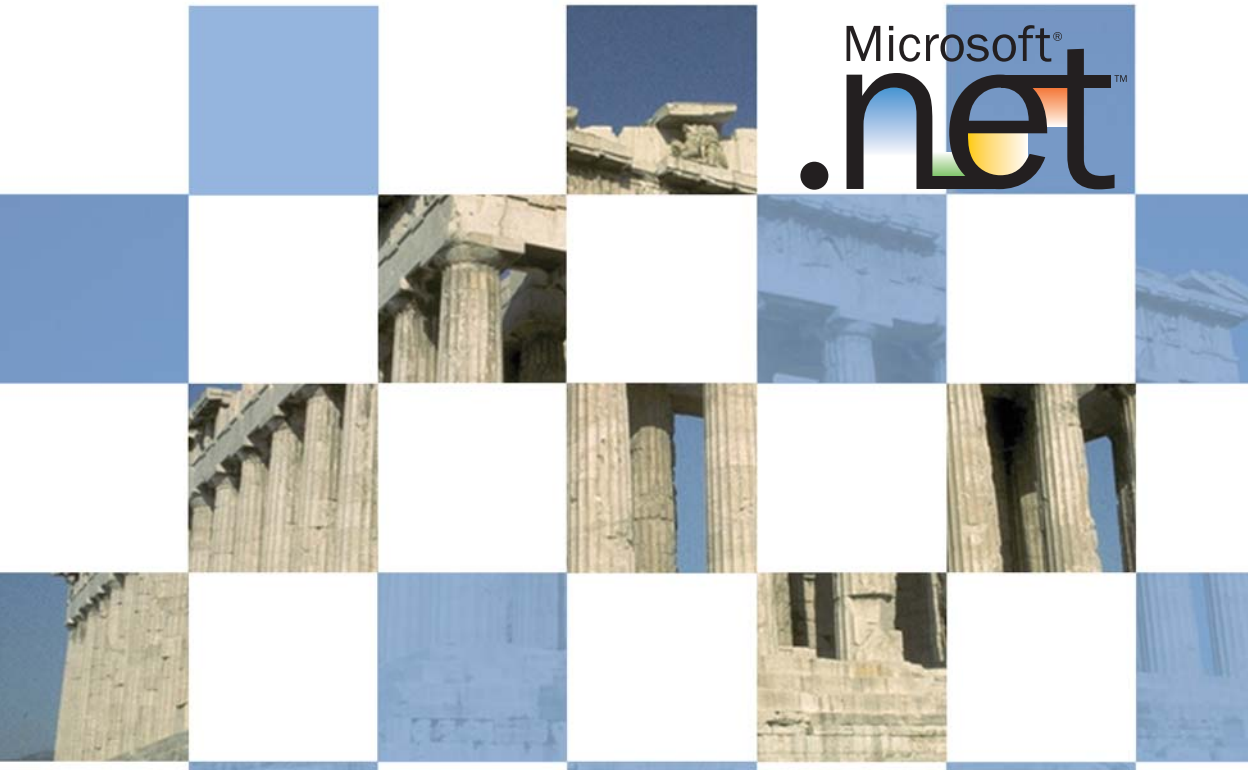
Основы Windows Workflow Foundation



Microsoft® .net™

Дхарма Шукла
Боб Шмидт

ДМК Пресс
издательство

УДК 004.4
ББК 32.973.26-018.2
Ш95

Д. Шукла, Б. Шмидт
Ш95 Основы Windows Workflow Foundation. Пер. с англ. А. Слинкина. – М.: ДМК Пресс, 2008. – 352 с.: ил.

ISBN 5-94074-400-1

Книга посвящена новейшей технологии разработки программ, включенной в состав каркаса .NET Framework 3.0. Речь идет о реактивных возобновляемых программах, которые выполняются эпизодически в ответ на появление внешнего стимула, а затем пассивируются – выгружаются из памяти во внешнее хранилище. Авторы – ведущие архитекторы и разработчики этой технологии – освещают фундаментальные принципы, на которых она основана. Рассматривается широкий круг вопросов: от понятия операции как возобновляемого предложения программы, закладки и до внешних служб, подключаемых к среде исполнения (сохранения, транзакционности, загрузки). Книга будет полезна программистам, желающим глубоко ознакомиться с новой перспективной технологией и осознанно применять ее в собственных проектах.

