

此站点使用 Cookie 进行分析、显示个性化内容和广告。继续浏览此站点，即表示您同意使用。 [了解详情](#)

Microsoft              

Microsoft Logo

Microsoft              






Gray Pipe

[Developer Network](#) [Developer Network](#) [Developer](#)

[MSDN 订阅](#)

[获取工具](#)

- [下载](#)
- [计划](#)
- [社区](#)
- [文档](#)

Microsoft              

search clear

[Parallel Processing and Concurrency](#) [Asynchronous Programming Patterns](#) [Task-based Asynchronous Pattern \(TAP\)](#)

[Task-based Asynchronous Pattern \(TAP\)](#) [Consuming the Task-based Asynchronous Pattern](#)

[打印](#) [导出 \(0\)](#)

[Consuming the Task-based Asynchronous Pattern](#)

[打印](#) [导出 \(0\)](#)

[Implementing the Task-based Asynchronous Pattern](#)

[Consuming the Task-based Asynchronous Pattern](#)

[Interop with Other Asynchronous Patterns and Types](#)

[TOC](#)

# Consuming the Task-based Asynchronous Pattern

.NET Framework (current version)

发布日期：2016年5月

当您使用基于任务的异步模式 (TAP) 时来使用异步操作，可以使用回调来实现等待，而不会阻塞。对于任务，这可以通过诸如 [Task.ContinueWith](#) 的方法来获得。基于语言的异步

支持通过使用提供与此 API 级别相同的支持，通过允许异步操作在规范控件流内等待来将回调隐藏起来。

## 挂起执行与请等待

---

从开始 .NET Framework 4.5，可以在 C# 使用 [await](#) 关键字和 [Await 运算符](#) 在 Visual Basic 异步等待 [Task](#) 和 [Task<TResult>](#) 对象。当您正在等待 [Task](#) 时，`await` 表达式的类型为 `void`。当您正在等待 [Task<TResult>](#) 时，`await` 表达式的类型为 `TResult`。`await` 表达式必须出现在异步方法的主体内。有关 .NET Framework 4.5 中 C# 和 Visual Basic 语言支持的更多信息，请参见 C# 和 Visual Basic 语言规范。

在封面下，等待功能通过持续性在任务上安装回调。此回调将在挂起点处继续异步方法。

在恢复异步方法时，如果等待操作已成功完成并且为 [Task<TResult>](#)，则将返回其 `TResult`。如果所等待的 [Task](#) 或 [Task<TResult>](#) 以 [Canceled](#) 关闭状态结束，则将引发 [OperationCanceledException](#) 异常。如果所等待的 [Task](#) 或 [Task<TResult>](#) 以 [Faulted](#) 状态关闭，则将引发导致其出错的异常。由于多个异常，`Task` 可能错误，但是只传播这些异常中的一个。但是，[Task.Exception](#) 属性返回包含所有错误的 [AggregateException](#) 异常。

如果同步上下文（[SynchronizationContext](#) 对象）与在挂起时与执行异步方法的线程相关联（例如，如果 [SynchronizationContext.Current](#) 属性不是 `null`），则该异步方法通过使用上下文的 [Post](#) 方法在同一上下文上继续。否则，它将依赖挂起时的当前任务计划程序（[TaskScheduler](#) 对象）。通常，这是默认的任务计划程序（[TaskScheduler.Default](#)），该任务计划程序把线程池作为目标。此任务计划程序确定等待的异步操作是否应在该操作完成时继续或是否应计划恢复。默认计划程序通常会允许延续在等待操作完成的任何线程上运行。

在调用时，异步方法同步执行该函数体，直到未被完成的可等待实例上的第一个等待表达式为止，此时调用返回到调用方。如果异步方法不返回 `void`，则将返回 `void`、[Task](#) 或 [Task<TResult>](#) 对象以表示正在进行的计算。在非 `void` 异步方法中，如果遇到返回语句或者达到方法主体的末尾，则任务将以 [RanToCompletion](#) 最终状态完成。如果未经处理的异常导致控件离开异步方法的主体，则任务在 [Faulted](#) 状态中结束。如果该异常是 [OperationCanceledException](#)，则任务在 [Canceled](#) 状态中结束。通过此方式，最终将发布结果或异常。

具有此行为的多个重要的变体。出于性能原因，如果任务在等待时已完成时，则不会生成控件且该函数将继续执行。此外，返回到原始上下文并不总是预期行为，并且可以进行更改；这将在下一节更详细地描述。

## 使用 `Yield` 和 `ConfigureAwait` 配置保存和还原

多个成员提供对异步方法执行的更多控制。例如，您可以使用 [Task.Yield](#) 方法来将让步点引入异步方法：

C#

```
public class Task : ...
{
    public static YieldAwaitable Yield();
    ...
}
```

