

Адреса и порты

Для функционирования коммуникаций компьютер или устройство должно иметь адрес. В Интернете применяются две системы адресации.

- **IPv4.** В настоящее время является доминирующей системой адресации; адреса IPv4 имеют ширину 32 бита. В строковом формате адреса IPv4 записываются в виде четырех десятичных чисел, разделенных точками (например, 101.102.103.104). Адрес может быть уникальным в мире или внутри отдельной подсети (такой как корпоративная сеть).
- **IPv6.** Более новая система 128-битной адресации. В строковом формате адреса IPv6 записываются в виде шестнадцатеричных чисел, разделенных двоеточиями (например, [3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]). В .NET адреса должны быть помещены в квадратные скобки.

Класс `IPAddress` из пространства имен `System.Net` представляет адрес в обоих протоколах. Он имеет конструктор, принимающий байтовый массив, и статический метод `Parse`, который принимает корректно сформированную строку:

```
IPAddress a1 = new IPAddress (new byte[] { 101, 102, 103, 104 });
IPAddress a2 = IPAddress.Parse ("101.102.103.104");
Console.WriteLine (a1.Equals (a2));           // True
Console.WriteLine (a1.AddressFamily);        // InterNetwork

IPAddress a3 = IPAddress.Parse
    ("[3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]");
Console.WriteLine (a3.AddressFamily);        // InterNetworkV6
```

Протоколы TCP и UDP рассредоточивают каждый IP-адрес на 65 535 портов, позволяя компьютеру с единственным адресом запускать множество приложений, каждое на своем порту. Многие приложения имеют стандартные назначения портов; скажем, протокол HTTP использует порт 80, а SMTP — порт 25.



Порты TCP и UDP с номерами от 49152 до 65535 официально свободны, поэтому они хорошо подходят для тестирования и небольших развертываний.

Комбинация IP-адреса и порта представлена в .NET классом `IPEndPoint`:

```
IPAddress a = IPAddress.Parse ("101.102.103.104");
IPEndPoint ep = new IPPEndPoint (a, 222);    // Порт 222
Console.WriteLine (ep.ToString());            // 101.102.103.104:222
```



Брандмауэры блокируют порты. Во многих корпоративных средах открыто лишь несколько портов — обычно порт 80 (для нешифрованного HTTP) и порт 443 (для защищенного HTTP).

Идентификаторы URI

Идентификатор URI представляет собой особым образом сформированную строку, которая описывает ресурс в Интернете или локальной сети, такой как веб-страница, файл или адрес электронной почты. Примерами могут служить `http://www.ietf.org`, `ftp://myisp/doc.txt` и `mailto:joe@bloggs.com`. Точный формат определен IETF (Internet Engineering Task Force — инженерная группа по развитию Интернета; `http://www.ietf.org/`).

Идентификатор URI может быть разбит на последовательность элементов, обычно включающую *схему, источник и путь*. Такое разделение осуществляется классом `Uri` из пространства имен `System`, в котором определены свойства для всех упомянутых элементов. На рис. 16.2 приведена соответствующая иллюстрация.

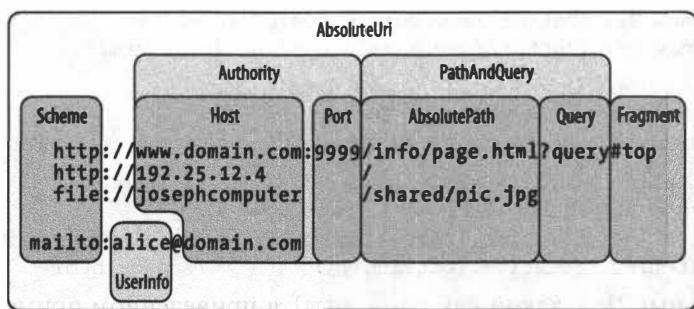


Рис. 16.2. Свойства класса Uri



Класс `Uri` полезен, когда нужно проверить правильность формата строки URI или разбить URI на компоненты. По-другому URI можно трактовать просто как строку — большинство методов, работающих с сетью, перегружены для приема либо объекта `Uri`, либо строки.

Для создания объекта `Uri` конструктору можно передать любую из перечисленных ниже строк:

- строка URI, такая как `http://www.ebay.com` или `file:///janespc/sharedpics/dolphin.jpg`;
- абсолютный путь к файлу на жестком диске вроде `c:\myfiles\data.xlsx` или `/tmp/myfiles/data.xlsx` в Unix;
- путь UNC к файлу в локальной сети наподобие `\janespc\sharedpics\dolphin.jpg`.

Путь к файлу и путь UNC автоматически преобразуются в идентификаторы URI: добавляется протокол `file:`, а символы обратной косой черты заменяются символами обычной косой черты. Перед созданием объекта `Uri` конструкторы класса `Uri` также выполняют некоторую базовую очистку строки, включая приведение схемы и имени хоста к нижнему регистру и удаление стандартных и пустых номеров портов. В случае указания строки URI без схемы, скажем, `www.test.com`, генерируется исключение `UriFormatException`.

Класс Uri имеет свойство IsLoopback, которое указывает, ссылается ли Uri на локальный хост (с IP-адресом 127.0.0.1), и свойство IsFile, которое указывает, ссылается ли Uri на локальный путь или путь UNC (IsUnc); свойство IsUnc возвращает false для общего ресурса Samba, смонтированного в файловой системе Linux. Если IsFile равно true, тогда свойство LocalPath возвращает версию AbsolutePath, которая является дружественной к локальной ОС (с символами прямой или обратной косой черты, принятой в ОС) и допускает вызов метода File.Open.

Экземпляры Uri имеют свойства, предназначенные только для чтения. Чтобы модифицировать существующий Uri, необходимо создать объект UriBuilder — он имеет записываемые свойства и может быть преобразован обратно через свойство Uri.

Класс Uri также предоставляет методы для сравнения и вычитания путей:

```
Uri info = new Uri ("http://www.domain.com:80/info/");
Uri page = new Uri ("http://www.domain.com/info/page.html");
Console.WriteLine (info.Host); // www.domain.com
Console.WriteLine (info.Port); // 80
Console.WriteLine (page.Port); // 80 (классу Uri известен стандартный порт HTTP)
Console.WriteLine (info.IsBaseOf (page)); // True
Uri relative = info.MakeRelativeUri (page);
Console.WriteLine (relative.IsAbsoluteUri); // False
Console.WriteLine (relative.ToString()); // page.html
```

Относительный Uri, такой как page.html в приведенном примере, сгенерирует исключение, если будет вызвано любое другое свойство или метод кроме IsAbsoluteUri и ToString. Создать относительный Uri напрямую можно так:

```
Uri u = new Uri ("page.html", UriKind.Relative);
```



Завершающий символ косой черты в URI важен и вносит отличие в то, каким образом сервер обрабатывает запрос, если присутствует компонент пути.

Например, на традиционном веб-сервере для URI вида `http://www.albahari.com/nutshell/` можно ожидать, что веб-сервер HTTP будет искать подкаталог nutshell в веб-папке сайта и возвратит стандартный документ (обычно `index.html`).

Когда завершающий символ обратной косой черты отсутствует, веб-сервер будет искать файл по имени `nutshell` (без расширения) прямо в корневой папке сайта — обычно это не то, что нужно. Если такой файл не существует, то большинство веб-серверов будут считать, что пользователь допустил опечатку и возвратят ошибку 301 Permanent Redirect (постоянное перенаправление), предлагая клиенту повторить попытку с завершающим символом обратной косой черты. По умолчанию HTTP-клиент .NET будет прозрачно реагировать на ошибку 301 тем же способом, что и веб-браузер — повторяя попытку с предложенным URI. Это значит, что если вы опустите завершающий символ обратной косой черты, когда он должен быть включен, то запрос все равно будет работать, но потребует излишнего двухстороннего обмена.

Класс Uri также предлагает статические вспомогательные методы вроде EscapeUriString, который преобразует строку в допустимый URL, заменяя все символы с ASCII-кодами больше 127 их шестнадцатеричными представлениями. Методы CheckHostName и CheckSchemeName принимают строку и проверяют, является ли она синтаксически правильной для заданного свойства (хотя они не пытаются определить, существует ли указанный хост или URI).

HttpClient

Класс HttpClient предоставляет современный API-интерфейс для операций HTTP-клиента, заменяя старые типы WebClient и WebRequest/WebResponse (которые с тех пор были помечены как устаревшие).

Класс HttpClient был реализован в ответ на развитие API-интерфейсов, предназначенных для взаимодействия с протоколом HTTP и веб-службами REST, чтобы предложить улучшенный стиль работы с протоколами, который выходит за рамки простого извлечения веб-страниц. Ниже описаны основные особенности класса HttpClient.

- Одиночный экземпляр HttpClient поддерживает параллельные запросы и хорошо работает с такими средствами, как специальные заголовки, cookie-наборы и схемы аутентификации.
- Класс HttpClient позволяет создавать и подключать специальные обработчики сообщений. Это делает возможными имитацию для модульного тестирования и построение специальных конвейеров (с целью регистрации в журнале, сжатия, шифрования и т.д.).
- Класс HttpClient имеет развитую и расширяемую систему типов для заголовков и содержимого.



Класс HttpClient не поддерживает сообщение о ходе работ. С решением для сообщения о ходе работ посредством HttpClient можно ознакомиться в файле HttpClient with Progress.linq по ссылке <http://www.albahari.com/nutshell/code/aspx> или в галерее интерактивных примеров LINQPad.

Простейший способ применения класса HttpClient предусматривает создание его экземпляра и вызов одного из методов Get* с передачей ему URI:

```
string html = await new HttpClient().GetStringAsync ("http://linqpad.net");
```

(Доступны также методы GetByteArrayAsync и GetStreamAsync.) Все методы с интенсивным вводом-выводом в классе HttpClient являются асинхронными.

В отличие от своих предшественников WebRequest/WebResponse для достижения более высокой производительности при работе с HttpClient вы должны повторно использовать тот же самый экземпляр (иначе могут повторяться излишние действия вроде распознавания DNS, к тому же сокеты останутся открытыми дольше, чем необходимо). Класс HttpClient разрешает параллельные

операции, поэтому следующий код допустим; в нем загружаются две веб-страницы за раз:

```
var client = new HttpClient();
var task1 = client.GetStringAsync ("http://www.linqpad.net");
var task2 = client.GetStringAsync ("http://www.albahari.com");
Console.WriteLine (await task1);
Console.WriteLine (await task2);
```

В классе `HttpClient` имеется свойство `Timeout` и свойство `BaseAddress`, которое добавляется в качестве префикса к URI каждого запроса. В определенной степени класс `HttpClient` является тонкой оболочкой: большинство других свойств, которые можно в нем обнаружить, определены в другом классе по имени `HttpClientHandler`. Для доступа к этому классу необходимо создать и передать его экземпляр конструктору класса `HttpClient`:

```
var handler = new HttpClientHandler { UseProxy = false };
var client = new HttpClient (handler);
...
```

В показанном примере мы сообщаем обработчику о необходимости отключения поддержки прокси-сервера, что иногда может приводить к увеличению производительности за счет устранения накладных расходов, связанных с автоматическим обнаружением прокси-сервера. Предусмотрены также свойства для управления cookie-наборами, автоматическим перенаправлением, аутентификацией и т.д. (мы рассмотрим их в последующих разделах).

Метод `GetAsync` и сообщения ответов

Методы `GetStringAsync`, `GetByteArrayAsync` и `GetStreamAsync` представляют собой удобные сокращения для вызова более общего метода `GetAsync`, который возвращает *сообщение ответа*:

```
var client = new HttpClient();
// Метод GetAsync также принимает объект CancellationToken.
HttpResponseMessage response = await client.GetAsync ("http://...");  
response.EnsureSuccessStatusCode();
string html = await response.Content.ReadAsStringAsync();
```

Класс `HttpResponseMessage` имеет свойства для доступа к заголовкам (см. раздел “Заголовки” далее в главе) и коду состояния HTTP (`StatusCodes`). Код состояния неудачи, такой как 404 (не найдено), не приводит к генерации исключения, если только не будет явно вызван метод `EnsureSuccessStatusCode`. Тем не менее, ошибки коммуникаций или DNS вызывают генерацию исключений.

Класс `HttpResponseMessage` располагает методом `CopyToAsync` для записи в другой поток, который удобен для сохранения вывода в файле:

```
using (var fileStream = File.Create ("linqpad.html"))
await response.Content.CopyToAsync (fileStream);
```

`GetAsync` является одним из четырех методов, соответствующих четырем командам HTTP (остальные методы — это `PostAsync`, `PutAsync` и `DeleteAsync`). Мы продемонстрируем применение метода `PostAsync` в разделе “Выгрузка данных формы” далее в главе.

Метод `SendAsync` и сообщения запросов

Методы `GetAsync`, `PostAsync`, `PutAsync` и `DeleteAsync` выступают в качестве сокращений для вызова метода `SendAsync` — единственного низкоуровневого метода, который делает всю работу. Сначала конструируется объект `HttpRequestMessage`:

```
var client = new HttpClient();
var request = new HttpRequestMessage (HttpMethod.Get, "http://...");  
HttpResponseMessage response = await client.SendAsync (request);  
response.EnsureSuccessStatusCode();  
...
```

Создание объекта `HttpRequestMessage` означает возможность настройки свойств запроса, таких как заголовки (см. раздел “Заголовки” далее в главе), и самого содержимого, позволяя выгружать данные.

Выгрузка данных и `HttpContent`

После создания объекта `HttpRequestMessage` можно выгружать содержимое, устанавливая его свойство `Content`. Типом этого свойства является абстрактный класс по имени `HttpContent`. В состав .NET входят следующие конкретные подклассы для различных видов содержимого (можно также построить собственный подкласс такого рода):

- `ByteArrayContent`
- `StreamContent`
- `FormUrlEncodedContent` (см. раздел “Выгрузка данных формы” далее в главе)
- `StreamContent`

Например:

```
var client = new HttpClient (new HttpClientHandler { UseProxy = false });
var request = new HttpRequestMessage (
    HttpMethod.Post, "http://www.albahari.com/EchoPost.aspx");
request.Content = new StringContent ("This is a test");
HttpResponseMessage response = await client.SendAsync (request);
response.EnsureSuccessStatusCode();
Console.WriteLine (await response.Content.ReadAsStringAsync());
```

`HttpMessageHandler`

Ранее мы упоминали, что большинство свойств для настройки запросов определено не в `HttpClient`, а в классе `HttpClientHandler`. Последний в действительности представляет собой подкласс абстрактного класса `HttpMessageHandler`, определенного следующим образом:

```
public abstract class HttpMessageHandler : IDisposable
{
    protected internal abstract Task<HttpResponseMessage> SendAsync
        (HttpRequestMessage request, CancellationToken cancellationToken);
    public void Dispose();
    protected virtual void Dispose (bool disposing);
}
```

Метод `SendAsync` вызывается внутри метода `SendAsync` класса `HttpClient`.

Из `HttpMessageHandler` довольно легко создавать подклассы, и он является точкой расширения для `HttpClient`.

Модульное тестирование и имитация

Подкласс `HttpMessageHandler` можно создавать для построения *имитированного обработчика*, который помогает проводить модульное тестирование:

```
class MockHandler : HttpMessageHandler
{
    Func<HttpRequestMessage, HttpResponseMessage> _responseGenerator;
    public MockHandler
        (Func<HttpRequestMessage, HttpResponseMessage> responseGenerator)
    {
        _responseGenerator = responseGenerator;
    }
    protected override Task<HttpResponseMessage> SendAsync
        (HttpRequestMessage request, CancellationToken cancellationToken)
    {
        cancellationToken.ThrowIfCancellationRequested();
        var response = _responseGenerator(request);
        response.RequestMessage = request;
        return Task.FromResult(response);
    }
}
```

Его конструктор принимает функцию, которая сообщает имитированному объекту, каким образом генерировать ответ из запроса. Такой подход наиболее универсален, потому что один и тот же обработчик способен тестировать множество запросов.

По причине вызова `Task.FromResult` метод `SendAsynch` синхронен. Мы могли бы поддерживать асинхронность, обеспечив возвращение генератором ответов объекта типа `Task<HttpResponseMessage>`, но это бессмысленно, учитывая возможность полагаться на то, что имитированная функция должна выполняться быстро. Ниже показано, как использовать наш имитированный обработчик:

```
var mocker = new MockHandler (request =>
    new HttpResponseMessage ( HttpStatusCode.OK )
    {
        Content = new StringContent ("You asked for " + request.RequestUri)
            // Запрошен URI
    });
var client = new HttpClient (mocker);
var response = await client.GetAsync ("http://www.linqpad.net");
string result = await response.Content.ReadAsStringAsync();
Assert.AreEqual ("You asked for http://www.linqpad.net/", result);
```

(`Assert.AreEqual` — метод, который мы ожидаем обнаружить в инфраструктурах модульного тестирования, подобных `NUnit`.)

Соединение обработчиков в цепочки с помощью DelegatingHandler

За счет создания подклассов класса `DelegatingHandler` можно построить обработчик сообщений, который вызывает другой обработчик (давая в результате цепочку обработчиков). Подобный прием может применяться для реализации специальных протоколов аутентификации, сжатия и шифрования. Ниже приведен код простого обработчика регистрации в журнале:

```
class LoggingHandler : DelegatingHandler
{
    public LoggingHandler (HttpMessageHandler nextHandler)
    {
        InnerHandler = nextHandler;
    }

    protected async override Task < HttpResponseMessage > SendAsync
        (HttpRequestMessage request, CancellationToken cancellationToken)
    {
        Console.WriteLine ("Requesting: " + request.RequestUri); // Запрос
        var response = await base.SendAsync (request, cancellationToken);
        Console.WriteLine ("Got response: " + response.StatusCode); // Ответ
        return response;
    }
}
```

Обратите внимание, что в переопределенном методе `SendAsync` поддерживается асинхронность. Введение модификатора `async` при переопределении метода, возвращающего задачу, совершенно допустимо, а в данном случае еще и желательно.

В более удачном решении, нежели простой вывод на консоль, можно было бы предложить конструктор, который принимает регистрирующий объект некоторого вида. А еще лучше было бы принимать пару делегатов `Action<T>`, сообщающих о том, каким образом регистрировать в журнале объекты запроса и ответа.

Прокси-серверы

Прокси-сервер — это посредник, через который могут маршрутизироваться запросы HTTP и FTP. Иногда организации настраивают прокси-сервер как единственное средство, с помощью которого сотрудники могут получать доступ в Интернет — главным образом потому, что это упрощает действия по обеспечению безопасности. Прокси-сервер имеет собственный адрес и может требовать аутентификации, так что доступ в Интернет будет разрешен только избранным пользователям локальной сети.

Чтобы использовать прокси-сервер вместе с `HttpClient`, сначала нужно создать объект `HttpClientHandler`, установить его свойство `Proxy` и затем передать результат конструктору `HttpClient`:

```
WebProxy p = new WebProxy ("192.178.10.49", 808);
p.Credentials = new NetworkCredential("имя-пользователя", "пароль", "домен");
var handler = new HttpClientHandler { Proxy = p };
var client = new HttpClient (handler);
...
```

Класс HttpClientHandler имеет также свойство UseProxy, которое можно установить в false вместо установки в null свойства Proxy для отмены автоматического определения параметров прокси-сервера.

Если при конструировании объекта NetworkCredential указан домен, то будут использоваться протоколы аутентификации на основе Windows. Чтобы задействовать текущего аутентифицированного пользователя Windows, установите статическое свойство CredentialCache.DefaultNetworkCredentials в значение свойства Credentials прокси-сервера.

В качестве альтернативы частой установке свойства Proxy можно установить глобальное стандартное значение:

```
HttpClient.DefaultWebProxy = myWebProxy;
```

Аутентификация

Вот как можно предоставить объекту HttpClient имя пользователя и пароль:

```
string username = "myuser";
string password = "mypassword";

var handler = new HttpClientHandler();
handler.Credentials = new NetworkCredential (username, password);
var client = new HttpClient (handler);
...
```

Данный прием работает с основанными на диалоговых окнах протоколами аутентификации, такими как Basic и Digest, и расширяется посредством класса AuthenticationManager. Он также поддерживает протоколы Windows NTLM и Kerberos (когда при конструировании объекта NetworkCredential указано имя домена). Если нужно использовать текущего аутентифицированного пользователя Windows, тогда свойство Credentials можно оставить равным null и вместо него установить свойство UseDefaultCredentials в true.

После предоставления учетных данных объект HttpClient автоматически согласовывает совместимый протокол. В ряде случаев может существовать выбор: например, если вы просмотрите начальный ответ от страницы веб-почты сервера Microsoft Exchange, то можете встретить следующие заголовки:

```
HTTP/1.1 401 Unauthorized
Content-Length: 83
Content-Type: text/html
Server: Microsoft-IIS/6.0
WWW-Authenticate: Negotiate
WWW-Authenticate: NTLM
WWW-Authenticate: Basic realm="exchange.somedomain.com"
X-Powered-By: ASP.NET
Date: Sat, 05 Aug 2006 12:37:23 GMT
```

Код 401 сигнализирует о том, что авторизация обязательна; заголовки WWW-Authenticate указывают, какие будут восприниматься протоколы аутентификации. Однако если сконфигурировать объект HttpClientHandler с корректным именем пользователя и паролем, то это сообщение будет скрыто,

поскольку исполняющая среда реагирует автоматически, выбирая совместимый протокол аутентификации и затем повторно отправляя исходный запрос с дополнительным заголовком. Например:

```
Authorization: Negotiate TlRMTVNTUAAABAAAt5II2gjACDARAAACAwACACgAAAAAQ
ATmKAAAAD01VDRdPUksHUq9VUA==
```

Такой механизм отличается прозрачностью, но приводит к дополнительному двухстороннему обмену для каждого запроса. Установив свойство `PreAuthenticate` объекта `HttpClientHandler` в `true`, дополнительных двухсторонних обменов при последующих запросах к тому же самому URI можно избежать.

CredentialCache

С помощью объекта `CredentialCache` можно принудительно применить конкретный протокол аутентификации. Кеш учетных данных (credential cache) содержит один или большее количество объектов `NetworkCredential`, каждый из которых связан с конкретным протоколом и префиксом URI. Например, во время входа на сервер Exchange Server может возникнуть желание пропустить протокол Basic, т.к. он передает пароли в виде открытого текста:

```
CredentialCache cache = new CredentialCache();
Uri prefix = new Uri ("http://exchange.somedomain.com");
cache.Add (prefix, "Digest", new NetworkCredential ("joe", "passwd"));
cache.Add (prefix, "Negotiate", new NetworkCredential ("joe", "passwd"));

var handler = new HttpClientHandler();
handler.Credentials = cache;
...
```

Протокол аутентификации указывается в форме строки со следующими допустимыми значениями:

```
Basic, Digest, NTLM, Kerberos, Negotiate
```

В этой конкретной ситуации будет выбрано значение `Negotiate`, потому что сервер не сообщил о поддержке протокола `Digest` в своих заголовках аутентификации. Здесь `Negotiate` представляет собой протокол Windows, который в зависимости от возможностей сервера сводится к `Kerberos` или `NTLM`, но гарантирует вашему приложению прямую совместимость в случае развертывания будущих стандартов.

Статическое свойство `CredentialCache.DefaultNetworkCredentials` позволяет добавлять текущего аутентифицированного пользователя Windows в кеш учетных данных без необходимости в предоставлении пароля:

```
cache.Add (prefix, "Negotiate", CredentialCache.DefaultNetworkCredentials);
```

Аутентификация через заголовки

Есть еще один способ аутентификации, предусматривающий установку заголовка аутентификации напрямую:

```
var client = new HttpClient();
client.DefaultRequestHeaders.Authorization =
    new AuthenticationHeaderValue ("Basic",
        Convert.ToBase64String (Encoding.UTF8.GetBytes ("username:password")));
...
```

Такая стратегия работает и со специальными системами аутентификации вроде OAuth.

Заголовки

Класс HttpClient позволяет добавлять в запрос специальные HTTP-заголовки, а также производить перечисление имеющихся заголовков в ответе. Заголовок представляет собой просто пару “ключ/значение” с метаданными, такими как тип содержимого сообщения или программное обеспечение сервера. Класс HttpClient открывает доступ к строго типизированным коллекциям со свойствами для стандартных HTTP-заголовков. Свойство DefaultRequestHeaders предназначено для заголовков, которые применяются к каждому запросу:

```
var client = new HttpClient (handler);
client.DefaultRequestHeaders.UserAgent.Add (
    new ProductInfoHeaderValue ("VisualStudio", "2015"));
client.DefaultRequestHeaders.Add ("CustomHeader", "VisualStudio/2015");
```

С другой стороны, свойство Headers класса HttpRequestMessage представляет заголовки, специфичные для запроса.

Строки запросов

Строка запроса — это просто добавляемая к URI строка со знаком вопроса, которая используется для отправки простых данных серверу. В строке запроса можно указывать множество пар “ключ/значение” с применением следующего синтаксиса:

```
?key1=value1&key2=value2&key3=value3...
```

Вот URI со строкой запроса:

```
string requestURI = "http://www.google.com/search?q=HttpClient&hl=fr";
```

Если существует вероятность того, что запрос будет включать нестандартные символы или пробелы, то для создания допустимого URI можно задействовать метод EscapeDataString класса Uri:

```
string search = Uri.EscapeDataString ("(HttpClient or
HttpRequestMessage)");
string language = Uri.EscapeDataString ("fr");
string requestURI = "http://www.google.com/search?q=" + search +
    "&hl=" + language;
```

Результирующий URI выглядит так:

```
http://www.google.com/search?q=(HttpClient%20OR%20HttpRequestMessage)&hl=fr
```

(Метод `EscapeDataString` похож на `EscapeUriString` за исключением того, что он также кодирует символы вроде & и =, которые иначе заполонили бы строку запроса.)

Выгрузка данных формы

Чтобы загрузить данные HTML-формы, создайте и заполните объект `FormUrlEncodedContent`. Затем можете либо передать его в метод `PostAsync`, либо присвоить свойству `Content` запроса:

```
string uri = "http://www.albahari.com/EchoPost.aspx";
var client = new HttpClient();
var dict = new Dictionary<string, string>
{
    { "Name", "Joe Albahari" },
    { "Company", "O'Reilly" }
};
var values = new FormUrlEncodedContent (dict);
var response = await client.PostAsync (uri, values);
response.EnsureSuccessStatusCode();
Console.WriteLine (await response.Content.ReadAsStringAsync());
```

Cookie-наборы

Cookie-набор — это строка с парой “имя/значение”, которую HTTP-сервер посылает клиенту в заголовке ответа. Клиентский веб-браузер обычно запоминает cookie-наборы и повторяет их для сервера в каждом последующем запросе (к тому же адресу) до тех пор, пока не истечет время их действия. Cookie-набор позволяет серверу знать, что он взаимодействует с тем же самым клиентом, с которым он имел дело минуту назад (или, скажем, вчера), без необходимости в наличии неаккуратной строки запроса в URI.

По умолчанию `HttpClient` игнорирует любые cookie-наборы, полученные от сервера. Чтобы принимать cookie-наборы, следует создать объект `CookieContainer` и присвоить его свойству `CookieContainer` объекта `HttpClientHandler`:

```
var cc = new CookieContainer();
var handler = new HttpClientHandler();
handler.CookieContainer = cc;
var client = new HttpClient (handler);
...
```

Для повторения полученных cookie-наборов в будущих запросах необходимо просто использовать тот же самый объект `CookieContainer`. В качестве альтернативы можно начать с нового объекта `CookieContainer` и затем добавлять cookie-наборы вручную:

```
Cookie c = new Cookie ("PREF",
                      "ID=6b10df1da493a9c4:TM=1179...",
                      "/",
                      ".google.com");
freshCookieContainer.Add (c);
```

Третий и четвертый аргументы отражают путь и домен источника. Объект `CookieContainer` на стороне клиента может хранить cookie-наборы из множества разных мест; объект `HttpClient` отправляет только те cookie-наборы, путь и домен которых совпадают с путем и доменом сервера.

Реализация HTTP-сервера



Если вам нужно реализовать HTTP-сервер, то альтернативным подходом более высокого уровня (из .NET 6) является использование минимального API-интерфейса ASP.NET. Вот все, что понадобится для начала:

```
var app = WebApplication.CreateBuilder().Build();
app.MapGet("/", () => "Hello, world!");
app.Run();
```

С помощью класса `HttpListener` можно создать собственный HTTP-сервер .NET. Ниже приведен код простого сервера, который прослушивает порт 51111, ожидает одиночный клиентский запрос и возвращает односторонний ответ.

```
using var server = new SimpleHttpServer();

// Сделать клиентский запрос:
Console.WriteLine (await new HttpClient().GetStringAsync
    ("http://localhost:51111/MyApp/Request.txt"));

class SimpleHttpServer : IDisposable
{
    readonly HttpListener listener = new HttpListener();

    public SimpleHttpServer() => ListenAsync();
    async void ListenAsync()
    {
        listener.Prefixes.Add ("http://localhost:51111/MyApp/"); // Прослушивать
        listener.Start(); // порт 51111

        // Ожидать поступления клиентского запроса:
        HttpListenerContext context = await listener.GetContextAsync();

        // Ответить на запрос:
        string msg = "You asked for: " + context.Request.RawUrl;
        context.Response.ContentLength64 = Encoding.UTF8.GetByteCount (msg);
        context.Response.StatusCode = (int) HttpStatusCode.OK;

        using (Stream s = context.Response.OutputStream)
        using (StreamWriter writer = new StreamWriter (s))
            await writer.WriteAsync (msg);
    }

    public void Dispose() => listener.Close();
}
```

Вывод выглядит так:

```
You asked for: /MyApp/Request.txt
```

В среде Windows класс `HttpListener` внутренне не использует .NET-объекты `Socket`; взамен он обращается к API-интерфейсу HTTP-серверов Windows (Windows HTTP Server API). Это позволяет множеству приложений на компьютере прослушивать один и тот же IP-адрес и порт — при условии, что каждое из них регистрирует отличающиеся адресные префиксы. В показанном выше примере мы регистрируем префикс `http://localhost/туаpp`, поэтому другое приложение способно прослушивать тот же самый IP-адрес и порт с другим префиксом, таким как `http://localhost/anotherapp`. Это имеет значение, потому что открытие новых портов в корпоративных брандмауэрах может быть затруднено из-за политик, действующих внутри компании.

Объект `HttpListener` ожидает следующего клиентского запроса, когда вызывается метод `GetContext`, возвращая объект со свойствами `Request` и `Response`. Указанные свойства аналогичны клиентскому запросу или ответу, но со стороны сервера. Например, можно читать и записывать заголовки и cookie-наборы для объектов запросов и ответов почти так же, как это делается на стороне клиента.

Основываясь на ожидаемой клиентской аудитории, можно выбрать полностью, с которой будут поддерживаться функциональные возможности протокола HTTP. В качестве самого минимума для каждого запроса должны устанавливаться длина содержимого и код состояния.

Ниже представлен код простого сервера веб-страниц, реализованного *асинхронным образом*:

```
using System;
using System.IO;
using System.Net;
using System.Text;
using System.Threading.Tasks;
class WebServer
{
    HttpListener _listener;
    string _baseFolder; // Папка для веб-страниц.
    public WebServer (string uriPrefix, string baseFolder)
    {
        _listener = new HttpListener();
        _listener.Prefixes.Add (uriPrefix);
        _baseFolder = baseFolder;
    }
    public async void Start()
    {
        _listener.Start();
        while (true)
            try
            {
                var context = await _listener.GetContextAsync();
                Task.Run (() => ProcessRequestAsync (context));
            }
            catch (HttpListenerException) { break; } // Прослушиватель
                                                // остановлен
            catch (InvalidOperationException) { break; } // Прослушиватель
                                                // остановлен
    }
}
```

```

public void Stop() => _listener.Stop();
async void ProcessRequestAsync (HttpListenerContext context)
{
    try
    {
        string filename = Path.GetFileName (context.Request.RawUrl);
        string path = Path.Combine (_baseFolder, filename);
        byte[] msg;
        if (!File.Exists (path))
        {
            Console.WriteLine ("Resource not found: " + path); // Ресурс
                                                       // не найден
            context.Response.StatusCode = (int) HttpStatusCode.NotFound;
            msg = Encoding.UTF8.GetBytes ("Sorry, that page does not exist");
                                                       // Страница не существует
        }
        else
        {
            context.Response.StatusCode = (int) HttpStatusCode.OK;
            msg = File.ReadAllBytes (path);
        }
        context.Response.ContentLength64 = msg.Length;
        using (Stream s = context.Response.OutputStream)
            await s.WriteAsync (msg, 0, msg.Length);
    }
    catch (Exception ex) { Console.WriteLine ("Request error: " + ex); }
                         // Ошибка запроса
}
}

```

Следующий код приводит все в действие:

```

// Прослушивать порт 51111, обслуживая файлы в d:\webroot:
var server = new WebServer ("http://localhost:51111/", @"d:\webroot");
try
{
    server.Start();
    Console.WriteLine ("Server running... press Enter to stop");
                           // Сервер выполняется... для его останова нажмите <Enter>
    Console.ReadLine();
}
finally { server.Stop(); }

```

Показанный выше код можно протестировать на стороне клиента с помощью любого браузера; URI в данном случае будет выглядеть как `http://localhost:51111/` плюс имя веб-страницы.



Прослушиватель `HttpListener` не запустится, когда за тот же самый порт соперничает другое программное обеспечение (если только оно тоже не использует Windows HTTP Server API). Примеры приложений, которые могут прослушивать стандартный порт 80, включают веб-сервер и программы для одноранговых сетей, подобные `Skype`.

Применение асинхронных функций делает наш сервер масштабируемым и эффективным. Однако его запуск в потоке пользовательского интерфейса будет препятствовать масштабируемости, т.к. для каждого запроса управление должно возвращаться в поток пользовательского интерфейса после каждого await. Привнесение таких накладных расходов не имеет смысла, особенно с учетом того, что совместно используемое состояние отсутствует, поэтому в сценарии с пользовательским интерфейсом мы могли бы поступить следующим образом:

```
Task.Run (Start);
```

или же вызвать `ConfigureAwait (false)` после вызова `GetContextAsync`.

Обратите внимание, что для вызова метода `ProcessRequestAsync` мы используем `Task.Run` несмотря на то, что упомянутый метод уже является асинхронным. Это позволяет вызывающему коду обработать другой запрос *немедленно* вместо того, чтобы сначала ожидать завершения синхронной фазы метода (вплоть до первого `await`).

Использование DNS

Статический класс `Dns` инкапсулирует службу доменных имен (Domain Name Service — DNS), которая осуществляет преобразования между низкоуровневыми IP-адресами наподобие 66.135.192.87 и понятными для человека доменными именами, такими как `eBay.com`.

Метод `GetHostAddresses` преобразует доменное имя в IP-адрес (или адреса):

```
foreach (IPAddress a in Dns.GetHostAddresses ("albahari.com"))
    Console.WriteLine (a.ToString()); // 205.210.42.167
```

Метод `GetHostEntry` двигается в обратном направлении, преобразуя IP-адрес в доменное имя:

```
IPHostEntry entry = Dns.GetHostEntry ("205.210.42.167");
Console.WriteLine (entry.HostName); // albahari.com
```

Метод `GetHostEntry` также принимает объект `IPAddress`, поэтому IP-адрес можно указывать в виде байтового массива:

```
IPAddress address = new IPAddress (new byte[] { 205, 210, 42, 167 });
IPHostEntry entry = Dns.GetHostEntry (address);
Console.WriteLine (entry.HostName); // albahari.com
```

В случае применения класса вроде `WebRequest` или `TcpClient` доменные имена преобразуются в IP-адреса автоматически. Однако если в приложении планируется выполнять множество сетевых запросов к одному и тому же адресу, то производительность иногда можно увеличить, сначала используя класс `Dns` для явного преобразования доменного имени в IP-адрес и затем организовав с ним коммуникации напрямую. Прием позволяет избежать повторяющихся циклов двухстороннего обмена для преобразования того же самого доменного имени и может быть полезен при работе с транспортным уровнем (через класс `TcpClient`, `UdpClient` или `Socket`).

Класс `Dns` также предлагает асинхронные методы, которые возвращают объекты задач, допускающих ожидание:

```
foreach (IPAddress a in await Dns.GetHostAddressesAsync ("albahari.com"))
    Console.WriteLine (a.ToString());
```

Отправка сообщений электронной почты с помощью SmtpClient

Класс SmtpClient из пространства имен System.Net.Mail позволяет отправлять сообщения электронной почты с применением простого протокола передачи почты (Simple Mail Transfer Protocol — SMTP). Чтобы отправить обычное текстовое сообщение, необходимо создать экземпляр SmtpClient, указать в его свойстве Host адрес SMTP-сервера и затем вызвать метод Send:

```
SmtpClient client = new SmtpClient ();
client.Host = "mail.myserver.com";
client.Send ("from@adomain.com", "to@adomain.com", "subject", "body");
```

При конструировании объекта MailMessage доступны и другие варианты, включая возможность добавления вложений:

```
SmtpClient client = new SmtpClient ();
client.Host = "mail.myserver.com";
MailMessage mm = new MailMessage ();

mm.Sender = new MailAddress ("kay@domain.com", "Kay");
mm.From = new MailAddress ("kay@domain.com", "Kay");
mm.To.Add (new MailAddress ("bob@domain.com", "Bob"));
mm.CC.Add (new MailAddress ("dan@domain.com", "Dan"));
mm.Subject = "Hello!";
mm.Body = "Hi there. Here's the photo!";
mm.IsBodyHtml = false;
mm.Priority = MailPriority.High;

Attachment a = new Attachment ("photo.jpg",
                               System.Net.Mime.MediaTypeNames.Image.Jpeg);
mm.Attachments.Add (a);
client.Send (mm);
```

Чтобы препятствовать рассылке спама, большинство SMTP-серверов в Интернете будут принимать подключения только от аутентифицированных клиентов, а также требовать связи через SSL:

```
var client = new SmtpClient ("smtp.myisp.com", 587)
{
    Credentials = new NetworkCredential ("me@myisp.com", "MySecurePass"),
    EnableSsl = true
};
client.Send ("me@myisp.com", "someone@somewhere.com", "Subject", "Body");
Console.WriteLine ("Sent");
```

Изменяя свойство DeliveryMethod, можно заставить SmtpClient использовать сервер IIS для отправки почтовых сообщений или просто записывать каждое сообщение в файл .eml внутри указанного каталога, что может быть удобно во время разработки:

```
SmtpClient client = new SmtpClient();
client.DeliveryMethod = SmtpDeliveryMethod.SpecifiedPickupDirectory;
client.PickupDirectoryLocation = @"c:\mail";
```

Использование TCP

TCP и UDP являются протоколами транспортного уровня, на основе которых построено большинство служб Интернета и локальных вычислительных сетей. Протоколы HTTP (версии 2 и выше), FTP и SMTP работают с TCP, а DNS и HTTP версии 3 — с UDP. Протокол TCP ориентирован на подключение и поддерживает механизмы обеспечения надежности; UDP является протоколом без установления подключения, характеризуется более низкими накладными расходами и поддерживает широковещательную передачу. Протокол BitTorrent применяет UDP, как и протокол Voice over IP (VoIP).

Транспортный уровень предлагает более высокую гибкость — и потенциально лучшую производительность — по сравнению с более высокими уровнями, но требует самостоятельного решения таких задач, как аутентификация и шифрование.

Благодаря поддержке TCP в .NET можно работать либо с легкими в пользовании фасадными классами TcpClient и TcpListener, либо с обладающим обширными возможностями классом Socket. (В действительности их можно сочетать, поскольку класс TcpClient через свое свойство Client открывает доступ к внутреннему объекту Socket.) Класс Socket предоставляет больше вариантов конфигурации и разрешает прямой доступ к сетевому уровню (IP) и протоколам, не основанным на Интернете, таким как SPX/IPX от Novell.

Как и другие протоколы, TCP различает концепции клиента и сервера: клиент инициирует запрос, а сервер запрос ожидает. Ниже представлена базовая структура для синхронного запроса клиента TCP:

```
using (TcpClient client = new TcpClient())
{
    client.Connect ("address", port);
    using (NetworkStream n = client.GetStream())
    {
        // Выполнять чтение и запись в сетевой поток...
    }
}
```

Метод Connect класса TcpClient блокируется вплоть до установления подключения (его асинхронным эквивалентом является метод ConnectAsync). Затем объект NetworkStream предоставляет средство двухсторонних коммуникаций для передачи и получения байтов данных из сервера.

Простой сервер TCP выглядит следующим образом:

```
TcpListener listener = new TcpListener (<IP-адрес>, port);
listener.Start();
while (keepProcessingRequests)
{
    using (TcpClient c = listener.AcceptTcpClient())
    using (NetworkStream n = c.GetStream())
    {
        // Выполнять чтение и запись в сетевой поток...
    }
}
listener.Stop();
```

Объект TcpListener требует указания локального IP-адреса для прослушивания (например, компьютер с двумя сетевыми адаптерами может иметь два адреса). Чтобы обеспечить прослушивание всех локальных IP-адресов (или только одного из них), можно применять поле IPAddress.Any. Вызов метода AcceptTcpClient класса TcpListener блокируется до тех пор, пока не будет получен клиентский запрос (есть также асинхронная версия этого метода), после чего мы вызываем метод GetStream, в точности как поступали на стороне клиента.

При работе на транспортном уровне необходимо принять решение, касающееся протокола, которое связано с тем, кто и когда передает данные — почти как при использовании портативной радиции. Если оба участника начинают говорить или слушать одновременно, то связь будет нарушена.

Давайте создадим протокол, при котором клиент начинает общение первым, сказав "Hello", после чего сервер отвечает фразой "Hello right back!". Ниже показан соответствующий код:

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;

new Thread (Server).Start(); // Запустить серверный метод параллельно
Thread.Sleep (500);          // Предоставить серверу время для запуска
Client();
void Client()
{
    using (TcpClient client = new TcpClient ("localhost", 51111))
    using (NetworkStream n = client.GetStream())
    {
        BinaryWriter w = new BinaryWriter (n);
        w.Write ("Hello");
        w.Flush();
        Console.WriteLine (new BinaryReader (n).ReadString());
    }
}
void Server() // Обрабатывает одиночный клиентский запрос и завершается
{
    TcpListener listener = new TcpListener (IPAddress.Any, 51111);
    listener.Start();
    using (TcpClient c = listener.AcceptTcpClient())
    using (NetworkStream n = c.GetStream())
    {
        string msg = new BinaryReader (n).ReadString();
        BinaryWriter w = new BinaryWriter (n);
        w.Write (msg + " right back!");
        w.Flush();           // Должен быть вызван метод Flush, потому
                           // что мы не освобождаем средство записи
    }
    listener.Stop();
}
```

Вот вывод:

Hello right back!

В данном примере мы применяем закольцовывание localhost, чтобы запустить клиент и сервер на одной машине. Мы произвольно выбрали порт из свободного диапазона (с номером больше 49152) и использовали классы `BinaryWriter` и `BinaryReader` для кодирования текстовых сообщений. Мы не закрывали и не освобождали средства чтения и записи, чтобы сохранить поток `NetworkStream` в открытом состоянии вплоть до завершения взаимодействия.

Классы `BinaryReader` и `BinaryWriter` могут показаться странным выбором для чтения и записи строк. Тем не менее, по сравнению с классами `StreamReader` и `StreamWriter` эти классы обладают важным преимуществом: они дополняют строки целочисленными префиксами, указывающими длину, так что объекту `BinaryReader` всегда известно, сколько байтов должно быть прочитано. Если вызвать метод `StreamReader.ReadToEnd`, то может возникнуть блокировка на неопределенное время, т.к. `NetworkStream` не имеет признака окончания. После того, как подключение открыто, сетевой поток никогда не может быть уверен в том, что клиент не собирается передавать дополнительную порцию данных.



В действительности класс `StreamReader` вообще запрещено применять вместе с `NetworkStream`, даже если вы планируете вызывать только метод `ReadLine`. Причина в том, что класс `StreamReader` имеет буфер опережающего чтения, который может привести к чтению большего числа байтов, чем доступно в текущий момент, и бесконечному блокированию (или вплоть до возникновения тайм-аута сокета). Другие потоки, такие как `FileStream`, не страдают подобной несовместимостью с классом `StreamReader`, потому что они поддерживают определенный признак окончания, при достижении которого метод `Read` немедленно завершается, возвращая значение 0.

Параллелизм и TCP

Классы `TcpClient` и `TcpListener` предлагают асинхронные методы на основе задач для реализации масштабируемого параллелизма. Их использование сводится просто к замене вызовов блокирующих методов версиями `*Async` этих методов и применению `await` к возвращаемым ими объектам задач.

В следующем примере мы создадим асинхронный сервер TCP, который принимает запросы длиной 5000 байтов, меняет порядок следования байтов на противоположный и затем отправляет их обратно клиенту:

```
async void RunServerAsync ()
{
    var listener = new TcpListener (IPAddress.Any, 51111);
    listener.Start ();
    try
    {
        while (true)
            Accept (await listener.AcceptTcpClientAsync ());
    }
    finally { listener.Stop(); }
```

```

async Task Accept (TcpClient client)
{
    await Task.Yield ();
    try
    {
        using (client)
        using (NetworkStream n = client.GetStream ())
        {
            byte[] data = new byte [5000];
            int bytesRead = 0; int chunkSize = 1;
            while (bytesRead < data.Length && chunkSize > 0)
                bytesRead += chunkSize =
                    await n.ReadAsync (data, bytesRead, data.Length - bytesRead);
            Array.Reverse (data); // Поменять порядок следования байтов
                                // на противоположный
            await n.WriteAsync (data, 0, data.Length);
        }
    }
    catch (Exception ex) { Console.WriteLine (ex.Message); }
}

```

Программа масштабируема в том отношении, что она не блокирует поток на время выполнения запроса. Таким образом, если 1000 клиентов одновременно подключаются к сети с использованием медленных каналов (так что от начала до конца каждого запроса проходит, скажем, несколько секунд), то программа не потребует на данный промежуток времени 1000 потоков (в отличие от синхронного решения). Взамен она арендует потоки только на краткие периоды времени, чтобы выполнить код до и после выражений `await`.

Получение почты POP3 с помощью TCP

Поддержка на прикладном уровне протокола POP3 в .NET отсутствует, поэтому для получения почты из сервера POP3 придется работать на уровне TCP. К счастью, данный протокол прост; взаимодействие в POP3 выглядит следующим образом:

Клиент	Почтовый сервер	Примечания
Клиент подключается...	+OK Hello there.	Приветственное сообщение
USER joe	+OK Password required.	
PASS password	+OK Logged in.	
LIST	+OK 1 1876 2 5412 3 845 •	Перечисляет идентификаторы и размеры файлов для каждого сообщения на сервере
RETR 1	+OK 1876 octets Содержимое сообщения #1...	Извлекает сообщение с указанным идентификатором
DELE 1	+OK Deleted.	Удаляет сообщение из сервера
QUIT	+OK Bye-bye.	

Каждая команда и ответ завершаются символом новой строки (<CR>+<LF>) за исключением многострочных команд LIST и RETR, которые завершаются одиночной точкой в отдельной строке. Поскольку мы не можем применять класс StreamReader вместе с NetworkStream, начнем с написания вспомогательного метода, предназначенного для чтения строки текста без буферизации:

```
string ReadLine (Stream s)
{
    List<byte> lineBuffer = new List<byte>();
    while (true)
    {
        int b = s.ReadByte();
        if (b == 10 || b < 0) break;
        if (b != 13) lineBuffer.Add ((byte)b);
    }
    return Encoding.UTF8.GetString (lineBuffer.ToArray());
}
```

Нам еще понадобится вспомогательный метод для отправки команды. Так как мы всегда ожидаем получения ответа, начинающегося с +OK, читать и проверять ответ можно в одно и то же время:

```
void SendCommand (Stream stream, string line)
{
    byte[] data = Encoding.UTF8.GetBytes (line + "\r\n");
    stream.Write (data, 0, data.Length);
    string response = ReadLine (stream);
    if (!response.StartsWith ("+OK"))
        throw new Exception ("POP Error: " + response); // Ошибка протокола POP
}
```

При наличии таких методов решить задачу извлечения почты довольно легко. Мы устанавливаем подключение TCP на порте 110 (стандартный порт POP3) и затем начинаем взаимодействие с сервером. В этом примере мы записываем каждое почтовое сообщение в произвольно именованный файл с расширением .eml и удаляем сообщение из сервера:

```
using (TcpClient client = new TcpClient ("mail.isp.com", 110))
using (NetworkStream n = client.GetStream())
{
    ReadLine (n); // Прочитать приветственное сообщение
    SendCommand (n, "USER имя-пользователя");
    SendCommand (n, "PASS пароль");
    SendCommand (n, "LIST"); // Извлечь идентификаторы сообщений
    List<int> messageIDs = new List<int>();
    while (true)
    {
        string line = ReadLine (n); // Например, "1 1876"
        if (line == ".") break;
        messageIDs.Add (int.Parse (line.Split (' ') [0])); // Идентификатор
        // сообщения
    }
}
```

```
foreach (int id in messageIDs)      // Извлечь каждое сообщение
{
    SendCommand (n, "RETR " + id);
    string randomFile = Guid.NewGuid().ToString() + ".eml";
    using (StreamWriter writer = File.CreateText (randomFile))
        while (true)
    {
        string line = ReadLine (n); // Прочитать следующую строку сообщения
        if (line == ".") break;    // Одиночная точка - конец сообщения
        if (line == "..") line = ".";
        writer.WriteLine (line);   // Записать в выходной файл
    }
    SendCommand (n, "DELE " + id); // Удалить сообщение из сервера
}
SendCommand (n, "QUIT");
}
```



В галерее NuGet вы можете обнаружить библиотеки с открытым кодом для работы с протоколом POP3, которые предлагают поддержку таких аспектов протокола, как аутентификация подключений TLS/SSL, разбор MIME и т.д.



Сборки

Сборка — это базовая единица развертывания в .NET, а также контейнер для всех типов. Сборка содержит скомпилированные типы с их кодом на промежуточном языке (IL), ресурсы времени выполнения и информацию, которая соответствует управлению версиями и ссылками на другие сборки. Сборка также определяет границы для распознавания типов. В .NET сборка представляет собой одиночный файл с расширением .dll.



При компиляции исполняемого приложения в .NET вы получаете два файла: файл сборки (.dll) и файл исполняемого модуля запуска (.exe), которые соответствуют платформе, выбранной в качестве цели. Ситуация отличается от того, что происходило в инфраструктуре .NET Framework, которая генерировала *переносимую исполняемую* (portable executable — PE) сборку. Файл PE имел расширение .exe и действовал как сборка и как исполняемый модуль запуска. Файл PE может быть одновременно нацелен на 32- и 64-разрядные версии Windows.

Большинство типов в главе находятся в следующих пространствах имен:

System.Reflection
System.Resources
System.Globalization

Содержимое сборки

Сборка содержит четыре вида информации.

- **Манифест сборки.** Предоставляет сведения для среды CLR, такие как имя сборки, версия и ссылки на другие сборки.
- **Манифест приложения.** Предоставляет сведения для операционной системы (ОС), такие как способ развертывания сборки и необходимость в подъеме полномочий до административных.
- **Скомпилированные типы.** Скомпилированный код IL и метаданные типов, определенных внутри сборки.
- **Ресурсы.** Другие встроенные в сборку данные, такие как изображения и локализуемый текст.

Из всего перечисленного обязательным является только *манифест сборки*, хотя сборка почти всегда содержит скомпилированные типы (если только это не ресурсная сборка; см. раздел “Ресурсы и подчиненные сборки” далее в главе).

Манифест сборки

Манифест сборки служит двум целям:

- описывает сборку для управляемой среды размещения;
- действует в качестве каталога для модулей, типов и ресурсов в сборке.

Таким образом, сборки являются *самоописательными*. Потребитель может обнаруживать данные, типы и функции всех сборок без необходимости в наличии дополнительных файлов.



Манифест сборки — это не сущность, добавляемая к сборке явно; он встраивается в сборку автоматически во время компиляции.

Ниже представлены краткие сведения по функционально значащим данным, хранящимся в манифесте:

- простое имя сборки;
- номер версии (`AssemblyVersion`);
- открытый ключ и подписанный хеш сборки, если она имеет строгое имя;
- список ссылаемых сборок, включающий их версии и открытые ключи;
- список типов, определенных в сборке;
- целевая культура в случае подчиненной сборки (`AssemblyCulture`).

Манифест также может хранить следующие информационные данные:

- полный заголовок и описание (`AssemblyTitle` и `AssemblyDescription`);
- информация о компании и авторском праве (`AssemblyCompany` и `AssemblyCopyright`);
- отображаемая версия (`AssemblyInformationalVersion`);
- дополнительные атрибуты для специальных данных.

Некоторые перечисленные данные выводятся из аргументов, переданных компилятору, например, список ссылаемых сборок или открытый ключ для подписания сборки. Остальные данные поступают из атрибутов сборки, указанных в круглых скобках.



Просмотреть содержимое манифеста сборки можно с помощью инструмента .NET под названием `ildasm.exe`. В главе 18 будет показано, как делать то же самое программно с применением рефлексии.

Указание атрибутов сборки

Часто используемые атрибуты можно указывать в окне свойств проекта среды Visual Studio. Такие настройки добавляются в файл проекта (.csproj).

Атрибуты сборки, которые не поддерживаются в окне свойств или не работают с файлом .csproj, можно указывать в исходном коде (что нередко делается в файле по имени AssemblyInfo.cs).

Выделенный файл атрибутов содержит только операторы using и объявления атрибутов сборки. Например, чтобы типы с внутренней областью видимости стали доступными проекту модульного тестирования, потребуется поступить следующим образом:

```
using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("MyUnitTestProject")]
```

Манифест приложения (Windows)

Манифест приложения — это XML-файл, который сообщает ОС информацию о сборке. Во время процесса построения манифест приложения встраивается в исполняемый модуль запуска как ресурс Win32. Если манифест приложения присутствует, тогда он читается и обрабатывается до загрузки сборки средой CLR и может повлиять на то, как Windows запускает процесс приложения.

Манифест приложения .NET имеет корневой элемент под названием assembly в пространстве имен XML вида urn:schemas-microsoft-com:asm.v1:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
    <!-- содержимое манифеста -->
</assembly>
```

Следующий манифест инструктирует ОС о запрашивании поднятия полномочий до административных:

```
<?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>
```

(Приложения UWP имеют гораздо более сложный манифест, описанный в файле Package.appxmanifest. Он включает объявление функциональных возможностей программы, которые определяют разрешения, выдаваемые ОС. Простейший способ редактирования данного файла предусматривает использование среды Visual Studio, которая отображает диалоговое окно в результате двойного щелчка на файле манифеста.)

