# ГЛАВА 16

# Многопоточность

Платформа Silverlight поддерживает многопоточность, что позволяет выполнять несколько частей кода одновременно. Многопоточность — ключевой компонент полнофункциональной структуры .NET Framework, часто используемый в мощных клиентских приложениях WPF и Windows Forms. Однако многопоточности нет в большинстве сред разработки приложений для браузеров. Наиболее заметно ее отсутствие в JavaScript и Flash.

Средства многопоточности в Silverlight очень похожи на аналогичные средства в .NET Framework. Как и в .NET Framework, разработчик приложений Silverlight может создавать новые потоки с помощью класса `Thread`, управлять длительными операциями с помощью класса `BackgroundWorker` и даже направлять задачи в пул рабочих потоков с помощью класса `ThreadPool`. Все эти компоненты Silverlight почти полностью аналогичны соответствующим компонентам .NET Framework, поэтому разработчик, знакомый с многопоточными клиентскими приложениями .NET, быстро почувствует себя в Silverlight как дома. Впрочем, есть некоторые ограничения. Например, в Silverlight нельзя управлять приоритетами потоков с помощью кода. Тем не менее незначительные ограничения потоков Silverlight не мешают им быть мощным средством управления приложением.

В этой главе рассматривается низкоуровневый класс `Thread`, предоставляющий гибкие средства создания новых потоков. Затем обсуждается модель потоков Silverlight и правила их выполнения. И наконец, рассматривается высокоуровневый класс `BackgroundWorker`, предоставляющий удобный способ обработки фоновых задач.

## Основы многопоточности

При создании потоков разработчик пишет код таким образом, будто каждый поток выполняется независимо от других. Операционная система Windows предоставляет каждому потоку короткий интервал времени, после которого замораживает поток, переводя его в состояние приостановленного выполнения. Через несколько миллисекунд операционная система размораживает поток, позволяя ему выполнить очередную порцию задачи.

Такая модель постоянных прерываний называется *вытесняющей многозадачностью* (preemptive multitasking). Она реализуется операционной системой и не управляется приложением Silverlight. Приложение работает так, будто все потоки выполняются одновременно, причем каждый поток выполняется как независимая программа, решающая собственную задачу.

**Примечание.** Если на компьютере установлено несколько процессоров или двухъядерный процессор, несколько потоков действительно могут выполняться одновременно. Однако это не обязательно, потому что надстройка Silverlight и другие клиентские приложения тоже могут потребовать у операционной системы предоставить им процессор. Кроме того, высокоуровневые коды Silverlight транслируются в низкоуровневые инструкции. В некоторых случаях несколько процессоров могут выполнять одновременно несколько инструкций. Таким образом, один поток иногда загружает несколько процессоров.

## Назначение многопоточности

При использовании нескольких потоков сложность приложения существенно увеличивается, потому что возникают многие малозаметные эффекты, приводящие к загадочным ошибкам. Прежде чем разбить приложение на несколько потоков, тщательно проанализируйте, окупятся ли дополнительные затраты труда и усложнение кода.

Ниже описаны три главные причины использования в программе многих потоков.

- **Чтобы клиентское приложение всегда немедленно реагировало на действия пользователя.** Если приложение выполняет длительную задачу, она захватывает ресурсы процессора, и приложение перестает реагировать на действия пользователя. Ему приходится ждать, пока закончится выполнение вспомогательной задачи. Пользователям такое поведение программы, естественно, не понравится. Радикальное решение проблемы состоит в направлении (маршализации) вспомогательной задачи в другой поток, в результате чего интерфейс, обслуживаемый первым потоком, будет немедленно реагировать на действия пользователя. Можно также предоставить пользователю возможность отменить вспомогательную задачу в любой момент, когда она еще не завершена.
- **Для решения нескольких задач одновременно.** При использовании типичного однопроцессорного компьютера многозадачность сама по себе не повышает производительность. Фактически производительность даже немного уменьшается вследствие накладных расходов на создание потоков. Однако для многих задач характерны существенные интервалы бездействия процессора. Например, процессор работает не все время при загрузке данных из внешнего источника (веб-сайта, базы данных и т.п.) или при коммуникации с дистанционным компонентом. Когда выполняются эти задачи, процессор большую часть времени не занят. Уменьшить время ожидания нельзя, потому что оно определяется пропускной способностью канала и производительностью сервера, однако можно полнее загрузить процессор другими задачами. Например, можно одновременно передать запросы трем разным веб-службам, уменьшив таким образом общее время ожидания.
- **Для обеспечения расширяемости приложения на стороне сервера.** Серверная часть приложения должна одновременно обслуживать произвольное (обычно большое) количество клиентов. Одновременность обеспечивается серверной технологией (например, ASP.NET). Разработчик приложений Silverlight может создать собственную инфраструктуру серверной части. Например, в главе 23 рассматривается создание приложений на основе сокетов и сетевых классов .NET. Однако подобные технологии обычно касаются серверных приложений, а не приложений Silverlight.

В данной главе рассматривается пример, в котором использование многопоточности предоставляет существенные преимущества. В этом примере трудоемкая операция выполняется в фоновом потоке. Вы узнаете, как обеспечить постоянную готовность интерфейса (немедленную реакцию приложения на действия пользователя), как избежать ошибок в потоках и создать индикатор прогресса и средства отмены фоновой операции.

**Совет.** Быстродействие процессора редко бывает лимитирующим фактором производительности приложения Silverlight. Обычно производительность ограничивается низкой пропускной способностью каналов, медлительностью веб-служб, необходимостью частого обращения к диску и другими лимитирующими факторами. В результате многопоточность редко улучшает общую производительность, даже при использовании двухъядерного процессора. Поэтому в приложениях Silverlight многопоточность полезна главным образом для обеспечения постоянной готовности интерфейса.

## Класс `DispatcherTimer`

Содержащийся в пространстве имен `System.Windows.Threading` класс `DispatcherTimer` позволяет избежать многих проблем, связанных с многопоточностью. В главе 10 он использовался в игре с перехватом бомб.