УДК 004.4
ББК 32.973.26-018.2

ISBN 0-321-39983-8 (англ.)
ISBN 5-94074-400-1

Оглавление

Глава 1. Составные части WF

Учебники программирования часто начинаются с программы «Здравствуй, мир», которая печатает простое сообщение на стандартном устройстве вывода. Вот как выглядит такая программа на языке C#:

```
using System;
class Program
{
  static void Main()
  {
    Console.WriteLine("Здравствуй, мир");
  }
}
```

Программа «Здравствуй, мир» хороша тем, что игнорирует ряд сложных проблем, возникающих в реальной жизни. Специалисты-практики знают, что даже столь простенькую программу можно без труда изменить так, что неприятные вопросы вылезут наружу. Рассмотрим, к примеру, небольшую вариацию на ту же тему, которую назовем «Сезам, откройся», – чтобы программа напечатала традиционное приветствие, нужно ввести ключевое слово:

```
using System;
class Program
{
  static void Main()
  {
    // Напечатать ключ
    string key = DateTime.Now.Millisecond.ToString();
    Console.WriteLine("Вот ваш ключ: " + key);
    string s = Console.ReadLine();
    // Если ключ введен правильно, напечатать приветствие
    if (key.Equals(s))
      Console.WriteLine("Здравствуй, мир");
  }
}
```

Консольная программа «Сезам, откройся» ничем не примечательна, если не считать одной вещи: поскольку она ожидает, что пользователь введет с клавиатуры некий текст, то может не завершаться сколь угодно долго. Вы можете откомпилировать и запустить ее, оставить на несколько недель, а только потом ввести ключ, после чего будет напечатано приветствие. Так и было задумано.

«Сезам, откройся» – пример реактивной программы. Такая программа отвечает на воздействие со стороны внешнего объекта. Более того, ее исполнение целиком и полностью зависит от внешних воздействий. Иногда в роли внешнего объекта выступает человек, а иногда – другая программа. В любом случае реактивная программа бóльшую часть времени проводит в ожидании входных

данных, поэтому для нее свойственны проблемы, не возникающие в программе типа «Здравствуй, мир».

Компьютерные программы в большинстве своем реактивны. Программное обеспечение вездесуще, будь то совместная работа над документом, обработка заказов клиентов, закупки сырья, заполнение налоговых деклараций, снабжение продовольствием, управление разработкой продукта, покупки в онлайновых магазинах, система взаимоотношений с клиентами или доставка компонентов со склада на сборочное предприятие. Список можно продолжать до бесконечности. Программы, находящиеся в сердцевине всех этих процессов, должны реагировать на действия со стороны людей и других программ.

Некоторые реактивные программы разрабатываются с помощью каркасов, например ASP.NET и Java Servlets. Другие создаются на месте и непосредственно используют ту или иную среду исполнения, скажем, Common Language Runtime (CLR) или Java Virtual Machine (JVM). Третьи пишутся на таких языках, как C или C++ (неуправляемый).

Но если посмотреть, как написаны реактивные программы, то окажется, что большинство из них совсем не похожи на программу «Сезам, откройся». Вот как можно было бы написать Web-сервис, делающий то же самое, что «Сезам, откройся» (Web-приложение в этом отношении ничуть не менее поучительно).

```
using System;
using System.Web.Services;
[WebService]
public class Service : WebService
{
  [WebMethod(EnableSession = true)]
  public string PrintKey()
  {
    string key = DateTime.Now.Millisecond.ToString();
    Session["key"] = key;
    return "Вот ваш ключ: " + key;
  }
  [WebMethod(EnableSession = true)]
  public string PrintGreeting(string s)
  {
    if (Session["key"].Equals(s))
      return "Здравствуй, мир";
    return null;
  }
}
```