Развертывание манифеста приложения

Чтобы добавить манифест приложения к проекту .NET в Visual Studio, необходимо щелкнуть правой кнопкой мыши на элементе проекта в проводнике решения, выбрать в контекстном меню пункт Add (Добавить), затем New Item (Новый элемент) и, наконец, Application Manifest File (Файл манифеста приложения). После построения манифест будет встроен в выходную сборку.



Инструмент .NET под названием `ildasm.exe` не замечает присутствия встроенного манифеста приложения. Тем не менее, среда Visual Studio указывает на наличие встроенного манифеста приложения, если дважды щелкнуть на сборке в проводнике решения.

Модули

Содержимое сборки в действительности упаковано внутри промежуточного контейнера, который называется *модулем*. Модуль соответствует файлу, вмещающему содержимое сборки. Причина наличия такого дополнительного контейнерного уровня — позволить сборке охватывать множество файлов; эта возможность имеется в .NET Framework, но отсутствует в .NET 5+ и .NET Core. Взаимосвязь между сборкой и модулем проиллюстрирована на рис. 17.1.

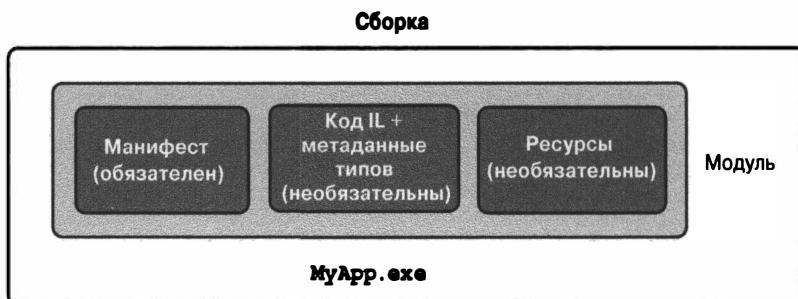


Рис. 17.1. Однофайловая сборка

Хотя .NET не поддерживает многофайловые сборки, временами нужно принимать во внимание дополнительный контейнерный уровень, предлагаемый модулями. Основной сценарий связан с рефлексией (как показано в разделах “Рефлексия сборок” и “Выпуск сборок и типов” главы 18).

Класс `Assembly`

Класс `Assembly` из пространства имен `System.Reflection` представляет собой шлюз для доступа к метаданным сборки во время выполнения. Существует несколько способов получения объекта сборки, простейший из которых предусматривает использование свойства `Assembly` класса `Type`:

```
Assembly a = typeof (Program).Assembly;
```

Получить объект Assembly можно также с помощью вызова одного из перечисленных ниже статических методов класса Assembly.

- `GetExecutingAssembly`. Возвращает сборку типа, в котором определена текущая выполняемая функция.
- `GetCallingAssembly`. Делает то же, что и метод `GetExecutingAssembly`, но для функции, которая вызвала текущую выполняемую функцию.
- `GetEntryAssembly`. Возвращает сборку, определяющую первоначальный метод точки входа в приложение.

После получения объекта Assembly можно применять его свойства и методы для запрашивания метаданных сборки и рефлексии ее типов. В табл. 17.1 представлена сводка по членам класса Assembly.

Таблица 17.1. Члены класса Assembly

Члены	Назначение	Разделы, в которых рассматриваются
<code>FullName, GetName</code>	Возвращает полностью заданное имя или объект <code>AssemblyName</code>	“Имена сборок” далее в главе
<code>CodeBase, Location</code>	Местоположение файла сборки	“Загрузка, распознавание и изолирование сборок” далее в главе
<code>Load, LoadFrom, LoadFile</code>	Вручную загружает сборку в текущий домен приложения	“Загрузка, распознавание и изолирование сборок” далее в главе
<code>GetSatelliteAssembly</code>	Находит подчиненную сборку с заданной культурой	“Ресурсы и подчиненные сборки” далее в главе
<code>GetType, GetTypes</code>	Возвращает тип или все типы, определенные в сборке	“Рефлексия и активизация типов” в главе 18
<code>EntryPoint</code>	Возвращает метод точки входа в приложение как объект <code>MethodInfo</code>	“Рефлексия и вызов членов” в главе 18
<code>GetModule, GetModules, ManifestModule</code>	Возвращает модуль, все модули или главный модуль сборки	“Рефлексия сборок” в главе 18
<code>GetCustomAttribute, GetCustomAttributes</code>	Возвращает атрибут или атрибуты сборки	“Работа с атрибутами” в главе 18

Строгие имена и подписание сборок



Назначение строгого имени сборке было важно в .NET Framework по двум причинам:

- оно позволяло загружать сборку в так называемый “глобальный кеш сборок”;
- оно позволяло ссылаться на сборку из других строго именованных сборок.

Назначение строгих имен сборкам в .NET 5+ и .NET Core гораздо менее важно, поскольку их исполняющие среды не имеют глобального кеша сборок и не накладывают ограничение, изложенное во второй причине.

Строго именованная сборка имеет уникальное удостоверение, подделать которое невозможно. Оно получается за счет добавления к манифесту следующих метаданных:

- *的独特 номера*, который принадлежит авторам сборки;
- *подписанного хеша* сборки, подтверждающего тот факт, что сборка создана владельцем уникального номера.

Такой подход требует пары открытого и секретного ключей. *Открытый ключ* предоставляет уникальный идентифицирующий номер, а *секретный ключ* облегчает подписание.



Подписание с помощью *строгого имени* — не то же самое, что подписание *Authenticode*. Система Authenticode рассматривается далее в главе.

Открытый ключ ценен для гарантирования уникальности ссылок на сборку: строго именованная сборка содержит в своем удостоверении открытый ключ.

В .NET Framework секретный ключ защищает сборку от подделки, т.к. без вашего секретного ключа никто не сможет выпустить модифицированную версию сборки, не нарушив подпись. На практике это полезно при загрузке сборки в глобальный кеш сборок .NET Framework. В .NET 5+ и .NET Core от подписи мало пользы, потому что она никогда не проверяется.

Добавление строгого имени к сборке, ранее обладающей “слабым” именем, изменяет ее удостоверение. По указанной причине сборке имеет смысл назначать строгое имя с самого начала, если вы думаете, что в будущем она потребует строгого имени.

Назначение сборке строгого имени

Чтобы назначить сборке строгое имя, сначала понадобится с помощью утилиты sn.exe сгенерировать пару открытого и секретного ключей:

```
sn.exe -k MyKeyValuePair.snk
```



Среда Visual Studio устанавливает ярлык под названием **Developer Command Prompt for VS** (Командная подсказка разработчика для Visual Studio), щелчок на котором приводит к открытию окна командной строки с переменной PATH, содержащей путь к папке с инструментами разработчика, такими как sn.exe.

Утилита sn.exe создает новую пару ключей и сохраняет ее в файле MyKeyPair.snk. Если вы впоследствии потеряете этот файл, то навсегда утратите возможность перекомпиляции своей сборки с тем же самым удостоверением.

Подписать сборку с помощью файла MyKeyPair.snk можно за счет обновления файла проекта. Откройте в Visual Studio окно свойств проекта и перейдите в нем на вкладку **Signing** (Подписание); отметьте флагок **Sign the assembly** (Подписать сборку) и выберите свой файл .snk.

Одной и той же парой ключей можно подписывать множество сборок — если их простые имена отличаются, то они получат разные удостоверения.

Имена сборок

“Удостоверение” сборки содержит в себе четыре фрагмента метаданных из манифеста сборки:

- простое имя;
- версия (0.0.0.0, если не указана);
- культура (neutral (нейтральная), если сборка не является подчиненной);
- маркер открытого ключа (null (пустой), если строгое имя не задано).

Простое имя поступает не из какого-то атрибута, а представляет собой имя файла, в который сборка была первоначально скомпилирована (не включая расширение). Таким образом, простое имя сборки System.Xml.dll выглядит как System.Xml. Переименование файла не изменяет простое имя сборки.

Номер версии берется из атрибута AssemblyVersion. Это строка, разделенная на четыре части:

старший_номер.младший_номер.компоновка.редакция

Указать номер версии можно следующим образом:

```
[assembly: AssemblyVersion ("2.5.6.7")]
```

Культура поступает из атрибута AssemblyCulture и применяется к подчиненным сборкам, которые описаны в разделе “Ресурсы и подчиненные сборки” далее в главе.

Маркер открытого ключа извлекается из строгого имени, предоставляемого на этапе компиляции, как обсуждалось в предыдущем разделе.

Полностью заданные имена

Полностью заданное имя сборки — это строка, включающая все четыре идентифицирующих компонента в таком формате:

простое_имя, Version=версия, Culture=культура, PublicKeyToken=открытый_ключ

Например, полностью заданное имя сборки System.Private.CoreLib.dll выглядит как System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e.

Если сборка не имеет атрибута AssemblyVersion, то ее версией будет 0.0.0.0. Для неподписанной сборки маркер открытого ключа отображается как null.

Свойство FullName объекта Assembly возвращает полностью заданное имя сборки. При занесении ссылок на сборки в манифест компилятор всегда использует полностью заданные имена.



Полностью заданное имя не включает путь к каталогу, чтобы тем самым содействовать в нахождении сборки на диске. Поиск сборки, расположенной в другом каталоге, является совершенно другой темой, которой посвящен раздел “Загрузка, распознавание и изолирование сборок” далее в главе.

Класс AssemblyName

AssemblyName — класс с типизированными свойствами для каждого из четырех компонентов полностью заданного имени сборки. Класс AssemblyName служит двум целям:

- он разбирает или строит полностью заданное имя сборки;
- он хранит некоторые дополнительные данные, помогающие распознавать (находить) сборку.

Получить объект AssemblyName можно любым из следующих способов:

- создать объект AssemblyName, предоставив полностью заданное имя;
- вызвать метод GetName на существующем объекте Assembly;
- вызвать метод AssemblyName.GetAssemblyName, указав путь к файлу сборки на диске.

Объект AssemblyName можно также создать безо всяких аргументов и затем установить все его свойства, построив в итоге полностью заданное имя. Когда объект AssemblyName сконструирован в подобной манере, он является изменяемым. Ниже перечислены важные свойства и методы AssemblyName:

```
string      FullName     { get; }           // Полностью заданное имя
string      Name         { get; set; }       // Простое имя
Version     Version      { get; set; }       // Версия сборки
CultureInfo CultureInfo { get; set; }       // Для подчиненных сборок
string      CodeBase     { get; set; }       // Местоположение
byte[]      GetPublicKey();                 // 160 байтов
void        SetPublicKey (byte[] key);
byte[]      GetPublicKeyToken();            // 8-байтовая версия
void        SetPublicKeyToken (byte[] publicToken);
```

Само свойство Version — это строго типизированное представление со свойствами Major, Minor, Build и Revision. Метод GetPublicKey возвра-

щает криптографически стойкий открытый ключ; метод GetPublicKeyToken возвращает последние восемь байтов, применяемых в устанавливаемом удостоверении.

Вот как использовать AssemblyName для получения простого имени сборки:

```
Console.WriteLine (typeof (string).Assembly.GetName ().Name);  
// System.Private.CoreLib
```

А так извлекается версия сборки:

```
string v = myAssembly.GetName ().Version.ToString();
```

Информационная и файловая версии сборки

Для представления информации, связанной с версиями, доступны еще два атрибута сборки. В отличие от AssemblyVersion описанные далее два атрибута не воздействуют на удостоверение сборки и потому не влияют на то, что происходит на этапе компиляции или во время выполнения:

- AssemblyInformationalVersion. Версия, отображаемая конечному пользователю. Она видна в поле Product Version (Версия продукта) диалогового окна свойств файла. Здесь можно указывать любую строку, например, "5.1 Beta 2". Обычно всем сборкам в приложении будет назначаться один и тот же номер информационной версии.
- AssemblyFileVersion. Позволяет ссылаться на номер текущего варианта данной сборки. Такой номер отображается в поле File Version (Файловая версия) диалогового окна свойств файла. Как и AssemblyVersion, атрибут должен содержать строку, которая состоит максимум из четырех чисел, разделенных точками.

Подпись Authenticode

Authenticode — это система подписания кода, назначение которой заключается в подтверждении удостоверения издателя. Система Authenticode и подписание с помощью строгого имени не зависят друг от друга: подписать сборку можно посредством либо какой-то одной, либо обеих систем.

В то время как подписание с помощью строгого имени подтверждает, что сборки A, B и C поступают от одного и того же издателя (предполагая, что не произошла утечка секретного ключа), они не позволяют выяснить, кто конкретно этот издатель. Чтобы узнать, кто является издателем — Джо Албахари или компания Microsoft Corporation — нужна система Authenticode.

Система Authenticode полезна при загрузке программ из Интернета, т.к. она дает гарантию того, что программа поступает от издателя, зарегистрированного в центре сертификации (Certificate Authority), и не была по пути модифицирована. Данная система также обеспечивает выдачу предупреждения о неизвестном издателе ("Unknown Publisher"), когда загруженное приложение запускается впервые. Кроме того, подписание Authenticode обязательно при отправке приложений в Магазин Microsoft.

Система Authenticode работает не только со сборками .NET, но также с неуправляемыми исполняемыми и двоичными модулями, такими как файлы развертывания .msi. Разумеется, система Authenticode не гарантирует, что программа свободна от вредоносного кода — хотя делает это менее вероятным. Дело в том, что физическое или юридическое лицо добровольно указало под исполняемым модулем или библиотекой свое реальное имя (подкрепленное паспортом или документом компании).



Среда CLR не трактует подпись Authenticode как часть удостоверения сборки. Тем не менее, как вскоре будет показано, она может читать и проверять достоверность подписей Authenticode по требованию.

Подписание с помощью Authenticode требует обращения в *центр сертификации* (Certificate Authority — CA) с доказательством вашей персональной личности или удостоверения компании (учредительный договор и т.п.). После того как в CA проверят предъявленные документы, они выдадут сертификат подписания кода X.509, который обычно действителен от одного до пяти лет. Сертификат позволяет подписывать сборки с применением утилиты signtool. Можно также создать сертификат самостоятельно с помощью утилиты makecert, однако он будет распознаваться только на компьютерах, на которых был явно установлен.

Тот факт, что сертификаты (не подписанные самостоятельно) могут работать на любом компьютере, опирается на инфраструктуру открытых ключей. По существу ваш сертификат подписывается с помощью другого сертификата, принадлежащего CA. Центр сертификации является доверенным, потому что все CA загружаются в операционную систему. (Чтобы увидеть их, откройте окно панели управления Windows и введите в поле поиска слово certificate. В разделе Администрирование щелкните на ссылке Управление сертификатами компьютеров. В результате запустится диспетчер сертификатов. Раскройте узел Доверенные корневые центры сертификации и щелкните на папке Сертификаты.) Центр сертификации может отзывать сертификат издателя, если произошла его утечка, поэтому проверка подписи Authenticode требует периодического запрашивания у CA актуальных списков отзываемых сертификатов.

Из-за того, что система Authenticode использует криптографические подписи, подпись Authenticode становится недействительной, если кто-то впоследствии подделает файл. Вопросы, связанные с криптографией, хешированием и подписанием, рассматриваются в главе 20.

Подписание с помощью системы Authenticode

Получение и установка сертификата

Первый шаг связан с получением сертификата для подписания кода в CA (как объясняется ниже во врезке “Где получать сертификат для подписания кода?”). Затем с сертификатом можно либо работать как с файлом, защищенным паролем, либо загрузить его в хранилище сертификатов на компьютере. Преимущество второго варианта в том, что при подписании не придется указывать пароль. Это позволяет избавиться от явного помещения пароля в сценарии построения сборок или пакетные файлы.

Где получать сертификат для подписания кода?

В Windows предварительно загружается несколько корневых центров сертификации, в том числе Comodo, GoDaddy, GlobalSign, DigiCert, Thawte и Symantec.

Существует также торговый посредник K Software, который предлагает сертификаты от вышеупомянутых центров сертификации со скидкой.

Сертификаты Authenticode, выдаваемые K Software, Comodo, GoDaddy и GlobalSign, рекламируются как менее ограничивающие в том, что с их помощью можно также подписывать программы для платформ, отличающихся от Microsoft. В остальном продукты всех поставщиков функционально эквивалентны.

Обратите внимание, что сертификат для SSL в общем случае не может применяться для подписания с помощью Authenticode (несмотря на использование той же самой инфраструктуры X.509). Частично это обусловлено тем, что сертификат для SSL касается доказательства прав собственности на домен, а сертификат Authenticode связан с подтверждением личности.

Чтобы загрузить сертификат в хранилище сертификатов на компьютере, запустите диспетчер сертификатов, как было описано ранее. Откройте папку Личное, щелкните на папке Сертификаты и выберите в контекстном меню пункт Все задачи\Импорт. Мастер импорта проведет вас через все необходимые шаги. После того как мастер импорта сертификатов завершит работу, при выбранном сертификате щелкните на кнопке Просмотр, в открывшемся диалоговом окне Сертификат перейдите на вкладку Состав и скопируйте *отпечаток* сертификата. Это хеш SHA-256, который впоследствии понадобится для удостоверения сертификата во время подписания.



Если вы также хотите подписать свою сборку с помощью строгого имени (что настоятельно рекомендуется), то должны делать это *до* подписания посредством Authenticode. Причина в том, что среди CLR известно о подписях Authenticode, но не наоборот. Таким образом, если вы подпишете свою сборку с помощью строгого имени *после* ее подписания с применением Authenticode, то система Authenticode будет рассматривать добавление средой CLR строгого имени как неавторизованную модификацию и считать, что сборка подделана.

Подписание с помощью `signtool.exe`

Подписывать свои программы с использованием системы Authenticode можно посредством утилиты signtool, входящей в состав Visual Studio (загляните в папку Microsoft SDKs\ClickOnce\SignTool внутри папки Program Files). Следующая команда подписывает файл LINQPad.exe с помощью сертификата, хранящегося в папке My Store компьютера по имени "Joseph Albahari", используя алгоритм хеширования SHA256:

```
signtool sign /n "Joseph Albahari" /fd sha256 LINQPad.exe
```

Можно также задать описание и URL продукта в переключателях /d и /du:

```
... /d LINQPad /du http://www.linqpad.net
```

В большинстве случаев будет также указываться *сервер отметок времени*.

Отметки времени

После истечения срока действия сертификата вы больше не сможете подписывать программы. Тем не менее, программы, которые были подписаны *до* истечения срока действия, по-прежнему будут действительными, если при подписании указывался *сервер отметок времени* с помощью переключателя /tr. Для такой цели центр сертификации предоставляет URI: ниже показан URI для Comodo (или K Software):

```
... /tr http://timestamp.comodoca.com/authenticode /td SHA256
```

Проверка, подписана ли программа

Простейший способ увидеть подпись Authenticode для файла — просмотреть свойства файла в проводнике Windows (на вкладке цифровых сертификатов). Утилита signtool также предоставляет такую возможность.

Ресурсы и подчиненные сборки

Приложение обычно содержит не только исполняемый код, но и такие элементы, как текст, изображения или XML-файлы. Содержимое подобного рода может быть представлено в сборке посредством *ресурсов*. Для ресурсов предусмотрены два перекрывающихся сценария использования:

- встраивание данных, которые не могут располагаться в исходном коде, вроде изображений;
- хранение данных, которым может потребоваться перевод в многоязычном приложении.

Ресурс сборки в конечном итоге представляет собой байтовый поток с именем. О сборке можно говорить, что она содержит словарь байтовых массивов со строковыми ключами. Вот что можно увидеть с помощью утилиты ildasm, если дизассемблировать сборку, которая содержит ресурсы banner.jpg и data.xml:

```
.mresource public banner.jpg
{
    // Offset: 0x00000F58 Length: 0x000004F6
    // Смещение: 0x00000F58 Длина: 0x000004F6
}

.mresource public data.xml
{
    // Offset: 0x00001458 Length: 0x0000027E
    // Смещение: 0x00001458 Длина: 0x0000027E
}
```

В данном случае элементы banner.jpg и data.xml были включены прямо в сборку — каждый в виде собственного встроенного ресурса. Это простейший способ работы.

.NET также позволяет добавлять содержимое через промежуточные контейнеры .resources. Они предназначены для хранения содержимого, которое может требовать перевода на разные языки. Локализованные контейнеры .resources могут быть упакованы как отдельные подчиненные сборки, которые автоматически выбираются во время выполнения на основе языка ОС пользователя.

На рис. 17.2 показана сборка, содержащая два напрямую встроенных ресурса, а также контейнер .resources по имени welcome.resources, для которого созданы две локализованные подчиненные сборки.

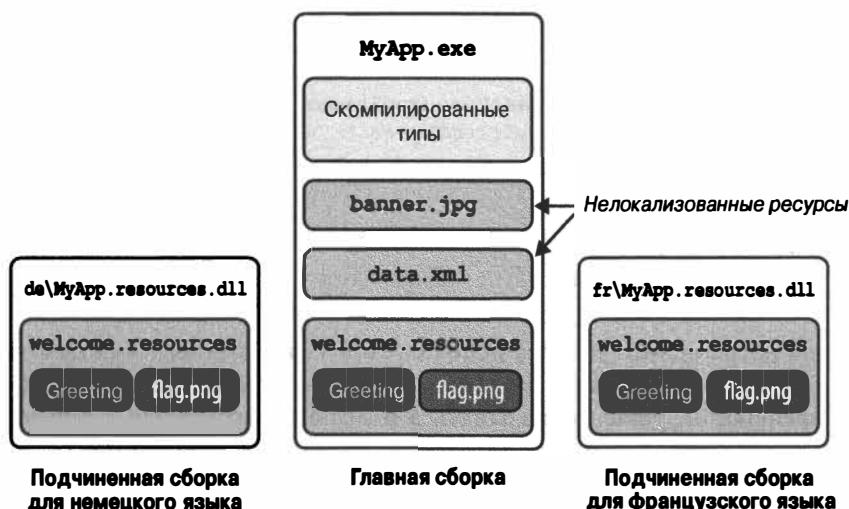


Рис. 17.2. Ресурсы

Встраивание ресурсов напрямую



Встраивание ресурсов внутрь сборок в приложениях для Магазина Microsoft не поддерживается. Взамен любые дополнительные файлы необходимо добавлять в пакет развертывания и обращаться к ним, читая в приложении объект StorageFolder (свойство Package.Current.InstalledLocation).

Для встраивания ресурса внутрь сборки напрямую с использованием Visual Studio понадобится выполнить следующие действия:

- добавить файл к проекту;
- установить действие построения проекта в Embedded Resource (Встроенный ресурс).

Среда Visual Studio всегда предваряет имена ресурсов стандартным пространством имен проекта, а также именами всех подпапок, ведущих к файлу. Таким образом, если стандартным пространством имен проекта было Westwind.Reports, а файл назывался banner.jpg и находился в папке pictures, то имя ресурса выглядело бы как Westwind.Reports.pictures.banner.jpg.



Имена ресурсов чувствительны к регистру символов, что делает имена подпапок с ресурсами в Visual Studio фактически чувствительными к регистру.

Чтобы извлечь ресурс, необходимо вызвать метод `GetManifestResourceStream` на объекте сборки, содержащей ресурс. Упомянутый метод возвращает поток, который затем допускается читать подобно любому другому потоку:

```
Assembly a = Assembly.GetEntryAssembly();
using (Stream s = a.GetManifestResourceStream ("TestProject.data.xml"))
using (XmlReader r = XmlReader.Create (s))
    ...
System.Drawing.Image image;
using (Stream s = a.GetManifestResourceStream ("TestProject.banner.jpg"))
    image = System.Drawing.Image.FromStream (s);
```

Возвращенный поток поддерживает позиционирование, поэтому можно поступить и так:

```
byte[] data;
using (Stream s = a.GetManifestResourceStream ("TestProject.banner.jpg"))
    data = new BinaryReader (s).ReadBytes ((int) s.Length);
```

Если для встраивания ресурса используется Visual Studio, тогда нужно не забыть о включении префикса, основанного на пространстве имен. Во избежание ошибки префикс разрешено указывать в отдельном аргументе с применением *типа*. В качестве префикса используется пространство имен данного типа:

```
using (Stream s = a.GetManifestResourceStream (typeof (X), "data.xml"))
```

Здесь X может быть любым типом с желаемым пространством имен ресурса (обычно это тип в той же папке проекта).



Установка действия построения в Visual Studio для элемента проекта в **Resource** (Ресурс) внутри WPF-приложения — *не то же самое*, что установка его действия построения в **Embedded Resource**. В первом случае фактически добавляется элемент в файл .resources по имени `<ИмяСборки>.g.resources`, к содержимому которого можно получать доступ через класс `Application` инфраструктуры WPF, применяя URI в качестве ключа.

Добавок в инфраструктуре WPF переопределен термин “ресурс”, что еще больше увеличивает путаницу. *Статические ресурсы* и *динамические ресурсы* не имеют никакого отношения к ресурсам сборки!

Метод `GetManifestResourceNames` возвращает имена всех ресурсов в сборке.

Файлы .resources

Файлы .resources представляют собой контейнеры для потенциально локализуемого содержимого. Файл .resources в итоге становится встроенным ресурсом внутри сборки — точно так же, как файл другого вида.

Разница заключается в том, что вы можете выполнять следующие действия:

- для начала упаковывать свое содержимое в файл .resources;
- получать доступ к содержимому через объект ResourceManager или URI типа "pack", а не посредством метода GetManifestResourceStream.

Файлы .resources являются двоичными и не предназначены для редактирования человеком; таким образом, при работе с ними придется полагаться на инструменты, предоставляемые .NET и Visual Studio. Стандартный подход со строками или простыми типами данных предусматривает применение файла в формате .resx, который может быть преобразован в файл .resources с помощью Visual Studio либо инструмента resgen. Формат .resx также подходит для изображений, предназначенных для приложения Windows Forms или ASP.NET.

В приложении WPF должно использоваться действие построения Resource (Ресурс) среды Visual Studio для изображений или похожего содержимого, нуждающегося в ссылке через URI. Это применимо вне зависимости от того, необходима локализация или нет.

Мы опишем упомянутые действия в последующих разделах.

Файлы .resx

Файл .resx имеет формат этапа проектирования, предназначенный для получения файлов .resources. Файл .resx использует XML и структурирован в виде пар "имя/значение", как показано ниже:

```
<root>
  <data name="Greeting">
    <value>hello</value>
  </data>
  <data name="DefaultFontSize" type="System.Int32, mscorelib">
    <value>10</value>
  </data>
</root>
```

Чтобы создать файл .resx в Visual Studio, понадобится добавить элемент проекта типа Resources File (Файл ресурсов). Оставшаяся часть работы будет произведена автоматически:

- создается корректный заголовок;
- предоставляется визуальный конструктор для добавления строк, изображений, файлов и других видов данных;
- файл .resx автоматически преобразуется в формат .resources и встраивается в сборку при компиляции;
- генерируется класс, предназначенный для облегчения доступа к данным в более позднее время.



Визуальный конструктор ресурсов добавляет изображения как объекты типа Image (сборка System.Drawing.dll), а не как байтовые массивы, делая изображения неподходящими для приложений WPF.

Чтение файлов .resources



В случае создания файла .resx в Visual Studio автоматически генерируется класс с таким же именем, который имеет свойства для извлечения каждого элемента.

Класс ResourceManager читает файлы .resources, встроенные внутрь сборки:

```
ResourceManager r = new ResourceManager ("welcome",
                                         Assembly.GetExecutingAssembly());
```

(Если ресурс был скомпилирован в Visual Studio, тогда первый аргумент должен быть предварен названием пространства имен.)

Далее можно получить доступ к содержимому, вызывая метод GetString или GetObject и выполняя приведение:

```
string greeting = r.GetString ("Greeting");
int fontSize    = (int) r.GetObject ("DefaultFontSize");
Image image     = (Image) r.GetObject ("flag.png");
```

Перечисление содержимого файла .resources производится следующим образом:

```
ResourceManager r = new ResourceManager (...);
ResourceSet set = r.GetResourceSet (CultureInfo.CurrentCulture,
                                    true, true);
foreach (System.Collections.DictionaryEntry entry in set)
    Console.WriteLine (entry.Key);
```

Создание ресурса с доступом через URI типа "pack" в Visual Studio

В приложении WPF файлы XAML должны иметь возможность доступа к ресурсам по URI. Например:

```
<Button>
    <Image Height="50" Source="flag.png"/>
</Button>
```

Или, если ресурс находится в другой сборке:

```
<Button>
    <Image Height="50" Source="UtilsAssembly;Component/flag.png"/>
</Button>
```

(Component — это литеральное ключевое слово.)

Для создания ресурсов, которые могут загружаться в подобной манере, применять файлы .resx не получится. Взамен придется добавлять файлы в проект и устанавливать для них действие построения в Resource (не Embedded Resource). Среда Visual Studio скомпилирует их в файл .resources по имени <ИмяСборки>.g.resources, в котором также содержатся скомпилированные файлы XAML (.baml).

Чтобы программно загрузить ресурс с ключом URI, необходимо вызвать метод Application.GetResourceStream:

```
Uri u = new Uri ("flag.png", UriKind.Relative);
using (Stream s = Application.GetResourceStream (u).Stream)
```

Обратите внимание на использование относительного URI. Можно также применять абсолютный URI в показанном ниже формате (три запятых подряд — это не опечатка):

```
Uri u = new Uri ("pack://application:,,,/flag.png");
```

Если вместо Application указать объект Assembly, то содержимое можно извлечь с помощью объекта ResourceManager:

```
Assembly a = Assembly.GetExecutingAssembly();  
ResourceManager r = new ResourceManager (a.GetName().Name + ".g", a);  
using (Stream s = r.GetStream ("flag.png"))  
    ...
```

Объект ResourceManager также позволяет выполнять перечисление содержимого контейнера .g.resources внутри заданной сборки.

Подчиненные сборки

Данные, встроенные в файл .resources, являются локализуемыми.

Проблема локализации ресурсов возникает, когда приложение запускается под управлением версии Windows, ориентированной на отображение всех элементов на другом языке. В целях согласованности приложение должно использовать тот же самый язык.

Типичная настройка предполагает наличие следующих компонентов:

- главная сборка, содержащая файлы .resources для стандартного, или запасного, языка;
- отдельные *подчиненные сборки*, которые содержат локализованные файлы .resources, переведенные на различные языки.

Когда приложение запускается, исполняющая среда .NET выясняет язык текущей ОС (через свойство CultureInfo.CurrentCulture). Всякий раз, когда запрашивается ресурс с применением объекта ResourceManager, исполняющая среда ищет локализованную подчиненную сборку. Если такая сборка доступна и содержит запрошенный ключ ресурса, тогда этот ресурс используется вместо его версии из главной сборки.

Другими словами, расширять языковую поддержку можно просто добавлением новых подчиненных сборок, не изменяя главную сборку.



Подчиненная сборка не может содержать исполняемый код — допускается наличие только ресурсов.

Подчиненные сборки развертываются в подкаталогах папки сборки, как показано ниже:

```
programBaseFolder\MyProgram.exe  
    \MyLibrary.exe  
    \XX\MyProgram.resources.dll  
    \XX\MyLibrary.resources.dll
```

Здесь XX — двухбуквенный код языка (скажем, de для немецкого) либо код языка и региона (например, en-GB для английского в Великобритании). Такая система именования позволяет среди CLR находить и автоматически загружать корректную подчиненную сборку.

Построение подчиненных сборок

Вспомните предшествующий пример файла .resx, который имел следующее содержимое:

```
<root>
  ...
<data name="Greeting">
  <value>hello</value>
</data>
</root>
```

Затем во время выполнения мы извлекали приветственное сообщение (Greeting):

```
ResourceManager r = new ResourceManager ("welcome",
                                         Assembly.GetExecutingAssembly ());
Console.WriteLine (r.GetString ("Greeting"));
```

Предположим, что при запуске в среде немецкоязычной ОС Windows вместо "hello" требуется вывести "hallo". Первый шаг заключается в добавлении еще одного файла .resx по имени welcome.de.resx, в котором строка hello заменяется строкой hallo:

```
<root>
  <data name="Greeting">
    <value>hallo</value>
  </data>
</root>
```

В Visual Studio это все, что необходимо сделать — при компиляции в подкаталоге de будет автоматически создана подчиненная сборка по имени MyApp.resources.dll.

Тестирование подчиненных сборок

Для эмуляции выполнения в среде ОС с другим языком потребуется изменить свойство CurrentUICulture с применением класса Thread:

```
System.Threading.Thread.CurrentThread.CurrentCulture
  = new System.Globalization.CultureInfo ("de");
```

CultureInfo.CurrentCulture — версия того же самого свойства, допускающая только чтение.



Удобная стратегия тестирования предусматривает локализацию слов, которые по-прежнему должны читаться как английские, но не использовать стандартные латинские символы Unicode (например, Łośałizę).

Поддержка со стороны визуальных конструкторов Visual Studio

Визуальные конструкторы в Visual Studio предлагают расширенную поддержку для локализуемых компонентов и визуальных элементов. Визуальный конструктор WPF имеет собственный рабочий поток для локализации; другие визуальные конструкторы, основанные на Component, применяют свойство, предназначеннное только для этапа проектирования, которое показывает, что компонент или элемент управления Windows Forms имеет свойство Language. Для настройки на другой язык нужно просто изменить значение свойства Language и затем приступить к модификации компонента. Значения всех свойств элементов управления с атрибутом Localizable будут сохраняться в файле .resx для конкретного языка. Переключаться между языками можно в любой момент, просто изменения свойство Language.

Культуры и подкультуры

Культуры разделяются на собственно культуры и подкультуры. Культура представляет конкретный язык, а подкультура — региональный вариант этого языка. Исполняющая среда .NET следует стандарту RFC-1766, который представляет культуры и подкультуры с помощью двухбуквенных кодов. Ниже показаны коды для английской и немецкой культур:

en
de

А вот коды для австралийской английской и австрийской немецкой подкультур:

en-AU
de-AT

Культура представляется в .NET с помощью класса System.Globalization.CultureInfo. Просмотреть текущую культуру в приложении можно следующим образом:

```
Console.WriteLine (System.Threading.Thread.CurrentCulture);  
Console.WriteLine (System.Threading.Thread.CurrentCultureUICulture);
```

Выполнение такого кода на компьютере, локализованном для Австралии, демонстрирует разницу между двумя указанными свойствами:

en-AU
en-US

Свойство CurrentCulture отражает региональные параметры в панели управления Windows, тогда как свойство CurrentUICulture указывает язык пользовательского интерфейса ОС.

Региональные параметры включают аспекты вроде часового пояса, а также форматов для валюты и дат. Свойство CurrentCulture определяет стандартное поведение для функций, подобных DateTime.Parse. Региональные параметры могут быть настроены в точке, где они больше не соответствуют какой-либо культуре.

Свойство `CurrentUICulture` определяет язык, посредством которого компьютер взаимодействует с пользователем. Австралия не нуждается в отдельной версии английского языка для данной цели, поэтому используется версия английского, принятая в США. Например, если в связи с работой приходится несколько месяцев проводить в Австрии, то имеет смысл изменить текущую культуру в панели управления на немецкий язык в Австрии. Однако в случае неумения говорить по-немецки свойство `CurrentUICulture` может остаться установленным в английский язык, принятый в США.

Для определения корректной подчиненной сборки, подлежащей загрузке, объект `ResourceManager` по умолчанию применяет свойство `CurrentUICulture` текущего потока. При загрузке ресурсов объект `ResourceManager` использует механизм обхода. Если сборка для подкультуры определена, то она и будет применяться; в противном случае будет задействована обобщенная культура. Если обобщенная культура отсутствует, тогда будет произведен возврат к стандартной культуре в главной сборке.

Загрузка, распознавание и изолирование сборок

Загрузка сборки из известного местоположения представляет собой относительно несложный процесс. Мы будем его называть просто *загрузкой сборок*.

Однако чаще вам (или среде CLR) придется загружать сборку, зная только ее полное (или простое) имя. Процесс называется *распознаванием сборок*. Распознавание сборок отличается от загрузки тем, что сборку сначала нужно найти.

Распознавание сборок инициируется в двух сценариях:

- средой CLR, когда ей необходимо распознать зависимость;
- явно, когда вы вызываете метод вроде `Assembly.Load(AssemblyName)`.

В качестве иллюстрации первого сценария рассмотрим приложение, состоящее из главной сборки и нескольких статически ссылаемых библиотечных сборок (зависимостей), как показано в следующем примере:

```
AdventureGame.dll      // Главная сборка
Terrain.dll           // Ссылаемая сборка
UIEngine.dll          // Ссылаемая сборка
```

Под “статически ссылаемой” мы подразумеваем, что сборка `AdventureGame.dll` была скомпилирована со ссылками на сборки `Terrain.dll` и `UIEngine.dll`. Сам компилятор не нуждается в выполнении распознавания сборок, т.к. ему было сообщено (либо явно, либо инструментом MSBuild), где найти `Terrain.dll` и `UIEngine.dll`. На этапе компиляции он записывает *полные имена* сборок `Terrain` и `UIEngine` в метаданные `AdventureGame.dll`, но не дает информации, где они находятся. Таким образом, во время выполнения сборки `Terrain` и `UIEngine` должны быть распознаны.

Загрузка и распознавание сборки обрабатывается контекстом загрузки сборки (`Assembly Load Context — ALC`), а именно — экземпляром клас-

са `AssemblyLoadContext` из пространства имен `System.Runtime.Loader`. Поскольку `AdventureGame.dll` является главной сборкой в приложении, для распознавания ее зависимостей среда CLR использует *стандартный контекст ALC* (`AssemblyLoadContext.Default`). Стандартный контекст ALC распознает зависимости, сначала производя поиск файла по имени `AdventureGame.deps.json` (который описывает, где искать зависимости), а если упомянутый файл отсутствует, тогда стандартный контекст ALC просмотрит базовую папку приложения и найдет там `Terrain.dll` и `UIEngine.dll`. (Стандартный контекст ALC также распознает сборки исполняющей среды .NET.)

Как разработчик вы можете динамически загружать дополнительные сборки во время выполнения программы. Например, вы можете принять решение упаковать необязательные функциональные средства в сборки, которые будут разворачиваться, только когда они приобретены. В таком случае при наличии дополнительных сборок их можно было бы загружать с помощью вызова `Assembly.Load(AssemblyName)`.

Более сложный пример предусматривает реализацию системы подключаемых модулей, позволяющей пользователю предоставлять сторонние сборки, которые ваше приложение обнаруживает и загружает во время выполнения, чтобы расширить свою функциональность. Здесь возникает сложность, потому что каждая подключаемая сборка может иметь собственные зависимости, которые тоже должны быть распознаны.

За счет создания подкласса класса `AssemblyLoadContext` и переопределения его метода распознавания сборок (`Load`) вы можете управлять тем, каким образом подключаемый модуль будет искать свои зависимости. Скажем, вы можете решить, что подключаемый модуль должен находиться в собственной папке и там же обязаны располагаться его зависимости.

Контексты ALC служат еще одной цели: создавая отдельный экземпляр `AssemblyLoadContext` для каждого набора (подключаемый модуль плюс зависимости), вы можете сохранять наборы изолированными, гарантируя тем самым, что их зависимости будут загружаться параллельно и не мешать друг другу (и размещающему приложению). Например, каждый контекст может иметь свою версию JSON.NET. Следовательно, в дополнение к загрузке и распознаванию контексты ALC также предлагают механизм для изолирования. При определенных условиях контексты ALC можно даже *выгружать*, освобождая занимаемую ими память.

В настоящем разделе подробно обсуждается каждый из этих принципов, а также рассматриваются перечисленные ниже темы:

- как контексты ALC обрабатывают загрузку и распознавание;
- роль стандартного контекста ALC;
- метод `Assembly.Load` и контекстные ALC;
- как использовать класс `AssemblyDependencyResolver`;
- как загружать и распознавать неуправляемые библиотеки;
- выгрузка контекстов ALC;
- унаследованные методы загрузки сборок.

Затем мы применим теорию на практике и продемонстрируем процесс написания системы подключаемых модулей с изоляцией контекстов ALC.



Класс `AssemblyLoadContext` появился в .NET 5+ и .NET Core. В инфраструктуре .NET Framework контексты ALC присутствовали, но были ограниченными и скрытыми: единственный способ создания и непрямого взаимодействия с ними предусматривал использование статических методов `LoadFile(string)`, `LoadFrom(string)` и `Load(byte[])` класса `Assembly`. По сравнению с API-интерфейсом контекстов ALC эти методы обладают низкой гибкостью, а их применение может приводить к неожиданностям (особенно при обработке зависимостей). По указанной причине в .NET 5+ и .NET Core лучше отдавать предпочтение явному использованию API-интерфейса `AssemblyLoadContext`.

Контексты загрузки сборок

Как только что отмечалось, класс `AssemblyLoadContext` несет ответственность за загрузку и распознавание сборок, а также за предоставление механизма для изолирования.

Каждый .NET-объект `Assembly` относится в точности к одному объекту `AssemblyLoadContext`. Вот как можно получить контекст ALC для сборки:

```
Assembly assem = Assembly.GetExecutingAssembly();
AssemblyLoadContext context = AssemblyLoadContext.GetLoadContext(assem);
Console.WriteLine(context.Name);
```

И наоборот, контекст ALC можно считать “содержащим” или “владеющим” сборками, которые легко получить через его свойство `Assemblies`. Продолжим предыдущий пример:

```
foreach (Assembly a in context.Assemblies)
    Console.WriteLine(a.FullName);
```

Класс `AssemblyLoadContext` также имеет статическое свойство `All`, которое позволяет организовать перечисление по всем контекстам ALC.

Создать новый контекст ALC можно просто за счет создания экземпляра `AssemblyLoadContext` и предоставления имени (имя удобно при отладке), хотя чаще всего вы будете сначала создавать подкласс класса `AssemblyLoadContext`, чтобы появилась возможность реализовать логику для распознавания зависимостей; другими словами, загружать сборку по ее имени.

Загрузка сборок

Класс `AssemblyLoadContext` предлагает следующие методы для явной загрузки сборки в контекст:

```
public Assembly LoadFromAssemblyPath(string assemblyPath);
public Assembly LoadFromStream(Stream assembly, Stream assemblySymbols);
```

Первый метод загружает сборку из файла по указанному пути, а второй — из объекта `Stream` (который может поступать прямо из памяти). Второй параметр необязателен и соответствует содержимому файла отладки проекта (.pdb),

который делает возможным включение в трассировку стека информации об исходном коде во время его выполнения (полезно при сообщении об исключениях).

Для обоих методов распознавание не происходит.

Показанный далее код загружает сборку `c:\temp\foo.dll` в ее собственный контекст ALC:

```
var alc = new AssemblyLoadContext ("Test");
Assembly assem = alc.LoadFromAssemblyPath (@"c:\temp\foo.dll");
```

Если сборка допустима, тогда загрузка всегда будет успешной при условии соблюдения одного важного правила: *простое имя* сборки должно быть уникальным внутри ее контекста ALC. Это означает, что вы не сможете загрузить несколько версий сборки с тем же самым именем внутрь единственного контекста ALC; в таком случае потребуется создать дополнительные контексты ALC. Вот как можно было бы загрузить еще одну копию `foo.dll`:

```
var alc2 = new AssemblyLoadContext ("Test 2");
Assembly assem2 = alc2.LoadFromAssemblyPath (@"c:\temp\foo.dll");
```

Обратите внимание, что типы, которые происходят из разных объектов `Assembly`, не будут совместимыми, даже если в остальном сборки идентичны. В нашем примере типы в `assem` несовместимы с типами в `assem2`.

После загрузки сборку нельзя выгрузить, кроме как путем выгрузки ее контекста ALC (см. раздел “Выгрузка контекстов ALC” далее в главе). Среда CLR поддерживает блокировку файла в течение периода времени, пока он загружен.



Вы можете избежать блокировки файла, загружая сборку через байтовый массив:

```
bytes[] bytes = File.ReadAllBytes (@"c:\temp\foo.dll");
var ms = new MemoryStream (bytes);
var assem = alc.LoadFromStream (ms);
```

Такой прием обладает двумя недостатками.

- Свойство `Location` сборки окажется пустым. Иногда полезно знать, откуда была загружена сборка (и некоторые API-интерфейсы полагаются на то, что свойство `Location` заполнено).
- Расход закрытой памяти должен немедленно увеличиться, чтобы полностью уместить сборку. Если взамен вы загружаете по имени файла, то среда CLR применяет размещенные в памяти файлы, делая возможным ленивую загрузку и совместное использование данных разными процессами. Кроме того, в случае нехватки памяти ОС может освободить память, занимаемую сборкой, и при необходимости загрузить ее заново, не осуществляя запись в файл подкачки.

LoadFromAssemblyName

Класс `AssemblyLoadContext` также предлагает метод для загрузки сборки по имени:

```
public Assembly LoadFromAssemblyName (AssemblyName assemblyName);
```

В отличие от двух только что обсужденных методов вы не передаете какую-либо информацию для указания, где находится сборка; взамен вы инструктируете контекст ALC о том, что сборку нужно распознать.

Распознавание сборок

Предыдущий метод инициирует *распознавание сборок*. Среда CLR тоже инициирует распознавание сборок, когда загружает зависимости. Скажем, пусть сборка A статически ссылается на сборку B. Чтобы распознать сборку B, среда CLR инициирует распознавание сборок в том *контексте ALC*, куда была загружена сборка A.



Среда CLR распознает зависимости путем запуска распознавания сборок безотносительно к тому, куда загружена запускающая сборка — в стандартный или в специальный контекст ALC. Разница лишь в том, что в стандартном контексте ALC правила распознавания жестко закодированы, а в специальном контексте ALC вы пишете правила самостоятельно.

Ниже описано, что затем происходит.

1. Среда CLR сначала проверяет, происходило или нет такое распознавание в этом контексте ALC (с совпадающим полным именем сборки); если происходило, тогда она возвращает объект `Assembly`, который возвращала ранее.
2. В противном случае среда CLR вызывает (виртуальный защищенный) метод `Load` на контексте ALC, который делает работу, связанную с нахождением и загрузкой сборки. Метод `Load` стандартного контекста ALC применяет правила, рассматриваемые в разделе “Стандартный контекст ALC” далее в главе. В ситуации со специальным контекстом ALC выбор местоположения для сборки целиком зависит от вас. Например, вы можете просмотреть какой-то каталог и при обнаружении сборки вызвать метод `LoadFromAssemblyPath`. Также совершенно законно возвращать уже загруженную сборку из того же самого или другого контекста ALC (мы продемонстрируем это в разделе “Реализация системы подключаемых модулей” далее в главе).
3. Если на шаге 2 возвращается `null`, тогда среда CLR вызывает метод `Load` на стандартном контексте ALC (что служит удобным “запасным вариантом” для распознавания сборок исполняющей среды .NET и общих сборок приложения).
4. Если на шаге 3 возвращается `null`, тогда среда CLR генерирует события `Resolving` для обоих контекстов ALC — сначала для стандартного, затем для специального.
5. (В целях совместимости с .NET Framework): если сборка все еще не была распознана, тогда генерируется событие `AppDomain.CurrentDomain.AssemblyResolve`.



После завершения такого процесса среда CLR проводит “контроль корректности” определенного вида, чтобы гарантировать, что любая загруженная сборка имеет имя, совместимое с запрошенным. Простое имя обязано совпадать; маркер открытого ключа должен совпадать, если он указан. Версия не обязательно должна совпадать — она может быть выше или ниже запрошенной.

Из всего изложенного можно сделать вывод, что существуют два способа реализации распознавания сборок в специальном контексте ALC.

- Переопределение метода Load контекста ALC. Это дает вашему контексту ALC “первое слово” касательно происходящего, что обычно желательно (и важно, когда вам нужна изоляция).
- Обработка события Resolving контекста ALC. Оно генерируется только после того, как стандартному контексту ALC не удалось распознать сборку.



Если вы присоедините к событию Resolving несколько обработчиков событий, тогда “победит” тот, который первым возвратит значение, отличающееся от null.

В целях иллюстрации давайте предположим, что мы хотим загрузить сборку, о которой нашему главному приложению ничего не было известно на этапе компиляции, имеющую имя foo.dll и расположенную в папке c:\temp (т.е. не там, где находится приложение). Вдобавок пусть сборка foo.dll имеет закрытую зависимость от bar.dll. Нам необходимо удостовериться в том, что при загрузке сборки c:\temp\foo.dll и выполнении ее кода сборка c:\temp\bar.dll может быть корректно распознана. Нам также нужно убедиться в том, что сборка foo и ее закрытая зависимость bar не мешают работе главного приложения.

Начнем с реализации специального контекста ALC, в котором переопределен метод Load:

```
using System.IO;
using System.Runtime.Loader;

class FolderBasedALC : AssemblyLoadContext
{
    readonly string _folder;
    public FolderBasedALC (string folder) => _folder = folder;
    protected override Assembly Load (AssemblyName assemblyName)
    {
        // Попытка найти сборку:
        string targetPath = Path.Combine (_folder, assemblyName.Name + ".dll");
        if (File.Exists (targetPath))
            return LoadFromAssemblyPath (targetPath); // Загрузить сборку
        return null; // Нам не удалось ее найти: это может
                    // быть сборка исполняющей среды .NET
    }
}
```

Обратите внимание, что в методе Load мы возвращаем null, если файл сборки отсутствует. Это важная проверка, потому что у foo.dll также имеются зависимости от сборок BCL исполняющей среды .NET; следовательно, метод Load будет вызываться для таких сборок, как System.Runtime. Возвращая null, мы позволяем среде CLR обратиться за помощью к стандартному контексту ALC, который корректно распознает сборки подобного рода.



Следует отметить, что мы не пытаемся загрузить в свой контекст ALC сборки BCL исполняющей среды .NET. Дело в том, что системные сборки не рассчитаны на запуск вне стандартного контекста ALC, и попытка их загрузки в свой контекст ALC может привести к некорректному поведению, ухудшению производительности и непредсказуемой несовместимости типов.

Вот как можно было бы использовать наш специальный контекст ALC для загрузки сборки foo.dll из c:\temp:

```
var alc = new FolderBasedALC (@"c:\temp");
Assembly foo = alc.LoadFromAssemblyPath (@"c:\temp\foo.dll");
...
```

Когда позже мы начнем обращаться к коду внутри сборки foo, в какой-то момент среде CLR потребуется распознать зависимость от bar.dll. Именно тогда запустится метод Load специального контекста ALC и успешно найдет сборку bar.dll в c:\temp.

В данном случае наш метод Load также способен распознавать foo.dll, так что мы можем упростить код:

```
var alc = new FolderBasedALC (@"c:\temp");
Assembly foo = alc.LoadFromAssemblyName (new AssemblyName ("foo"));
...
```

А теперь давайте рассмотрим альтернативное решение: вместо создания подкласса класса AssemblyLoadContext и переопределения метода Load можно было бы создать экземпляр простого класса AssemblyLoadContext и обработать событие Resolving:

```
var alc = new AssemblyLoadContext ("test");
alc.Resolving += (loadContext, assemblyName) =>
{
    string targetPath = Path.Combine (@"c:\temp", assemblyName.Name + ".dll");
    return alc.LoadFromAssemblyPath (targetPath); // Загрузить сборку
};
Assembly foo = alc.LoadFromAssemblyName (new AssemblyName ("foo"));
```

Обратите внимание, что нам не нужно проверять, существует ли сборка. Поскольку событие Resolving инициируется *после* того, как стандартный контекст ALC получил шанс распознать сборку (и только когда он потерпел неудачу), наш обработчик не запускается для сборок BCL исполняющей среды .NET. Это делает решение более простым, хотя существует и затруднение. Вспомните, что в имеющемся сценарии главному приложению ничего не было известно о

foo.dll или bar.dll на этапе компиляции. Но вполне вероятно, что главное приложение зависит от сборок под названием foo.dll или bar.dll. В таком случае событие Resolving никогда не возникнет, а взамен загрузятся сборки foo и bar. Другими словами, добиться изоляции нам не удастся.



Наш класс FolderBasedALC хороший для иллюстрации концепции распознавания сборок, но в реальности от него меньше пользы, т.к. он не способен справляться с зависимостями NuGet, специфичными к платформе, и с зависимостями NuGet, относящимися к разработке (в случае библиотечных проектов). В разделе “Класс AssemblyDependencyResolver” далее в главе будет описано решение этой задачи, а в разделе “Реализация системы подключаемых модулей” приводится подробный пример.

Стандартный контекст ALC

Когда приложение запускается, среда CLR присваивает статическому свойству AssemblyLoadContext.Default специальный контекст ALC. Стандартный контекст ALC — это место, куда загружается стартовая сборка вместе со своими статически ссылаемыми зависимостями и сборками BCL исполняющей среды .NET.

Стандартный контекст ALC сначала просматривает пути *стандартного зондирования* для автоматического распознавания сборок (см. раздел “Стандартное зондирование” далее в главе). Обычно они соответствуют местоположениям, указанным в файлах .deps.json и .runtimeconfig.json приложения.

Если контекст ALC не смог найти сборку в путях стандартного зондирования, тогда инициируется его событие Resolving. Обработка этого события позволяет загружать сборку из других местоположений, что означает возможность развертывания зависимостей приложения в дополнительных местах, таких как подпапки, общие папки или даже двоичные ресурсы внутри размещающей сборки:

```
AssemblyLoadContext.Default.Resolving += (loadContext, assemblyName) =>
{
    // Попробовать найти assemblyName, возвратив объект Assembly или null
    // Обычно после нахождения файла будет вызываться метод LoadFromAssemblyPath
    // ...
};
```

Событие Resolving в стандартном контексте ALC также инициируется, когда специальный контекст ALC потерпел неудачу с распознаванием (выражаясь по-другому, когда его метод Load вернул null), и стандартному контексту ALC не удается распознать сборку.

Кроме того, вы можете загружать сборки в стандартный контекст ALC и за пределами обработчика события Resolving. Однако прежде чем продолжить, вы должны сначала выяснить, есть ли возможность решить задачу лучше за счет применения отдельного контекста ALC или с помощью подходов, описанных в следующем разделе (которые используют *исполняемые и контекстные*

ALC). Жесткая привязка к стандартному контексту ALC делает ваш код хрупким, потому что он не может быть изолирован как единое целое (скажем, посредством инфраструктур модульного тестирования или LINQPad).

