C#窗体多线程

如果应用程序在控制用户界面的线程上执行非 UI 处理，则会使应用程序的运行
显得缓慢而迟钝，让用户难以忍受。但是长期以来，编写适用于 Windows 的多线程应用程序只限于 C++ 开发人员。现在有了 .NET Framework，您就可以充分利用 C# 中的多线程来控制程序中的指令流，并使 UI 线程独立出来以便用户界面能够迅速响应。本文将向您介绍如何实现这一目标。此外，本文还将讨论多线程的缺陷并提供一个框架来保护并发线程执行的安全。

用 户不喜欢反应慢的程序。程序反应越慢，就越没有用户会喜欢它。在执行耗时较长的操作时，使用多线程是明智之举，它可以提高程序 UI 的响应速度，使得一切运行显得更为快速。在 Windows 中进行多线程编程曾经是 C++ 开发人员的专属特权，但是现在，可以使用所有兼容 Microsoft .NET 的语言来编写，其中包括 Visual Basic.NET。不过，Windows 窗体对线程的使用强加了一些重要限制。本文将对这些限制进行阐释，并说明如何利用它们来提供快速、高质量的 UI 体验，即使是程序要执行的任务本身速度就较慢。

为什么选择多线程？

多线程程序要比单线程程序更难于编写，并且不 加选择地使用线程也是导致难以找到细小错误的重要原因。这就自然会引出两个问题：为什么不坚持编写单线程代码？如果必须使用多线程，如何才能避免缺陷呢？ 本文的大部分篇幅都是在回答第二个问题，但首先我要来解释一下为什么确实需要多线程。

多线程处理可以使您能够通过确保程序“永不睡眠”从 而保持 UI 的快速响应。大部分程序都有不响应用户的时候：它们正忙于为您执行某些操作以便响应进一步的请求。也许最广为人知的例子就是出现在“打开文件”对话框顶部 的组合框。如果在展开该组合框时，CD-ROM 驱动器里恰好有一张光盘，则计算机通常会在显示列表之前先读取光盘。这可能需要几秒钟的时间，在此期间，程 序既不响应任何输入，也不允许取消该操作，尤其是在确实并不打算使用光驱的时候，这种情况会让人无法忍受。

执行这种操作期间 UI 冻结的原因在于，UI 是个单线程程序，单线程不可能在等待 CD-ROM 驱动器读取操作的同时处理用户输入，如图 1 所示。“打开文件”对话框会调用某些阻塞 (blocking) API 来确定 CD-ROM 的标题。阻塞 API 在未完成自己的工作之前不会返回，因此这期间它会阻止线程做其他事情。

在多线程下，像这样耗时较长的任务就可以在其自己 的线程中运行，这些线程通常称为辅助线程。因为只有辅助线程受到阻止，所以阻塞操作不再导致用户界面冻结，如图 2 所示。应用程序的主线程可以继续处理用户的鼠标和键盘输入

的同时，受阻的另一个线程将等待 CD-ROM 读取，或执行辅助线程可能做的任何操作。

其 基本原则是，负责响应用户输入和保持用户界面为最新的线程（通常称为 UI 线程）不应该用于执行任何耗时较长的操作。惯常做法是，任何耗时超过 30ms 的操作都要考虑从 UI 线程中移除。这似乎有些夸张，因为 30ms 对于大多数人而言只不过是他们可以感觉到的最短的瞬间停顿，实际上该停顿略短于电影屏幕中显示的连续帧之间的间隔。

如果鼠标单击和相应的 UI 提示（例如，重新绘制按钮）之间的延迟超过 30ms，那么操作与显示之间就会稍显不连贯，并因此产生如同影片断帧那样令人心烦的感觉。为了达到完全高质量的响应效果，上限必须是 30ms。另一方面，如果您确实不介意感觉稍显不连贯，但也不想因为停顿过长而激怒用户，则可按照通常用户所能容忍的限度将该间隔设为 100ms。

这意味着如果想让用户界面保持响应迅速，则任何阻塞操作都应该在辅助线程中执行 — 不管是机械等待某事发生（例如，等待 CD-ROM 启动或者硬盘定位数据），还是等待来自网络的响应。