Этот Web-сервис содержит две операции, сложным его не назовешь. Однако мы полностью утратили поток управления, который присутствовал в программе «Сезам, откройся», – тот очевидный факт, что вызов `PrintKey` должен предшествовать вызову `PrintGreeting` и что исполнение программы можно считать законченным, если каждый из них выполнен ровно один раз. Чтобы наложить ограничение на порядок выполнения операций, можно было бы модифицировать Web-сервис, добавив код, выделенный полужирным шрифтом:

```
using System;
using System.Web.Services;
[WebService]
public class Service : WebService
{
  [WebMethod(EnableSession = true)]
  public string PrintKey()
  {
    bool alreadyDidStep1 = (Session["key"] != null);
    if (alreadyDidStep1)
      throw new InvalidOperationException();
    string key = DateTime.Now.Millisecond.ToString();
    Session["key"] = key;
    return "Вот ваш ключ: " + key;
  }
  [WebMethod(EnableSession = true)]
  public string PrintGreeting(string s)
  {
    bool didNotDoStep1Yet = (Session["key"] == null);
    if (didNotDoStep1Yet)
      throw new InvalidOperationException();
    bool alreadyDidStep2 = (Session["programDone"] != null);
    if (alreadyDidStep2)
      throw new InvalidOperationException();
    Session["programDone"] = true;
    if (Session["key"].Equals(s))
      return "Здравствуй, мир";
    return null;
  }
}
```

Мы включили ряд проверок во время выполнения, которые гарантируют требуемый поток управления в Web-сервисе, но логика оказалась разбросанной по разным местам, неочевидной и чреватой ошибками. Линейная последовательность предложений в программе «Сезам, откройся» разбилась на кусочки, рассеянные по разным методам Web-сервиса. Представьте, что вам нужно восстановить поток управления (не говоря уже о потоке данных), видя только исходный текст Web-сервиса. В таком простом примере на это уйдет несколько секунд, поскольку есть всего две операции, но что если исходный текст будет в десять раз больше по объему и содержать ветвления и циклы?

Почему нельзя использовать естественные управляющие конструкции C# (в конце концов, мы же пишем на этом языке), чтобы описать взаимосвязи между операциями Web-сервиса и наложить ограничения на то, что за чем должно следовать? Поток управления и манипуляции локальными переменными в консольной программе «Сезам, откройся» точно соответствуют потребностям. Так почему невозможно так же написать Web-сервис или Web-приложение, вообще любую реальную программу?

На то есть две причины:

- в консольной программе «Сезам, откройся» вызов метода `Console.ReadLine` блокирует выполнение текущего потока. Программа может

целыми днями дожидаться входных данных. Если бы одновременно было запущено много подобных программ и все они ожидали ввода, то система просто остановилась бы. Выделение потоков таким способом для реальных программ неприемлемо, особенно если они развертываются в многопользовательской среде;

- реальные процессы могут занимать дни, недели и даже месяцы. Было бы легкомыслием надеяться, что процесс операционной системы (или домен приложения CLR), в котором программа начала выполняться, проживет так долго.

Для консольной программы, используемой в учебных целях, такие проблемы нас обычно не волнуют. Следовательно, можно написать программу «Сезам, откройся» естественным образом – так же, как «Здравствуй, мир». Из исходного текста совершенно понятно, что программа делает. Но с Web-сервисом ситуация прямо противоположная. Для Web-сервисов и Web-приложений масштабируемость и надежность имеют первостепенное значение.

Среда исполнения ASP.NET спроектирована в расчете на эффективное управление многими сервисами и приложениями и может надежно сохранять состояние отдельных сеансов (при правильной конфигурации сеанс даже может передаваться на другую машину в случае отказа). Но код не лжет. За масштабируемость и надежность приходится платить. В предыдущем примере переменная `key`, общая для двух операций, трактуется как слабо типизированная пара имя–значение. Более того, логика, управляющая порядком выполнения операций, реализована очень неэлегантно – в начале каждой операции проверяется, было ли уже присвоено переменной `key` какое-нибудь значение.

Похоже, что масштабируемость и надежность вступают в противоречие с желанием естественным образом выразить состояние и поток управления реактивных программ. Существование миллионов Web-приложений и Web-сервисов подтверждает согласие программистов работать в рамках ограничений сегодняшних моделей программирования, лишь бы задача была решена. Но даже если так, современные парадигмы Web-разработки годятся лишь для определенного подмножества задач, в которых требуются реактивные программы. Наша цель – отыскать более универсальный подход к разработке реактивных программ как в контексте Web, так и для других целей. Такой подход должен удовлетворять следующим условиям:

1. Не приносить в жертву, а, напротив, обогащать естественный императивный стиль описания потока управления.
2. Обеспечить масштабируемый и отказоустойчивый способ исполнения реактивных программ.