Если все-таки хотите продолжить, тогда предпочтительнее вызывать *метод распознавания* (т.е. `LoadFromAssemblyName`), а не *метод загрузки* (такой как `LoadFromAssemblyPath`) — особенно если ваша сборка является статически ссылаемой. Причина в том, что сборка может быть уже загружена, и в такой ситуации метод `LoadFromAssemblyName` возвратит объект загруженной сборки, в то время как метод `LoadFromAssemblyPath` сгенерирует исключение.

(В случае метода `LoadFromAssemblyPath` вы также можете рискнуть загрузить сборку из места, которое несовместимо с местоположениями, где ее искал бы стандартный механизм распознавания контекста ALC.)

Если сборка находится в месте, где контекст ALC не способен найти ее автоматически, вы все равно можете следовать этой процедуре и дополнительно обработать событие `Resolving` контекста ALC.

Следует отметить, что при вызове метода `LoadFromAssemblyName` не нужно указывать полное имя; подойдет простое имя (и оно допустимо, даже если сборка строго именована):

```
AssemblyLoadContext.Default.LoadFromAssemblyName ("System.Xml");
```

Тем не менее, если вы включаете маркер открытого ключа, то он обязан совпадать с тем, что загружается.

Стандартное зондирование

Пути стандартного зондирования обычно включают перечисленные ниже.

- Пути, указанные в файле `AppName.deps.json` (где `AppName` — имя главной сборки приложения). Если этот файл отсутствует, тогда взамен используется базовая папка приложения.
- Папки, содержащие системные сборки исполняющей среды .NET (если приложение зависит от них).

MSBuild автоматически генерирует файл по имени `AppName.deps.json`, в котором описано, где искать все зависимости приложения. Сюда входят сборки, независимые от платформы, которые размещаются в базовой папке приложения, и сборки, специфичные к платформе, которые находятся в подкаталоге `runtimes\` внутри подпапки, подобной `win` или `unix`.

Пути, указанные в сгенерированном файле `.deps.json`, являются относительными к базовой папке приложения — или любым дополнительным папкам, которые вы указываете в разделе `additionalProbingPaths` конфигурационных файлов `AppName.runtimeconfig.json` и/или `AppName.runtimeconfig.dev.json` (последний предназначен только для среды разработки).

“Текущий” контекст ALC

В предыдущем разделе мы предостерегали от явной загрузки сборок в стандартный контекст ALC. Взамен вы обычно хотите загружать/распознавать в “текущем” контексте ALC.

В большинстве случаев “текущий” контекст ALC содержит сборку, выполняющуюся в настоящее время:

```
var executingAssem = Assembly.GetExecutingAssembly();
var alc = AssemblyLoadContext.GetLoadContext(executingAssem);

Assembly assem = alc.LoadFromAssemblyName(...); // для распознавания по имени
// ИЛИ: = alc.LoadFromAssemblyPath(...); // для загрузки по пути
```

Ниже показан более гибкий и явный способ получения контекста ALC:

```
var myAssem = typeof(SomeTypeInMyAssembly).Assembly;
var alc = AssemblyLoadContext.GetLoadContext(myAssem);
...
```

Иногда вывести “текущий” контекст ALC невозможно. Например, предположим, что вы отвечали за реализацию двоичного сериализатора .NET (сериализация описана в дополнительных материалах, доступных для загрузки на веб-сайте издательства). Сериализатор подобного рода записывает полные имена сериализуемых типов (включая имена их сборок), которые должны быть распознаны во время десериализации. Вопрос в том, какой контекст ALC нужно задействовать? Проблема с использованием контекста выполняющейся сборки связана с тем, что он возвратит объект сборки, где содержится десериализатор, а не сборки, которая *вызывает* десериализатор.

Лучшее решение — не угадывать, а запрашивать:

```
public object Deserialize(Stream stream, AssemblyLoadContext alc)
{
    ...
}
```

Явный подход максимизирует гибкость и минимизирует шансы допустить ошибку. Теперь вызывающий код может решить, что должно считаться “текущим” контектом ALC:

```
var assem = typeof(SomeTypeThatIWillBeDeserializing).Assembly;
var alc = AssemblyLoadContext.GetLoadContext(assem);
var object = Deserialize(someStream, alc);
```

Метод `Assembly.Load` и контекстные ALC

Чтобы помочь с распространенным сценарием загрузки сборки в исполняемый в текущий момент контекст ALC, т.е.:

```
var executingAssem = Assembly.GetExecutingAssembly();
var alc = AssemblyLoadContext.GetLoadContext(executingAssem);
Assembly assem = alc.LoadFromAssemblyName(...);
```

в классе `Assembly` определен следующий метод:

```
public static Assembly Load(string assemblyString);
```

а также функционально идентичная версия, принимающая объект AssemblyName:

```
public static Assembly Load (AssemblyName assemblyRef);
```

(Не путайте эти методы с унаследованным методом Load(byte[]), который ведет себя совершенно иначе; см. раздел “Унаследованные методы загрузки” далее в главе.)

Как и с методом LoadFromAssemblyName, у вас есть возможность указать простое, частичное или полное имя сборки:

```
Assembly a = Assembly.Load ("System.Private.Xml");
```

Приведенный выше код загружает сборку System.Private.Xml в тот контекст ALC, куда была загружена *сборка исполняемого кода*.

В данном случае было указано простое имя. Следующие строки также допустимы и приводят к одному и тому же результату в .NET:

```
"System.Private.Xml, PublicKeyToken=cc7b13ffcd2ddd51"  
"System.Private.Xml, Version=4.0.1.0"  
"System.Private.Xml, Version=4.0.1.0, PublicKeyToken=cc7b13ffcd2ddd51"
```

Если вы решите указать маркер открытого ключа, тогда он должен совпадать с тем, что был загружен.



Разработчики в сети MSDN предостерегают от загрузки сборки с частичным именем, рекомендуя указывать точную версию и маркер открытого ключа. Их обоснование основано на факторах, относящихся к .NET Framework, таких как влияние глобального кеша сборок и безопасности доступа кода (Code Access Security). В .NET 5+ и .NET Core упомянутые факторы отсутствуют, поэтому загружать сборку с простым или частичным именем в целом безопасно.

Оба метода предназначены исключительно для *распознавания*, так что указывать путь к файлу нельзя. (Если вы заполните свойство CodeBase в объекте AssemblyName, то он будет проигнорирован.)



Избегайте попасть в ловушку с применением метода Assembly.Load для загрузки статически ссылаемой сборки. Все, что вам понадобится сделать в данном случае — сослаться на тип в сборке и получить сборку из него:

```
Assembly a = typeof (System.Xml.Formatting).Assembly;
```

Или можете поступить даже следующим образом:

```
Assembly a = System.Xml.Formatting.Indented.GetType ().Assembly;
```

Это предотвращает жесткое кодирование имени сборки (которое в будущем может измениться) наряду с обеспечением запуска распознавания сборки в контексте ALC *исполняемого кода* (как произошло бы с методом Assembly.Load).

Если бы вы писали код метода Assembly.Load самостоятельно, то он выглядел бы (почти) так:

```
[MethodImpl(MethodImplOptions.NoInlining)]
Assembly Load (string name)
{
    Assembly callingAssembly = Assembly.GetCallingAssembly();
    var callingAcl = AssemblyLoadContext.GetLoadContext(callingAssembly);
    return callingAcl.LoadFromAssemblyName (new AssemblyName (name));
}
```

Метод EnterContextualReflection

Стратегия использования контекста ALC вызываемой сборки, принятая в `Assembly.Load`, терпит неудачу, когда метод `Assembly.Load` вызывается через посредника, такого как десериализатор или механизм запуска модульного теста. Если посредник определен в другой сборке, тогда вместо контекста загрузки вызываемой сборки применяется контекст загрузки сборки посредника.



Такой сценарий был описан ранее, когда речь шла о том, каким образом вы могли бы реализовать десериализатор. В подобных случаях идеальное решение — вынудить вызывающий код указывать контекст ALC, а не выводить его с помощью `Assembly.Load(string)`. Но поскольку версии .NET 5+ и .NET Core произошли от .NET Framework, где изоляция достигалась посредством доменов приложений, а не контекстов ALC, идеальное решение не является превалирующим, и временами `Assembly.Load(string)` нецелесообразно используется в сценариях, в которых контекст ALC не может быть надежно выведен. Примером служит двоичный сериализатор .NET.

Чтобы позволить методу `Assembly.Load` по-прежнему работать в таких сценариях, разработчики из Microsoft добавили в класс `AssemblyLoadContext` метод по имени `EnterContextualReflection`. Он присваивает контекст ALC свойству `AssemblyLoadContext.CurrentContextualReflectionContext`. Хотя это статическое свойство, его значение хранится в переменной `AsyncLocal`, так что оно способно удерживать в разных потоках отдельные значения (но все же предохраняться в течение асинхронных операций).

Если свойство `AssemblyLoadContext.CurrentContextualReflectionContext` не равно `null`, то метод `Assembly.Load` автоматически применяет его вместо обращения к контексту ALC:

```
Method1();
var myALC = new AssemblyLoadContext ("test");
using (myALC.EnterContextualReflection())
{
    Console.WriteLine (
        AssemblyLoadContext.CurrentContextualReflectionContext.Name); // тест
    Method2();
}
// После освобождения EnterContextualReflection() больше не действует.
Method3();
void Method1 () => Assembly.Load (...); // Будет использоваться
                                                // обращение к контексту ALC
void Method2 () => Assembly.Load (...); // Будет использоваться myALC
void Method3 () => Assembly.Load (...); // Будет использоваться
                                                // обращение к контексту ALC
```

Ранее мы демонстрировали, как можно было бы написать метод, функционально подобный `Assembly.Load`. Вот более точная версия, которая учитывает контекст контекстной рефлексии:

```
[MethodImpl(MethodImplOptions.NoInlining)]
Assembly Load (string name)
{
    var alc = AssemblyLoadContext.CurrentContextualReflectionContext;
    ?? AssemblyLoadContext.GetLoadContext (Assembly.GetCallingAssembly ());
    return alc.LoadFromAssemblyName (new AssemblyName (name));
}
```

Хотя контекст контекстной рефлексии может быть полезен для разрешения запуска унаследованного кода, более надежное решение (как было описано ранее в главе) предусматривает изменение кода, который вызывает `Assembly.Load`, чтобы взамен он вызывал метод `LoadFromAssemblyName` на контексте ALC, переданном вызывающим кодом.



Инфраструктура .NET Framework не имеет эквивалента метода `EnterContextualReflection` и не нуждается в нем, несмотря на наличие таких же методов `Assembly.Load`. Дело в том, что изоляция в .NET Framework достигается главным образом с помощью **доменов приложений**, а не контекстов ALC. Домены приложений обеспечивают более сильную модель изоляции, в соответствии с которой каждый домен приложения имеет собственный стандартный контекст загрузки, поэтому изоляция по-прежнему может работать даже при использовании только стандартного контекста загрузки.

Загрузка и распознавание неуправляемых библиотек

Контексты ALC также способны загружать и распознавать низкоуровневые библиотеки. Низкоуровневое распознавание запускается, когда производится вызов внешнего метода, помеченного атрибутом `[DllImport]`:

```
[DllImport ("SomeNativeLibrary.dll")]
static extern int SomeNativeMethod (string text);
```

Поскольку полный путь в атрибуте `[DllImport]` не был указан, вызов метода `SomeNativeMethod` запускает распознавание в контексте ALC, содержащем сборку, в которой определен метод `SomeNativeMethod`.

В контексте ALC виртуальный метод *распознавания* называется `LoadUnmanagedDll`, а метод *загрузки* — `LoadUnmanagedDllFromPath`:

```
protected override IntPtr LoadUnmanagedDll (string unmanagedDllName)
{
    // Определить полный путь к unmanagedDllName...
    string fullPath = ...
    return LoadUnmanagedDllFromPath (fullPath); // Загрузить DLL-библиотеку
}
```

Если вы не в состоянии определить местонахождение файла, тогда можете возвратить `IntPtr.Zero`. Затем среда CLR инициирует событие `Resolving UnmanagedDll` контекста ALC.

Интересно отметить, что метод LoadUnmanagedDllFromPath является защищенным, поэтому обычно у вас не будет возможности вызвать его из обработчика событий ResolvingUnmanagedDll. Однако вы можете добиться того же результата, вызывая статический метод NativeLibrary.Load:

```
someALC.ResolvingUnmanagedDll += (requestingAssembly, unmanagedDllName) =>
{
    return NativeLibrary.Load ("(полный путь к неуправляемой DLL-библиотеке)");
};
```

Несмотря на то что низкоуровневые библиотеки, как правило, распознаются и загружаются контекстами ALC, они им не “принадлежат”. После загрузки низкоуровневая библиотека становится самостоятельной и возлагает на себя ответственность за распознавание любых кратковременных зависимостей, которые она может иметь. Более того, низкоуровневые библиотеки являются глобальными по отношению к процессу, так что загрузить две разных версии низкоуровневой библиотеки не удастся, если они имеют одно и то же имя файла.

Класс AssemblyDependencyResolver

В разделе “Стандартное зондирование” ранее в главе упоминалось о том, что стандартный контекст ALC читает файлы .deps.json и .runtimeconfig.json при их наличии с целью определения, где искать зависимости NuGet, специфичные к платформе и относящиеся к разработке.

Если вы хотите загрузить сборку в специальный контекст ALC, который имеет зависимости, специфичные к платформе, или зависимости NuGet, тогда вам придется как-то воспроизвести эту логику. Вы могли бы достичь цели за счет разбора конфигурационных файлов и аккуратного следования правилам для имен, специфичных к платформе. Тем не менее, поступать так не только сложно, но вдобавок написанный вами код перестанет работать, если в будущей версии .NET правила изменятся.

Задачу решает класс AssemblyDependencyResolver. Для его использования необходимо создать экземпляр с указанием пути к сборке, чьи зависимости нужно прозондировать:

```
var resolver = new AssemblyDependencyResolver (@"c:\temp\foo.dll");
```

Затем, чтобы найти путь к зависимости, понадобится вызвать метод ResolveAssemblyToPath:

```
string path = resolver.ResolveAssemblyToPath (new AssemblyName ("bar"));
```

Если файл .deps.json отсутствует (или файл .deps.json не содержит ничего, имеющего отношения к bar.dll), тогда path получит значение c:\temp\bar.dll.

Аналогичным образом можно распознавать неуправляемые зависимости, вызывая метод ResolveUnmanagedDllToPath.

Замечательный способ проиллюстрировать более сложный сценарий предусматривает создание проекта консольного приложения по имени ClientApp и добавление ссылки NuGet на Microsoft.Data.SqlClient. Введите показанный далее код класса:

```
using Microsoft.Data.SqlClient;  
namespace ClientApp  
{  
    public class Program  
    {  
        public static SqlConnection GetConnection() => new SqlConnection();  
        static void Main() => GetConnection(); // Проверить, распознан ли он  
    }  
}
```

Теперь скомпилируйте приложение и загляните в выходную папку: вы увидите файл по имени Microsoft.Data.SqlClient.dll. Однако этот файл *никогда не загружается* при запуске, а попытка загрузить его явно приводит к генерации исключения. Фактически загружаемая сборка находится в подпапке runtimes\win (или runtimes/unix); стандартному контексту ALC известно о необходимости ее загрузки, т.к. он разбирает файл ClientApp.deps.json.

Если бы вы попытались загрузить сборку ClientApp.dll из другого приложения, то пришлось бы реализовать контекст ALC, который способен распознать ее зависимость от Microsoft.Data.SqlClient.dll. При этом было бы недостаточно просто заглянуть в папку, в которой находится ClientApp.dll (как делалось в разделе “Распознавание сборок” ранее в главе). Взамен необходимо применять класс AssemblyDependencyResolver, чтобы выяснить, где расположен файл ClientApp.dll для используемой платформы:

```
string path = @"C:\source\ClientApp\bin\Debug\netcoreapp3.0\ClientApp.dll";  
var resolver = new AssemblyDependencyResolver (path);  
var sqlClient = new AssemblyName ("Microsoft.Data.SqlClient");  
Console.WriteLine (resolver.ResolveAssemblyToPath (sqlClient));
```

На машине Windows вывод будет следующим:

```
C:\source\ClientApp\bin\Debug\netcoreapp3.0\runtimes\win\lib\netcoreapp2.1  
\Microsoft.Data.SqlClient.dll
```

Полный пример будет представлен в разделе “Реализация системы подключаемых модулей” далее в главе.

Выгрузка контекстов ALC

В простых случаях нестандартный контекст AssemblyLoadContext можно выгрузить, освободив память и сняв файловые блокировки с загруженных сборок. Чтобы это работало, контекст ALC должен быть создан с параметром isCollectible, установленным в true:

```
var alc = new AssemblyLoadContext ("test", isCollectible:true);
```

Для инициирования процесса выгрузки необходимо вызвать метод Unload на контексте ALC.

Модель выгрузки является кооперативной, а не вытесняющей. В случае выполнения любых методов в любых сборках контекста ALC выгрузка откладывается до тех пор, пока методы не завершатся.

Фактическая выгрузка происходит во время сборки мусора; она не начнется, если что-либо извне контекста ALC имеет любую (*не* слабую) ссылку на что-то внутри контекста ALC (включая объекты, типы и сборки). Нередко API-интерфейсы (в том числе из .NET BCL) кешируют объекты в статических полях или словарях — либо подписываются на события — и это легко приводит к созданию ссылок, которые будут препятствовать выгрузке, особенно когда код в контексте ALC использует внешние API-интерфейсы нетривиальным образом. Выяснить причину неудавшейся выгрузки сложно и придется применять инструменты вроде WinDbg.

Унаследованные методы загрузки

Если вы по-прежнему используете .NET Framework (или разрабатываете библиотеку, нацеленную на .NET Standard, и хотите поддерживать .NET Framework), то не сможете применять класс `AssemblyLoadContext`. В такой ситуации загрузка выполняется с использованием следующих методов:

```
public static Assembly LoadFrom (string assemblyFile);
public static Assembly LoadFile (string path);
public static Assembly Load (byte[] rawAssembly);
```

Методы `LoadFile` и `Load(byte[])` обеспечивают изоляцию, тогда как `LoadFrom` — нет. Распознавание достигается обработкой события `AssemblyResolve` домена приложения, которое похоже на событие `Resolving` стандартного контекста ALC.

Доступен также метод `Assembly.Load(string)`, предназначенный для запуска распознавания, который работает аналогичным образом.

Метод `LoadFrom`

Метод `LoadFrom` загружает сборку по заданному пути в стандартный контекст ALC. Он немного похож на метод `AssemblyLoadContext.Default.LoadFromAssemblyPath` за исключением перечисленных ниже аспектов.

- Если сборка с таким же простым именем уже присутствует в стандартном контексте ALC, тогда метод `LoadFrom` возвращает объект сборки, а не генерирует исключение.
- Если сборка с таким же простым именем *не* присутствует в стандартном контексте ALC, а загрузка произошла, тогда сборка назначается особый статус “`LoadFrom`”. Этот статус влияет на логику распознавания стандартного контекста ALC: если данная сборка имеет любые зависимости в *той же самой папке*, то такие зависимости будут распознаваться автоматически.



В .NET Framework имеется глобальный кеш сборок. Если сборка присутствует в глобальном кеше сборок, тогда среда CLR будет всегда загружать ее оттуда. Это применимо ко всем трем методам загрузки.

Способность метода LoadFrom автоматически распознавать кратковременные зависимости в той же самой папке может быть удобной — до тех пор, пока метод не загрузит сборку, которая загружаться не должна. Поскольку сценарий подобного рода может оказаться трудным в отладке, лучше использовать метод Load(string) или LoadFile и распознавать кратковременные зависимости, обрабатывая событие AssemblyResolve домена приложения. Такой подход позволяет вам решать, каким образом распознавать каждую сборку, и делает возможной отладку (создавая точку останова внутри обработчика события).

Методы LoadFile и Load(byte[])

Методы LoadFile и Load(byte[]) загружают сборку по заданному пути к файлу либо из байтового массива в новый контекст ALC. В отличие от LoadFrom эти методы обеспечивают изоляцию и позволяют загружать несколько версий той же самой сборки. Тем не менее, есть два предостережения:

- повторный вызов метода LoadFile с идентичным путем возвратит ранее загруженную сборку;
- в .NET Framework оба метода сначала проверяют глобальный кеш сборок и при наличии в нем сборки загружают из него.

Применяя методы LoadFile и Load(byte[]), вы получаете отдельный контекст ALC для каждой сборки (оставляя в стороне предостережения). В итоге становится возможной изоляция, хотя может усложниться управление.

Для распознавания зависимостей вы обрабатываете событие Resolving домена приложения, которое инициируется на всех контекстах ALC:

```
AppDomain.CurrentDomain.AssemblyResolve += (sender, args) =>
{
    string fullAssemblyName = args.Name;
    // Возвратить объект Assembly или null
    ...
};
```

Переменная args также включает свойство по имени RequestingAssembly, которое сообщает, какая сборка запустила распознавание.

После определения местоположения сборки можно вызвать метод Assembly.LoadFile для ее загрузки.



С помощью метода AppDomain.CurrentDomain.GetAssemblies можно организовать перечисление всех сборок, которые были загружены в текущий домен приложения. Прием работает и в .NET 5+, где он эквивалентен следующему коду:

```
AssemblyLoadContext.All.SelectMany (a => a.Assemblies)
```

Реализация системы подключаемых модулей

В целях полной демонстрации концепций, раскрытых в этом разделе, мы реализуем систему подключаемых модулей, которая использует выгружаемые контексты ALC для изоляции каждого подключаемого модуля.

Изначально система будет включать в себя три проекта .NET:

- **Plugin.Common** (библиотека). Определяет интерфейс, который будут реализовывать подключаемые модули.
- **Capitalizer** (библиотека). Подключаемый модуль, преобразующий строчные буквы в заглавные.
- **Plugin.Host** (консольное приложение). Определяет местоположение и активизирует подключаемые модули.

Давайте предположим, что проекты располагаются в перечисленных ниже каталогах:

```
c:\source\PluginDemo\Plugin.Common  
c:\source\PluginDemo\Capitalizer  
c:\source\PluginDemo\Plugin.Host
```

Все проекты будут ссылаться на библиотеку **Plugin.Common**, а другие ссылки между проектами отсутствуют.



Если бы проект **Plugin.Host** ссылался на **Capitalizer**, то не имело бы смысла реализовывать систему подключаемых модулей; основная идея в том, что подключаемые модули пишутся сторонними разработчиками после опубликования **Plugin.Host** и **Plugin.Common**.

В случае использования Visual Studio все три проекта удобно объединить в одно решение. Для этого щелкните правой кнопкой мыши на проекте **Plugin.Host**, выберите в контекстном меню пункт **Build Dependencies**⇒**Project Dependencies** (Зависимости построения⇒Зависимости проекта) и отметьте проект **Capitalizer**, что обеспечит компиляцию проекта **Capitalizer** при запуске проекта **Plugin.Host** без добавления ссылки.

Проект **Plugin.Common**

Мы начнем с проекта **Plugin.Common**. Наши подключаемые модули будут решать очень простую задачу, связанную с трансформацией строки. Вот как будет определен интерфейс:

```
namespace Plugin.Common  
{  
    public interface ITextPlugin  
    {  
        string TransformText (string input);  
    }  
}
```

Это все, что касается **Plugin.Common**.

Проект **Capitalizer** (подключаемый модуль)

Подключаемый модуль **Capitalizer** будет ссылаться на **Plugin.Common** и содержать единственный класс. Пока что мы оставим логику простой, чтобы у подключаемого модуля не было излишних зависимостей:

```
public class CapitalizerPlugin : Plugin.Common.ITextPlugin
{
    public string TransformText (string input) => input.ToUpper();
}
```

Если вы скомпилируете оба проекта и заглянете в выходную папку Capitalizer, то увидите следующие две сборки:

Capitalizer.dll	// Сборка подключаемого модуля
Plugin.Common.dll	// Ссылаемая сборка

Проект Plugin.Host

Проект Plugin.Host представляет собой консольное приложение с двумя классами. Первый класс — это специальный контекст ALC для загрузки подключаемого модуля:

```
class PluginLoadContext : AssemblyLoadContext
{
    AssemblyDependencyResolver _resolver;

    public PluginLoadContext (string pluginPath, bool collectible)
        // Назначить контексту дружественное имя для содействия в отладке
        : base (name: Path.GetFileName (pluginPath), collectible)
    {
        // Создать распознаватель, который поможет находить зависимости
        _resolver = new AssemblyDependencyResolver (pluginPath);
    }

    protected override Assembly Load (AssemblyName assemblyName)
    {
        // См. ниже:
        if (assemblyName.Name == typeof (ITextPlugin).Assembly.GetName().Name)
            return null;

        string target = _resolver.ResolveAssemblyToPath (assemblyName);
        if (target != null)
            return LoadFromAssemblyPath (target);

        // Может быть сборкой BCL. Позволить стандартному
        // контексту выполнить распознавание
        return null;
    }

    protected override IntPtr LoadUnmanagedDll (string unmanagedDllName)
    {
        string path = _resolver.ResolveUnmanagedDllToPath (unmanagedDllName);
        return path == null
            ? IntPtr.Zero
            : LoadUnmanagedDllFromPath (path);
    }
}
```

Конструктору передается путь к главной сборке подключаемых модулей и флаг, который указывает, должен ли контекст ALC подвергаться сборке мусора (чтобы его можно было выгружать).

Распознавание зависимостей обрабатывается в методе Load. Все подключаемые модули обязаны ссылаться на Plugin.Common, поэтому они в состоянии реализовывать ITextPlugin. Таким образом, метод Load в какой-то момент будет запущен, чтобы распознать Plugin.Common. Нам нужно проявить осторожность, поскольку выходная папка подключаемого модуля, скорее всего, содержит не только Capitalizer.dll, но и собственную копию Plugin.Common.dll. Если бы мы загрузили такую копию Plugin.Common.dll в PluginLoadContext, то получили бы в итоге две копии сборки: одну в стандартном контексте хоста и еще одну в контексте PluginLoadContext подключаемого модуля. Эти сборки не будут совместимыми, а хост сообщит о том, что подключаемый модуль не реализует ITextPlugin!

Чтобы решить проблему, мы явно проверяем данное условие:

```
if (assemblyName.Name == typeof (ITextPlugin).Assembly.GetName().Name)
    return null;
```

Возвращение null дает возможность распознать сборку стандартному контексту ALC хоста.



Вместо возвращения null мы могли бы возвращать typeof(ITextPlugin).Assembly, что также обеспечило бы корректную работу. Как мы можем быть уверены в том, что реализация ITextPlugin будет распознана контектом ALC хоста, а не контекстом PluginLoadContext? Вспомните, что наш класс PluginLoadContext определен в сборке Plugin.Host. Следовательно, любые типы, на которые вы статически ссылаетесь из этого класса, будут запускать распознавание сборки в контексте ALC, куда была загружена его сборка Plugin.Host.

После проверки общей сборки мы применяем класс AssemblyDependencyResolver для нахождения любых закрытых зависимостей, которые может иметь подключаемый модуль. (В данный момент их нет.)

Обратите внимание, что мы также переопределяем метод LoadUnmanagedDll, гарантируя тем самым, что если подключаемый модуль имеет неуправляемые зависимости, то они тоже корректно загружаются.

Второй класс в Plugin.Host является главной программой. Ради простоты давайте жестко закодируем путь к подключаемому модулю Capitalizer (в реальном проекте можно было бы обнаруживать пути подключаемых модулей, выполняя поиск файлов DLL в известных местоположениях или читая конфигурационный файл):

```
class Program
{
    const bool UseCollectibleContexts = true;
    static void Main()
    {
        const string capitalizer = @"C:\source\PluginDemo\
            + @"Capitalizer\bin\Debug\netcoreapp3.0\Capitalizer.dll";
        Console.WriteLine (TransformText ("big apple", capitalizer));
    }
}
```

```

static string TransformText (string text, string pluginPath)
{
    var alc = new PluginLoadContext (pluginPath, UseCollectibleContexts);
    try
    {
        Assembly assem = alc.LoadFromAssemblyPath (pluginPath);
        // Найти в сборке тип, который реализует ITextPlugin:
        Type pluginType = assem.ExportedTypes.Single (t =>
            typeof (ITextPlugin).IsAssignableFrom (t));
        // Создать экземпляр реализации ITextPlugin:
        var plugin = (ITextPlugin)Activator.CreateInstance (pluginType);
        // Вызвать метод TransformText:
        return plugin.TransformText (text);
    }
    finally
    {
        if (UseCollectibleContexts) alc.Unload(); // Выгрузить контекст ALC
    }
}

```

Давайте рассмотрим метод `TransformText`. Сначала мы создаем новый контекст ALC для нашего подключаемого модуля и затем запрашиваем у него загрузку главной сборки подключаемых модулей. Далее мы используем рефлексию для нахождения типа, который реализует интерфейс `ITextPlugin` (рефлексия подробно обсуждается в главе 18). Наконец, мы создаем экземпляр подключаемого модуля, вызываем метод `TransformText` и выгружаем контекст ALC.



Если необходимо многократно вызывать метод `TransformText`, тогда лучше кешировать контекст ALC, чем выгружать его после каждого вызова.

Ниже показан вывод:

BIG APPLE

Добавление зависимостей

Наш код полностью способен распознавать и изолировать зависимости. В целях иллюстрации давайте добавим ссылку NuGet на пакет `Humanizer.Core` версии 2.6.2. Вы можете сделать это с помощью пользовательского интерфейса Visual Studio или добавления в файл `Capitalizer.csproj` следующего элемента:

```

<ItemGroup>
    <PackageReference Include="Humanizer.Core" Version="2.6.2" />
</ItemGroup>

```

А теперь модифицируем `CapitalizerPlugin`:

```

using Humanizer;
namespace Capitalizer
{

```

```
public class CapitalizerPlugin : Plugin.Common.ITextPlugin
{
    public string TransformText (string input) => input.Pascalize();
}
}
```

В результате повторного запуска программы получается такой вывод:

```
BigApple
```

Далее мы реализуем еще один подключаемый модуль по имени Pluralizer. Создадим новый проект библиотеки .NET и добавим ссылку NuGet на пакет Humanizer.Core версии 2.7.9:

```
<ItemGroup>
    <PackageReference Include="Humanizer.Core" Version="2.7.9" />
</ItemGroup>
```

Добавим класс по имени PluralizerPlugin. Он аналогичен классу CapitalizerPlugIn, но только в нем вызывается метод Pluralize:

```
using Humanizer;
namespace Pluralizer
{
    public class PluralizerPlugin : Plugin.Common.ITextPlugin
    {
        public string TransformText (string input) => input.Pluralize();
    }
}
```

В заключение необходимо добавить в метод Main внутри Plugin.Host код для загрузки и запуска подключаемого модуля Pluralizer:

```
static void Main()
{
    const string capitalizer = @"C:\source\PluginDemo\
        + @"Capitalizer\bin\Debug\netcoreapp3.0\Capitalizer.dll";
    Console.WriteLine (TransformText ("big apple", capitalizer));
    const string pluralizer = @"C:\source\PluginDemo\
        + @"Pluralizer\bin\Debug\netcoreapp3.0\Pluralizer.dll";
    Console.WriteLine (TransformText ("big apple", pluralizer));
}
```

Вот как теперь выглядит вывод:

```
BigApple
big apples
```

Чтобы более ясно видеть, что происходит, изменим значение константы UseCollectibleContexts на false и добавим в метод Main приведенный ниже код для организации перечисления контекстов ALC и их сборок:

```
foreach (var context in AssemblyLoadContext.All)
{
    Console.WriteLine ($"Context: {context.GetType().Name} {context.Name}");
    foreach (var assembly in context.Assemblies)
        Console.WriteLine ($"Assembly: {assembly.FullName}");
}
```

В выводе легко заметить две разных версии Humanizer, каждая из которых загружена в собственный контекст ALC:

```
Context: PluginLoadContext Capitalizer.dll
Assembly: Capitalizer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=...
Assembly: Humanizer, Version=2.6.0.0, Culture=neutral, PublicKeyToken=...
Context: PluginLoadContext Pluralizer.dll
Assembly: Pluralizer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=...
Assembly: Humanizer, Version=2.7.0.0, Culture=neutral, PublicKeyToken=...
Context: DefaultAssemblyLoadContext Default
Assembly: System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, ...
Assembly: Host, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
...
...
```



Даже если бы оба подключаемых модуля использовали одну и ту же версию Humanizer, изоляция отдельных сборок все равно оказывается полезной, поскольку каждая сборка будет иметь собственные статические переменные.



Рефлексия и метаданные

Как было показано в главе 17, программа на языке C# компилируется в сборку, содержащую метаданные, скомпилированный код и ресурсы. Процесс инспектирования метаданных и скомпилированного кода во время выполнения называется *рефлексией*.

Скомпилированный код в сборке включает почти все содержимое первоначального исходного кода. Некоторая информация утрачивается, например, имена локальных переменных, комментарии и директивы препроцессора. Тем не менее, рефлексия позволяет получить доступ практически ко всему остальному, даже делая возможным написание декомпилятора.

Многие службы, доступные в .NET и открытые через язык C# (такие как динамическое связывание, сериализация и привязка данных), полагаются на присутствие метаданных. Ваши программы тоже могут использовать метаданные в своих интересах и даже расширять их новой информацией, применяя специальные атрибуты. В пространстве имен System.Reflection находится API-интерфейс рефлексии. Кроме того, с помощью классов из пространства имен System.Reflection.Emit во время выполнения можно динамически создавать новые метаданные и исполняемые инструкции на промежуточном языке (IL).

В примерах настоящей главы предполагается, что вы импортировали пространства имен System и System.Reflection, а также System.Reflection.Emit.



Когда мы используем в главе термин “динамическое”, то имеем в виду применение рефлексии с целью решения задачи, для которой безопасность типов обеспечивается только во время выполнения. По принципу это похоже на *динамическое связывание* посредством ключевого слова `dynamic` в C#, хотя механизм и функциональность здесь другие.

Динамическое связывание намного проще в использовании и задействует среду DLR для взаимодействия с динамическими языками. Рефлексия относительно неудобна в применении, но обладает большей гибкостью в плане того, что можно делать с помощью среды CLR. Например, рефлексия позволяет получать списки типов и членов, создавать объекты, имена которых указываются в виде строк, и строить сборки на лету.

Рефлексия и активизация типов

В этом разделе мы рассмотрим, как получать экземпляр класса Type, инспектировать его метаданные и использовать для динамического создания объекта.

Получение экземпляра Type

Экземпляр класса System.Type представляет метаданные для типа. Поскольку класс Type применяется очень широко, он находится в пространстве имен System, а не в System.Reflection.

Получить экземпляр System.Type можно путем вызова метода GetType на любом объекте или с помощью операции typeof языка C#:

```
Type t1 = DateTime.Now.GetType();           // Экземпляр Type, полученный
                                                // во время выполнения
Type t2 = typeof (DateTime);                // Экземпляр Type, полученный
                                                // на этапе компиляции
```

Операцию typeof можно использовать для получения типов массивов и обобщенных типов:

```
Type t3 = typeof (DateTime[]);             // Тип одномерного массива
Type t4 = typeof (DateTime[,]);            // Тип двухмерного массива
Type t5 = typeof (Dictionary<int,int>); // Закрытый обобщенный тип
Type t6 = typeof (Dictionary<,>);        // Несвязанный обобщенный тип
```

Экземпляр Type можно также извлекать по имени. При наличии ссылки на его сборку (Assembly) необходимо вызвать метод Assembly.GetType (как будет описано более подробно в разделе “Рефлексия сборок” далее в главе):

```
Type t = Assembly.GetExecutingAssembly().GetType ("Demos.TestProgram");
```

Если объект Assembly отсутствует, то тип можно получить через его имя с указанием сборки (полное имя типа, за которым следует полностью или частично заданное имя сборки). Сборка неявно загружается, как если бы вызывался метод Assembly.Load(string):

```
Type t = Type.GetType ("System.Int32, System.Private.CoreLib");
```

Имя объект System.Type, его свойства можно применять для доступа к имени типа, сборке, базовому типу, уровню видимости и т.д.:

```
Type stringType = typeof (string);
string name      = stringType.Name;          // String
Type baseType   = stringType.BaseType;        // typeof(Object)
Assembly assem  = stringType.Assembly;        // System.Private.CoreLib
bool isPublic   = stringType.IsPublic;         // true
```

Экземпляр `System.Type` — своего рода окно в мир метаданных для этого типа, а также для сборки, в которой он определен.



Класс `System.Type` является абстрактным, так что операция `typeof` должна в действительности давать подкласс класса `Type`. Среда CLR использует внутренний подкласс .NET по имени `RuntimeType`.

Класс `TypeInfo`

Если вы планируете нацеливаться на .NET Core 1.x (или более старый профиль Windows Store), то обнаружите, что большинство членов `Type` отсутствует. Взамен доступ к отсутствующим членам открывается через класс по имени `TypeInfo`, экземпляр которого получается вызовом `GetTypeInfo`. Таким образом, чтобы заставить код предыдущего примера выполняться, вот как нужно поступить:

```
Type stringType = typeof(string);
string name = stringType.Name;
Type baseType = stringType.GetTypeInfo().BaseType;
Assembly assem = stringType.GetTypeInfo().Assembly;
bool isPublic = stringType.GetTypeInfo().IsPublic;
```

Класс `TypeInfo` существует в .NET Core 2 и 3, а также в .NET 5+ (и в .NET Framework 4.5+ и всех версиях .NET Standard), поэтому предыдущий код работает почти везде. Класс `TypeInfo` также включает дополнительные свойства и методы для выполнения рефлексии членов.

Получение типов массивов

Как только что было указано, операция `typeof` и метод `GetType` имеют дело с типами массивов. Получить тип массива можно также за счет вызова метода `MakeArrayType` на типе элементов массива:

```
Type simpleArrayType = typeof(int).MakeArrayType();
Console.WriteLine(simpleArrayType == typeof(int[])); // True
```

Передавая методу `MakeArrayType` целочисленный аргумент, можно создавать многомерные массивы:

```
Type cubeType = typeof(int).MakeArrayType(3); // В форме куба
Console.WriteLine(cubeType == typeof(int[, ,])); // True
```

Метод `GetElementType` делает обратное, т.е. извлекает тип элементов массива:

```
Type e = typeof(int[]).GetElementType(); // e == typeof(int)
```

Метод `GetArrayRank` возвращает количество измерений в многомерном массиве:

```
int rank = typeof(int[, ,]).GetArrayRank(); // 3
```

Получение вложенных типов

Чтобы извлечь вложенные типы, нужно вызвать метод `GetNestedTypes` на содержащем их типе:

```
foreach (Type t in typeof (System.Environment).GetNestedTypes())
    Console.WriteLine (t.FullName);
```

Вот вывод:

```
System.Environment+SpecialFolder
```

Или можно поступить так:

```
foreach (TypeInfo t in typeof (System.Environment).GetTypeInfo()
                    .DeclaredNestedTypes)
    Debug.WriteLine (t.FullName);
```

С вложенными типами связано одно предостережение: среда CLR трактует вложенный тип как имеющий специальные "вложенные" уровни доступности:

```
Type t = typeof (System.Environment.SpecialFolder);
Console.WriteLine (t.IsPublic);           // False
Console.WriteLine (t.IsNestedPublic);     // True
```

Имена типов

Тип имеет свойства `Namespace`, `Name` и `FullName`. В большинстве случаев `FullName` является объединением первых двух свойств:

```
Type t = typeof (System.Text.StringBuilder);
Console.WriteLine (t.Namespace);          // System.Text
Console.WriteLine (t.Name);              // StringBuilder
Console.WriteLine (t.FullName);          // System.Text.StringBuilder
```

Из указанного правила существуют два исключения: вложенные типы и закрытые обобщенные типы.



Класс `Type` также имеет свойство по имени `AssemblyQualified Name`, возвращающее значение свойства `FullName`, за которым следует запятая и полное имя сборки. Это та самая строка, которую можно передавать методу `Type.GetType`, и она уникальным образом идентифицирует тип внутри стандартного контекста загрузки.

Имена вложенных типов

В случае вложенных типов содержащий тип присутствует только в `FullName`:

```
Type t = typeof (System.Environment.SpecialFolder);
Console.WriteLine (t.Namespace);          // System
Console.WriteLine (t.Name);              // SpecialFolder
Console.WriteLine (t.FullName);          // System.Environment+SpecialFolder
```

Символ `+` отделяет содержащий тип от вложенного пространства имен.

Имена обобщенных типов

Имена обобщенных типов снабжаются суффиксами в виде символа '`,`', за которым следует количество параметров типа. Если обобщенный тип является несвязанным, то такое правило применяется и к `Name`, и к `FullName`:

```
Type t = typeof (Dictionary<,>); // Unbound (несвязанный)
Console.WriteLine (t.Name); // Dictionary'2
Console.WriteLine (t.FullName); // System.Collections.Generic.Dictionary'2
```

Однако если обобщенный тип является закрытым, то свойство FullName (единственное) приобретает важное дополнение: список всех параметров типа, для каждого из которых указывается полное имя, включающее сборку:

```
Console.WriteLine (typeof (Dictionary<int,string>).FullName);
```

Вывод выглядит так:

```
System.Collections.Generic.Dictionary`2[[System.Int32,
System.Private.CoreLib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=7cec85d7bea7798e], [System.String, System.Private.CoreLib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e]]
```

В итоге гарантируется, что свойство AssemblyQualifiedName (комбинация полного имени типа и имени сборки) содержит достаточный объем информации для исчерпывающей идентификации как обобщенного типа, так и его параметров типа.

Имена типов массивов и указателей

Массивы представляются с тем же суффиксом, который используется в выражении typeof:

```
Console.WriteLine (typeof ( int[] ).Name); // Int32[]
Console.WriteLine (typeof ( int[,] ).Name); // Int32[,]
Console.WriteLine (typeof ( int[,] ).FullName); // System.Int32[,]
```

Типы указателей похожи:

```
Console.WriteLine (typeof (byte*).Name); // Byte*
```

Имена типов параметров ref и out

Экземпляр Type, описывающий параметр ref или out, имеет суффикс &:

```
public void RefMethod (ref int p)
{
    Type t = MethodInfo.GetCurrentMethod().GetParameters()[0].ParameterType;
    Console.WriteLine (t.Name); // Int32&
}
```

Более подробно об этом речь пойдет в разделе “Рефлексия и вызов членов” далее в главе.

Базовые типы и интерфейсы

Класс Type открывает доступ к свойству BaseType:

```
Type base1 = typeof (System.String).BaseType;
Type base2 = typeof (System.IO.FileStream).BaseType;
Console.WriteLine (base1.Name); // Object
Console.WriteLine (base2.Name); // Stream
```

Метод GetInterfaces возвращает интерфейсы, которые тип реализует:

```
foreach (Type iType in typeof (Guid).GetInterfaces())
    Console.WriteLine (iType.Name);
```

Вот вывод:

```
IFormattable
IComparable
IComparable'1
IEquatable'1
```

(Метод GetInterfaceMap возвращает структуру, которая показывает, каким образом каждый член интерфейса реализован в классе или структуре — использование этого расширенного функционального средства иллюстрируется в разделе “Вызов статических виртуальных/абстрактных членов интерфейсов” далее в главе.)

Рефлексия предоставляет три динамических эквивалента статической операции `is` языка C#.

- `IsInstanceOfType`. Принимает тип и экземпляр.
- `IsAssignableFrom` и (начиная с .NET 5) `IsAssignableTo`. Принимают два типа.

Ниже приведен пример применения первого метода:

```
object obj      = Guid.NewGuid();
Type target    = typeof (IFormattable);
bool isTrue    = obj is IFormattable;           // Статическая операция C#
bool alsoTrue = target.IsInstanceOfType (obj); // Динамический эквивалент
```

Метод `IsAssignableFrom` более универсален:

```
Type target = typeof (IComparable), source = typeof (string);
Console.WriteLine (target.IsAssignableFrom (source)); // True
```

Метод `IsSubclassOf` работает по тому же самому принципу, что и `IsAssignableFrom`, но исключает интерфейсы.

Создание экземпляров типов

Динамически создать объект из его типа можно двумя способами:

- вызвать статический метод `Activator.CreateInstance`;
- вызвать метод `Invoke` на объекте `ConstructorInfo`, который получен в результате вызова метода `GetConstructor` на экземпляре `Type` (расширенные сценарии).

Метод `Activator.CreateInstance` принимает экземпляр `Type` и дополнительные аргументы, передаваемые конструктору:

```
int i = (int) Activator.CreateInstance (typeof (int));
DateTime dt = (DateTime) Activator.CreateInstance (typeof (DateTime),
2000, 1, 1);
```

Метод `CreateInstance` позволяет указывать многие другие данные, такие как сборка, из которой загружается тип, и необходимость привязки к неоткрытым конструкторам. Если исполняющей среде не удается найти подходящий конструктор, то генерируется исключение `MissingMethodException`.

Вызов метода `Invoke` класса `ConstructorInfo` нужен, когда значения аргументов не позволяют устраниить неоднозначность между перегруженными конструкторами. Например, пусть класс `X` имеет два конструктора: один принимает параметр типа `string`, а другой — параметр типа `StringBuilder`. В случае передачи аргумента `null` методу `Activator.CreateInstance` выбор целевого конструктора будет неоднозначным. В такой ситуации взамен должен использоваться класс `ConstructorInfo`:

```
// Извлечь конструктор, который принимает единственный параметр типа string:  
ConstructorInfo ci = typeof(X).GetConstructor(new[] { typeof(string) });  
  
// Сконструировать объект с применением перегруженной версии,  
// передавая значение null:  
object foo = ci.Invoke(new object[] { null });
```

Или при нацеливании на .NET Core 1, более старый профиль Windows Store:

```
ConstructorInfo ci = typeof(X).GetTypeInfo().DeclaredConstructors  
.FirstOrDefault(c =>  
    c.GetParameters().Length == 1 &&  
    c.GetParameters()[0].ParameterType == typeof(string));
```

Чтобы получить неоткрытый конструктор, потребуется указать соответствующее значение перечисления `BindingFlags` — данный вопрос обсуждается в разделе “Доступ к неоткрытым членам” далее в главе.



Динамическое создание экземпляров добавляет несколько микросекунд ко времени, которое занимает конструирование объекта. В относительном выражении это довольно много, потому что CLR обычно создает объекты очень быстро (выполнение простой операции `new` на небольшом классе требует нескольких десятков наносекунд).

Чтобы динамически создать объект массива на основе только типа его элементов, сначала понадобится вызвать метод `MakeArrayType`. Можно также создавать экземпляры обобщенных типов, как будет показано в следующем разделе.

Для динамического создания объекта делегата необходимо вызвать метод `Delegate.CreateDelegate`. Ниже приведен пример, демонстрирующий создание делегата экземпляра и статического делегата:

```
class Program  
{  
    delegate int IntFunc (int x);  
  
    static int Square (int x) => x * x;           // Статический метод  
    int         Cube   (int x) => x * x * x;       // Метод экземпляра  
  
    static void Main()  
    {  
        Delegate staticD = Delegate.CreateDelegate  
            (typeof(IntFunc), typeof(Program), "Square");
```

```

        Delegate instanceD = Delegate.CreateDelegate
            (typeof (IntFunc), new Program(), "Cube");
        Console.WriteLine (staticD.DynamicInvoke (3));           // 9
        Console.WriteLine (instanceD.DynamicInvoke (3));         // 27
    }
}

```

Запустить возвращенный объект `Delegate` можно за счет вызова метода `DynamicInvoke`, как делалось в показанном примере, либо путем приведения к типизированному делегату:

```

IntFunc f = (IntFunc) staticD;
Console.WriteLine (f(3)); // 9 (но выполняется намного быстрее!)

```

Вместо имени метода в `CreateDelegate` можно передать объект `MethodInfo`. В разделе “Рефлексия и вызов членов” далее в главе мы опишем класс `MethodInfo` вместе с обоснованием приведения динамически созданного делегата обратно к типу статического делегата.

Обобщенные типы

Класс `Type` способен представлять закрытый или несвязанный обобщенный тип. Как и на этапе компиляции, экземпляр закрытого обобщенного типа может быть создан, а экземпляр несвязанного обобщенного типа — нет:

```

Type closed = typeof (List<int>);
List<int> list = (List<int>) Activator.CreateInstance (closed); // Допускается

Type unbound = typeof (List<>);
object anError = Activator.CreateInstance (unbound);           // Ошибка времени
                                                               // выполнения

```

Метод `MakeGenericType` преобразует несвязанный обобщенный тип в закрытый. Необходимо просто передать желаемые аргументы типа:

```

Type unbound = typeof (List<>);
Type closed = unbound.MakeGenericType (typeof (int));

```

Метод `GetGenericTypeDefinition` делает противоположное:

```

Type unbound2 = closed.GetGenericTypeDefinition(); // unbound == unbound2

```

Свойство `IsGenericType` возвращает `true`, если экземпляр `Type` является обобщенным, а свойство `IsGenericTypeDefinition` возвращает `true`, если обобщенный тип несвязанный. Приведенный далее код проверяет, является ли указанный тип типом значения, допускающим `null`:

```

Type nullable = typeof (bool?);
Console.WriteLine (
    nullable.IsGenericType &&
    nullable.GetGenericTypeDefinition() == typeof (Nullable<>)); // True

```

Метод `GetGenericArguments` возвращает аргументы типа для закрытых обобщенных типов:

```

Console.WriteLine (closed.GetGenericArguments () [0]); // System.Int32
Console.WriteLine (nullable.GetGenericArguments () [0]); // System.Boolean

```

Для несвязанных обобщенных типов метод `GetGenericArguments` возвращает псевдотипы, которые представляют типы-заполнители, указанные в определениях обобщенных типов:

```
Console.WriteLine (unbound.GetGenericArguments () [0]); // T
```



Во время выполнения все обобщенные типы будут либо **несвязанными**, либо **закрытыми**. Они оказываются несвязанными в (относительно редком) случае такого выражения, как `typeof (Foo<>)`; иначе они закрыты. Во время выполнения нет понятия *открытого* обобщенного типа: все открытые типы закрываются компилятором. Метод `Test` в следующем классе всегда выводит `False`:

```
class Foo<T>
{
    public void Test()
        => Console.Write (GetType ().IsGenericTypeDefinition);
}
```

Рефлексия и вызов членов

Метод `GetMembers` возвращает члены типа. Взгляните на показанный ниже класс:

```
class Walnut
{
    private bool cracked;
    public void Crack() { cracked = true; }
}
```

Выполнить рефлексию его открытых членов можно следующим образом:

```
MethodInfo[] members = typeof (Walnut).GetMembers ();
foreach (MethodInfo m in members)
    Console.WriteLine (m);
```

Вот результат:

```
Void Crack()
System.Type GetType()
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
Void.ctor()
```

Выполнение рефлексии членов с помощью `TypeInfo`

Класс `TypeInfo` открывает доступ к другому (и в чем-то более простому) протоколу для проведения рефлексии членов. Использовать данный API-интерфейс не обязательно (за исключением приложений .NET Core 1 и более старых приложений Windows Store, т.к. точный эквивалент метода `GetMembers` в них отсутствует).

Вместо открытия доступа к методам, подобным `GetMembers`, который возвращает массивы, класс `TypeInfo` предлагает *свойства*, возвращающие объекты `IEnumerable<T>`, на которых обычно запускаются запросы LINQ. Самым широко применяемым свойством является `DeclaredMembers`:

```
IEnumerable<MemberInfo> members =
    typeof(Walnut).GetTypeInfo().DeclaredMembers;
```

В отличие от метода `GetMembers` из результата исключены унаследованные члены:

```
Void Crack()
Void .ctor()
Boolean cracked
```

Предусмотрены также свойства для возвращения специфических разновидностей членов (`DeclaredProperties`, `DeclaredMethods`, `DeclaredEvents` и т.д.) и методы для возвращения конкретных членов по именам (например, `GetMethodByName`). Последние не могут применяться для перегруженных методов (поскольку нет никакого способа указать типы параметров). Взамен в отношении свойства `DeclaredMethods` запускается запрос LINQ:

```
MethodInfo method = typeof(int).GetTypeInfo().DeclaredMethods
    .FirstOrDefault(m => m.Name == "ToString" &&
        m.GetParameters().Length == 0);
```

В случае вызова без аргументов метод `GetMembers` возвращает все открытые члены для типа (и его базовых типов). Метод `GetMember` извлекает отдельный член по имени, хотя и возвращает массив, т.к. члены могут быть перегруженными:

```
MemberInfo[] m = typeof(Walnut).GetMember("Crack");
Console.WriteLine(m[0]); // Void Crack()
```

В классе `MemberInfo` также имеется свойство по имени `MemberType` типа `MemberTypes`, который представляет собой перечисление флагов со следующими значениями:

All	Custom	Field	NestedType	TypeInfo
Constructor	Event	Method	Property	

Вызываемому методу `GetMembers` можно передать экземпляр `MemberTypes`, чтобы ограничить виды возвращаемых членов. В качестве альтернативы допускается ограничивать результатирующий набор, вызывая методы `GetMethods`, `GetFields`, `GetProperties`, `GetEvents`, `GetConstructors` и `GetNestedTypes`. Для каждого из перечисленных методов доступны также версии с именами в единственном числе, позволяющие получать конкретный член.



При извлечении члена типа полезно придерживаться максимально возможной конкретизации, чтобы работа кода не нарушалась, если позже будут добавлены дополнительные члены. Если осуществляется извлечение метода по имени, тогда указание типов всех параметров гарантирует, что код сохранит работоспособность и после перегрузки метода в будущем (примеры будут приведены в разделе “Параметры методов” далее в главе).

Объект `MethodInfo` имеет свойство `Name` и два свойства, возвращающие экземпляр `Type`.

- `DeclaringType`. Возвращает экземпляр `Type`, который определяет член.
- `ReflectedType`. Возвращает экземпляр `Type`, на котором был вызван метод `GetMembers`.