异步委托调用

在 辅助线程中运行代码的最简单方式是使用异步委托调用（所有委托都提供该功能）。委托通常是以同步方式进行调用，即，在调用委托时，只有包装方法返回后该调 用才会返回。要以异步方式调用委托，请调用 BeginInvoke 方法，这样会对该方法排队以在系统线程池的线程中运行。调用线程会立即返回，而不用等待该方法完成。这比较适合于 UI 程序，因为可以用它来启动耗时较长的作业，而不会使用户界面反应变慢。

例如，在以下代码中，System.Windows.Forms.MethodInvoker 类型是一个系统定义的委托，用于调用不带参数的方法。

程序代码

```
private void StartSomeWorkFromUIThread () {
    // The work we want to do is too slow for the UI
    // thread, so let's farm it out to a worker thread.

    MethodInvoker mi = new MethodInvoker(
        RunsOnWorkerThread);
    mi.BeginInvoke(null, null); // This will not block.
}

// The slow work is done here, on a thread
```

```
// from the system thread pool.
private void RunsOnWorkerThread() {
    DoSomethingSlow();
}
```

如 果想要传递参数，可以选择合适的系统定义的委托类型，或者自己来定义委托。MethodInvoker 委托并没有什么神奇之处。和其他委托一样，调用 BeginInvoke 会使该方法在系统线程池的线程中运行，而不会阻塞 UI 线程以便其可执行其他操作。对于以上情况，该方法不返回数据，所以启动它后就不用再去管它。如果您需要该方法返回的结果，则 BeginInvoke 的返回值很重要，并且您可能不传递空参数。然而，对于大多数 UI 应用程序而言，这种“启动后就不管”的风格是最有效的，稍后会对原因进行简要讨论。您应该注意到，BeginInvoke 将返回一个 IAsyncResult。这可以和委托的 EndInvoke 方法一起使用，以在该方法调用完毕后检索调用结果。

还有其他一些可用于 在另外的线程上运行方法的技术，例如，直接使用线程池 API 或者创建自己的线程。然而，对于大多数用户界面应用程序而言，有异步委托调用就足够了。采用这种技术不仅编码容易，而且还可以避免创建并非必需的线程，因 为可以利用线程池中的共享线程来提高应用程序的整体性能。

线程和控件

Windows 窗体体系结构对线程使用制定了严格的规则。如果只是编写单线程应用程序，则没必要知道这些规则，这是因为单线程的代码不可能违反这些规则。然而，一旦采用 多线程，就需要理解 Windows 窗体中最重要的一条线程规则：除了极少数的例外情况，否则都不要在它的创建线程以外的线程中使用控件的任何成员。

本 规则的例外情况有文档说明，但这样的情况非常少。这适用于其类派生自 System.Windows.Forms.Control 的任何对象，其中几乎包括 UI 中的所有元素。所有的 UI 元素（包括表单本身）都是从 Control 类派生的对象。此外，这条规则的结果是一个被包含的控件（如，包含在一个表单中的按钮）必须与包含它控件位处于同一个线程中。也就是说，一个窗口中的所有 控件属于同一个 UI 线程。实际中，大部分 Windows 窗体应用程序最终都只有一个线程，所有 UI 活动都发生在这个线程上。这个线程通常称为 UI 线程。这意味着您不能调用用户界面中任意控件上的任何方法，除非在该方法的文档说明中指出可以调用。该规则的例外情况（总有文档记录）非常少而且它们之间 关系也不大。请注意，以下代码是非法的：

程序代码

```
// Created on UI thread
```

```
private Label lblStatus;
...
// Doesn't run on UI thread
private void RunsOnWorkerThread() {
    DoSomethingSlow();
    lblStatus.Text = "Finished!";    // BAD!!
}
```

如 果您在 .NET Framework 1.0 版本中尝试运行这段代码，也许会侥幸运行成功，或者初看起来是如此。这就是多线程错误中的主要问题，即它们并不会立即显现出来。甚至当出现了一些错误时， 在第一次演示程序之前一切看起来也都很正常。但不要搞错 — 我刚才显示的这段代码明显违反了规则，并且可以预见，任何抱希望于“试运行时良好，应该就没有问题”的人在即将到来的调试期是会付出沉重代价的。