这与异步发送或计划返回当前上下文等效。

C#

```
Task.Run(async delegate
{
    for(int i=0; i<1000000; i++)
    {
        await Task.Yield(); // fork the continuation into a separate work item
        ...
    }
});
```

您还可以使用更好的控制挂起的 [Task.ConfigureAwait](#) 方法并在异步方法中的恢复。如前所述，默认情况下，捕获了异步方法挂起时的当前上下文，该捕获的上下文用于在恢复时调用异步方法继续。在大多数情况下，这是您所需的确切行为。在其他某些情况下，您不关心有关延续内容，并且您可以通过避免这样的文章返回到原始上下文，从而获得更佳的性能。若要实现此功能，请使用 [Task.ConfigureAwait](#) 方法以通知不得在文本上捕获和使用的等待操作，但在完成等待异步操作的任何位置继续执行。

C#

```
await someTask.ConfigureAwait(continueOnCapturedContext:false);
```

## 取消异步操作

---

从开始 .NET Framework 4，支持取消的分接头方法提供接受一个取消标记至少有一个重载（[CancellationToken](#) 对象）。

通过取消标记源（[CancellationTokenSource](#) 对象）创建取消标记。源的 [Token](#) 属性将返回取消标记，该编辑将在调用源的 [Cancel](#) 方法时发出信号。例如，如果要下载单个网页，并且您希望能够取消该操作，则创建一 [CancellationTokenSource](#) 对象，将其标记传递给 TAP 方法，然后，当您准备好取消操作时，调用源的 [Cancel](#) 方法：

C#

```
var cts = new CancellationTokenSource();
string result = await DownloadStringAsync(url, cts.Token);
... // at some point later, potentially on another thread
cts.Cancel();
```

若要移除多个异步调用，您可以将同一标记传递到所有调用：

C#

```
var cts = new CancellationTokenSource();
    IList<string> results = await Task.WhenAll(from url in urls select
DownloadStringAsync(url, cts.Token));
    // at some point later, potentially on another thread
    ...
    cts.Cancel();
```

或者，您可以将同一标记传递给操作的选择性子集：

C#

```
var cts = new CancellationTokenSource();
    byte [] data = await DownloadDataAsync(url, cts.Token);
    await SaveToDiskAsync(outputPath, data, CancellationToken.None);
    ... // at some point later, potentially on another thread
    cts.Cancel();
```

取消请求可以从任何线程启动。

您可以传递 [CancellationToken.None](#) 值到接受取消标记指示绝不会取消的任何方法。这使得 [CancellationToken.CanBeCanceled](#) 属性返回 `false`，因此，被调用方法可以相应地

优化。出于测试目的，也可以使用接受布尔值的构造函数对实例化的预取消的标记进行传递，以指示在已取消或不可取消状态中是否应启动该标记。

此取消方法具有如下几个优势：

- 您可以传递同样的取消标记到异步和同步操作的任意编号。
- 相同的取消请求可能激增任何侦听器的数量。
- 异步 API 的开发人员在完全控制中移除是否可以请求，并在可能时生效。
- 使用 API 的代码可以有选择地确定取消请求将传播的异步调用。

## 监视进度

---

某些异步方法通过传递到异步方法的进度接口公开进度。例如，请考虑异步下载文本字符串的函数以及引发包括至今为止下载的百分比的进度更新的方法。此类方法可用于如下 Windows Presentation Foundation (WPF) 应用程序中：

C#

```
private async void btnDownload_Click(object sender, RoutedEventArgs e)
{
    btnDownload.IsEnabled = false;
    try
    {
        txtResult.Text = await DownloadStringAsync(txtUrl.Text,
            new Progress<int>(p => pbDownloadProgress.Value = p));
    }
    finally { btnDownload.IsEnabled = true; }
}
```

## 使用内置基于任务的组合器

---

[System.Threading.Tasks](#) 命名空间包括几种用于组成和处理任务的方法。

### Task.Run

[Task](#) 类包括若干 [Run](#) 方法，使您轻松地卸载工作作为线程池的 [Task](#) 或 [Task<TResult>](#)，例如：

C#

```

public async void button1_Click(object sender, EventArgs e)
{
    textBox1.Text = await Task.Run(() =>
    {
        // ... do compute-bound work here
        return answer;
    });
}

```

存在 [Run](#) 方法，如 [Task.Run\(Func<Task>\)](#) 重载，作为 [TaskFactory.StartNew](#) 方法的简略形式。但是，其他重载（例如 [Task.Run\(Func<Task>\)](#)）启用等待在被卸载的工作中使用，例如：

