Async/Await

异步编程中的最佳做法

[**Stephen Cleary**](https://msdn.microsoft.com/zh-cn/magazine/ee532098.aspx?sdmr=StephenCleary&sdmi=authors)

近日来，涌现了许多关于 Microsoft .NET Framework 4.5 中新增了对 async 和 await 支持的信息。 本文旨在作为学习异步编程的“第二步”；我假设您已阅读过有关这一方面的至少一篇介绍性文章。 本文不提供任何新内容，Stack Overflow、MSDN 论坛和 async/await FAQ 这类在线资源提供了同样的建议。 本文只重点介绍一些淹没在文档海洋中的最佳做法。

本文中的最佳做法更大程度上是“指导原则”，而不是实际规则。 其中每个指导原则都有一些例外情况。 我将解释每个指导原则背后的原因，以便可以清楚地了解何时适用以及何时不适用。 **图 1** 中总结了这些指导原则；我将在以下各节中逐一讨论。

**图 1 异步编程指导原则总结**

|  |  |  |
| --- | --- | --- |
| “名称” | 说明 | 异常 |
| 避免 Async Void | 最好使用 async Task 方法而不是 async void 方法 | 事件处理程序 |
| 始终使用 Async | 不要混合阻塞式代码和异步代码 | 控制台 main 方法 |
| 配置上下文 | 尽可能使用 ConfigureAwait(false) | 需要上下文的方法 |

**避免 Async Void**

Async 方法有三种可能的返回类型： Task、Task<T> 和 void，但是 async 方法的固有返回类型只有 Task 和 Task<T>。 当从同步转换为异步代码时，任何返回类型 T 的方法都会成为返回 Task<T> 的 async 方法，任何返回 void 的方法都会成为返回 Task 的 async 方法。 下面的代码段演示了一个返回 void 的同步方法及其等效的异步方法：

C#

[复制](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_301818e6-2e6d-4a7e-b7a0-790b783781a4');" \o "复制到剪贴板。)

void MyMethod()

{

  // Do synchronous work.

Thread.Sleep(1000);

}

async Task MyMethodAsync()

{

  // Do asynchronous work.

await Task.Delay(1000);

}

返回 void 的 async 方法具有特定用途： 用于支持异步事件处理程序。 事件处理程序可以返回某些实际类型，但无法以相关语言正常工作；调用返回类型的事件处理程序非常困难，事件处理程序实际返回某些内容这一概念也没有太大意义。 事件处理程序本质上返回 void，因此 async 方法返回 void，以便可以使用异步事件处理程序。 但是，async void 方法的一些语义与 async Task 或 async Task<T> 方法的语义略有不同。

Async void 方法具有不同的错误处理语义。 当 async Task 或 async Task<T> 方法引发异常时，会捕获该异常并将其置于 Task 对象上。 对于 async void 方法，没有 Task 对象，因此 async void 方法引发的任何异常都会直接在 SynchronizationContext（在 async void 方法启动时处于活动状态）上引发。 **图 2** 演示本质上无法捕获从 async void 方法引发的异常。

**图 2 无法使用 Catch 捕获来自 Async Void 方法的异常**

C#

[复制](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_265bd054-337d-47fe-b150-12f9079a6c7e');)

private async void ThrowExceptionAsync()

{

  throw new InvalidOperationException();

}

public void AsyncVoidExceptions\_CannotBeCaughtByCatch()

{

  try

  {

    ThrowExceptionAsync();

  }

  catch (Exception)

  {

    // The exception is never caught here!

throw;

  }

}

可以通过对 GUI/ASP.NET 应用程序使用 AppDomain.UnhandledException 或类似的全部捕获事件观察到这些异常，但是使用这些事件进行常规异常处理会导致无法维护。

Async void 方法具有不同的组合语义。 返回 Task 或 Task<T> 的 async 方法可以使用 await、Task.WhenAny、Task.WhenAll 等方便地组合而成。 返回 void 的 async 方法未提供一种简单方式，用于向调用代码通知它们已完成。 启动几个 async void 方法不难，但是确定它们何时结束却不易。 Async void 方法会在启动和结束时通知 SynchronizationContext，但是对于常规应用程序代码而言，自定义 SynchronizationContext 是一种复杂的解决方案。