要 注意，在明确创建线程之前会发生这样的问题。使用委托的异步调用实用程序（调用它的 BeginInvoke 方法）的任何代码都可能出现同样的问题。委托提供了一个非常吸引人的解决方案来处理 UI 应用程序中缓慢、阻塞的操作，因为这些委托能使您轻松地让此种操作运行在 UI 线程外而无需自己创建新线程。但是由于以异步委托调用方式运行的代码在一个来自线程池的线程中运行，所以它不能访问任何 UI 元素。上述限制也适用于线程池中的线程和手动创建的辅助线程。

在正确的线程中调用控件

有关控件的限制看 起来似乎对多线程编程非常不利。如果在辅助线程中运行的某个缓慢操作不对 UI 产生任何影响，用户如何知道它的进行情况呢？至少，用户如何知道工作何时完成或者是否出现错误？幸运的是，虽然此限制的存在会造成不便，但并非不可逾越。 有多种方式可以从辅助线程获取消息，并将该消息传递给 UI 线程。理论上讲，可以使用低级的同步原理和池化技术来生成自己的机制，但幸运的是，因为有一个以 Control 类的 Invoke 方法形式存在的解决方案，所以不需要借助于如此低级的工作方式。

Invoke 方法是 Control 类中少数几个有文档记录的线程规则例外之一：它始终可以对来自任何线程的 Control 进行 Invoke 调用。Invoke 方法本身只是简单地携带委托以及可选的参数列表，并在 UI 线程中为您调用委托，而不考虑 Invoke 调用是由哪个线程发出的。实际上，为控件获取任何方法以在正确的线程上运行非常简单。但应该注意，只有在 UI 线程当前未受到阻塞时，这种机制才有效 — 调用只有在 UI 线程准备处理用户输入时才能通过。从不阻塞 UI 线程还有另一个好理由。Invoke 方法会进行测试以了解调用线程是否就是 UI 线程。如果是，它就直接调用委托。否则，它将安排线程切换，并在 UI 线程上调用委托。无论是哪种情况，委托所包装的方法都会在 UI 线程中运行，并且只有

当该方法完成时，Invoke 才会返回。

Control 类也支持异步版本的 Invoke，它会立即返回并安排该方法以便在将来某一时间在 UI 线程上运行。这称为 BeginInvoke，它与异步委托调用很相似，与委托的明显区别在于，该调用以异步方式在线程池的某个线程上运行，然而在此处，它以异步方式在 UI 线程上运行。实际上，Control 的 Invoke、BeginInvoke 和 EndInvoke 方法，以及 InvokeRequired 属性都是 ISynchronizeInvoke 接口的成员。该接口可由任何需要控制其事件传递方式的类实现。

由 于 BeginInvoke 不容易造成死锁，所以尽可能多用该方法；而少用 Invoke 方法。因为 Invoke 是同步的，所以它会阻塞辅助线程，直到 UI 线程可用。但是如果 UI 线程正在等待辅助线程执行某操作，情况会怎样呢？应用程序会死锁。BeginInvoke 从不等待 UI 线程，因而可以避免这种情况。

现 在，我要回顾一下前面所展示的代码片段的合法版本。首先，必须将一个委托传递给 Control 的 BeginInvoke 方法，以便可以在 UI 线程中运行对线程敏感的代码。这意味着应该将该代码放在它自己的方法中，如图 3 所示。一旦辅助线程完成缓慢的工作后，它就会调用 Label 中的 BeginInvoke，以便在其 UI 线程上运行某段代码。通过这样，它可以更新用户界面。

包装 Control.Invoke

虽 然图 3 中的代码解决了这个问题，但它相当繁琐。如果辅助线程希望在结束时提供更多的反馈信息，而不是简单地给出“Finished!”消息，则 BeginInvoke 过于复杂的使用方法会令人生畏。为了传达其他消息，例如“正在处理”、“一切顺利”等等，需要设法向 UpdateUI 函数传递一个参数。可能还需要添加一个进度栏以提高反馈能力。这么多次调用 BeginInvoke 可能导致辅助线程受该代码支配。这样不仅会造成不便，而且考虑到辅助线程与 UI 的协调性，这样设计也不好。对这些进行分析之后，我们认为包装函数可以解决这两个问题，如图 4 所示。