Независимость от процесса и потока

Наш вариант реализации Web-сервиса «Сезам, откройся» в ASP.NET является масштабируемым и отказоустойчивым, чего нельзя сказать о консольной

программе. Рассмотрим внимательнее то предложение, которое является источником неприятностей:

```
string s = Console.ReadLine();
```

Фундаментальная проблема заключается в том, что после вызова метода `Console.ReadLine` выполнение программы «Сезам, откройся» приостанавливается (если не рассматривать выход из-за ошибки) до тех пор, пока не поступят входные данные. Выделение потока – а это довольно дефицитный ресурс – каждому экземпляру программы затрудняет создание масштабируемого решения в ситуации, когда одновременно может исполняться много экземпляров.

Для решения этой проблемы часто применяют асинхронные вызовы методов. Например, каркас .NET Framework позволяет заменить метод `ReadLine` парой асинхронных вызовов:

```
public static System.IAsyncResult BeginReadLine(
  System.AsyncCallback asyncCallback,
  object state
);
public static string EndReadLine(System.IAsyncResult ar);
```

Внутри метода `BeginReadLine` можно создать запрос на выполнение работы и поставить его в очередь, организованную внутри CLR. Этот запрос будет обслужен асинхронно потоком из пула потоков CLR, а метод `BeginReadLine` сразу вернет управление вызывающей программе.

Поток, вызвавший `BeginReadLine`, может опрашивать свойство `IsCompleted` объекта `System.IAsyncResult`, пока не будет получен результат, либо воспользоваться свойством `AsyncWaitHandle` (как показано ниже), чтобы получить извещение о готовности результата. Последнее более эффективно.

```
using System;
class Program
{
  static void Main()
  {
    // Напечатать ключ
    string key = DateTime.Now.Millisecond.ToString();
    Console.WriteLine("Вот ваш ключ: " + key);
    IAsyncResult result = BeginReadLine(null, null);
    result.AsyncWaitHandle.WaitOne();
    string s = EndReadLine(result);
    // Если ключ введен правильно, напечатать приветствие
    if (key.Equals(s))
      Console.WriteLine("Здравствуй, мир");
  }
}
```

Но хотя в этой программе и применяется асинхронный вызов, поток все равно потребляется, поскольку мы вызываем метод `WaitOne` для объекта `IAsyncResult`.

Выход из ситуации начинает вырисовываться, если взглянуть на последние два параметра метода `BeginReadLine`. В каркасе .NET Framework при асинхронном вызове метода `Begin` можно указать делегат типа `System.AsyncCallback`

(наряду с объектом, в котором будет храниться состояние[1], общее для метода обратного вызова и для программы, вызвавшей `Begin`; в наших примерах мы в качестве такого объекта передаем `null`). Предполагается, что асинхронно вызванный метод известит о завершении операции посредством вызова делегата.

Вот вариант программы «Сезам, откройся», переписанный с использованием механизма `AsyncCallback`:

```
using System;
using System.Threading;
class Program
{
  static string key;
  static void Main()
  {
    // Напечатать ключ
    key = DateTime.Now.Millisecond.ToString();
    Console.WriteLine("Вот ваш ключ: " + key);
    BeginReadLine(ContinueAt, null);
    Thread.Sleep(Timeout.Infinite);
  }
  static void ContinueAt(IAsyncResult ar)
  {
    string s = EndReadLine(ar);
    // Если ключ введен правильно, напечатать приветствие
    if (key.Equals(s))
      Console.WriteLine("Здравствуй, мир");
    Environment.Exit(0);
  }
}
```

Хотя эта программа все еще удерживает начальный поток (на этот раз из-за обращения к методу `Sleep` с бесконечным временем ожидания), но теперь часть ее может исполняться в произвольном потоке. Метод `ContinueAt` будет исполняться не в том же потоке, что и `Main`.

В этой версии программы независимость от потока достигается очень простым способом. Мы пользуемся асинхронным вызовом, чтобы разделить части программы, оставив между ними связь. Важно отметить, что мы иначе реализовали переменную `key`, которая нужна на разных стадиях выполнения программы. Теперь она выделяется не в стеке, как раньше, а является статическим полем, которое видно различным методам. Отказ от использования стека – ключ к уменьшению зависимости программы от потока при сохранении строго типизированных объявлений данных.