Эти два свойства отличаются при вызове на члене, который определен в базовом типе: `DeclaringType` возвращает базовый тип, тогда как `ReflectedType` — подтип, что отражено в следующем примере:

```
// MethodInfo — это подкласс MethodInfo; см. рис. 18.1.  
MethodInfo test = typeof (Program).GetMethod ("ToString");  
MethodInfo obj = typeof (object) .GetMethod ("ToString");  
  
Console.WriteLine (test.DeclaringType);           // System.Object  
Console.WriteLine (obj.DeclaringType);           // System.Object  
Console.WriteLine (test.ReflectedType);          // Program  
Console.WriteLine (obj.ReflectedType);          // System.Object  
Console.WriteLine (test == obj);                 // False
```

Так как объекты `test` и `obj` имеют разные значения в свойстве `ReflectedType`, они не равны. Тем не менее, отличие между ними — чистая “выдумка” API-интерфейса рефлексии; тип `Program` не имеет отдельного метода `ToString` во внутренней системе типов. Мы можем удостовериться в том, что эти два объекта `MethodInfo` ссылаются на тот же самый метод, одним из двух способов:

```
Console.WriteLine (test.MethodHandle == obj.MethodHandle);    // True  
Console.WriteLine (test.MetadataToken == obj.MetadataToken  
    && test.Module == obj.Module);                                // True
```

Свойство `MethodHandle` уникально для каждого (по-настоящему отличающегося) метода внутри процесса, а свойство `MetadataToken` уникально среди всех типов и членов в рамках модуля сборки.

В классе `MethodInfo` также определены методы для возвращения специальных атрибутов (они рассматриваются в разделе “Извлечение атрибутов во время выполнения” далее в главе).



Получить объект `MethodBase` текущего выполняющегося метода можно путем вызова статического метода `MethodBase.GetCurrentMethod`.

Типы членов

Сам класс `MethodInfo` в плане членов легковесен, потому что он является абстрактным базовым классом для типов, показанных на рис. 18.1.

Экземпляр класса `MethodInfo` можно приводить к его подтипу на основе свойства `MemberType`. Если член получен через методы `GetMethod`, `GetField`, `GetProperty`, `GetEvent`, `GetConstructor` или `GetNestedType` (либо с помощью их версий с именами во множественном числе), тогда приведение не будет обязательным.

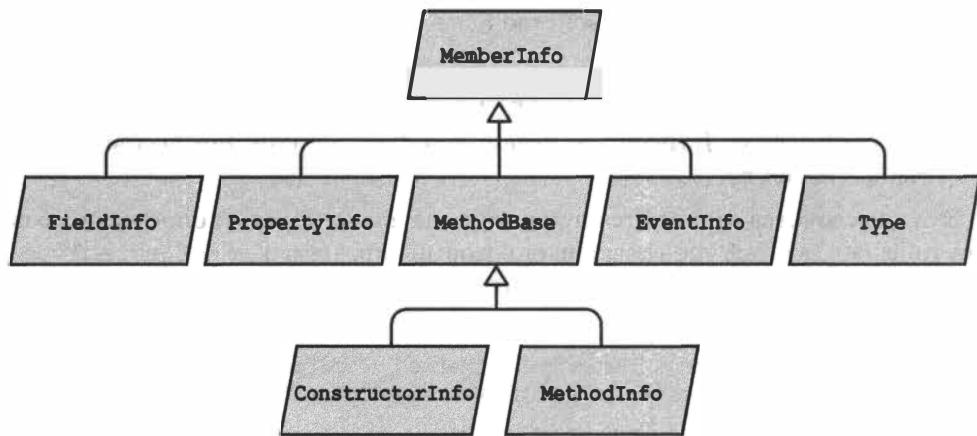


Рис. 18.1. Типы членов

В табл. 18.1 показано, какие методы должны использоваться для всех видов конструкций языка C#.

Таблица 18.1. Извлечение метаданных членов

Конструкция C#	Используемый метод	Используемое имя	Результат
Метод	GetMethod	(имя метода)	MethodInfo
Свойство	GetProperty	(имя свойства)	PropertyInfo
Индексатор	GetDefaultMembers		MemberInfo[] (массив, содержащий объекты PropertyInfo, если скомпилирован в C#)
Поле	GetField	(имя поля)	FieldInfo
Член перечисления	GetField	(имя члена)	FieldInfo
Событие	GetEvent	(имя события)	EventInfo
Конструктор	GetConstructor		ConstructorInfo
Финализатор	GetMethod	"Finalize"	MethodInfo
Операция	GetMethod	"op_" + имя операции	MethodInfo
Вложенный тип	GetNestedType	(имя типа)	Type

Каждый подкласс MemberInfo имеет множество свойств и методов, которые отражают все аспекты метаданных члена, в том числе видимость, модификаторы, аргументы обобщенных типов, параметры, возвращаемый тип и специальные атрибуты. Ниже демонстрируется применение метода GetMethod:

```

MethodInfo m = typeof (Walnut).GetMethod ("Crack");
Console.WriteLine (m); // Void Crack()
Console.WriteLine (m.ReturnType); // System.Void
  
```

Все экземпляры `*Info` кешируются API-интерфейсом рефлексии при первом использовании:

```
MethodInfo method = typeof (Walnut).GetMethod ("Crack");
MemberInfo member = typeof (Walnut).GetMember ("Crack") [0];
Console.WriteLine (method == member); // True
```

Кроме предохранения идентичности объектов кеширование улучшает показатели производительности в противном случае довольно медленно работающего API-интерфейса рефлексии.

Сравнение членов C# и членов CLR

В табл. 18.1 видно, что некоторые функциональные конструкции C# не имеют однозначного соответствия с конструкциями CLR. Причина в том, что среди CLR и API-интерфейс рефлексии были спроектированы с учетом всех языков .NET; рефлексию можно использовать даже из кода Visual Basic.

Некоторые конструкции языка C# — в частности, индексаторы, перечисления, операции и финализаторы — обрабатываются средой CLR особым образом.

- Индексатор C# транслируется в свойство, принимающее один или более аргументов, которое помечено как `[DefaultMember]` на уровне типа.
- Перечисление C# транслируется в подтип `System.Enum` со статическим полем для каждого члена.
- Операция C# транслируется в статический метод со специальным именем, начинающимся с `op_`; примером может служить `op>Addition`.
- Финализатор C# транслируется в метод, который переопределяет `Finalize`.

Еще одна сложность связана с тем, что свойства и события на самом деле заключают в себе два компонента:

- метаданные, описывающие свойство или событие (инкапсулированные посредством `PropertyInfo` или `EventInfo`);
- один или два поддерживающих метода.

В программе C# поддерживающие методы инкапсулированы внутри определения свойства или события. Но после компиляции в IL поддерживающие методы представляются как обычные методы, которые можно вызывать подобно любым другим. Другими словами, `GetMethod` наряду с обычными методами возвращает поддерживающие методы свойств и событий:

```
class Test { public int X { get { return 0; } set {} } }
void Demo()
{
    foreach (MethodInfo mi in typeof (Test).GetMethods())
        Console.WriteLine (mi.Name + " " );
}
```

Вот вывод:

```
get_X  set_X  GetType  ToString  Equals  GetHashCode
```

Идентифицировать эти методы можно через свойство `IsSpecialName` в классе `MethodInfo`. Свойство `IsSpecialName` возвращает `true` для методов доступа к свойствам, индексаторам и событиям, а также для операций. Оно возвращает `false` только для обычных методов C# и для метода `Finalize`, если определен финализатор.

Ниже представлены поддерживающие методы, генерируемые C#.

Конструкция C#	Тип члена	Методы в IL
Свойство	Property	<code>get_XXX</code> и <code>set_XXX</code>
Индексатор	Property	<code>get_Item</code> и <code>set_Item</code>
Событие	Event	<code>add_XXX</code> и <code>remove_XXX</code>

Каждый поддерживающий метод имеет собственный ассоциированный с ним объект `MethodInfo`. Получить к нему доступ можно следующим образом:

```
 PropertyInfo pi = typeof (Console).GetProperty ("Title");
 MethodInfo getter = pi.GetGetMethod(); // get_Title
 MethodInfo setter = pi.GetSetMethod(); // set_Title
 MethodInfo[] both = pi.GetAccessors(); // Length==2
```

Методы `GetAddMethod` и `GetRemoveMethod` делают аналогичную работу для класса `EventInfo`.

Чтобы двигаться в обратном направлении — из `MethodInfo` в связанный объект `PropertyInfo` или `EventInfo` — необходимо выполнять запрос. Для такой цели идеально подходит LINQ:

```
 PropertyInfo p = mi.DeclaringType.GetProperties()
    .First (x => x.GetAccessors (true).Contains (mi));
```

Свойства, допускающие только инициализацию

Свойства, допускающие только инициализацию, которые появились в версии C# 9, могут устанавливаться через инициализатор объекта, но впоследствии трактуются компилятором как допускающие только чтение. С точки зрения среды CLR средство доступа `init` похоже на обычное средство доступа `set`, но со специальным флагом, применяемым к возвращаемому типу метода `set` (который что-то означает для компилятора).

Любопытно, что этот флаг не кодируется как обычный атрибут. Взамен он применяет довольно непонятный механизм, называемый *обязательным модификатором* (`modreq`), который гарантирует, что предшествующие версии компилятора C# (не распознающие `modreq`) будут игнорировать средство доступа, а не интерпретировать свойство как записываемое.

Идентификатор `modreq` для свойств, допускающих только инициализацию, называется `IsExternalInit`, и запросить его можно следующим образом:

```
bool IsInitOnly ( PropertyInfo pi ) => pi
    .GetSetMethod ().ReturnParameter.GetRequiredCustomModifiers ()
    .Any (t => t.Name == "IsExternalInit");
```

NullabilityInfoContext

Начиная с версии .NET 6, класс NullabilityInfoContext позволяет получать информацию о возможности принятия значений null полем, свойством, событием или параметром:

```
void PrintPropertyNullability ( PropertyInfo pi )
{
    var info = new NullabilityInfoContext () .Create ( pi );
    Console.WriteLine ( pi.Name + " read " + info.ReadState );
    Console.WriteLine ( pi.Name + " write " + info.WriteState );
    // Использовать info.Element для получения информации о возможности
    // принятия значений null элементами массива
}
```

Члены обобщенных типов

Метаданные членов можно получать как для несвязанных, так и для закрытых обобщенных типов:

```
 PropertyInfo unbound = typeof ( IEnumarator < > ) .GetProperty ( "Current" );
 PropertyInfo closed = typeof ( IEnumarator < int > ) .GetProperty ( "Current" );
 Console.WriteLine ( unbound );                                     // T Current
 Console.WriteLine ( closed );                                    // Int32 Current
 Console.WriteLine ( unbound.PropertyType .IsGenericParameter ); // True
 Console.WriteLine ( closed.PropertyType .IsGenericParameter ); // False
```

Объекты MemberInfo, возвращаемые из несвязанных и закрытых обобщенных типов, всегда отличаются — даже для членов, сигнатуры которых не содержат параметров обобщенных типов:

```
 PropertyInfo unbound = typeof ( List < > ) .GetProperty ( "Count" );
 PropertyInfo closed = typeof ( List < int > ) .GetProperty ( "Count" );
 Console.WriteLine ( unbound );                                // Int32 Count
 Console.WriteLine ( closed );                               // Int32 Count
 Console.WriteLine ( unbound == closed ); // False
 Console.WriteLine ( unbound.DeclaringType .IsGenericTypeDefinition ); // True
 Console.WriteLine ( closed.DeclaringType .IsGenericTypeDefinition ); // False
```

Члены несвязанных обобщенных типов не могут вызываться динамически.

Динамический вызов члена



Динамический вызов члена можно выполнить проще с помощью библиотеки с открытым кодом под названием Uncapsulator (<https://github.com/albahari/uncapsulator>), которая доступна на NuGet и GitHub. Она предлагает текущий API-интерфейс для вызова открытых и неоткрытых членов посредством рефлексии с применением специальной динамической привязки.

Получив объект MethodInfo, PropertyInfo или FieldInfo, к нему можно динамически обращаться либо извлекать/устанавливать его значение. Это на-

зывается *поздним связыванием*, т.к. выбор вызываемого члена производится во время выполнения, а не на этапе компиляции.

Например, в следующем коде применяется обычное *статическое связывание*:

```
string s = "Hello";
int length = s.Length;
```

А вот так же самое можно сделать динамически с помощью позднего связывания:

```
object s = "Hello";
 PropertyInfo prop = s.GetType().GetProperty ("Length");
 int length = (int) prop.GetValue (s, null); // 5
```

Методы `GetValue` и `SetValue` извлекают и устанавливают значение объекта `PropertyInfo` или `FieldInfo`. Первый аргумент — экземпляр, который может быть равен `null` для статического члена. Доступ к индексатору подобен доступу к свойству по имени `Item` за исключением того, что при вызове метода `GetValue` или `SetValue` во втором аргументе указываются значения индексатора.

Чтобы вызвать метод динамически, необходимо обратиться к методу `Invoke` на объекте `MethodInfo`, предоставив массив аргументов, которые должны передаваться вызываемому методу. Если окажется, что хотя бы один из аргументов имеет неподходящий тип, тогда во время выполнения генерируется исключение. При динамическом вызове утрачивается безопасность типов этапа компиляции, но по-прежнему поддерживается безопасность типов времени выполнения (такая же, как в случае использования ключевого слова `dynamic`).

Параметры методов

Предположим, что необходимо динамически вызвать метод `Substring` типа `string`. Статически пришлось бы поступить следующим образом:

```
Console.WriteLine ("stamp".Substring(2)); // "amp"
```

Ниже показан динамический эквивалент, в котором применяются рефлексия и позднее связывание:

```
Type type = typeof (string);
Type[] parameterTypes = { typeof (int) };
MethodInfo method = type.GetMethod ("Substring", parameterTypes);
object[] arguments = { 2 };
object returnValue = method.Invoke ("stamp", arguments);
Console.WriteLine (returnValue); // "amp"
```

Поскольку метод `Substring` перегружен, мы должны передать методу `GetMethod` массив типов параметров, чтобы указать желаемую версию. Без типов параметров метод `GetMethod` генерирует исключение `AmbiguousMatchException`.

Метод `GetParameters`, определенный в `MethodBase` (базовый класс для `MethodInfo` и `ConstructorInfo`), возвращает метаданные параметров. Предыдущий пример можно продолжить:

```

ParameterInfo[] paramList = method.GetParameters();
foreach (ParameterInfo x in paramList)
{
    Console.WriteLine (x.Name);                      // startIndex
    Console.WriteLine (x.ParameterType);             // System.Int32
}

```

Работа с параметрами `ref` и `out`

Чтобы передать параметры `ref` или `out`, перед получением объекта метода необходимо вызвать `MakeByRefType` на типе. Например, представленный далее код:

```

int x;
bool successfulParse = int.TryParse ("23", out x);

```

можно выполнить динамически следующим образом:

```

object[] args = { "23", 0 };
Type[] argTypes = { typeof (string), typeof (int).MakeByRefType () };
MethodInfo tryParse = typeof (int).GetMethod ("TryParse", argTypes);
bool successfulParse = (bool) tryParse.Invoke (null, args);
Console.WriteLine (successfulParse + " " + args[1]);           // True 23

```

Тот же самый подход работает для типов параметров `ref` и `out`.

Извлечение и вызов обобщенных методов

Явное указание типов параметров при вызове метода `GetMethod` может оказаться жизненно важным в разрешении неоднозначности перегруженных методов. Тем не менее, указывать типы обобщенных параметров невозможно. Например, рассмотрим класс `System.Linq.Enumerable`, в котором метод `Where` перегружен:

```

public static IEnumerable<TSource> Where<TSource>
    (this IEnumerable<TSource> source, Func<TSource, bool> predicate);
public static IEnumerable<TSource> Where<TSource>
    (this IEnumerable<TSource> source, Func<TSource, int, bool> predicate);

```

Чтобы получить конкретную перегруженную версию, потребуется извлечь все методы и затем вручную найти желаемую версию. Приведенный далее запрос извлекает первую перегруженную версию метода `Where`:

```

from m in typeof (Enumerable).GetMethods()
where m.Name == "Where" && m.IsGenericMethod
let parameters = m.GetParameters()
where parameters.Length == 2
let genArg = m.GetGenericArguments ().First ()
let enumerableOfT = typeof (IEnumerable<>).MakeGenericType (genArg)
let funcOfTBool = typeof (Func<,>).MakeGenericType (genArg, typeof (bool))
where parameters[0].ParameterType == enumerableOfT
    && parameters[1].ParameterType == funcOfTBool
select m

```

Вызов `.Single()` здесь дает корректный объект `MethodInfo` с параметрами несвязанного типа. Следующий шаг предусматривает закрытие параметров типа посредством вызова метода `MakeGenericMethod`:

```
var closedMethod = unboundMethod.MakeGenericMethod (typeof (int));
```

В данном случае мы закрываем `TSource` с использованием `int`, что позволяет вызывать метод `Enumerable.Where` с `source` типа `IEnumerable<int>` и `predicate` типа `Func<int, bool>`:

```
int[] source = { 3, 4, 5, 6, 7, 8 };
Func<int, bool> predicate = n => n % 2 == 1; // Только нечетные числа
```

Теперь закрытый обобщенный метод можно вызывать:

```
var query = (IEnumerable<int>) closedMethod.Invoke
    (null, new object[] { source, predicate });
foreach (int element in query) Console.Write (element + "|"); // 3|5|7|
```



В случае применения API-интерфейса `System.Linq.Expressions` для динамического построения выражений (см. главу 8) беспокоиться по поводу указания обобщенного метода не придется. Метод `Expression.Call` перегружен, чтобы позволить указывать аргументы закрытого типа метода, который требуется вызвать:

```
int[] source = { 3, 4, 5, 6, 7, 8 };
Func<int, bool> predicate = n => n % 2 == 1;
var sourceExpr = Expression.Constant (source);
var predicateExpr = Expression.Constant (predicate);
var callExpression = Expression.Call (
    typeof (Enumerable), "Where",
    new[] { typeof (int) }, // Закрытый обобщенный тип аргумента.
    sourceExpr, predicateExpr);
```

Использование делегатов для повышения производительности

Динамические вызовы относительно неэффективны и характеризуются накладными расходами, которые обычно укладываются в диапазон из нескольких микросекунд. Если метод вызывается многократно в цикле, то накладные расходы, приходящиеся на вызов, можно сместить в наносекундный диапазон, обращаясь вместо метода к динамически созданному экземпляру делегата, который нацелен на необходимый динамический метод. В следующем примере мы динамически вызываем метод `Trim` типа `string` миллион раз без значительных накладных расходов:

```
MethodInfo trimMethod = typeof (string).GetMethod ("Trim", new Type[0]);
var trim = (StringToString) Delegate.CreateDelegate
    (typeof (StringToString), trimMethod);
for (int i = 0; i < 1000000; i++)
    trim ("test");
delegate string StringToString (string s);
```

Такой код работает быстрее, потому что затратное позднее связывание (код, выделенный полужирным) происходит только один раз.

Доступ к неоткрытым членам

Все методы типов, применяемых для зондирования метаданных (например, `GetProperty`, `GetField` и т.д.), имеют перегруженные версии, которые принимают перечисление `BindingFlags`. Это перечисление служит фильтром метаданных и позволяет изменять стандартный критерий поиска. Наиболее распространенное использование связано с извлечением неоткрытых членов (работает только в настольных приложениях).

Например, пусть имеется следующий класс:

```
class Walnut
{
    private bool cracked;
    public void Crack() { cracked = true; }

    public override string ToString() { return cracked.ToString(); }
}
```

Вот как с ним можно поступить:

```
Type t = typeof (Walnut);
Walnut w = new Walnut();
w.Crack();

FieldInfo f = t.GetField ("cracked", BindingFlags.NonPublic |
                           BindingFlags.Instance);

f.SetValue (w, false);
Console.WriteLine (w); // False
```

Применение рефлексии для доступа к неоткрытым членам является мощным средством, однако оно также и небезопасно, поскольку позволяет обойти инкапсуляцию, создавая неуправляемую зависимость от внутренней реализации типа.

Перечисление `BindingFlags`

Перечисление `BindingFlags` предназначено для побитового комбинирования. Чтобы получить любое совпадение, необходимо начать с одной из следующих четырех комбинаций:

```
BindingFlags.Public | BindingFlags.Instance
BindingFlags.Public | BindingFlags.Static
BindingFlags.NonPublic | BindingFlags.Instance
BindingFlags.NonPublic | BindingFlags.Static
```

Флаг `NonPublic` охватывает квалификаторы доступа `internal`, `protected`, `protected internal` и `private`.

Приведенный ниже код извлекает все открытые статические члены типа `object`:

```
BindingFlags publicStatic = BindingFlags.Public | BindingFlags.Static;
MemberInfo[] members = typeof (object).GetMembers (publicStatic);
```

В показанном далее примере извлекаются все неоткрытые члены типа `object`, как статические, так и члены экземпляра:

```
BindingFlags nonPublicBinding =  
    BindingFlags.NonPublic | BindingFlags.Static | BindingFlags.Instance;  
MemberInfo[] members = typeof (object).GetMembers (nonPublicBinding);
```

Флаг `DeclaredOnly` исключает функции, унаследованные от базовых типов, если только они не были переопределены.



Флаг `DeclaredOnly` может несколько запутывать тем, что он *ограничивает* результатирующий набор (тогда как все остальные флаги *расширяют* результатирующий набор).

Обобщенные методы

Обобщенные методы не могут вызываться напрямую; следующий код приведет к генерации исключения:

```
class Program  
{  
    public static T Echo<T> (T x) { return x; }  
    static void Main()  
    {  
        MethodInfo echo = typeof (Program).GetMethod ("Echo");  
        Console.WriteLine (echo.IsGenericMethodDefinition); // True  
        echo.Invoke (null, new object[] { 123 } ); // Генерируется исключение  
    }  
}
```

Здесь потребуется дополнительный шаг, который предусматривает вызов метода `MakeGenericMethod` на объекте `MethodInfo` с указанием конкретных значений для аргументов обобщенных типов. В результате возвращается другой объект `MethodInfo`, к которому можно затем обращаться, как показано ниже:

```
MethodInfo echo = typeof (Program).GetMethod ("Echo");  
MethodInfo intEcho = echo.MakeGenericMethod (typeof (int));  
Console.WriteLine (intEcho.IsGenericMethodDefinition); // False  
Console.WriteLine (intEcho.Invoke (null, new object[] { 3 } )); // 3
```

Анонимный вызов членов обобщенного интерфейса

Рефлексия удобна, когда необходимо вызвать член обобщенного интерфейса, а параметры типа не известны вплоть до времени выполнения. Теоретически если типы спроектированы идеально, то потребность в подобном действии возникает редко; тем не менее, естественно, типы далеко не всегда проектируются идеальным образом.

Например, предположим, что нужно написать более мощную версию метода `ToString`, которая могла бы развертывать результат выполнения запросов LINQ. Мы могли бы начать так:

```
public static string ToStringEx <T> (IEnumerable<T> sequence)  
{  
    ...  
}
```

Это уже довольно ограничено. Что если параметр `sequence` содержит вложенные коллекции, по которым также необходимо выполнить перечисление? Чтобы справиться с такой задачей, приведенный метод придется перегрузить:

```
public static string ToStringEx <T> (IEnumerable<IEnumerable<T>> sequence)
```

А если `sequence` содержит группы или проекции вложенных последовательностей? Статическое решение перегрузки методов становится непрактичным — нам необходим подход, который допускает масштабирование с целью обработки произвольного графа объектов вроде такого:

```
public static string ToStringEx (object value)
{
    if (value == null) return "<null>";
    StringBuilder sb = new StringBuilder();
    if (value is List<>) // Ошибка
        sb.Append ("List of " + ((List<> value).Count + " items")); // Ошибка
    if (value is IGrouping<,>) // Ошибка
        sb.Append ("Group with key=" + ((IGrouping<,> value).Key)); // Ошибка
    // Выполнить перечисление элементов коллекции, если это коллекция,
    // рекурсивно вызывая метод ToStringEx
    // ...
    return sb.ToString();
}
```

К сожалению, код не скомпилируется: обращаться к членам *несвязанного* обобщенного типа, такого как `List<>` или `IGrouping<,>`, нельзя. В случае `List<>` проблему можно решить за счет использования вместо него необобщенного интерфейса `IList`:

```
if (value is IList)
    sb.AppendLine ("A list with " + ((IList) value).Count + " items");
```



Так можно поступать из-за того, что проектировщики типа `List<>` предусмотрительно реализовали классический интерфейс `IList` (а также *обобщенный* интерфейс `IList`). Тот же самый принцип полезно принимать во внимание при написании собственных обобщенных типов: наличие необобщенного интерфейса или базового класса, к которому потребители смогут прибегнуть как к запасному варианту, может оказаться исключительно полезным.

Для `IGrouping<,>` решение не настолько простое. Интерфейс `IGrouping<,>` определен следующим образом:

```
public interface IGrouping < TKey, TElement > : IEnumerable < TElement >,
    IEnumerable
{
    TKey Key { get; }
}
```

Здесь нет никакого необобщенного типа, который можно было бы применить для доступа к свойству `Key`, поэтому в данном случае придется использовать рефлексию. Решение заключается в том, чтобы обращаться не к членам *несвязанного* обобщенного типа (что невозможно), а к членам *закрытого* обобщенного типа, чьи аргументы типа устанавливаются во время выполнения.



В следующей главе мы решим такую задачу более простым способом с помощью ключевого слова `dynamic` языка C#. Хорошим признаком для применения динамического связывания является ситуация, когда в противном случае приходится предпринимать разнообразные трюки с типами, как делается в настоящий момент.

На первом шаге понадобится выяснить, реализует ли `value` интерфейс `IGrouping<,>`, и если да, то получить закрытый обобщенный интерфейс. Проще всего это сделать, выполнив запрос LINQ. Затем производится извлечение и обращение к свойству `Key`:

```
public static string ToStringEx (object value)
{
    if (value == null) return "<null>";
    if (value.GetType().IsPrimitive) return value.ToString();
    StringBuilder sb = new StringBuilder();
    if (value is IList)
        sb.Append ("List of " + ((IList)value).Count + " items: ");
    Type closedIGrouping = value.GetType () .GetInterfaces ()
        .Where (t => t.IsGenericType &&
            t.GetGenericTypeDefinition () == typeof (IGrouping<,>))
        .FirstOrDefault ();
    if (closedIGrouping != null)      // Обратиться к свойству Key
        // реализации IGrouping<,>
    {
        PropertyInfo pi = closedIGrouping.GetProperty ("Key");
        object key = pi.GetValue (value, null);
        sb.Append ("Group with key=" + key + ": ");
    }
    if (value is IEnumerable)
        foreach (object element in ((IEnumerable)value))
            sb.Append (ToStringEx (element) + " ");
    if (sb.Length == 0) sb.Append (value.ToString ());
    return "\r\n" + sb.ToString ();
}
```

Такой подход надежен: он работает независимо от того, как реализован интерфейс `IGrouping<,>` — неявно или явно. В следующем коде демонстрируется использование метода `ToStringEx`:

```
Console.WriteLine (ToStringEx (new List<int> { 5, 6, 7 } ));
Console.WriteLine (ToStringEx ("xyyzzz".GroupBy (c => c )));
```

Вот вывод:

```
List of 3 items: 5 6 7
Group with key=x: x
Group with key=y: y y
Group with key=z: z z z
```

ВЫЗОВ СТАТИЧЕСКИХ ВИРТУАЛЬНЫХ/АБСТРАКТНЫХ ЧЛЕНОВ ИНТЕРФЕЙСОВ

Начиная с .NET 7 и C# 11, в интерфейсах можно определять статические виртуальные и абстрактные члены (см. раздел “Статические виртуальные/абстрактные члены интерфейсов” в главе 3). Примером является интерфейс `IParsable<TSelf>` в .NET:

```
public interface IParsable<TSelf> where TSelf : IParsable<TSelf>
{
    static abstract TSelf Parse (string s, IFormatProvider provider);
    ...
}
```

С помощью ограниченного параметра типа статические абстрактные члены интерфейса можно вызывать полиморфным образом:

```
T ParseAny<T> (string s) where T : IParsable<T> => T.Parse (s, null);
```

Чтобы вызвать статический абстрактный член интерфейса посредством рефлексии, понадобится получить объект `MethodInfo` из конкретного типа, реализующего интерфейс, а не из самого интерфейса. Очевидным решением будет получение конкретного члена по сигнатуре:

```
MethodInfo GetParseMethod (Type concreteType) =>
    concreteType.GetMethod ("Parse",
        new[] { typeof (string), typeof (IFormatProvider) });
```

Однако поступить так не удастся, если член был реализован явно. Для решения этой проблемы в общем виде мы начнем с написания функции, которая извлекает `MethodInfo` для конкретного типа, реализующего указанный метод интерфейса:

```
MethodInfo GetImplementedInterfaceMethod (Type concreteType,
    Type interfaceType, string methodName, Type[] paramTypes)
{
    var map = concreteType.GetInterfaceMap (interfaceType);
    return map.InterfaceMethods
        .Zip (map.TargetMethods)
        .Single (m => m.First.Name == methodName &&
            m.First.GetParameters ().Select (p => p.ParameterType)
                .SequenceEqual (paramTypes))
        .Second;
}
```

Основой здесь является вызов метода `GetInterfaceMap`, который возвращает следующую структуру:

```
public struct InterfaceMapping
{
    public MethodInfo[] InterfaceMethods;           // Все эти массивы имеют
    public MethodInfo[] TargetMethods;              // одну и ту же длину
    ...
}
```

Структура InterfaceMapping сообщает, каким образом члены реализованного интерфейса (InterfaceMethods) сопоставляются с членами конкретного типа (TargetMethods).



Метод GetInterfaceMap также работает с обычными методами (экземпляра); просто он оказывается особенно полезным, когда применяется в отношении статических абстрактных членов интерфейсов.

Затем мы использовали метод Zip из LINQ для выстраивания элементов в двух массивах, что позволило легко получить целевой метод, соответствующий методу интерфейса с желаемой сигнатурой.

Теперь мы можем применять это для написания метода ParseAny на основе рефлексии:

```
object ParseAny (Type type, string value)
{
    MethodInfo parseMethod = GetImplementedInterfaceMethod (type,
        type.GetInterface ("IParsable`1"),
        "Parse",
        new[] { typeof (string), typeof (IFormatProvider) });
    return parseMethod.Invoke (null, new[] { value, null });
}
Console.WriteLine (ParseAny (typeof (float), ".2")); // 0.2
```

При вызове метода GetImplementedInterfaceMethod необходимо предоставить (закрытый) тип интерфейса, который получен в результате вызова GetInterface("IParsable`1") для конкретного типа. Учитывая, что (в данном сценарии) желаемый интерфейс известен во время компиляции, взамен можно было бы использовать следующее выражение:

```
typeof (IParsable<>).MakeGenericType (type)
```

Рефлексия сборок

Для выполнения рефлексии сборки динамическим образом понадобится вызвать метод GetType или GetTypes на объекте Assembly. Приведенный ниже код извлекает из текущей сборки тип по имени TestProgram, определенный в пространстве имен Demos:

```
Type t = Assembly.GetExecutingAssembly ().GetType ("Demos.TestProgram");
```

Сборку можно также получить из существующего типа:

```
typeof (Foo).Assembly.GetType ("Demos.TestProgram");
```

В следующем примере выводится список всех типов в сборке mylib.dll из каталога e:\demo:

```
Assembly a = Assembly.LoadFile (@"e:\demo\mylib.dll");
foreach (Type t in a.GetTypes ())
    Console.WriteLine (t);
```

или:

```
Assembly a = typeof (Foo).GetTypeInfo().Assembly;
foreach (Type t in a.ExportedTypes)
    Console.WriteLine (t);
```

Метод `GetTypes` и свойство `ExportedTypes` возвращают только типы верхнего уровня, но не вложенные типы.

Модули

Вызов метода `GetTypes` на многомодульной сборке возвращает все типы из всех модулей. В результате существование модулей можно проигнорировать и трактовать сборку как контейнер для типов. Однако есть один случай, когда модули имеют значение — работа с маркерами метаданных.

Маркер метаданных представляет собой целое число, которое уникальным образом ссылается на тип, член, строку или ресурс внутри области видимости модуля. Язык IL использует маркеры метаданных, а потому при синтаксическом разборе кода IL вы должны иметь возможность их распознавать. Предназначенные для такой цели методы определены в типе `Module` и называются `ResolveType`, `ResolveMember`, `ResolveString` и `ResolveSignature`. Мы вернемся к ним в последнем разделе главы при написании дизассемблера.

Получить список всех модулей в сборке можно с помощью метода `GetModules`. Свойство `ManifestModule` позволяет напрямую обращаться к главному модулю сборки.

Работа с атрибутами

Среда CLR позволяет посредством атрибутов присоединять дополнительные метаданные к типам, членам и сборкам. Это механизм, с помощью которого производится управление рядом важных функций CLR (таких как идентификация сборок или маршализация типов для собственной возможности взаимодействия), что делает атрибуты неотъемлемой частью приложения.

Ключевая характеристика механизма атрибутов заключается в том, что можно создавать собственные атрибуты и затем применять их подобно любым другим атрибутам для “декорирования” элементов кода дополнительной информацией. Такая дополнительная информация компилируется внутрь лежащей в основе сборки и может быть извлечена во время выполнения с использованием рефлексии для построения декларативно работающих служб, подобных автоматизированному модульному тестированию.

Основы атрибутов

Существуют три вида атрибутов:

- атрибуты с побитовым отображением;
- специальные атрибуты;
- псевдоспециальные атрибуты.

Из них расширяемыми являются только *специальные атрибуты*.



Сам по себе термин “атрибуты” может относиться к любому из указанных выше трех разновидностей, хотя в мире C# чаще всего будут иметься в виду специальные или псевдоспециальные атрибуты.

Атрибуты с побитовым отображением (bit-mapped attributes; наш термин) отображаются на выделенные биты в метаданных типа. Большинство ключевых слов модификаторов C# вроде public, abstract и sealed компилируются именно в атрибуты с побитовым отображением. Эти атрибуты очень эффективны, т.к. они задействуют минимальное пространство в метаданных (обычно всего лишь один бит), и среда CLR может находить их с небольшими затратами или вообще без таковых. Доступ к ним в API-интерфейсе рефлексии открывается через выделенные свойства класса Type (и других подклассов MemberInfo), такие как IsPublic, IsAbstract и IsSealed. Свойство Attributes возвращает перечисление флагов, которое описывает большинство атрибутов:

```
static void Main()
{
    TypeAttributes ta = typeof (Console).Attributes;
    MethodAttributes ma = MethodInfo.GetCurrentMethod().Attributes;
    Console.WriteLine (ta + "\r\n" + ma);
}
```

Ниже показан результат:

```
AutoLayout, AnsiClass, Class, Public, Abstract, Sealed, BeforeFieldInit
PrivateScope, Private, Static, HideBySig
```

По контрасту *специальные атрибуты* компилируются в двоичный блок, который находится в главной таблице метаданных типа. Все специальные атрибуты представлены подклассом класса System.Attribute и в отличие от атрибутов с побитовым отображением являются расширяемыми. Двоичный блок в метаданных идентифицирует класс атрибута, а также хранит значения любых позиционных либо именованных аргументов, которые были указаны, когда атрибут применялся. Специальные атрибуты, определяемые вами самостоятельно, архитектурно идентичны атрибутам, которые определены в библиотеках .NET.

В главе 4 было показано, каким образом присоединять специальные атрибуты к типу или члену в C#. Вот как присоединить предопределенный атрибут Obsolete к классу Foo:

```
[Obsolete] public class Foo { ... }
```

Тем самым компилятору сообщается о необходимости встраивания в метаданные для Foo экземпляра ObsoleteAttribute, который затем может быть извлечен через рефлексию во время выполнения посредством вызова метода GetCustomAttributes на объекте Type или MemberInfo.

Псевдоспециальные атрибуты выглядят и ведут себя подобно стандартным специальным атрибутам. Они представлены подклассом класса System.Attribute и присоединяются в стандартной манере:

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Sequential)]
class SystemTime { ... }
```

Отличие в том, что компилятор или среда CLR внутренне оптимизирует псевдоспециальные атрибуты, преобразуя их в атрибуты с побитовым отображением. Примеры включают `StructLayout`, `In` и `Out` (см. главу 24). Рефлексия открывает доступ к псевдоспециальным атрибутам через выделенные свойства вроде `IsLayoutSequential`, и во многих случаях они также возвращаются в виде объектов `System.Attribute` при вызове метода `GetCustomAttributes`. Это значит, что разница между псевдоспециальными и специальными атрибутами может быть (практически) проигнорирована (заметное исключение — использование пространства имен `Reflection.Emit` для динамической генерации типов во время выполнения; данная тема будет раскрыта в разделе “Выпуск сборок и типов” далее в главе).

Атрибут `AttributeUsage`

`AttributeUsage` является атрибутом, применяемым к классам атрибутов. Он инструктирует компилятор, каким образом должен использоваться целевой атрибут:

```
public sealed class AttributeUsageAttribute : Attribute
{
    public AttributeUsageAttribute (AttributeTargets validOn);
    public bool AllowMultiple      { get; set; }
    public bool Inherited         { get; set; }
    public AttributeTargets ValidOn { get; }
}
```

Свойство `AllowMultiple` управляет тем, может ли определяемый атрибут применяться к одной и той же цели более одного раза. Свойство `Inherited` указывает на то, должен ли атрибут, примененный к базовому классу, применяться также и к производным классам (или в случае методов — должен ли атрибут, примененный к виртуальному методу, применяться также к переопределенным методам). Свойство `ValidOn` определяет набор целей (классов, интерфейсов, свойств, методов, параметров и т.д.), к которым может быть присоединен атрибут. Оно принимает любую комбинацию значений перечисления `AttributeTargets`, которое содержит следующие члены:

All	Delegate	GenericParameter	Parameter
Assembly	Enum	Interface	Property
Class	Event	Method	ReturnValue
Constructor	Field	Module	Struct

В целях иллюстрации ниже показано, как авторы .NET применили атрибут `AttributeUsage` к атрибуту `Serializable`:

```
[AttributeUsage (AttributeTargets.Delegate |
                  AttributeTargets.Enum |
                  AttributeTargets.Struct |
                  AttributeTargets.Class,     Inherited = false)
]
public sealed class SerializableAttribute : Attribute { }
```

Фактически это почти полное определение атрибута `Serializable`. Написание класса атрибута, не имеющего свойств или специальных конструкторов, столь же просто.

Определение собственного атрибута

Вот шаги, которые потребуется выполнить для определения собственного атрибута.

1. Создайте класс, производный от `System.Attribute` или от потомка `System.Attribute`. По соглашению имя класса должно заканчиваться словом “Attribute”, хотя поступать так не обязательно.
2. Примените атрибут `AttributeUsage`, описанный в предыдущем разделе. Если атрибут не требует каких-либо свойств или аргументов в своем конструкторе, то работа закончена.
3. Напишите один или более открытых конструкторов. Параметры конструктора определяют позиционные параметры атрибута и становятся обязательными при использовании атрибута.
4. Объявите открытое поле или свойство для каждого именованного параметра, который планируется поддерживать. При использовании атрибута именованные параметры будут необязательными.



Свойства атрибута и параметры конструктора должны относиться к следующим типам:

- запечатанный примитивный тип, т.е. `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `short` или `string`;
- тип `Type`;
- тип перечисления;
- одномерный массив любого из упомянутых выше типов.

Когда атрибут применяется, у компилятора также должна быть возможность статической оценки каждого свойства или аргумента конструктора.

В следующем классе определяется атрибут для содействия системе автоматизированного модульного тестирования. Он указывает, что метод должен быть протестирован, устанавливает количество повторений теста и задает сообщение, выдаваемое в случае неудачи:

```
[AttributeUsage (AttributeTargets.Method)]
public sealed class TestAttribute : Attribute
{
    public int Repetitions;
    public string FailureMessage;

    public TestAttribute () : this (1) { }
    public TestAttribute (int repetitions) { Repetitions = repetitions; }
}
```

Ниже представлен код класса Foo с методами, которые декорированы атрибутом Test разнообразными способами:

```
class Foo
{
    [Test]
    public void Method1() { ... }

    [Test(20)]
    public void Method2() { ... }

    [Test(20, FailureMessage="Debugging Time!")]
    public void Method3() { ... }
}
```

Извлечение атрибутов во время выполнения

Есть два стандартных способа извлечения атрибутов во время выполнения:

- вызов метода GetCustomAttributes на любом объекте Type или MemberInfo;
- вызов метода Attribute.GetCustomAttribute или Attribute.GetCustomAttributes.

Последние два метода перегружены для приема любого объекта рефлексии, который соответствует допустимой цели атрибута (Type, Assembly, Module, MemberInfo или ParameterInfo).



Для получения информации об атрибутах можно также вызывать метод GetCustomAttributesData на типе или члене. Отличие между этим методом и GetCustomAttributes в том, что первый из них сообщает, *каким образом* атрибут создавался: он указывает перегруженную версию конструктора, которая была использована, и значение каждого аргумента и именованного параметра конструктора. Такие сведения полезны, когда требуется выпускать код или IL для воссоздания атрибута в том же самом состоянии (как объясняется в разделе “Выпуск членов типа” далее в главе).

Ниже показано, каким образом можно выполнить перечисление всех методов в предшествующем классе Foo, которые имеют атрибут TestAttribute:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods())
{
    TestAttribute att = (TestAttribute) Attribute.GetCustomAttribute
        (mi, typeof (TestAttribute));
    if (att != null)
        Console.WriteLine ("Method {0} will be tested; reps={1}; msg={2}",
                           mi.Name, att.Repetitions, att.FailureMessage);
}
```

или:

```
foreach (MethodInfo mi in typeof (Foo).GetTypeInfo().DeclaredMethods)
    ...
```

Вот вывод:

```
Method Method1 will be tested; reps=1; msg=
Method Method2 will be tested; reps=20; msg=
Method Method3 will be tested; reps=20; msg=Debugging Time!
```

Чтобы завершить демонстрацию применения таких приемов при написании системы модульного тестирования, ниже представлен тот же самый пример, расширенный так, чтобы на самом деле вызывать методы, декорированные атрибутом [Test]:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods ())
{
    TestAttribute att = (TestAttribute) Attribute.GetCustomAttribute
        (mi, typeof (TestAttribute));

    if (att != null)
        for (int i = 0; i < att.Repetitions; i++)
            try
            {
                mi.Invoke (new Foo(), null); // Вызвать метод без аргументов
            }
            catch (Exception ex) // Поместить исключение внутрь att.FailureMessage
            {
                throw new Exception ("Error: " + att.FailureMessage, ex); // Ошибка
            }
}
```

Возвращаясь к рефлексии атрибутов, далее представлен пример, в котором выводится список атрибутов, присутствующих в заданном типе:

```
object[] atts = Attribute.GetCustomAttributes (typeof (Test));
foreach (object att in atts) Console.WriteLine (att);

[Serializable, Obsolete]
class Test
{
}
```

Вывод будет выглядеть следующим образом:

```
System.ObsoleteAttribute
System.SerializableAttribute
```

Динамическая генерация кода

Пространство имен `System.Reflection.Emit` содержит классы для создания метаданных и кода IL во время выполнения. Генерация кода динамическим образом полезна для решения определенных видов задач программирования. Примером может служить API-интерфейс регулярных выражений, который выпускает типы, настроенные на специфические регулярные выражения. Еще одним примером является инфраструктура Entity Framework Core, которая использует `Reflection.Emit` для генерации классов-посредников, чтобы сделать возможной ленивую загрузку.

Генерация кода IL с помощью класса `DynamicMethod`

Класс `DynamicMethod` — это легковесный инструмент в пространстве имен `System.Reflection.Emit`, предназначенный для генерации методов на лету. В отличие от `TypeBuilder` он не требует предварительной установки динамической сборки, модуля и типа, в котором должен содержаться метод. Такие характеристики делают класс `DynamicMethod` подходящим средством для решения простых задач, а также хорошим введением в пространство имен `Reflection.Emit`.



Объект `DynamicMethod` и связанный с ним код IL подвергаются сборке мусора, когда на них больше нет ссылок. Это значит, что динамические методы можно генерировать многократно, не заполняя излишне память. (Чтобы делать то же самое с динамическими сборками, при создании сборки потребуется применить флаг `AssemblyBuilderAccess.RunAndCollect`.)

Ниже представлен простой пример использования класса `DynamicMethod` для создания метода, который выводит на консоль строку `Hello world`:

```
public class Test
{
    static void Main()
    {
        var dynMeth = new DynamicMethod ("Foo", null, null, typeof (Test));
        ILGenerator gen = dynMeth.GetILGenerator();
        gen.EmitWriteLine ("Hello world");
        gen.Emit (OpCodes.Ret);
        dynMeth.Invoke (null, null);                                // Hello world
    }
}
```

Для каждого кода операции IL в классе `OpCodes` имеется статическое поле, допускающее только чтение. Большая часть функциональности доступна через различные коды операций, хотя в классе `ILGenerator` также есть специализированные методы для генерации меток и локальных переменных и для обработки исключений. Метод всегда завершается кодом операции `OpCodes.Ret`, который означает “возврат”, или разновидностью инструкции ветвления/генерации. Метод `EmitWriteLine` класса `ILGenerator` — это сокращение для выпуска нескольких кодов операций более низкого уровня. Мы могли бы получить тот же самый результат, заменив вызов `EmitWriteLine` следующим образом:

```
MethodInfo writeLineStr = typeof (Console).GetMethod ("WriteLine",
    new Type[] { typeof (string) });
gen.Emit (OpCodes.Ldstr, "Hello world");                      // Загрузить строку
gen.Emit (OpCodes.Call, writeLineStr);                          // Вызвать метод
```

Обратите внимание, что мы передаем конструктору `DynamicMethod` аргумент `typeof (Test)`. Это предоставляет динамическому методу доступ к неоткрытым методам данного типа, разрешая поступать следующим образом:

```

public class Test
{
    static void Main()
    {
        var dynMeth = new DynamicMethod ("Foo", null, null, typeof (Test));
        ILGenerator gen = dynMeth.GetILGenerator();

        MethodInfo privateMethod = typeof (Test).GetMethod ("HelloWorld",
            BindingFlags.Static | BindingFlags.NonPublic);
        gen.Emit (OpCodes.Call, privateMethod);           // Вызвать метод HelloWorld
        gen.Emit (OpCodes.Ret);
        dynMeth.Invoke (null, null);                    // Hello world
    }

    static void HelloWorld()      // Закрытый метод, но мы можем вызвать его
    {
        Console.WriteLine ("Hello world");
    }
}

```

Освоение языка IL требует существенного времени. Вместо запоминания всех кодов операций намного проще скомпилировать какую-нибудь программу C# и затем исследовать, копировать и настраивать код IL. Средство LINQPad отображает код IL для любого метода или фрагмента кода, который вы введете, а инструменты для просмотра сборок, такие как ildasm или .NET Reflector, удобны для изучения существующих сборок.

Стек вычислений

Центральной концепцией в IL является *стек вычислений*. Чтобы вызвать метод с аргументами, сначала понадобится затолкнуть (“загрузить”) аргументы в стек вычислений и затем вызвать метод. Впоследствии метод извлекает необходимые аргументы из стека вычислений. Мы демонстрировали прием ранее при вызове `Console.WriteLine`. Ниже приведен похожий пример с целым числом:

```

var dynMeth = new DynamicMethod ("Foo", null, null, typeof (void));
ILGenerator gen = dynMeth.GetILGenerator();
MethodInfo writeLineInt = typeof (Console).GetMethod ("WriteLine",
    new Type[] { typeof (int) });

//Коды операций Ldc* загружают числовые литералы различных типов и размеров
gen.Emit (OpCodes.Ldc_I4, 123); //Затолкнуть в стек 4-байтовое целое число
gen.Emit (OpCodes.Call, writeLineInt);

gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null);      // 123

```

Чтобы сложить два числа, нужно загрузить их в стек вычислений и вызвать `Add`. Код операции `Add` извлекает два значения из стека вычислений и засыпывает результат обратно в стек. Следующий код суммирует числа 2 и 2, после чего выводит результат с применением полученного ранее метода `WriteLine`:

```
gen.Emit (OpCodes.Ldc_I4, 2);      // Затолкнуть 4-байтовое целое число,  
                                    // значение = 2  
gen.Emit (OpCodes.Ldc_I4, 2);      // Затолкнуть 4-байтовое целое число,  
                                    // значение = 2  
gen.Emit (OpCodes.Add);           // Сложить и получить результат  
gen.Emit (OpCodes.Call, writeLineInt);
```

Чтобы вычислить выражение $10/2+1$, можно поступить либо так:

```
gen.Emit (OpCodes.Ldc_I4, 10);  
gen.Emit (OpCodes.Ldc_I4, 2);  
gen.Emit (OpCodes.Div);  
gen.Emit (OpCodes.Ldc_I4, 1);  
gen.Emit (OpCodes.Add);  
gen.Emit (OpCodes.Call, writeLineInt);
```

либо так:

```
gen.Emit (OpCodes.Ldc_I4, 1);  
gen.Emit (OpCodes.Ldc_I4, 10);  
gen.Emit (OpCodes.Ldc_I4, 2);  
gen.Emit (OpCodes.Div);  
gen.Emit (OpCodes.Add);  
gen.Emit (OpCodes.Call, writeLineInt);
```

Передача аргументов динамическому методу

Коды операций `Ldarg` и `Ldarg_XXX` загружают в стек аргумент, переданный методу. Чтобы значение возвратилось, перед завершением оно должно оставаться единственным значением в стеке. Для этого при вызове конструктора `DynamicMethod` потребуется указать возвращаемый тип и типы аргументов. В показанном ниже коде создается динамический метод, который возвращает сумму двух целых чисел:

```
DynamicMethod dynMeth = new DynamicMethod ("Foo",  
    typeof (int),                                // Возвращаемый тип: int  
    new[] { typeof (int), typeof (int) },          // Типы параметров: int, int  
    typeof (void));  
  
ILGenerator gen = dynMeth.GetILGenerator();  
  
gen.Emit (OpCodes.Ldarg_0); // Затолкнуть в стек вычислений первый аргумент  
gen.Emit (OpCodes.Ldarg_1); // Затолкнуть в стек вычислений второй аргумент  
gen.Emit (OpCodes.Add);   // Сложить аргументы (результат остается в стеке)  
gen.Emit (OpCodes.Ret);   // Возврат при стеке, содержащем одно значение  
  
int result = (int) dynMeth.Invoke (null, new object[] { 3, 4 } ); // 7
```



По завершении стек вычислений должен содержать в точности 0 или 1 элемент (в зависимости от того, возвращает ли метод значение). Если нарушить данное требование, то среда CLR откажется выполнять метод. Удалить элемент из стека без обработки можно с помощью кода операции `OpCodes.Pop`.

Вместо вызова `Invoke` иногда удобнее оперировать динамическим методом как типизированным делегатом, для чего предназначен метод `CreateDelegate`. В нашем случае необходимый делегат имеет два целочисленных параметра и целочисленный возвращаемый тип. Для этой цели мы можем использовать делегат `Func<int, int, int>`. Тогда последнюю строку в предыдущем примере можно было бы заменить следующими строками:

```
var func = (Func<int,int,int>) dynMeth.CreateDelegate  
          (typeof (Func<int,int,int>));  
int result = func (3, 4); // 7
```



Делегат также устраняет накладные расходы, связанные с динамическим вызовом метода, экономя несколько микросекунд на вызов.

Мы покажем, как передавать ссылку, в разделе “Выпуск членов типа” далее в главе.

Генерация локальных переменных

Объявить локальную переменную можно путем вызова метода `DeclareLocal` на экземпляре `ILGenerator`. В результате возвращается объект `LocalBuilder`, который можно использовать в сочетании с кодами операций, такими как `Ldloc` (загрузить локальную переменную) или `Stloc` (сохранить локальную переменную). Операция `Ldloc` засыпает в стек вычислений, а `Stloc` извлекает из него. Например, взгляните на показанный далее код C#:

```
int x = 6;  
int y = 7;  
x *= y;  
Console.WriteLine (x);
```

Приведенный ниже код динамически генерирует предыдущий код:

```
var dynMeth = new DynamicMethod ("Test", null, null, typeof (void));  
ILGenerator gen = dynMeth.GetILGenerator();  
  
LocalBuilder localX = gen.DeclareLocal (typeof (int)); // Объявить  
// переменную x  
LocalBuilder localY = gen.DeclareLocal (typeof (int)); // Объявить  
// переменную y  
  
gen.Emit (OpCodes.Ldc_I4, 6); // Затолкнуть в стек вычислений литерал 6  
gen.Emit (OpCodes.Stloc, localX); // Сохранить в localX  
gen.Emit (OpCodes.Ldc_I4, 7); // Затолкнуть в стек вычислений литерал 7  
gen.Emit (OpCodes.Stloc, localY); // Сохранить в localY  
  
gen.Emit (OpCodes.Ldloc, localX); // Затолкнуть в стек вычислений localX  
gen.Emit (OpCodes.Ldloc, localY); // Затолкнуть в стек вычислений localY  
gen.Emit (OpCodes.Mul); // Перемножить значения  
gen.Emit (OpCodes.Stloc, localX); // Сохранить результат в localX  
  
gen.EmitWriteLine (localX); // Вывести значение localX  
gen.Emit (OpCodes.Ret);  
  
dynMeth.Invoke (null, null); // 42
```

Ветвление

В языке IL отсутствуют циклы вроде `while`, `do` и `for`; вся работа делается с помощью меток плюс эквивалентов оператора `goto` и условного оператора `goto`. Существуют коды операций ветвления, такие как `Br` (безусловное ветвление), `Brtrue` (ветвление, если значение в стеке вычислений равно `true`) и `Blt` (ветвление, если первое значение меньше второго значения).

Для установки цели ветвления сначала понадобится вызвать метод `DefineLabel` (он возвращает объект `Label`) и затем вызвать метод `MarkLabel` в месте, к которому должна быть прикреплена метка. Например, рассмотрим следующий код C#:

```
int x = 5;
while (x <= 10) Console.WriteLine (x++);
```

Выпустить его можно так:

```
ILGenerator gen = ...
Label startLoop = gen.DefineLabel(); // Объявить метки
Label endLoop = gen.DefineLabel();
LocalBuilder x = gen.DeclareLocal (typeof (int)); // int x
gen.Emit (OpCodes.Ldc_I4, 5); // // x = 5
gen.Emit (OpCodes.Stloc, x);
gen.MarkLabel (startLoop);
gen.Emit (OpCodes.Ldc_I4, 10); // Загрузить в стек вычислений 10
gen.Emit (OpCodes.Ldloc, x); // Загрузить в стек вычислений x
gen.Emit (OpCodes.Blt, endLoop); // if (x > 10) goto endLoop
gen.EmitWriteLine (x); // Console.WriteLine (x)
gen.Emit (OpCodes.Ldloc, x); // Загрузить в стек вычислений x
gen.Emit (OpCodes.Ldc_I4, 1); // Загрузить в стек вычислений 1
gen.Emit (OpCodes.Add); // Выполнить сложение
gen.Emit (OpCodes.Stloc, x); // Сохранить результат в x
gen.Emit (OpCodes.Br, startLoop); // Вернуться в начало цикла
gen.MarkLabel (endLoop);
gen.Emit (OpCodes.Ret);
```

Создание объектов и вызов методов экземпляра

Эквивалентом операции `new` в языке IL является код операции `Newobj`, который обращается к конструктору и загружает созданный объект в стек вычислений. Например, следующий код конструирует объект `StringBuilder`:

```
var dynMeth = new DynamicMethod ("Test", null, null, typeof (void));
ILGenerator gen = dynMeth.GetILGenerator();
ConstructorInfo ci = typeof (StringBuilder).GetConstructor (new Type[0]);
gen.Emit (OpCodes.Newobj, ci);
```

После загрузки объекта в стек вычислений можно применять код операции `Call` или `Callvirt` для вызова его методов экземпляра. Расширяя рассматриваемый пример, мы запросим свойство `MaxCapacity` объекта `StringBuilder` путем вызова метода доступа `get` свойства и затем выведем результат:

```

gen.Emit (OpCodes.Callvirt, typeof (StringBuilder)
           .GetProperty ("MaxCapacity").GetMethod());
gen.Emit (OpCodes.Call, typeof (Console).GetMethod ("WriteLine",
           new[] { typeof (int) } ));
gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null); // 2147483647

```

Вот как эмулировать семантику вызовов C#:

- используйте код операции Call для обращения к статическим методам и методам экземпляра типов значений;
- применяйте код операции Callvirt для обращения к методам экземпляра ссылочных типов (независимо от того, объявлены они виртуальными или нет).

В данном примере мы использовали Callvirt на экземпляре StringBuilder, несмотря на то, что свойство MaxCapacity не является виртуальным. Это не приводит к ошибке, а просто выполняет невиртуальный вызов. Вызов методов экземпляра ссылочных типов с помощью Callvirt позволяет избежать риска возникновения противоположного условия: обращения к виртуальному методу посредством Call. (Риск вполне реален. Автор целевого метода может позже изменить его объявление.) Преимущество операции Callvirt также в том, что она обеспечивает проверку получателя на равенство null.



Вызов виртуального метода с помощью операции Call обходит семантику виртуальных вызовов и обращается к методу напрямую, что редко является желательным и на самом деле нарушает безопасность типов.

В следующем примере мы создаем объект StringBuilder, передавая конструктору два аргумента, добавляем к нему строку ", world!" и вызываем метод ToString на этом объекте:

```

// Мы будем вызывать: new StringBuilder ("Hello", 1000)
ConstructorInfo ci = typeof (StringBuilder).GetConstructor (
    new[] { typeof (string), typeof (int) } );
gen.Emit (OpCodes.Ldstr, "Hello"); // Загрузить в стек вычислений строку
gen.Emit (OpCodes.Ldc_I4, 1000); // Загрузить в стек вычислений целое число
gen.Emit (OpCodes.Newobj, ci); // Сконструировать объект StringBuilder
Type[] strT = { typeof (string) };
gen.Emit (OpCodes.Ldstr, ", world!");
gen.Emit (OpCodes.Call, typeof (StringBuilder).GetMethod ("Append", strT));
gen.Emit (OpCodes.Callvirt, typeof (object).GetMethod ("ToString"));
gen.Emit (OpCodes.Call, typeof (Console).GetMethod ("WriteLine", strT));
gen.Emit (OpCodes.Ret);
dynMeth.Invoke (null, null); // Hello, world!

```

Ради интереса мы вызвали метод GetMethod на typeof (object), после чего использовали операцию Callvirt для выполнения вызова виртуального ме-

тода на `ToString`. Тот же результат можно было бы получить, вызвав метод `ToString` на самом типе `StringBuilder`:

```
gen.Emit (OpCodes.Callvirt, typeof (StringBuilder).GetMethod ("ToString",
    new Type[0] ));
```

(При вызове методу `GetMethod` должен передаваться пустой массив `Type`, т.к. `StringBuilder` перегружает метод `ToString` с применением другой сигнатуры.)



Если бы метод `ToString` типа `object` вызывался невиртуальным образом:

```
gen.Emit (OpCodes.Call,
    typeof (object).GetMethod ("ToString"));
```

тогда результатом оказалась бы строка `System.Text.StringBuilder`. Другими словами, мы должны обойти версию `ToString`, переопределенную в классе `StringBuilder`, и вызвать версию данного метода из `object`.

Обработка исключений

Класс `ILGenerator` предлагает выделенные методы для обработки исключений. Скажем, приведенный ниже код C#:

```
try                                { throw new NotSupportedException(); }
catch (NotSupportedException ex) { Console.WriteLine (ex.Message);      }
finally                            { Console.WriteLine ("Finally");     }
```

можно сгенерировать следующим образом:

```
MethodInfo getMessageProp = typeof (NotSupportedException)
    .GetProperty ("Message").GetGetMethod();
MethodInfo writeLineString = typeof (Console).GetMethod ("WriteLine",
    new[] { typeof (object) } );
gen.BeginExceptionBlock ();
ConstructorInfo ci = typeof (NotSupportedException).GetConstructor (
    new Type[0] );
gen.Emit (OpCodes.Newobj, ci);
gen.Emit (OpCodes.Throw);
gen.BeginCatchBlock (typeof (NotSupportedException));
gen.Emit (OpCodes.Callvirt, getMessageProp);
gen.Emit (OpCodes.Call, writeLineString);
gen.BeginFinallyBlock ();
gen.EmitWriteLine ("Finally");
gen.EndExceptionBlock();
```

Как и в языке C#, можно иметь много блоков `catch`. Для повторной генерации исключения понадобится выпустить код операции `Rethrow`.



Класс `ILGenerator` предоставляет вспомогательный метод по имени `ThrowException`. Однако он содержит ошибку, которая не дает возможности его использовать с экземпляром `DynamicMethod`. Упомянутый метод работает только с экземпляром `MethodBuilder` (как будет показано в следующем разделе).

Выпуск сборок и типов

Несмотря на удобство класса `DynamicMethod`, он может генерировать только методы. Если необходимо выпускать любые другие конструкции (или целый тип), то придется применять полный “тяжеловесный” API-интерфейс. Это означает динамическое построение сборки и модуля. Тем не менее, сборка не обязательно должна находиться на диске (на самом деле она и не может, потому что .NET 5+ и .NET Core не разрешают сохранять сгенерированные сборки на диске).

Давайте предположим, что требуется динамически построить тип. Поскольку тип должен находиться в модуле внутри сборки, необходимо сначала создать сборку и модуль, чтобы создание типа стало возможным. За такую работу отвечают классы `AssemblyBuilder` и `ModuleBuilder`:

```
AssemblyName fname = new AssemblyName ("MyDynamicAssembly");
AssemblyBuilder assemBuilder =
    AssemblyBuilder.DefineDynamicAssembly (fname, AssemblyBuilderAccess.Run);
ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule ("DynModule");
```



Добавить тип в существующую сборку не удастся, т.к. после создания сборка становится неизменяемой.

Динамические сборки не подвергаются обработке сборщиком мусора и остаются в памяти вплоть до окончания процесса, если только при их определении не был указан флаг `AssemblyBuilderAccess.RunAndCollect`. К сборкам, которые могут быть обработаны сборщиком мусора, применяются различные ограничения (<http://albahari.com/dynamiccollect>).

Получив модуль, в котором способен находиться тип, для создания типа можно использовать класс `TypeBuilder`. Вот как определить класс по имени `Widget`:

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
```

Перечисление флагов `TypeAttributes` поддерживает модификаторы типов CLR, которые можно увидеть после дизассемблирования типа с помощью `ildasm`. Помимо флагов видимости членов это перечисление включает такие модификаторы типов, как `Abstract` и `Sealed`, а также `Interface` для определения интерфейса .NET. Кроме того, имеется флаг `Serializable`, который эквивалентен применению атрибута `[Serializable]` в C#, и `Explicit`, эквивалентный применению атрибута `[StructLayout(LayoutKind.Explicit)]`. Мы покажем, как работать с другими разновидностями атрибутов, в разделе “Присоединение атрибутов” далее в главе.



Метод `DefineType` также принимает необязательный базовый тип:

- для определения структуры укажите базовый тип `System.ValueType`;
- для определения делегата укажите базовый тип `System.MulticastDelegate`;

- для реализации интерфейса используйте конструктор, который принимает массив типов интерфейсов;
- для определения интерфейса укажите комбинацию `TypeAttributes.Interface | TypeAttributes.Abstract`.

Определение типа делегата требует выполнения нескольких дополнительных шагов. Джоэль Побар объясняет, как это сделать, в своей статье “Creating delegate types via `Reflection.Emit`” (“Создание типов делегатов через `Reflection.Emit`”), доступной по ссылке <http://www.albahari.com/joelpob>.

Теперь внутри типа можно создавать члены:

```
MethodBuilder methBuilder = tb.DefineMethod ("SayHello",
                                             MethodAttributes.Public,
                                             null, null);
ILGenerator gen = methBuilder.GetILGenerator();
gen.EmitWriteLine ("Hello world");
gen.Emit (OpCodes.Ret);
```

Для создания типа все готово и ниже представлено завершение его определения:

```
Type t = tb.CreateType();
```

После того, как тип создан, с помощью обычной рефлексии его можно инспектировать и производить позднее связывание:

```
object o = Activator.CreateInstance (t);
t.GetMethod ("SayHello").Invoke (o, null); // Hello world
```

Объектная модель `Reflection.Emit`

На рис. 18.2 показаны основные типы в пространстве имен `System.Reflection.Emit`. Каждый тип описывает конструкцию CLR и основан на эквиваленте из пространства `System.Reflection`. В результате при построении какого-то типа на месте обычных конструкций можно применять генерированные конструкции. Например, ранее мы вызывали метод `Console.WriteLine` следующим образом:

```
MethodInfo writeLine = typeof (Console).GetMethod ("WriteLine",
                                                   new Type[] { typeof (string) });
gen.Emit (OpCodes.Call, writeLine);
```

Мы могли бы столь же легко вызвать динамически генерированный метод, обратившись к методу `gen.Emit` и передав ему объект `MethodInfo`, а не `MethodInfo`. Это очень важно — иначе не было бы возможности написать один динамический метод, который вызывает другой метод в том же типе.

Вспомните, что по завершении наполнения объекта `TypeBuilder` должен быть вызван его метод `CreateType`. Вызов `CreateType` запечатывает объект `TypeBuilder` и все его члены — так что ничего больше не может быть добавлено либо изменено — и возвращает обратно реальный тип `Type`, экземпляры которого можно создавать.

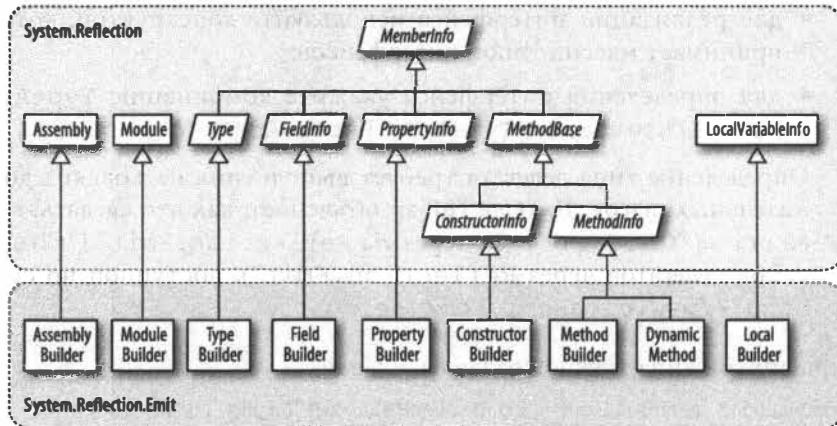


Рис. 18.2. Пространство имен System.Reflection.Emit

Перед вызовом метода `CreateType` объект `TypeBuilder` и его члены находятся в “несозданном” состоянии. Существуют значительные ограничения относительно того, что можно делать с несозданными конструкциями. В частности, нельзя вызывать члены, возвращающие объекты `MethodInfo`, такие как `GetMembers`, `GetMethod` или `GetProperty` — это приведет к генерации исключения. Чтобы сослаться на члены несозданного типа, придется использовать исходные выпуски:

```
TypeBuilder tb = ...  
  
MethodBuilder method1 = tb.DefineMethod ("Method1", ...);  
MethodBuilder method2 = tb.DefineMethod ("Method2", ...);  
  
ILGenerator gen1 = method1.GetILGenerator();  
  
// Предположим, что method1 должен вызывать method2:  
gen1.Emit (OpCodes.Call, method2); // Правильно  
gen1.Emit (OpCodes.Call, tb.GetMethod ("Method2")); // Неправильно
```

После вызова метода `CreateType` можно проводить рефлексию и активизацию не только возвращенного объекта `Type`, но также исходного объекта `TypeBuilder`. Фактически `TypeBuilder` превращается в посредника для реального `Type`. Мы покажем, почему такая возможность важна, в разделе “Сложности, связанные с генерацией” далее в главе.

Выпуск членов типа

Во всех примерах настоящего раздела предполагается, что объект типа `TypeBuilder` по имени `tb` был создан следующим образом:

```
AssemblyName fname = new AssemblyName ("MyEmissions");  
  
AssemblyBuilder assemBuilder = AssemblyBuilder.DefineDynamicAssembly (fname, AssemblyBuilderAccess.Run);  
  
ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule ("MainModule");  
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
```

Выпуск методов

При вызове метода `DefineMethod` можно указывать возвращаемый тип и типы параметров в той же самой манере, как и при создании объекта `DynamicMethod`. Например, следующий метод:

```
public static double SquareRoot (double value) => Math.Sqrt (value);
```

может быть сгенерирован так:

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
    MethodAttributes.Static | MethodAttributes.Public,
    CallingConventions.Standard,
    typeof (double), // Возвращаемый тип
    new[] { typeof (double) } ); // Типы параметров

mb.DefineParameter (1, ParameterAttributes.None, "value"); // Назначить имя

ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0); // Загрузить первый аргумент
gen.Emit (OpCodes.Call, typeof (Math).GetMethod ("Sqrt"));
gen.Emit (OpCodes.Ret);

Type realType = tb.CreateType();
double x = (double) tb.GetMethod ("SquareRoot").Invoke (null,
                                                       new object[] { 10.0 });
Console.WriteLine (x); // 3.16227766016838
```

Вызов метода `DefineParameter` является необязательным и обычно делается для назначения параметру имени. Число 1 ссылается на первый параметр (0 соответствует возвращаемому значению). Если `DefineParameter` не вызывается, тогда параметры неявно именуются как `_p1`, `_p2` и т.д. Назначение имен имеет смысл, если сборка будет записываться на диск; оно делает методы дружественными к потребителям.



Метод `DefineParameter` возвращает объект `ParameterBuilder`, на котором можно вызывать метод `SetCustomAttribute` для присоединения атрибутов (см. раздел “Присоединение атрибутов” далее в главе).

Для выпуска параметров, передаваемых по ссылке, таких как параметр в следующем методе C#:

```
public static void SquareRoot (ref double value)
    => value = Math.Sqrt (value);
```

необходимо вызвать метод `MakeByRefType` на типе параметра (или типах):

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
    MethodAttributes.Static | MethodAttributes.Public,
    CallingConventions.Standard,
    null,
    new Type[] { typeof (double).MakeByRefType () } );

mb.DefineParameter (1, ParameterAttributes.None, "value");

ILGenerator gen = mb.GetILGenerator();
```

```
gen.Emit (OpCodes.Ldarg_0);
gen.Emit (OpCodes.Ldarg_0);
gen.Emit (OpCodes.Ldind_R8);
gen.Emit (OpCodes.Call, typeof (Math).GetMethod ("Sqrt"));
gen.Emit (OpCodes.Stind_R8);
gen.Emit (OpCodes.Ret);

Type realType = tb.CreateType();
object[] args = { 10.0 };
tb.GetMethod ("SquareRoot").Invoke (null, args);
Console.WriteLine (args[0]); // 3.16227766016838
```

Здесь коды операций были скопированы из дизассемблированного метода C#. Обратите внимание на разницу в семантике для доступа к параметрам, передаваемым по ссылке: коды операций Ldind и Stind означают соответственно “загрузить косвенно” (load indirectly) и “сохранить косвенно” (store indirectly). Сuffix R8 означает 8-байтовое число с плавающей точкой. Процесс выпуска параметров out идентичен за исключением того, что метод DefineParameter вызывается следующим образом:

```
mb.DefineParameter (1, ParameterAttributes.Out, "value");
```

Генерация методов экземпляра

Чтобы сгенерировать метод экземпляра, при вызове DefineMethod понадобится указать флаг MethodAttributes.Instance:

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
    MethodAttributes.Instance | MethodAttributes.Public
    ...
```

В случае методов экземпляра нулевым аргументом неявно является this; нумерация остальных аргументов начинается с 1. Таким образом, Ldarg_0 загружает в стек вычислений this, а Ldarg_1 загружает первый реальный аргумент метода.

Переопределение методов

Переопределять виртуальный метод в базовом классе легко: нужно просто определить метод с идентичным именем, сигнатурой и возвращаемым типом, указав при вызове DefineMethod флаг MethodAttributes.Virtual. То же самое применимо при реализации методов интерфейса.

В классе TypeBuilder также доступен метод по имени DefineMethod Override, который переопределяет метод с другим именем. Использовать его имеет смысл только с явной реализацией интерфейса; в остальных сценариях следует применять метод DefineMethod.

Флаг HideBySig

В случае построения подкласса другого типа при определении методов почти всегда полезно указывать флаг MethodAttributes.HideBySig. Флаг HideBySig обеспечивает использование семантики скрытия методов в стиле C#, которая заключается в том, что метод базового класса скрывается только в случае, если в подтипе определен метод с такой же сигнатурой. Без HideBySig скрытие методов основывается только на имени, поэтому метод Foo(string) в подтипе скроет метод Foo() в базовом типе, хотя подобное обычно нежелательно.

Выпуск полей и свойств

Для создания поля необходимо вызвать метод `DefineField` на объекте `TypeBuilder`, указав ему желаемое имя поля, тип и видимость. Следующий код создает закрытое целочисленное поле по имени `length`:

```
FieldBuilder field = tb.DefineField ("length", typeof (int),  
    FieldAttributes.Private);
```

Создание свойства или индексатора требует выполнения нескольких дополнительных шагов. Первый из них — вызов метода `DefineProperty` на объекте `TypeBuilder` с передачей ему имени и типа свойства:

```
PropertyBuilder prop = tb.DefineProperty (  
    "Text", // Имя свойства  
    PropertyAttributes.None,  
    typeof (string), // Тип свойства  
    new Type[0] // Типы индексатора  
) ;
```

(При создании индексатора последний аргумент представляет собой массив типов индексатора.) Обратите внимание, что мы не указываем видимость свойства: это делается в индивидуальном порядке на основе методов аксессора.

Следующий шаг заключается в написании методов `get` и `set`. По соглашению их имена имеют префикс `get_` или `set_`. Затем готовые методы можно присоединить к свойству с помощью вызова методов `SetGetMethod` и `SetSetMethod` на объекте `PropertyBuilder`.

В качестве полного примера мы возьмем показанное ниже объявление поля и свойства:

```
string _text;  
public string Text  
{  
    get      => _text;  
    internal set => _text = value;  
}
```

и сгенерируем его динамически:

```
FieldBuilder field = tb.DefineField ("_text", typeof (string),  
    FieldAttributes.Private);  
PropertyBuilder prop = tb.DefineProperty (  
    "Text", // Имя свойства  
    PropertyAttributes.None,  
    typeof (string), // Тип свойства  
    new Type[0] // Типы индексатора  
MethodBuilder getter = tb.DefineMethod (  
    "get_Text", // Имя метода  
    MethodAttributes.Public | MethodAttributes.SpecialName,  
    typeof (string), // Возвращаемый тип  
    new Type[0]); // Типы параметров  
ILGenerator getGen = getter.GetILGenerator();  
getGen.Emit (OpCodes.Ldarg_0); // Загрузить в стек вычислений this  
getGen.Emit (OpCodes.Ldfld, field); // Загрузить в стек вычислений  
// значение свойства
```

```

getGen.Emit (OpCodes.Ret);           // Выполнить возврат
MethodBuilder setter = tb.DefineMethod (
    "set_Text",
    MethodAttributes.Assembly | MethodAttributes.SpecialName,
    null,                                // Возвращаемый тип
    new Type[] { typeof (string) } );      // Типы параметров
ILGenerator setGen = setter.GetILGenerator();
setGen.Emit (OpCodes.Ldarg_0);         // Загрузить в стек вычислений this
setGen.Emit (OpCodes.Ldarg_1);         // Загрузить в стек вычислений
                                       // второй аргумент, т.е. значение
setGen.Emit (OpCodes.Stfld, field);   // Сохранить значение в поле
setGen.Emit (OpCodes.Ret);            // Выполнить возврат
prop.SetGetMethod (getter);          // Связать метод get и свойство
prop.SetSetMethod (setter);          // Связать метод set и свойство

```

Теперь свойство можно протестировать:

```

Type t = tb.CreateType();
object o = Activator.CreateInstance (t);
t.GetProperty ("Text").SetValue (o, "Good emissions!", new object[0]);
string text = (string) t.GetProperty ("Text").GetValue (o, null);
Console.WriteLine (text);             // Good emissions!

```

Обратите внимание, что в определении `MethodAttributes` для аксессора был включен флаг `SpecialName`. Он инструктирует компилятор о том, что прямое связывание с такими методами при статической ссылке на сборку не разрешено. Это также гарантирует соответствующую поддержку аксессоров инструментами рефлексии и средством `IntelliSense` в `Visual Studio`.



Выпускать события можно аналогично, вызывая метод `DefineEvent` на объекте `TypeBuilder`. Затем можно написать явные методы аксессора и присоединить их к объекту `EventBuilder` путем вызова методов `SetAddOnMethod` и `SetRemoveOnMethod`.

Выпуск конструкторов

Чтобы определить собственные конструкторы, понадобится вызвать метод `DefineConstructor` на объекте `TypeBuilder`. Поступать так не обязательно — стандартный конструктор без параметров будет предоставлен автоматически, если не было явно определено ни одного конструктора. В случае подтипа стандартный конструктор вызывает конструктор базового класса — точно как в C#. Определение одного или большего числа конструктоeв приводит к устранению стандартного конструктора.

Конструктор является удобным местом для инициализации полей. На самом деле он представляет собой единственное такое место: инициализаторы полей C# не имеют специальной поддержки в CLR — это просто синтаксическое сокращение для присваивания значений полям в конструкторе.

Таким образом, для воспроизведения следующего кода:

```
class Widget
{
    int _capacity = 4000;
}
```

потребуется определить конструктор, как показано ниже:

```
FieldBuilder field = tb.DefineField ("_capacity", typeof (int),
                                      FieldAttributes.Private);
ConstructorBuilder c = tb.DefineConstructor (
    MethodAttributes.Public,
    CallingConventions.Standard,
    new Type[0]);                                // Параметры конструктора
ILGenerator gen = c.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);          // Загрузить в стек вычислений this
gen.Emit (OpCodes.Ldc_I4, 4000);      // Загрузить в стек вычислений 4000
gen.Emit (OpCodes.Stfld, field);     // Сохранить это в поле field
gen.Emit (OpCodes.Ret);
```

Вызов конструкторов базовых классов

При построении подкласса другого типа конструктор, который был только что написан, *обойдет конструктор базового класса*. Ситуация отличается от C#, где конструктор базового класса вызывается всегда, прямо или косвенно. Например, имея приведенный далее код:

```
class A { public A() { Console.Write ("A"); } }
class B : A { public B() {} }
```

компилятор в действительности будет транслировать вторую строку следующим образом:

```
class B : A { public B() : base() {} }
```

Однако это не так, когда генерируется код IL: если нужно, чтобы конструктор базового класса был выполнен, то он должен вызываться явно (что происходит почти всегда). Предполагая, что базовый класс имеет имя A, вот как нужно поступить:

```
gen.Emit (OpCodes.Ldarg_0);
ConstructorInfo baseConstr = typeof (A).GetConstructor (new Type[0]);
gen.Emit (OpCodes.Call, baseConstr);
```

Конструкторы с аргументами вызываются точно так же, как обычные методы.

Присоединение атрибутов

Присоединить специальные атрибуты к динамической конструкции можно путем вызова метода `SetCustomAttribute` с передачей ему объекта `CustomAttributeBuilder`. Например, пусть необходимо присоединить к полю или свойству следующее объявление атрибута:

```
[XmlElement ("FirstName", Namespace="http://test/", Order=3)]
```

Объявление полагается на конструктор класса `XmlElementAttribute`, который принимает одиночную строку. Для работы с объектом `CustomAttributeBuilder` потребуется извлечь как указанный конструктор, так и два дополнительных свойства, подлежащие установке (`Namespace` и `Order`):

```
Type attType = typeof (XmlElementAttribute);
ConstructorInfo attConstructor = attType.GetConstructor (
    new Type[] { typeof (string) } );
var att = new CustomAttributeBuilder (
    attConstructor, // Конструктор
    new object[] { "FirstName" }, // Аргументы конструктора
    new PropertyInfo[]
    {
        attType.GetProperty ("Namespace"), // Свойства
        attType.GetProperty ("Order")
    },
    new object[] { "http://test/", 3 } // Значения свойств
);
myFieldBuilder.SetCustomAttribute (att);
// или propBuilder.SetCustomAttribute (att);
// или typeBuilder.SetCustomAttribute (att); и т.д.
```

Выпуск обобщенных методов и типов

Во всех примерах раздела предполагается, что объект `modBuilder` был создан следующим образом:

```
AssemblyName fname = new AssemblyName ("MyEmissions");
AssemblyBuilder assemBuilder = AssemblyBuilder.DefineDynamicAssembly (
    fname, AssemblyBuilderAccess.Run);
ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule ("MainModule");
```

Определение обобщенных методов

Для выпуска обобщенного метода выполните перечисленные шаги.

1. Вызовите метод `DefineGenericParameters` на объекте `MethodBuilder`, чтобы получить массив объектов `GenericTypeParameterBuilder`.
2. Вызовите метод `SetSignature` на объекте `MethodBuilder` с применением этих параметров обобщенных типов (т.е. массива объектов `GenericTypeParameterBuilder`).
3. При желании назначьте параметрам другие имена.

Например, следующий обобщенный метод:

```
public static T Echo<T> (T value)
{
    return value;
}
```

можно было бы сгенерировать так:

```

TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
MethodBuilder mb = tb.DefineMethod ("Echo", MethodAttributes.Public |
                                     MethodAttributes.Static);
GenericTypeParameterBuilder[] genericParams
    = mb.DefineGenericParameters ("T");
mb.SetSignature (genericParams[0],           // Возвращаемый тип
                 null, null,
                 genericParams,          // Типы параметров
                 null, null);

mb.DefineParameter (1, ParameterAttributes.None, "value"); // Необязательно
ILGenerator gen = mb.GetILGenerator();
gen.Emit (OpCodes.Ldarg_0);
gen.Emit (OpCodes.Ret);

```

Метод `DefineGenericParameters` принимает любое количество строковых аргументов — они соответствуют именам желаемых обобщенных типов. В данном примере необходим только один обобщенный тип по имени `T`. Класс `GenericTypeParameterBuilder` основан на `System.Type`, поэтому он может использоваться на месте `TypeBuilder` при выпуске кодов операций.

Класс `GenericTypeParameterBuilder` также позволяет указывать ограничение базового типа:

```
genericParams[0].SetBaseTypeConstraint (typeof (Foo));
```

и ограничения интерфейсов:

```
genericParams[0].SetInterfaceConstraints (typeof (IComparable));
```

Чтобы воспроизвести приведенный ниже код:

```
public static T Echo<T> (T value) where T : IComparable<T>
```

потребуется записать так:

```
genericParams[0].SetInterfaceConstraints (
    typeof (IComparable<>).MakeGenericType (genericParams[0]));
```

Для других видов ограничений нужно вызывать метод `SetGenericParameterAttributes`. Он принимает член перечисления `GenericParameterAttributes`, которое содержит следующие значения:

- `DefaultConstructorConstraint`
- `NotNullableValueTypeConstraint`
- `ReferenceTypeConstraint`
- `Covariant`
- `Contravariant`

Последние два значения эквивалентны применению к параметрам типа модификаторов `out` и `in`.

Определение обобщенных типов

Обобщенные типы определяются в похожей манере. Отличие заключается в том, что метод `DefineGenericParameters` вызывается на объекте `TypeBuilder`, а не `MethodBuilder`. Таким образом, для воспроизведения следующего определения:

```
public class Widget<T>
{
    public T Value;
}
```

потребуется написать такой код:

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
GenericTypeParameterBuilder[] genericParams
    = tb.DefineGenericParameters ("T");
tb.DefineField ("Value", genericParams[0], FieldAttributes.Public);
```

Как и в случае методов, можно добавлять обобщенные ограничения.

Сложности, связанные с генерацией

Во всех примерах данного раздела предполагается, что объект `modBuilder` создавался, как было показано в предшествующих разделах.

Несозданные закрытые обобщения

Пусть необходимо сгенерировать метод, который использует закрытый обобщенный тип:

```
public class Widget
{
    public static void Test() { var list = new List<int>(); }
```

Процесс довольно прямолинеен:

```
TypeBuilder tb = modBuilder.DefineType ("Widget", TypeAttributes.Public);
MethodBuilder mb = tb.DefineMethod ("Test", MethodAttributes.Public |
    MethodAttributes.Static);
ILGenerator gen = mb.GetILGenerator();
Type variableType = typeof (List<int>);
ConstructorInfo ci = variableType.GetConstructor (new Type[0]);
LocalBuilder listVar = gen.DeclareLocal (variableType);
gen.Emit (OpCodes.Newobj, ci);
gen.Emit (OpCodes.Stloc, listVar);
gen.Emit (OpCodes.Ret);
```

Теперь предположим, что вместо списка целых чисел требуется список объектов `Widget`:

```
public class Widget
{
    public static void Test() { var list = new List<Widget>(); }
```

Теоретически модификация проста — нужно лишь заменить строку:

```
Type variableType = typeof (List<int>);
```

строкой:

```
Type variableType = typeof (List<>).MakeGenericType (tb);
```

К сожалению, при последующем вызове метода `GetConstructor` генерируется исключение `NotSupportedException`. Проблема в том, что вызывать `GetConstructor` на обобщенном типе, закрытом с помощью несозданного построителя типа, не допускается. То же самое касается методов `GetField` и `GetMethod`.

Решение нельзя считать интуитивно понятным. В классе `TypeBuilder` присутствуют три статических метода:

```
public static ConstructorInfo GetConstructor (Type, ConstructorInfo);
public static FieldInfo GetField (Type, FieldInfo);
public static MethodInfo GetMethod (Type, MethodInfo);
```

Хотя они таковыми не выглядят, методы предназначены специально для получения членов обобщенных типов, закрытых посредством несозданных построителей типов! Первый параметр представляет собой закрытый обобщенный тип, а второй параметр — желаемый член из *несвязанного* обобщенного типа. Ниже приведена скорректированная версия рассматриваемого примера:

```
MethodBuilder mb = tb.DefineMethod ("Test", MethodAttributes.Public |
    MethodAttributes.Static);
ILGenerator gen = mb.GetILGenerator();
Type variableType = typeof (List<>).MakeGenericType (tb);
ConstructorInfo unbound = typeof (List<>).GetConstructor (new Type[0]);
ConstructorInfo ci = TypeBuilder.GetConstructor (variableType, unbound);
LocalBuilder listVar = gen.DeclareLocal (variableType);
gen.Emit (OpCodes.Newobj, ci);
gen.Emit (OpCodes.Stloc, listVar);
gen.Emit (OpCodes.Ret);
```

Циклические зависимости

Предположим, что необходимо построить два типа, которые ссылаются друг на друга, например:

```
class A { public B Bee; }
class B { public A Aye; }
```

Сгенерировать это динамически можно следующим образом:

```
var publicAtt = FieldAttributes.Public;
TypeBuilder aBuilder = modBuilder.DefineType ("A");
TypeBuilder bBuilder = modBuilder.DefineType ("B");

FieldBuilder bee = aBuilder.DefineField ("Bee", bBuilder, publicAtt);
FieldBuilder aye = bBuilder.DefineField ("Aye", aBuilder, publicAtt);

Type realA = aBuilder.CreateType();
Type realB = bBuilder.CreateType();
```

Обратите внимание, что мы не вызывали метод CreateType на объекте aBuilder или bBuilder, пока оба объекта не были заполнены. Здесь применяется следующий принцип: сначала все связывается, а затем производится вызов метода CreateType на каждом построителе типа.

Интересно отметить, что тип realA является допустимым, но *дисфункциональным* до тех пор, пока не будет вызван метод CreateType на bBuilder. (Если вы начнете использовать объект aBuilder до такого момента, то при попытке доступа к полю Bee генерируется исключение.)

Вас может заинтересовать, каким образом bBuilder узнает о необходимости “исправления” типа realA после создания realB? На самом деле он вовсе не знает об этом: тип realA может исправить себя *самостоятельно* при следующем его применении. Исправление возможно из-за того, что после вызова метода CreateType объект TypeBuilder превращается в посредника для действительного типа времени выполнения. Таким образом, благодаря своим ссылкам на bBuilder тип realA может легко получить метаданные, требующиеся для обновления.

Описанная система работает, когда построитель типа запрашивает простую информацию о несозданном типе — информацию, которая может быть *предварительно определена* — такую как тип, член и объектные ссылки. При создании realA построителю типа не нужно знать, скажем, сколько байтов памяти будет в итоге занимать realB. И это вполне нормально, т.к. тип realB пока еще не создан! Но теперь представьте, что тип realB был структурой. Окончательный размер realB теперь становится критически важной информацией при создании типа realA.

Если отношение между типами не является циклическим, например:

```
struct A { public B Bee; }  
struct B { }
```

то задачу можно решить, сначала создав структуру B, а затем структуру A. Но взгляните на следующие определения:

```
struct A { public B Bee; }  
struct B { public A Aye; }
```

Мы даже не будем пытаться выпустить такой код, поскольку определение двух структур, содержащих друг друга, лишено смысла (компилятор C# генерирует ошибку на этапе компиляции). Но показанная далее вариация как законна, так и полезна:

```
public struct S<T> { ... }           // Структура S может быть пустой  
                                         // и эта демонстрация будет работать.  
  
class A { S<B> Bee; }  
class B { S<A> Aye; }
```

При создании класса A построитель типа теперь должен располагать знанием отпечатка памяти класса B и наоборот. В целях иллюстрации предположим, что структура S определена статически. Код для выпуска классов A и B мог бы выглядеть так:

```

var pub = FieldAttributes.Public;
TypeBuilder aBuilder = modBuilder.DefineType ("A");
TypeBuilder bBuilder = modBuilder.DefineType ("B");
aBuilder.DefineField ("Bee", typeof(S<>).MakeGenericType (bBuilder), pub);
bBuilder.DefineField ("Aye", typeof(S<>).MakeGenericType (aBuilder), pub);
Type realA = aBuilder.CreateType(); // Ошибка: не удается загрузить тип B
Type realB = bBuilder.CreateType();

```

Метод `CreateType` теперь генерирует исключение `TypeLoadException` независимо от порядка выполнения:

- если первым идет вызов `aBuilder.CreateType`, то исключение сообщает о невозможности загрузки типа B;
- если первым идет вызов `bBuilder.CreateType`, то исключение сообщает о невозможности загрузки типа A.

Чтобы решить проблему, вы должны позволить построителю типа создать `realB` частично через создание `realA`. Это делается за счет обработки события `TypeResolve` в классе `AppDomain` непосредственно перед вызовом метода `CreateType`. Таким образом, в рассматриваемом примере мы заменяем последние две строки следующим кодом:

```

TypeBuilder[] uncreatedTypes = { aBuilder, bBuilder };
ResolveEventHandler handler = delegate (object o, ResolveEventArgs args)
{
    var type = uncreatedTypes.FirstOrDefault (t => t.FullName == args.Name);
    return type == null ? null : type.CreateType().Assembly;
};
AppDomain.CurrentDomain.TypeResolve += handler;
Type realA = aBuilder.CreateType();
Type realB = bBuilder.CreateType();
AppDomain.CurrentDomain.TypeResolve -= handler;

```

Событие `TypeResolve` инициируется во время вызова метода `aBuilder.CreateType`, в точке, где нужно, чтобы вы вызвали `CreateType` на `bBuilder`.



Обработка события `TypeResolve`, как в представленном примере, также необходима при определении вложенного типа, когда вложенный и родительский типы ссылаются друг на друга.

Синтаксический разбор IL

Для получения информации о содержимом существующего метода понадобится вызвать метод `GetMethodBody` на объекте `MethodBase`. Вызов возвращает объект `MethodBody`, который имеет свойства для инспектирования локальных переменных метода, конструкций обработки исключений, размера стека и низкоуровневого кода IL. Очень похоже на противоположность метода `Reflection.Emit!`

Инспектирование низкоуровневого кода IL метода может быть полезно при профилировании кода. Простой сценарий использования мог бы предусматривать выяснение, какие методы в сборке изменились в результате ее обновления.

Для демонстрации синтаксического разбора IL мы напишем приложение, которое дизассемблирует код IL, работая в стиле ildasm. Приложение подобного рода могло бы служить отправной точкой для построения инструмента анализа кода или дизассемблера языка более высокого уровня.



Вспомните, что в API-интерфейсе рефлексии все функциональные конструкции C# либо представлены подтипом MethodBase, либо (в случае свойств, событий и индексаторов) имеют присоединенные к ним объекты MethodBase.

Написание дизассемблера

Ниже приведен пример вывода, который будет производить наш дизассемблер:

```
IL_00EB: ldfld      Disassembler._pos
IL_00F0: ldloc.2
IL_00F1: add
IL_00F2: ldelema    System.Byte
IL_00F7: ldstr      "Hello world"
IL_00FC: call       System.Byte.ToString
IL_0101: ldstr      "
IL_0106: call       System.String.Concat
```

Чтобы получить такой вывод, потребуется провести синтаксический разбор двоичных лексем, из которых сформирован код IL. Первый шаг заключается в вызове метода GetILAsByteArray на объекте MethodBody для получения кода IL в виде байтового массива. Для упрощения оставшейся работы мы реализуем решение такой задачи в форме класса:

```
public class Disassembler
{
    public static string Disassemble (MethodBase method)
        => new Disassembler (method).Dis();
    StringBuilder _output; // Результат, который будет постоянно дополняться
    Module _module;       // Это пригодится в дальнейшем
    byte[] _il;           // Низкоуровневый байтовый код
    int _pos;              // Позиция внутри байтового кода
    Disassembler (MethodBase method)
    {
        _module = method.DeclaringType.Module;
        _il = method.GetMethodBody ().GetILAsByteArray ();
    }
    string Dis()
    {
        _output = new StringBuilder ();
        while (_pos < _il.Length) DisassembleNextInstruction ();
        return _output.ToString ();
    }
}
```

Статический метод Disassemble будет единственным открытым членом в данном классе. Все другие члены будут закрытыми по отношению к процессу дисассемблирования. Метод Dis содержит “главный” цикл, в котором мы обрабатываем каждую инструкцию.

Имея такой скелет, остается лишь написать метод DisassembleNextInstruction. Но перед тем как делать это, полезно загрузить все коды операций в статический словарь, чтобы к ним можно было обращаться по их 8- или 16-битным значениям. Простейший способ достичь такой цели — воспользоваться рефлексией для извлечения всех статических полей типа OpCode из класса OpCodes:

```
static Dictionary<short, OpCode> _opcodes = new Dictionary<short, OpCode>();
static Disassembler()
{
    Dictionary<short, OpCode> opcodes = new Dictionary<short, OpCode>();
    foreach (FieldInfo fi in typeof(OpCodes).GetFields(
        BindingFlags.Public | BindingFlags.Static))
        if (typeof(OpCode).IsAssignableFrom (fi.FieldType))
    {
        OpCode code = (OpCode) fi.GetValue (null); // Получить значение поля
        if (code.OpCodeType != OpCodeType.Nternal)
            _opcodes.Add (code.Value, code);
    }
}
```

Мы поместили код в статический конструктор, так что он будет выполняться только один раз.

Теперь можно заняться реализацией метода DisassembleNextInstruction. Каждая инструкция IL состоит из однобайтового или двухбайтового кода операции, за которым следует operand длиной 0, 1, 2, 4 или 8 байтов. (Исключением являются коды операций встроенных переключателей, за которыми следует переменное количество operandов.) Итак, мы читаем код операции, далее operand и затем выводим результат:

```
void DisassembleNextInstruction()
{
    int opStart = _pos;
    OpCode code = ReadOpCode();
    string operand = ReadOperand (code);
    _output.AppendFormat ("IL_{0:X4}: {1,-12} {2}",
        opStart, code.Name, operand);
    _output.AppendLine();
}
```

Для чтения кода операции мы продвигаемся вперед на один байт и выясняем, является ли он допустимой инструкцией. Если нет, тогда мы продвигаемся вперед еще на один байт и проверяем, существует ли двухбайтовая инструкция:

```
OpCode ReadOpCode ()
{
    byte byteCode = _il [_pos++];
    if (_opcodes.ContainsKey (byteCode)) return _opcodes [byteCode];
```

```

if (_pos == _il.Length) throw new Exception ("Unexpected end of IL");
                                // Неожиданный конец кода IL

short shortCode = (short) (byteCode * 256 + _il [_pos++]);

if (!_opcodes.ContainsKey (shortCode))
    throw new Exception ("Cannot find opcode " + shortCode);

return _opcodes [shortCode];
}

```

Чтобы прочитать операнд, сначала потребуется выяснить его длину. Это можно сделать на основе типа операнда. Поскольку большинство из них имеют 4 байта в длину, отклонения можно довольно легко отфильтровать в условной конструкции.

Следующий шаг заключается в вызове метода `FormatOperand`, который попытается сформатировать операнд:

```

string ReadOperand (OpCode c)
{
    int operandLength =
        c.OperandType == OperandType.InlineNone
            ? 0 :
        c.OperandType == OperandType.ShortInlineBrTarget ||
        c.OperandType == OperandType.ShortInlineI ||
        c.OperandType == OperandType.ShortInlineVar
            ? 1 :
        c.OperandType == OperandType.InlineVar
            ? 2 :
        c.OperandType == OperandType.InlineI8 ||
        c.OperandType == OperandType.InlineR
            ? 8 :
        c.OperandType == OperandType.InlineSwitch
            ? 4 * (BitConverter.ToInt32 (_il, _pos) + 1) :
            4; // Все остальные имеют длину 4 байта

    if (_pos + operandLength > _il.Length)
        throw new Exception ("Unexpected end of IL");
                                // Неожиданный конец кода IL

    string result = FormatOperand (c, operandLength);
    if (result == null)
    { // Вывести байты операнда в шестнадцатеричном виде
        result = "";
        for (int i = 0; i < operandLength; i++)
            result += _il [_pos + i].ToString ("X2") + " ";
    }
    _pos += operandLength;
    return result;
}

```

Если после вызова метода `FormatOperand` значение `result` равно `null`, то это означает, что операнд не нуждается в специальном форматировании, и мы просто выводим его в шестнадцатеричном виде. Мы могли бы протестировать дизассемблер в данной точке, написав метод `FormatOperand`, который всегда возвращает `null`. Ниже показано, как будет выглядеть вывод:

```
IL_00A8: ldfld      98 00 00 04
IL_00AD: ldloc.2
IL_00AE: add
IL_00AF: ldelema    64 00 00 01
IL_00B4: ldstr      26 04 00 70
IL_00B9: call        B6 00 00 0A
IL_00BE: ldstr      11 01 00 70
IL_00C3: call        91 00 00 0A
...

```

Хотя коды операций корректны, операнды в таком виде не особенно полезны. Вместо шестнадцатеричных цифр нам необходимы имена членов и строки. После реализации метод FormatOperand решит проблему, идентифицируя специальные случаи, которые выигрывают от такого форматирования. Они включают большинство 4-байтовых операндов и сокращенные инструкции ветвления:

```
string FormatOperand (OpCode c, int operandLength)
{
    if (operandLength == 0) return "";
    if (operandLength == 4)
        return Get4ByteOperand (c);
    else if (c.OperandType == OperandType.ShortInlineBrTarget)
        return GetShortRelativeTarget();
    else if (c.OperandType == OperandType.InlineSwitch)
        return GetSwitchTarget (operandLength);
    else
        return null;
}
```

Есть три вида 4-байтовых операндов, которые мы трактуем специальным образом. Первый вид относится к членам или типам — в данном случае мы извлекаем имя члена или типа, вызывая метод ResolveMember определяющего модуля. Второй вид — строки; они хранятся в метаданных модуля сборки и могут быть извлечены вызовом метода ResolveString. Третий вид касается целей ветвления, когда операнды ссылаются на байтовое смещение в коде IL. Мы форматируем их за счет работы с абсолютным адресом *после* текущей инструкции (+ 4 байта):

```
string Get4ByteOperand (OpCode c)
{
    int intOp = BitConverter.ToInt32 (_il, _pos);
    switch (c.OperandType)
    {
        case OperandType.InlineTok:
        case OperandType.InlineMethod:
        case OperandType.InlineField:
        case OperandType.InlineType:
            MemberInfo mi;
            try { mi = _module.ResolveMember (intOp); }
            catch { return null; }
            if (mi == null) return null;
    }
}
```

```

        if (mi.ReflectedType != null)
            return mi.ReflectedType.FullName + "." + mi.Name;
        else if (mi is Type)
            return ((Type)mi).FullName;
        else
            return mi.Name;
    case OperandType.InlineString:
        string s = _module.ResolveString (intOp);
        if (s != null) s = "" + s + "";
        return s;
    case OperandType.InlineBrTarget:
        return "IL_" + (_pos + intOp + 4).ToString ("X4");
    default:
        return null;
    }
}

```



Точка, где мы вызываем `ResolveMember`, представляет собой хорошее окно для инструмента анализа кода, который сообщает о зависимостях методов.

Для любого другого 4-байтового кода операции мы возвращаем `null` (что заставляет метод `ReadOperand` форматировать operand в виде шестнадцатеричных цифр).

Последняя разновидность operandов, которая требует особого внимания — сокращенные цели ветвления и встроенные переключатели. Сокращенная цель ветвления описывает смещение назначения в виде одиночного байта со знаком, как в конце текущей инструкции (т.е. + 1 байт). За целью переключателя следует переменное количество 4-байтовых целей ветвления:

```

string GetShortRelativeTarget ()
{
    int absoluteTarget = _pos + (sbyte) _il [_pos] + 1;
    return "IL_" + absoluteTarget.ToString ("X4");
}
string GetSwitchTarget (int operandLength)
{
    int targetCount = BitConverter.ToInt32 (_il, _pos);
    string [] targets = new string [targetCount];
    for (int i = 0; i < targetCount; i++)
    {
        int ilTarget = BitConverter.ToInt32 (_il, _pos + (i + 1) * 4);
        targets [i] = "IL_" + (_pos + ilTarget + operandLength).ToString ("X4");
    }
    return "(" + string.Join (", ", targets) + ")";
}

```

На этом написание дизассемблера завершено. Чтобы протестировать класс `Disassembler`, можно дизассемблировать один из его собственных методов:

```

MethodInfo mi = typeof (Disassembler).GetMethod (
    "ReadOperand", BindingFlags.Instance | BindingFlags.NonPublic);
Console.WriteLine (Disassembler.Disassemble (mi));

```



Динамическое программирование

В главе 4 объяснялась работа динамического связывания в языке C#. В этой главе мы кратко рассмотрим исполняющую среду динамического языка (Dynamic Language Runtime — DLR), после чего раскроем следующие паттерны динамического программирования:

- динамическое распознавание перегруженных членов;
- специальное связывание (реализация динамических объектов);
- взаимодействие с динамическими языками.



В главе 24 будет показано, каким образом ключевое слово `dynamic` может улучшить взаимодействие с COM.

Типы, рассматриваемые в главе, находятся в пространстве имен `System.Dynamic` за исключением типа `CallSite<>`, который расположен в пространстве имен `System.Runtime.CompilerServices`.

Исполняющая среда динамического языка

При выполнении динамического связывания язык C# полагается на среду DLR. Несмотря на свое название, DLR не является динамической версией среды CLR. В действительности она представляет собой библиотеку, которая функционирует поверх CLR — точно так же, как любая другая библиотека вроде `System.Xml.dll`. Ее основная роль — снабжать службами времени выполнения для унификации динамического программирования на статически и динамически типизированных языках. Следовательно, такие языки, как C#, VB, IronPython и IronRuby, используют один и тот же протокол для вызова функций динамическим образом. Это позволяет им совместно использовать библиотеки и обращаться к коду, написанному на других языках.

Среда DLR также позволяет сравнительно легко создавать новые динамические языки в .NET. Вместо выпуска кода IL авторы динамических языков работают на уровне деревьев выражений (тех самых деревьев выражений из пространства имен System.Linq.Expressions, которые обсуждались в главе 8).

Среда DLR дополнительно гарантирует, что все потребители получают преимущество кеширования места вызова, представляющего собой оптимизацию, в соответствии с которой DLR избегает излишнего повторения потенциально затратных действий по распознаванию членов, предпринимаемых во время динамического связывания.

Что такое место вызова?

Когда компилятор встречает динамическое выражение, он не имеет никакого представления о том, кто или что будет вычислять это выражение во время выполнения. Например, рассмотрим следующий метод:

```
public dynamic Foo (dynamic x, dynamic y)
{
    return x / y;      // Динамическое выражение
}
```

Переменные *x* и *y* могут быть любыми объектами CLR, объектами COM или даже объектами, размещенными в среде какого-то динамического языка. Таким образом, компилятор не в состоянии применить обычный статический подход с выпуском вызова известного метода из известного типа. Взамен компилятор выпускает код, который в итоге дает дерево выражения. Такое дерево выражения описывает операцию, управляемую местом вызова (call site), к которому среда DLR привязывается во время выполнения. По существу место вызова действует как посредник между вызывающим и вызываемым компонентами.

Место вызова представлено классом CallSite<> из сборки System.Core.dll. В этом можно убедиться, дизассемблировав предыдущий метод; результат будет выглядеть приблизительно так:

```
static CallSite<Func<CallSite,object,object,object>> divideSite;
[return: Dynamic]
public object Foo ([Dynamic] object x, [Dynamic] object y)
{
    if (divideSite == null)
        divideSite =
            CallSite<Func<CallSite,object,object,object>>.Create (
                Microsoft.CSharp.RuntimeBinder.Binder.BinaryOperation (
                    CSharpBinderFlags.None,
                    ExpressionType.Divide,
                    /* Для краткости остальные аргументы не показаны */));
    return divideSite.Target (divideSite, x, y);
}
```

Как видите, место вызова кешируется в статическом поле, чтобы избежать накладных расходов, обусловленных его повторным созданием в каждом вызове. Среда DLR дополнительно кеширует результат фазы привязки и

фактические целевые объекты метода. (В зависимости от типов *x* и *y* может существовать множество целевых объектов.)

Затем происходит действительный динамический вызов за счет обращения к полю Target (делегат) места вызова с передачей ему операндов *x* и *y*.

Обратите внимание, что класс Binder специфичен для C#. Каждый язык с поддержкой динамического связывания предоставляет специфичный для языка связыватель, помогающий среде DLR интерпретировать выражения в манере, которая присуща языку и не является неожиданной для программиста. Например, если мы вызываем метод Foo с целочисленными значениями 5 и 2, то связыватель C# обеспечит получение обратно значения 2. В противоположность этому связыватель VB.NET приведет к возвращению значения 2.5.

Динамическое распознавание перегруженных членов

Вызов статически известных методов с динамически типизированными аргументами откладывает распознавание перегруженных членов с этапа компиляции до времени выполнения. Такой подход содействует упрощению решения определенных задач программирования вроде реализации паттерна проектирования “Посетитель” (Visitor). Кроме того, он удобен для обхода ограничений, накладываемых статической типизацией языка C#.

Упрощение паттерна “Посетитель”

По существу паттерн “Посетитель” позволяет “добавлять” метод в иерархию классов, не изменяя существующие классы. Несмотря на полезность, данный паттерн в своем статическом воплощении является неочевидным и не интуитивно понятным по сравнению с большинством других паттернов проектирования. Он также требует, чтобы посещаемые классы были сделаны “дружественными к паттерну ‘Посетитель’” за счет открытия доступа к методу Accept, что может оказаться невозможным, если классы находятся вне вашего контроля.