Класс `DispatcherTimer` не создает потоки. Вместо этого он периодически генерирует событие `Tick` в главном потоке приложения. Событие `Tick` прерывает выполнение кода в любой точке, в которой оно застало код, и предоставляет возможность запустить иную задачу. Если нужно часто выполнять небольшие коды (например, запускать новые анимации каждые полсекунды), класс `DispatcherTimer` работает так же гладко, как классы действительной многопото чности.

Главное преимущество класса `DispatcherTimer` состоит в том, что событие `Tick` всегда генерируется только в главном потоке приложения. Благодаря этому устраняются проблемы синхронизации потоков и другие неприятности, рассматриваемые далее. Однако появляется ряд ограничений. Например, если код обработки событий таймера выполняет трудоемкую задачу, пользовательский интерфейс блокируется, пока она не закончится. Следовательно, таймер не обеспечивает постоянную готовность пользовательского интерфейса и не позволяет сократить время ожидания при выполнении операций, включающих интервалы простаивания процессора. Для решения этих задач необходимо применить истинную многопоточность.

Тем не менее в некоторых ситуациях класс `DispatcherTimer` при умелом использовании позволяет достичь требуемых эффектов. Например, с его помощью рекомендуется периодически проверять веб-службы на наличие новых данных. Вызовы веб-служб выполняются асинхронно в фоновом потоке (см. главу 19). Следовательно, класс `DispatcherTimer` можно использовать в приложении для периодической загрузки данных из медлительных веб-служб. Например, можно задать генерацию событий `Tick` каждые несколько минут. Вследствие того что веб-служба вызывается асинхронно, загрузка будет выполняться в фоновом потоке.

# Класс `Thread`

Наиболее простой способ создания многопоточного приложения Silverlight состоит в использовании класса `Thread`, содержащегося в пространстве имен `System.Threading`. Каждый объект `Thread` представляет один поток.

Для создания объекта `Thread` нужно предоставить делегат методу, вызываемому асинхронно. Объект `Thread` указывает только на один метод. Сигнатура метода жестко ограничена. Он не может возвращать значение. Кроме того, он либо не должен иметь параметров (для делегата `ThreadStart`), либо должен иметь один объектный параметр (для делегата `ParameterizedThreadStart`).

Предположим, необходимо вызвать метод.

```
private void DoSomething()
{ ... }
```

Создайте для него объект потока `thread`.

```
ThreadStart asyncMethod = new ThreadStart(DoSomething);
Thread thread = new Thread(asyncMethod);
```

После этого запустите поток, вызвав метод `Thread.Start()`. Если поток принимает объектный параметр, его следует передать методу `Start()`. Ниже приведена инструкция, запускающая поток без параметров.

```
thread.Start();
```

Метод `Start()` немедленно возвращает управление (не завершаясь), и код начинает выполняться асинхронно в новом потоке. Когда метод завершается, поток уничтожается и не может быть использован повторно. Во время выполнения потока его свойства и методы (табл. 16.1) можно использовать для управления выполнением.

**Таблица 16.1. Члены класса `Thread`**

| *Имя* | *Описание* |
|---|---|
| `IsAlive` | Это свойство имеет значение `false`, когда поток остановлен, уничтожен или еще не запущен |
| `ManagedThreadId` | Целое число, уникально идентифицирующее поток |
| `Name` | Строка, идентифицирующая поток. Полезна главным образом для отладки приложения, но может быть использована и в рабочем режиме для идентификации потока. После первой установки свойства `Name` установить его повторно нельзя |
| `ThreadState` | Комбинация значений, указывающих на состояние потока (запущен, выполняется, завершен, остановлен и т.п.). Используется только для отладки |
| `Start()` | Запуск потока на выполнение в первый раз. Метод `Start()` нельзя использовать для повторного запуска потока после его завершения |
| `Join()` | Ожидание завершения потока или истечения заданного интервала времени |
| `Sleep()` | Приостановка текущего потока на заданное количество миллисекунд. Этот метод статический |

**Примечание.** Опытные программисты .NET заметят в версии Silverlight класса `Thread` отсутствие нескольких деталей. В Silverlight все потоки являются фоновыми. Им нельзя присвоить приоритеты. Кроме того, хотя класс `Thread` содержит метод `Abort()`, уничтожающий поток с необработанным исключением, этот метод отмечен атрибутом `SecurityCritical`. Поэтому он может быть вызван только надстройкой Silverlight, но не кодом приложения.

При программировании потоков главная трудность состоит в обеспечении коммуникации между фоновым потоком и главным потоком приложения. Передать информацию в фоновый поток несложно с помощью параметров при его запуске.

Однако обмен информацией с потоком, когда он выполняется, и возвращение данных при завершении потока — сложные задачи. Чтобы предотвратить одновременное обращение к одним и тем же данным из двух разных потоков, часто приходится использовать блокировки. Для предотвращения доступа фонового потока к элементам пользовательского интерфейса используется диспетчеризация. Хуже всего то, что ошибки потоков не порождают предупреждения и ошибки во время компиляции, а во время выполнения не всегда приводят к аварийному завершению. Часто они порождают тонкие, незаметные проблемы, не проявляющиеся во время диагностики. Правила безопасного применения потоков рассматриваются в следующих разделах.

## Маршализация кода в поток пользовательского интерфейса

Как и клиентские приложения .NET (например, приложения WPF и Windows Forms), платформа Silverlight поддерживает *модель однопоточного выполнения* (single-threaded apartment model). В этой модели один поток управляет всем приложением и владеет всеми объектами, представляющими пользовательский интерфейс. Поток, создавший объект, владеет им. Другие потоки не могут взаимодействовать с объектом непосредственно. При нарушении этого правила (например, при попытке обратиться к объекту пользовательского интерфейса из другого потока) могут возникнуть блокировки, исключения или более тонкие проблемы.