ShowProgress 方法对将调用引向正确线程的工作进行封装。这意味着辅助线程代码不再担心需要过多关注 UI 细节，而只要定期调用 ShowProgress 即可。请注意，我定义了自己的方法，该方法违背了“必须在 UI 线程上进行调用”这一规则，因为它进而只调用不受该规则约束的其他方法。这种技术会引出一个较为常见的话题：为什么不在控件上编写公共方法呢（这些方法记 录为 UI 线程规则的例外）？

刚好 Control 类为这样的方法提供了一个有用的工具。如果我提供一个设计为可从任何线程调用的公共方法，则完全有可能某人会从 UI 线程调用这个方法。在这种情况下，没必要调用 BeginInvoke，因为我已经处于正确的线程中。调用

Invoke 完全是浪费时间和资源，不如直接调用适当的方法。为了避免这种情况，Control 类将公开一个称为 InvokeRequired 的属性。这是“只限 UI 线程”规则的另一个例外。它可从任何线程读取，如果调用线程是 UI 线程，则返回假，其他线程则返回真。这意味着我可以按以下方式修改包装：

程序代码

```
public void ShowProgress(string msg, int percentDone) {
    if (InvokeRequired) {
        // As before
        ...
    } else {
        // We're already on the UI thread just
        // call straight through.
        UpdateUI(this, new MyProgressEvents(msg,
            PercentDone));
    }
}
```

ShowProgress 现在可以记录为可从任何线程调用的公共方法。这并没有消除复杂性 — 执行 BeginInvoke 的代码依然存在，它还占有一席之地。不幸的是，没有简单的方法可以完全摆脱它。

锁定

任 何并发系统都必须面对这样的事实，即，两个线程可能同时试图使用同一块数据。有时这并不是问题 — 如果多个线程在同一时间试图读取某个对象中的某个字段，则不会有问题。然而，如果有线程想要修改该数据，就会出现问题。如果线程在读取时刚好另一个线程正 在写入，则读取线程有可能会看到虚假值。如果两个线程在同一时间、在同一个位置执行写入操作，则在同步写入操作发生之后，所有从该位置读取数据的线程就有 可能看到一堆垃圾数据。虽然这种行为只在特定情况下才会发生，读取操作甚至不会与写入操作发生冲突，但是数据可以是两次写入结果的混加，并保持错误结果直 到下一次写入值为止。为了避免这种问题，必须采取措施来确保一次只有一个线程可以读取或写入某个对象的状态。

防止这些问题出现所采用的方 式是，使用运行时的锁定功能。C# 可以让您利用这些功能、通过锁定关键字来保护代码（Visual Basic 也有类似构造，称为 SyncLock）。规则是，任何想要在多个线程中调用其方法的对象在每次访问其字段时（不管是读取还是写入）都应该使用锁定构造。例如，请参见图 5。

锁 定构造的工作方式是：公共语言运行库 (CLR) 中的每个对象都有一个与之相关的锁，任何线程均可获得该锁，但每次只能有一个线程拥有它。如果某个线程

试图获取另一个线程已经拥有的锁，那么它必须等待， 直到拥有该锁的线程将锁释放为止。C# 锁定构造会获取该对象锁（如果需要，要先等待另一个线程利用它完成操作），并保留到大括号中的代码退出为止。如果执行语句运行到块结尾，该锁就会被释放， 并从块中部返回，或者抛出在块中没有捕捉到的异常。

请注意，MoveBy 方法中的逻辑受同样的锁语句保护。当所做的修改比简单的读取或写入更复杂时，整个过程必须由单独的锁语句保护。这也适用于对多个字段进行更新 — 在对象处于一致状态之前，一定不能释放该锁。如果该锁在更新状态的过程中释放，则其他线程也许能够获得它并看到不一致状态。如果您已经拥有一个锁，并调用 一个试图获取该锁的方法，则不会导致问题出现，因为单独线程允许多次获得同一个锁。对于需要锁定以保护对字段的低级访问和对字段执行的高级操作的代码，这 非常重要。MoveBy 使用 Position 属性，它们同时获得该锁。只有最外面的锁阻塞完成后，该锁才会恰当地释放。