C#

```

public async void button1_Click(object sender, EventArgs e)
{
    pictureBox1.Image = await Task.Run(async () =>
    {
        using(Bitmap bmp1 = await DownloadFirstImageAsync())
        using(Bitmap bmp2 = await DownloadSecondImageAsync())
        return Mashup(bmp1, bmp2);
    });
}

```

此类重载逻辑上是等效于在任务并行库中结合 [Unwrap](#) 展开扩展方法使用 [TaskFactory.StartNew](#) 方法。

## Task.FromResult

对数据可能可用且只需要从任务返回的提升到 [Task<TResult>](#) 的方法返回的情况，可以使用 [FromResult<TResult>](#) 方法：

C#

```

public Task<int> GetValueAsync(string key)
{
    int cachedValue;
    return TryGetCachedValue(out cachedValue) ?
        Task.FromResult(cachedValue) :

```

```

        GetValueAsyncInternal();
    }

    private async Task<int> GetValueAsyncInternal(string key)
    {
        ...
    }

```

## Task.WhenAll

[WhenAll](#) 方法在为任务表示多个异步操作用于异步等待。该方法包含多个支持一组非泛型的任务或一组非规范的泛型任务（例如异步等待多个 void 返回的操作或异步等待多个值返回的方法，其中各个值的类型可能会不同），以及支持一组规范的泛型任务（如异步等待多个 TResult 返回方法）的重载。

假设您想要向多位客户发送电子邮件。您可以重叠发送消息，因此在发送下一个之前，您不等待消息完成。还可以查找发送操作完成的时间，以及是否发生了任何错误：

C#

```

IEnumerable<Task> asyncOps = from addr in addrs select SendMailAsync(addr);
await Task.WhenAll(asyncOps);

```

此代码不显式处理可能发生的异常，但是让异常从 [WhenAll](#) 传播到所生成任务上的 await 之外。若要处理异常，可以使用如下代码：

C#

```

IEnumerable<Task> asyncOps = from addr in addrs select SendMailAsync(addr);
try
{
    await Task.WhenAll(asyncOps);
}
catch (Exception exc)
{
    ...
}

```

在本例中，如果任何异步操作失败，那么所有异常将合并到将存储在从 [WhenAll](#) 方法中返回的 [Task](#) 中的 [AggregateException](#) 异常。但是，只有这些异常将会通过 `await` 关键字传播。如果想要检查所有异常，您可以重写先前的代码如下：

```
C#
Task [] asyncOps = (from addr in addrs select SendMailAsync(addr)).ToArray();
try
{
    await Task.WhenAll(asyncOps);
}
catch(Exception exc)
{
    foreach(Task faulted in asyncOps.Where(t => t.IsFaulted))
    {
        ... // work with faulted and faulted.Exception
    }
}
```

考虑从 Web 异步下载多个文件的另一个示例。在这种情况下，所有异步操作均具有同类的结果类型，因此，易于访问结果的：

```
C#
string [] pages = await Task.WhenAll(
    from url in urls select DownloadStringAsync(url));
```

您可以使用我们在前面的 `void` 返回方案中讨论的相同的异常处理技术：

```
C#
Task [] asyncOps =
    (from url in urls select DownloadStringAsync(url)).ToArray();
try
{
    string [] pages = await Task.WhenAll(asyncOps);
    ...
}
catch(Exception exc)
{
}
```



```
foreach(Task<string> faulted in asyncOps.Where(t => t.IsFaulted))
{
    ... // work with faulted and faulted.Exception
}
}
```

## Task.WhenAny

可使用 [WhenAny](#) 方法，异步等待其中一个异步操作视为完成。此方法提供了四个主要用例：

- 冗余：多次执行一个操作时选择首先完成的一个（例如：联系都将生成一个结果的多个股票行情 Web 服务，并选择完成最快的一个）。
- 交错：启动多个操作并等待所有这些操作完成，但是，在完成这些操作时对其进行处理。
- 遏制：允许其他操作以其他完成开始。此交替情景的扩展。
- 早期缓解办法：例如，任务 t1 表示的操作可在包含另一个任务 t2 的 [WhenAny](#) 任务中分组，并且您可以等待 [WhenAny](#) 任务。任务 t2 可以表示超时或者取消，或将导致 [WhenAny](#) 任务在 t1 完成之前完成的一些其他信号。

## 冗余

考虑您要制定有关是否购买股票的决策的情况。我们具有您信任的多个常用推荐 Web 服务，但基于每日负载，每种服务可以在不同时间以相当缓慢的速度结束。当所有操作完成时，您可以使用 [WhenAny](#) 方法接收通知。

