## 第14章 并发和异步:

大多数应用程序需要处理一次发生的多件事情（并发）。在本章中，我们从基本的先决条件开始，即线程和任务的基础，然后详细描述异步的原理和C#的异步函数。

在第21章中，我们将更详细地回顾多线程，在第22章中，将介绍并行编程的相关主题。

### 简介

下面是最通常的并发场景:

*编写响应式用户界面*

在Windows Presentation Foundation（WPF）、移动和Windows窗体应用程序中，必须与运行用户界面的代码同时运行耗时的任务，以保持响应能力。

*允许同时处理请求*

在服务器上，客户端请求可以并发到达，因此必须并行处理以保持可伸缩性。如果您使用ASP.NET核心或Web API，运行时会自动为您执行此操作。但是，您仍然需要了解共享状态（例如，使用静态变量进行缓存的效果）。

*并行编程*

如果工作负载在内核之间分配，执行密集计算的代码可以在多核多处理器计算机上更快地执行（第22章专门介绍这一点）。

*推测性执行*

在多核计算机上，有时可以通过预测可能需要做的事情，然后提前完成来提高性能。LINQPad使用这种技术来加快新查询的创建速度。一种变体是并行运行许多不同的算法，这些算法都能解决同一任务。无论哪一个先完成，都会“获胜”——当你无法提前知道哪种算法执行得最快时，这是有效的。

程序可以同时执行代码的通用机制称为多线程。CLR和操作系统都支持多线程，这是并发中的一个基本概念。因此，了解线程的基本知识，特别是线程对共享状态的影响是至关重要的。

### 线程(Threading)

线程是一个可以独立于其他线程进行的执行路径

每个线程都在一个操作系统进程中运行，该进程提供了一个程序运行的独立环境。对于单线程程序，只有一个线程在进程的隔离环境中运行，因此线程可以独占访问它。对于多线程程序，多个线程在单个进程中运行，共享相同的执行环境（尤其是内存）。这在一定程度上就是多线程之所以有用的原因：例如，一个线程可以在后台提取数据，而另一个线程在数据到达时显示数据。此数据称为共享状态。

#### 创建线程(Creating a Thread)

客户端程序（Console、WPF、UWP或Windows窗体）在操作系统自动创建的单个线程（“主”线程）中启动。在这里，除非你做了其他事情，通过创建更多的线程（直接或间接）, 它作为一个单线程应用程序度过了它的生命。

您可以通过实例化thread对象并调用其start方法来创建和启动新线程。Thread最简单的构造函数接受ThreadStart委托：一个无参数的方法，指示应该从哪里开始执行。以下是一个示例：

// NB: All samples in this chapter assume the following namespace imports:

using System;

using System.Threading;

Thread t = new Thread (WriteY); // Kick off a new thread

t.Start(); // running WriteY()

// Simultaneously, do something on the main thread.

for (int i = 0; i < 1000; i++) Console.Write ("x");

void WriteY()

{

for (int i = 0; i < 1000; i++) Console.Write ("y");

}

// Typical Output:

xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy

yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy

yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

...

主线程创建一个新线程t，在该线程上运行一个重复打印字符y的方法。同时，主线程重复打印字符x，如图14-1所示。在单核计算机上，操作系统必须为每个线程分配“时间片”（在Windows中通常为20ms）来模拟并发性，从而产生重复的x和y块。在多核或多处理器计算机上，这两个线程可以真正并行执行（受计算机上其他活动进程的竞争），尽管在本例中，由于控制台处理并发请求的机制的微妙性，您仍然会得到重复的x或y块。



提示:

一个线程在其执行与另一个线程上的代码执行交叉在一起的点上被称为抢占(preempted)。这个词经常出现在解释为什么出了问题时！

启动后，线程的IsAlive属性将返回true，直到线程结束。当传递给线程的构造函数的委托完成执行时，线程结束。结束后，线程将无法重新启动。

每个线程都有一个Name属性，您可以设置该属性以便于调试。这在Visual Studio中特别有用，因为线程的名称显示在“线程窗口”和“调试位置”工具栏中。您只能设置一次线程的名称；稍后尝试更改它将引发异常。

Thread.CurrentThead静态属性为你提供当前正在执行的线程:

Console.WriteLine (Thread.CurrentThread.Name);

#### 合并和休眠(Join and Sleep)

你可以调用另一个线程的Join方法, 来等待它结束。

Thread t = new Thread (Go);

t.Start();

t.Join();

Console.WriteLine ("Thread t has ended!");

void Go() { for (int i = 0; i < 1000; i++) Console.Write ("y"); }

这会打印1000次“y”，然后立即打印“线程t已结束！”。调用Join时可以包含超时，以毫秒为单位，也可以作为TimeSpan。如果线程结束，则返回true；如果线程超时，则返回false。

Thread.Sleep将暂停当前线一段指定的时间：

Thread.Sleep (TimeSpan.FromHours (1)); // Sleep for 1 hour

Thread.Sleep (500); // Sleep for 500 milliseconds

Thread.Sleep（0）立即放弃线程的当前时间片，自动将CPU移交给其他线程。Thread.Yield（）做同样的事情，只是它只向在同一处理器上运行的线程放弃。

提示：

Sleep（0）或Yield在高级性能调整的生产代码中偶尔有用。它也是一个很好的诊断工具，可以帮助发现线程安全问题：如果插入thread.Yield（）在你的代码中的任何地方破坏了程序，几乎可以肯定你有一个bug。

在等待睡眠或加入时，线程被阻止。

#### 阻塞（Blocking）

当一个线程由于某种原因暂停执行时，例如在休眠或等待另一个线程通过Join结束时，它被视为被阻塞。被阻塞的线程立即产生其处理器时间片，从那时起，在满足其阻塞条件之前，它不消耗处理器时间。您可以通过线程的ThreadState属性测试被阻止的线程：

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

提示：

ThreadState是一个标志枚举，以逐位方式组合了三层数据。但是，大多数值都是多余的、未使用的或不推荐使用的。以下扩展方法将ThreadState剥离为四个有用值之一：Unstarted、Running、WaitSleepJoin和Stopped：

public static ThreadState Simplify (this ThreadState ts)

{

return ts & (ThreadState.Unstarted |

ThreadState.WaitSleepJoin |

ThreadState.Stopped);

}

ThreadState属性可用于诊断目的，但不适用于同步，因为线程的状态可能在测试ThreadState和对该信息执行操作之间发生变化。

当线程阻塞或取消阻塞时，操作系统会执行上下文切换。这会产生较小的开销，通常为一到两微秒。

##### I/O-bound versus compute-bound

一个花费大部分时间等待事情发生的操作被称为I/O绑定——例如下载网页或调用Console.ReadLine。（I/O绑定操作通常涉及输入或输出，但这不是硬性要求：Thread.Sleep也被视为I/O绑定。）相比之下，将大部分时间用于执行CPU密集型工作的操作被称为计算机绑定。

##### Blocking versus spinning

I/O绑定操作的工作方式有两种：它要么在当前线程上同步等待，直到操作完成（如Console.ReadLine、thread.Sleep或thread.Join），要么异步操作，在将来操作完成时触发回调（稍后会详细介绍）。

同步等待的I/O绑定操作大部分时间都在阻塞线程。它们还可以周期性地在一个循环中“旋停”：

while (DateTime.Now < nextStartTime)

Thread.Sleep (100);

抛开有更好的方法（如定时器或信号结构）不谈，另一种选择是线程可以连续旋停：

while (DateTime.Now < nextStartTime)

一般来说，这在处理器时间上是非常浪费的：就CLR和OS而言，线程正在执行重要的计算，因此会相应地分配资源。实际上，我们已经将本应是I/O绑定的操作变成了计算绑定的操作。

提示：

关于旋停和阻塞，有一些细微差别。首先，当你期望一个条件很快得到满足时（可能在几微秒内）时，非常短暂的旋转是有效的，因为它避免了上下文切换的开销和延迟。NET提供了特殊的方法和类来提供帮助——请参阅在线补充“SpinLock和SpinWait”。

其次，阻塞不会产生零成本。这是因为每个线程在其生命周期内占用大约1MB的内存，并导致CLR和操作系统的持续管理开销。因此，在需要处理数百或数千个并发操作的I/O绑定严重的程序中，阻塞可能会很麻烦。相反，这样的程序需要使用基于回调的方法，在等待时完全取消线程。这（在一定程度上）就是我们稍后讨论的异步模式的目的。

#### Local Versus Shared State

CLR为每个线程分配自己的内存堆栈，以便将局部变量分开。在下一个示例中，我们使用局部变量定义一个方法，然后在主线程和新创建的线程上同时调用该方法：

new Thread (Go).Start(); // Call Go() on a new thread

Go(); // Call Go() on the main thread

void Go()

{

// Declare and use a local variable - 'cycles'

for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');

}

在每个线程的内存堆栈上创建一个单独的循环变量副本，因此可以预见，输出是10个问号。

如果线程对同一对象或变量有公共引用，则线程共享数据：

bool \_done = false;

new Thread (Go).Start();

Go();

void Go()

{

if (!\_done) { \_done = true; Console.WriteLine ("Done"); }

}

两个线程共享\_done变量，因此“完成”打印一次而不是两次。

lambda表达式捕获的局部变量也可以共享：

bool done = false;

ThreadStart action = () =>

{

if (!done) { done = true; Console.WriteLine ("Done"); }

};

new Thread (action).Start();

action();

不过，更常见的情况是，字段用于在线程之间共享数据。在以下示例中，两个线程都在同一ThreadTest实例上调用了Go（），因此它们共享相同的\_done字段：

var tt = new ThreadTest();

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

tt.Go();

class ThreadTest

{

bool \_done;

public void Go()

{

if (!\_done) { \_done = true; Console.WriteLine ("Done"); }

}

}

静态字段提供了另一种在线程之间共享数据的方式：

class ThreadTest

{

static bool \_done; // Static fields are shared between all threads

// in the same process.

static void Main()

{

new Thread (Go).Start();

Go();

}

static void Go()

{

if (!\_done) { \_done = true; Console.WriteLine ("Done"); }

}

}

所有四个例子都说明了另一个关键概念：线程安全性（或者更确切地说，缺乏它！）。输出实际上是不确定的：“完成”可能会打印两次（尽管可能性不大）。然而，如果我们在Go方法中交换语句的顺序，则“完成”打印两次的几率会急剧上升：

static void Go()

{

if (!\_done) { Console.WriteLine ("Done"); \_done = true; }

}

问题是，一个线程可能会在另一个线程执行WriteLine语句的同时评估if语句——在它有机会将done设置为true之前。

提示：

我们的示例说明了共享可写状态可能引入间歇性错误的多种方式之一，而多线程正是这种间歇性错误的罪魁祸首。接下来，我们来看看如何通过锁定程序修复；然而，最好尽可能避免共享状态。稍后我们将看到异步编程模式如何帮助实现这一点。

#### 加锁和线程安全（Locking and Thread Safety）

提示:

加锁和线程安全是一个非常大的论题. 全面的讨论, 请查看第890页的”排他锁”和第898页的”加锁与线程安全”

我们可以通过在读取和写入共享字段时获得独占锁来修复前面的示例。C#提供lock语句就是为了这个目的：

class ThreadSafe

{

static bool \_done;

static readonly object \_locker = new object();

static void Main()

{

new Thread (Go).Start();

Go();

}

static void Go()

{

lock (\_locker)

{

if (!\_done) { Console.WriteLine ("Done"); \_done = true; }

}

}

}

当两个线程同时争用一个锁（可以在任何引用类型的对象上；在本例中为\_locker）时，一个线程会等待或阻塞，直到该锁可用。在这种情况下，它确保一次只有一个线程可以进入其代码块，并且“Done”将只打印一次。以这种方式保护的代码——在多线程上下文中不受不确定性的影响——被称为线程安全。

注意:

即使是自动递增变量的行为也不是线程安全的：表达式x++在底层处理器上作为不同的读增量写操作执行。因此，如果两个线程在锁外同时执行x++，那么变量最终可能会增加一次，而不是两次（或者更糟的是，在某些条件下，x可能会被撕裂，最终出现新旧内容的逐位混合）。

锁定并不是线程安全的灵丹妙药——访问字段时很容易忘记锁定，而且锁定本身也会产生问题（例如死锁）。

可以使用锁定的一个很好的例子是ASP.NET应用程序中频繁访问的数据库对象的共享内存缓存。这种类似的应用程序很容易上手，而且不存在死锁的可能性。我们在第901页的“应用程序服务器中的线程安全”中给出了一个例子。

#### 向线程传递数据

有时，您需要将参数传递给线程的启动方法。最简单的方法是使用lambda表达式，该表达式使用所需的参数调用方法：

Thread t = new Thread ( () => Print ("Hello from t!") );

t.Start();

void Print (string message) => Console.WriteLine (message);

使用这种方法，可以向该方法传递任意数量的参数。您甚至可以将整个实现封装在多语句lambda中：

new Thread (() =>

{

Console.WriteLine ("I'm running on another thread!");

Console.WriteLine ("This is so easy!");

}).Start();

另一种（灵活性较低）技术是将参数传递到Thread的Start方法中：

Thread t = new Thread (Print);

t.Start ("Hello from t!");

void Print (object messageObj)

{

string message = (string) messageObj; // We need to cast here

Console.WriteLine (message);

}

这是因为Thread的构造函数被重载以接受两个委托中的任何一个：

public delegate void ThreadStart();

public delegate void ParameterizedThreadStart (object obj);

#### Lambda表达式和捕获的变量

正如我们所看到的，lambda表达式是将数据传递到线程的最方便、最强大的方法。但是，必须小心在启动线程后意外修改捕获的变量。例如，考虑以下内容：

for (int i = 0; i < 10; i++)

new Thread (() => Console.Write (i)).Start();

输出是不确定的！以下是一个典型的结果： 0223557799

问题是，i变量在循环的整个生命周期中都指向相同的内存位置。因此，每个线程都调用Console。写一个变量，它的值可以在运行时更改！解决方案是使用一个临时变量，如下所示：

for (int i = 0; i < 10; i++)

{

int temp = i;

new Thread (() => Console.Write (temp)).Start();

}

然后，数字0到9中的每一个都被精确地写入一次。（由于线程可以在不确定的时间启动，因此顺序仍然未定义。）

提示:

这类似于我们在第434页“捕获的变量”中描述的问题。问题不仅在于多线程，还在于C#在for循环中捕获变量的规则。

变量temp现在是每个循环迭代的本地变量。因此，每个线程捕获不同的内存位置，这是没有问题的。我们可以通过以下示例更简单地说明早期代码中的问题：

string text = "t1";

Thread t1 = new Thread ( () => Console.WriteLine (text) );

text = "t2";

Thread t2 = new Thread ( () => Console.WriteLine (text) );

t1.Start(); t2.Start();

因为两个lambda表达式都捕获相同的文本变量，所以t2被打印两次。

#### 异常处理

创建线程时生效的任何try/catch/finaly块在线程开始执行时都与线程无关。考虑以下程序：

try

{

new Thread (Go).Start();

}

catch (Exception ex)

{

// We'll never get here!