对于需要锁 定的代码，必须严格进行锁定。稍有疏漏，便会功亏一篑。如果一个方法在没有获取对象锁的情况下修改状态，则其余的代码在使用它之前即使小心地锁定对象也是 徒劳。同样，如果一个线程在没有事先获得锁的情况下试图读取状态，则它可能读取到不正确的值。运行时无法进行检查来确保多线程代码正常运行。

死锁

锁是确保多线程代码正常运行的基本条件，即使它们本身也会引入新的风险。在另一个线程上运行代码的最简单方式是，使用异步委托调用（请参见图 6）。

如 果曾经调用过 Foo 的 CallBar 方法，则这段代码会慢慢停止运行。CallBar 方法将获得 Foo 对象上的锁，并直到 BarWork 返回后才释放它。然后，BarWork 使用异步委托调用，在某个线程池线程中调用 Foo 对象的 FooWork 方法。接下来，它会在调用委托的 EndInvoke 方法前执行一些其他操作。EndInvoke 将等待辅助线程完成，但辅助线程却被阻塞在 FooWork 中。它也试图获取 Foo 对象的锁，但锁已被 CallBar 方法持有。所以，FooWork 会等待 CallBar 释放锁，但 CallBar 也在等待 BarWork 返回。不幸的是，BarWork 将等待 FooWork 完成，所以 FooWork 必须先完成，它才能开始。结果，没有线程能够进行下去。

这就是一个死锁的例子，其中有两个或更多线程都被阻塞以等待对方进行。这里的情形 和标准死锁情况还是有些不同，后者通常包括两个锁。这表明如果有某个因果性（过程调用链）超出线程界限，就会发生死锁，即使只包括一个 锁！Control.Invoke 是一种跨线程调用过程的方法，这是个不争的重要事实。BeginInvoke 不会遇到这样的问题，因为它并不会使因果性跨线程。实际上，它会在某个线程池线程中启动一个全新的因果性，以允许原有的那个独立进行。然而，如果保留 BeginInvoke 返回的 IAsyncResult，并用它调用 EndInvoke，则又会出现问题，因为 EndInvoke 实际上已将两个因果性合二为一。避免这种情

况的最简单方法是，当持有一个对象锁时，不要等待跨线程调用完成。要确保这一点，应该避免在锁语句中调用 Invoke 或 EndInvoke。其结果是，当持有一个对象锁时，将无需等待其他线程完成某操作。要坚持这个规则，说起来容易做起来难。

在 检查代码的 BarWork 时，它是否在锁语句的作用域内并不明显，因为在该方法中并没有锁语句。出现这个问题的唯一原因是 BarWork 调用自 Foo.CallBar 方法的锁语句。这意味着只有确保正在调用的函数并不拥有锁时，调用 Control.Invoke 或 EndIn-voke 才是安全的。对于非私有方法而言，确保这一点并不容易，所以最佳规则是，根本不调用 Control.Invoke 和 EndInvoke。这就是为什么“启动后就不管”的编程风格更可取的原因，也是为什么 Control.BeginInvoke 解决方案通常比 Control.Invoke 解决方案好的原因。

有时候除了破坏规则别无选择，这种情况下就需要仔细严格地分析。但只要可能，在持有锁时就应该避免阻塞，因为如果不这样，死锁就难以消除。

使其简单

如 何既从多线程获益最大，又不会遇到困扰并发代码的棘手错误呢？如果提高的 UI 响应速度仅仅是使程序时常崩溃，那么很难说是改善了用户体验。大部分在多线程代码中普遍存在的问题都是由要一次运行多个操作的固有复杂性导致的，这是因为 大多数人更善于思考连续过程而非并发过程。通常，最好的解决方案是使事情尽可能简单。

UI 代码的性质是：它从外部资源接收事件，如用户输入。它会在事件发生时对其进行处理，但却将大部分时间花在了等待事件的发生。如果可以构造辅助线程和 UI 线程之间的通信，使其适合该模型，则未必会遇到这么多问题，因为不会再有新的东西引入。我是这样使事情简单化的：将辅助线程视为另一个异步事件源。如同 Button 控件传递诸如 Click 和 MouseEnter 这样的事件，可以将辅助线程视为传递事件（如 ProgressUpdate 和 WorkComplete）的某物。只是简单地将这看作一种类比，还是真正将辅助对象封装在一个类中，并按这种方式公开适当的事件，这完全取决于您。后一 种选择可能需要更多的代码，但会使用户界面代码看起来更加统一。不管哪种情况，都需要 Control.BeginInvoke 在正确的线程上传递这些事件。