C#

```
var recommendations = new List<Task<bool>>()
{
    GetBuyRecommendation1Async(symbol),
    GetBuyRecommendation2Async(symbol),
    GetBuyRecommendation3Async(symbol)
};
Task<bool> recommendation = await Task.WhenAny(recommendations);
if (await recommendation) BuyStock(symbol);
```

不同于 [WhenAll](#)（它返回成功完成了的所有任务的已打开结果），[WhenAny](#) 返回完成了的任务。如果任务失败，有必要知道它已失败，如果任务成功，有必要知道返回值与哪个任务相关联。因此，您需要访问返回任务的结果，或进一步等待，如本示例中所示。

与 [WhenAll](#) 一样，您能够调节异常。由于已接收回已完成任务，您可以等待返回的任务以便传播错误，并相应地 `try/catch` 它们，例如：

C#

```
Task<bool> [] recommendations = ...;
while(recommendations.Count > 0)
{
    Task<bool> recommendation = await Task.WhenAny(recommendations);
    try
    {
        if (await recommendation) BuyStock(symbol);
        break;
    }
    catch(WebException exc)
    {
        recommendations.Remove(recommendation);
    }
}
```

此外，即使第一个任务成功完成，后续任务仍可能会失败。此时，您已处理的若干选项异常：可以等待，直到所有启动的任务已完成，在能使用 [WhenAll](#) 方法的情况下，或可以决定所有异常是重要的，且必须记录。为此，当任务异步完成后，您可以使用延续任务来接收通知：

C#

```
foreach(Task recommendation in recommendations)
{
    var ignored = recommendation.ContinueWith(
        t => { if (t.IsFaulted) Log(t.Exception); });
}
```

或：

C#

```
foreach(Task recommendation in recommendations)
{
    var ignored = recommendation.ContinueWith(
        t => Log(t.Exception), TaskContinuationOptions.OnlyOnFaulted);
}
```

或者甚至：

```
private static async void LogCompletionIfFailed(IEnumerable<Task> tasks)
{
    foreach(var task in tasks)
    {
        try { await task; }
        catch(Exception exc) { Log(exc); }
    }
}
...
LogCompletionIfFailed(recommendations);
```

最后，您可能想要移除所有剩余操作：

C#

```
var cts = new CancellationTokenSource();
var recommendations = new List<Task<bool>>()
{
    GetBuyRecommendation1Async(symbol, cts.Token),
    GetBuyRecommendation2Async(symbol, cts.Token),
    GetBuyRecommendation3Async(symbol, cts.Token)
};
```

```
Task<bool> recommendation = await Task.WhenAny(recommendations);
cts.Cancel();
if (await recommendation) BuyStock(symbol);
```

## 交替

考虑从 Web 上下载图像并对每个图像进行一些处理（例如，将图像添加到 UI 控件）的情况。对 UI 线程，您不必按顺序执行处理，但是您希望尽可能同时下载图像。此外，您不希望保存图像添加到 UI，直到它们已下载 — 您要在其完成时添加：

C#

```
List<Task<Bitmap>> imageTasks =  
    (from imageUrl in urls select GetBitmapAsync(imageUrl)).ToList();  
while(imageTasks.Count > 0)  
{  
    try  
    {  
        Task<Bitmap> imageTask = await Task.WhenAny(imageTasks);  
        imageTasks.Remove(imageTask);  
  
        Bitmap image = await imageTask;  
        panel.AddImage(image);  
    }  
    catch{}  
}
```

还可以交替应用于涉及下载映像上的 [ThreadPool](#) 的计算密集型处理的方案；例如：

C#

```
List<Task<Bitmap>> imageTasks =  
    (from imageUrl in urls select GetBitmapAsync(imageUrl)  
        .ContinueWith(t => ConvertImage(t.Result)).ToList();  
while(imageTasks.Count > 0)  
{  
    try  
    {  
        Task<Bitmap> imageTask = await Task.WhenAny(imageTasks);  
        imageTasks.Remove(imageTask);
```

```

        Bitmap image = await imageTask;
        panel.AddImage(image);
    }
    catch {}
}

```

## 遏制

考虑交错的示例中，用户下载的图像过多，需要显示限制下载的情况除外，需要调节许多图像，例如：只有指定个数下载可以同时发生。为此，可以启动异步操作的子集。操作完成时，可以启动其他操作发生：