Закладки

Мы показали, что можно сделать программу «Сезам, откройся» независимой (хотя бы отчасти) от потока за счет асинхронного вызова методов и делегата `AsyncCallback`. Но мы еще ничего не сделали для того, чтобы программа

[1] Это напоминает слабо типизированное состояние сеанса, использованное нами при программировании Web-сервиса.

оказалась «независимой от процесса», а это, как мы знаем, необходимое условие масштабируемости и надежности.

Впрочем, намеченный подход (использование `AsyncCallback`) вселяет надежды. Делегат, ссылающийся на метод `ContinueAt`, выступает в роли закладки – логической отметки того места программы, с которого исполнение должно быть возобновлено при поступлении сигнала извне. Это позволяет наметить следующий путь:

1. Мы можем присвоить закладкам имена и реализовать менеджер закладок.
2. Мы можем сделать закладки сериализуемыми, а значит, сохранять их во внешнем хранилище и извлекать оттуда.
3. Мы можем написать программу-прослушиватель, которая будет единственной точкой входа для данных, подлежащих доставке любой из зарегистрированных в системе закладок.

Начнем с определения класса `Bookmark`, который, по существу, является именованной оберткой для делегата типа `BookmarkLocation`:

```
[Serializable]
public class Bookmark
{
  public Bookmark(string name, BookmarkLocation continueAt);
  public string Name { get; }
  public BookmarkLocation ContinueAt { get; }
  public object Payload { get; }
  public BookmarkManager BookmarkManager { get; }
}
public delegate void BookmarkLocation(Bookmark resumed);
```

Для управления закладками служит класс `BookmarkManager`:

```
public class BookmarkManager
{
  public void Add(Bookmark bookmark);
  public void Remove(Bookmark bookmark);
  public void Resume(string bookmarkName, object payload);
}
```

Закладка – это объект-*продолжение*, который представляет замороженную в некоторой точке программу. Физическая точка продолжения определяется свойством `ContinueAt` объекта-закладки. У закладки есть имя, чтобы на нее можно было сослаться и манипулировать ею независимо от физической точки продолжения (она может быть общей для нескольких закладок). Объект `BookmarkManager` просто управляет набором закладок. При вызове метода `BookmarkManager.Resume` соответствующая программа возобновляется с места, указанного в закладке. Входные данные (стимул), передаваемые при возобновлении, программа может получить с помощью свойства `Payload` возобновленной закладки.

Мы пометили класс `Bookmark` атрибутом `[Serializable]`, чтобы менеджер закладок мог сохранять закладки во внешнем хранилище, например в базе данных. А это означает, что, когда бы мы ни создали объект `BookmarkManager`, он сможет

надежно получить свой набор закладок, при условии что имеет доступ к внешнему хранилищу (например, в виде строки соединения с базой данных). Если реактивная программа приостанавливается в ожидании стимула, то мы можем пассивировать ее, то есть сохранить во внешнем хранилище в виде набора закладок.

Чтобы можно было поставить закладку в любом месте программы, необходимо полностью исключить, а не просто уменьшить зависимость от стека. Это означает, что метод `Main` не сможет вызывать `Thread.Sleep`, как в последней версии «Сезам, откройся». Вместо этого придется написать класс `OpenSesame`, в котором будут методы экземпляра и состояние, выделенное из кучи.

Вот как выглядит новая реализация программы «Сезам, откройся»:

```
[Serializable]
public class OpenSesame
{
  string key;
  public void Start(BookmarkManager mgr)
  {
    // Напечатать ключ
    key = DateTime.Now.Millisecond.ToString();
    Console.WriteLine("Вот ваш ключ: " + key);
    mgr.Add(new Bookmark("read", ContinueAt));
  }
  void ContinueAt(Bookmark resumed)
  {
    string s = (string) resumed.Payload;
    BookmarkManager mgr = resumed.BookmarkManager;
    mgr.Remove(resumed);
    // Если ключ введен правильно, напечатать приветствие
    if (key.Equals(s))
      Console.WriteLine("Здравствуй, мир");
  }
}
```

Теперь программа «Сезам, откройся» реализована объектом типа `OpenSesame`. Такой объект легко создать и «запустить» (место `Main` занимает метод `Start`) в контексте прослушивателя[1], который ожидает входные данные:

```
BookmarkManager mgr = new BookmarkManager();
OpenSesame openSesameProgram = new OpenSesame();
openSesameProgram.Start(mgr);
...
string str = // получить входные данные
mgr.Resume("read", str);
...
```

Возобновляемые предложения программы

Показанный выше код будет работать для одного экземпляра одной программы, но было бы гораздо лучше, если бы нас обслуживала некая среда исполнения –

[1] Прослушиватель может получать данные откуда угодно: из базы данных, из источника типа очереди MSMQ, от Web-сервиса или Web-приложения, даже с заслуживающей доверия консоли. Эта логика теперь отделена от самой программы «Сезам, откройся».