对于辅助线程，最简单的方式是将代码编写为正常顺序的代码块。但如果想要使用刚才介绍的“将辅助线程作为事件源”模型，那又该如何呢？这个模型非常适用，但它对该代码与用户界面的交互提出了限制：这个线程只能向 UI 发送消息，并不能向它提出请求。

例 如，让辅助线程中途发起对话以请求完成结果需要的信息将非常困难。如果确实需要这样做，也最好是在辅助线程中发起这样的对话，而不要在主 UI 线程

中发起。该约束是有利的，因为它将确保有一个非常简单且适用于两线程间通信的模型 — 在这里简单是成功的关键。这种开发风格的优势在于，在等待另一个线程时，不会出现线程阻塞。这是避免死锁的有效策略。

图 7 显示了使用异步委托调用以在辅助线程中执行可能较慢的操作（读取某个目录的内容），然后将结果显示在 UI 上。它还不至于使用高级事件语法，但是该调用确实是以与处理事件（如单击）非常相似的方式来处理完整的辅助代码。

取消

前 面示例所带来的问题是，要取消操作只能通过退出整个应用程序实现。虽然在读取某个目录时 UI 仍然保持迅速响应，但由于在当前操作完成之前程序将禁用相关按钮，所以用户无法查看另一个目录。如果试图读取的目录是在一台刚好没有响应的远程机器上，这 就很不幸，因为这样的操作需要很长时间才会超时。

要取消一个操作也比较困难，尽管这取决于怎样才算取消。一种可能的理解是“停止等待这个 操作完成，并继续另一个操作。”这实际上是抛弃进行中的操作，并忽略最终完成时可能产生的后果。对于当前示例，这是最好的选择，因为当前正在处理的操作 （读取目录内容）是通过调用一个阻塞 API 来执行的，取消它没有关系。但即使是如此松散的“假取消”也需要进行大量工作。如果决定启动新的读取操作而不等待原来的操作完成，则无法知道下一个接收到 的通知是来自这两个未处理请求中的哪一个。

支持取消在辅助线程中运行的请求的唯一方式是，提供与每个请求相关的某种调用对象。最简单的做 法是将它作为一个 Cookie，由辅助线程在每次通知时传递，允许 UI 线程将事件与请求相关联。通过简单的身份比较（参见图 8），UI 代码就可以知道事件是来自当前请求，还是来自早已废弃的请求。

如果简单抛弃就行，那固然很好，不过您可能想要做得更好。如果辅助线程执行的是进行一连串阻塞操作的复杂操作，那么您可能希望辅助线程在最早的时机停止。否则，它可能会继续几分钟的无用操作。在这种情况下，调用对象需要做的就不 止是作为一个被动 Cookie。它至少还需要维护一个标记，指明请求是否被取消。UI 可以随时设置这个标记，而辅助线程在执行时将定期测试这个标记，以确定是否需要放弃当前工作。

对于这个方案，还需要做出几个决定：如果 UI 取消了操作，它是否要等待直到辅助线程注意到这次取消？如果不等待，就需要考虑一个争用条件：有可能 UI 线程会取消该操作，但在设置控制标记之前辅助线程已经决定传递通知了。因为 UI 线程决定不等待，直到辅助线程处理取消，所以 UI 线程有可能会继续从辅助线程接收通知。如果辅助线程使用 BeginInvoke 异步传递通知，则 UI 甚至有可能收到多个通知。UI 线程也可以始终按与“废弃”做法相同的方式处理通知 — 检查调用对象的标识并忽略它不再关心的操作通知。或者，在调用对象中

进行锁定并决不从辅助线程调用 BeginInvoke 以解决问题。但由于让 UI 线程在处理一个事件之前简单地对其进行检查以确定是否有用也比较简单，所以使用该方法碰到的问题可能会更少。