C#

```

const int CONCURRENCY_LEVEL = 15;
Uri [] urls = ...;
int nextIndex = 0;
var imageTasks = new List<Task<Bitmap>>();
while(nextIndex < CONCURRENCY_LEVEL && nextIndex < urls.Length)
{
    imageTasks.Add(GetBitmapAsync(urls[nextIndex]));
    nextIndex++;
}

while(imageTasks.Count > 0)
{
    try
    {
        Task<Bitmap> imageTask = await Task.WhenAny(imageTasks);
        imageTasks.Remove(imageTask);

        Bitmap image = await imageTask;
        panel.AddImage(image);
    }
    catch(Exception exc) { Log(exc); }

    if (nextIndex < urls.Length)

```

```

{
    imageTasks.Add(GetBitmapAsync(urls[nextIndex]));
    nextIndex++;
}
}

```

## 早期缓解办法

考虑异步等待该操作完成，同时响应用户的取消请求（例如，用户单击取消按钮）。下面的代码说明了这种方案：

C#

```

private CancellationTokenSource m_cts;

public void btnCancel_Click(object sender, EventArgs e)
{
    if (m_cts != null) m_cts.Cancel();
}

public async void btnRun_Click(object sender, EventArgs e)
{
    m_cts = new CancellationTokenSource();
    btnRun.Enabled = false;
    try
    {
        Task<Bitmap> imageDownload = GetBitmapAsync(txtUrl.Text);
        await UntilCompletionOrCancellation(imageDownload, m_cts.Token);
        if (imageDownload.IsCompleted)
        {
            Bitmap image = await imageDownload;
            panel.AddImage(image);
        }
        else imageDownload.ContinueWith(t => Log(t));
    }
    finally { btnRun.Enabled = true; }
}

```

```

private static async Task UntilCompletionOrCancellation(
    Task asyncOp, CancellationToken ct)
{
    var tcs = new TaskCompletionSource<bool>();
    using(ct.Register(() => tcs.TrySetResult(true)))
        await Task.WhenAny(asyncOp, tcs.Task);
    return asyncOp;
}

```

此实现重新启用用户界面，即决定放弃，但是不取消基础异步操作。另一种方法是在决定放弃时取消挂起的操作，但是不重新建立用户界面，知道操作自动完成，可能会由于取消请求较早结束。

C#

```

private CancellationTokenSource m_cts;

public async void btnRun_Click(object sender, EventArgs e)
{
    m_cts = new CancellationTokenSource();

    btnRun.Enabled = false;
    try
    {
        Task<Bitmap> imageDownload = GetBitmapAsync(txtUrl.Text, m_cts.Token);
        await UntilCompletionOrCancellation(imageDownload, m_cts.Token);
        Bitmap image = await imageDownload;
        panel.AddImage(image);
    }
    catch(OperationCanceledException) {}
    finally { btnRun.Enabled = true; }
}

```

早期财政援救的另一个示例包括使用 [WhenAny](#) 方法与 [Delay](#) 方法相结合，如下一节所述。

## Task.Delay

可以使用 [Delay](#) 方法用于引入暂停到一个异步方法的执行中。这对多种功能很有用，包括生成轮询循环和延迟用户输入预设时间段的处理。[Delay](#) 方法还可与 [WhenAny](#) 组合一起使用，以实现等待超时。

如果属于大型异步操作一部分的任务（例如，ASP.NET Web 服务）耗时太长而未能完成，则整个操作都可能受到影响，尤其是在该操作从未完成的情况下。出于此原因，能够对异步操作进行超时等待非常重要。同步 [Wait](#)、[Overload:System.Threading.Tasks.Task.

WaitAll](assetId:///Overload:System.Threading.Tasks.Task.WaitAll?

qualifyHint=False&autoUpgrade=True) 和 [Overload:System.Threading.Tasks.Task.

WaitAny](assetId:///Overload:System.Threading.Tasks.Task.WaitAny?

qualifyHint=False&autoUpgrade=True) 方法接受超时值，但是相应的

[ContinueWhenAll/WhenAny](#) 和前面提到的 [WhenAll/WhenAny](#) 方法不接受。相反，可以使用 [Delay](#) 和 [WhenAny](#) 可用于组合以实现超时。

例如，在您的 UI 应用程序中，假设您要下载图像并在下载图像时禁用 UI。但是，如果下载花费的时间过长，您需要重新启用 UI 并放弃该下载：

C#

```
public async void btnDownload_Click(object sender, EventArgs e)
{
    btnDownload.Enabled = false;
    try
    {
        Task<Bitmap> download = GetBitmapAsync(url);
        if (download == await Task.WhenAny(download, Task.Delay(3000)))
        {
            Bitmap bmp = await download;
            pictureBox.Image = bmp;
            status.Text = "Downloaded" ;
        }
        else
        {
            pictureBox.Image = null;
            status.Text = "Timed out" ;
            var ignored = download.ContinueWith(
```



```

        t => Trace( "Task finally completed" ));
    }
}
finally { btnDownload.Enabled = true; }
}

