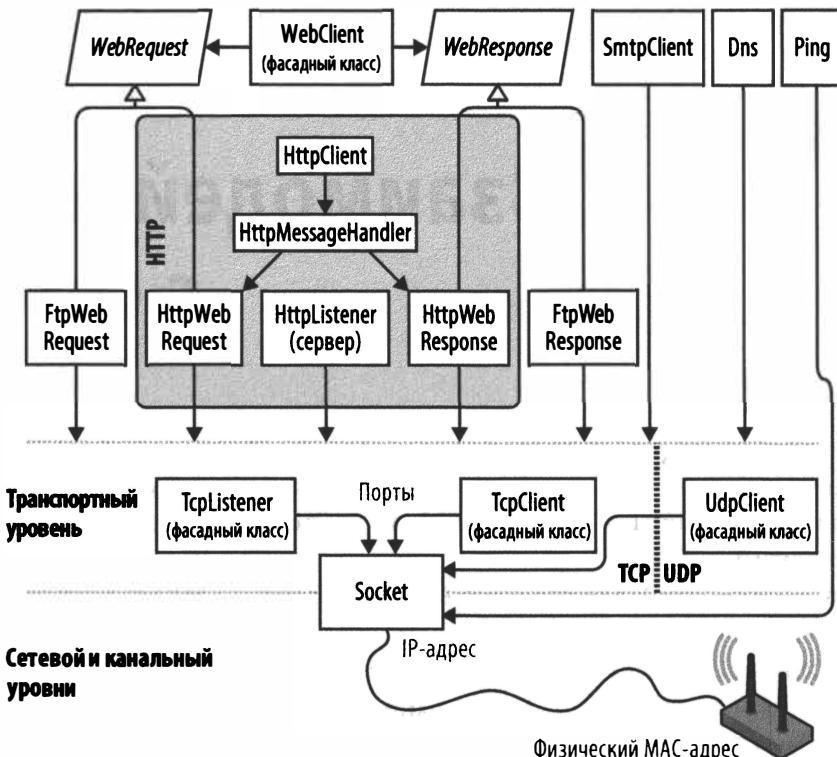


Рис. 16.1. Сетевая архитектура

Обычно удобнее всего программировать на прикладном уровне; тем не менее, есть пара причин, по которым может требоваться работа непосредственно на транспортном уровне. Одна из них связана с необходимостью взаимодействия с прикладным протоколом, не предоставляемым .NET, таким как POP3 для извлечения сообщений электронной почты. Другая причина касается реализации нестандартного протокола для специального приложения, подобного клиенту одноранговой сети.

Внутри набора прикладных протоколов особенность протокола HTTP заключается в том, что он применим к универсальным коммуникациям. Его основной режим работы — “предоставьте мне веб-страницу по заданному URL” — хорошо приспосабливается к варианту “предоставьте мне результат обращения к этой конечной точке с заданными аргументами”. (В дополнение к команде GET имеются команды PUT, POST и DELETE, делая возможными веб-службы на основе REST.)

Протокол HTTP также располагает богатым набором средств, которые полезны в многоуровневых бизнес-приложениях и архитектурах, ориентированных на службы. В их число входят протоколы для аутентификации и шифрования, разбиение сообщений на части, расширяемые заголовки и cookie-наборы, а также возможность совместного использования единственного порта и IP-адреса несколькими серверными приложениями. По этим причинам протокол HTTP

широко поддерживается в .NET — и напрямую, как описано в текущей главе, и на более высоком уровне через такие технологии, как Web API и ASP.NET Core.

Из предыдущего обсуждения должно быть ясно, что работа в сети представляет собой область, которая изобилует аббревиатурами. Самые распространенные аббревиатуры объясняются в табл. 16.1.

Таблица 16.1. Аббревиатуры, связанные с сетью

Аббревиатура	Расшифровка	Примечания
DNS	Domain Name Service (служба доменных имен)	Выполняет преобразования между доменными именами (скажем, ebay.com) и IP-адресами (например, 199.54.213.2)
FTP	File Transfer Protocol (протокол передачи файлов)	Используемый в Интернете протокол для отправки и получения файлов
HTTP	Hypertext Transfer Protocol (протокол передачи гипертекста)	Извлекает веб-страницы и запускает веб-службы
IIS	Internet Information Services (информационные службы Интернета)	Программное обеспечение веб-сервера производства Microsoft
IP	Internet Protocol (протокол Интернета)	Протокол сетевого уровня, находящийся ниже TCP и UDP
LAN	Local Area Network (локальная вычислительная сеть)	Большинство локальных вычислительных сетей применяют основанные на Интернете протоколы, такие как TCP/IP
POP	Post Office Protocol (протокол почтового офиса)	Извлекает сообщения электронной почты Интернета
REST	REpresentational State Transfer (передача состояния представления)	Популярная архитектура веб-служб, которая использует ссылки в ответах и может работать поверх базового протокола HTTP
SMTP	Simple Mail Transfer Protocol (простой протокол передачи почты)	Отправляет сообщения электронной почты Интернета
TCP	Transmission and Control Protocol (протокол управления передачей)	Интернет-протокол транспортного уровня, поверх которого построено большинство служб более высокого уровня
UDP	Universal Datagram Protocol (универсальный протокол передачи дейтаграмм)	Интернет-протокол транспортного уровня, применяемый для служб с низкими накладными расходами, таких как VoIP
UNC	Universal Naming Convention (соглашение об универсальном назначении имен)	\\\компьютер\имя_общего_ресурса\имя_файла
URI	Uniform Resource Identifier (универсальный идентификатор ресурса)	Вездесущая система именования ресурсов (например, http://www.amazon.com или mailto:joe@bloggs.org)
URL	Uniform Resource Locator (унифицированный указатель ресурса)	Формальный смысл (используется редко): подмножество URI; популярный смысл: синоним URI