Посредством динамического связывания той же самой цели можно достигнуть более просто — и без необходимости в модификации существующих классов. В качестве иллюстрации рассмотрим следующую иерархию классов:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    // Коллекция Friends может содержать объекты Customer и Employee:
    public readonly IList<Person> Friends = new Collection<Person> ();
}

class Customer : Person { public decimal CreditLimit { get; set; } }
class Employee : Person { public decimal Salary { get; set; } }
```

Предположим, что требуется написать метод, который программно экспортирует детали объекта Person в XML-элемент (объект XElement). Наиболее очевидное решение предусматривает реализацию внутри класса Person виртуального метода по имени To XElement, который возвращает объект XElement, заполненный значениями свойств объекта Person. Затем метод To XElement в классах Customer и Employee можно было бы переопределить, чтобы объект XElement также заполнялся значениями свойств CreditLimit и Salary. Однако такой паттерн может оказаться трудным в реализации по двум причинам.

- Вы можете не владеть кодом классов Person, Customer и Employee, что делает невозможным добавление к ним методов. (А расширяющие методы не обеспечивают полиморфное поведение.)
- Классы Person, Customer и Employee могут уже быть довольно большими. Часто встречающимся антипаттерном является “Божественный объект” (“God Object”), при котором класс, подобный Person, возлагает на себя настолько много функциональности, что его сопровождение превращается в самый настоящий кошмар. Хорошее противодействие этому — избегание добавления в класс Person функций, которым не нужен доступ к закрытому состоянию Person. Великолепным кандидатом может служить метод To XElement.

Благодаря динамическому распознаванию перегруженных членов мы можем реализовать функциональность метода To XElement в отдельном классе, не прибегая к неуклюжим операторам switch на основе типа:

```
class To XElementPersonVisitor
{
    public XElement DynamicVisit (Person p) => Visit ((dynamic)p);

    XElement Visit (Person p)
    {
        return new XElement ("Person",
            new XAttribute ("Type", p.GetType().Name),
            new XElement ("FirstName", p.FirstName),
            new XElement ("LastName", p.LastName),
            p.Friends.Select (f => DynamicVisit (f))
        );
    }

    XElement Visit (Customer c) // Специализированная логика для объектов Customer
    {
        XElement xe = Visit ((Person)c); // Вызов "базового" метода
        xe.Add (new XElement ("CreditLimit", c.CreditLimit));
        return xe;
    }

    XElement Visit (Employee e) // Специализированная логика для объектов Employee
    {
        XElement xe = Visit ((Person)e); // Вызов "базового" метода
        xe.Add (new XElement ("Salary", e.Salary));
        return xe;
    }
}
```

Метод `DynamicVisit` осуществляет динамическую диспетчеризацию — вызывает наиболее специфическую версию метода `Visit`, как определено во время выполнения. Обратите внимание на выделенную полужирным строку кода, в которой мы вызываем `DynamicVisit` на каждом объекте `Person` в коллекции `Friends`. Такой прием гарантирует, что если элемент коллекции `Friends` является объектом `Customer` или `Employee`, то будет вызвана корректная перегруженная версия метода.

Продемонстрировать использование класса `ToXMLElementPersonVisitor` можно следующим образом:

```
var cust = new Customer
{
    FirstName = "Joe", LastName = "Bloggs", CreditLimit = 123
};
cust.Friends.Add (
    new Employee { FirstName = "Sue", LastName = "Brown", Salary = 50000 }
);
Console.WriteLine (new ToXMLElementPersonVisitor().DynamicVisit (cust));
```

Вот как выглядит результат:

```
<Person Type="Customer">
    <FirstName>Joe</FirstName>
    <LastName>Bloggs</LastName>
    <Person Type="Employee">
        <FirstName>Sue</FirstName>
        <LastName>Brown</LastName>
        <Salary>50000</Salary>
    </Person>
    <CreditLimit>123</CreditLimit>
</Person>
```

Вариации

Если планируется работа с несколькими классами посетителя, тогда удобная вариация предусматривает определение абстрактного базового класса для посетителей:

```
abstract class PersonVisitor<T>
{
    public T DynamicVisit (Person p) { return Visit ((dynamic)p); }

    protected abstract T Visit (Person p);
    protected virtual T Visit (Customer c) { return Visit ((Person) c); }
    protected virtual T Visit (Employee e) { return Visit ((Person) e); }
}
```

Тогда в подклассах не придется определять собственный метод `DynamicVisit`: они будут лишь переопределять версии метода `Visit`, поведение которых должно быть специализировано. Вдобавок появляются преимущества централизации методов, охватывающих иерархию `Person`, и возможность у реализующих классов вызывать базовые методы более естественным образом:

```

class To XElementPersonVisitor : PersonVisitor< XElement>
{
    protected override XElement Visit (Person p)
    {
        return new XElement ("Person",
            new XAttribute ("Type", p.GetType ().Name),
            new XElement ("FirstName", p.FirstName),
            new XElement ("LastName", p.LastName),
            p.Friends.Select (f => DynamicVisit (f))
        );
    }

    protected override XElement Visit (Customer c)
    {
        XElement xe = base.Visit (c);
        xe.Add (new XElement ("CreditLimit", c.CreditLimit));
        return xe;
    }

    protected override XElement Visit (Employee e)
    {
        XElement xe = base.Visit (e);
        xe.Add (new XElement ("Salary", e.Salary));
        return xe;
    }
}

```

В дальнейшем можно даже создавать подклассы самого класса `To XElementPersonVisitor`.

Анонимный вызов членов обобщенного типа

Строгость статической типизации C# — палка о двух концах. С одной стороны, она обеспечивает определенную степень корректности на этапе компиляции. С другой стороны, иногда она делает некоторые виды кода трудными в представлении или вовсе невозможными, и тогда приходится прибегать к рефлексии. В таких ситуациях динамическое связывание является более чистой и быстрой альтернативой рефлексии.

Примером может служить необходимость работы с объектом `G<T>`, где тип `T` неизвестен. Проиллюстрировать сказанное можно, определив следующий класс:

```
public class Foo<T> { public T Value; }
```

Предположим, что затем мы записываем метод, как показано ниже:

```
static void Write (object obj)
{
    if (obj is Foo<>)                                // Недопустимо
        Console.WriteLine ((Foo<>) obj).Value;          // Недопустимо
}
```

Такой код не скомпилируется: члены *несвязанных* обобщенных типов вызывать нельзя.

Динамическое связывание предлагает два средства, с помощью которых можно обойти данную проблему. Первое из них — доступ к члену `Value` динамическим образом:

```
static void Write (dynamic obj)
{
    try { Console.WriteLine (obj.Value); }
    catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException) {...}
}
```

Множественная диспетчеризация

Язык C# и среда CLR всегда поддерживали ограниченную форму динамизма в виде вызовов виртуальных методов. Она отличается от динамического связывания C# тем, что для вызовов виртуальных методов компилятор должен фиксировать отдельный виртуальный член на этапе компиляции — основываясь на имени и сигнатуре вызываемого члена. Это означает, что справедливы приведенные ниже утверждения:

- выражение вызова должно полностью восприниматься компилятором (например, на этапе компиляции должно приниматься решение о том, чем является целевой член — полем или свойством);
- распознавание перегруженных членов должно осуществляться полностью компилятором на основе типов аргументов в течение этапа компиляции.

Следствие последнего утверждения заключается в том, что возможность выполнения вызовов виртуальных методов известна как *одиночная диспетчеризация*. Чтобы понять причину, взгляните на приведенный ниже вызов метода (где `Walk` — виртуальный метод):

```
animal.Walk (owner);
```

Принятие во время выполнения решения о том, какой метод `Walk` вызывать — класса `Dog` (собака) или класса `Cat` (кошка) — зависит только от типа *получателя*, т.е. `animal` (животное); отсюда и “одиночная” диспетчеризация. Если многочисленные перегруженные версии `Walk` принимают разные типы `owner`, тогда перегруженная версия выбирается на этапе компиляции безотносительно к тому, каким будет действительный тип объекта `owner` во время выполнения. Другими словами, только тип *получателя* во время выполнения может изменить то, какой метод будет вызван.

По контрасту динамический вызов откладывает распознавание перегруженных членов вплоть до времени выполнения:

```
animal.Walk ((dynamic) owner);
```

Окончательный выбор метода `Walk`, подлежащего вызову, теперь зависит и от `animal`, и от `owner` — это называется *множественной диспетчеризацией*, потому что в определении вызываемого метода `Walk` принимает участие не только тип получателя, но и типы аргументов времени выполнения.

Здесь имеется (потенциальное) преимущество работы с любым объектом, который определяет поле или свойство Value. Тем не менее, осталась еще пара проблем. Во-первых, перехват исключений в подобной манере несколько запутан и неэффективен (к тому же отсутствует возможность заранее узнать у DLR, будет ли эта операция успешной). Во-вторых, такой подход не будет работать, если Foo является интерфейсом (скажем, IFoo<T>) и удовлетворено одно из следующих условий:

- член Value не реализован явно;
- недоступен тип, который реализует интерфейс IFoo<T> (подробнее об этом речь пойдет позже).

Более удачное решение предусматривает написание перегруженного вспомогательного метода по имени GetFooValue и его вызов с применением *динамического распознавания перегруженных членов*:

```
static void Write (dynamic obj)
{
    object result = GetFooValue (obj);
    if (result != null) Console.WriteLine (result);
}
static T GetFooValue<T> (Foo<T> foo) { return foo.Value; }
static object GetFooValue (object foo) { return null; }
```

Обратите внимание, что мы перегрузили метод GetFooValue с целью приема параметра object, который действует в качестве запасного варианта для любого типа. Во время выполнения при вызове GetFooValue с динамическим аргументом динамический связыватель C# выберет наилучшую перегруженную версию. Если рассматриваемый объект не основан на Foo<T>, то вместо генерации исключения связыватель выберет перегруженную версию с параметром object.



Альтернатива предусматривает написание только первой перегруженной версии метода GetFooValue и последующий перехват исключения RuntimeBinderException. Преимущество такого подхода в том, что он различает случай, когда значение foo.Value равно null. Недостаток связан с появлением накладных расходов в плане производительности, которые обусловлены генерацией и перехватом исключения.

В главе 18 мы решали ту же самую проблему с интерфейсом, используя рефлексию — и прикладывали гораздо больше усилий (см. раздел “Анонимный вызов членов обобщенного интерфейса” в главе 18). Там рассматривался пример проектирования более мощной версии метода ToString, которая воспринимала бы объекты, реализующие IEnumerable и IGrouping<, >. Далее приведен тот же пример, решенный более элегантно за счет динамического связывания:

```
static string GetGroupKey< TKey, TElement > (IGrouping< TKey, TElement > group)
{
    return "Group with key=" + group.Key + ": ";
```

```

static string GetGroupKey (object source) { return null; }
public static string ToStringEx (object value)
{
    if (value == null) return "<null>";
    if (value is string s) return s;
    if (value.GetType().IsPrimitive) return value.ToString();
    StringBuilder sb = new StringBuilder();
    string groupKey = GetGroupKey ((dynamic)value); // Динамическая
                                                    // диспетчеризация
    if (groupKey != null) sb.Append (groupKey);
    if (value is IEnumerable)
        foreach (object element in ((IEnumerable)value))
            sb.Append (ToStringEx (element) + " ");
    if (sb.Length == 0) sb.Append (value.ToString());
    return "\r\n" + sb.ToString();
}

```

Ниже код демонстрируется в действии:

```
Console.WriteLine (ToStringEx ("xyyzzz".GroupBy (c => c)));
```

Вывод:

```

Group with key=x: x
Group with key=y: y y
Group with key=z: z z z

```

Обратите внимание, что для решения задачи мы применяли динамическое распознавание перегруженных членов. Если бы взамен мы поступили следующим образом:

```

dynamic d = value;
try { groupKey = d.Value; }
catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException) {...}

```

то код потерпел бы неудачу, потому что LINQ-операция GroupBy возвращает тип, реализующий интерфейс `IGrouping<,>`, который сам по себе является внутренним и по этой причине недоступным:

```

internal class Grouping : IGrouping<TKey,TElement>, ...
{
    public TKey Key;
    ...
}

```

Хотя свойство `Key` объявлено как `public`, содержащий его класс ограничивает данное свойство до `internal`, делая доступным только через интерфейс `IGrouping<,>`. И как объяснялось в главе 4, при динамическом обращении к члену `Value` нет никакого способа сообщить среде DLR о необходимости привязки к указанному интерфейсу.

Реализация динамических объектов

Объект может предоставить свою семантику привязки, реализуя интерфейс `IDynamicMetaObjectProvider` или более просто — создавая подкласс `DynamicObject`, который предлагает стандартную реализацию этого интерфейса. Мы кратко демонстрировали такой подход в главе 4 с помощью следующего примера:

```
dynamic d = new Duck();
d.Quack();           // Вызван метод Quack
d.Waddle();         // Вызван метод Waddle

public class Duck : DynamicObject
{
    public override bool TryInvokeMember (
        InvokeMemberBinder binder, object[] args, out object result)
    {
        Console.WriteLine (binder.Name + " method was called");
        result = null;
        return true;
    }
}
```

DynamicObject

В предыдущем примере мы переопределили метод `TryInvokeMember`, который позволяет потребителю вызывать на динамическом объекте метод вроде `Quack` или `Waddle`. Класс `DynamicObject` открывает дополнительные виртуальные методы, которые дают потребителям возможность использовать также и другие программные конструкции. Ниже перечислены конструкции, имеющие представления в языке C#.

Метод	Конструкция программирования
<code>TryInvokeMember</code>	Метод
<code>TryGetMember</code> , <code>TrySetMember</code>	Свойство или поле
<code>TryGetIndex</code> , <code>TrySetIndex</code>	Индексатор
<code>TryUnaryOperation</code>	Унарная операция, такая как !
<code>TryBinaryOperation</code>	Бинарная операция, такая как ==
<code>TryConvert</code>	Преобразование (приведение) в другой тип
<code>TryInvoke</code>	Вызов на самом объекте, например, <code>d("foo")</code>

При успешном выполнении эти методы должны возвращать `true`. Если они возвращают `false`, тогда среда DLR будет прибегать к услугам связывателя языка в поиске подходящего члена в самом (подклассе) `DynamicObject`. В случае неудачи генерируется исключение `RuntimeBinderException`.

Мы можем продемонстрировать работу `TryGetMember` и `TrySetMember` с классом, который позволяет динамически получать доступ к атрибуту в объекте `XElement` (`System.Xml.Linq`):

```

static class XExtensions
{
    public static dynamic DynamicAttributes (this XElement e)
        => new XWrapper (e);
    class XWrapper : DynamicObject
    {
        XElement _element;
        public XWrapper ( XElement e) { _element = e; }
        public override bool TryGetMember (GetMemberBinder binder,
                                         out object result)
        {
            result = _element.Attribute (binder.Name).Value;
            return true;
        }
        public override bool TrySetMember (SetMemberBinder binder,
                                         object value)
        {
            _element.SetAttributeValue (binder.Name, value);
            return true;
        }
    }
}

```

А так он применяется:

```

 XElement x = XElement.Parse (@"<Label Text=""Hello"" Id=""5""/>");
 dynamic da = x.DynamicAttributes();
 Console.WriteLine (da.Id);           // 5
 da.Text = "Foo";
 Console.WriteLine (x.ToString());     // <Label Text="Foo" Id="5" />

```

Следующий код выполняет аналогичное действие для интерфейса System.Data.IDataRecord, упрощая его использование средствами чтения данных:

```

public class DynamicReader : DynamicObject
{
    readonly IDataRecord _dataRecord;
    public DynamicReader (IDataRecord dr) { _dataRecord = dr; }
    public override bool TryGetMember (GetMemberBinder binder,
                                     out object result)
    {
        result = _dataRecord [binder.Name];
        return true;
    }
}
...
using (IDataReader reader = someDbCommand.ExecuteReader())
{
    dynamic dr = new DynamicReader (reader);
    while (reader.Read())
    {
        int id = dr.ID;
        string firstName = dr.FirstName;
        DateTime dob = dr.DateOfBirth;
        ...
    }
}

```

В приведенном ниже коде демонстрируется работа TryBinaryOperation и TryInvoke:

```
dynamic d = new Duck();
Console.WriteLine (d + d);                                // foo
Console.WriteLine (d (78, 'x'));                           // 123
public class Duck : DynamicObject
{
    public override bool TryBinaryOperation (BinaryOperationBinder binder,
                                             object arg, out object result)
    {
        Console.WriteLine (binder.Operation);                // Add
        result = "foo";
        return true;
    }
    public override bool TryInvoke (InvokeBinder binder,
                                   object[] args, out object result)
    {
        Console.WriteLine (args[0]);                          // 78
        result = 123;
        return true;
    }
}
```

Класс DynamicObject также открывает доступ к ряду виртуальных методов в интересах динамических языков. В частности, переопределение метода GetDynamicMemberNames позволяет возвращать список имен всех членов, которые предоставляет динамический объект.



Еще одна причина реализации GetDynamicMemberNames связана с тем, что отладчик Visual Studio задействует данный метод при отображении представления динамического объекта.

ExpandoObject

Другое простое практическое применение DynamicObject касается написания динамического класса, который хранит и извлекает объекты в словаре с ключами-строками. Тем не менее, эта функциональность уже предлагается классом ExpandoObject:

```
dynamic x = new ExpandoObject();
x.FavoriteColor = ConsoleColor.Green;
x.FavoriteNumber = 7;
Console.WriteLine (x.FavoriteColor);                      // Green
Console.WriteLine (x.FavoriteNumber);                     // 7
```

Класс ExpandoObject реализует интерфейс `IDictionary<string, object>` и потому мы можем продолжить наш пример, как показано ниже:

```
var dict = (IDictionary<string, object>) x;
Console.WriteLine (dict ["FavoriteColor"]);               // Green
Console.WriteLine (dict ["FavoriteNumber"]);              // 7
Console.WriteLine (dict.Count);                          // 2
```

Взаимодействие с динамическими языками

Хотя в языке C# поддерживается динамическое связывание через ключевое слово `dynamic`, оно не заходит настолько далеко, чтобы позволить вычислять выражение, описанное в строке, во время выполнения:

```
string expr = "2 * 3";
// "Выполнить" expr не удастся
```

Причина в том, что код для трансляции строки в дерево выражения требует лексического и семантического анализатора. Такие средства встроены в компилятор C# и не доступны в виде какой-то службы времени выполнения. Во время выполнения компилятор C# просто предоставляет *связыватель*, который сообщает среде DLR о том, как интерпретировать уже построенное дерево выражения.

Подлинные динамические языки, подобные IronPython и IronRuby, позволяют выполнять произвольную строку, что полезно при решении таких задач, как написание сценариев, динамическое конфигурирование и реализация процессоров динамических правил. Таким образом, хотя большую часть приложения можно написать на C#, для решения указанных задач удобно обращаться к какому-то динамическому языку. Кроме того, может возникнуть желание задействовать API-интерфейс, реализованный на динамическом языке, функциональность которого не имеет эквивалента в библиотеке .NET.



Пакет NuGet для написания сценариев Roslyn под названием `Microsoft.CodeAnalysis.CSharp.Scripting` предлагает API-интерфейс, который позволяет выполнять строку с кодом C#, хотя делает это, сначала компилируя код в программу. Накладные расходы на компиляцию делают его медленнее взаимодействия с Python, если только вы не намерены выполнять одно и то же выражение многократно.

В следующем примере мы используем язык IronPython, чтобы вычислить выражение, созданное во время выполнения, внутри кода C#. Данный сценарий мог бы применяться при реализации калькулятора.



Чтобы запустить этот код, добавьте в свое приложение NuGet-пакеты `DynamicLanguageRuntime` (не путайте его с пакетом `System.Dynamic.Runtime`) и `IronPython`.

```
using System;
using IronPython.Hosting;
using Microsoft.Scripting;
using Microsoft.Scripting.Hosting;

int result = (int) Calculate ("2 * 3");
Console.WriteLine (result); // 6
object Calculate (string expression)
{
    ScriptEngine engine = Python.CreateEngine();
    return engine.Execute (expression);
}
```

Поскольку мы передаем строку в Python, выражение будет вычисляться согласно правилам языка Python, а не C#. Это также означает возможность применения языковых средств Python, таких как списки:

```
var list = (IEnumerable) Calculate ("[1, 2, 3] + [4, 5]");
foreach (int n in list) Console.Write (n); // 12345
```

Передача состояния между C# и сценарием

Чтобы передать переменные из C# в Python, потребуется предпринять несколько дополнительных шагов, проиллюстрированных в следующем примере, который может служить основой для построения процессора правил:

```
// Следующая строка может поступать из файла или базы данных:
string auditRule = "taxPaidLastYear / taxPaidThisYear > 2";
ScriptEngine engine = Python.CreateEngine ();
ScriptScope scope = engine.CreateScope ();
scope.SetVariable ("taxPaidLastYear", 20000m);
scope.SetVariable ("taxPaidThisYear", 8000m);
ScriptSource source = engine.CreateScriptSourceFromString (
    auditRule, SourceCodeKind.Expression);
bool auditRequired = (bool) source.Execute (scope);
Console.WriteLine (auditRequired); // True
```

Вызвав метод `GetVariable`, переменные можно получить обратно:

```
string code = "result = input * 3";
ScriptEngine engine = Python.CreateEngine ();
ScriptScope scope = engine.CreateScope ();
scope.SetVariable ("input", 2);
ScriptSource source = engine.CreateScriptSourceFromString (code,
    SourceCodeKind.SingleStatement);
source.Execute (scope);
Console.WriteLine (engine.GetVariable (scope, "result")); // 6
```

Обратите внимание, что во втором примере мы указали значение `SourceCodeKind.SingleStatement` (а не `Expression`), чтобы сообщить процессору о необходимости выполнения оператора.

Типы автоматически маршализируются между мирами .NET и Python. Можно даже обращаться к членам объектов .NET со стороны сценария:

```
string code = @"sb.Append ("Hello")";
ScriptEngine engine = Python.CreateEngine ();
ScriptScope scope = engine.CreateScope ();
var sb = new StringBuilder ("Hello");
scope.SetVariable ("sb", sb);
ScriptSource source = engine.CreateScriptSourceFromString (
    code, SourceCodeKind.SingleStatement);
source.Execute (scope);
Console.WriteLine (sb.ToString()); // Hello
```



Криптография

В настоящей главе обсуждаются основные API-интерфейсы криптографии в .NET:

- защита данных Windows;
- хеширование;
- симметричное шифрование;
- шифрование с открытым ключом и подписание.

Типы, рассматриваемые в главе, определены в следующих пространствах имен:

`System.Security;`
`System.Security.Cryptography;`

Обзор

В табл. 20.1 приведена сводка по вариантам криптографии в .NET. Мы кратко рассмотрим их в оставшихся разделах главы.

Пространство имен `System.Security.Cryptography.Xml` в .NET обеспечивает более специализированную поддержку для создания и проверки подписей, основанных на XML, а пространство имен `System.Security.Cryptography.X509Certificates` предлагает типы для работы с цифровыми сертификатами.

Защита данных Windows



Защита данных Windows доступна только в Windows, а попытка ее использования в средах других ОС приводит к генерации исключения `PlatformNotSupportedException`.

В разделе “Операции с файлами и каталогами” главы 15 было показано, как использовать `File.Encrypt` для запрашивания у операционной системы прозрачного шифрования файла:

```
File.WriteAllText ("myfile.txt", "");  
File.Encrypt ("myfile.txt");  
File.AppendAllText ("myfile.txt", "sensitive data");
```

Таблица 20.1. Варианты шифрования и хеширования в .NET

Вариант	Количество ключей, подлежащих управлению	Скорость	Прочность	Примечания
File.Encrypt	0	Высокая	Зависит от пароля пользователя	Защищает файлы прозрачным образом при поддержке со стороны файловой системы. Ключ неявно выводится из учетных данных вошедшего в систему пользователя. Только Windows
Защита данных Windows	0	Высокая	Зависит от пароля пользователя	Шифрует и расшифровывает байтовые массивы, используя неявно выведенный ключ
Хеширование	0	Высокая	Высокая	Однонаправленная (необратимая) трансформация. Применяется для хранения паролей, сравнения файлов и проверки данных на предмет разрушения
Симметричное шифрование	1	Высокая	Высокая	Для универсального шифрования/расшифровки. При шифровании и расшифровке используется один и тот же ключ. Может применяться для защиты сообщений при транспортировке
Шифрование с открытым ключом	2	Низкая	Высокая	При шифровании и расшифровке используются разные ключи. Применяется для обмена симметричным ключом во время передачи сообщений и для цифрового подписания файлов

В данном случае шифрование применяет ключ, выведенный из пароля пользователя, который вошел в систему. Тот же самый неявно выведенный ключ можно использовать для шифрования байтового массива с помощью API-интерфейса защиты данных Windows (Data Protection API — DPAPI). Интерфейс DPAPI доступен через класс `ProtectedData` — простой тип с двумя статическими методами:

```
public static byte[] Protect (byte[] userData, byte[] optionalEntropy,
                           DataProtectionScope scope);
public static byte[] Unprotect (byte[] encryptedData, byte[] optionalEntropy,
                               DataProtectionScope scope);
```

Все, что вы включите в `optionalEntropy`, добавится к ключу, повышая тем самым его безопасность. Аргумент типа перечисления `DataProtectionScope` имеет два члена: `CurrentUser` и `LocalMachine`. В случае `CurrentUser` ключ выводится из учетных данных вошедшего в систему пользователя, а в случае

LocalMachine применяется ключ уровня машины, общий для всех пользователей. Это означает, что в области действия CurrentUser данные, зашифрованные одним пользователем, не могут быть расшифрованы другим. Ключ LocalMachine обеспечивает менее сильную защиту, но работает с Windows-службой или программой, которая должна функционировать под управлением множества учетных записей.

Ниже приведена простая демонстрация шифрования и расшифровки:

```
byte[] original = {1, 2, 3, 4, 5};  
DataProtectionScope scope = DataProtectionScope.CurrentUser;  
  
byte[] encrypted = ProtectedData.Protect (original, null, scope);  
byte[] decrypted = ProtectedData.Unprotect (encrypted, null, scope);  
// decrypted теперь содержит {1, 2, 3, 4, 5}
```

Защита данных Windows обеспечивает умеренное противодействие атакующему злоумышленнику, имеющему полный доступ к компьютеру, которое зависит от прочности пользовательского пароля. На уровне LocalMachine такая защита эффективна только против злоумышленников с ограниченным физическим и электронным доступом.

Хеширование

Алгоритм хеширования превращает потенциально крупное количество байтов в небольшой хеш-код фиксированной длины. Алгоритмы хеширования спроектированы так, что изменение одиночного бита где угодно в исходных данных приводит к получению существенно отличающегося хеш-кода. Данный факт делает их подходящими для сравнения файлов либо выявления случайной (либо умышленной) порчи файла или потока данных.

Хеширование также действует как одностороннее шифрование, потому что преобразовать хеш-код обратно в исходные данные практически невозможно. Оно идеально подходит для хранения паролей в базе данных, т.к. в случае взлома вашей базы данных вы не хотите, чтобы злоумышленник получил доступ к паролям в виде простого текста. Для аутентификации вы всего лишь хешируете то, что вводит пользователь, и сравниваете его с хеш-кодом, который хранится в базе данных.

Для выполнения хеширования вы вызываете метод ComputeHash на одном из подклассов HashAlgorithm, таком как SHA1 или SHA256:

```
byte[] hash;  
using (Stream fs = File.OpenRead ("checkme.doc"))  
    hash = SHA1.Create ().ComputeHash (fs); // Хеш SHA1 имеет длину 20 байтов
```

Метод ComputeHash также принимает байтовый массив, что удобно при хешировании паролей (более защищенный прием будет описан в разделе “Хеширование паролей” далее в главе):

```
byte[] data = System.Text.Encoding.UTF8.GetBytes ("strHong%pwd");  
byte[] hash = SHA256.Create ().ComputeHash (data);
```



Метод `GetBytes` объекта `Encoding` преобразует строку в байтовый массив; метод `GetString` осуществляет обратное преобразование. Тем не менее, объект `Encoding` не может преобразовывать зашифрованный или хешированный байтовый массив в строку, потому что такие данные обычно нарушают правила кодирования текста. Взамен придется использовать методы `Convert.ToString` и `Convert.FromString`: они выполняют преобразования между любым байтовым массивом и допустимой (к тому же дружественной к XML или JSON) строкой.

Алгоритмы хеширования в .NET

`SHA1` и `SHA256` — два подтипа `HashAlgorithm`, предоставляемые .NET. Ниже представлены основные алгоритмы, расположенные в порядке возрастания степени безопасности.

Класс	Алгоритм	Длина хеша в байтах	Надежность
MD5	MD5	16	Очень низкая
SHA1	SHA-1	20	Низкая
SHA256	SHA-2	32	Высокая
SHA384	SHA-2	48	Высокая
SHA512	SHA-2	64	Высокая

Все текущие реализации классов обеспечивают примерно одинаковую скорость работы за исключением `SHA256`, который в 2-3 раза быстрее (скорость может варьироваться в зависимости от оборудования и операционной системы). На современном настольном компьютере или сервере для всех алгоритмов можно ожидать производительность не менее 500 Мбайт в секунду. Более длинные хеши уменьшают вероятность возникновения коллизии (когда два отличающихся файла дают один и тот же хеш).



При хешировании паролей и других уязвимых данных применяйте, *по меньшей мере*, алгоритм `SHA256`. Алгоритмы `MD5` и `SHA1` для этих целей считаются ненадежными, и они подходят только для защиты от случайного повреждения данных, а не от их преднамеренной подделки.



В .NET 8 и более поздних версиях также поддерживается последняя версия алгоритма хеширования `SHA-3` через классы `SHA3_256`, `SHA3_384` и `SHA3_512`. Алгоритмы `SHA-3` считаются более безопасными (и медленными), чем перечисленные ранее алгоритмы, но требуют операционной системы Windows сборки 25324+ или Linux с OpenSSL 1.1.1+. Вы можете проверить, доступна ли поддержка со стороны операционной системы, с помощью статического свойства `IsSupported` указанных классов.

Хеширование паролей

Более длинные алгоритмы SHA подходят как основа для хеширования паролей, если вы обеспечите применение политики сильных паролей во избежание *словарной атаки* — стратегии, при которой злоумышленник строит таблицу поиска пароля путем хеширования каждого слова из словаря.

Стандартный прием при хешировании паролей предусматривает включение “начального значения” — длинной последовательности байтов, которую вы предварительно получаете через генератор случайных чисел и затем перед хешированием объединяете с каждым паролем. Такой подход срывает планы взломщиков двумя способами:

- они также должны знать байты начального значения;
- они не могут использовать *радужные таблицы* (rainbow table), которые представляют собой заранее рассчитанные базы данных паролей и их хеш-кодов, хотя словарная атака по-прежнему может быть возможной при наличии достаточной вычислительной мощности.

Вы можете дополнительно усилить защиту, “растягивая” хеши паролей, т.е. многократно производя повторное хеширование для получения байтовых последовательностей, которые требуют более интенсивных вычислений. Если выполнить повторное хеширование 100 раз, то словарная атака, которая иначе заняла бы месяц, может потребовать примерно восемь лет. Именно такой вид растяжения выполняют классы KeyDerivation, Rfc2898DeriveBytes и PasswordDeriveBytes, одновременно также позволяя удобно указывать начальные значения. Наилучшее хеширование предлагает метод KeyDerivation.Pbkdf2:

```
byte[] encrypted = KeyDerivation.Pbkdf2 (
    password: "stRhong&pword",
    salt: Encoding.UTF8.GetBytes ("j78Y#p") /saREN!y30"),
    prf: KeyDerivationPrf.HMACSHA512,
    iterationCount: 100,
    numBytesRequested: 64);
```



Метод KeyDerivation.Pbkdf2 требует NuGet-пакета Microsoft.AspNetCore.Cryptography.KeyDerivation. Несмотря на то что он находится в пространстве имен ASP.NET Core, его можно применять в любом приложении .NET.

Симметричное шифрование

При симметричном шифровании один и тот же ключ используется для шифрования и расшифровки. В .NET BCL предлагаются четыре алгоритма симметричного шифрования, главный из которых — алгоритм Рэндала (Rijndael); остальные алгоритмы предназначены по большей части для совместимости со старыми приложениями. Алгоритм Рэндала является быстрым и надежным и имеет две реализации:

- класс Rijndael;
- класс Aes.

Указанные два класса в основном идентичны за исключением того, что Aes не позволяет ослаблять шифр, изменяя размер блока. Класс Aes рекомендуется к применению командой разработчиков, которая отвечает за безопасность CLR.

Классы Rijndael и Aes допускают использование симметричных ключей длиной 16, 24 или 32 байта: все они считаются безопасными. Ниже показано, как зашифровать последовательности байтов при записи их в файл с применением 16-байтового ключа:

```
byte[] key = {145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50};  
byte[] iv = {15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7};  
byte[] data = { 1, 2, 3, 4, 5 }; // Данные, которые будут зашифрованы.  
  
using (SymmetricAlgorithm algorithm = Aes.Create())  
using (ICryptoTransform encryptor = algorithm.CreateEncryptor (key, iv))  
using (Stream f = File.Create ("encrypted.bin"))  
using (Stream c = new CryptoStream (f, encryptor, CryptoStreamMode.Write))  
c.Write (data, 0, data.Length);
```

Следующий код расшифровывает содержимое этого файла:

```
byte[] key = {145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50};  
byte[] iv = {15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7};  
byte[] decrypted = new byte[5];  
  
using (SymmetricAlgorithm algorithm = Aes.Create())  
using (ICryptoTransform decryptor = algorithm.CreateDecryptor (key, iv))  
using (Stream f = File.OpenRead ("encrypted.bin"))  
using (Stream c = new CryptoStream (f, decryptor, CryptoStreamMode.Read))  
for (int b; (b = c.ReadByte ()) > -1;)  
    Console.Write (b + " "); // 1 2 3 4 5
```

В приведенном примере мы формируем ключ из 16 случайно выбранных байтов. Если при расшифровке указан неправильный ключ, тогда CryptoStream генерирует исключение CryptographicException. Перехват данного исключения — единственный способ проверки корректности ключа.

Помимо ключа мы строим *вектор инициализации* (Initialization Vector — IV). Такая 16-байтоваая последовательность формирует часть шифра (почти как ключ), но не считается *секретной*. При передаче зашифрованного сообщения вектор IV можно отправлять в виде простого текста (скажем, в заголовке сообщения) и затем *изменять в каждом сообщении*. Это сделает каждое зашифрованное сообщение нераспознаваемым на основе любых предшествующих сообщений, даже если их незашифрованные версии были похожими либо идентичными.



Если защита посредством вектора IV не нужна или нежелательна, то ее можно аннулировать, используя одно и то же 16-байтовое значение для ключа и вектора IV. Тем не менее, отправка множества сообщений с одинаковым вектором IV ослабляет шифр и даже делает возможным его взлом.

Работа, связанная с криптографией, разделена между классами. Класс Aes решает математические задачи; он применяет алгоритм шифрования вместе с его объектами шифратора и дешифратора. Класс CryptoStream является связующим звеном; он заботится о взаимодействии с потоками. Класс Aes можно заменить другим классом симметричного алгоритма, но по-прежнему пользоваться CryptoStream.

Класс CryptoStream является *дву направленным*, что означает возможность чтения или записи в поток в зависимости от выбора CryptoStreamMode.Read или CryptoStreamMode.Write. Шифратор и дешифратор способны выполнять чтение и запись, давая в результате четыре комбинации. Чтение может быть удобно моделировать как “выталкивание”, а запись — как “заталкивание”. В случае сомнений начните с Write для шифрования и Read для расшифровки; зачастую такой подход будет наиболее естественным.

Для генерации случайного ключа или вектора IV применяйте класс RandomNumberGenerator из пространства имен System.Cryptography. Числа, которые он производит, являются по-настоящему непредсказуемыми, или *криптостойкими* (класс System.Random подобное не гарантирует). Вот пример:

```
byte[] key = new byte [16];
byte[] iv = new byte [16];
RandomNumberGenerator rand = RandomNumberGenerator.Create();
rand.GetBytes (key);
rand.GetBytes (iv);
```

Или, начиная с версии .NET 6:

```
byte[] key = RandomNumberGenerator.GetBytes (16);
byte[] iv = RandomNumberGenerator.GetBytes (16);
```

Если ключ и вектор IV не указаны, тогда криптостойкие случайные числа генерируются автоматически. Ключ и вектор IV можно получить через свойства Key и IV объекта Aes.

Шифрование в памяти

В .NET 6 и последующих версиях можно использовать методы EncryptCbc и DecryptCbc для ускорения процесса шифрования и расшифровки байтовых массивов:

```
public static byte[] Encrypt (byte[] data, byte[] key, byte[] iv)
{
    using Aes algorithm = Aes.Create();
    algorithm.Key = key;
    return algorithm.EncryptCbc (data, iv);
}

public static byte[] Decrypt (byte[] data, byte[] key, byte[] iv)
{
    using Aes algorithm = Aes.Create();
    algorithm.Key = key;
    return algorithm.DecryptCbc (data, iv);
}
```

Вот эквивалентный код, который работает во всех версиях .NET:

```
public static byte[] Encrypt (byte[] data, byte[] key, byte[] iv)
{
    using (Aes algorithm = Aes.Create())
        using (ICryptoTransform encryptor = algorithm.CreateEncryptor (key, iv))
            return Crypt (data, encryptor);
}

public static byte[] Decrypt (byte[] data, byte[] key, byte[] iv)
{
    using (Aes algorithm = Aes.Create())
        using (ICryptoTransform decryptor = algorithm.CreateDecryptor (key, iv))
            return Crypt (data, decryptor);
}

static byte[] Crypt (byte[] data, ICryptoTransform cryptor)
{
    MemoryStream m = new MemoryStream();
    using (Stream c = new CryptoStream (m, cryptor, CryptoStreamMode.Write))
        c.Write (data, 0, data.Length);
    return m.ToArray();
}
```

Здесь `CryptoStreamMode.Write` хорошо работает как для шифрования, так и для расшифровки, поскольку в обоих случаях осуществляется “заталкивание” внутрь нового потока в памяти.

Ниже приведены перегруженные версии методов, которые принимают и возвращают строки:

```
public static string Encrypt (string data, byte[] key, byte[] iv)
{
    return Convert.ToBase64String (
        Encrypt (Encoding.UTF8.GetBytes (data), key, iv));
}

public static string Decrypt (string data, byte[] key, byte[] iv)
{
    return Encoding.UTF8.GetString (
        Decrypt (Convert.FromBase64String (data), key, iv));
}
```

Их использование демонстрируется в следующем коде:

```
byte[] key = new byte[16];
byte[] iv = new byte[16];

var cryptoRng = RandomNumberGenerator.Create();
cryptoRng.GetBytes (key);
cryptoRng.GetBytes (iv);

string encrypted = Encrypt ("Yeah!", key, iv);
Console.WriteLine (encrypted); // R1/5gYvcxyR2vzPjnT7yaQ==

string decrypted = Decrypt (encrypted, key, iv);
Console.WriteLine (decrypted); // Yeah!
```

Соединение в цепочку потоков шифрования

Класс `CryptoStream` представляет собой декоратор, что означает возможность его соединения в цепочки с другими потоками. В показанном ниже примере мы записываем сжатый зашифрованный текст в файл, после чего читаем его обратно:

```
byte[] key = new byte [16];
byte[] iv = new byte [16];
var cryptoRng = RandomNumberGenerator.Create();
cryptoRng.GetBytes (key);
cryptoRng.GetBytes (iv);
using (Aes algorithm = Aes.Create())
{
    using (ICryptoTransform encryptor = algorithm.CreateEncryptor(key, iv))
        using (Stream f = File.Create ("serious.bin"))
            using (Stream c = new CryptoStream (f, encryptor, CryptoStreamMode.Write))
                using (Stream d = new DeflateStream (c, CompressionMode.Compress))
                    using (StreamWriter w = new StreamWriter (d))
                        await w.WriteLineAsync ("Small and secure!");
    using (ICryptoTransform decryptor = algorithm.CreateDecryptor(key, iv))
        using (Stream f = File.OpenRead ("serious.bin"))
            using (Stream c = new CryptoStream (f, decryptor, CryptoStreamMode.Read))
                using (Stream d = new DeflateStream (c, CompressionMode.Decompress))
                    using (StreamReader r = new StreamReader (d))
                        Console.WriteLine (await r.ReadLineAsync()); // Small and secure!
}
```

(В качестве финального штриха мы сделали программу асинхронной, вызывая методы `WriteLineAsync` и `ReadLineAsync`, а затем ожидая результат.)

В данном примере все однобуквенные переменные формируют часть цепочки. Объекты `algorithm`, `encryptor` и `decryptor` помогают `CryptoStream` выполнять работу по шифрованию, как проиллюстрировано на рис. 20.1.

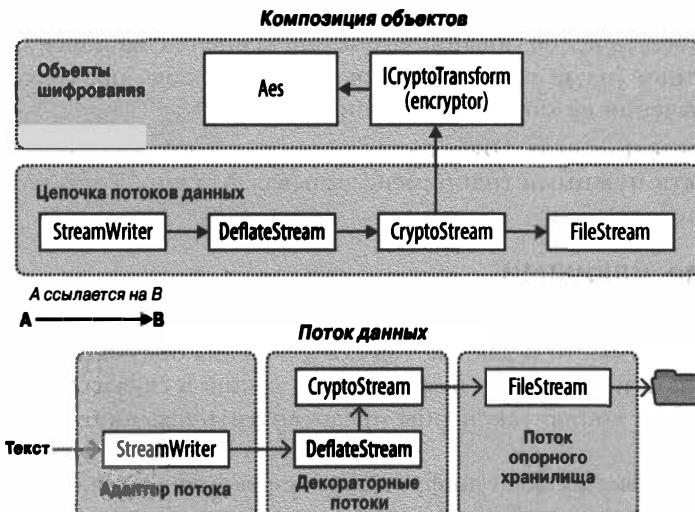


Рис. 20.1. Соединение в цепочку потоков шифрования и сжатия

Соединение в цепочку потоков в такой манере требует мало памяти вне зависимости от конечных размеров потоков.

Освобождение объектов шифрования

Освобождение объекта `CryptoStream` гарантирует, что содержимое его внутреннего кеша данных будетброшено в лежащий в основе поток. Внутреннее кеширование является необходимым для алгоритмов шифрования, т.к. они обрабатывают данные блоками, а не по одному байту за раз.

Класс `CryptoStream` необычен тем, что его метод `Flush` ничего не делает. Чтобы сбросить поток (не освобождая его), потребуется вызвать метод `FlushFinalBlock`. В противоположность методу `Flush` метод `FlushFinalBlock` может быть вызван только однократно, после чего никакие дополнительные данные записать не удастся.

В рассматриваемых примерах мы также освобождаем объект `Aes` и объекты, реализующие `ICryptoTransform` (`encryptor` и `decryptor`). Когда трансформации Рэндала освобождаются, они очищают в памяти симметричный ключ и связанные с ним данные, предотвращая последующее их обнаружение другим программным обеспечением, которое функционирует на компьютере (речь идет о вредоносном ПО). В выполнении этой работы нельзя полагаться на сборщик мусора, поскольку он просто помечает разделы памяти как свободные, не обнуляя все их байты.

Простейший способ освободить объект `Aes` за пределами оператора `using` — вызвать метод `Clear`. Его метод `Dispose` скрыт через явную реализацию (чтобы сигнализировать о необычной семантике освобождения, согласно которой он очищает память, а не освобождает управляемые ресурсы).



Вы можете дополнительно снизить уязвимость своего приложения в плане утечки секретных данных через освобожденную память, предпринимая следующие меры:

- избегать использования строк для хранения защищенной информации (из-за того, что строки неизменяемы, после создания их значения не могут быть очищены);
- перезаписывать буферы сразу после того, как они перестают быть нужными (например, вызывая `Array.Clear` на байтовом массиве).

Управление ключами

Управление ключами — критически важный элемент безопасности: если ваши ключи уязвимы, то и данные тоже. Вы должны обдумать, кто будет иметь доступ к ключам, и как делать их резервную копию в случае аппаратного сбоя, при этом хранить копию так, чтобы предотвратить несанкционированный доступ к ней.

Жестко кодировать ключи шифрования не рекомендуется, потому что существуют популярные инструменты для декомпиляции сборок, требующие незначительного опыта для их использования. Более удачное решение (в Windows)

предусматривает построение для каждой установки случайного ключа и его сохранение безопасным образом с помощью защиты данных Windows.

Для приложений, развертываемых в облаке, Microsoft Azure и Amazon Web Services (AWS) предлагают системы управления ключами с дополнительными средствами, которые могут оказаться полезными в производственной среде (например, аудит).

Если вы шифруете поток сообщений, тогда шифрование с открытым ключом обеспечивает еще лучший вариант.

Шифрование с открытым ключом и подписание

Криптография с открытым ключом является *асимметричной*, что означает применение разных ключей для шифрования и расшифровки.

В отличие от симметричного шифрования, где ключом может служить любая произвольная последовательность байтов подходящей длины, асимметричная криптография требует специально сформированных пар ключей. Пара ключей содержит компоненты *открытого ключа* и *секретного ключа*, которые работают вместе следующим образом:

- открытый ключ шифрует сообщения;
- секретный ключ расшифровывает сообщения.

Участник, “формирующий” пару ключей, хранит секретный ключ вдали от глаз, а открытый ключ распространяет свободно. Особенность такого типа криптографии заключается в том, что вычислить секретный ключ на основе открытого ключа невозможно. Таким образом, в случае утери секретного ключа зашифрованные данные не могут быть восстановлены; если же произошла утечка секретного ключа, тогда вся система шифрования становится бесполезной.

Предоставление открытого ключа позволяет двум компьютерам взаимодействовать защищенным образом через публичную сеть без предварительного контакта и без существующего общего секрета. Чтобы посмотреть, как это работает, предположим, что компьютер *Origin* должен отправить конфиденциальное сообщение компьютеру *Target*.

1. Компьютер *Target* генерирует пару открытого и секретного ключей и затем отправляет открытый ключ компьютеру *Origin*.
2. Компьютер *Origin* шифрует конфиденциальное сообщение с использованием открытого ключа компьютера *Target*, после чего отправляет его *Target*.
3. Компьютер *Target* расшифровывает конфиденциальное сообщение с помощью своего секретного ключа.

А вот что будет видеть пассивный перехватчик сообщений:

- открытый ключ компьютера *Target*;
- конфиденциальное сообщение, зашифрованное посредством открытого ключа компьютера *Target*.

Однако без секретного ключа компьютера *Target* сообщение не может быть расшифровано.



Шифрование с открытым ключом не предотвращает атаки типа “человек посередине”: другими словами, компьютер *Origin* не может знать, является компьютер *Target* злумышленным участником или нет. Для аутентификации получателя отправитель уже должен знать открытый ключ получателя либо иметь возможность проверить достоверность ключа через цифровой сертификат сайта.

Из-за того, что шифрование с открытым ключом выполняется относительно медленно, а размер сообщений ограничен, конфиденциальное сообщение, отправленное из компьютера *Origin* в компьютер *Target*, обычно содержит новый ключ для последующего симметричного шифрования. Это позволяет прекратить шифрование с открытым ключом для оставшейся части сеанса и отдать предпочтение симметричному алгоритму, способному обрабатывать более крупные сообщения. Такой протокол особенно безопасен, если для каждого сеанса генерируется новая пара открытого и секретного ключей, так что хранить какие-либо ключи ни на том, ни на другом компьютере не понадобится.



Алгоритмы шифрования с открытым ключом полагаются на то, что сообщение по размерам меньше ключа. В итоге они становятся подходящими для шифрования только небольших объемов данных, таких как ключ для последующего симметричного шифрования. Если вы попытаетесь зашифровать сообщение, которое намного больше половины размера ключа, тогда поставщик криптографии генерирует исключение.

Класс RSA

.NET предлагает несколько асимметричных алгоритмов, самым популярным из которых является RSA. Ниже показано, как шифровать и расшифровывать с помощью RSA:

```
byte[] data = { 1, 2, 3, 4, 5 }; // Это данные, которые будут шифроваться.  
using (var rsa = new RSACryptoServiceProvider())  
{  
    byte[] encrypted = rsa.Encrypt (data, true);  
    byte[] decrypted = rsa.Decrypt (encrypted, true);  
}
```

Поскольку мы не указали открытый или секретный ключ, поставщик криптографии автоматически генерирует пару ключей, применяя стандартную длину 1024 бита; посредством конструктора можно запросить более длинные ключи с приращением в 8 байтов. Для приложений, критических в отношении безопасности, разумно запрашивать длину в 2048 бит:

```
var rsa = new RSACryptoServiceProvider (2048);
```

Генерация пары ключей связана с интенсивными вычислениями, требуя примерно 10 мс. По указанной причине реализация RSA задерживает генерацию вплоть до момента, когда ключ действительно необходим, скажем, при вызове метода Encrypt. Это дает шанс загрузить существующий ключ или пару ключей, если она существует.

Методы ImportCspBlob и ExportCspBlob загружают и сохраняют ключи в формате байтового массива. Методы FromXmlString и ToXmlString делают то же самое в формате строки, содержащей XML-фрагмент. Флаг типа bool позволяет указывать, нужно ли при сохранении включать секретный ключ. Ниже демонстрируется построение пары ключей и сохранение ее на диске:

```
using (var rsa = new RSACryptoServiceProvider())
{
    File.WriteAllText ("PublicKeyOnly.xml", rsa.ToXmlString (false));
    File.WriteAllText ("PublicPrivate.xml", rsa.ToXmlString (true));
}
```

Так как мы не предоставили существующие ключи, метод ToXmlString создаст новую пару ключей (при первом вызове). В следующем примере мы читаем эти ключи и используем их для шифрования и расшифровки сообщения:

```
byte[] data = Encoding.UTF8.GetBytes ("Message to encrypt");

string publicKeyOnly = File.ReadAllText ("PublicKeyOnly.xml");
string publicPrivate = File.ReadAllText ("PublicPrivate.xml");

byte[] encrypted, decrypted;
using (var rsaPublicOnly = new RSACryptoServiceProvider())
{
    rsaPublicOnly.FromXmlString (publicKeyOnly);
    encrypted = rsaPublicOnly.Encrypt (data, true);
    // Следующая строка кода генерирует исключение, потому что
    // для расшифровки необходим секретный ключ:
    // decrypted = rsaPublicOnly.Decrypt (encrypted, true);
}
using (var rsaPublicPrivate = new RSACryptoServiceProvider())
{
    // С помощью секретного ключа можно успешно расшифровывать:
    rsaPublicPrivate.FromXmlString (publicPrivate);
    decrypted = rsaPublicPrivate.Decrypt (encrypted, true);
}
```

Цифровые подписи

Алгоритмы с открытым ключом могут также применяться для цифрового подписания сообщений или документов. Подпись подобна хешу за исключением того, что ее создание требует секретного ключа, а потому она не может быть подделана. Для проверки подлинности подписи используется открытый ключ. Ниже приведен пример:

```
byte[] data = Encoding.UTF8.GetBytes ("Message to sign");
byte[] publicKey;
byte[] signature;
object hasher = SHA1.Create(); // Выбранный алгоритм хеширования.
```

```
// Сгенерировать новую пару ключей, затем с ее помощью подписать данные:  
using (var publicPrivate = new RSACryptoServiceProvider())  
{  
    signature = publicPrivate.SignData (data, hasher);  
    publicKey = publicPrivate.ExportCspBlob (false); // Получить открытый ключ  
}  
  
// Создать новый объект поставщика шифрования RSA, используя  
// только открытый ключ, затем протестировать подпись  
using (var publicOnly = new RSACryptoServiceProvider())  
{  
    publicOnly.ImportCspBlob (publicKey);  
    Console.Write (publicOnly.VerifyData (data, hasher, signature)); // True  
  
    // Давайте теперь подделаем данные и перепроверим подпись:  
    data[0] = 0;  
    Console.Write (publicOnly.VerifyData (data, hasher, signature)); // False  
  
    // Следующий вызов генерирует исключение из-за отсутствия секретного ключа:  
    signature = publicOnly.SignData (data, hasher);  
}
```

Подписание работает за счет хеширования данных с последующим применением к результатирующему хешу асимметричного алгоритма. Из-за того, что хеши имеют небольшой фиксированный размер, крупные документы могут подписываться относительно быстро (шифрование с открытым ключом намного интенсивнее эксплуатирует центральный процессор, чем хеширование). При желании можно выполнить хеширование самостоятельно, а затем вызвать метод `SignHash` вместо `SignData`:

```
using (var rsa = new RSACryptoServiceProvider())  
{  
    byte[] hash = SHA1.Create().ComputeHash (data);  
    signature = rsa.SignHash (hash, CryptoConfig.MapNameToOID ("SHA1"));  
    ...  
}
```

Методу `SignHash` по-прежнему необходимо знать, какой алгоритм хеширования использовался; метод `CryptoConfig.MapNameToOID` предоставляет эту информацию в корректном формате на основе дружественного имени, такого как "SHA1".

Класс `RSACryptoServiceProvider` генерирует подписи, размер которых соответствует размеру ключа. В настоящее время ни один из основных алгоритмов не генерирует защищенные подписи, длина которых была бы значительно меньше 128 байтов (подходящие, например, для кодов активации продуктов).



Чтобы подписание было эффективным, получатель должен знать и доверять открытому ключу отправителя, что можно обеспечить через заблаговременные коммуникации, предварительную конфигурацию или сертификат сайта. Сертификат сайта представляет собой электронную запись открытого ключа и имени отправителя, которая сама подписана независимым доверенным центром. Типы для работы с сертификатами определены в пространстве имен `System.Security.Cryptography.X509Certificates`.



Расширенная многопоточность

Глава 14 начиналась с рассмотрения основ многопоточности в качестве подготовки к исследованию задач и асинхронности. В частности, было показано, каким образом запускать и конфигурировать поток. Вдобавок были раскрыты такие важные концепции, как организация пула потоков, блокировка, зацикливание и контексты синхронизации. Кроме того, обсуждалась блокировка и безопасность к потокам и демонстрировалась простейшая сигнализирующая конструкция `ManualResetEvent`.

В настоящей главе мы возвращаемся к теме многопоточности. В первых трех разделах будут предоставлены дополнительные сведения по синхронизации, блокировке и безопасности в отношении потоков. Затем мы рассмотрим следующие темы:

- немонопольное блокирование (`Semaphore` и блокировки объектов чтения/записи);
- все сигнализирующие конструкции (`AutoResetEvent`, `ManualResetEvent`, `CountdownEvent` и `Barrier`);
- ленивая инициализация (`Lazy<T>` и `LazyInitializer`);
- локальное хранилище потока (`ThreadStaticAttribute`, `ThreadLocal<T>` и `GetData/SetData`);
- таймеры.

Многопоточность является настолько обширной темой, так что по ссылке <http://albahari.com/threading/> доступны дополнительные материалы, посвященные перечисленным ниже более тонким темам:

- использование методов `Monitor.Wait` и `Monitor.Pulse` в специализированных сигнализирующих сценариях;
- приемы неблокирующей синхронизации для микрооптимизации (`Interlocked`, барьеры памяти, `volatile`);
- применение типов `SpinLock` и `SpinWait` в сценариях с высоким уровнем параллелизма.

Обзор синхронизации

Синхронизация представляет собой акт координирования параллельно выполняемых действий с целью получения предсказуемых результатов. Синхронизация особенно важна, когда множество потоков получают доступ к одним и тем же данным; в этой области удивительно легко столкнуться с серьезными трудностями.

Вероятно, простейшими и наиболее удобными инструментами синхронизации считаются продолжения и комбинаторы задач, описанные в главе 14. За счет представления параллельных программ в виде асинхронных операций, связанных вместе продолжениями и комбинаторами, уменьшается необходимость в блокировке и сигнализации. Тем не менее, по-прежнему встречаются ситуации, когда в игру вступают низкоуровневые конструкции. Конструкции синхронизации могут быть разделены на три описанные ниже категории.

- **Монопольное блокирование.** Конструкции монопольного блокирования позволяют выполнять некоторое действие или запускать определенный раздел кода только одному потоку в каждый момент времени. Их основное назначение заключается в том, чтобы предоставить потокам возможность доступа к допускающему запись совместно используемому состоянию, не влияя друг на друга. Конструкциями монопольного блокирования являются `lock`, `Mutex` и `SpinLock`.
- **Немонопольное блокирование.** Немонопольное блокирование позволяет ограничивать параллелизм. Конструкциями немонопольного блокирования являются `Semaphore` (`SemaphoreSlim`) и `ReaderWriterLock` (`ReaderWriterLockSlim`).
- **Сигнализация.** Сигнализация позволяет потоку блокироваться вплоть до получения одного или большего числа уведомлений от другого потока (потоков). Сигнализирующие конструкции включают `ManualResetEvent` (`ManualResetEventSlim`), `AutoResetEvent`, `CountdownEvent` и `Barrier`. Первые три конструкции называются *дескрипторами ожидания событий*.

Кроме того, возможно (хотя и сложно) выполнять определенные параллельные операции на совместно используемом состоянии без блокирования за счет использования *неблокирующих конструкций синхронизации*. Существуют методы `Thread.MemoryBarrier`, `Thread.VolatileRead` и `Thread.VolatileWrite`, ключевое слово `volatile`, а также класс `Interlocked`. Эта тема раскрывается в статье по ссылке <http://albahari.com/threading/>; там же приведено описание методов `Wait/Pulse` класса `Monitor`, которые могут применяться для написания специальной сигнализирующей логики.

Монопольное блокирование

Доступны три конструкции монопольного блокирования: оператор `lock`, класс `Mutex` и структура `SpinLock`. Конструкция `lock` является наиболее удобной и часто используемой, в то время как другие две конструкции ориентированы на собственные сценарии:

- класс Mutex позволяет охватывать множество процессов (блокировки на уровне компьютера);
- структура SpinLock реализует микрооптимизацию, которая может уменьшить количество переключений контекста в сценариях с высоким уровнем параллелизма (см. <http://albahari.com/threading/>).

Оператор lock

Для иллюстрации потребности в блокировании рассмотрим следующий класс:

```
class ThreadUnsafe
{
    static int _val1 = 1, _val2 = 1;
    static void Go()
    {
        if (_val2 != 0) Console.WriteLine (_val1 / _val2);
        _val2 = 0;
    }
}
```

Класс ThreadUnsafe не безопасен в отношении потоков: если метод Go был вызван двумя потоками одновременно, то появляется возможность получения ошибки деления на ноль. Дело в том, что в одном потоке поле _val2 может быть установлено в 0 как раз тогда, когда выполнение в другом потоке находится между оператором if и вызовом метода Console.WriteLine. Ниже показано, как исправить проблему с помощью lock:

```
class ThreadSafe
{
    static readonly object _locker = new object();
    static int _val1 = 1, _val2 = 1;
    static void Go()
    {
        lock (_locker)
        {
            if (_val2 != 0) Console.WriteLine (_val1 / _val2);
            _val2 = 0;
        }
    }
}
```

В каждый момент времени блокировать объект синхронизации (в данном случае _locker) может только один поток, и любые соперничающие потоки задерживаются до тех пор, пока блокировка не будет освобождена. Если за блокировку соперничают несколько потоков, тогда они ставятся в “очередь готовности” с предоставлением блокировки на основе “первым пришел — первым обслужен”¹. Говорят, что монопольные блокировки иногда приводят к последовательному доступу к объекту, защищаемому блокировкой, т.к. доступ одного

¹ Равноправие в этой очереди временами может нарушаться из-за нюансов поведения Windows и CLR.

потока не может совмещаться с доступом другого. В рассматриваемом случае мы защищаем логику внутри метода Go, а также поля `_val1` и `_val2`.

Monitor.Enter И Monitor.Exit

Фактически оператор `lock` в C# является синтаксическим сокращением для вызова методов `Monitor.Enter` и `Monitor.Exit` с добавленным блоком `try/finally`. Ниже показана упрощенная версия того, что на самом деле происходит внутри метода `Go` из предыдущего примера:

```
Monitor.Enter (_locker);
try
{
    if (_val2 != 0) Console.WriteLine (_val1 / _val2);
    _val2 = 0;
} finally { Monitor.Exit (_locker); }
```

Вызов метода `Monitor.Exit` без предварительного вызова `Monitor.Enter` на том же объекте приводит к генерации исключения.

Перегруженная версия Monitor.Enter, принимающая аргумент lockTaken

В продемонстрированном выше коде присутствует тонкая уязвимость. Представим себе (маловероятный) случай генерации исключения между вызовом метода `Monitor.Enter` и блоком `try` (возможно, `OutOfMemoryException` или прекращения работы потока в .NET Framework). При таком сценарии блокировка может быть получена или не получена. Если блокировка *получена*, то она не будет освобождена, поскольку мы никогда не войдем в блок `try/finally`. В результате происходит утечка блокировки. Во избежание подобной опасности была определена следующая перегруженная версия `Monitor.Enter`:

```
public static void Enter (object obj, ref bool lockTaken);
```

Значение `lockTaken` станет равным `false`, если (и только если) метод `Enter` сгенерировал исключение, и блокировка не была получена.

Вот как выглядит более надежный шаблон применения (именно так компилятор C# транслирует оператор `lock`):

```
bool lockTaken = false;
try
{
    Monitor.Enter (_locker, ref lockTaken);
    // Выполнить необходимые действия...
}
finally { if (lockTaken) Monitor.Exit (_locker); }
```

TryEnter

Класс `Monitor` также предлагает метод `TryEnter`, который позволяет указывать тайм-аут в миллисекундах или в виде структуры `TimeSpan`. Метод `TryEnter` возвращает `true`, если блокировка получена, или `false`, если никаких блокировок не получено из-за истечения времени тайм-аута.

Метод TryEnter можно также вызывать без аргументов, что дает возможность “проверить” блокировку, немедленно инициируя тайм-аут, если блокировка не может быть получена сразу. Как и Enter, метод TryEnter перегружен для приема аргумента lockTaken.

Выбор объекта синхронизации

Использовать в качестве объекта синхронизации можно любой объект, видимый каждому участвующему потоку, но при одном жестком условии: он должен быть ссылочного типа. Объект синхронизации обычно является закрытым (потому что это помогает инкапсулировать логику блокирования) и, как правило, представляет собой поле экземпляра или статическое поле. Объект синхронизации может дублировать защищаемый посредством него объект, что и делает поле _list в следующем примере:

```
class ThreadSafe
{
    List <string> _list = new List <string>();
    void Test()
    {
        lock (_list)
        {
            _list.Add ("Item 1");
            ...
        }
    }
}
```

Поле, выделенное для целей блокирования (вроде _locker в предыдущем примере), обеспечивает точный контроль над областью видимости и степенью детализации блокировки. Применяться в качестве объекта синхронизации может также содержащий объект (this) либо даже его тип:

```
lock (this) { ... }
```

или:

```
lock (typeof (Widget)) { ... } // Для защиты доступа к статическим членам
```

Недостаток блокирования подобным образом связан с тем, что логика блокирования не инкапсулирована, а потому предотвратить взаимоблокировки и избыточные блокировки становится труднее.

Можно также блокировать локальные переменные, захваченные лямбда-выражениями или анонимными методами.



Блокирование никак не ограничивает доступ к самому объекту синхронизации. Другими словами, вызов x.ToString() не будет блокироваться из-за того, что другой поток вызывает lock (x); чтобы блокирование произошло, вызывать lock (x) должны оба потока.

Когда нужна блокировка

Запомните базовое правило: блокировка необходима при доступе к **любому совместно используемому полю, допускающему запись**. Синхронизация должна приниматься во внимание даже в таком простейшем случае, как операция при-

сваивания для одиночного поля. В следующем классе ни метод Increment, ни метод Assign не является безопасным в отношении потоков:

```
class ThreadUnsafe
{
    static int _x;
    static void Increment() { _x++; }
    static void Assign() { _x = 123; }
}
```

А вот безопасные к потокам версии методов Increment и Assign:

```
static readonly object _locker = new object();
static int _x;

static void Increment() { lock (_locker) _x++; }
static void Assign() { lock (_locker) _x = 123; }
```

Когда блокировки отсутствуют, могут возникать две проблемы.

- Операции вроде инкрементирования значения переменной (а в определенных обстоятельствах даже чтение/запись переменной) не являются атомарными.
- Компилятор, среда CLR и процессор имеют право изменять порядок следования инструкций и кешировать переменные в регистрах центрального процессора в целях улучшения производительности — до тех пор, пока такие оптимизации не изменяют поведение *однопоточной* программы (или многопоточной программы, в которой используются блокировки).

Блокирование смягчает вторую проблему, т.к. оно создает *барьер памяти* до и после блокировки. Барьер памяти представляет собой “заграждающую метку”, которую не могут пересечь указанные эффекты либо изменение порядка следования и кеширование.



Сказанное применимо не только к блокировкам, но и ко всем конструкциям синхронизации. Таким образом, если используется, например, *сигнализирующая* конструкция, которая гарантирует, что в каждый момент времени переменная читается/записывается только одним потоком, тогда блокировка не нужна. Следовательно, показанный ниже код является безопасным к потокам без блокирования x:

```
var signal = new ManualResetEvent (false);
int x = 0;
new Thread (() => { x++; signal.Set(); }).Start();
signal.WaitOne();
Console.WriteLine (x); // 1 (всегда)
```

В разделе “Nonblocking Synchronization” (“Неблокирующая синхронизация”) по ссылке <http://albahari.com/threading/> мы объясняем, как возникла такая потребность, и показываем, каким образом барьеры памяти и класс Interlocked могут предложить альтернативы блокированию в подобных ситуациях.

Блокирование и атомарность

Если группа переменных всегда читается и записывается внутри одной и той же блокировки, то можно говорить о том, что эти переменные читаются и записываются *атомарно*. Давайте предположим, что поля `x` и `y` всегда читаются и устанавливаются внутри блокировки на объекте `locker`:

```
lock (locker) { if (x != 0) y /= x; }
```

Можно сказать, что доступ к `x` и `y` производится атомарно, поскольку блок кода не может быть разделен или вытеснен действиями другого потока так, что он изменит содержимое `x` или `y` и *сделает результаты недействительными*. Обеспечивая доступ к `x` и `y` всегда внутри одной и той же монопольной блокировки, вы никогда не получите ошибку деления на ноль.



Предоставляемая блокировкой атомарность нарушается, если внутри блока `lock` генерируется исключение (независимо от наличия или отсутствия многопоточности). Например, взгляните на следующий код:

```
decimal _savingsBalance, _checkBalance;  
void Transfer (decimal amount)  
{  
    lock (_locker)  
    {  
        _savingsBalance += amount;  
        _checkBalance -= amount + GetBankFee();  
    }  
}
```

Если метод `GetBankFee` генерирует исключение, тогда банк потеряет деньги. В таком случае мы могли бы избежать проблемы, вызвав `GetBankFee` раньше. Решение для более сложных ситуаций предусматривает реализацию логики “отката” внутри блока `catch` или `finally`.

Атомарность инструкций представляет собой другую, хотя и похожую концепцию: инструкция считается атомарной, если она выполняется неделимым образом на лежащем в основе процессоре.

Вложенное блокирование

Поток может многократно блокировать один и тот же объект вложенным (реинтерабельным) образом:

```
lock (locker)  
lock (locker)  
lock (locker)  
{  
    // Выполнить необходимые действия...  
}
```

или по-другому:

```
Monitor.Enter (locker); Monitor.Enter (locker); Monitor.Enter (locker);  
// Выполнить необходимые действия...  
Monitor.Exit (locker); Monitor.Exit (locker); Monitor.Exit (locker);
```

В таких сценариях объект деблокируется, только когда завершается самый внешний оператор `lock` или выполняется совпадающее количество операторов `Monitor.Exit`. Вложенное блокирование удобно, если один метод вызывает другой изнутри блокировки:

```
object locker = new object();

lock (locker)
{
    AnotherMethod();
    // Мы по-прежнему имеем блокировку, т.к. она является реентерабельной.
}

void AnotherMethod()
{
    lock (locker) { Console.WriteLine ("Another method"); }
}
```

Поток может блокироваться только на первой (самой внешней) блокировке.

Взаимоблокировки

Взаимоблокировка случается, когда каждый из двух потоков ожидает ресурс, удерживаемый другим потоком, так что ни один из них не может продолжить работу. Сказанное проще всего проиллюстрировать с помощью двух блокировок:

```
object locker1 = new object();
object locker2 = new object();

new Thread (() => {
    lock (locker1)
    {
        Thread.Sleep (1000);
        lock (locker2);      // Взаимоблокировка
    }
}).Start();

lock (locker2)
{
    Thread.Sleep (1000);
    lock (locker1);      // Взаимоблокировка
}
```

Три и большее количество потоков могут породить более сложные цепочки взаимоблокировок.



В стандартной среде размещения система CLR не похожа на SQL Server; она не обнаруживает и не устраняет взаимоблокировки автоматически, принудительно прекращая работу одного из нарушителей. Взаимоблокировка потоков приводит к тому, что участвующие потоки блокируются на неопределенный срок, если только не был указан тайм-аут блокировки. (Тем не менее, под управлением хоста интеграции CLR с SQL Server взаимоблокировки *обнаруживаются* автоматически с генерацией перехватываемого исключения в одном из потоков.)

Взаимоблокировка является одной из самых сложных проблем многопоточности — особенно, когда есть множество взаимосвязанных объектов. По существу сложность кроется в том, что вы не можете с уверенностью сказать, какие блокировки получил *вызывающий поток*.

Таким образом, вы можете блокировать закрытое поле а внутри своего класса x, не зная, что вызывающий поток (или поток, обращающийся к вызывающему потоку) уже заблокировал поле b в классе у. Тем временем другой поток делает обратное, создавая взаимоблокировку. По иронии судьбы проблема усугубляется (хорошими) паттернами объектно-ориентированного проектирования, потому что паттерны подобного рода создают цепочки вызовов, которые не определены вплоть до стадии выполнения.

Хотя популярный совет блокировать объекты в согласованном порядке во избежание взаимоблокировок был полезен в начальном примере, он труден в применении к только что описанному сценарию. Лучшая стратегия заключается в том, чтобы проявлять осторожность при блокировании обращений к методам в объектах, которые могут иметь ссылки на ваш объект. Кроме того, следует подумать, действительно ли нужна блокировка обращений к методам в других классах (как будет показано в разделе “Блокирование и безопасность к потокам” далее в главе, это делается часто, но иногда доступны другие возможности). В большей степени полагаясь на высокоуровневые средства синхронизации, такие как продолжения/комбинаторы задач, параллелизм данных и неизменяемые типы (рассматриваются далее в главе), потребность в блокировании можно снизить.



Существует альтернативный путь восприятия данной проблемы: когда вы обращаетесь к другому коду, удерживая блокировку, инкапсуляция такой блокировки незаметно исчезает. Это не ошибка в CLR, а фундаментальное ограничение блокирования в целом. Проблемы блокирования решаются в рамках разнообразных исследовательских проектов, включая проект *Software Transactional Memory* (Программная транзакционная память).

Еще один сценарий взаимоблокировки возникает при вызове метода `Dispatcher.Invoke` (в приложении WPF) или `Control.Invoke` (в приложении Windows Forms) во время владения блокировкой. Если случится так, что пользовательский интерфейс выполняет другой метод, который ожидает ту же самую блокировку, то именно здесь и возникнет взаимоблокировка. Часто проблему можно устраниТЬ, просто вызывая метод `BeginInvoke` вместо `Invoke` (или положиться на асинхронные функции, которые делают это неявно, когда присутствует контекст синхронизации). В качестве альтернативы перед вызовом `Invoke` можно освободить свою блокировку, хотя прием не сработает, если блокировку отобрал *вызывающий поток*.

Производительность

Блокировка выполняется быстро: можно ожидать, что получение и освобождение блокировки на современном (2020-х годов) компьютере займет менее 20 нс при отсутствии соперничества за эту блокировку. В случае соперничества

побочное переключение контекста смешает накладные расходы ближе к микросекундной области, хотя они могут оказаться еще больше перед тем, как действительно произойдет повторное планирование потока.

Mutex

Класс Mutex похож на оператор lock языка C#, но он способен работать во множестве процессов. Другими словами, Mutex может иметь область действия на уровне компьютера и приложения. Получение и освобождение объекта Mutex требует около половины микросекунды при отсутствии соперничества, т.е. он более чем в 20 раз медленнее оператора lock.

В случае класса Mutex для блокирования вызывается его метод WaitOne, а для разблокирования — метод ReleaseMutex. Как и оператор lock, объект Mutex может быть освобожден из того же самого потока, в котором он был получен.



Если вы забудете вызвать метод ReleaseMutex и просто сделаете вызов Close или Dispose, то в любом другом потоке, ожидающем данный объект Mutex, генерируется исключение AbandonedMutexException.

Межпроцессный объект Mutex часто используется для обеспечения того, что в каждый момент времени может выполняться только один экземпляр программы. Ниже показано, как это делается.

```
//Назначение объекту Mutex имени делает его доступным на уровне всего компьютера
// Используйте имя, являющееся уникальным для вашей компании и приложения
// (например, включите в него URL компании)

using var mutex = new Mutex (true, @"Global\oreilly.com OneAtATimeDemo");
// Ожидать несколько секунд, если возникло соперничество; в этом случае
// другой экземпляр программы все еще находится в процессе завершения
if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false))
{
    Console.WriteLine ("Another instance of the app is running. Bye!");
    // Выполняется другой экземпляр программы. Завершение
    return;
}
try { RunProgram(); }
finally { mutex.ReleaseMutex (); }

void RunProgram()
{
    Console.WriteLine ("Running. Press Enter to exit");
    // Программа выполняется; нажмите Enter для завершения
    Console.ReadLine ();
}
```



При выполнении под управлением терминальных служб (Terminal Services) или в отдельных консолях Unix объект Mutex уровня компьютера обычно виден только приложениям в том же самом сеансе. Чтобы сделать его видимым всем сессиям терминального сервера, добавьте к его имени префикс Global \, как было сделано в примере.

Блокирование и безопасность к потокам

Программа или метод является безопасным в отношении потоков, если обладает способностью корректно работать в любом многопоточном сценарии. Безопасность к потокам достигается главным образом за счет блокирования и уменьшения возможностей взаимодействия потоков.

По перечисленным ниже причинам универсальные типы редко бывают безопасными к потокам в полном объеме.

- Затраты при разработке, необходимые для обеспечения полной безопасности к потокам, могут оказаться значительными, особенно если тип имеет множество полей (в произвольном многопоточном контексте каждое поле потенциально открыто для взаимодействия).
- Безопасность к потокам может повлечь за собой снижение производительности (частично зависящее от того, применяется ли тип во множестве потоков).
- Безопасный в отношении потоков тип не обязательно автоматически превращает использующую его программу в безопасную к потокам. Часто работа, связанная с построением программы, делает избыточными усилия по достижению безопасности к потокам самого типа.

Таким образом, безопасность к потокам обычно реализуется только там, где она нужна, с целью поддержки специфичного многопоточного сценария.

Однако есть несколько способов “схитрить” и заставить крупные и сложные классы безопасно выполнять в многопоточной среде. Один из них предусматривает принесение в жертву степени детализации за счет помещения больших разделов кода — даже кода доступа ко всему объекту — внутрь единственной монопольной блокировки, обеспечивая последовательный доступ на высоком уровне. На самом деле такая тактика жизненно важна, если необходимо применять небезопасный к потокам код третьей стороны (или большинство типов .NET, если уж на то пошло) в многопоточном контексте. Уловка заключается просто в использовании одной и той же монопольной блокировки для защиты доступа ко всем свойствам, методам и полям небезопасного к потокам объекта. Такое решение хорошо работает, если все методы объекта выполняются быстро (в противном случае будет много блокирований).



Оставив в стороне примитивные типы, лишь очень немногие типы .NET позволяют создавать экземпляры, которые безопасны к потокам за рамками простого параллельного доступа только для чтения. Ответственность за обеспечение безопасности к потокам, обычно посредством монопольных блокировок, возлагается на разработчика. (Исключением являются коллекции из пространства имен `System.Collections.Concurrent`, которые мы рассмотрим в главе 22.)

Другой способ схитрить предполагает минимизацию взаимодействия потоков за счет сведения к минимуму совместно используемых данных. Это великолепный подход, который неявно применяется в лишенных состояния серверах

приложений среднего уровня и серверах веб-страниц. Поскольку множественные клиентские запросы могут поступать одновременно, серверные методы, к которым они обращаются, должны быть безопасными к потокам. Проектное решение, при котором состояние не запоминается (популярное по причинам масштабируемости), по существу ограничивает возможность взаимодействия, т.к. классы не сохраняют данные между запросами. Взаимодействие потоков затем ограничивается только статическими полями, которые могут создаваться для таких целей, как кеширование часто используемых данных в памяти и предоставление инфраструктурных служб вроде аутентификации и аудита.

Еще одно решение (в насыщенных клиентских приложениях) предусматривает запуск кода, который получает доступ к совместно используемому состоянию в потоке пользовательского интерфейса. Как было показано в главе 14, асинхронные функции упрощают реализацию такого подхода.

Безопасность к потокам и типы .NET

Блокирование может использоваться для преобразования небезопасного к потокам кода в код, безопасный в отношении потоков. Хорошим сценарием его применения следует считать саму платформу .NET. Почти все непримитивные типы в ней не являются безопасными к потокам (когда задачи выходят за рамки простого доступа только по чтению), но при этом они могут использоваться в многопоточном коде, если доступ к любому объекту защищен посредством блокировки. Ниже приведен пример, в котором два потока одновременно добавляют элемент к одной и той же коллекции List, после чего организуют перечисление этой коллекции:

```
class ThreadSafe
{
    static List <string> _list = new List <string>();
    static void Main()
    {
        new Thread (AddItem).Start();
        new Thread (AddItem).Start();
    }
    static void AddItem()
    {
        lock (_list) _list.Add ("Item " + _list.Count);
        string[] items;
        lock (_list) items = _list.ToArray();
        foreach (string s in items) Console.WriteLine (s);
    }
}
```

В данном случае мы блокируем сам объект _list. Если бы существовали два взаимосвязанных списка, то для применения блокировки мы должны были бы выбрать общий объект (на его место можно было бы назначить один из списков или — что еще лучше — использовать независимое поле).

Перечисление коллекций .NET также не является безопасным к потокам в том смысле, что если список изменяется во время перечисления, тогда генери-

руется исключение. В приведенном примере вместо блокирования на протяжении всего перечисления мы сначала копируем элементы в массив, что позволяет избежать удержания блокировки чрезмерно долго, если действия, предпринимаемые при перечислении, потенциально могут отнимать много времени. (Другое решение предусматривает использование блокировки объекта чтения/записи, как объясняется в разделе “Блокировки объектов чтения/записи” далее в главе.)

Блокирование безопасных к потокам объектов

Иногда блокировку также необходимо применять при доступе к объектам, безопасным в отношении потоков. В целях иллюстрации предположим, что класс `List` из .NET на самом деле безопасен к потокам, и нужно добавить элемент в список:

```
if (!_list.Contains (newItem)) _list.Add (newItem);
```

Вне зависимости от того, безопасен список к потокам или нет, приведенный оператор таковым определенно не является! Весь этот оператор `if` должен быть помещен внутрь блокировки, чтобы предотвратить вытеснение в промежутке между проверкой наличия элемента в списке и добавлением нового элемента. Ту же самую блокировку затем нужно использовать везде, где список модифицируется. Например, следующий оператор также требует помещения в идентичную блокировку, чтобы его не вытеснил предшествующий оператор:

```
_list.Clear();
```

Другими словами, нам пришлось бы применять блокировки точно так же, как мы поступали с классами небезопасных к потокам коллекций (делая гипотетическую безопасность к потокам класса `List` избыточной).



Применение блокировки к коду доступа в коллекцию может привести к чрезмерному блокированию в средах с высокой степенью параллелизма. Именно потому .NET предлагает безопасные к потокам версии очереди, стека и словаря, которые обсуждаются в главе 22.

Статические члены

Помещение кода доступа к объекту внутрь специальной блокировки работает, только если все параллельные потоки осведомлены — и используют — данную блокировку. Это может быть не так, если объект имеет широкую область видимости. Худший случай касается статических членов в открытом типе. Например, представьте ситуацию, когда статическое свойство структуры `DateTime`, такое как `DateTime.Now`, не является безопасным к потокам, и два параллельных вызова могут дать в результате искаженный вывод либо исключение. Единственный способ устраниТЬ проблему с помощью внешнего блокирования может предусматривать блокировку самого типа, т.е. `lock (typeof(DateTime))`, перед вызовом `DateTime.Now`. Прием сработает, только если все программисты согласятся поступать подобным образом (что маловероятно). Более того, блокировка типа привносит собственные проблемы.

По указанной причине статические члены структуры `DateTime` были осмотрительно запрограммированы как безопасные к потокам. Такой шаблон при-

меняется в .NET повсеместно: *статические члены являются безопасными к потокам, а члены экземпляра — нет*. Следовать упомянутому шаблону также имеет смысл при написании типов для общественного потребления, поскольку он позволяет избежать создания неразрешимых проблем с безопасностью в отношении потоков. Другими словами, делая статические методы безопасными к потокам, вы программируете так, чтобы не препятствовать безопасности к потокам для потребителей данного типа.



Безопасность к потокам в статических методах придется кодировать явным образом: она не появляется автоматически только в силу того, что метод определен как статический!

Безопасность к потокам для доступа только по чтению

Превращение типов в безопасные к потокам для параллельного доступа только по чтению (там, где возможно) дает преимущество в том, что потребители могут избежать излишнего блокирования. Данный принцип соблюдают многие типы в .NET: например, коллекции являются безопасными к потокам для параллельных объектов чтения.

Следовать такому принципу довольно просто: если вы документировали тип как безопасный к потокам для параллельного доступа только по чтению, то не производите запись в поля внутри методов, которые по ожиданиям потребителя должны допускать только чтение (или помещаете такой код внутрь блокировки). Например, реализация метода `ToArray` в коллекции может начинаться с уплотнения внутренней структуры коллекции. Тем не менее, это сделало бы метод небезопасным к потокам для потребителей, которые ожидают, что он допускает только чтение.

Безопасность к потокам для доступа только по чтению является одной из причин, по которым перечислители отделены от классов, поддерживающих перечисление: два потока могут одновременно перечислять коллекцию, потому что каждый из них получает отдельный объект перечислителя.



Если документация по типу отсутствует, тогда имеет смысл проявлять осторожность в предположениях о том, что тот или иной метод по своей природе является предназначенным только для чтения. Хорошим примером может служить класс `Random`: при вызове метода `Random.Next` его внутренняя реализация требует обновления закрытых начальных значений. Следовательно, вы должны либо применять блокировку к коду, использующему класс `Random`, либо поддерживать отдельные экземпляры `Random` для каждого потока.

Безопасность к потокам в серверах приложений

Серверы приложений должны быть многопоточными, чтобы обрабатывать одновременные клиентские запросы. Приложения ASP.NET Core и Web API являются неявно многопоточными. Это означает, что при написании кода на серверной стороне вы должны принимать во внимание безопасность к потокам,

если есть хотя бы малейшая возможность взаимодействия между потоками, обрабатывающими клиентские запросы. К счастью, такая ситуация возникает редко; типичный серверный класс либо не сохраняет состояние (поля отсутствуют), либо имеет модель активизации, которая создает отдельный его экземпляр для каждого клиента или каждого запроса. Взаимодействие обычно происходит только через статические поля, которые иногда применяются для кеширования в памяти частей базы данных с целью повышения производительности.

Например, предположим, что имеется метод `RetrieveUser`, выдающий запрос к базе данных:

```
// User - специальный класс с полями для хранения данных о пользователе
internal User RetrieveUser (int id) { ... }
```

Если метод `RetrieveUser` вызывается часто, тогда показатели производительности можно было бы улучшить, кешируя результаты в статическом объекте `Dictionary`. Ниже показано концептуально простое решение, учитывающее безопасность к потокам:

```
static class UserCache
{
    static Dictionary <int, User> _users = new Dictionary <int, User>();
    internal static User GetUser (int id)
    {
        User u = null;
        lock (_users)
            if (_users.TryGetValue (id, out u))
                return u;
        u = RetrieveUser (id); // Метод для извлечения информации из базы данных
        lock (_users) _users [id] = u;
        return u;
    }
}
```

Для обеспечения безопасности в отношении потоков мы должны, как минимум, применить блокировку к чтению и обновлению словаря. В приведенном примере мы отдаляем предпочтение практическому компромиссу между простотой и производительностью в блокировании. Наше проектное решение создает небольшой потенциал для неэффективности: если два потока одновременно вызовут данный метод с одним и тем же ранее не извлеченным идентификатором `id`, то метод `RetrieveUser` будет вызван дважды — и словарь обновится лишний раз. Одиночное блокирование всего метода могло бы предотвратить такую ситуацию, но породить серьезную неэффективность: на протяжении вызова `RetrieveUser` блокировался бы целый кеш и в это время другие потоки не могли бы извлекать информацию о любых пользователях.

Для идеального решения необходимо использовать стратегию, описанную в разделе “Синхронное завершение” главы 14. Вместо кеширования объекта `User` мы кешируем объект `Task<User>`, на котором вызывающий код организует ожидание:

```

static class UserCache
{
    static Dictionary <int, Task<User>> _userTasks =
        new Dictionary <int, Task<User>>();

    internal static Task<User> GetUserAsync (int id)
    {
        lock (_userTasks)
            if (_userTasks.TryGetValue (id, out var userTask))
                return userTask;
            else
                return _userTasks [id] = Task.Run (() => RetrieveUser (id));
    }
}

```

Обратите внимание, что теперь у нас есть единственная блокировка, которая охватывает логику целого метода. Мы можем поступать так без ущерба параллелизму, поскольку все, что делается внутри блокировки, связано с доступом в словарь и (потенциально) инициированием асинхронной операции (посредством вызова `Task.Run`). Если два потока вызовут этот метод одновременно с тем же самым идентификатором, то в итоге они будут ожидать ту же самую задачу, что как раз и является желательным исходом.

Неизменяемые объекты

Неизменяемым является такой объект, состояние которого не может быть модифицировано, ни внешне, ни внутренне. Поля в неизменяемом объекте обычно объявляются как предназначенные только для чтения и полностью инициализируются во время его конструирования.

Неизменяемость является признаком функционального программирования, где вместо изменения существующего объекта создается новый объект с отличающимися свойствами. Указанной парадигме следует язык LINQ. Неизменяемость также полезна в случае многопоточности — она позволяет избежать проблемы допускающего запись совместно используемого состояния, устроняя (или сводя к минимуму) возможность записи.

Один из шаблонов предусматривает применение неизменяемых объектов для инкапсуляции группы связанных полей, чтобы снизить до минимума продолжительность действия блокировок. В качестве очень простого примера предположим, что имеются два следующих поля:

```

int _percentComplete;
string _statusMessage;

```

Пусть их необходимо читать и записывать атомарным образом. Вместо применения блокировки к этим полям мы можем определить неизменяемый класс, как показано ниже:

```

class ProgressStatus // Представляет ход некоторого действия
{
    public readonly int PercentComplete;
    public readonly string StatusMessage;
    // Этот класс может иметь намного больше полей...
}

```

```
public ProgressStatus (int percentComplete, string statusMessage)
{
    PercentComplete = percentComplete;
    StatusMessage = statusMessage;
}
```

Затем можно определить одиночное поле такого типа вместе с объектом блокировки:

```
readonly object _statusLocker = new object();
ProgressStatus _status;
```

Теперь значения типа `ProgressStatus` можно читать и записывать, не удерживая блокировку для чего-то большего, чем одиночное присваивание:

```
var status = new ProgressStatus (50, "Working on it");
// Здесь можно было бы выполнять присваивание многих других полей...
// ...
lock (_statusLocker) _status = status; // Очень короткая блокировка
```

Чтобы прочитать объект, мы сначала получаем копию ссылки на него (внутри блокировки). Затем мы можем читать его значения без необходимости в удержании блокировки:

```
ProgressStatus status;
lock (_statusLocker) status = _status; // И снова короткая блокировка
int pc = status.PercentComplete;
string msg = status.StatusMessage;
...
```

Немонопольное блокирование

Конструкции немонопольного блокирования предназначены для ограничения параллелизма. В этом разделе будут раскрыты семафоры и блокировки объектов чтения/записи, а также показано, как класс `SemaphoreSlim` может ограничивать параллелизм с помощью асинхронных операций.

Семафор

Семафор чем-то похож на ночной клуб с ограниченной вместительностью, за которой следит вышибала. Когда клуб переполнен, никто в него больше не сможет войти, и снаружи образуется очередь.

Счетчик семафора соответствует количеству мест в ночном клубе. Освобождение семафора увеличивает счетчик; обычно это происходит, когда кто-то покидает клуб (что соответствует освобождению ресурса), а также когда семафор инициализируется (для установки его начальной емкости). Можно также в любой момент вызвать `Release`, чтобы увеличить емкость.

Ожидание семафора уменьшает счетчик и обычно происходит до получения ресурса. Вызов `Wait` на семафоре, текущее значение счетчика которого больше 0, завершается немедленно.

Семафор может необязательно иметь максимальное значение счетчика, которое служит жестким ограничением. Увеличение счетчика сверх этого ограниче-

ния приводит к генерации исключения. При создании семафора вы указываете начальное значение счетчика (начальную емкость) и при необходимости максимальный предел.

Семафор с начальным значением счетчика, равным единице, подобен Mutex или lock за исключением того, что семафор не имеет “владельца” — он независим от потоков. Любой поток способен вызывать метод Release объекта Semaphore, тогда как в случае Mutex и lock освободить блокировку может только поток, который ее получил.



Существуют две функционально похожие версии данного класса: Semaphore и SemaphoreSlim. Последняя версия оптимизирована для удовлетворения требованиям низкой задержки, которые предъявляются параллельным программированием. Она также полезна при традиционном многопоточном программировании, т.к. позволяет указывать признак отмены во время ожидания (см. раздел “Отмена” в главе 14) и открывает доступ к методу WaitAsync для асинхронного программирования. Тем не менее, SemaphoreSlim не может использоваться для сигнализирования между процессами.

Класс Semaphore требует около одной микросекунды при вызове метода WaitOne или Release, а класс SemaphoreSlim — примерно одну десятую этого времени.

Семафоры могут оказаться удобными для ограничения параллелизма, предотвращая выполнение отдельной порции кода слишком большим количеством потоков. В следующем примере пять потоков пытаются войти в ночной клуб, который разрешает вход только трем потокам одновременно:

```
class TheClub // Никаких списков дверей!
{
    static SemaphoreSlim _sem = new SemaphoreSlim (3); // Вместительность равна 3
    static void Main()
    {
        for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);
    }
    static void Enter (object id)
    {
        Console.WriteLine (id + " wants to enter"); // Поток, желающий войти
        _sem.Wait();                                // .
        Console.WriteLine (id + " is in!");           // Одновременно здесь
        Thread.Sleep (1000 * (int) id);              // могут находиться
        Console.WriteLine (id + " is leaving");       // только три потока
        _sem.Release();                            .
    }
}
```

А так выглядит вывод:

```
1 wants to enter
1 is in!
2 wants to enter
```

```
2 is in!
3 wants to enter
3 is in!
4 wants to enter
5 wants to enter
1 is leaving
4 is in!
2 is leaving
5 is in!
```

Также допустимо создать семафор с начальным значением счетчика (емкости), равным 0, а затем вызвать `Release` для увеличения его счетчика. Следующие два семафора эквивалентны:

```
var semaphore1 = new SemaphoreSlim (3);
var semaphore2 = new SemaphoreSlim (0); semaphore2.Release (3);
```

Если объекту `Semaphore` назначено имя, тогда он может охватывать множество процессов тем же способом, что и `Mutex` (именованные объекты `Semaphore` доступны только в Windows, тогда как именованные объекты `Mutex` работают также и в средах Unix).

Асинхронные семафоры и блокировки

Блокировать оператор `await` не разрешено:

```
lock (_locker)
{
    await Task.Delay (1000); // Ошибка на этапе компиляции
    ...
}
```

Поступать так не имеет смысла, потому что блокировки удерживаются потоком, который обычно изменяется при возвращении из `await`. Кроме того, блокировки приводят к **блокированию**, а блокирование в течение потенциально длительного периода времени — это вовсе *не* та цель, к которой вы стремитесь, применяя асинхронные функции.

Однако иногда все-таки желательно выполнять асинхронные операции последовательно или ограничивать параллелизм так, чтобы одновременно выполнялось не более *n* операций. Например, возьмем веб-браузер, который должен выполнять асинхронные загрузки параллельно: он может наложить ограничение, которое устанавливает максимальное количество одновременных загрузок равным 10. Достичь цели можно с использованием объекта `SemaphoreSlim`:

```
SemaphoreSlim _semaphore = new SemaphoreSlim (10);
async Task<byte[]> DownloadWithSemaphoreAsync (string uri)
{
    await _semaphore.WaitAsync ();
    try { return await new WebClient ().DownloadDataTaskAsync (uri); }
    finally { _semaphore.Release (); }
}
```

Уменьшение значения `initialCount` семафора до 1 снижает максимальный параллелизм до единицы, превращая все в асинхронную блокировку.

Написание расширяющего метода EnterAsync

Следующий расширяющий метод упрощает асинхронное применение объекта SemaphoreSlim за счет использования класса Disposable, который был написан в разделе “Анонимное освобождение” главы 12:

```
public static async Task<IDisposable> EnterAsync (this SemaphoreSlim ss)
{
    await ss.WaitAsync().ConfigureAwait (false);
    return Disposable.Create (() => ss.Release());
}
```

С помощью метода EnterAsync мы можем переделать метод DownloadWithSemaphoreAsync, как показано ниже:

```
async Task<byte[]> DownloadWithSemaphoreAsync (string uri)
{
    using (await _semaphore.EnterAsync())
        return await new WebClient().DownloadDataTaskAsync (uri);
}
```

Parallel.ForEachAsync

Начиная с версии .NET 6, доступен еще один подход к ограничению асинхронного параллелизма, предусматривающий использование метода Parallel.ForEachAsync. Предполагая, что в массиве uris находятся URI, подлежащие загрузке, вот как можно загрузить их параллельно, ограничивая при этом параллелизм максимум десятью параллельными загрузками:

```
await Parallel.ForEachAsync (uris,
    new ParallelOptions { MaxDegreeOfParallelism = 10 },
    async (uri, cancelToken) =>
{
    var download = await new HttpClient().GetByteArrayAsync (uri);
    Console.WriteLine ($"Downloaded {download.Length} bytes");
});
```

Остальные методы класса Parallel предназначены для сценариев параллельного программирования (связанных с вычислениями), которые будут описаны в главе 22.

Блокировки объектов чтения/записи

Довольно часто экземпляры типа являются безопасными в отношении потоков для параллельных операций чтения, но не для параллельных обновлений (и не для параллельных операций чтения с обновлением). Это также может быть верным для ресурсов, подобных файлам. Хотя защита экземпляров таких типов посредством простой монопольной блокировки для всех режимов доступа обычно требует ухищрений, она может чрезмерно ограничить параллелизм, когда существует много операций чтения и только несколько операций обновления. Примером, когда такая ситуация может возникнуть, является сервер бизнес-приложений, где часто используемые данные кешируются в статических полях с целью их быстрого извлечения. Класс ReaderWriterLockSlim предназначен для обеспечения блокирования с максимальной доступностью именно в таких сценариях.



Класс `ReaderWriterLockSlim` представляет собой замену более старого “тяжеловесного” класса `ReaderWriterLock`. Класс `ReaderWriterLock` обладает похожей функциональностью, но он в несколько раз медленнее и содержит внутреннюю проектную ошибку в механизме, который отвечает за обработку повышенных уровней блокировок.

Тем не менее, по сравнению с обычным оператором `lock` (`Monitor.Enter/Monitor.Exit`) класс `ReaderWriterLockSlim` все равно работает в два раза медленнее. Компромиссом является меньшая степень соперничества (когда производится много операций чтения и минимум операций записи).

С обоими классами связаны два базовых вида блокировок — блокировка чтения и блокировка записи:

- блокировка записи является универсально монопольной;
- блокировка чтения совместима с другими блокировками чтения.

Следовательно, поток, удерживающий блокировку записи, блокирует все другие потоки, которые пытаются получить блокировку чтения или записи (и наоборот). Но если потоки, удерживающие блокировку записи, отсутствуют, тогда параллельно получить блокировку чтения может любое количество потоков.

В классе `ReaderWriterLockSlim` определены методы для получения и освобождения блокировок чтения/записи:

```
public void EnterReadLock();
public void ExitReadLock();
public void EnterWriteLock();
public void ExitWriteLock();
```

Вдобавок есть версии `Try` всех методов `EnterXXX`, которые принимают аргументы тайм-аута в стиле метода `Monitor.TryEnter` (тайм-ауты могут происходить довольно часто, если ресурс подвержен серьезному соперничеству). Класс `ReaderWriterLock` предлагает аналогичные методы, именуемые `AcquireXXX` и `ReleaseXXX`. Когда случается тайм-аут, вместо возвращения `false` они генерируют исключение `ApplicationException`.

В приведенной далее программе демонстрируется применение класса `ReaderWriterLockSlim`. Три потока постоянно выполняют перечисление списка, в то время как два других потока каждые 100 мс добавляют в список случайное число. Блокировка чтения защищает потоки, читающие список, а блокировка записи — потоки, выполняющие запись в список.

```
class SlimDemo
{
    static ReaderWriterLockSlim _rw = new ReaderWriterLockSlim();
    static List<int> _items = new List<int>();
    static Random _rand = new Random();

    static void Main()
    {
        new Thread (Read).Start();
        new Thread (Write).Start();
    }

    static void Read()
    {
        _rw.EnterReadLock();
        foreach (int item in _items)
            Console.WriteLine(item);
        _rw.ExitReadLock();
    }

    static void Write()
    {
        _rw.EnterWriteLock();
        int item = _rand.Next(1, 100);
        _items.Add(item);
        _rw.ExitWriteLock();
    }
}
```

```

        new Thread (Read) .Start ();
        new Thread (Read) .Start ();
        new Thread (Write) .Start ("A");
        new Thread (Write) .Start ("B");
    }

    static void Read()
    {
        while (true)
        {
            _rw.EnterReadLock ();
            foreach (int i in _items) Thread.Sleep (10);
            _rw.ExitReadLock ();
        }
    }

    static void Write (object threadID)
    {
        while (true)
        {
            int newNumber = GetRandNum (100);
            _rw.EnterWriteLock ();
            _items.Add (newNumber);
            _rw.ExitWriteLock ();
            Console.WriteLine ("Thread " + threadID + " added " + newNumber);
            Thread.Sleep (100);
        }
    }

    static int GetRandNum (int max) { lock (_rand) return _rand.Next (max); }
}

```



В производственный код обычно будут добавляться блоки `try/finally`, гарантирующие освобождение блокировок в случае генерации исключения.

Вот результат:

```

Thread B added 61
Thread A added 83
Thread B added 55
Thread A added 33
...

```

Класс `ReaderWriterLockSlim` делает возможным действие `Read` с большей степенью параллелизма, чем простая блокировка. Это можно проиллюстрировать помещением следующей строки в начало цикла `while` внутри метода `Write`:

```
Console.WriteLine (_rw.CurrentReadCount + " concurrent readers");
```

Данная строка почти всегда будет сообщать о наличии трех параллельных читающих потоков (большую часть своего времени методы `Read` тратят внутри циклов `foreach`). Помимо `CurrentReadCount` класс `ReaderWriterLockSlim` предлагает следующие свойства для слежения за блокировками:

```
public bool IsReadLockHeld          { get; }
public bool IsUpgradeableReadLockHeld { get; }
public bool IsWriteLockHeld         { get; }

public int WaitingReadCount        { get; }
public int WaitingUpgradeCount     { get; }
public int WaitingWriteCount       { get; }

public int RecursiveReadCount      { get; }
public int RecursiveUpgradeCount   { get; }
public int RecursiveWriteCount     { get; }
```

Блокировки с возможностью повышения уровня

Иногда в одиночной атомарной операции блокировку чтения удобно заменять блокировкой записи. Например, предположим, что вы хотите добавлять элемент в список, только если этот элемент в списке отсутствует. В идеальном случае желательно минимизировать время, затрачиваемое на удержание (монопольной) блокировки записи, а потому можно поступить так, как описано ниже.

1. Получить блокировку чтения.
2. Проверить, существует ли элемент в списке; если он существует, тогда освободить блокировку и произвести возврат.
3. Освободить блокировку чтения.
4. Получить блокировку записи.
5. Добавить элемент.

Проблема в том, что между шагом 3 и шагом 4 может проскользнуть другой поток и модифицировать список (например, добавив тот же самый элемент). Класс `ReaderWriterLockSlim` решает такую проблему через блокировку третьего вида, которая называется *блокировкой с возможностью повышения уровня*. Блокировка с возможностью повышения уровня похожа на блокировку чтения за исключением того, что позже она может быть повышенена до уровня блокировки записи в атомарной операции. Вот как ее использовать.

1. Вызвать метод `EnterUpgradeableReadLock`.
2. Выполнить действия, связанные с чтением (например, проверить, существует ли элемент в списке).
3. Вызвать метод `EnterWriteLock` (что преобразует блокировку с возможностью повышения уровня в блокировку записи).
4. Выполнить действия, связанные с записью (например, добавить элемент в список).
5. Вызвать метод `ExitWriteLock` (что преобразует блокировку записи обратно в блокировку с возможностью повышения уровня).
6. Выполнить любые другие действия, связанные с чтением.
7. Вызвать метод `ExitUpgradeableReadLock`.

С точки зрения вызывающего кода все довольно похоже на вложенное или рекурсивное блокирование. Тем не менее, функционально на третьем шаге

`ReaderWriterLockSlim` освобождает блокировку чтения и получает новую блокировку записи атомарным образом.

Между блокировками с возможностью повышения уровня и блокировками чтения имеется еще одно важное отличие. Несмотря на то что блокировка с возможностью повышения уровня способна сосуществовать с любым количеством блокировок чтения, в каждый момент времени может быть получена только одна блокировка с возможностью повышения уровня. Это предотвращает взаимоблокировки преобразований за счет сериализации соперничающих преобразований — почти как в случае блокировок обновлений в SQL Server:

SQL Server	<code>ReaderWriterLockSlim</code>
Совместно используемая блокировка	Блокировка чтения
Монопольная блокировка	Блокировка записи
Блокировка обновления	Блокировка с возможностью повышения уровня

Мы можем продемонстрировать работу блокировки с возможностью повышения уровня, изменив метод `Write` из предыдущего примера так, чтобы он добавлял в список число, только если оно в нем отсутствует:

```
while (true)
{
    int newNumber = GetRandNum (100);
    _rw.EnterUpgradeableReadLock ();
    if (!_items.Contains (newNumber))
    {
        _rw.EnterWriteLock ();
        _items.Add (newNumber);
        _rw.ExitWriteLock ();
        Console.WriteLine ("Thread " + threadID + " added " + newNumber);
    }
    _rw.ExitUpgradeableReadLock ();
    Thread.Sleep (100);
}
```



Класс `ReaderWriterLock` также может выполнять преобразования блокировок, но ненадежно, потому что он не поддерживает концепцию блокировок с возможностью повышения уровня. Именно поэтому проектировщикам класса `ReaderWriterLockSlim` пришлось начинать полностью с нового класса.

Рекурсия блокировок

Обычно вложенное или рекурсивное блокирование с участием класса `ReaderWriterLockSlim` запрещено. Таким образом, следующий код генерирует исключение:

```
var rw = new ReaderWriterLockSlim();
rw.EnterReadLock ();
rw.EnterReadLock ();
rw.ExitReadLock ();
rw.ExitReadLock ();
```

Однако он выполнится без ошибок, если объект `ReaderWriterLockSlim` конструируется так:

```
var rw = new ReaderWriterLockSlim (LockRecursionPolicy.SupportsRecursion);
```

Это гарантирует, что рекурсивное блокирование может произойти, только если оно запланировано. Рекурсивное блокирование может создать нежелательную сложность, т.к. появляется возможность получить более одного вида блокировок:

```
rw.EnterWriteLock();  
rw.EnterReadLock();  
Console.WriteLine (rw.IsReadLockHeld); // True  
Console.WriteLine (rw.IsWriteLockHeld); // True  
rw.ExitReadLock();  
rw.ExitWriteLock();
```

Базовое правило гласит, что после получения блокировки последующие рекурсивные блокировки могут быть меньше, но не больше следующей шкалы:

*Блокировка чтения → Блокировка с возможностью повышения уровня →
Блокировка записи*

Тем не менее, запрос на повышение блокировки с возможностью повышения уровня до блокировки записи законен всегда.

Сигнализирование с помощью дескрипторов ожидания событий

Простейшая разновидность сигнализирующих конструкций называется *дескрипторами ожидания событий* (они никак не связаны с событиями C#). Дескрипторы ожидания событий поступают в трех формах: `AutoResetEvent`, `ManualResetEvent` (`ManualResetEventSlim`) и `CountdownEvent`. Первые две формы основаны на общем классе `EventWaitHandle`, от которого происходит вся их функциональность.

AutoResetEvent

Класс `AutoResetEvent` похож на турникет: вставка билета позволяет пройти в точности одному человеку. Наличие слова “Auto” в имени класса отражает тот факт, что открытый турникет автоматически закрывается, или “сбрасывается”, после того, как кто-то через него прошел. Поток ожидает, или блокируется, на турникете вызовом метода `WaitOne` (ожидает до тех пор, пока этот “один” турникет не откроется), а билет вставляется вызовом метода `Set`. Если метод `WaitOne` вызван несколькими потоками, тогда перед турникетом выстраивается очередь². Билет может поступать из любого потока; другими словами, любой (неблокированный) поток с доступом к объекту `AutoResetEvent` может вызвать на нем метод `Set` для освобождения одного заблокированного потока.

² Как и в случае блокировок, равноправие в такой очереди временами может нарушаться из-за нюансов поведения операционной системы.

Создать объект AutoResetEvent можно двумя способами. Первый из них — применение конструктора:

```
var auto = new AutoResetEvent (false);
```

(Передача конструктору значения true эквивалентна немедленному вызову метода Set на результирующем объекте.) Второй способ создания объекта AutoResetEvent выглядит следующим образом:

```
var auto = new EventWaitHandle (false, EventResetMode.AutoReset);
```

В приведенном далее примере запускается поток, работа которого заключается в том, чтобы просто ожидать, пока он не будет сигнализирован другим потоком (рис. 21.1):

```
class BasicWaitHandle
{
    static EventWaitHandle _waitHandle = new AutoResetEvent (false);

    static void Main()
    {
        new Thread (Waiter).Start();
        Thread.Sleep (1000);                                // Пауза в течение секунды...
        _waitHandle.Set();                                 // Пробудить Waiter.
    }

    static void Waiter()
    {
        Console.WriteLine ("Waiting...");                  // Ожидание...
        _waitHandle.WaitOne();                            // Ожидание уведомления
        Console.WriteLine ("Notified");                   // Уведомлен
    }
}
```

Вот вывод:

```
Waiting... (пауза) Notified.
```

Если метод Set вызван, когда нет ни одного ожидающего потока, то дескриптор остается открытым до тех пор, пока он не дождется вызова метода WaitOne каким-либо потоком. Такое поведение помогает избежать состязаний между потоком, направляющимся к турникуту, и потоком, вставляющим билет.

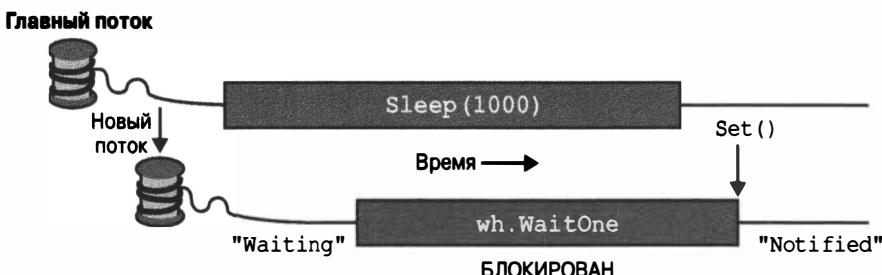


Рис. 21.1. Сигнализирование с помощью EventWaitHandle

Тем не менее, неоднократный вызов `Set` на турнике, перед которым никто не ожидает, не позволяет пройти целой компании, когда она соберется: пройти будет разрешено только следующему человеку, а дополнительные билеты растратятся впустую.

Освобождение дескрипторов ожидания

По завершении работы с дескриптором ожидания можно вызвать его метод `Close`, чтобы освободить ресурс ОС. В качестве альтернативы можно просто удалить все ссылки на дескриптор ожидания и позволить сборщику мусора сделать всю работу в какой-то момент позже (дескрипторы ожидания реализуют шаблон освобождения, в соответствии с которым финализатор вызывает метод `Close`). Это один из немногих сценариев, в которых вполне приемлемо полагаться на такой запасной вариант, потому что с дескрипторами ожидания связаны легковесные накладные расходы ОС.

Дескрипторы ожидания освобождаются автоматически, когда процесс завершается.

Вызов метода `Reset` на объекте `AutoResetEvent` закрывает турникет (если он был открыт) без ожидания или блокирования.

Метод `WaitOne` принимает дополнительный параметр тайм-аута, возвращая `false`, если ожидание закончилось по тайм-ауту, а не из-за получения сигнала.



Вызов метода `WaitOne` с тайм-аутом, равным 0, осуществляет проверку, является ли дескриптор ожидания “открытым”, не блокируя вызывающий поток. Однако помните, что такое действие сбросит объект `AutoResetEvent`, если он открыт.

Двунаправленное сигнализирование

Предположим, что главный поток должен сигнализировать рабочий поток три раза в какой-то строке. Если главный поток просто вызовет метод `Set` на дескрипторе ожидания несколько раз в быстрой последовательности, тогда второй или третий сигнал может потеряться, т.к. рабочему потоку необходимо время на обработку каждого сигнала.

Решение для главного потока предусматривает ожидание перед выдачей сигнала до тех пор, пока рабочий поток не будет готов, что можно сделать посредством еще одного объекта `AutoResetEvent`:

```
class TwoWaySignaling
{
    static EventWaitHandle _ready = new AutoResetEvent (false);
    static EventWaitHandle _go = new AutoResetEvent (false);
    static readonly object _locker = new object ();
    static string _message;

    static void Main()
    {
        new Thread (Work).Start();
    }
```

```

_ready.WaitOne(); // Сначала ожидать готовности рабочего потока
lock (_locker) _message = "ooo";
_go.Set(); // Сообщить рабочему потоку о начале продвижения

_ready.WaitOne();
lock (_locker) _message = "aah"; // Предоставить рабочему потоку
// другое сообщение
_go.Set();

_ready.WaitOne();
lock (_locker) _message = null; // Сигнализировать рабочий поток
// о завершении
_go.Set();

}

static void Work()
{
    while (true)
    {
        _ready.Set(); // Указать на готовность
        _go.WaitOne(); // Ожидать поступления сигнала...
        lock (_locker)
        {
            if (_message == null) return; // Аккуратно завершить
            Console.WriteLine (_message);
        }
    }
}
}

```

Вот вывод:

ooo
aah

На рис. 21.2 процесс представлен визуально.

Здесь сообщение null используется для указания на то, что рабочий поток должен завершиться. Для потоков, которые выполняются бесконечно, очень важно иметь стратегию завершения!

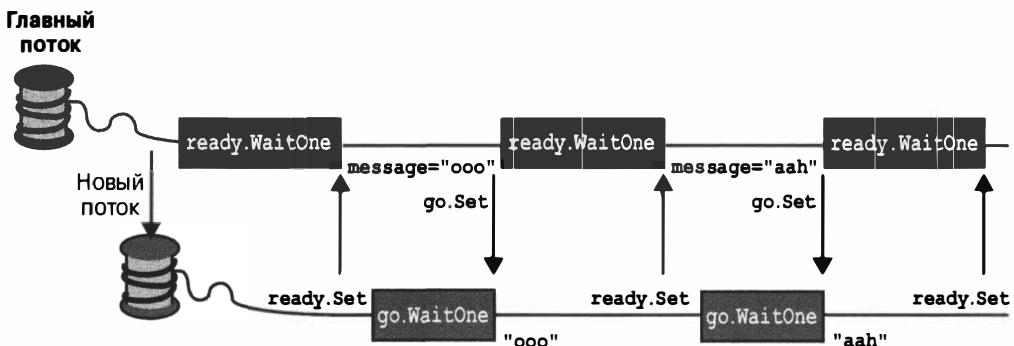


Рис. 21.2. Двунаправленное сигнализирование

ManualResetEvent

Как было описано в главе 14, объект `ManualResetEvent` функционирует подобно простым воротам. Вызов метода `Set` открывает ворота, позволяя *любому* количеству потоков вызывать `WaitOne`, чтобы получить разрешение пройти. Вызов метода `Reset` закрывает ворота. Потоки, которые вызывают `WaitOne` на закрытых воротах, блокируются; когда ворота откроются в следующий раз, все эти потоки будут одновременно освобождены. Помимо упомянутых отличий объект `ManualResetEvent` функционирует подобно объекту `AutoResetEvent`.

Как и `AutoResetEvent`, объект `ManualResetEvent` можно конструировать двумя способами:

```
var manual1 = new ManualResetEvent (false);
var manual2 = new EventWaitHandle (false, EventResetMode.ManualReset);
```



Доступна еще одна версия класса `ManualResetEvent` по имени `ManualResetEventSlim`. Она оптимизирована под краткие периоды ожидания — с возможностью выбора зацикливания для установленного количества итераций. Класс `ManualResetEventSlim` также имеет более эффективную управляемую реализацию и позволяет методу `Wait` быть отмененным через `CancellationToken`. Тем не менее, данный класс не может применяться для межпроцессного сигнализирования. Класс `ManualResetEventSlim` не является подклассом `WaitHandle`; однако он открывает доступ к свойству `WaitHandle`, которое возвращает основанный на `WaitHandle` объект (с профилем производительности традиционного дескриптора ожидания).

Сигнализирующие конструкции и производительность

Ожидание или сигнализирование `AutoResetEvent` либо `ManualResetEvent` занимают около одной микросекунды (при отсутствии блокирования).

Классы `ManualResetEventSlim` и `CountdownEvent` могут быть до 50 раз быстрее в сценариях с кратким ожиданием, поскольку они не зависят от ОС и благородно используют конструкции зацикливания. Тем не менее, в большинстве сценариев накладные расходы, связанные с самими сигнализирующими классами, не создают узких мест, поэтому они редко принимаются во внимание.

Класс `ManualResetEvent` удобен в предоставлении одному потоку возможности разблокировать множество других потоков. Обратный сценарий покрывается классом `CountdownEvent`.

CountdownEvent

Класс CountdownEvent позволяет организовать ожидание на нескольких потоках; он обладает эффективной и целиком управляемой реализацией. Для применения данного класса создайте его экземпляр с нужным количеством потоков, или “счетчиков”, на которых необходимо ожидать:

```
var countdown = new CountdownEvent (3);      // Инициализировать со
                                                // "счетчиком", равным 3
```

Вызов метода `Signal` декрементирует счетчик; вызов метода `Wait` приводит к блокированию до тех пор, пока счетчик не станет равным нулю:

```
new Thread (SaySomething).Start ("I am thread 1");
new Thread (SaySomething).Start ("I am thread 2");
new Thread (SaySomething).Start ("I am thread 3");
countdown.Wait(); // Блокируется до тех пор, пока Signal
// не будет вызван 3 раза
Console.WriteLine ("All threads have finished speaking!");
// Все потоки завершили взаимодействие
void SaySomething (object thing)
{
    Thread.Sleep (1000);
    Console.WriteLine (thing);
    countdown.Signal();
}
```



Задачи, для решения которых эффективно использовать класс `CountdownEvent`, иногда удается решить более просто с применением конструкций *структурированного параллелизма*, которые будут рассматриваться в главе 22 (PLINQ и класс `Parallel`).

Повторно инкрементировать счетчик `CountdownEvent` можно вызовом метода `AddCount`. Однако если он уже достиг нуля, то такой вызов приведет к генерации исключения: “отменить сигнал” `CountdownEvent` вызовом метода `AddCount` нельзя. Чтобы устранить возможность возникновения исключения, можно вызвать метод `TryAddCount`, который возвращает `false`, если счетчик достиг нуля.

Для отмены сигнала `CountdownEvent` необходимо вызвать метод `Reset`: он и отменит сигнал, и сбросит счетчик в исходное значение.

Подобно `ManualResetEventSlim` класс `CountdownEvent` открывает свойство `WaitHandle` для сценариев, в которых какой-то другой класс или метод ожидает объект, основанный на `WaitHandle`.

Создание межпроцессного объекта EventWaitHandle

Конструктор `EventWaitHandle` позволяет “именовать” создаваемый объект `EventWaitHandle`, что дает ему возможность действовать в нескольких процессах. Имя — это просто строка, которая может иметь любое значение, не конфликтующее с именем какого-то другого объекта. Если указанное имя уже используется на данном компьютере, то вы получите ссылку на связанный с ним

объект EventWaitHandle; в противном случае ОС создаст новый объект. Ниже приведен пример:

```
EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.AutoReset,
                                         @"Global\MyCompany.MyApp.SomeName");
```

Если данный код запускают два приложения, то они получат возможность сигнализировать друг друга: дескриптор ожидания будет работать для всех потоков в обоих процессах.

Именованные объекты EventWaitHandle доступны только в Windows.

Дескрипторы ожидания и продолжение

Вместо того чтобы ждать на дескрипторе ожидания (и тем самым блокировать поток), к нему можно присоединить “продолжение”, вызвав метод ThreadPool.RegisterWaitForSingleObject, который принимает делегат, выполняющийся, когда дескриптор ожидания сигнализирован:

```
var starter = new ManualResetEvent (false);
RegisteredWaitHandle reg = ThreadPool.RegisterWaitForSingleObject
    (starter, Go, "Some Data", -1, true);
Thread.Sleep (5000);
Console.WriteLine ("Signaling worker...");
starter.Set();
Console.ReadLine();
reg.Unregister (starter); // Произвести очистку, когда все сделано
void Go (object data, bool timedOut)
{
    Console.WriteLine ("Started - " + data);
    // Выполнить задачу...
}
```

Вот вывод:

```
(пятисекундная задержка)
Signaling worker...
Started - Some Data
```

Когда дескриптор ожидания сигнализируется (либо истекает время тайм-аута), делегат запускается в потоке из пула. Затем понадобится вызвать метод Unregister для освобождения неуправляемого дескриптора обратного вызова.

В дополнение к дескриптору ожидания и делегату метод RegisterWaitForSingleObject принимает объект “черного ящика”, который передается методу делегата (подобно ParameterizedThreadStart), а также длительность тайм-аута в миллисекундах (-1 означает отсутствие тайм-аута) и булевский флаг, указывающий, является запрос одноразовым или повторяющимся.



Надежно вызывать RegisterWaitForSingleObject можно только один раз на дескриптор ожидания. Повторный вызов этого метода на том же самом дескрипторе ожидания приводит к перемежающемуся отказу, из-за чего несигнализированный дескриптор ожидания инициирует обратный вызов, как если бы он был сигнализирован.

По причине такого ограничения дескрипторы ожидания (не Slim) плохо подходят для асинхронного программирования.

WaitAny, WaitAll и SignalAndWait

В дополнение к методам Set, WaitOne и Reset в классе WaitHandle определены статические методы, предназначенные для решения более сложных задач синхронизации. Методы WaitAny, WaitAll и SignalAndWait выполняют операции сигнализирования и ожидания на множество дескрипторов. Дескрипторы ожидания могут быть разных типов (в том числе Mutex и Semaphore, поскольку они также являются производными от абстрактного класса WaitHandle). Классы ManualResetEventSlim и CountdownEvent также могут принимать участие в указанных методах через свои свойства WaitHandle.



Методы WaitAll и SignalAndWait имеют странную связь с унаследованной архитектурой COM: они требуют, чтобы вызывающий поток находился в многопоточном апартаменте — модель, меньше всего подходящая для взаимодействия. Например, в таком режиме главный поток приложения WPF или Windows Forms не может взаимодействовать с буфером обмена. Вскоре мы обсудим доступные альтернативы.

Метод WaitHandle.WaitAny ожидает любой дескриптор ожидания из массива таких дескрипторов, а метод WaitHandle.WaitAll ожидает все указанные дескрипторы атомарным образом. Это означает, что в случае ожидания двух объектов AutoResetEvent:

- метод WaitAny никогда не закончится “зашелкиванием” обоих событий;
- метод WaitAll никогда не закончится “зашелкиванием” только одного события.

Метод SignalAndWait вызывает Set на WaitHandle и затем WaitOne на другом WaitHandle. После сигнализирования первого дескриптора произойдет переход в начало очереди в ожидании второго дескриптора, что помогает ему двигаться вперед (хотя операция не является по-настоящему атомарной). Можете думать об этом методе, как о “подменяющем” один сигнал другим, и применять его на паре объектов EventWaitHandle для настройки двух потоков на randevu, или “встречу”, в одной и той же точке во времени. Такой трюк будет предпринимать либо AutoResetEvent, либо ManualResetEvent. Первый поток выполняет следующий вызов:

```
WaitHandle.SignalAndWait (wh1, wh2);
```

Второй поток делает противоположное:

```
WaitHandle.SignalAndWait (wh2, wh1);
```

Альтернативы методам WaitAll и SignalAndWait

Методы WaitAll и SignalAndWait не будут запускаться в однопоточном апартаменте. К счастью, существуют альтернативы. В случае SignalAndWait редко когда требуется его семантика перехода в начало очереди: скажем, в примере с randevu было бы допустимо просто вызвать Set на первом дескрипторе ожидания и затем WaitOne на втором, если дескрипторы ожидания использо-

вались исключительно для этого randevu. В следующем разделе мы рассмотрим еще один вариант реализации randevu потоков.

В случае методов `WaitAny` и `WaitAll`, если атомарность не нужна, то код, написанный в предыдущем разделе, можно применить для преобразования дескрипторов ожидания в задачи, после чего использовать методы `Task.WhenAny` и `Task.WhenAll` (см. главу 14).

Когда атомарность необходима, можно принять низкоуровневый подход к сигнализированию и самостоятельно написать логику с применением методов `Wait` и `Pulse` класса `Monitor`. Методы `Wait` и `Pulse` детально описаны в статье по ссылке <http://albahari.com/threading/>.

Класс `Barrier`

Класс `Barrier` реализует *барьер выполнения потоков*, позволяя множеству потоков организовать randevu в какой-то момент времени (не путайте его с методом `Thread.MemoryBarrier`). Класс `Barrier` отличается высокой скоростью и эффективностью, а построен он на основе `Wait`, `Pulse` и блокировок на базе счетчиков. Для использования класса `Barrier` потребуется выполнить следующие действия.

1. Создать его экземпляр, указав количество потоков, которые должны принять участие в randevu (позже их число можно изменить, вызывая методы `AddParticipants` и `RemoveParticipants`).
2. Заставить каждый поток вызывать метод `SignalAndWait`, когда он желает участвовать в randevu.

Создание экземпляра `Barrier` со значением 3 приводит к блокированию вызова `SignalAndWait` до тех пор, пока данный метод не будет вызван три раза. Затем все начинается заново: вызов `SignalAndWait` снова блокируется, пока таких вызовов не станет три. Это сохраняет каждый поток синхронным с любым другим потоком.

В приведенном далее примере каждый из трех потоков выводит числа от 0 до 4, не отставая от других потоков:

```
var barrier = new Barrier (3);  
  
new Thread (Speak).Start();  
new Thread (Speak).Start();  
new Thread (Speak).Start();  
void Speak()  
{  
    for (int i = 0; i < 5; i++)  
    {  
        Console.Write (i + " ");  
        barrier.SignalAndWait();  
    }  
}
```

Вот вывод:

```
0 0 0 1 1 1 2 2 2 3 3 3 4 4 4
```

По-настоящему полезная характеристика Barrier связана с возможностью указывать во время создания экземпляра также действие, выполняемое после каждой фазы. Такое действие представлено в виде делегата, который запускается после того, как метод SignalAndWait будет вызван *n* раз, но перед тем, как потоки деблокируются (как показано в затененной области на рис. 21.3). Если в рассматриваемом примере создать барьер следующим образом:

```
static Barrier _barrier = new Barrier (3, barrier => Console.WriteLine());
```

то вывод будет выглядеть так:

```
0 0 0  
1 1 1  
2 2 2  
3 3 3  
4 4 4
```

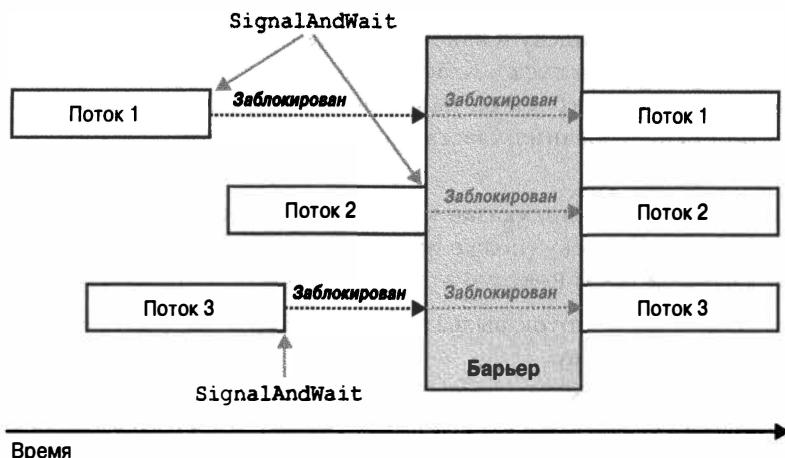


Рис. 21.3. Барьер

Действие, выполняемое после каждой фазы, может быть удобно для объединения данных из каждого рабочего потока. Беспокоиться о вытеснении не придется, потому что во время выполнения данного действия все рабочие потоки заблокированы.

Ленивая инициализация

Частой проблемой в области многопоточности является определение способа ленивой инициализации совместно используемого поля в манере, безопасной к потокам. Такая потребность возникает при наличии поля, которое относится к типу, затратному в плане конструирования:

```
class Foo  
{  
    public readonly Expensive Expensive = new Expensive();  
    ...  
}  
class Expensive { /* Предположим, что это является затратным  
   в конструировании */ }
```

Проблема с показанным кодом заключается в том, что создание экземпляра Foo оказывает влияние на производительность из-за создания экземпляра класса Expensive, причем независимо от того, будет позже осуществляться доступ к полю Expensive или нет. Очевидное решение предусматривает конструирование экземпляра *по требованию*:

```
class Foo
{
    Expensive _expensive;
    public Expensive Expensive // Ленивое создание экземпляра Expensive
    {
        get
        {
            if (_expensive == null) _expensive = new Expensive();
            return _expensive;
        }
    }
    ...
}
```

Здесь возникает вопрос: является ли такой код безопасным в отношении потоков? Оставив в стороне тот факт, что доступ к _expensive производится за пределами блокировки без барьера памяти, давайте подумаем, что произойдет, если два потока обратятся к данному свойству одновременно. Они оба могут дать true в условии оператора if, и каждый поток в конечном итоге получит отличающийся экземпляр Expensive. Поскольку это может привести к возникновению тонких ошибок, в общем можно было бы сказать, что код не является безопасным к потокам.

Упомянутая проблема решается применением блокировки к коду проверки и инициализации объекта:

```
Expensive _expensive;
readonly object _expenseLock = new object();

public Expensive Expensive
{
    get
    {
        lock (_expenseLock)
        {
            if (_expensive == null) _expensive = new Expensive();
            return _expensive;
        }
    }
}
```

Lazy<T>

Класс Lazy<T> помогает обеспечивать ленивую инициализацию. В случае создания его экземпляра с аргументом true он реализует только что описанный шаблон инициализации, безопасной в отношении потоков.



Класс `Lazy<T>` на самом деле реализует микрооптимизированную версию этого шаблона, которая называется *блокированием с двойным контролем*. Блокирование с двойным контролем выполняет дополнительное временное (*volatile*) чтение, чтобы избежать затрат на получение блокировки, если объект уже инициализирован.

Для использования `Lazy<T>` создайте его экземпляр с делегатом фабрики значений, который сообщает, каким образом инициализировать новое значение, и аргументом `true`. Затем получайте доступ к его значению через свойство `Value`:

```
Lazy<Expensive> _expensive = new Lazy<Expensive>
    (() => new Expensive(), true);
public Expensive Expensive { get { return _expensive.Value; } }
```

Если конструктору класса `Lazy<T>` передать `false`, тогда он реализует шаблон ленивой инициализации, небезопасной к потокам, который был описан в начале настоящего раздела — это имеет смысл, когда класс `Lazy<T>` необходимо применять в однопоточном контексте.

LazyInitializer

`LazyInitializer` — статический класс, который работает в точности как `Lazy<T>` за исключением перечисленных ниже моментов.

- Его функциональность открыта через статический метод, который оперирует прямо на поле вашего типа, что позволяет избежать дополнительного уровня косвенности, улучшая производительность в ситуациях, когда нужна высшая степень оптимизации.
- Он предлагает другой режим инициализации, при котором множество потоков могут состязаться за инициализацию.

Чтобы использовать класс `LazyInitializer`, перед доступом к полю необходимо вызвать его метод `EnsureInitialized`, передав ему ссылку на поле и фабричный делегат:

```
Expensive _expensive;
public Expensive Expensive
{
    get // Реализовать блокирование с двойным контролем
    {
        LazyInitializer.EnsureInitialized (ref _expensive,
                                            () => new Expensive());
        return _expensive;
    }
}
```

Можно также передать еще один аргумент, чтобы запросить *состязание* за инициализацию конкурирующих потоков. Это звучит подобно исходному небезопасному к потокам примеру, исключая то, что первый пришедший к финишу поток всегда выигрывает — и потому в конечном итоге остается только один экземпляр. Преимущество такого приема связано с тем, что

он даже быстрее (на многоядерных процессорах), чем блокирование с двойным контролем. Причина в том, что он может быть реализован полностью без блокировок с применением расширенных технологий, которые описаны в разделах “Nonblocking Synchronization” (“Неблокирующая синхронизация”) и “Lazy Initialization” (“Ленивая инициализация”) в статье по ссылке <http://albahari.com/threading/>. Это предельная (и редко востребованная) степень оптимизации, за которую придется заплатить определенную цену, как описано ниже.

- Такой подход будет медленнее, когда потоков, состязающихся за инициализацию, оказывается больше, чем ядер процессора.
- Потенциально он приводит к непроизводительным расходам ресурсов центрального процессора на выполнение избыточной инициализации.
- Логика инициализации обязана быть безопасной к потокам (в рассмотренном выше примере она может стать небезопасной к потокам, если конструктор `Expensive` будет производить запись в статические поля).
- Если инициализатор создает объект, требующий освобождения, то ставший “ненужным” такой объект не сможет быть освобожден без написания дополнительной логики.

Локальное хранилище потока

Большая часть главы сосредоточена на конструкциях синхронизации и проблемах, возникающих из-за наличия у потоков возможности параллельного доступа к одним и тем же данным. Однако иногда данные должны храниться изолированно, гарантируя тем самым, что каждый поток имеет их отдельную копию. Именно этого позволяют добиться локальные переменные, но они пригодны только для переходных данных.

Решением является *локальное хранилище потока*. Здесь может возникнуть затруднение с пониманием требования: данные, которые вы хотели бы сохранить изолированными в потоке, как правило, являются переходными по своей природе. Основное использование локального хранилища касается хранения “внешних” данных, с помощью которых осуществляется поддержка инфраструктуры пути выполнения, такой как обмен сообщениями, транзакция и маркеры безопасности. Передача подобного рода данных в параметрах методов может оказаться неудобным и чуждым приемом для всех методов кроме написанных лично вами. С другой стороны, хранение такой информации в обычных статических полях означает ее совместное использование всеми потоками.

Локальное хранилище потока может также быть полезным при оптимизации параллельного кода. Оно позволяет каждому потоку иметь монопольный доступ к собственной версии объекта, небезопасного к потокам, без необходимости в блокировке — и без потребности в воссоздании этого объекта между вызовами методов.

Существуют четыре способа реализации локального хранилища потока, которые обсуждаются в последующих разделах.

[ThreadStatic]

Простейший подход к реализации локального хранилища потока предусматривает пометку статического поля с помощью атрибута [ThreadStatic]:

```
[ThreadStatic] static int _x;
```

После этого каждый поток будет видеть отдельную копию `_x`.

К сожалению, атрибут [ThreadStatic] не работает с полями экземпляра (он просто ничего не делает), а также не сочетается нормально с инициализаторами полей — в функционирующем потоке они выполняются только один раз, когда запускается статический конструктор. Если необходимо работать с полями экземпляра или начать с нестандартного значения, то более подходящим вариантом является `ThreadLocal<T>`.

ThreadLocal<T>

Класс `ThreadLocal<T>` предоставляет локальное хранилище потока для статических полей и для полей экземпляра, а также позволяет указывать стандартные значения.

Вот как создать объект `ThreadLocal<int>` со стандартным значением 3 для каждого потока:

```
static ThreadLocal<int> _x = new ThreadLocal<int> (() => 3);
```

Далее для получения или установки значения, локального для потока, применяется свойство `Value` объекта `_x`. Дополнительным преимуществом использования `ThreadLocal` является ленивая оценка значений: фабричная функция оценивается только при первом ее вызове (для каждого потока).

ThreadLocal<T> и поля экземпляра

Класс `ThreadLocal<T>` также удобен при работе с полями экземпляра и захваченными локальными переменными. Например, рассмотрим задачу генерации случайных чисел в многопоточной среде. Класс `Random` не является безопасным в отношении потоков, поэтому мы должны либо применять блокировку вокруг кода, использующего `Random` (ограничивая степень параллелизма), либо генерировать отдельный объект `Random` для каждого потока. Класс `ThreadLocal<T>` делает второй подход простым:

```
var localRandom = new ThreadLocal<Random> () => new Random();
Console.WriteLine (localRandom.Value.Next());
```

Указанная фабричная функция, создающая объект `Random`, несколько упрощена, т.к. конструктор без параметров класса `Random` при выборе начального значения для генерации случайных чисел полагается на системные часы. Начальные значения могут оказаться одинаковыми для двух объектов `Random`, созданных внутри приблизительно 10 мс промежутка времени. Ниже продемонстрирован один из способов решения проблемы:

```
var localRandom = new ThreadLocal<Random>
( () => new Random (Guid.NewGuid().GetHashCode()) );
```

Мы будем использовать такой прием в главе 22 (см. пример параллельной программы проверки орфографии в разделе “PLINQ”).

GetData и SetData

Третий подход предполагает применение двух методов класса Thread: GetData и SetData. Они сохраняют данные в “ячейках”, специфичных для потока. Метод Thread.GetData выполняет чтение из изолированного хранилища данных потока, а метод Thread.SetData осуществляет запись в него. Оба метода требуют объекта LocalDataStoreSlot для идентификации ячейки. Одна и та же ячейка может использоваться во всех потоках, но они по-прежнему будут получать отдельные значения. Ниже приведен пример:

```
class Test
{
    //Один и тот же объект LocalDataStoreSlot может использоваться во всех потоках
    LocalDataStoreSlot _secSlot = Thread.GetNamedDataSlot ("securityLevel");

    // Это свойство имеет отдельное значение в каждом потоке.
    int SecurityLevel
    {
        get
        {
            object data = Thread.GetData (_secSlot);
            return data == null ? 0 : (int) data; // null == не инициализировано
        }
        set { Thread.SetData (_secSlot, value); }
    }
    ...
}
```

В показанном примере мы вызываем метод Thread.GetNamedDataSlot, который создает именованную ячейку — это позволяет разделять данную ячейку в рамках всего приложения. В качестве альтернативы можно самостоятельно управлять областью видимости ячейки посредством неименованной ячейки, получаемой с помощью вызова метода Thread.AllocateDataSlot:

```
class Test
{
    LocalDataStoreSlot _secSlot = Thread.AllocateDataSlot();
    ...
}
```

Метод Thread.FreeNamedDataSlot освободит именованную ячейку данных во всех потоках, но только если все ссылки на объект LocalDataStoreSlot покинули области видимости и были обработаны сборщиком мусора. Это гарантирует, что потоки не потеряют свои ячейки данных, т.к. они хранят ссылку на соответствующий объект LocalDataStoreSlot, пока ячейка нужна.

AsyncLocal<T>

Рассмотренные до сих пор подходы к реализации локального хранилища потока несовместимы с асинхронными функциями, потому что после await выполнение может возобновиться в другом потоке. Проблему решает класс AsyncLocal<T> за счет предохранения своего значения через await:

```

static AsyncLocal<string> _asyncLocalTest = new AsyncLocal<string>();
async void Main()
{
    _asyncLocalTest.Value = "test";
    await Task.Delay (1000);
    // Следующий оператор работает, даже если мы возвратились из другого потока:
    Console.WriteLine (_asyncLocalTest.Value); // test
}

```

Класс AsyncLocal<T> по-прежнему способен сохранять операции, запущенные в разных потоках, обособленно вне зависимости от того, инициированы они вызовом Thread.Start или Task.Run. Следующий код выводит one one и two two:

```

static AsyncLocal<string> _asyncLocalTest = new AsyncLocal<string>();
void Main()
{
    // Вызвать Test два раза в двух параллельных потоках:
    new Thread (() => Test ("one")).Start();
    new Thread (() => Test ("two")).Start();
}
async void Test (string value)
{
    _asyncLocalTest.Value = value;
    await Task.Delay (1000);
    Console.WriteLine (value + " " + _asyncLocalTest.Value);
}

```

С классом AsyncLocal<T> связан интересный и уникальный нюанс: если объект AsyncLocal<T> уже имеет значение, когда поток запускается, то новый поток “унаследует” это значение:

```

static AsyncLocal<string> _asyncLocalTest = new AsyncLocal<string>();
void Main()
{
    _asyncLocalTest.Value = "test";
    new Thread (AnotherMethod).Start();
}

void AnotherMethod() => Console.WriteLine (_asyncLocalTest.Value); // test

```

Тем не менее, новый поток получает копию значения, так что любые вносимые им изменения не будут влиять на исходное значение:

```

static AsyncLocal<string> _asyncLocalTest = new AsyncLocal<string>();
void Main()
{
    _asyncLocalTest.Value = "test";
    var t = new Thread (AnotherMethod);
    t.Start(); t.Join();
    Console.WriteLine (_asyncLocalTest.Value); // test (не ha-ha!)
}
void AnotherMethod() => _asyncLocalTest.Value = "ha-ha!";

```

Имейте в виду, что новый поток получает *поверхностную* копию значения. Таким образом, если бы вы заменили Async<string> классом Async<StringBuilder> или Async<List<string>>, то новый поток мог бы

очищать `StringBuilder` или добавлять/удалять значения в `List<string>` и это не повлияло бы на оригинал.

Таймеры

Если некоторый метод необходимо выполнять многократно через регулярные интервалы, то проще всего прибегнуть к помощи *таймера*. Таймеры удобны и эффективны в плане использования ими памяти и других ресурсов, если сравнивать их с такими приемами, как показанный ниже:

```
new Thread (delegate() {
    while (enabled)
    {
        DoSomeAction();
        Thread.Sleep (TimeSpan.FromHours (24));
    }
}).Start();
```

Здесь не только надолго связывается ресурс потока, но без написания дополнительного кода метод `DoSomeAction` будет вызываться в более позднее время каждый день. Проблемы подобного рода решаются с помощью таймеров.

В .NET предлагаются пять таймеров. Два из них являются универсальными многопоточными таймерами:

- `System.Threading.Timer`
- `System.Timers.Timer`

Еще два представляют собой специализированные однопоточные таймеры:

- `System.Windows.Forms.Timer` (таймер Windows Forms)
- `System.Windows.Threading.DispatcherTimer` (таймер WPF)

Многопоточные таймеры характеризуются большей мощностью, точностью и гибкостью; однопоточные таймеры безопаснее и удобнее для запуска простых задач, которые обновляют элементы управления Windows Forms либо элементы WPF. Наконец, начиная с версии .NET 6, доступен вариант `PeriodicTimer`, который будет рассматриваться первым.

PeriodicTimer

На самом деле `PeriodicTimer` не является таймером; это класс, помогающий с организацией асинхронных циклов. Важно учитывать, что после появления `async` и `await` традиционные таймеры обычно не востребованы. Взамен хорошо подходит следующий шаблон:

```
StartPeriodicOperation();

async void StartPeriodicOperation()
{
    while (true)
    {
        await Task.Delay (1000);
        Console.WriteLine ("Tick"); // Выполнить какое-то действие
    }
}
```



Если вызвать метод `StartPeriodicOperation` из потока пользователяского интерфейса, то он будет вести себя как однопоточный таймер, поскольку `await` всегда осуществляет возврат в тот же самый контекст синхронизации.

Его можно заставить работать как многопоточный таймер, просто добавив `.ConfigureAwait(false)` к `await`.

С помощью класса `PeriodicTimer` этот шаблон можно упростить:

```
var timer = new PeriodicTimer (TimeSpan.FromSeconds (1));
StartPeriodicOperation();
// Необязательно освобождать таймер, когда необходимо остановить цикл
async void StartPeriodicOperation()
{
    while (await timer.WaitForNextTickAsync ())
        Console.WriteLine ("Tick");           // Выполнить какое-то действие
}
```

Класс `PeriodicTimer` также позволяет остановить таймер, освободив экземпляр таймера. В результате метод `WaitForNextTickAsync` возвращает `false`, позволяя циклу завершиться.

Многопоточные таймеры

Класс `System.Threading.Timer` представляет простейший многопоточный таймер: он имеет только конструктор и два метода (предмет восхищения для минималистов, к которым себя относит и автор книги). В следующем примере таймер вызывает метод `Tick`, который выводит строку `tick...` спустя пять секунд и затем ежесекундно, пока пользователь не нажмет клавишу `<Enter>`:

```
using System;
using System.Threading;
// Первый интервал составляет 5000 мс; последующие интервалы - 1000 мс
Timer tmr = new Timer (Tick, "tick...", 5000, 1000);
Console.ReadLine();
tmr.Dispose(); // Это останавливает таймер и производит очистку.
void Tick (object data)
{
    // Это запускается в потоке из пула
    Console.WriteLine (data);           // Выводит tick...
}
```



Обсуждение освобождения многопоточных таймеров можно найти в разделе “Таймеры” главы 12.

Позже интервал таймера можно изменить, вызвав его метод `Change`. Если нужно, чтобы таймер запустился только раз, тогда в последнем аргументе конструктора следует указать `Timeout.Infinite`.

В .NET предоставляется еще один класс таймера с тем же именем, но в пространстве имен `System.Timers`. Это просто оболочка для `System.Threading.Timer`, которая предлагает дополнительные удобства, однако имеет идентичный внутренний механизм. Ниже приведена сводка по добавленным возможностям:

- реализация интерфейса `IComponent`, которая позволяет классу находиться в панели компонентов визуального редактора Visual Studio;
- свойство `Interval` вместо метода `Change`;
- событие `Elapsed` вместо делегата обратного вызова;
- свойство `Enabled` для запуска и останова таймера (стандартным значением является `false`);
- методы `Start` и `Stop` на тот случай, если вам не нравится работать со свойством `Enabled`;
- флаг `AutoReset` для указания повторяющегося события (стандартным значением является `true`);
- свойство `SynchronizingObject` с методами `Invoke` и `BeginInvoke` для безопасного вызова методов на элементах WPF и элементах управления Windows Forms.

Рассмотрим пример:

```
using System;
using System.Timers;           // Пространство имен Timers, а не Threading
var tmr = new Timer();          // Не требует никаких аргументов
tmr.Interval = 500;             // Использует событие вместо делегата
tmr.Elapsed += tmr_Elapsed;     // Запустить таймер
tmr.Start();
Console.ReadLine();
tmr.Stop();                    // Остановить таймер
Console.ReadLine();
tmr.Start();                   // Запустить таймер повторно
Console.ReadLine();
tmr.Dispose();                 // Остановить таймер навсегда
void tmr_Elapsed (object sender, EventArgs e)
=> Console.WriteLine ("Tick");
```

Многопоточные таймеры применяют пул потоков, чтобы позволить нескольким потокам обслуживать множество таймеров. Это означает, что метод обратного вызова или событие `Elapsed` может инициироваться каждый раз в новом потоке, когда к нему производится обращение. Кроме того, событие `Elapsed` всегда инициируется (приблизительно) вовремя — независимо от того, завершило ли выполнение предыдущее событие `Elapsed`. Следовательно, обратные вызовы или обработчики событий должны быть безопасными в отношении потоков.

Точность многопоточных таймеров зависит от ОС и обычно находится в диапазоне 10–20 мс. Если нужна более высокая точность, тогда можете прибегнуть к собственному взаимодействию и обратиться к мультимедиа-таймеру Windows. Его точность достигает одной миллисекунды, а сам он определен в сборке `winmm.dll`. Сначала вызовите функцию `timeBeginPeriod`, чтобы проинформировать ОС о том, что необходима высокая точность измерения времени, а затем обратитесь к функции `timeSetEvent` для запуска мультимедиа-таймера. По завершении работы вызовите функцию `timeKillEvent`, чтобы остановить таймер, и функцию `timeEndPeriod` для сообщения ОС о том, что высокая точность измерения времени больше не нужна. Вызов внешних мето-

дов с помощью P/Invoke демонстрируется в главе 24. Полнозначные примеры работы с мультимедиа-таймером можно найти в Интернете, выполнив поиск по ключевым словам `dllimport winmm.dll timesetevent`.

Однопоточные таймеры

.NET предлагает таймеры, которые предназначены для устранения проблем с безопасностью к потокам в приложениях WPF и Windows Forms:

- `System.Windows.Threading.DispatcherTimer` (WPF)
- `System.Windows.Forms.Timer` (Windows Forms)



Однопоточные таймеры не проектировались для работы за пределами соответствующих сред. Например, если попытаться использовать таймер Windows Forms в приложении Windows Service, то даже не будет инициировано событие таймера!

Оба однопоточных таймера похожи на `System.Timers.Timer` в плане открытых членов — `Interval`, `Start` и `Stop` (а также `Tick`, который эквивалентен `Elapsed`) — и применяются в аналогичной манере. Однако они отличаются своей внутренней работой. Вместо запуска событий таймера в потоках из пула они отправляют события циклу сообщений WPF или Windows Forms. В результате событие `Tick` всегда инициируется в том же самом потоке, который первоначально создал таймер — в нормальном приложении это поток, используемый для управления всеми элементами пользовательского интерфейса. Такой подход обеспечивает несколько преимуществ:

- вы можете вообще забыть о безопасности к потокам;
- новый вызов `Tick` никогда не будет инициирован до тех пор, пока предыдущий вызов `Tick` не завершит обработку;
- обновлять элементы управления пользовательского интерфейса можно напрямую из кода обработки события `Tick`, не вызывая `Control.BeginInvoke` или `Dispatcher.BeginInvoke`.

Таким образом, программа, эксплуатирующая такие таймеры, в действительности не является многопоточной: в итоге получается та же разновидность псевдопараллелизма, которая была описана в главе 14 при рассмотрении асинхронных функций, выполняющихся в потоке пользовательского интерфейса. Один поток обслуживает все таймеры — равно как и обрабатывает события пользовательского интерфейса. Это значит, что обработчик события `Tick` должен выполняться быстро, иначе пользовательский интерфейс перестанет быть отзывчивым.

Следовательно, таймеры WPF и Windows Forms подходят для выполнения небольших работ, обычно связанных с обновлением какого-то аспекта пользовательского интерфейса (например, часов или счетчика с обратным отсчетом).

В терминах точности однопоточные таймеры похожи на многопоточные таймеры (десятки миллисекунд), хотя они обычно менее точны, поскольку могут задерживаться на время, пока обрабатываются другие запросы пользовательского интерфейса (или другие события таймеров).



Параллельное программирование

В настоящей главе будут раскрыты многопоточные API-интерфейсы и конструкции, нацеленные на использование преимуществ многоядерных процессоров:

- параллельный LINQ (Parallel LINQ), или *PLINQ*;
- класс *Parallel*;
- конструкции *параллелизма задач*;
- *параллельные коллекции*.

Все конструкции вместе известны под (свободным) названием PFX (Parallel Framework — параллельная инфраструктура). Класс *Parallel* и конструкции параллелизма задач называют *библиотекой параллельных задач* (Task Parallel Library — TPL).

Чтение главы требует знания основ, изложенных в главе 14, в частности блокирования, безопасности к потокам и класса *Task*.



.NET предлагает несколько дополнительных специализированных API-интерфейсов для параллельного и асинхронного программирования.

- *System.Threading.Channels.Channel* — высокопроизводительная асинхронная очередь производителей/потребителей, появившаяся в .NET Core 3.
- *Microsoft Dataflow* (из пространства имен *System.Threading.Tasks.Dataflow*) представляет собой сложно устроенный API-интерфейс для создания сетей буферизированных блоков, которые выполняют действия или трансформации данных параллельно, напоминая программирование с использованием акторов/агентов.
- *Reactive Extensions (Rx)* реализует LINQ поверх *IObservable* (альтернативная абстракция *IAsyncEnumerable*) и прекрасно справляется с объединением асинхронных потоков. *Reactive Extensions* поставляется в NuGet-пакете *System.Reactive*.

Для чего нужна инфраструктура PFX?

На протяжении последних 15 лет производители центральных процессоров (ЦП) перешли с одноядерной архитектуры на многоядерную. В результате создается дополнительная проблема для нас как программистов, поскольку однопоточный код не будет автоматически выполняться быстрее только по причине наличия дополнительных ядер.

Использовать в своих интересах множество ядер довольно легко в большинстве серверных приложений, где каждый поток может независимо обрабатывать отдельный клиентский запрос, но труднее в настольных приложениях, т.к. обычно это требует применения к коду с интенсивными вычислениями следующих действий.

1. *Разбиение* кода на небольшие части.
2. Выполнение частей кода параллельно через многопоточность.
3. *Объединение* результатов по мере их получения в безопасной к потокам и высокопроизводительной манере.

Хотя все указанные действия можно реализовать с помощью классических многопоточных конструкций, выполнять их довольно утомительно — особенно шаги разбиения и объединения. Еще одна проблема связана с тем, что обычная стратегия блокирования для обеспечения безопасности в отношении потоков приводит к большому числу состязаний, когда множество потоков одновременно работают с одними и теми же данными.

Библиотеки PFX были спроектированы специально для оказания помощи в сценариях подобного рода.



Программирование с целью получения выгоды от множества ядер или процессоров называют *параллельным программированием*. Оно представляет собой подмножество более широкой концепции многопоточности.

Концепции PFX

Существуют две стратегии разбиения работы между потоками: *параллелизм данных* и *параллелизм задач*.

Когда набор задач должен быть выполнен над множеством значений данных, мы можем распараллелить работу, заставив каждый поток выполнять (тот же самый) набор задач на подмножестве значений. Это называется *параллелизмом данных*, потому что мы распределяем *данные* между потоками. Напротив, при *параллелизме задач* мы распределяем *задачи*; другими словами, заставляем каждый поток выполнять свою задачу.

В общем случае параллелизм данных реализуется легче и масштабируется лучше для оборудования с высокой степенью параллелизма, т.к. он сокращает или устраняет совместно используемые данные (и тем самым сводит к минимуму проблемы, связанные с состязаниями и безопасностью к потокам). Кроме того, параллелизм данных опирается на тот факт, что значений данных часто имеется больше, чем дискретных задач, увеличивая в итоге потенциал параллелизма.

Параллелизм данных также способствует *структурированному параллелизму*, который означает, что параллельные единицы работы начинаются и завершаются в одном и том же месте внутри программы. В отличие от него параллелизм задач имеет тенденцию быть неструктурированным, т.е. параллельные единицы работы могут начинаться и завершаться в разных местах, разбросанных по программе. Структурированный параллелизм проще, менее подвержен ошибкам и позволяет поручить выполнение сложной работы по разбиению и координации потоков (и даже объединение результатов) библиотекам.

Компоненты PFX

Как показано на рис. 22.1, инфраструктура PFX содержит два уровня функциональности. Верхний уровень состоит из двух API-интерфейсов *структурированного параллелизма данных*: PLINQ и класс Parallel. Нижний уровень включает классы параллелизма задач, а также набор дополнительных конструкций, помогающих выполнять действия параллельного программирования.

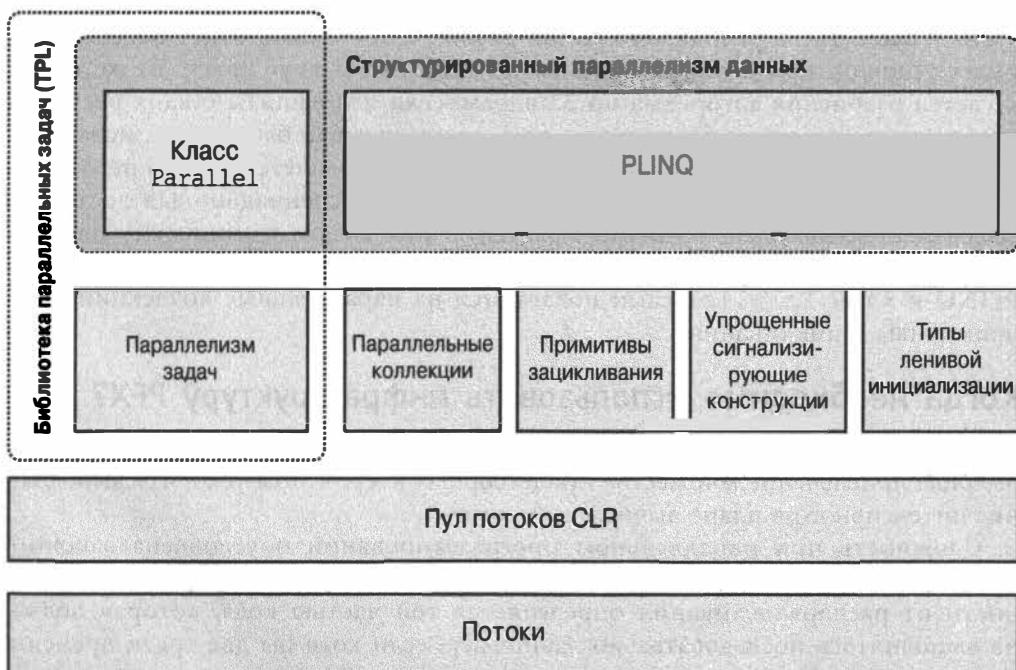


Рис. 22.1. Компоненты PFX

Язык PLINQ предлагает самую развитую функциональность: он автоматизирует все шаги по распараллеливанию — включая разбиение работы на задачи, выполнение этих задач в потоках и объединение результатов в единственную выходную последовательность. Он называется *декларативным*, поскольку вы просто декларируете, что хотите распараллелить свою работу (структурируя ее как запрос LINQ), и позволяете исполняющей среде позаботиться о деталях реализации. Напротив, другие подходы являются *императивными* в том,

что вы должны явно писать код для разбиения и объединения. В случае класса Parallel вам придется объединять результаты самостоятельно; имея дело с конструкциями параллелизма задач, придется реализовывать самостоятельно также и разбиение работы.

	Разбивает работу	Объединяет результаты
PLINQ	Да	Да
Класс Parallel	Да	Нет
Параллелизм задач PFX	Нет	Нет

Параллельные коллекции и примитивы зацикливания помогают справиться с действиями параллельного программирования нижнего уровня. Они важны из-за того, что инфраструктура PFX спроектирована для работы не только с современным оборудованием, но и с будущими поколениями процессоров с гораздо большим числом ядер. Если вы хотите перенести штабель бревен и для этого у вас есть 32 рабочих, то самой сложной проблемой будет обеспечение таких условий, при которых рабочие не мешали бы друг другу. То же самое касается разбиения алгоритма по 32 ядрам: если для защиты общих ресурсов применяются обычные блокировки, то результирующая блокировка может означать, что на самом деле одновременно занятыми является только некоторая доля ядер. Параллельные коллекции настраиваются специально для доступа с высокой степенью параллелизма, преследуя цель свести к минимуму или вообще устраниить блокирование. Для эффективного управления работой язык PLINQ и класс Parallel сами полагаются на параллельные коллекции и на примитивы зацикливания.

Когда необходимо использовать инфраструктуру PFX?

Основным сценарием использования PFX является *параллельное программирование*: привлечение множества процессорных ядер, чтобы ускорить выполнение интенсивного в плане вычислений кода.

Сложность при параллельном программировании обусловлена законом Амдала, который утверждает, что максимальное улучшение производительности от распараллеливания определяется той частью кода, которая должна выполняться последовательно. Например, если хотя бы две трети времени выполнения алгоритма поддаются распараллеливанию, то никогда не удастся превысить трехкратный выигрыш в производительности — даже при неограниченном числе ядер.

Таким образом, прежде чем продолжать, полезно проверить, что узкое место находится в распараллеливаемом коде. Также имеет смысл обдумать, должен ли ваш код действительно быть интенсивным в плане вычислений — часто простейшим и наиболее эффективным подходом будет оптимизация. Тем не менее, следует соблюдать компромисс, потому что некоторые технологии оптимизации могут затруднить распараллеливание кода.

Конструкции параллельного программирования полезны не только для работы с многоядерными процессорами, но также и в других сценариях.

- Параллельные коллекции иногда подходят, когда нужна безопасная к потокам очередь, стек или словарь.
 - Класс `BlockingCollection` предоставляет простые средства для реализации структур производителей/потребителей и является хорошим способом ограничения параллелизма.
 - Задачи являются основой асинхронного программирования, как было показано в главе 14.
-

Самый простой выигрыш получается с так называемыми *естественно параллельными* случаями — когда работа может быть легко разбита на задачи, которые сами по себе выполняются эффективно (здесь очень хорошо подходит структурированный параллелизм). Примеры включают многие задачи обработки изображений, метод трассировки лучей и прямолинейные подходы в математике или криптографии. Примером неестественной параллельной задачи может считаться реализация оптимизированной версии алгоритма быстрой сортировки — хороший результат требует некоторых размышлений и возможно неструктурированного параллелизма.

PLINQ

Инфраструктура PLINQ автоматически распараллеливает локальные запросы LINQ. Преимущество PLINQ заключается в простоте использования, т.к. ответственность за выполнение работ по разбиению и объединению результатов перекладывается на исполняющую среду .NET.

Для применения PLINQ просто вызовите метод `AsParallel` на входной последовательности и затем продолжайте запрос LINQ обычным образом. Приведенный ниже запрос вычисляет простые числа между 3 и 100 000, обеспечивая полную загрузку всех ядер процессора на целевой машине:

```
//Вычислить простые числа с использованием простого (неоптимизированного) алгоритма
IEnumerable<int> numbers = Enumerable.Range (3, 100000-3);
var parallelQuery =
    from n in numbers.AsParallel()
    where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
    select n;
int[] primes = parallelQuery.ToArray();
```

`AsParallel` представляет собой расширяющий метод в классе `System.Linq.ParallelEnumerable`. Он помещает входные данные в оболочку последовательности, основанной на `ParallelQuery<TSource>`, что вызывает привязку последующих операций запросов LINQ к альтернативному набору расширяю-

щих методов, которые определены в классе `ParallelEnumerable`. Они представляют параллельные реализации для всех стандартных операций запросов. По существу они разбивают входную последовательность на порции, которые выполняются в разных потоках, и объединяют результаты снова в единственную выходную последовательность для дальнейшего потребления, как иллюстрируется на рис. 22.2.

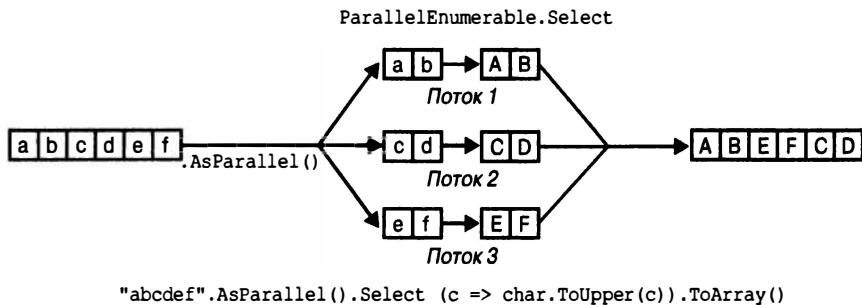


Рис. 22.2. Модель выполнения PLINQ

Вызов метода `AsSequential` извлекает последовательность из оболочки `ParallelQuery`, так что дальнейшие операции запросов привязываются к стандартному набору операций и выполняются последовательно. Такое действие нужно предпринимать перед вызовом методов, которые имеют побочные эффекты или не являются безопасными в отношении потоков.

Для операций запросов, принимающих две входные последовательности (`Join`, `GroupJoin`, `Concat`, `Union`, `Intersect`, `Except` и `Zip`), метод `AsParallel` должен быть применен к обеим входным последовательностям (иначе генерируется исключение). Однако по мере продвижения запроса применять к нему `AsParallel` нет необходимости, т.к. операции запросов PLINQ выдают еще одну последовательность `ParallelQuery`. На самом деле дополнительный вызов `AsParallel` привносит неэффективность, связанную с тем, что он инициирует слияние и повторное разбиение запроса:

```
mySequence.AsParallel() // Помещает последовательность
                            // в оболочку ParallelQuery<int>
    .Where (n => n > 100) // Выводит другую последовательность
                                // ParallelQuery<int>
    .AsParallel() // Необязательно – и неэффективно!
    .Select (n => n * n)
```

Не все операции запросов можно эффективно распараллеливать. Для операций, не поддающихся распараллеливанию (см. раздел “Ограничения PLINQ” далее в главе), PLINQ взамен реализует последовательное выполнение. Инфраструктура PLINQ может также оперировать последовательно, если ожидает, что накладные расходы от распараллеливания в действительности замедлят определенный запрос.

Инфраструктура PLINQ предназначена только для локальных коллекций: скажем, она не работает с Entity Framework, потому что в таких ситуациях LINQ

транслируется в код SQL, который затем выполняется на сервере баз данных. Тем не менее, PLINQ можно использовать для выполнения дополнительных локальных запросов в результирующих наборах, полученных из запросов к базам данных.



Если запрос PLINQ генерирует исключение, то оно повторно генерируется как объект `AggregateException`, свойство `InnerExceptions` которого содержит реальное исключение (либо исключения). Дополнительные сведения можно найти в разделе “Работа с `AggregateException`” далее в главе.

Почему метод `AsParallel` не выбран в качестве варианта по умолчанию?

С учетом того, что метод `AsParallel` прозрачно распараллеливает запросы LINQ, возникает вопрос: почему разработчики в Microsoft не приняли решение распараллеливать стандартные операции запросов, просто сделав PLINQ вариантом по умолчанию?

Есть несколько причин выбора подхода с *включением*. Первая причина связана с тем, что для получения пользы от PLINQ в наличии должен быть обоснованный объем работы с интенсивными вычислениями, которую можно было бы поручить рабочим потокам. Большинство потоков LINQ to Objects выполняются очень быстро, и распараллеливание для них не только окажется излишним, но накладные расходы на разбиение, объединение и координацию дополнительных потоков фактически могут даже замедлить их выполнение. Ниже перечислены другие причины.

- Вывод запроса PLINQ (по умолчанию) может отличаться от вывода запроса LINQ в том, что касается порядка следования элементов (как объясняется в разделе “PLINQ и упорядочивание” далее в главе).
- PLINQ помещает исключения в оболочку `AggregateException` (чтобы учить возможность генерации множества исключений).
- PLINQ будет давать ненадежные результаты, если запрос вызывает не-безопасные к потокам методы.

Наконец, PLINQ предлагает немало способов настройки. Обременение стандартного API-интерфейса LINQ to Objects нюансами подобного рода добавило бы путаницу.

Продвижение параллельного выполнения

Подобно обычным запросам LINQ запросы PLINQ оцениваются ленивым образом. Другими словами, выполнение будет инициировано, только когда начнется потребление результатов — как правило, посредством цикла `foreach` (хотя оно также может происходить через операцию преобразования, такую как `ToArrayList`, или операцию, которая возвращает одиночный элемент либо значение).