```

同样适用于多个下载，因为 [WhenAll](#) 返回任务：

C#

```

public async void btnDownload_Click(object sender, RoutedEventArgs e)
{
    btnDownload.Enabled = false;
    try
    {
        Task<Bitmap[]> downloads =
            Task.WhenAll(from url in urls select GetBitmapAsync(url));
        if (downloads == await Task.WhenAny(downloads, Task.Delay(3000)))
        {
            foreach(var bmp in downloads) panel.AddImage(bmp);
            status.Text = "Downloaded" ;
        }
        else
        {
            status.Text = "Timed out" ;
            downloads.ContinueWith(t => Log(t));
        }
    }
    finally { btnDownload.Enabled = true; }
}

```

## 生成基于任务的组合器

---

由于任务完全表示异步操作以及提供同步和异步功能以联接操作、检索其结果等的能力，可生成有用的组合器库，该库组成生成更大的模式的任务。如上节所述，.NET Framework 包括一些内置组合器，但是，您还可以生成您自己的。以下部分提供潜在连结符方法和类型的几个示例。

## RetryOnFault

在多数情况下，如果上次尝试失败，那么可能还要再尝试一次操作。对同步代码，您可能会生成帮助器方法来完成此操作，如下例中的 `RetryOnFault`：

C#

```
public static T RetryOnFault<T>(
    Func<T> function, int maxTries)
{
    for(int i=0; i<maxTries; i++)
    {
        try { return function(); }
        catch { if (i == maxTries-1) throw; }
    }
    return default(T);
}
```

可以生成一个几乎完全相同的帮助器方法，但是对于异步操作，可以使用 TAP 实现，从而返回任务。

C#

```
public static async Task<T> RetryOnFault<T>(
    Func<Task<T>> function, int maxTries)
{
    for(int i=0; i<maxTries; i++)
    {
        try { return await function().ConfigureAwait(false); }
        catch { if (i == maxTries-1) throw; }
    }
    return default(T);
}
```

然后，您可以使用此连结符编码重试到应用程序的逻辑中，例如：

C#

```
// Download the URL, trying up to three times in case of failure
string pageContents = await RetryOnFault(
    () => DownloadStringAsync(url), 3);
```

您可以进一步扩展 `RetryOnFault` 功能。例如，该函数可能接受在重试次数之间调用的另一 `Func<Task>`，以确定何时再次尝试操作，例如：

C#

```
public static async Task<T> RetryOnFault<T>(
    Func<Task<T>> function, int maxTries, Func<Task> retryWhen)
{
    for(int i=0; i<maxTries; i++)
    {
        try { return await function().ConfigureAwait(false); }
        catch { if (i == maxTries-1) throw; }
        await retryWhen().ConfigureAwait(false);
    }
    return default(T);
}
```

然后您可以在重新尝试操作前使用如下函数等待一秒：

C#

```
// Download the URL, trying up to three times in case of failure,
// and delaying for a second between retries
string pageContents = await RetryOnFault(
    () => DownloadStringAsync(url), 3, () => Task.Delay(1000));
```

## NeedOnlyOne

有时利用冗余来改进操作的延迟和成功机率。考虑均提供股票行情的多个 Web 服务，但是，在每天的很多时候，每个服务可以提供不同的质量级别和响应时间。为了应对这些波动情况，可以对所有 Web 服务发出请求，并且我们获取所有响应后，立即取消其余请求。可以实现 Helper 函数，从而更易于实现多个操作的这一常用模式、等待任一奥佐，然后取消剩余的操作：下面的示例中的 `NeedOnlyOne` 函演示了此情景：

C#

```
public static async Task<T> NeedOnlyOne(
    params Func<CancellationToken,Task<T>> [] functions)
{
    var cts = new CancellationTokenSource();
    var tasks = (from function in functions
        select function(cts.Token)).ToArray();
    var completed = await Task.WhenAny(tasks).ConfigureAwait(false);
    cts.Cancel();
    foreach(var task in tasks)
    {
        var ignored = task.ContinueWith(
            t => Log(t), TaskContinuationOptions.OnlyOnFaulted);
    }
    return completed;
}
```

然后，您可以按如下方式使用此函数：

C#

```
double currentPrice = await NeedOnlyOne(
    ct => GetCurrentPriceFromServer1Async( "msft" , ct),
    ct => GetCurrentPriceFromServer2Async( "msft" , ct),
    ct => GetCurrentPriceFromServer3Async( "msft" , ct));
```