Для поддержания работоспособности многопоточного приложения используется объект *диспетчера*. Он владеет главным потоком приложения, *маршализирует* коды (распределяет фрагменты кодов по потокам) и управляет очередью рабочих компонентов. При выполнении приложения диспетчер принимает рабочие запросы и выполняет их по очереди.

---

**Примечание.** Диспетчер — это экземпляр класса `System.Windows.Threading.Dispatcher`, впервые появившегося в WPF.

---

Диспетчер можно извлечь из любого элемента с помощью свойства `Dispatcher`. Класс `Dispatcher` содержит только два члена: метод `CheckAccess()`, позволяющий выяснить, может ли поток взаимодействовать с пользовательским интерфейсом, и метод `BeginInvoke()`, маршализующий код в главный поток приложения.

---

**Совет.** В окнах подсказки Visual Studio метод `Dispatcher.CheckAccess()` не выводится. Тем не менее его можно использовать в коде.

---

Приведенный ниже код реагирует на щелчок на кнопке созданием нового объекта `System.Threading.Thread`. Затем созданный поток используется для запуска небольшого фрагмента кода, который изменяет текстовое поле на текущей странице.

```
private void cmdBreakRules_Click(object sender, RoutedEventArgs e)
{
  Thread thread = new Thread(UpdateTextWrong);
  thread.Start();
}

private void UpdateTextWrong()
{
  // С задержкой 5 секунд выполняется запись
  Thread.Sleep(TimeSpan.FromSeconds(5));
  txt.Text = "Некоторый текст.";
}
```

Код обречен на неудачу. Метод `UpdateTextWrong()` выполняется в новом потоке, который не должен обращаться непосредственно к объектам пользовательского интерфейса Silverlight. В результате будет сгенерировано исключение `UnauthorizedAccessException`, нарушающее работу кода.

Чтобы исправить код, нужно получить ссылку на диспетчер, владеющий объектом `TextBox` (этот же диспетчер владеет страницей и другими объектами пользовательского интерфейса). Получив доступ к диспетчеру, можно вызвать метод `Dispatcher.BeginImvoke()` для маршализации кода в поток диспетчера. Важно отметить, что метод `BeginImvoke()` включает код в расписание задач диспетчера. После этого диспетчер выполняет код.

Ниже приведен правильный код.

```
private void cmdFollowRules_Click(object sender, RoutedEventArgs e)
{
  Thread thread = new Thread(UpdateTextRight);
  thread.Start();
}

private void UpdateTextRight()
{
  // Задержка на 5 секунд
  Thread.Sleep(TimeSpan.FromSeconds(5));

  // Получение диспетчера текущей страницы и
  // обновление текстового поля
  this.Dispatcher.BeginInvoke((ThreadStart) delegate()
    {
      txt.Text = "Некоторый текст.";
    }
  );
}
```

Метод `Dispatcher.BeginInvoke()` принимает единственный параметр — делегат, указывающий на метод, который содержит выполняемый код. Этот метод может находиться где-нибудь в другом месте приложения. Можно также использовать анонимный метод для выполнения встроенного кода (как в данном примере). Встроенный код рекомендуется использовать для простых задач, например для обновления одной строки. Однако для решения более сложных задач лучше добавить выполняемый код в отдельный метод следующим образом:

```
private void UpdateTextRight()
{
    // Имитация работы на протяжении 5 секунд
    Thread.Sleep(TimeSpan.FromSeconds(5));

    // Получение диспетчера текущей страницы
    // и обновление текста
    this.Dispatcher.BeginInvoke(SetText);
}

private void UpdateTextRight()
{
    txt.Text = "Here is some new text.";
}
```

**Примечание.** Метод `BeginInvoke()` возвращает объект `DispatcherOperation` (в предыдущем примере он не используется). Объект `DispatcherOperation` позволяет отследить статус диспетчеризации, а также выяснить, когда код фактически выполняется. Однако объект `DispatcherOperation` редко бывает полезным, потому что код, передаваемый в метод `BeginInvoke()`, обычно выполняется короткое время.

Трудоемкую фоновую операцию необходимо выполнять в отдельном потоке, а затем направить результат в поток диспетчера (в этот момент фактически обновляется пользовательский интерфейс или изменяется совместно используемый объект). Выполнять трудоемкую фоновую операцию в коде метода, передаваемого методу `BeginInvoke()`, не имеет смысла. Рассмотрим немного измененный код. Он вполне работоспособен, однако в реальных задачах такой подход никогда не применяется.

```
private void UpdateTextRight()
{
  // Получение диспетчера текущей страницы
  this.Dispatcher.BeginInvoke((ThreadStart) delegate()
    {
      // Трудоемкая операция
      Thread.Sleep(TimeSpan.FromSeconds(5));
      txt.Text = "Некоторый текст.";
    }
  );
}
```

Проблема состоит в том, что вся работа выполняется в потоке диспетчера. Это означает, что код связывает диспетчер так же, как однопоточное приложение. Зачем же было огород городить?

## Создание оболочки потока

В предыдущих примерах продемонстрировано обновление пользовательского интерфейса непосредственно в фоновом потоке. Однако такой подход не идеален. При его использовании создается запутанный код, в котором рабочие операции перемешаны с выводом данных. В результате приложение получается сложным и негибким. Его тяжело обновлять. Например, если в предыдущем примере понадобится изменить имя текстового поля или заменить его другим элементом управления, то, кроме кода интерфейса, придется редактировать также код потока.

Лучший подход состоит в создании потока, который передает информацию обратно в главный поток и предоставляет приложению возможность самому позаботиться о выводе информации на экран. Чтобы облегчить реализацию такого подхода, код потока и данные обычно заключают в отдельный класс. В него можно добавить свойства специально для ввода и вывода информации. Такой пользовательский класс называется *оболочкой потока*.

Перед созданием оболочки рекомендуется переместить все потоковые операции в базовый класс. Тогда можно будет использовать один и тот же шаблон кода для решения многих задач, не создавая повторно один и тот же код.