Console.WriteLine ("Exception!");

}

void Go() { throw null; } // Throws a NullReferenceException

此示例中的try/catch语句无效，并且新创建的线程将被未处理的NullReferenceException阻塞。当您考虑到每个线程都有一个独立的执行路径时，这种行为是有意义的。

补救方法是将异常处理程序移到Go方法中：

new Thread (Go).Start();

void Go()

{

try

{

...

throw null; // The NullReferenceException will get caught below

...

}

catch (Exception ex)

{

// Typically log the exception and/or signal another thread

// that we've come unstuck

...

}

}

您需要在生产应用程序中的所有线程入口方法上都有一个异常处理程序，就像您在主线程上所做的那样（通常在执行堆栈的更高级别）。一个未经处理的异常会导致整个应用程序关闭——出现一个丑陋的对话框！

提示:

在编写此类异常处理块时，很少会忽略错误：通常，您会记录异常的详细信息。对于客户端应用程序，您可能会显示一个对话框，允许用户自动将这些详细信息提交到web服务器。然后您可能会选择重新启动应用程序，因为意外的异常可能会使程序处于无效状态。

#### 中心化异常处理(Centralized exception handling)

在WPF、UWP和Windows窗体应用程序中，您可以订阅“全局”异常处理事件Application。DispatcherUnhandledException和应用程序。ThreadException。在通过消息循环调用的程序的任何部分中出现未处理的异常后（这相当于应用程序处于活动状态时在主线程上运行的所有代码），就会触发这些异常。这对于记录和报告错误非常有用（尽管它不会为您创建的工作线程上未处理的异常而触发）。处理这些事件可以防止程序关闭，尽管您可以选择重新启动应用程序以避免未处理的异常可能导致的潜在状态损坏。

#### 前台和后台线程(Foreground Versus Background Threads)

默认情况下，显式创建的线程是前台线程。只要其中任何一个线程在运行，前台线程就会使应用程序保持活动状态，而后台线程则不会。在所有前台线程完成后，应用程序结束，并且任何仍在运行的后台线程都会突然终止。

提示:

线程的前台/后台状态与其优先级（执行时间的分配）无关。

您可以使用线程的IsBackground属性查询或更改线程的后台状态：

static void Main (string[] args)

{

Thread worker = new Thread ( () => Console.ReadLine() );

if (args.Length > 0) worker.IsBackground = true;

worker.Start();

}

如果在没有参数的情况下调用此程序，则工作线程将处于前台状态，并将等待ReadLine语句，等待用户按Enter键。同时，主线程退出，但应用程序仍在运行，因为前台线程仍处于活动状态。另一方面，如果一个参数被传递给Main（），则工作线程被分配后台状态，并且程序几乎在主线程结束时立即退出（终止ReadLine）。

当进程以这种方式终止时，将绕过后台线程执行堆栈中的任何finally块。如果您的程序使用finally（或using）块来执行清理工作，例如删除临时文件，您可以通过在退出应用程序时显式地等待这些后台线程来避免这种情况，方法是加入线程或使用信令结构（请参阅第643页的“信令”）。在任何一种情况下，您都应该指定一个超时，这样您就可以在叛徒线程拒绝完成时放弃它；否则，您的应用程序将无法关闭，而用户无需从任务管理器（或Unix上的kill命令）寻求帮助。

前景线程不需要这种处理，但您必须小心避免可能导致线程无法结束的错误。应用程序无法正常退出的一个常见原因是存在活动的前台线程。

#### 线程优化级(Thread Priority)

线程的Priority属性确定相对于操作系统中的其他活动线程分配的执行时间，具体如下：

enum ThreadPriority { Lowest, BelowNormal, Normal,

AboveNormal, Highest }

当多个线程同时处于活动状态时，这一点就变得重要起来。提升线程的优先级时需要小心，因为它可能会使其他线程挨饿。如果希望某个线程的优先级高于其他进程中的线程，则还必须使用System.Diagnostics中的process类来提升进程优先级：

这可以很好地用于非UI进程，这些进程只需做最少的工作，并且在工作中需要低延迟（快速响应的能力）。对于计算机应用程序（尤其是那些具有用户界面的应用程序），提高进程优先级可能会使其他进程挨饿，从而减慢整个计算机的速度。

#### 信号(Signaling)

有时，您需要一个线程等待，直到收到来自其他线程的通知。这被称为信号。最简单的信号结构是ManualResetEvent。在ManualResetEvent上调用WaitOne会阻塞当前线程，直到另一个线程通过调用Set“打开”信号。在下面的示例中，我们启动一个等待ManualResetEvent的线程。它保持阻塞两秒钟，直到主线程发出信号：

var signal = new ManualResetEvent (false);

new Thread (() =>

{

Console.WriteLine ("Waiting for signal...");

signal.WaitOne();

signal.Dispose();

Console.WriteLine ("Got signal!");

}).Start();

Thread.Sleep(2000);

signal.Set(); // “Open” the signal

调用Set后，信号保持打开状态；您可以通过调用Reset再次关闭它。

ManualResetEvent是CLR提供的几个信号结构之一；我们将在第21章中详细介绍所有这些。

#### 富客户端应用程序中的线程

在WPF、UWP和Windows窗体应用程序中，在主线程上执行长时间运行的操作会使应用程序无响应，因为主线程还处理执行渲染和处理键盘和鼠标事件的消息循环。

一种流行的方法是启动“工作”线程进行耗时的操作。工作线程上的代码运行一个耗时的操作，然后在完成后更新UI。但是，所有富客户端应用程序都有一个线程模型，通过该模型，UI元素和控件只能从创建它们的线程（通常是主UI线程）访问。违反此规则会导致不可预测的行为或引发异常。

因此，当您想从工作线程更新UI时，必须将请求转发到UI线程（技术术语是封送）。实现这一点的低级方法如下（稍后，我们将讨论基于这些解决方案的其他解决方案）：

* 在WPF中，对元素的Dispatcher对象上调用BeginInvoke或Invoke。
* 在UWP应用程序中，对Dispatcher对象调用RunAsync
* 在Windows窗体中，调用控件的BeginInvoke或Invoke

所有这些方法都接受引用要运行的方法的委托。BeginInvoke/RunAsync通过将委托排入UI线程的消息队列（与处理键盘、鼠标和计时器事件的队列相同）来工作。Invoke执行相同的操作，但随后会阻塞，直到UI线程读取并处理完消息为止。正因为如此，Invoke允许您从方法中获取返回值。如果不需要返回值，BeginInvoke/RunAsync更可取，因为它们不会阻塞调用程序，也不会引入死锁的可能性（请参阅第896页的“死锁”）。

提示:

您可以想象，当您调用Application.Run时，执行以下伪代码：

while (!thisApplication.Ended)

{

wait for something to appear in message queue

Got something: what kind of message is it?

Keyboard/mouse message -> fire an event handler

User BeginInvoke message -> execute delegate

User Invoke message -> execute delegate & post result

}

正是这种循环使工作线程能够将委托封送到UI线程上执行。

为了进行演示，假设我们有一个WPF窗口，其中包含一个名为txtMessage的文本框，我们希望工作线程在执行耗时的任务（我们将通过调用thread.Sleep来模拟）后更新其内容。以下是我们的做法：

partial class MyWindow : Window

{

public MyWindow()

{

InitializeComponent();

new Thread (Work).Start();

}

void Work()

{

Thread.Sleep (5000); // Simulate time-consuming task

UpdateMessage ("The answer");

}

void UpdateMessage (string message)

{

Action action = () => txtMessage.Text = message;

Dispatcher.BeginInvoke (action);

}

}

多个UI线程

如果多个UI线程各自拥有不同的窗口，那么它们就有可能具有多个。主要情况是，您有一个具有多个顶级窗口的应用程序，通常称为单文档接口（SDI）应用程序，如Microsoft Word。每个SDI窗口通常在任务栏上显示为一个单独的“应用程序”，并且在功能上与其他SDI窗口基本隔离。通过给每个这样的窗口赋予其自己的UI线程，可以使每个窗口相对于其他窗口更具响应性。

运行此操作会立即显示一个响应窗口。五秒钟后，它会更新文本框。代码与Windows窗体类似，只是我们改为调用（Form的）BeginInvoke方法：

void UpdateMessage (string message)

{

Action action = () => txtMessage.Text = message;

this.BeginInvoke (action);

}

#### 同步上下文(Synchronization Contexts)

在System.ComponentModel命名空间中，有一个名为Synchronization Context的类，它可以实现线程通用化的编排(marshaling)。移动和桌面的富客户端API（UWP、WPF和Windows窗体）分别定义和实例化SynchronizationContext这个子类，它可以通过静态属性SynchronizaContext.Current获得（在UI线程上运行的时候）。捕获此属性可以让您稍后从工作线程“post”到UI控件：

partial class MyWindow : Window

{

SynchronizationContext \_uiSyncContext;

public MyWindow()

{

InitializeComponent();

// Capture the synchronization context for the current UI thread:

\_uiSyncContext = SynchronizationContext.Current;

new Thread (Work).Start();

}

void Work()

{

Thread.Sleep (5000); // Simulate time-consuming task

UpdateMessage ("The answer");

}

void UpdateMessage (string message)

{

// Marshal the delegate to the UI thread:

\_uiSyncContext.Post (\_ => txtMessage.Text = message, null);

}

}

这很有用，因为相同的技术适用于所有富客户端用户界面的API

调用Post相当于在Dispatcher或Control上调用BeginInvoke；还有一个与Invoke等效的Send方法。

#### 线程池(The Thread Pool)

无论何时启动一个线程，都要花费几百微秒来组织新的局部变量堆栈。线程池通过具有预先创建的可回收线程池来减少这种开销。线程池对于高效的并行编程和细粒度并发至关重要；它允许短操作运行，而不会被线程启动的开销淹没。

使用池线程时需要注意以下几点：

* + 不能设置池线程的名称，这会使调试更加困难（尽管在Visual Studio的“线程”窗口中调试时可以附加描述）。
  + 线程池的线程总是后台线程
  + 阻塞池线程会降低性能（请参阅第647页的“线程池中的卫生”）。

您可以自由更改池线程的优先级——当释放回池中时，它将恢复正常。

您可以通过Thread.CurrentThread.IsThreadPoolThread属来确定当池线程是否正在执行。

##### 进入线程池（Entering the thread pool）

在池线程上显式运行某些东西的最简单方法是使用Task.Run（我们将在下一节中对此进行更详细的介绍）：

// Task is in System.Threading.Tasks

Task.Run (() => Console.WriteLine ("Hello from the thread pool"));

因为以前NET Framework 4.0不存在Task静态类，一种常见的替代方法是调用ThreadPool.QueueUserWorkItem：

ThreadPool.QueueUserWorkItem(notUsed => Console.WriteLine ("Hello"));

提示：

下面将隐式的使用线程池

1. ASP.NET Core和WebAPI应用服务
2. System.Timers.Timer 和 System.Threading.Timer
3. 并行编程结构（将在第22章详细介绍）
4. 废弃的BackgroundWorker类

##### 线程池中的卫生（Hygiene in the thread pool）

线程池提供了另一个功能，即确保临时过量的计算绑定工作不会导致CPU超额订阅。超额订阅是指活动线程比CPU内核多的情况，操作系统必须对线程进行时间切片。超额订阅会损害性能，因为时间切片需要昂贵的上下文切换，并且可能会使CPU缓存失效，而这些缓存在向现代处理器提供性能方面已经变得至关重要。