окружение, подобное нашему классу OpenSesame, – которая помогала бы управлять несколькими экземплярами различных программ такого рода.

Назовем это окружение мифической средой исполнения (MythicalRuntime):

```
public class MythicalRuntime
{
  // Запускает новую программу
  public ProgramHandle RunProgram(ProgramStatement program);
  // Возвращает описатель ранее запущенной программы
  public ProgramHandle GetProgramHandle(Guid programId);
  // Пассивирует все программы, находящиеся в памяти
  public void Shutdown();
}
public class ProgramHandle
{
  // Уникальный идентификатор данной программы
  public Guid ProgramId { get; }
  // Пассивировать программу
  public void Passivate();
  // Возобновить исполнение с закладки
  public void Resume(string bookmarkName, object payload);
}
```

Мифическая среда исполнения позволяет запустить объект ProgramStatement.

Настало время формализовать идею о том, что класс OpenSesame – это действительно новый вид программы. С учетом разумного требования о том, чтобы класс OpenSesame мог использоваться (как одно предложение) в более сложных программах (состоящих из нескольких предложений), мы будем называть его *возобновляемым предложением программы*.

Тип ProgramStatement стандартизует точку входа для выполнения и служит базовым классом для всех *возобновляемых предложений*.

```
[Serializable]
public abstract class ProgramStatement
{
  public abstract void Run(BookmarkManager mgr);
}
```

При таком определении класса ProgramStatement класс OpenSesame можно переписать следующим образом:

```
[Serializable]
public class OpenSesame : ProgramStatement
{
  string key;
  public override void Run(BookmarkManager mgr)
  {
    // Напечатать ключ
    key = DateTime.Now.Millisecond.ToString();
    Console.WriteLine("Вот ваш ключ: " + key);
    mgr.Add(new Bookmark("read", ContinueAt));
  }
  // ContinueAt не изменился, для краткости код опущен
  ...
}
```

Объект ProgramStatement, который передается методу RunProgram мифической среды исполнения, – это возобновляемая программа. Когда такой объект попадает в мифическую среду исполнения, он становится больше, чем обычным объектом CLR. Теперь это временное представление возобновляемой программы в памяти. Поскольку возобновляемую программу можно пассивировать, у нее есть логическое время жизни, которое потенциально больше, чем время жизни начального объекта ProgramStatement и даже домена приложения CLR, в котором этот объект был создан.

У каждой программы, управляемой мифической средой исполнения, есть глобально уникальный идентификатор, который можно получить от объекта ProgramHandle, возвращаемого методом MythicalRuntime.RunProgram. Этот идентификатор можно передавать между машинами, так что пассивированную программу можно загрузить в память с помощью метода MythicalRuntime.GetProgramHandle всюду, где есть доступ к долговременному хранилищу, в котором сохраняются пассивированные программы.

Поскольку логика исполнения возобновляемой программы основана на закладках, то программа будет ожидать внешнего стимула, не блокируя поток, в котором достигла точки приостановки. Когда это происходит, мифическая среда исполнения может пассивировать программу. Как только входные данные поступят, программа загружается в память и ее исполнение возобновляется (с помощью метода ProgramHandle.Resume). Возобновление пассивированной программы может произойти 47 недель спустя совсем на другой машине.

Теперь можно привести эскиз прослушивателя, который ожидает поступления входных данных, определяет, какой программе они предназначены, и возобновляет эту программу с подходящей закладки[1]:

```
MythicalRuntime runtime = new MythicalRuntime();
...
while (true)
{
  // получить входные данные
  object input = ...
  // определить на основе данных, какую программу вызывать
  ProgramHandle handle = runtime.GetProgram(...)
  // определить на основе данных, с какой закладки продолжить исполнение
  string bookmarkName = ...
  handle.Resume(bookmarkName, input);
}
```

При такой реализации прослушиватель, очевидно, волен сам решать, как сопоставить входные данные с закладками внутри программ. Но это и хорошо, ведь наша цель – поддержать различные концептуальные модели, надстроенные над

[1] На практике прослушиватель также может решить, что поступившие данные требуют создания новой программы. В таком случае вызывается метод MythicalRuntime.RunProgram, а не ProgramHandle.Resume.