请查看“代码下载 ”（本文顶部的链接）中的 AsyncUtils，它是一个有用的基类，可为基于辅助线程的操作提供取消功能。图 9 显示了一个派生类，它实现了支持取消的递归目录搜索。这些类阐明了一些有趣的技术。它们都使用 C# 事件语法来提供通知。该基类将公开一些在操作成功完成、取消和抛出异常时出现的事件。派生类对此进行了扩充，它们将公开通知客户端搜索匹配、进度以及显示 当前正在搜索哪个目录的事件。这些事件始终在 UI 线程中传递。实际上，这些类并未限制为 Control 类 — 它们可以将事件传递给实现 ISynchronizeInvoke 接口的任何类。图 10 是一个示例 Windows 窗体应用程序，它为 Search 类提供一个用户界面。它允许取消搜索并显示进度和结果。

程序关闭

某些情况下，可以采用“启动后就不管”的异步操作，而不需要其他复杂要求来使操作可取消。然而，即使用户界面不要求取消，有可能还是需要实现这项功能以使程序可以彻底关闭。

当 应用程序退出时，如果由线程池创建的辅助线程还在运行，则这些线程会被终止。终止是简单粗暴的操作，因为关闭甚至会绕开任何还起作用的 Finally 块。如果异步操作执行的某些工作不应该以这种方式被打断，则必须确保在关闭之前这样的操作已经完成。此类操作可能包括对文件执行的写入操作，但由于突然中 断后，文件可能被破坏。

一种解决办法是创建自己的线程，而不用来自辅助线程池的线程，这样就自然会避开使用异步委托调用。这样，即使主线 程关闭，应用程序也会等到您的线程退出后才终止。System.Threading.Thread 类有一个 IsBackground 属性可以控制这种行为。它默认为 false，这种情况下，CLR 会等到所有非背景线程都退出后才正常终止应用程序。然而，这会带来另一个问题，因为应用程序挂起时间可能会比您预期的长。窗口都关闭了，但进程仍在运行。 这也许不是个问题。如果应用程序只是因为要进行一些清理工作才比正常情况挂起更长时间，那没问题。另一方面，如果应用程序在用户界面关闭后还挂起几分钟甚 至几小时，那就不可接受了。例如，如果它仍然保持某些文件打开，则可能妨碍用户稍后重启该应用程序。

最佳方法是，如果可能，通常应该编写自己的异步操作以便可以将其迅速取消，并在关闭应用程序之前等待所有未完成的操作完成。这意味着您可以继续使用异步委托，同时又能确保关闭操作彻底且及时。

错误处理

在 辅助线程中出现的错误一般可以通过触发 UI 线程中的事件来处理，这样错误处理方式就和完成及进程更新方式完全一样。因为很难在辅助线程上进行错误恢复，所以最简单的策略就是让所有错误都为致命错 误。错误恢复的最佳策略是使操作完全失败，并在 UI 线程上执行重试逻辑。如果需要用户干涉来修复造成错误的问题，简单的做法是给出恰当的提示。

AsyncUtils 类处理错误以及取消。如果操作抛出异常，该基类就会捕捉到，并通过 Failed 事件将异常传递给 UI。

小结

谨慎地使用多线程代码可以使 UI 在执行耗时较长的任务时不会停止响应，从而显著提高应用程序的反应速度。异步委托调用是将执行速度缓慢的代码从 UI 线程迁移出来，从而避免此类间歇性无响应的最简单方式。

Windows Forms Control 体系结构基本上是单线程，但它提供了实用程序以将来自辅助线程的调用封送返回至 UI 线程。处理来自辅助线程的通知(不管是成功、失败还是正在进行的指示）的最简单策略是，以对待来自常规控件的事件（如鼠标单击或键盘输入）的方式对待它 们。这样可以避免在 UI 代码中引入新的问题，同时通信的单向性也不容易导致出现死锁。

有时需要让 UI 向一个正在处理的操作发送消息。其中最常见的是取消一个操作。通过建立一个表示正在进行的调用的对象并维护由辅助线程定期检查的取消标记可实现这一目的。 如果用户界面线程需要等待取消被认可（因为用户需要知道工作已确实终止，或者要求彻底退出程序），实现起来会有些复杂，但所提供的示例代码中包含了一个将 所有复杂性封装在内的基类。派生类只需要执行一些必要的工作、周期性测试取消，以及要是因为取消请求而停止工作，就将结果通知基类。