CLR通过排队和限制任务的启动来防止线程池中的超额订阅。它首先运行与硬件核心一样多的并发任务，然后通过爬山算法调整并发级别，不断向特定方向调整工作负载。如果吞吐量提高，它将继续朝着相同的方向发展（否则将逆转）。这确保了它始终跟踪最佳性能曲线——即使面对计算机上的竞争过程活动。

如果满足以下两个条件，CLR的策略效果最佳：

* 工作项大多是短时间运行的（<250毫秒，理想情况下<100毫秒），因此CLR有很多机会进行测量和调整。
* 耗时多被阻塞的工作并没有占据主导地位

阻塞很麻烦，因为它给CLR一个错误的想法，即它正在加载CPU。CLR足够聪明，可以检测和补偿（通过向池中注入更多线程），尽管这会使池容易受到后续超额订阅的影响。它还可能引入延迟，因为CLR会抑制注入新线程的速率，尤其是在应用程序生命的早期（在客户端操作系统上更是如此，因为它倾向于降低资源消耗）。

当您想要充分利用CPU时（例如，通过第22章中的并行编程API），在线程池中保持良好的卫生状况尤其重要。

### Tasks

线程是一种用于创建并发的低等级工具，因此，它有局限性，尤其是以下方面：

* 尽管将数据传递到启动的线程中很容易，但要从合并（Join）的线程中获取“返回值”却没有简单的方法。您需要设置某种共享字段。如果线程中操作抛出异常，那么捕获和传播该异常同样痛苦。
* 你不能让线程在它完成时启动其他东西；相反，你必须合并（Join）加入它（在这个过程中是阻塞你自己的线程）。

这些限制阻碍了细粒度并发；换言之，它们使得通过组合较小的并发操作来组合较大的并发操作变得困难（这对于我们将在下面的部分中讨论的异步编程来说是必不可少的）。这反过来又导致了对手动同步（锁定、信号等）的更大依赖以及随之而来的问题。

线程的直接使用也会对性能产生影响，我们在第646页的“线程池”中对此进行了讨论。如果您需要运行数百或数千个并发I/O绑定操作，基于线程的方法纯粹在线程开销中会消耗数百或数千兆字节的内存。

Task类可以帮助解决所有这些问题。与线程相比，Task是更高级的抽象——它表示可能由线程支持也可能不由线程支持的并发操作。任务是组合的（您可以通过使用continuation将它们链接在一起）。他们可以使用线程池来减少启动延迟，通过TaskCompletionSource，他们可以使用回调方法，在等待I/O绑定操作时完全避免线程。

Task类型是在Framework 4.0中作为并行编程库的一部分引入的。然而，它们后来得到了增强（通过使用awaiters），以便在更通用的并发场景中发挥同样的作用，并且是C#异步函数的支持类型。

提示：

在本节中，我们忽略了专门针对并行编程的任务的特征；我们将在第22章中介绍它们。

#### 开启一个任务(Starting a Task)

启动由线程支持的Task的最简单方法是使用静态方法Task.Run（Task类位于System.Threading.Tasks命名空间中）。只需传入一个Action委托：

Task.Run (() => Console.WriteLine ("Foo"));

提示:

默认情况下，任务使用池线程，即后台线程。这意味着当主线程结束时，您创建的任何任务也会结束。因此，要从控制台应用程序运行这些示例，必须在启动任务后阻止主线程（例如，通过等待任务或调用console.ReadLine）：

Task.Run (() => Console.WriteLine ("Foo"));

Console.ReadLine

在本书的LINQPad配套示例中，Console.ReadLine被省略，因为LINQPad进程使后台线程保持活动状态。

调用Task.Run以这种与下所示的启动线程类似（除了我们稍后讨论的线程池含义）：

new Thread (() => Console.WriteLine ("Foo")).Start();

Task.Run返回一个Task对象，我们可以使用它来监视它的进度，就像Thread对象一样。（但是，请注意，我们在调用Task.Run之后没有调用Start，因为此方法会创建“热”任务；您可以使用Task的构造函数来创建“冷”任务，尽管在实践中很少这样做。）

您可以通过任务的status属性跟踪任务的执行状态。

#### Wait

在Task上调用Wait会阻塞直到任务完成，相当于在线程上调用Join：

Task task = Task.Run (() =>

{

Thread.Sleep (2000);

Console.WriteLine ("Foo");

});

Console.WriteLine (task.IsCompleted); // False

task.Wait(); // Blocks until task is complete

Wait允许您选择指定超时和取消令牌以提前结束等待（请参阅第681页的“取消”）

#### 长运行任务(Long-running tasks)

默认情况下，CLR在池线程上运行任务，这是短时间运行计算绑定工作的理想选择。对于长时间运行和阻塞操作（如前面的示例），您可以按照如下方式防止使用池线程：

Task task = Task.Factory.StartNew (() => ...,

TaskCreationOptions.LongRunning);

提示:

在池线程上运行一个长时间运行的任务不会造成麻烦；当并行运行多个长时间运行的任务（尤其是那些阻塞的任务）时，性能可能会受到影响。在这种情况下，通常有比TaskCreationOptions. LongRunning更好的解决方案：

* 如果任务是I/O绑定的，那么TaskCompletionSource和异步函数允许您使用回调（continuations）而不是线程来实现并发。
* 如果任务是计算绑定的，生产者/消费者队列允许您限制这些任务的并发性，避免其他线程和进程的饥饿（请参阅第970页的“编写生产者/消费者排队”）。

#### 返回值(Returning Values)

Task有一个名为Task＜TResult＞的泛型子类，它允许任务发出返回值。您可以通过调用Task来获得Task＜TResult＞。使用Func＜TResult＞委托（或兼容的lambda表达式）而不是Action运行：

Task<int> task = Task.Run (() => { Console.WriteLine ("Foo"); **return 3**; });

// ...

您可以稍后通过查询result属性来获取结果。如果任务尚未完成，访问此属性将阻止当前线程，直到任务完成：

int result = task.Result; // Blocks if not already finished

Console.WriteLine (result); // 3

在以下示例中，我们创建了一个任务，该任务使用LINQ来计算前三百万（+2）个整数中的素数数量：

Task<int> primeNumberTask = Task.Run (() =>

Enumerable.Range (2, 3000000).Count (n =>

Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));

Console.WriteLine ("Task running...");

Console.WriteLine ("The answer is " + primeNumberTask.Result);

输出“Task running…”，几秒钟后输出The answer is 216816。

提示:

Task<Tresult>可以被视为"未来来",因为它封装了稍后可用的结果。

#### Exceptions

与线程不同，任务可以方便地传播异常。因此，如果任务中的代码抛出未处理的异常（换言之，如果任务出错），则该异常将自动重新抛出给调用Wait（）或访问task＜TResult＞的Result属性的人：

// Start a Task that throws a NullReferenceException:

Task task = Task.Run (() => { throw null; });

try

{

task.Wait();

}

catch (AggregateException aex)

{

if (aex.InnerException is NullReferenceException)

Console.WriteLine ("Null!");

else

throw;

}

（CLR将异常封装在AggregateException中，以便更好地处理并行编程场景；我们将在第22章中对此进行讨论。）

您可以通过任务的IsFaulted和IsCanceled属性测试出现故障的任务，而无需重新引发异常。如果两个属性都返回false，则没有发生错误；如果IsCanceled为true，则为该任务引发OperationCanceledException（请参阅第941页的“取消”）；如果IsFaulted为true，则引发另一种类型的异常，exception属性将指示错误。

#### 异常和自主任务(Exceptions and autonomous tasks)

对于自主的“设置并忘记Set-and-forget”任务（即那些你没有通过Wait（）或Result进行约定的任务，或者执行相同操作的延续任务），很好的做法是显式异常处理任务代码，以避免静默失败，就像处理线程一样。

提示:

当异常仅表示无法获得您不再感兴趣的结果时，忽略异常是可以的。例如，如果用户取消了下载网页的请求，我们不会在乎该网页是否不存在。

当异常指示程序中存在错误时，忽略异常是有问题的，原因有两个：

* 该错误可能使您的程序处于无效状态。
* 由于该错误，以后可能会出现更多异常，并且未能记录初始错误可能会使诊断变得困难。

您可以通过静态事件TaskScheduler. UnobservedTaskException在全局级别订阅未观察到的异常；处理此事件并记录错误是很有意义的。

关于什么是未被观察到的，有几个有趣的细微差别：

* 如果故障发生在超时间隔之后，则等待超时的任务将生成未观察到的异常。
* 在任务出现故障后检查其Exception属性的行为会使异常"observed."

#### 任务延续(Continuations)

延续任务就是：“当你完成任务时，继续做其他事情。”延续任务通常由一个回调实现，该回调在操作完成后执行一次。有两种方法可以将延续附加到任务。第一个特别重要，因为它被C#的异步函数使用，您很快就会看到。我们可以用我们不久前在650页的“返回值”中写的素数计数任务来证明这一点：

Task<int> primeNumberTask = Task.Run (() =>

Enumerable.Range (2, 3000000).Count (n =>

Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));

var awaiter = primeNumberTask.GetAwaiter();

awaiter.OnCompleted (() =>

{

int result = awaiter.GetResult();

Console.WriteLine (result); // Writes result

});

对任务调用GetAwaiter返回一个awaiter对象，其OnCompleted方法告诉先行任务（primeNumberTask）在完成（或出错）时执行委托。将延续附加到已经完成的任务是有效的，在这种情况下，延续将被安排立即执行。

提示:

awaiter是公开我们刚刚看到的两个方法（OnCompleted和GetResult）和一个名为IsCompleted的布尔属性的任何对象。没有接口或基类以统一所有这些成员（尽管OnCompleted是接口INotifyCompletion的一部分）。我们在661页的“C#中的异步函数”中解释了该模式的重要性。

如果先行任务出错，则在延续代码调用awaiter.GetResult（）时会重新引发异常。我们可以简单地访问先行词的Result属性，而不是调用GetResult。调用GetResult的好处是，如果先行任务出错，则直接抛出异常，而不将其封装在AggregateException中，从而实现更简单、更干净的捕获块。

对于非泛型任务，GetResult（）有一个void返回值。它的有用功能仅用于重新引发异常。

如果存在同步上下文，OnCompleted会自动捕获它并将延续发布到该上下文。这在富客户端应用程序中非常有用，因为它会将延续返回到UI线程。然而，在编写库时，这通常是不可取的，因为相对昂贵的UI线程反弹应该在离开库时发生一次，而不是在方法调用之间发生。因此，您可以使用ConfigureAwait方法战胜它：

var awaiter = primeNumberTask.ConfigureAwait (false).GetAwaiter();

如果不存在同步上下文，或者使用ConfigureAwait（false），则延续将（通常）在池线程上执行。

附加延续的另一种方法是调用任务的ContinueWith方法：

primeNumberTask.ContinueWith (antecedent =>

{

int result = antecedent.Result;

Console.WriteLine (result); // Writes 123

});

ContinueWith本身返回一个Task，如果您想附加更多的continuation，这很有用。但是，如果任务出现故障，则必须直接处理AggregateException，并编排(marshal)额外的代码来封送UI应用程序中的延续（请参阅第962页的“任务调度器”）。在非UI上下文中，必须指定TaskContinuationOptions。如果希望在同一线程上执行延续，则执行同步；否则它将反弹到线程池。Continue With在并行编程场景中特别有用；我们将在第22章中详细介绍它。

#### TaskCompletionSource

我们已经看到Task.Run是如何做到的创建一个任务，在池线程（或非池线程）上运行委托。另一种创建任务的方法是使用TaskCompletionSource。TaskCompletionSource允许您从将来完成的任何操作中创建任务。它的工作原理是为您提供一个手动驱动的“从属”任务——通过指示操作何时完成或出现故障。这是I/O绑定工作的理想选择：您可以获得任务的所有好处（具有传播返回值、异常和延续的能力），而不会在操作期间阻塞线程。

public class TaskCompletionSource<TResult>

{

public void SetResult (TResult result);

public void SetException (Exception exception);

public void SetCanceled();

public bool TrySetResult (TResult result);

public bool TrySetException (Exception exception);

public bool TrySetCanceled();

public bool TrySetCanceled (CancellationToken cancellationToken);

...

}

调用这些方法中的任何一个都会发出任务的信号，使其处于已完成、有故障或已取消的状态（我们在第681页的“取消”一节中介绍了后者）。您应该只调用其中一个方法一次：如果再次调用，SetResult、SetException或SetCanceled将抛出异常，而Try\*方法返回false。

以下示例在等待5秒钟后打印42：

var tcs = new TaskCompletionSource<int>();

new Thread (() => { Thread.Sleep (5000); tcs.SetResult (42); })

{ IsBackground = true }

.Start();

Task<int> task = tcs.Task; // Our "slave" task.

Console.WriteLine (task.Result); // 42

使用TaskCompletionSource，我们可以编写自己的Run方法：

Task<TResult> Run<TResult> (Func<TResult> function)

{

var tcs = new TaskCompletionSource<TResult>();

new Thread (() =>

{

try { tcs.SetResult (function()); }

catch (Exception ex) { tcs.SetException (ex); }

}).Start();

return tcs.Task;

}

...