универсальным механизмом закладок, который мы собираемся создать; мифическая среда исполнения не привязана к какой-то конкретной реализации прослушивателя.

Нам удалось организовать гибкий механизм закладок и среду исполнения возобновляемых программ. Это универсальный каркас, обладающий двумя важными характеристиками:

- программы с закладками не блокируют потоки, ожидая стимула для продолжения работы;
- пока программа с закладками ожидает стимула, она может храниться в долговременной памяти в виде объекта-продолжения.

Подведем итоги. Пока что мы воспользовались идеей закладок, позволяющей писать возобновляемые программы, которые можно пассивировать. Поскольку такие программы сохраняют состояние в сериализуемых полях, их можно представить в виде объектов-продолжений, хранящихся в долговременной памяти. Такие программы независимы от потока и процесса. Более того, общность модели закладок означает, что ее можно использовать при разработке разнообразных реактивных программ – не только ориентированных на Web.

Композиция

Наша работа не завершена. Возобновляемая программа `OpenSesame`, определенно, не обладает тем потоком управления, который нам нужен. В первоначальном варианте «Сезам, откройся» имеется линейная последовательность предложений.

```
static void Main()
 {
  // Напечатать ключ
  string key = DateTime.Now.Millisecond.ToString();
  Console.WriteLine("Вот ваш ключ: " + key);
  string s = Console.ReadLine();
  // Если ключ введен правильно, напечатать приветствие
  if (key.Equals(s))
    Console.WriteLine("Здравствуй, мир");
 }
```

Мы можем применить стандартную технику разбиения на модули и разложить возобновляемую программу `OpenSesame` на меньшие структурные единицы, каждая из которых устроена так же, как `OpenSesame`. Тогда и другие программы смогут воспользоваться этими строительными блоками.

---

[1] Например, модель программирования для Web, скажем, ASP.NET, где входящие запросы отображаются на программы с помощью идентификаторов сеансов (аналогичных `ProgramHandle.ProgramId`), и статическую модель для описания точек возобновления программы (имена операций Web-сервиса аналогичны именам закладок).

Приведенный ниже класс Read – это ориентированная на закладки замена предложению Console.ReadLine в исходной консольной программе:

```
[Serializable]
public class Read : ProgramStatement
{
  private string text;
  public string Text
  {
    get { return text; }
  }
  public override void Run(BookmarkManager mgr)
  {
    mgr.Add(new Bookmark("read", ContinueAt));
  }
  void ContinueAt(Bookmark resumed)
  {
    text = (string) resumed.Payload;
    BookmarkManager mgr = resumed.BookmarkManager;
    mgr.Remove(resumed);
  }
}
```

В классе Read есть свойство Read.Text, чтобы программа, вызывающая метод Read.Run, могла получить доступ к строке, передаваемой при возобновлении программы с закладки. Но как вызывающая программа узнает, когда значение свойства Read.Text доступно? Можно сделать так, что закладка будет извещать ее, когда завершится выполнение метода Read.ContinueAt.

В нашей реализации класса Read мы используем закладки, чтобы можно было приостанавливать выполнение программы, не блокируя поток, и возобновлять, когда от внешнего объекта придут данные. Но тот же механизм закладок можно применить и для внутренних извещений, и тогда метод Read.Run приобретет такую же способность приостанавливаться и возобновляться по завершении выполнения Read!

Вот несколько модифицированный класс Read (реализация свойства Text не изменилась, так что мы ее не приводим):

```
[Serializable]
public class Read : ProgramStatement
{
  // Свойство Text для краткости опущено...
  private string outerBookmarkName;
  public Read(string outerBookmarkName)
  {
    this.outerBookmarkName = outerBookmarkName;
  }
  public void Run(BookmarkManager mgr)
  {
    mgr.Add(new Bookmark("read", ContinueAt));
  }
  public void ContinueAt(Bookmark resumed)
  {
    text = (string) resumed.Payload;
    BookmarkManager mgr = resumed.BookmarkManager;
```

```
    mgr.Remove(resumed);
    mgr.Resume(outerBookmarkName, this);
  }
}
```

Конструктор класса `Read` принимает в качестве параметра имя «внешней» закладки. Это имя сохраняется и после установки поля `text` (в методе `ContinueAt`), объекту `BookmarkManager` отправляется запрос возобновить выполнение с «внешней» закладки. Класс `Read` логически завершил выполнение, а вызвавшая его программа получила уведомление об этом факте.