## 交错操作

使用非常大任务的集合时，存在支持一个交替的方案中使用 [WhenAny](#) 方法的潜在的性能问题。每个对 [WhenAny](#) 的调用造成每个任务注册的继续操作。对于数量为 N 的任务，这会导致在交错操作生存期内创建 O (N<sup>2</sup>) 的持续量。如果您使用大型任务集，则您可以使用连结符（在以下示例中的 `Interleaved`）来解决性能问题：

C#

```
static IEnumerable<Task<T>> Interleaved<T>(IEnumerable<Task<T>> tasks)
{
    var inputTasks = tasks.ToList();
    var sources = (from _ in Enumerable.Range(0, inputTasks.Count)
        select new TaskCompletionSource<T>()).ToList();
```

```

int nextTaskIndex = -1;
foreach (var inputTask in inputTasks)
{
    inputTask.ContinueWith(completed =>
    {
        var source = sources[Interlocked.Increment(ref nextTaskIndex)];
        if (completed.IsFaulted)
            source.TrySetException(completed.Exception.InnerExceptions);
        else if (completed.IsCanceled)
            source.TrySetCanceled();
        else
            source.TrySetResult(completed.Result);
    }, CancellationToken.None,
    TaskContinuationOptions.ExecuteSynchronously,
    TaskScheduler.Default);
}
return from source in sources
    select source.Task;
}

```

可以使用连结符来处理任务的结果作为完成；例如：

C#

```

IEnumerable<Task<int>> tasks = ...;
foreach(var task in Interleaved(tasks))
{
    int result = await task;
    ...
}

```

## WhenAllOrFirstException

在某些散点/收集方案中，除非其中一个任务出错，这种情况下，您希望尽快出现异常而停止等待，否则您可能希望等待集内的所有任务。您可以通过连结符方法实现这一点，如

`WhenAllOrFirstException` 在以下示例中：

C#

```

public static Task<T[]> WhenAllOrFirstException<T>(IEnumerable<Task<T>>
tasks)
{
    var inputs = tasks.ToList();
    var ce = new CountdownEvent(inputs.Count);
    var tcs = new TaskCompletionSource<T[]>();

    Action<Task> onCompleted = (Task completed) =>
    {
        if (completed.IsFaulted)
            tcs.TrySetException(completed.Exception.InnerExceptions);
        if (ce.Signal() && !tcs.Task.IsCompleted)
            tcs.TrySetResult(inputs.Select(t => t.Result).ToArray());
    };

    foreach (var t in inputs) t.ContinueWith(onCompleted);
    return tcs.Task;
}

```

## 生成基于任务的数据结构

除了能生成基于任务的自定义组合器之外，如果在 [Task](#) 和 [Task<TResult>](#)（表示异步操作的结果和要加入它的必要异步操作）中具有数据结构，这会使其成为强类型，在该类型上将生成要在异步方案中使用的自定义数据结构。

### AsyncCache

任务的一个重要方面是它可分发给多个使用者，所有的人都可以等待，用它来注册延续，获取其结果或异常（在 [Task<TResult>](#) 的情况下），依此类推。这使得完全交互用于异步缓存基础结构 [Task](#) 和 [Task<TResult>](#)。这是在 [Task<TResult>](#) 顶部生成的功能强大的小型异步缓存的示例：

C#

```

public class AsyncCache<TKey, TValue>
{
    private readonly Func<TKey, Task<TValue>> _valueFactory;

```

```
private readonly ConcurrentDictionary<TKey, Lazy<Task<TValue>>> _map;
```

```
public AsyncCache(Func<TKey, Task<TValue>> valueFactory)
{
    if (valueFactory == null) throw new ArgumentNullException("loader");
    _valueFactory = valueFactory;
    _map = new ConcurrentDictionary<TKey, Lazy<Task<TValue>>>();
}
```

```
public Task<TValue> this[TKey key]
{
    get
    {
        if (key == null) throw new ArgumentNullException("key");
        return _map.GetOrAdd(key, toAdd =>
            new Lazy<Task<TValue>>(() => _valueFactory(toAdd))).Value;
    }
}
```

[AsyncCache<TKey, TValue>](#) 类接受作为委托给其构造函数采用 TKey 并返回 [Task<TResult>](#) 的函数。缓存中所有以前访问的值在内部字典中存储，并且 AsyncCache 确保每个键只有一任务生成，即使缓存被同时访问。

例如，您可以为下载的网页创建缓存：

C#

```
private AsyncCache<string,string> m_webPages =
    new AsyncCache<string,string>(DownloadStringAsync);
```