Рассмотрим базовый класс оболочки `ThreadWrapperBase`. Это абстрактный класс, поэтому создать его экземпляр нельзя. Вместо этого нужно создать производный класс.

```
public abstract class ThreadWrapperBase
{ ... }
```

Класс ThreadWrapperBase определяет два открытых свойства. Свойство Status возвращает один из трех элементов перечисления: Unstarted (Не начат), InProgress (Выполняется) и Completed (Завершен).

```
// Отслеживание статуса задачи
private StatusState status = StatusState.Unstarted;
public StatusState Status
{
  get { return status; }
}
```

Класс ThreadWrapperBase является оболочкой объекта Thread и предоставляет метод Start(), который создает и запускает поток.

```
// Это поток, в котором выполняется задача
private Thread thread;

// Запуск новой операции
public void Start()
{
  if (status == StatusState.InProgress)
  {
    throw new InvalidOperationException("Уже выполняется.");
  }
  else
  {
    // Инициализация новой задачи
    status = StatusState.InProgress;

    // Создание потока в фоновом режиме, чтобы он
    // был автоматически закрыт при завершении приложения
    thread = new Thread(StartTaskAsync);
    thread.IsBackground = true;

    // Запуск потока
    thread.Start();
  }
}
```

Поток выполняет закрытый метод StartTaskAsync(), который передает задачу двум другим методам: DoTask() и OnCompleted(). Вся работа (в данном примере вычисление простых чисел) выполняется методом DoTask(). Метод OnCompleted() генерирует событие завершения или выполняет обратный вызов для извещения клиента. Содержимое упомянутых операций зависит от решаемых задач, поэтому оба метода реализованы как абстрактные, т.е. переопределяемые в производном классе.

```
private void StartTaskAsync()
{
  DoTask();
  status = StatusState.Completed;
  OnCompleted();
}

// Переопределите эти методы
protected abstract void DoTask();
protected abstract void OnCompleted();
```

Теперь осталось создать производный класс, в котором используются упомянутые методы. В следующих разделах рассматривается практический пример оболочки, реализующей алгоритм вычисления простых чисел.

## Создание рабочего класса

Рассмотрим пример, в котором приложение вычисляет простые числа в заданном диапазоне методом просеивания. Другое название метода — решето Эратосфена (древнегреческий математик Эратосфен изобрел его приблизительно в 240 году до нашей эры). Сначала код создает список всех целых чисел в заданном диапазоне. Затем из списка удаляются числа, кратные простым числам, которые меньше квадратного корня максимального числа или равны ему. Оставшиеся числа являются простыми.

Мы не будем касаться теории, доказывающей работоспособность алгоритма. Приведем лишь простой код, реализующий его. Также не пытайтесь оптимизировать код или сравнить его с другими методами. Наша задача сейчас — реализовать алгоритм Эратосфена в фоновом потоке.

Полный код класса `FindPrimesThreadWrapper` можно найти в примерах для данной главы. Как и любой класс, производный от `ThreadWrapperBase`, класс `FindPrimesThresdWrapper` должен содержать четыре компонента.

- Поля или свойства, в которых находятся начальные данные. В рассматриваемом примере начальными данными служат границы диапазона.
- Поля или свойства, в которых сохраняются результаты. В данном примере результатом является список простых чисел, сохраняемый в массиве.
- Переопределенный метод `DoTask()`, в котором фактически реализован алгоритм Эратосфена. В методе используются начальные данные, а результаты работы метода записываются в поле класса.
- Переопределенный метод `OnCompleted()`, который генерирует событие завершения. Обычно событие завершения предоставляет объект типа `EventArgs`, содержащий результирующие данные. В рассматриваемом примере класс `FindPrimesCompletedEventArgs` содержит значения, определяющие диапазон и массив простых чисел.

Ниже приведен код класса `FindPrimesThreadWrapper`.

```
public class FindPrimesThreadWrapper : ThreadWrapperBase
{
  // Хранение входной и выходной информации
  private int fromNumber, toNumber;
  private int[] primeList;

  public FindPrimesThreadWrapper(int from, int to)
  {
    this.fromNumber = from;
    this.toNumber = to;
  }

  protected override void DoTask()
  {
    // Здесь находится код, вычисляющий массив простых чисел
    // (см. загружаемый код примера)
  }
```

```
  public event EventHandler<FindPrimesCompletedEventArgs> Completed;
  protected override void OnCompleted()
  {
    // Генерация события завершения
    if (Completed != null)
        Completed(this,
          new FindPrimesCompletedEventArgs(fromNumber,
             toNumber, primeList));
  }
}
```

Важно отметить, что данные, используемые в классе `FindPrimesThreadWrapper` (значения границ диапазона и массив простых чисел), не предоставляются открыто. Это предотвращает обращение к ним из главного потока приложения во время работы фонового потока. Если необходимо сделать массив простых чисел доступным, лучше всего добавить в код открытое свойство, проверяющее значение свойства `ThreadWrapperBase.State` и заполняемое по завершении потока.

Рекомендуется известить пользователя о завершении потока с помощью обратного вызова или события. Однако важно учитывать, что событие, сгенерированное в фоновом потоке, продолжает выполняться в этом же потоке независимо от того, где определен его код. Это означает, что при обработке события `Completed` необходимо использовать код диспетчеризации для передачи управления главному потоку приложения перед обновлением пользовательского интерфейса или любых данных текущей страницы.

---

**Примечание.** Если нужно предоставить один и тот же объект двум потокам, в которых он будет использоваться одновременно, необходимо защитить доступ к объекту средствами блокировки. Как и в полнофункциональном приложении .NET, для получения монопольного доступа к объекту, находящемуся в памяти, используется ключевое слово `lock`. Однако блокировка усложняет приложение и может породить ряд проблем. Производительность приложения ухудшается, потому что другие потоки, пытающиеся обратиться к объекту, должны ждать снятия блокировки. Кроме того, может возникнуть взаимоблокировка, когда два потока пытаются заблокировать один и тот же объект.