Task<int> task = Run (() => { Thread.Sleep (5000); return 42; });

调用此方法相当于以CreationOptions.LongRunning作参数数为调用Task.Factory.StartNew去请求非池化线程。

TaskCompletionSource的真正力量在于创建不占用线程的任务。例如，考虑一个等待5秒然后返回数字42的任务。我们可以通过使用Timer类在没有线程的情况下编写这篇文章，该类在CLR（以及操作系统）的帮助下，在x毫秒内触发一个事件（我们在第21章中重新讨论计时器）：

Task<int> GetAnswerToLife()

{

var tcs = new TaskCompletionSource<int>();

// Create a timer that fires once in 5000 ms:

var timer = new System.Timers.Timer (5000) { AutoReset = false };

timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult (42); };

timer.Start();

return tcs.Task;

}

因此，我们的方法返回一个任务，该任务在五秒钟后完成，结果为42。通过将延续附加到任务，我们可以在不阻塞任何线程的情况下写入其结果：

var awaiter = GetAnswerToLife().GetAwaiter();

awaiter.OnCompleted (() => Console.WriteLine (awaiter.GetResult()));

我们可以通过参数化延迟时间并去掉返回值，使其更有用，并将其转化为通用的延迟方法。这意味着让它返回一个Task，而不是Task＜int＞。然而，TaskCompletionSource没有非通用版本，这意味着我们不能直接创建非通用任务。解决方法很简单：因为Task＜TResult＞派生自Task，所以我们创建一个TaskCompletionSource＜anything＞，然后隐式地将它提供给您的Task＜anythiny＞转换为Task，如下所示：

var tcs = new TaskCompletionSource<object>();

Task task = tcs.Task;

现在我们可以编写我们的通用Delay方法：

Task Delay (int milliseconds)

{

var tcs = new TaskCompletionSource<object>();

var timer = new System.Timers.Timer (milliseconds) { AutoReset = false };

timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult (null); };

timer.Start();

return tcs.Task;

}

提示:

.NET5引入了一个非通用的TaskCompletionSource，因此如果您要确定目标。NET 5或更高版本，可以用TaskCompletionSource＜object＞代替TaskCompletationSource。

以下是我们如何在五秒钟后使用它来写“42”：

Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));

我们在没有线程的情况下使用TaskCompletionSource意味着只有在五秒钟后继续操作开始时，线程才会参与。我们可以通过一次启动10000个这样的操作来证明这一点，而不会出现错误或过度消耗资源：

for (int i = 0; i < 10000; i++)

Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));

提示:

定时器在池线程上启动回调，因此五秒钟后，线程池将收到10000个请求，以调用TaskCompletionSource上的SetResult（null）。如果请求到达的速度快于处理的速度，线程池将通过排队进行响应，然后以CPU的最佳并行度处理这些请求。如果线程绑定作业是短时间运行的，这是理想的，在这种情况下也是如此：线程绑定作业只是对SetResult的调用，加上将延续发布到同步上下文（在UI应用程序中）或延续本身（Console.WriteLine（42））的操作。

#### Task.Delay

我们刚刚编写的Delay方法非常有用，可以作为Task类上的静态方法使用：

**Task.Delay** (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));

或

Task.Delay (5000).ContinueWith (ant => Console.WriteLine (42));

Task.Delay是Thread.Sleep的异步等价物。

### 异步原理(Principles of Asynchrony)

在演示TaskCompletionSource时，我们最终编写了异步方法。在本节中，我们将准确定义什么是异步操作，并解释这是如何导致异步编程的。

#### 同步与异常对比

同步操作在返回到调用者之前完成其工作。

异步操作可以在返回到调用方后完成（大部分或全部）工作

您编写和调用的大多数方法都是同步的。List<T>.Add就是一个例子。或Console.WriteLine或Thread.Sleep异步方法不太常见，并且启动并发，因为工作与调用方并行进行。异步方法通常会快速（或立即）返回给调用者；因此，它们也被称为非阻塞方法。

到目前为止，我们看到的大多数异步方法都可以描述为通用方法：

* Thread.Start
* Task.Run
* 将延续附加到任务的方法

此外，我们在645页“同步上下文”中讨论的一些方法（Dispatcher.BeginInvoke、Control.BeginInvoke和SynchronizationContext.Post）是异步的，我们在653页“任务完成源”中写的方法也是异步的，包括延迟。

#### 什么是异步编程

异步编程的原理是异步编写长时间运行（或可能长时间运行）的函数。这与同步编写长时间运行的函数，然后从新线程或任务调用这些函数以根据需要引入并发性的传统方法形成了鲜明对比。

与异步方法的区别在于，并发是在长时间运行的函数内部启动的，而不是从函数外部启动的。这有两个好处：

* I/O绑定并发可以在不占用线程的情况下实现（正如我们在第653页的“TaskCompletionSource”中所展示的），从而提高可扩展性和效率。
* 富客户端应用程序最终会减少工作线程上的代码，从而简化线程安全性

这反过来又导致了异步编程的两种不同用途。首先是编写（通常是服务器端）能有效处理大量并发I/O的应用程序。这里的挑战不是线程安全（因为通常有最小的共享状态），而是线程效率；特别是不消耗每个网络请求的线程。因此，在这种情况下，只有I/O绑定操作才能从异步中受益。

第二个用途是简化富客户端应用程序中的线程安全性。随着程序规模的增长，这一点尤其重要，因为为了处理复杂性，我们通常会将较大的方法重构为较小的方法，从而产生相互调用的方法链（调用图）。

对于传统的同步调用图，如果图中的任何操作是长时间运行的，我们必须在工作线程上运行整个调用图，以维护响应UI。因此，我们最终会得到一个跨越许多方法的单个并发操作（粗粒度并发），这需要考虑图中每个方法的线程安全性。

对于异步调用图，在实际需要之前，我们不需要启动线程，通常是在图的低位（或者在I/O绑定操作的情况下根本不启动）。所有其他方法都可以完全在UI线程上运行，线程安全性大大简化。这导致了细粒度的并发性——一系列小的并发操作，在这些操作之间执行会跳转到UI线程。

提示:

为了从中受益，I/O和计算绑定操作都需要异步编写；一个好的经验法则是包括任何可能需要超过50ms的内容。

（另一方面，过于细粒度的异步可能会损害性能，因为异步操作会产生开销——请参阅第677页的“优化”。）

在本章中，我们主要关注富客户端场景，这是两者中比较复杂的一个。在第16章中，我们给出了两个示例来说明I/Obound场景（请参见第762页上的“与TCP的并发”和第755页的“编写HTTP服务器”）。

提示:

UWP框架鼓励异步编程，以至于某些长期运行方法的同步版本要么不公开，要么抛出异常。相反，您必须调用返回任务（或可以通过AsTask扩展方法转换为任务的对象）的异步方法。

#### 异步程序设计与延续(Asynchronous Programming and Continuations)

任务非常适合异步编程，因为它们支持连续性，而连续性对异步至关重要（请考虑我们在第653页的“TaskCompletionSource”中所写的Delay方法）。在编写Delay时，我们使用了TaskCompletionSource，这是实现“底层”I/O绑定异步方法的标准方法。

对于计算绑定方法，我们使用Task.Run以启动线程绑定并发。只需将任务返回给调用者，我们就可以创建一个异步方法。异步编程的区别在于，我们的目标是在调用图中做得更低，这样在富客户端应用程序中，更高层的方法可以保留在UI线程上，访问控制和共享状态，而不会出现线程安全问题。为了说明这一点，考虑以下使用所有可用核计算和计数素数的方法（我们在第22章中讨论ParallelEnumerable）:

int GetPrimesCount (int start, int count)

{

return

ParallelEnumerable.Range (start, count).Count (n =>

Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0));

}

如何运作的这些细并不重要；重要的是它可能需要一段时间才能运行。我们可以通过编写另一个方法来调用它：

void DisplayPrimeCounts()

{

for (int i = 0; i < 10; i++)

Console.WriteLine (GetPrimesCount (i\*1000000 + 2, 1000000) +

" primes between " + (i\*1000000) + " and " + ((i+1)\*1000000-1));

Console.WriteLine ("Done!");

}

下面是输出的结果:

78498 primes between 0 and 999999

70435 primes between 1000000 and 1999999

67883 primes between 2000000 and 2999999

66330 primes between 3000000 and 3999999

65367 primes between 4000000 and 4999999

64336 primes between 5000000 and 5999999

63799 primes between 6000000 and 6999999

63129 primes between 7000000 and 7999999

62712 primes between 8000000 and 8999999

62090 primes between 9000000 and 9999999

现在我们有一个调用图，DisplayPrimeCounts调用GetPrimeCount。前者使用Console.WriteLine是为了简单起见，尽管实际上它更可能是更新富客户端应用程序中的UI控件，正如我们稍后所演示的那样。我们可以为这个调用图启动粗粒度并发，如下所示：

Task.Run (() => DisplayPrimeCounts());

使用细粒度异步方法，我们首先编写异步版本的GetPrimesCount：

Task<int> GetPrimesCountAsync (int start, int count)

{

return Task.Run (() =>

ParallelEnumerable.Range (start, count).Count (n =>

Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0)));

}

#### 为什么语言集成支持很重要

现在，我们必须修改DisplayPrimeCounts，使其调用GetPrimeCountAsync。这就是C#的等待和异步关键字发挥作用的地方，因为在其他情况下这样做比听起来更棘手。如果我们简单地修改循环如下：

for (int i = 0; i < 10; i++)

{

var awaiter = GetPrimesCountAsync (i\*1000000 + 2, 1000000).GetAwaiter();

awaiter.OnCompleted (() =>

Console.WriteLine (awaiter.GetResult() + " primes between... "));

}

Console.WriteLine ("Done");

循环将快速旋转10次迭代（方法是非阻塞的），并且所有10次操作都将并行执行（然后是过早的“完成”）。

提示:

在这种情况下，并行执行这些任务是不可取的，因为它们的内部实现已经并行化；这只会让我们等待更长的时间来看到第一个结果（并破坏排序）。

然而，需要序列化任务的执行还有一个更常见的原因，那就是任务B取决于任务a的结果。例如，在获取网页时，DNS查找必须在HTTP请求之前。

为了让它们按顺序运行，我们必须从延续本身触发下一个循环迭代。这意味着消除for循环，并在延续中使用递归调用：

void DisplayPrimeCounts()

{

DisplayPrimeCountsFrom (0);

}

void DisplayPrimeCountsFrom (int i)

{

var awaiter = GetPrimesCountAsync (i\*1000000 + 2, 1000000).GetAwaiter();

awaiter.OnCompleted (() =>

{

Console.WriteLine (awaiter.GetResult() + " primes between...");

if (++i < 10) DisplayPrimeCountsFrom (i);

else Console.WriteLine ("Done");

});

}

如果我们想使DisplayPrimesCount本身异步，返回它在完成时发出信号的任务，情况会变得更糟。要做到这一点，需要创建一个TaskCompletionSource：

Task DisplayPrimeCountsAsync()

{

var machine = new PrimesStateMachine();

machine.DisplayPrimeCountsFrom (0);

return machine.Task;

}

class PrimesStateMachine

{

TaskCompletionSource<object> \_tcs = new TaskCompletionSource<object>();

public Task Task { get { return \_tcs.Task; } }

public void DisplayPrimeCountsFrom (int i)

{

var awaiter = GetPrimesCountAsync (i\*1000000+2, 1000000).GetAwaiter();

awaiter.OnCompleted (() =>

{

Console.WriteLine (awaiter.GetResult());

if (++i < 10) DisplayPrimeCountsFrom (i);

else { Console.WriteLine ("Done"); \_tcs.SetResult (null); }

});

}

}

幸运的是，C#的异步函数为我们完成了所有这些工作。使用async和await关键字，我们只需要这样写：

async Task DisplayPrimeCountsAsync()

{

for (int i = 0; i < 10; i++)

Console.WriteLine (await GetPrimesCountAsync (i\*1000000 + 2, 1000000) +

" primes between " + (i\*1000000) + " and " + ((i+1)\*1000000-1));

Console.WriteLine ("Done!");

}

因此，异步和等待对于在没有过多复杂性的情况下实现异步至关重要。现在让我们看看这些关键字是如何工作的。

提示:

另一种看待这个问题的方法是，命令式循环构造（例如foreach等）不能很好地与continuation混合，因为它们依赖于方法的当前本地状态（“这个循环还要运行多少次？”）

尽管async和await关键字提供了一种解决方案，但有时也可以用另一种方式来解决，方法是用等效的函数（换句话说，LINQ查询）替换命令式循环结构。这是反应扩展（Rx）的基础，当您想对结果执行查询运算符或组合多个序列时，这是一个很好的选择。要付出的代价是，为了防止阻塞，Rx对基于推送的序列进行操作，这在概念上可能很棘手。

### C#中的异常函数

async和await关键字使您能够编写与同步代码具有相同结构和简单性的异步代码，同时消除异步编程的“管道”。

#### Awaiting

await关键字简化了continuation的附加。从一个基本场景开始，编译器扩展如下：

var result = **await** expression;

statement(s);

在功能上类似于此：

var awaiter = expression.GetAwaiter();

awaiter.OnCompleted (() =>

{

var result = awaiter.GetResult();

statement(s);

});

编译器还发出代码，在同步完成的情况下缩短延续（请参阅第677页的“优化”），并处理我们在后面章节中了解的各种细微差别。

为了演示，让我们重新审视我们之前编写的计算和计数素数的异步方法：

Task<int> GetPrimesCountAsync (int start, int count)