然后，不论何时您需要网页的内容，您可以以异步方法使用此缓存。 AsyncCache 选件类确保下载尽可能少的页尽可能并缓存结果。

C#

```
private async void btnDownload_Click(object sender, RoutedEventArgs e)
{
```

```

btnDownload.IsEnabled = false;
try
{
    txtContents.Text = await m_webPages["http://www.microsoft.com"];
}
finally { btnDownload.IsEnabled = true; }
}

```

## AsyncProducerConsumerCollection

还可以使用可构造异步活动之间协调的数据结构的任务。考虑一个经典的并行设计模式：制造者/使用者。在此模式中，制造者生成由使用者使用的数据，并且，制造者和使用者可以并行运行。例如，使用者处理项 1，之前由制造者的生成现在生成项 2。对于制造者-使用者模式，您不需要的某些数据结构存储制造者创建的工作，以便通知使用者可以将新数据并发现，可用时。

这是在任务顶部生成的单数据结构，可以将异步方法用作制造者和使用者：

C#

```

public class AsyncProducerConsumerCollection<T>
{
    private readonly Queue<T> m_collection = new Queue<T>();
    private readonly Queue<TaskCompletionSource<T>> m_waiting =
        new Queue<TaskCompletionSource<T>>();

    public void Add(T item)
    {
        TaskCompletionSource<T> tcs = null;
        lock (m_collection)
        {
            if (m_waiting.Count > 0) tcs = m_waiting.Dequeue();
            else m_collection.Enqueue(item);
        }
        if (tcs != null) tcs.TrySetResult(item);
    }

    public Task<T> Take()

```



```

{
    lock (m_collection)
    {
        if (m_collection.Count > 0)
        {
            return Task.FromResult(m_collection.Dequeue());
        }
        else
        {
            var tcs = new TaskCompletionSource<T>();
            m_waiting.Enqueue(tcs);
            return tcs.Task;
        }
    }
}

```

使用现有数据结构，您可以编写代码如下所示：

C#

```

private static AsyncProducerConsumerCollection<int> m_data = ...;
...
private static async Task ConsumerAsync()
{
    while(true)
    {
        int nextItem = await m_data.Take();
        ProcessNextItem(nextItem);
    }
}
...
private static void Produce(int data)
{
    m_data.Add(data);
}

```

```
}
```

[System.Threading.Tasks.Dataflow](#) 命名空间包括 [BufferBlock<T>](#) 类型，可以类似的方式加以使用，且无需生成自定义集合类型：

C#

```
private static BufferBlock<int> m_data = ...;

...

private static async Task ConsumerAsync()
{
    while(true)
    {
        int nextItem = await m_data.ReceiveAsync();
        ProcessNextItem(nextItem);
    }
}

...

private static void Produce(int data)
{
    m_data.Post(data);
}
```

说明
<a href="#">System.Threading.Tasks.Dataflow</a> 命名空间在 .NET Framework 4.5 通过 <b>NuGet</b> 提供。若要安装包含 <a href="#">System.Threading.Tasks.Dataflow</a> 命名空间的程序集， 请在此处

请在  
Visual  
Studio  
2012 中打  
开项目，  
从“项  
目”菜单  
中选  
择“**管理  
NuGet 程  
序包**”，  
并联机搜  
索  
Microsoft  
.Tpl.Dataf  
low 程序  
包。



System\_CAPS\_ICON\_note.jpg

## 请参阅

---

[Task-based Asynchronous Pattern \(TAP\)](#)

[Implementing the Task-based Asynchronous Pattern](#)

[Interop with Other Asynchronous Patterns and Types](#)

显示: 继承 保护

[打印共享](#)

本文内容

- [挂起执行与请等待](#)
- [取消异步操作](#)
- [监视进度](#)
- [使用内置基于任务的组合器](#)
- [生成基于任务的组合器](#)
- [生成基于任务的数据结构](#)
- [请参阅](#)

此页面有帮助吗？

## 开发人员中心

- [Windows](#)
- [Office](#)
- [Visual Studio](#)

- [Microsoft Azure](#)
- [更多...](#)

## 学习资源

- [Microsoft 虚拟学院](#)
- [第 9 频道](#)
- [MSDN 杂志](#)

## 社区

- [论坛](#)
- [博客](#)
- [Codeplex](#)

## 支持

- [自助支持](#)

## 计划

- [BizSpark \( 针对新创企业 \)](#)
- [Microsoft Imagine \(for students\)](#)

## [中国 \(简体中文\)](#)

- [新闻稿](#)
- [隐私& Cookie](#)
- [使用条款](#)
- [商标](#)