---

## Использование оболочки потока

Рассмотрим приложение Silverlight, в котором используется класс `FindPrimesThreadWrapper` (рис. 16.1). Пользователь задает диапазон. При щелчке на кнопке Найти простые числа начинается выполнение фонового потока. Когда вычисление заканчивается, список простых чисел выводится в элемент `DataGrid`.

В момент щелчка приложение отключает кнопку `cmdFind` (на ней написано Найти простые числа), чтобы предотвратить создание нескольких потоков. В принципе это возможно, но сбивает с толку пользователя. Затем приложение извлекает значения границ диапазона, создает объект `FindPrimesThreadWrapper`, подключает обработчик к событию `Completed` и вызывает метод `Start()`, чтобы начать вычисление.

```
private FindPrimesThreadWrapper threadWrapper;

private void cmdFind_Click(object sender, RoutedEventArgs e)
{
  // Отключение кнопки и очистка полей
  // от предыдущего результата
  cmdFind.IsEnabled = false;
  gridPrimes.ItemsSource = null;
```

```
  // Извлечение границ диапазона
  int from, to;
  if (!Int32.TryParse(txtFrom.Text, out from))
  {
    lblStatus.Text = "Неправильная нижняя граница.";
    return;
  }
  if (!Int32.TryParse(txtTo.Text, out to))
  {
    lblStatus.Text = "Неправильная верхняя граница.";
    return;
  }

  // Запуск потока
  threadWrapper = new FindPrimesThreadWrapper(from, to);
  threadWrapper.Completed += threadWrapper_Completed;
  threadWrapper.Start();

  lblStatus.Text = "Выполняется поиск...";
}
```

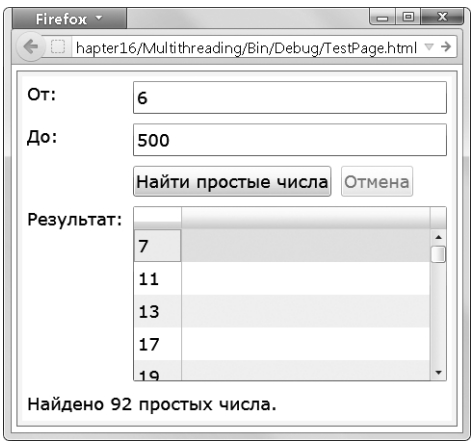


*Рис. 16.1. Список простых чисел*

Когда задача выполняется, приложение реагирует на действия пользователя. Пользователь может щелкать на других элементах управления, вводить числа в текстовые поля и т.п., не обращая внимание на то, что процессор занят фоновым потоком.

По завершении задачи генерируется событие `Completed`, в результате чего список извлекается из класса и выводится в решетку.

```
private void threadWrapper_Completed(object sender,
 FindPrimesCompletedEventArgs e)
{
  FindPrimesThreadWrapper thread =
  (FindPrimesThreadWrapper)sender;

  this.Dispatcher.BeginInvoke(delegate()
    {
      if (thread.Status == StatusState.Completed)
        {
```

```
            int[] primes = e.PrimeList;
            lblStatus.Text = "Найдено " + primes.Length +
              " простых чисел.";
            gridPrimes.ItemsSource = primes;
          }

          cmdFind.IsEnabled = true;
        }
      );
}
```

## Управление потоком

Когда базовая инфраструктура приложения готова, несложно добавить в нее индикатор прогресса и средства отмены потока.

Чтобы организовать отмену потока, нужно добавить булево поле, сигнализирующее о необходимости прерывания. Код потока периодически проверяет значение булева поля и, когда оно равно `true`, закрывает поток. Ниже приведен код, добавленный в класс `ThreadWrapperBase`.

```
// Флажок, сигнализирующий о необходимости закрыть поток
private bool cancelRequested = false;
protected bool CancelRequested
{
  get { return cancelRequested; }
}

// Чтобы отменить задачу, нужно вызвать этот метод
public void RequestCancel()
{
  cancelRequested = true;
}

// Для отмены нужно вызвать метод OnCancelled(),
// чтобы сгенерировать событие Cancelled
public event EventHandler Cancelled;
protected void OnCancelled()
{
  if (Cancelled != null)
   Cancelled(this, EventArgs.Empty);
}
```

Ниже приведен код, добавленный в метод `FindPrimesThreadWrapper.DoWork()`, для периодической проверки флажка отмены. Проверять флажок в каждой итерации не нужно. Проверка выполняется в одной из ста итераций.

```
int iteration = list.Length / 100;
if (i % iteration == 0)
{
  if (CancelRequested)
  {
    return;
  }
}
```

Кроме того, нужно изменить метод `ThreadWrapperBase.StartTaskAsync()`, чтобы он распознавал, как был завершен поток: в результате отмены или нормального завершения операции.

```
private void StartTaskAsync()
{
  DoTask();
  if (CancelRequested)
  {
    status = StatusState.Cancelled;
    OnCancelled();
  }
  else
  {
    status = StatusState.Completed;
    OnCompleted();
  }
}
```

Необходимо также подключить обработчик события `Cancelled` и добавить кнопку Отмена. Ниже приведен код, инициирующий запрос отмены текущей задачи.

```
private void cmdCancel_Click(object sender,
 RoutedEventArgs e)
{
  threadWrapper.RequestCancel();
}
```

А это — обработчик, выполняющийся по завершении процедуры отмены.

```
private void threadWrapper_Cancelled(object sender, EventArgs e)
{
  this.Dispatcher.BeginInvoke(delegate() {
    lblStatus.Text = "Поиск отменен.";
    cmdFind.IsEnabled = true;
    cmdCancel.IsEnabled = false;
  });
}
```

Учитывайте, что поток Silverlight нельзя остановить с помощью метода `Abort()`, поэтому все, что можно сделать, — это добавить в код потока условный оператор, проверяющий состояние флажка, и команду, инициирующую завершение потока изнутри.