Жизненный цикл программы

Продемонстрированная выше техника внутренних извещений прекрасно работает, но не видно причин, по которым класс `BookmarkManager` не мог бы поддержать этот простой паттерн асинхронных извещений при работе с внутренними закладками. К механизму функционирования внутренних закладок мы скоро вернемся, а пока взгляните на упрощенный вариант класса `Read`:

```
[Serializable]
public class Read : ProgramStatement
{
  // Свойство Text для краткости опущено...
  public override void Run(BookmarkManager mgr)
  {
    mgr.Add(new Bookmark("read", ContinueAt));
  }
  public void ContinueAt(Bookmark resumed)
  {
    text = (string) resumed.Payload;
    BookmarkManager mgr = resumed.BookmarkManager;
    mgr.Remove(resumed);
    mgr.Done();
  }
}
```

Вместо того чтобы заботиться о «внешней» закладке, класс `Read` просто информирует менеджер закладок о том, что завершил выполнение. Вторая половина паттерна реализована в программе, вызывающей `Read`; в данном случае это программа `OpenSesame`:

```
[Serializable]
public class OpenSesame : ProgramStatement
{
  string key;
  public override void Run(BookmarkManager mgr)
  {
    // Напечатать ключ
    key = DateTime.Now.Millisecond.ToString();
    Console.WriteLine("Вот ваш ключ: " + key);
    mgr.RunProgramStatement(new Read(), ContinueAt);
  }
  public void ContinueAt(Bookmark resumed)
  {
    Read read = (Read) resumed.Payload;
```

```
    string s = read.Text;
    // Если ключ введен правильно, напечатать приветствие
    if (key.Equals(s))
      Console.WriteLine("Здравствуй, мир");
    mgr.Done();
  }
}
```

Мы разложили класс `OpenSesame` на составные части, воспользовавшись объектом `Read` для получения требуемой строки. Вместо того чтобы вызывать метод `Read.Run` напрямую, `OpenSesame` вызывает `BookmarkManager.RunProgramStatement` и передает точку возобновления для закладки, которой управляет объект `BookmarkManager`. Менеджер закладок вызывает метод `Read.Run` от имени `OpenSesame`. Внутренняя закладка не раскрывается объекту `Read`, этот объект просто сообщает о завершении работы, вызывая метод `BookmarkManager.Done`. В методе `Done` объект `BookmarkManager` возобновляет исполнение с внутренней закладки, которая была создана в момент предыдущего обращения к методу `BookmarkManager.RunProgramStatement`. В этой точке `OpenSesame` может опросить свойство `Read.Text` (объект `Read` передан для удобства в виде полезной нагрузки возобновляемой закладки).

Теперь класс `BookmarkManager` выглядит следующим образом:

```
public class BookmarkManager
{
  public void Add(Bookmark bookmark);
  public void Remove(Bookmark bookmark);
  // Запрос на выполнение предложения программы с помощью
  // неявной закладки, с которой будет возобновлено исполнение,
  // когда это предложение завершится
  public void RunProgramStatement(ProgramStatement statement,
    BookmarkLocation continueAt)
  // Указывает, что текущее предложение завершилось, так что
  // можно возобновлять выполнение с внутренней закладки
  public void Done();
}
```

Четко определив, что такое начало (`ProgramStatement.Run`) и конец (`BookmarkManager.Done`) выполнения любого возобновляемого предложения программы, мы можем описать жизненный цикл предложения в терминах конечного автомата, показанного на рис. 1.1.

Рис. 1.1. Конечный автомат, представляющий жизненный цикл предложения программы

Только что созданное предложение программы находится в «латентном» состоянии, то есть ожидает запуска. После вызова метода `Run` программа переходит в состоянии «исполняется» и остается в нем неопределенно долго, пока асинхронно не сообщит о своем завершении.