{

return Task.Run (() =>

ParallelEnumerable.Range (start, count).Count (n =>

Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));

}

使用await关键字，我们可以如下调用它：

int result = **await** GetPrimesCountAsync (2, 1000000);

Console.WriteLine (result);

要进行编译，我们需要将async修饰符添加到包含方法中：

async void DisplayPrimesCount()

{

int result = await GetPrimesCountAsync (2, 1000000);

Console.WriteLine (result);

}

async修饰符指示编译器在方法中出现歧义时将wait视为关键字而非标识符（这确保了在C#5之前编写的可能使用wait作为标识符的代码在编译时仍然不会出错）。async修饰符只能应用于返回void或（稍后将看到）Task或Task＜TResult＞的方法（和lambda表达式）。

提示:

async修饰符与unsafe修饰符的相似之处在于，它对方法的签名或公共元数据没有影响；它只影响方法内部发生的事情。因此，在接口中使用异步是没有意义的。然而，例如，在重写非异步虚拟方法时引入异步是合法的，只要签名保持不变。

带有async修饰符的方法被称为异步函数，因为它们本身通常是异步的。为了了解原因，让我们看看执行是如何通过异步函数进行的。

在遇到await表达式时，执行（通常）返回给调用方——就像迭代器中的yield返回一样。但在返回之前，运行时会将一个continuation附加到等待中的任务，以确保任务完成时，执行跳回到方法并继续执行它停止的地方。如果任务出错，则会重新抛出其异常，否则会将其返回值分配给等待表达式。我们可以通过查看刚才检查的异步方法的逻辑扩展来总结我们刚才所说的一切：

void DisplayPrimesCount()

{

var awaiter = GetPrimesCountAsync (2, 1000000).GetAwaiter();

awaiter.OnCompleted (() =>

{

int result = awaiter.GetResult();

Console.WriteLine (result);

});

}

您等待的表达式通常是一项任务；但是，任何具有返回awaiter的GetAwaiter方法的对象（实现INotifyCompletion.OnCompleted并具有适当类型的GetResult方法和bool IsCompleted属性）都将满足编译器的要求。

请注意，我们的await表达式的计算结果为int类型；这是因为我们等待的表达式是Task<int>（其中GetAwaiter().GetResult()方法返回一个int）。

等待非泛型任务是合法的，并会生成一个void表达式：

await Task.Delay (5000);

Console.WriteLine ("Five seconds passed!");

#### 捕获本地状态(Capturing local state)

await表达式的真正威力在于，它们几乎可以出现在代码中的任何位置。具体地说，await表达式可以代替任何表达式（在异步函数中）出现的地方，锁定语句或不安全上下文中除外。

下面的示例, 我们在loop循环中使用await.

async void DisplayPrimeCounts()

{

for (int i = 0; i < 10; i++)

Console.WriteLine (await GetPrimesCountAsync (i\*1000000+2, 1000000));

}

在第一次执行GetPrimesCountAsync时，执行将通过等待表达式返回给调用方。当方法完成（或出现故障）时，将从停止的地方恢复执行，并保留局部变量和循环计数器的值。

如果没有await关键字，最简单的等价词可能是我们在第659页“为什么语言支持很重要”中写的例子。然而，编译器采用了更通用的策略，将这些方法重构到状态机中（就像使用迭代器一样）。

编译器依靠延续（通过awaiter模式）在等待表达式之后恢复执行。这意味着，如果在富客户端应用程序的UI线程上运行，同步上下文将确保在相同的线程。否则，无论任务在哪个线程上完成，都会继续执行。线程的更改不会影响执行顺序，也不会产生什么影响，除非您以某种方式依赖线程相关性，可能是通过使用线程本地存储（请参阅第923页的“线程本地存储”）。这就像游览一个城市，叫出租车从一个目的地到另一个目的。有了同步上下文，您将总是得到相同的出租车；如果没有同步上下文，您通常每次都会得到不同的出租车。然而，无论哪种情况，旅程都是一样的。

#### Awaiting in a UI

我们可以在更实用的上下文中演示异步函数，方法是编写一个简单的UI，在调用计算绑定方法时保持响应。让我们从同步解决方案开始：

class TestUI : Window

{

Button \_button = new Button { Content = "Go" };

TextBlock \_results = new TextBlock();

public TestUI()

{

var panel = new StackPanel();

panel.Children.Add (\_button);

panel.Children.Add (\_results);

Content = panel;

\_button.Click += (sender, args) => Go();

}

void Go()

{

for (int i = 1; i < 5; i++)

\_results.Text += GetPrimesCount (i \* 1000000, 1000000) +

" primes between " + (i\*1000000) + " and " + ((i+1)\*1000000-1) +

Environment.NewLine;

}

int GetPrimesCount (int start, int count)

{

return ParallelEnumerable.Range (start, count).Count (n =>

Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0));

}

}

按下“Go”按钮后，应用程序在执行计算绑定代码所需的时间内变得没有响应。异步化有两个步骤；首先是切换到我们在前面的示例中使用的异步版本的GetPrimesCount：

**Task<**int**>** GetPrimesCount**Async** (int start, int count)

{

return **Task.Run (() =>**

ParallelEnumerable.Range (start, count).Count (n =>

Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0))**)**;

}

第二步是修改Go以调用GetPrimesCountAsync

**async** void Go()

{

**\_button.IsEnabled = false;**

for (int i = 1; i < 5; i++)

\_results.Text += **await** GetPrimesCount**Async** (i \* 1000000, 1000000) +

" primes between " + (i\*1000000) + " and " + ((i+1)\*1000000-1) +

Environment.NewLine;

**\_button.IsEnabled = true;**

}

这说明了使用异步函数编程的简单性：您可以像同步编程一样进行编程，但调用异步函数而不是阻塞函数并等待它们。只有GetPrimesCountAsync中的代码在工作线程上运行；Go中的代码在UI线程上“租用”时间。我们可以说Go与消息循环伪并行执行（因为它的执行与UI线程处理的其他事件穿插在一起）。有了这种伪并发性，唯一可以发生抢占的点是在await时。这简化了线程安全：在我们的情况下，这可能导致的唯一问题是可重入性（在按钮运行时再次单击按钮，我们通过禁用按钮来防止这种情况）。真正的并发发生在调用堆栈的较低位置，内部的代码被Task.Run调用。使用这个模型的关键是:真正并发的代码不应该访问共享状态或UI控件。

举另一个例子，假设我们不想计算素数，而是想下载几个网页并对它们的长度求和。NET公开了许多任务返回异步方法，其中之一就是System.Net中的WebClient类。DownloadDataTaskAsync是一个把URI下载到字节数组的异步方法，返回一个Task＜byte[]＞，因此通过等待它，我们得到一个byte[]。现在让我们重写Go方法：

**async** void Go()

{

\_button.IsEnabled = false;

string[] urls = "www.albahari.com www.oreilly.com www.linqpad.net".Split();

int totalLength = 0;

try

{

foreach (string url in urls)

{

var uri = new Uri ("http://" + url);

byte[] data = **await new WebClient().DownloadDataTaskAsync (uri);**

\_results.Text += "Length of " + url + " is " + data.Length +

Environment.NewLine;

totalLength += data.Length;

}

\_results.Text += "Total length: " + totalLength;

}

catch (WebException ex)

{

\_results.Text += "Error: " + ex.Message;

}

finally { \_button.IsEnabled = true; }

}

同样，这反映了我们如何同步编写它——包括使用catch和finally块。即使在第一次等待之后执行返回给调用者，finally块也不会执行，直到方法在逻辑上完成（由于其所有代码都在执行——或者早期返回或未处理的异常）

仔细想想下面到底发生了什么可能会很有帮助。首先，我们需要重新访问在UI线程上运行消息循环的伪代码：

Set synchronization context for this thread to WPF sync context

while (!thisApplication.Ended)

{

wait for something to appear in message queue

Got something: what kind of message is it?

Keyboard/mouse message -> fire an event handler

User **BeginInvoke/Invoke** message -> execute delegate

}

我们附加到UI元素的事件处理程序通过此消息循环执行。当我们的Go方法运行时，执行一直进行到await表达式，然后返回到消息循环（释放UI以响应进一步的事件）。然而，编译器对wait的扩展确保了在返回之前，设置一个延续，以便在任务完成后从停止的地方恢复执行。因为我们在UI线程上等待，continuation发布到同步上下文，同步上下文通过消息循环执行它，使我们的整个Go方法在UI线程中伪并行执行。真正的并发是在DownloadDataTaskAsync的实现中（I/O绑定）的。

#### 与精细度并发比较

在C#5之前，异步编程很困难，不仅因为没有语言支持，还因为。NET框架通过称为EAP和APM的笨拙模式（请参阅第689页的“过时模式”）而不是任务返回方法来暴露异步功能。

流行的解决方法是粗粒度并发（事实上，甚至有一种叫做BackgroundWorker的类型可以帮助实现这一点）。回到我们使用GetPrimesCount的原始同步示例，我们可以通过修改按钮的事件处理程序来演示粗粒度异步，如下所示：

...

\_button.Click += (sender, args) =>

{

\_button.IsEnabled = false;

Task.Run (() => Go());

};

（我们选择使用Task.Run而不是BackgroundWorker，因为后者对简化我们的特定示例没有任何作用。）无论哪种情况，最终结果都是我们的整个同步调用图（Go加GetPrimesCount）在工作线程上运行。由于Go更新了UI元素，我们现在必须用Dispatcher.BeginInvoke来调用代码：

void Go()

{

for (int i = 1; i < 5; i++)

{

int result = GetPrimesCount (i \* 1000000, 1000000);

**Dispatcher.BeginInvoke (new Action (() =>**

\_results.Text += result + " primes between " + (i\*1000000) +

" and " + ((i+1)\*1000000-1) + Environment.NewLine));

}

**Dispatcher.BeginInvoke (new Action (() =>** \_button.IsEnabled = true**));**

}

与异步版本不同，循环本身在工作线程上运行。这可能看起来无害，然而，即使在这种简单的情况下，我们对多线程的使用也引入了竞争条件。（你能发现它吗？如果不能，试着运行程序：它几乎肯定会变得明显。）

实现取消和进度报告为线程安全错误创造了更多的可能性，方法中的任何其他代码也是如此。例如，假设循环的上限不是硬编码的，而是来自方法调用：

for (int i = 1; i < GetUpperBound(); i++)

现在假设从GetUpperBound（）延迟加载的配置文件中读取值，该配置文件在第一次调用时从磁盘加载。所有这些代码现在都在您的工作线程上运行，这些代码很可能不是线程安全的。这时危险在调用图的在高层启动工作线程。

### 写异步函数

对于任何异步函数，您都可以将void返回类型替换为Task，以使方法本身成为有用的异步（和awaitable）。无需进一步更改：

async **Task** PrintAnswerToLife() // We can return Task instead of void

{

await Task.Delay (5000);

int answer = 21 \* 2;

Console.WriteLine (answer);

}

请注意，我们没有在方法体中显式返回任务。编译器生成任务，并在方法完成时（或在未处理的异常时）发出信号。这使得创建异步调用链变得容易：

async Task Go()

{

await PrintAnswerToLife();

Console.WriteLine ("Done");

}

因为我们已经声明了Go返回Task类型，所以它是可等待的(awaitable)。

编译器将返回任务的异步函数扩展为使用TaskCompletionSource创建任务的代码，然后发出信号或出错

撇开优雅不谈，我们可以将PrintAnswerToLife扩展为以下等效功能：

Task PrintAnswerToLife()

{

**var tcs = new TaskCompletionSource<object>();**

var awaiter = Task.Delay (5000).GetAwaiter();

awaiter.OnCompleted (() =>

{

try

{

awaiter.GetResult(); // Re-throw any exceptions

int answer = 21 \* 2;

Console.WriteLine (answer);

**tcs.SetResult (null);**

}

catch (Exception ex) { **tcs.SetException (ex);** }

});

**return tcs.Task;**

}

因此, 每当任务返回异步方法完成时，执行就会跳回到等待它的任何东西（通过延续）。

提示:

在富客户端场景中，执行到这一点上然后弹回到UI线程（如果它还没有在UI线程上）。否则，它会在继续返回的任何线程上继续。这意味着，除了UI线程启动的第一次“反弹”之外，异步调用图的冒泡没有延迟成本。

#### 返回Task<TResult>

如果方法体返回TResult，则可以返回Task<TResult>：

async **Task<int>** GetAnswerToLife()

{

await Task.Delay (5000);

int answer = 21 \* 2;

return **answer**; // Method has return type Task<**int**> we return **int**

}

在内部，这会导致TaskCompletionSource被用一个值而不是null发出信号。我们可以通过PrintAns werToLife调用GetAnswerToLife（顺序是从Go调用PrintAnswerToLift）来演示它：

async Task Go()

{

await PrintAnswerToLife();

Console.WriteLine ("Done");

}

async Task PrintAnswerToLife()

{

int answer = await GetAnswerToLife();

Console.WriteLine (answer);

}

async Task<int> GetAnswerToLife()

{

await Task.Delay (5000);

int answer = 21 \* 2;

return answer;

}

实际上，我们已经将原来的PrintAnswerToLife重构为两个方法——就像同步编程一样简单。与有意的同步编程的相似性；我们调用图的同步等价是: 调用Go()后在阻塞五秒钟后,会得到相同的结果：

void Go()