# Класс `BackgroundWorker`

При использовании класса `Thread` потоки создаются вручную. Разработчик должен определить экземпляр объекта `Thread`, создать асинхронный код, организовать его взаимодействие с приложением и запустить код на выполнение с помощью метода `Thread.Start()`. Такой подход весьма эффективен, потому что класс `Thread` ничего не скрывает от разработчика. Разработчик может создать десятки потоков, передать им информацию в любой момент времени, приостановить любой из них с помощью метода `Thread.Sleep()` и т.п. Однако данный подход также довольно опасен. При совместном обращении к данным необходимо применять блокировки для предотвращения малозаметных ошибок. Если потоки создаются часто или в большом количестве, объекты потоков потребляют слишком много ресурсов.

Класс `System.ComponentModel.BackgroundWorker` предоставляет простой и безопасный способ создания потоков. Впервые он был добавлен в .NET 2.0 для упрощения создания потоков в приложениях Windows Forms. В классе `BackgroundWorker` неявно используется диспетчер потоков. Кроме того, класс `BackgroundWorker`

перемещает средства диспетчеризации за пределы модели событий, что существенно упрощает код.

Для облегчения кодирования все подробности создания потоков и управления ими в классе BackgroundWorker скрыты от разработчика. В класс добавлены средства индикации прогресса и отмены потока. Благодаря этому класс BackgroundWorker является наиболее практичным инструментом создания многопоточных приложений Silverlight.

---

**Примечание.** Лучше всего класс BackgroundWorker приспособлен для решения изолированной асинхронной задачи в фоновом режиме от начала до конца потока. Если нужно что-либо иное (например, асинхронный поток, выполняющийся на протяжении всего жизненного цикла приложения, или поток, непрерывно сообщающийся с приложением), то лучше применить класс Thread.

---

## Создание объекта BackgroundWorker

Для применения класса BackgroundWorker нужно создать его экземпляр в коде и программно подключить обработчики к событиям класса. Наиболее полезны события DoWork, ProgressChanged и RunWorkerCompleted. Каждое из них рассматривается в следующем примере.

---

**Совет.** Для решения многих асинхронных задач довольно часто создают несколько объектов BackgroundWorker и сохраняют их в коллекции. В рассматриваемом далее примере используется один объект BackgroundWorker, создаваемый в коде в момент первой инициализации страницы.

---

Ниже приведен код инициализации, обеспечивающий поддержку индикатора прогресса и средств отмены, а также подключающий обработчики к событиям. Этот код находится в конструкторе страницы BackgroundWorkerTest.

```
private BackgroundWorker backgroundWorker =
 new BackgroundWorker();

public BackgroundWorkerTest()
{
  InitializeComponent();

  backgroundWorker.WorkerReportsProgress = true;
  backgroundWorker.WorkerSupportsCancellation = true;
  backgroundWorker.DoWork += backgroundWorker_DoWork;
  backgroundWorker.ProgressChanged +=
   backgroundWorker_ProgressChanged;
  backgroundWorker.RunWorkerCompleted +=
   backgroundWorker_RunWorkerCompleted;
}
```

## Выполнение потока BackgroundWorker

Рассмотрим пример использования класса BackgroundWorker для вычисления массива простых чисел. Сначала нужно создать пользовательский класс, передающий входные параметры в объект BackgroundWorker. При вызове метода BackgroundWorker.RunWorkerAsync() ему можно передать любой объект. Этот объект будет автоматически предоставлен событию DoWork. Однако передать можно только один объект. Следовательно, чтобы передать два числа (границы диапазона), необходимо заключить их в один класс.

```
public class FindPrimesInput
{
  public int From
  { get; set; }

  public int To
  { get; set; }

  public FindPrimesInput(int from, int to)
  {
    From = from;
    To = to;
  }
}
```

Для запуска потока нужно вызвать метод `BackgrounWorker.RunWorkerAsync()` и передать ему объект `FindPrimesInput`. Ниже приведен код, выполняющий эту операцию после щелчка на кнопке `cmdFind`.

```
private void cmdFind_Click(object sender, RoutedEventArgs e)
{
  // Отключение кнопки и очистка предыдущих результатов
  cmdFind.IsEnabled = false;
  cmdCancel.IsEnabled = true;
  lstPrimes.Items.Clear();

  // Извлечение диапазона из текстовых полей
  int from, to;
  if (!Int32.TryParse(txtFrom.Text, out from))
  {
    MessageBox.Show("Неправильная нижняя граница.");
    return;
  }
  if (!Int32.TryParse(txtTo.Text, out to))
  {
    MessageBox.Show("Неправильная верхняя граница.");
    return;
  }

  // Запуск потока, вычисляющего простые числа
  FindPrimesInput input = new FindPrimesInput(from, to);
  backgroundWorker.RunWorkerAsync(input);
}
```

Когда объект `BackgroundWorker` начинает выполняться, он генерирует событие `DoWork` в отдельном потоке. Вместо создания нового потока (это потребовало бы накладных расходов) объект `BackgroundWorker` "заимствует" существующий поток из пула времени выполнения. По завершении задачи объект `BackgroundWorker` возвращает поток в пул для повторного использования другими задачами. Потоки пула используются также в асинхронных операциях, таких как получение ответа вебслужбы, загрузка страницы или создание сокетного соединения.

---

**Примечание.** Пул содержит набор существующих потоков. При одновременном выполнении большого количества асинхронных задач пул может исчерпаться. В этом случае новые потоки не создаются. Поступающие задачи ставятся в очередь и находятся в ней, пока не будет освобожден очередной поток. Это сделано для повышения производительности и предотвращения засорения операционной системы сотнями потоков.

---

Всю трудоемкую работу выполняет обработчик события DoWork. Нужно быть аккуратным, чтобы не допустить совместного использования данных (например, полей других классов) или объектов интерфейса. По завершении работы объект BackgroundWorker генерирует событие RunWorkerCompleted, чтобы известить об этом приложение. Событие генерируется в потоке диспетчера, что позволяет обменяться данными с пользовательским интерфейсом, не порождая проблемы совместного использования данных.