{

PrintAnswerToLife();

Console.WriteLine ("Done");

}

void PrintAnswerToLife()

{

int answer = GetAnswerToLife();

Console.WriteLine (answer);

}

int GetAnswerToLife()

{

Thread.Sleep (5000);

int answer = 21 \* 2;

return answer;

}

提示:

这也说明了如何在C#中使用异步函数进行设计的基本原理：

1. 正常的写同步方法.
2. 用异步方法替换同步方法调用. 并且用await等待这些异步方法
3. 除了“顶级”方法（通常是UI控件的事件处理程序）之外，请将异步方法的返回类型升级为Task或Task＜TResult＞，以便它们可等待(awaitable)。

编译器有能力制造异步函数任务意味着，在大多数情况下，您只需要在启动I/O绑定并发的底层方法中（相对罕见的情况下）显式实例化TaskCompletionSource。（对于启动计算绑定并发的方法，可以使用task.Run创建任务。）

#### 异步调用执行图

为了确切地了解它是如何执行的，按如下方式重新排列我们的代码是很有帮助的：

async Task Go()

{

**var task = PrintAnswerToLife();**

await task; Console.WriteLine ("Done");

}

async Task PrintAnswerToLife()

{

**var task = GetAnswerToLife();**

int answer = await task; Console.WriteLine (answer);

}

async Task<int> GetAnswerToLife()

{

**var task = Task.Delay (5000);**

await task; int answer = 21 \* 2; return answer;

}

简约的叙述同步情况: Go调用PrintAnswerToLife，后者调用GetAnswerToLife，后者呼叫Delay，然后等待。等待导致执行返回到PrintAnswerToLife，它本身正在等待，返回到Go，Go也在等待并返回给调用者。所有这些都是在调用Go的线程上同步发生的；

五秒钟后，Delay上的continuation启动，执行返回到池线程上的Get-AnswerToLife。（如果我们从一个UI线程开始，执行现在会跳到该线程。）然后运行GetAnswerToLife中的其余语句，之后方法的Task＜int＞以42的结果完成，并在PrintAnswerToLife中执行continuation，后者执行该方法中的剩余语句。这个过程一直持续到Go的任务被指示完成为止。

执行流与我们前面展示的同步调用图匹配，因为我们遵循的模式是，在调用每个异步方法后立即等待它。这会创建一个顺序流，在调用图中没有并行或重叠执行。每个等待表达式都会在执行中创建一个“间隙”，之后程序会从停止的地方恢复。

#### 并行(Parallelism)

在不等待的情况下调用异步方法允许后面的代码并行执行。您可能已经在前面的示例中注意到，我们有一个按钮，其事件处理程序名为Go，如下所示：\_button.Click += (sender, args) => Go();

尽管Go是一个异步方法，但我们没有等待它，这确实促进了维护响应UI所需的并发性。

我们可以使用相同的原理并行运行两个异步操作：

var task1 = PrintAnswerToLife();

var task2 = PrintAnswerToLife();

await task1; await task2;

（通过等待这两个操作，我们在这一点上“结束”并行性。稍后，我们将描述WhenAll任务组合子如何帮助实现这种模式。）

无论操作是否在UI线程上启动，以这种方式创建的并发都会发生，尽管发生方式不同。在这两种情况下，我们都会在启动它的”下层bottom-level”操作（如Task.Delay或转移到Task.Run的代码）中获得相同的“真实”并发性。只有在没有同步上下文的情况下启动操作时，调用堆栈中高于此级别的方法才会受到真正的并发性的约束；否则，它们将受到我们前面讨论的伪并发性（和简化的线程安全性）的约束，因此，我们可以被抢占的唯一位置是等待语句。例如，这使我们可以定义一个共享字段\_x，并在GetAnswerToLife中递增它而不锁定：

async Task<int> GetAnswerToLife()

{

**\_x++;**

await Task.Delay (5000);

return 21 \* 2;

}

（不过，我们无法假设\_x在等待之前和之后具有相同的值。）

#### Asynchronous Lambda Expressions

正如普通的命名方法可以是异步的一样：

**async Task** NamedMethod()

{

await Task.Delay (1000);

Console.WriteLine ("Foo");

}

未命名的方法（lambda表达式和匿名方法）也是如此，如果前面有async关键字：

Func<**Task**> unnamed = **async** () =>

{

await Task.Delay (1000);

Console.WriteLine ("Foo");

};

我们可以用同样的方式调用并等待：

await NamedMethod();

await unnamed();

在附加事件处理程序时，我们可以使用异步lambda表达式：

myButton.Click += **async** (sender, args) =>

{

await Task.Delay (1000);

myButton.Content = "Done";

};

这比具有相同效果的以下内容更简洁：

myButton.Click += ButtonHandler;

...

**async** void ButtonHandler (object sender, EventArgs args)

{

await Task.Delay (1000);

myButton.Content = "Done";

};

异步lambda表达式也可以返回Task＜TResult＞：

Func<**Task<int>**> unnamed = async () =>

{

await Task.Delay (1000);

return 123;

};

int answer = await unnamed();

#### 异步流(Asynchronous Streams)

使用yield return，您可以编写一个迭代器；使用await，您可以编写一个异步函数。异步流（来自C#8）结合了这些概念，并允许您编写一个等待的迭代器，异步生成元素。这种支持建立在以下一对接口的基础上，这些接口与我们在第203页的“枚举和迭代程序”中描述的枚举接口异步对应：

public interface IAsyncEnumerable<out T>

{

IAsyncEnumerator<T> GetAsyncEnumerator (...);

}

public interface IAsyncEnumerator<out T>: IAsyncDisposable

{

T Current { get; }

ValueTask<bool> MoveNextAsync();

}

ValueTask<T>是一个包装Task<T>的结构，在行为上与Task<T>相似，同时在任务同步完成时实现更高效的执行（在枚举序列时经常发生这种情况）。有关差异的讨论，请参见第679页的“ValueTask<T>”。IAsyncDisposable是IDisposable的异步版本；如果您选择手动实现接口，它提供了执行清理的机会：

public interface IAsyncDisposable

{

ValueTask DisposeAsync();

}

提示:

从序列中提取每个元素的行为（Move-NextAsync）是一种异步操作，因此当元素以逐段方式到达时（例如处理视频流中的数据时），异步流是合适的。相反，当序列作为一个整体被延迟时，以下类型更适合，但当元素到达时，它们会一起到达：

Task<IEnumerable<T>>

要生成异步流，您需要编写一个结合迭代器和异步方法原理的方法。换句话说，您的方法应该同时包括yield return和await，并且它应该返回IasyncEnumerable<T>：

async IAsyncEnumerable<int> RangeAsync (

int start, int count, int delay)

{

for (int i = start; i < start + count; i++)

{

await Task.Delay (delay);

yield return i;

}

}

要使用异步流，请使用await foreach语句：

**await foreach** (var number in RangeAsync (0, 10, 500))

Console.WriteLine (number);

请注意，数据稳定地到达，每500毫秒（或者，在现实生活中，当它变得可用时）。与之相比，使用Task<Ienumerable<T>>的类似构造在最后一段数据可用之前不会返回任何数据：

static async Task<IEnumerable<int>> RangeTaskAsync (int start, int count,int delay)

{

List<int> data = new List<int>();

for (int i = start; i < start + count; i++)

{

await Task.Delay (delay);

data.Add (i);

}

return data;

}

下是怎么用foreach语句使用它:

foreach (var data in await RangeTaskAsync(0, 10, 500))

Console.WriteLine (data);

#### 查询(Querying)IAsyncEnumerable<T>

System.Linq.Async Nuget包定义了使用IasyncEnumberable<T> 的LINQ查询方法, 让你就如同使用IEnumerable<T>一样写查询.

例如，我们可以像在上一节中定义的那样在RangeAsync方法上编写一个LINQ查询, 如下:

IAsyncEnumerable<int> query =

from i in RangeAsync (0, 10, 500)

where i % 2 == 0 // Even numbers only.

select i \* 10; // Multiply by 10.

await foreach (var number in query)

Console.WriteLine (number);

这将输出0,20,40,…

提示:

如果您熟悉Rx，您也可以通过调用ToObservable扩展方法从其（更强大的）查询运算符中受益，该方法将IAsyncEnumerable<T>转换为IOobservable<T>。一个ToAsyncEnumerable扩展方法也可用于反向转换。

#### 在Asp.Net Core中的IasyncEnumerable<T>

ASP.Net Core控制器操作现在可以返回IAsyncEnumerable＜T＞。此类方法必须标记为async。例如

[HttpGet]

public async IAsyncEnumerable<string> Get()

{

using var dbContext = new BookContext();

await foreach (var title in dbContext.Books

.Select(b => b.Title)

.AsAsyncEnumerable())

yield return title;

}

#### 在WinRT中使用异步方法

如果您正在开发UWP应用程序，则需要使用操作系统中定义的WinRT类型。WinRT的等效Task是IAsyncAction，等效Task<Tresult>是IasyncOperation<Tresult>。对于报告进度的操作，等效的是IasyncActionWithProgress<Tprogress>和IasyncOperationWithProgress<Tresult,Tprogress>。它们都是在Windows.Foundation中定义的。

您可以通过AsTask扩展方法将其转换为Task或Task<Tresult>

Task<StorageFile> fileTask = KnownFolders.DocumentsLibrary.CreateFileAsync("test.txt")

.**AsTask()**;

或你能直接await它们:

StorageFile file = **await** KnownFolders.DocumentsLibrary

.CreateFileAsync("test.txt");

提示:

由于COM类型系统的限制，IAsyncAction WithProgress<Tprogress>和IasyncOperationWithProgress<Tresult,Tprogress>并不像您所期望的那样基于IAsyncAction。相反，两者都继承自一个共同被称为IAsyncInfo的基本类型。

AsTask方法也被重载以接受取消令牌（请参阅第681页的“取消”）。当链接到WithProgress变体时，它也可以接受IProgress<T>对象（请参见第683页的“进度报告”）。

#### 异步和同步上下文

我们已经看到了同步上下文的存在在发布连续性方面是多么重要。还有其他一些更微妙的方式可以让这种同步上下文与返回void的异步函数一起发挥作用。这些不是C#编译器扩展的直接结果，而是System .CompilerServices中的Async\*MethodBuilder类型并被编译器使用的扩展异步函数。

##### 异常发送(Exception posting)

富客户端应用程序中的常见做法是依靠中心异常处理事件（WPF中的Application.DispatcherUnhandledException）来处理UI线程上引发的未处理异常。并且在ASP。NET核心应用程序，Startup.cs的ConfigureServices方法中的自定义ExceptionFilterAttribute也可以执行类似的工作。在内部，它们通过在自己的try/catch块中调用UI事件（或在ASP.NET Core中，页面处理方法的管道）来工作。

顶层异步函数使这一问题复杂化。请考虑以下按钮单击的事件处理程序：

**async** void ButtonClick (object sender, RoutedEventArgs args)

{

await Task.Delay(1000);

throw new Exception ("Will this be ignored?");

}

当单击按钮并运行事件处理程序时，执行将正常返回到await语句之后的消息循环，并且一秒钟后抛出的异常无法被消息循环中的catch块捕获。

为了缓解这个问题，AsyncVoidMethodBuilder捕获未处理的异常（在返回异步函数的void中），并将它们发布到同步上下文（如果存在），以确保全局异常处理事件仍然激发。

提示:

编译器仅将此逻辑应用于void返回的异步函数。因此，如果我们更改ButtonClick以返回Task而不是void，则未处理的异常将导致Task出错，从而导致Task无处可去（导致未观察到的异常）。

一个有趣的细微差别是，无论你是在等待之前还是之后抛出异常，都没有区别。因此，在以下示例中，异常被发布到同步上下文（如果存在），而从不发布到调用方：

async void Foo() { **throw null**; await Task.Delay(1000); }

（如果不存在同步上下文，则异常将在冒泡到线程池上传播，这将终止应用程序。）

不直接将异常抛出给调用者的原因是为了确保可预测性和一致性。在以下示例中，InvalidOperationException将始终具有与导致Task出错相同的效果，而与someCondition无关：

async Task Foo()

{

if (*someCondition*) await Task.Delay (100);

throw new InvalidOperationException();

}

迭代器的工作方式类似：

IEnumerable<int> Foo() { **throw null**; yield return 123; }

在本例中，从不直接将异常抛出给调用方：只有枚举了序列，才会抛出异常。

##### 操作开如和完成

如果存在同步上下文，void返回的异步函数在进入函数时也会调用其OperationStarted方法，在函数完成时调用其OperationCompleted方法。

如果为单元测试编写自定义同步上下文以使返回的异步方法无效，那么重写这些方法非常有用。这在微软的并行编程博客上有讨论。

#### 优化

##### Completing synchronously

异步函数可以在等待之前返回。考虑以下缓存网页下载的方法：

static Dictionary<string,string> \_cache = new Dictionary<string,string>();

async Task<string> GetWebPageAsync (string uri)

{

string html;

if (\_cache.TryGetValue (uri, out html)) **return html**;

return \_cache [uri] =

**await** new WebClient().DownloadStringTaskAsync (uri);

}

如果缓存中已经存在URI，则执行返回给调用方，而没有发生等待，并且该方法返回一个已发出信号的任务。这被称为同步完成。

当您等待一个同步完成的任务时，执行不会返回给调用者，而是通过一个延续返回；相反，它会立即进入下一个语句。编译器通过检查awaiter上的IsCompleted属性来实现此优化；换句话说，无论你如何等待

Console.WriteLine (await GetWebPageAsync ("http://oreilly.com"));

编译器发出代码以在同步完成的情况下使延续短路：

var awaiter = GetWebPageAsync().GetAwaiter();

**if (awaiter.IsCompleted)**

**Console.WriteLine (awaiter.GetResult());**

else

awaiter.OnCompleted (() => Console.WriteLine (awaiter.GetResult());

提示:

等待同步返回的异步函数仍然会产生（非常）小的开销——在2019年的PC上可能是20纳秒。

相比之下，跳转到线程池会带来上下文切换的成本——可能是一到两微秒——以及跳转到UI消息循环的成本，至少是这个成本的10倍（如果UI线程繁忙，则会更长）。

编写从不等待的异步方法甚至是合法的，尽管编译器会生成警告：

async Task<string> Foo() { return "abc"; }

如果您的实现不需要异步，那么在重写虚拟/抽象方法时，这样的方法可能很有用。（例如MemoryStream的ReadAsync/WriteAsync方法；请参阅第15章。）实现相同结果的另一种方法是使用Task.FromResult，返回一个已发出信号的任务：

Task<string> Foo() { return **Task.FromResult ("abc")**; }

如果从UI线程调用，我们的GetWebPageAsync方法是隐式线程安全的，因为您可以连续多次调用它（从而启动多个并发下载），并且不需要锁定来保护缓存。然而，如果这一系列调用是对同一个URI的，我们最终会启动多个冗余下载，所有这些下载最终都会更新同一个缓存条目（最后一个成功）。虽然没有错误，但如果对同一URI的后续调用可以（异步）等待正在进行的请求的结果，则效率会更高。

有一种简单的方法可以实现这一点——不需要使用锁或信号结构。我们创建的不是字符串缓存，而是“futures”缓存（Task＜string＞）：

static Dictionary<string,Task<string>> \_cache =

new Dictionary<string,Task<string>>();

Task<string> GetWebPageAsync (string uri)

{

if (\_cache.TryGetValue (uri, out var downloadTask))

return downloadTask;

return \_cache [uri] = new WebClient().DownloadStringTaskAsync (uri);

}

（请注意，我们没有将该方法标记为async，因为我们直接返回从调用WebClient的方法中获得的任务。）

如果我们使用相同的URI重复调用GetWebPageAsync，现在就可以保证返回相同的Task＜string＞对象。（这还有一个额外的好处，那就是最大限度地减少垃圾收集负载。）由于我们刚刚讨论的编译器优化，如果任务完成，等待它是很便宜的。

我们可以进一步扩展我们的示例，通过锁定整个方法体，使其在没有同步上下文保护的情况下实现线程安全：

lock (\_cache)

if (\_cache.TryGetValue (uri, out var downloadTask))

return downloadTask;

else

return \_cache [uri] = new WebClient().DownloadStringTaskAsync (uri);

}

这之所以有效，是因为我们在下载页面的过程中没有锁定（这会损害并发性）；我们在检查缓存的短时间内进行锁定，必要时启动一个新任务，并使用该任务更新缓存。

##### ValueTask<T>

提示:

ValueTask<T>适用于微优化场景，您可能永远不需要编写返回此类型的方法。然而，注意我们在下一节中概述的预防措施仍然是值得的，因为有些预防措施的.NET方法返回ValueTask<T>，AsyncEnumerable<T>也使用了它。

我们刚刚描述了编译器如何在一个快速完成的任务上优化等待表达式——通过缩短延续并立即进行到下一个语句。如果同步完成是由于缓存，我们看到缓存任务本身可以提供一个优雅高效的解决方案。

然而，在所有同步完成场景中缓存任务是不现实的。有时，必须实例化一个新任务，这会产生（微小的）潜在的低效率。这是因为Task和Task<T>是引用类型，因此实例化需要基于堆的内存分配和后续集合。优化的一种极端形式是编写无分配的代码；换句话说，这不会实例化任何引用类型，不会给垃圾收集增加负担。为了支持这种模式，引入了ValueTask和ValueTask<T>结构，编译器允许它们代替Task和Task<T>：

async **ValueTask**<int> Foo() { ... }

如果操作同步完成，则等待ValueTask＜T＞是无分配的：

int answer = await Foo(); // (Potentially) allocation-free

如果操作没有同步完成，ValueTask<t>会在后台创建一个普通的Task<T>（它将等待转发到该任务），但不会获得任何结果。

您可以通过调用AsTask将ValueTask＜T＞转换为普通Task＜T

方法

还有一个非泛型版本——ValueTask——类似于Task。

##### 使用ValueTask<T>注意事项

ValueTask<T>相对不常见，因为它被定义为纯粹出于性能原因的结构。这意味着它被不适当的值类型语义所困扰，这可能会导致意外。为了避免不正确的行为，您必须避免以下行为：

* Await同一个ValueTask<T>多次
* 当操作未完成时调用 .GetAwaiter().GetResult()

如果您需要执行这些操作,请调有.AsTask(),并对生成的Task进行操作。

提示:

避免这些陷阱的最简单方法是直接等待方法调用

await Foo(); // Safe

当您将（值）任务分配给变量时，错误行为的大门打开了：

ValueTask<int> **valueTask = Foo()**; // Caution!

// Our use of valueTask can now lead to errors.

这可以通过立即转换为普通任务来减轻：

Task<int> task = Foo()**.AsTask()**; // Safe

// task is safe to work with.

##### 避免过度弹跳

对于在循环中多次调用的方法，可以通过调用ConfigureAwait来避免重复跳转到UI消息循环的成本。这会强制任务不要将连续性反弹到同步上下文，从而将开销降低到更接近上下文切换的成本（如果您正在等待的方法同步完成，则会降低很多）：

async void A() { ... await B(); ... }

async Task B()

{

for (int i = 0; i < **1000**; i++)

await C()**.ConfigureAwait (false)**;

}

async Task C() { ... }

这意味着，对于B和C方法，我们取消了UI应用程序中的简单线程安全模型，即代码在UI线程上运行，并且只能在等待语句期间被抢占。但是，方法A不受影响，如果在UI线程上启动，它将保留在其中一个线程上。

这种优化在编写库时尤其重要：您不需要简化线程安全性的好处，因为您的代码通常不与调用方共享状态，也不访问UI控件。（在我们的例子中，如果方法C知道操作可能是短时间运行的，那么它同步完成也是有意义的。）

### 异步模式(Asynchronous Patterns)

#### 取消(Cancellation)

能够在并发操作启动后取消它通常很重要，可能是为了响应用户请求。实现这一点的一个简单方法是使用取消标志，我们可以通过编写这样的类来封装它：

class CancellationToken

{

public bool IsCancellationRequested { get; private set; }

public void Cancel() { IsCancellationRequested = true; }

public void ThrowIfCancellationRequested()

{

if (IsCancellationRequested)

throw new OperationCanceledException();

}

}

然后，我们可以编写一个可取消的异步方法，如下所示：

async Task Foo (CancellationToken cancellationToken)

{

for (int i = 0; i < 10; i++)

{

Console.WriteLine (i);

await Task.Delay (1000);

cancellationToken.ThrowIfCancellationRequested();

}

}

当调用者想要取消时，它会对传递到Foo的取消令牌调用cancel。这将IsCancellationRequested设置为true，这将导致Foo在短时间后出现OperationCanceledException（系统命名空间中为此目的设计的预定义异常）故障。

撇开线程安全不谈（我们应该在读/写IsCancellation-Requested时锁定），这种模式是有效的，CLR提供了一种称为CancellationToken的类型，它与我们刚才展示的非常相似。但是，它缺少Cancel方法；相反，此方法在另一个名为CancellationTokenSource的类型上公开。这种分离提供了一些安全性：只访问CancellationToken对象的方法可以检查但不能启动取消。

要获得取消令牌，我们首先实例化一个CancellationTokenSource：

var cancelSource = new CancellationTokenSource();

它公开了一个Token属性，该属性返回一个CancellationToken。因此，我们可以调用我们的Foo方法，如下所示：

var cancelSource = new CancellationTokenSource();

Task foo = Foo (cancelSource.Token);

...

... *(sometime later)*

cancelSource.Cancel();

CLR中的大多数异步方法都支持取消标记，包括Delay。如果我们修改Foo，使其将其令牌传递到Delay方法中，则任务将根据请求立即结束（而不是一秒钟后）：

async Task Foo (CancellationToken cancellationToken)

{

for (int i = 0; i < 10; i++)

{

Console.WriteLine (i);

await Task.Delay (1000, **cancellationToken**);

}

}

请注意，我们不再需要调用ThrowIfCancellationRequested, 因为Task.Delay为我做了一切。取消令牌在调用堆栈中很好地向下传播（就像取消请求由于是异常而级联到调用堆栈一样）。

提示:

UWP依赖于WinRT类型，其异步方法遵循较差的取消协议，IAsyncInfo类型公开了Cancel方法，而不是接受CancellationToken。然而，AsTask扩展方法被重载以接受取消令牌，从而弥补了这一差距。

同步方法也可以支持取消（例如Task的Wait方法）。在这种情况下，取消的指令需要异步发出（例如，来自另一个任务）。例如:

var cancelSource = new CancellationTokenSource();

**Task.Delay (5000).ContinueWith (ant => cancelSource.Cancel());**

...

事实上，您可以在构建CancellationTokenSource时指定一个时间间隔，以便在一段时间后启动取消（正如我们所演示的）。它对于实现超时非常有用，无论是同步的还是异步的：

var cancelSource = new CancellationTokenSource (5000);

try { await Foo (cancelSource.Token); }

catch (OperationCanceledException ex) {

Console.WriteLine ("Cancelled");

}

CancellationToken结构提供了一个Register方法，用于注册将在取消时激发的回调委托；它返回一个可以用来撤消注册的对象。

编译器的异步函数生成的任务在出现未处理的OperationCanceledException（IsCanceled返回true，IsFaulted返回false）时自动进入“Canceled”状态。使用Task.Run将其构造函数据参数传递（相同的）CancellationToken而创建的任务也是如此,。在异步场景中，故障任务和取消任务之间的区别并不重要，因为两者在等待时都会抛出OperationCanceledException；它在高级并行编程场景（特别是条件延续）中很重要。我们可以在第957页的“取消任务”中找到这个主题。

#### 进度报告(Process Resporting)

有时，您会希望异步操作在运行时报告进度。一个简单的解决方案是将Action委托传递给异步方法，每当进度发生变化时，该方法就会激发该方法：

Task Foo (Action<int> onProgressPercentChanged)

{

return Task.Run (() =>

{

for (int i = 0; i < 1000; i++)

{

if (i % 10 == 0) onProgressPercentChanged (i / 10);

// Do something compute-bound...

}

});

}

下是我们怎么调用它:

Action<int> progress = i => Console.WriteLine (i + " %");

await Foo (progress);

尽管这在控制台应用程序中运行良好，但在富客户端场景中并不理想，因为它报告来自工作线程的进度，从而给使用者带来潜在的线程安全问题。（实际上，我们已经允许并发的副作用“泄漏”到外部世界，这是不幸的，因为如果从UI线程调用该方法，则该方法会被隔离。）

#### IProgress<T> and Progress<T>

CLR提供了一对类型来解决这个问题：一个名为IProgress<T>的接口和一个实现该接口的类Progress<T>。实际上，它们的目的是“包装”委托，以便UI应用程序可以通过同步上下文安全地报告进度。

这个接口只定义了一个方法:

public interface IProgress<in T>

{

void Report (T value);

}

使用IProgress＜T＞很容易；我们的方法几乎没有改变：

Task Foo (**IProgress<int>** onProgressPercentChanged)

{

return Task.Run (() =>

{

for (int i = 0; i < 1000; i++)

{

if (i % 10 == 0) onProgressPercentChanged**.Report** (i / 10);

// Do something compute-bound...