Получив поток, объект BackgroundWorker генерирует событие DoWork. В обработчик события DoWork можно вставить вызов метода Worker.FindPrimes(). Событие DoWork предоставляет объект DoWorkEventArgs, используемый для получения и возвращения информации. Входные данные извлекаются из свойства DoWorkEventArgs.Argument, а результаты возвращаются посредством свойства DoWorkEventArgs.Result.

```
private void backgroundWorker_DoWork(object sender,
 DoWorkEventArgs e)
{
  // Получение входных значений
  FindPrimesInput input = (FindPrimesInput)e.Argument;

  // Запуск потока
  int[] primes = Worker.FindPrimes(input.From, input.To);

  // Возвращение результата в пользовательский интерфейс
  e.Result = primes;
}
```

По завершении метода объект BackgroundWorker генерирует событие RunWorkerCompleted в потоке диспетчера. В этот момент можно извлечь результат из свойства RunWorkerCompletedEventArgs.Result. Полученный таким образом результат позволяет без проблем обновить пользовательский интерфейс, обращаясь к переменным на уровне страницы.

```
private void backgroundWorker_RunWorkerCompleted(
 object sender, RunWorkerCompletedEventArgs e)
{
  if (e.Error != null)
  {
    // Обработчик DoWork сгенерировал ошибку
    MessageBox.Show(e.Error.Message, "Ошибка!");
  }
  else
  {
    int[] primes = (int[])e.Result;
    foreach (int prime in primes)
    {
      lstPrimes.Items.Add(prime);
    }
  }
  cmdFind.IsEnabled = true;
  cmdCancel.IsEnabled = false;
  progressBar.Width = 0;
}
```

Обратите внимание на то, что блокировка не нужна. Кроме того, не используется метод Dispatcher.BeginInvoke(). Объект BackgroundWorker взял всю "грязную" работу на себя.

## Индикация прогресса

Класс BackgroundWorker предоставляет также встроенные средства индикации процента выполнения фоновой задачи. С их помощью пользователь может приблизительно оценить, сколько времени осталось до завершения задачи.

Чтобы установить в приложении индикатор прогресса, нужно присвоить значение true свойству BackgroundWorker.WorkerReportsProgress. Вывод информации о проценте выполнения состоит из нескольких этапов. Сначала обработчик события DoWork должен вызвать метод BackgroundWorker.ReportProgress() и предоставить ему приблизительный процент выполнения (от 0 до 100). Метод ReportProgress можно вызывать в каждой итерации, однако, чтобы не терять время, обычно в коде вызывают его в каждой десятой или сотой итерации. Метод ReportProgress() генерирует событие ProgressChanged. В обработчике события ProgressChanged можно прочесть текущий процент выполнения и обновить пользовательский интерфейс. Событие ProgressChanged генерируется в потоке пользовательского интерфейса, поэтому использовать метод Dispatcher.BeginInvoke() нет необходимости.

Рассматриваемый в качестве примера метод FindPrimes() сообщает об изменении с шагом 1% с помощью следующего кода:

```
int iteration = list.Length / 100;
for (int i = 0; i < list.Length; i++)
{
   ...

  // Событие генерируется, только когда произошло
  // изменение на 1%. Кроме того, событие не генерируется,
  // если объекта BackgroundWorker нет или индикация
  // прогресса отключена.
  if ((i % iteration == 0) &&
   (backgroundWorker != null) &&
     backgroundWorker.WorkerReportsProgress)
  {
    backgroundWorker.ReportProgress(i / iteration);
  }
}
```

**Примечание.** Рабочий код должен иметь доступ к объекту BackgroundWorker, чтобы вызвать метод ReportProgress(). В данном примере конструктор класса FindPrimesWorker принимает ссылку на объект BackgroundWorker. Объект FindPrimewWorker использует объект BackgroundWorker для индикации прогресса и отмены потока.

Когда свойство BackgroundWorker.WorkerReportsProgress установлено, на изменение можно отреагировать путем обработки события ProgressChanged. Однако в Silverlight (в отличие от полнофункциональной среды .NET) нет специализированного элемента управления, выводящего индикатор прогресса, поэтому его нужно создать вручную. Конечно, можно вывести процент выполнения в текстовом блоке, однако лучше имитировать графический индикатор прогресса с помощью встроенных элементов управления Silverlight. В данном примере индикатор прогресса состоит из двух прямоугольников Rectangle (один — для вывода фона, другой — для отображения процента выполнения) и текстового блока TextBlock, содержащего числовое значение процента выполнения. Все три элемента размещены в одной ячейке Grid, поэтому они перекрываются.

```
<Rectangle x:Name="progressBarBackground" Fill="AliceBlue"
 Stroke="SlateBlue" Grid.Row="4" Grid.ColumnSpan="2"
 Margin="5" Height="30" />
<Rectangle x:Name="progressBar" Width="0"
 HorizontalAlignment="Left" Grid.Row="4" Grid.ColumnSpan="2"
 Margin="5" Fill="SlateBlue" Height="30" />
<TextBlock x:Name="lblProgress" HorizontalAlignment="Center"
 Foreground="White" VerticalAlignment="Center" Grid.Row="4"
 Grid.ColumnSpan="2" />
```

Чтобы индикатор прогресса выглядел одинаково при разных значениях ширины окна браузера, необходим приведенный ниже код, который реагирует на событие `SizeChanged` и растягивает индикатор прогресса.

```
private double maxWidth;

private void UserControl_SizeChanged(object sender,
 SizeChangedEventArgs e)
{
  maxWidth = progressBarBackground.ActualWidth;
}
```