}

});

}

Progress＜T＞类有一个构造函数，它接受包装的Action＜T＞类型的委托：

var progress = **new Progress<int>** (i => Console.WriteLine (i + " %"));

await Foo (progress);

（Progress＜T＞还有一个ProgressChanged事件，您可以订阅该事件，而不是[或除了]将操作委托传递给构造函数。）实例化Progress＜int＞后，该类将捕获同步上下文（如果存在）。当Foo随后调用Report时，将通过该上下文调用委托。

异步方法可以通过将int替换为公开一系列属性的自定义类型来实现更详细的进度报告。

提示:

如果您熟悉Rx，您会注意到IProgress＜T＞与异步函数返回的任务一起提供了一个类似于IObserver＜T＞的功能集。不同之处在于，除了IProgress＜T＞发出的值之外，任务还可以公开“最终”返回值（并对其进行不同类型的处理）。

IProgress＜T＞发出的值通常是“一次性”值（例如，到目前为止完成的百分比或下载的字节数），而IObserver＜T＞的OnNext推送的值通常包括结果本身，也是调用它的原因。

WinRT中的异步方法也提供进度报告，尽管COM的（相对）基元类型系统使协议变得复杂。报告进度的异步WinRT方法不接受IProgress＜T＞对象，而是返回以下接口之一，以代替IAsyncAction和IAsyncOperation＜TResult＞：

IAsyncActionWithProgress<TProgress>

IAsyncOperationWithProgress<TResult, TProgress>

有趣的是，两者都基于IAsyncInfo（而不是IAsyncAction和IAsyncOperation＜TResult＞）。

好消息是，AsTask扩展方法也被重载以接受前面提到的接口的IProgress＜T＞，因此是.NET使用者，您可以忽略COM接口并执行以下操作：

var progress = new Progress<int> (i => Console.WriteLine (i + " %"));

CancellationToken cancelToken = ...

var task = someWinRTobject.FooAsync().**AsTask** (cancelToken, progress);

#### 基于任务的异步模式

.NET公开了数百个可以等待的任务返回异步方法（主要与I/O有关）。这些方法中的大多数（至少部分）遵循一种称为基于任务的异步模式（TAP）的模式，这是我们迄今为止所描述的内容的合理形式化。TAP方法执行以下操作：

* 返回一个"热"(正在运行)Task或Task<Tresult>
* 有一个”Async”后缀(除了特殊的比如任务组合器)
* 如果支持取消和/或进度报告，则重载以接受取消令牌和/或IProgress<T>
* 快速返回给调用者（只有一个小的初始同步阶段）
* 如果I/O绑定，则不会占用线程

正如所见, TAP方法非常容易的写C#异步函数

#### 任务组合器(Task Combinators)

异步函数有一个一致的协议（即它们一致地返回任务）的一个好结果是，可以使用和编写任务组合符——这些函数可以有效地组合任务，而不考虑这些特定任务的作用。

CLR包括两个任务组合子：Task.WhenAny和Task.WheAll。在描述它们时，我们将假设定义了以下方法：

async Task<int> Delay1() { await Task.Delay (1000); return 1; }

async Task<int> Delay2() { await Task.Delay (2000); return 2; }

async Task<int> Delay3() { await Task.Delay (3000); return 3; }

##### WhenAny

Task.WhenAny返回在一组任务中的任何一个任务完成时完成的任务。

以下操作将在一秒钟内完成：

Task<int> winningTask = await Task.WhenAny (Delay1(), Delay2(), Delay3());

Console.WriteLine ("Done");

Console.WriteLine (winningTask.Result); // 1

因为任务。当任何一个返回任务时，我们等待它，它会返回最先完成的任务。我们的示例是完全非阻塞的——包括访问Result属性时的最后一行（因为winningTask已经完成）。尽管如此，通常最好等待获胜任务：

Console.WriteLine (await winningTask); // 1

因为任何异常都会在没有AggregateException包装的情况下重新抛出。事实上，我们可以在一个步骤中执行两个等待：

int answer = await await Task.WhenAny (Delay1(), Delay2(), Delay3());

如果一个未获胜的任务随后出现故障，则除非您随后等待该任务（或查询其exception属性），否则将不会观察到异常。

WhenAny可用于将超时或取消应用于不支持它的操作：

Task<string> task = SomeAsyncFunc();

Task winner = await (Task.WhenAny (task, Task.Delay(5000)));

if (winner != task) throw new TimeoutException();

string result = await task; // Unwrap result/re-throw

请注意，因为在这种情况下，我们使用不同类型的任务调用WhenAny，所以获胜者被报告为普通任务（而不是Task<string>）。

##### WhenAll

Task.WhenAll返回在您传递给它的所有任务完成时完成的任务。三秒钟后完成以下操作（并演示fork/join模式）：

await Task.WhenAll (Delay1(), Delay2(), Delay3());

我们可以通过依次等待task1、task2和task3而不是使用WhenAll来获得类似的结果：

Task task1 = Delay1(), task2 = Delay2(), task3 = Delay3();

await task1; await task2; await task3;

不同之处（除了由于需要三个等待而不是一个等待而效率较低之外）在于，如果task1出现故障，我们将永远无法等待task2/task3，并且它们的任何异常都将无法观察到。

相比之下，Task.WhenAll直到所有任务都完成才完成——即使出现故障。如果有多个错误，它们的异常会合并到任务的AggregateException中（此时AggregateException实际上变得有用——如果您对所有异常都感兴趣的话）。然而，等待组合任务只会引发第一个异常，因此要查看所有异常，您需要执行以下操作：

Task task1 = Task.Run (() => { throw null; } );

Task task2 = Task.Run (() => { throw null; } );

Task all = Task.WhenAll (task1, task2);

try { await all; }

catch

{

Console.WriteLine (all.Exception.InnerExceptions.Count); // 2

}

使用Task<Tresult>类型的任务调用WhenAll会返回Task<Trasult[]>，给出所有任务的组合结果。当等待时，这会减少为TResult[]：

Task<int> task1 = Task.Run (() => 1);

Task<int> task2 = Task.Run (() => 2);

int[] results = await Task.WhenAll (task1, task2); // { 1, 2 }

为了给出一个实际的例子，以下并行下载URI，并对它们的总长度求和：

async Task<int> GetTotalSize (string[] uris)

{

IEnumerable<Task<byte[]>> downloadTasks = uris.Select (uri =>

new WebClient().DownloadDataTaskAsync (uri));

byte[][] contents = await Task.WhenAll (downloadTasks);

return contents.Sum (c => c.Length);

}

不过，这里有一点效率低下，因为我们不必要地挂在下载的字节数组上，直到每个任务都完成。如果我们在下载字节数组后立即将其折叠成长度，那么效率会更高。这就是异步lambda派上用场的地方，因为我们需要将等待表达式馈送到LINQ的Select查询运算符中：

async Task<int> GetTotalSize (string[] uris)

{

IEnumerable<Task<**int**>> downloadTasks = uris.Select (**async** uri =>

(**await** new WebClient().DownloadDataTaskAsync (uri)).**Length**);

**int[]** contentLengths = await Task.WhenAll (downloadTasks);

return contentLengths.**Sum()**;

}

##### 自定义组合器(Custom Combinators)

编写自己的任务组合器可能很有用。最简单的“组合器”接受单个任务，例如下面的任务，它允许您等待任何有超时的任务：

async static Task<TResult> WithTimeout<TResult> (this Task<TResult> task,TimeSpan timeout)

{

Task winner = await Task.WhenAny (task, Task.Delay (timeout))

.ConfigureAwait (false);

if (winner != task) throw new TimeoutException();

return await task.ConfigureAwait (false); // Unwrap result/re-throw

}

因为这在很大程度上是一种不访问外部共享状态的“库方法”，所以我们在等待时使用ConfigureAwait（false），以避免可能跳到UI同步上下文。我们可以通过取消任务来进一步提高效率。当任务按时完成时延迟（这避免了定时器的小开销）：

async static Task<TResult> WithTimeout<TResult> (this Task<TResult> task,TimeSpan timeout)

{

var cancelSource = new CancellationTokenSource();

var delay = Task.Delay (timeout, cancelSource.Token);

Task winner = await Task.WhenAny (task, delay)

.ConfigureAwait (false);

if (winner == task)

cancelSource.Cancel();

else

throw new TimeoutException();

return await task.ConfigureAwait (false); // Unwrap result/re-throw

}

以下允许您通过CancellationToken“放弃”任务

static Task<TResult> WithCancellation<TResult> (this Task<TResult> task,CancellationToken cancelToken)

{

var tcs = new TaskCompletionSource<TResult>();

var reg = cancelToken.Register (() => tcs.TrySetCanceled ());

task.ContinueWith (ant =>

{

reg.Dispose();

if (ant.IsCanceled)

tcs.TrySetCanceled();

else if (ant.IsFaulted)

tcs.TrySetException (ant.Exception.InnerExceptions);

else

tcs.TrySetResult (ant.Result);

});

return tcs.Task;

}

任务组合子的编写可能很复杂，有时需要使用信号结构，我们将在第21章中介绍。这实际上是一件好事，因为它将与并发相关的复杂性排除在业务逻辑之外，并引入可以单独测试的可重用方法中。

下一个组合子的工作方式与WhenAll类似，只是如果任何任务出错，则生成的任务会立即出错：

async Task<TResult[]> WhenAllOrError<TResult>

(params Task<TResult>[] tasks)

{

var killJoy = new TaskCompletionSource<TResult[]>();

foreach (var task in tasks)

task.ContinueWith (ant =>

{

if (ant.IsCanceled)

killJoy.TrySetCanceled();

else if (ant.IsFaulted)

killJoy.TrySetException (ant.Exception.InnerExceptions);

});

return await await Task.WhenAny (killJoy.Task, Task.WhenAll(tasks))

.ConfigureAwait (false);

}

我们首先创建一个TaskCompletionSource，它的唯一工作是在任务出错时结束参与方。因此，我们从不调用它的SetResult方法，只调用它的TrySetCanceled和TrySetException方法。在这种情况下，ContinueWith比GetAwaiter().OnCompleted更方便。因为我们没有访问任务的结果，并且不想在这一点上跳转到UI线程。

##### 异步锁

在第906页的“异步信号量和锁”中，我们描述了如何使用SemaphoreSlim异步锁定或限制并发。

### 废弃模式

.NET采用了其他异步模式，这些模式位于任务和异步函数之前。由于基于任务的异步已经成为主流模式，因此很少需要这些功能。

#### 异步编程模型(Asynchronous Programming Model)

最古老的模式称为异步编程模型（APM），使用一对从“Begin”和“End”开始的方法和一个名为IAsyncResult的接口。为了进行说明，让我们以System中的Stream类为例。IO并查看其读取方法。首先，同步版本：

public int Read (byte[] buffer, int offset, int size);

您可能可以预测基于任务的异步版本是什么样子的：

public **Task<**int**>** Read**Async** (byte[] buffer, int offset, int size);

现在，让我们检查APM版本：

public **IAsyncResult Begin**Read (byte[] buffer, int offset, int size,

**AsyncCallback callback, object state**);

**public int EndRead (IAsyncResult asyncResult);**

调用Begin\*方法启动操作，返回一个IAsyncResult对象，该对象充当异步操作的令牌。当操作完成（或出现故障）时，AsyncCallback委托将激发：

public delegate void AsyncCallback (IAsyncResult ar);

然后，处理此委托的人调用End\*方法，该方法提供操作的返回值，并在操作出错时重新引发异常。

APM不仅难以使用，而且难以正确实现。处理APM方法最简单的方法是调用Task.Factory.FromAsync适配器方法，该方法将APM方法对转换为Task。在内部，它使用TaskCompletionSource为您提供一个任务，该任务在APM操作完成或出现故障时发出信号。

FromAsync方法需要以下参数：

* 一个BeginXXX的委托方法
* 一个EndXXX的委托方法
* 传递到上面两个方法的附加参数

FromAsync重载以接受与中几乎所有异步方法签名匹配的委托类型和参数。NET。例如，假设流是stream，缓冲区是byte[]，我们可以这样做：

Task<int> readChunk = Task<int>.Factory.FromAsync(

stream.BeginRead, stream.EndRead, buffer, 0, 1000, null);

#### 事件异步模式(Event-Based Asynchronous Pattern)

基于事件的异步模式（EAP）于2005年引入，为APM提供了一种更简单的替代方案，尤其是在UI场景中。然而，它只在少数几种类型中实现，最显著的是System.Net中的WebClient. EAP只是一种模式；没有提供任何类型的帮助。从本质上讲，模式是这样的：类提供了一个内部管理并发性的成员家族，类似于以下内容：

// These members are from the WebClient class:

public byte[] DownloadData (Uri address); // Synchronous version

public void DownloadData**Async** (Uri address);

public void DownloadData**Async** (Uri address, **object userToken**);

public event DownloadData**CompletedEventHandler** DownloadData**Completed**;

**public void CancelAsync (object userState); // Cancels an operation**

**public bool IsBusy { get; } // Indicates if still running**

\*Async方法异步启动操作。操作完成后，会触发\*Completed事件（如果存在，则自动发布到捕获的同步上下文）。此事件传递回一个包含以下内容的事件参数对象：

* 指示操作是否已取消的标志（由调用CancelAsync的使用者取消）
* 一个Error对象，指示抛出的异常（如果有）
* userToken对象（如果在调用Async方法时提供）

EAP类型还可以公开进度报告事件，每当进度发生变化时就会触发该事件（也通过同步上下文发布）：

public event DownloadProgressChangedEventHandler DownloadProgressChanged;

实现EAP需要大量的样板代码，这使得模式的组成性很差。

#### BackgroundWorker

System. ComponentModel中的BackgroundWorker是EAP的通用实现。它允许富客户端应用程序启动工作线程并报告完成情况和基于百分比的进度，而无需显式捕获同步上下文。以下是一个示例：

var worker = new BackgroundWorker { WorkerSupportsCancellation

= true };

worker.DoWork += (sender, args) =>

{ // This runs on a worker thread

if (args.Cancel) return;

Thread.Sleep(1000);

args.Result = 123;

};

worker.RunWorkerCompleted += (sender, args) =>

{ // Runs on UI thread

// We can safely update UI controls here...

if (args.Cancelled)

Console.WriteLine ("Cancelled");

else if (args.Error != null)

Console.WriteLine ("Error: " + args.Error.Message);

else

Console.WriteLine ("Result is: " + args.Result);

};

worker.RunWorkerAsync();//Captures sync context and starts

// operation