Теперь необходимо обработать событие `BackgroundWorker.ProgressChanged`, отображая текущий процент выполнения.

```
private void backgroundWorker_ProgressChanged(object sender,
 ProgressChangedEventArgs e)
{
  progressBar.Width =
   (double)e.ProgressPercentage/100 * maxWidth;
  lblProgress.Text =
   ((double)e.ProgressPercentage/100).ToString("P0");
}
```

Кроме процента выполнения, через событие прогресса можно передать дополнительную информацию. Для этого используется перегруженная версия метода `ReportProgress()`, принимающая два параметра. Первый параметр — процент выполнения, второй — любой пользовательский объект, посредством которого можно передать дополнительную информацию. В примере с вычислением простых чисел через него можно передать количество найденных чисел или последнее найденное число. Ниже приведен измененный код, задающий возвращение последнего найденного простого числа вместе с процентом выполнения.

```
backgroundWorker.ReportProgress(i / iteration, i);
```

В обработчике события `ProgressChanged` можно извлечь последнее число и вывести его на экран.

```
if (e.UserState != null)
 lblStatus.Text = "Последнее найденное простое число: " +
  e.UserState.ToString();
```

На рис. 16.2 показан индикатор прогресса во время работы фонового потока.

## Поддержка отмены задачи

При использовании класса `BackgroundWorker` для решения длительной задачи часто полезен код ее отмены. В первую очередь нужно присвоить значение `true` свойству `BackgroundWorker.WorkerSupportsCancellation`.

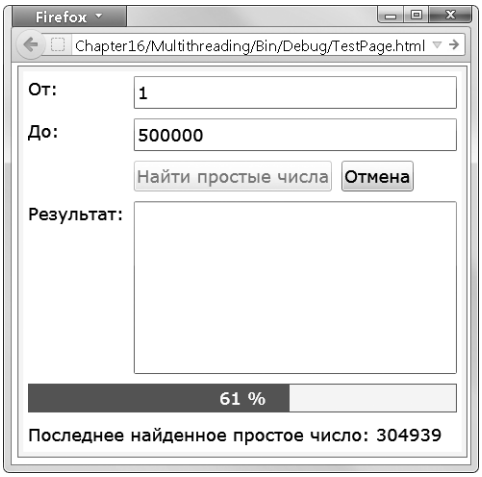


*Рис. 16.2. Индикатор прогресса, отображающий процент выполнения асинхронной задачи*

Чтобы отменить выполнение потока, код должен вызвать метод `BackgroundWorker.CancelAsync()`. Ниже приведен код вызова этого метода при щелчке на кнопке Отмена.

```
private void cmdCancel_Click(object sender,
 RoutedEventArgs e)
{
  backgroundWorker.CancelAsync();
}
```

Однако при вызове метода `CancelAsync()` отмены потока не происходит. Метод всего лишь извещает поток о необходимости отмены. Код потока должен явно проверить наличие запроса на отмену, выполнить необходимые операции, включая очистку, и только после этого возвратить управление. Ниже приведен код метода `FindPrimes()`, проверяющий запрос на отмену непосредственно перед извещением о прогрессе.

```
for (int i = 0; i < list.Length; i++)
{
  ...
  if ((i % iteration) && (backgroundWorker != null))
  {
    if (backgroundWorker.CancellationPending)
    {
      // Завершиться, ничего больше не делая.
      return;
    }
    if (backgroundWorker.WorkerReportsProgress)
    {
      backgroundWorker.ReportProgress(i / iteration);
    }
  }
}
```

Кроме того, обработчик события DoWork должен явно присвоить значение true свойству DoWorkEventArgs.Cancel, чтобы завершить отмену. Затем можно завершить метод, не вычисляя строку простых чисел.

```
private void backgroundWorker_DoWork(object sender,
 DoWorkEventArgs e)
{
  FindPrimesInput input = (FindPrimesInput)e.Argument;
  int[] primes = Worker.FindPrimes(input.From, input.To,
   backgroundWorker);

  if (backgroundWorker.CancellationPending)
  {
    e.Cancel = true;
    return;
  }

  // Возвращение результатов
  e.Result = primes;
}
```

Событие RunWorkerCompleted генерируется, даже если задача отменена. В этот момент можно проверить, была ли отменена задача, и выполнить необходимые операции.

```
private void backgroundWorker_RunWorkerCompleted(
 object sender, RunWorkerCompletedEventArgs e)
{
  if (e.Cancelled)
  {
    MessageBox.Show("Задача отменена");
  }
  else if (e.Error != null)
  {
    // Обработчик события DoWork генерирует ошибку
    MessageBox.Show(e.Error.Message, "Произошла ошибка!");
  }
  else
  {
    int[] primes = (int[])e.Result;
    foreach (int prime in primes)
    {
      lstPrimes.Items.Add(prime);
    }
  }
  cmdFind.IsEnabled = true;
  cmdCancel.IsEnabled = false;
  progressBar.Value = 0;
}
```

Теперь объект BackgroundWorker позволяет как запустить операцию поиска простых чисел, так и отменить ее в процессе выполнения.

# Резюме

В этой главе были рассмотрены два мощных средства добавления многопоточности в приложение Silverlight. Конечно, возможность создания многопоточного приложения не означает, что каждое приложение нужно делать многопоточным.

Многопоточность существенно усложняет приложение и при неаккуратном использовании может привести к малозаметным ошибкам, особенно если приложение выполняется в разных операционных системах и на разном оборудовании. Поэтому официальные руководства Microsoft рекомендуют применять многопоточность, только тщательно взвесив все "за" и "против". Несомненно, многопоточность следует применять, когда в приложении есть длительные операции, во время выполнения которых необходимо обеспечить постоянную готовность интерфейса. В большинстве случаев лучше применить удобный высокоуровневый класс `BackgroundWorker`, а не низкоуровневый класс `Thread`. Когда класс `Thread` все же необходим, не создавайте больше одного-двух фоновых потоков. Рекомендуется также обрабатывать в потоках разные фрагменты информации, как можно меньше взаимодействующие друг с другом, чтобы избежать усложнений, связанных с блокировкой и синхронизацией.