Async void 方法难以测试。 由于错误处理和组合方面的差异，因此调用 async void 方法的单元测试不易编写。 MSTest 异步测试支持仅适用于返回 Task 或 Task<T> 的 async 方法。 可以安装 SynchronizationContext 来检测所有 async void 方法都已完成的时间并收集所有异常，不过只需使 async void 方法改为返回 Task，这会简单得多。

显然，async void 方法与 async Task 方法相比具有几个缺点，但是这些方法在一种特定情况下十分有用： 异步事件处理程序。 语义方面的差异对于异步事件处理程序十分有意义。 它们会直接在 SynchronizationContext 上引发异常，这类似于同步事件处理程序的行为方式。 同步事件处理程序通常是私有的，因此无法组合或直接测试。 我喜欢采用的一个方法是尽量减少异步事件处理程序中的代码（例如，让它等待包含实际逻辑的 async Task 方法）。 下面的代码演示了这一方法，该方法通过将 async void 方法用于事件处理程序而不牺牲可测试性：

C#

[复制](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_c80c4057-a109-48e4-8963-541ccae75650');)

private async void button1\_Click(object sender, EventArgs e)

{

  await Button1ClickAsync();

}

public async Task Button1ClickAsync()

{

  // Do asynchronous work.

await Task.Delay(1000);

}

如果调用方不希望 async void 方法是异步的，则这些方法可能会造成严重影响。 当返回类型是 Task 时，调用方知道它在处理将来的操作；当返回类型是 void 时，调用方可能假设方法在返回时完成。 此问题可能会以许多意外方式出现。 在接口（或基类）上提供返回 void 的方法的 async 实现（或重写）通常是错误的。 某些事件也假设其处理程序在返回时完成。 一个不易察觉的陷阱是将 async lambda 传递到采用 Action 参数的方法；在这种情况下，async lambda 返回 void 并继承 async void 方法的所有问题。 一般而言，仅当 async lambda 转换为返回 Task 的委托类型（例如，Func<Task>）时，才应使用 async lambda。

总结这第一个指导原则便是，应首选 async Task 而不是 async void。 Async Task 方法更便于实现错误处理、可组合性和可测试性。 此指导原则的例外情况是异步事件处理程序，这类处理程序必须返回 void。 此例外情况包括逻辑上是事件处理程序的方法，即使它们字面上不是事件处理程序（例如 ICommand.Execute implementations）。

**始终使用 Async**

异步代码让我想起了一个故事，有个人提出世界是悬浮在太空中的，但是一个老妇人立即提出质疑，她声称世界位于一个巨大乌龟的背上。 当这个人问乌龟站在哪里时，老夫人回答：“很聪明，年轻人，下面是一连串的乌龟！”在将同步代码转换为异步代码时，您会发现，如果异步代码调用其他异步代码并且被其他异步代码所调用，则效果最好 — 一路向下（或者也可以说“向上”）。 其他人已注意到异步编程的传播行为，并将其称为“传染”或将其与僵尸病毒进行比较。 无论是乌龟还是僵尸，无可置疑的是，异步代码趋向于推动周围的代码也成为异步代码。 此行为是所有类型的异步编程中所固有的，而不仅仅是新 async/await 关键字。

“始终异步”表示，在未慎重考虑后果的情况下，不应混合使用同步和异步代码。 具体而言，通过调用 Task.Wait 或 Task.Result 在异步代码上进行阻塞通常很糟糕。 对于在异步编程方面“浅尝辄止”的程序员，这是个特别常见的问题，他们仅仅转换一小部分应用程序，并采用同步 API 包装它，以便代码更改与应用程序的其余部分隔离。 不幸的是，他们会遇到与死锁有关的问题。 在 MSDN 论坛、Stack Overflow 和电子邮件中回答了许多与异步相关的问题之后，我可以说，迄今为止，这是异步初学者在了解基础知识之后最常提问的问题： “为何我的部分异步代码死锁？”

**图 3** 演示一个简单示例，其中一个方法发生阻塞，等待 async 方法的结果。 此代码仅在控制台应用程序中工作良好，但是在从 GUI 或 ASP.NET 上下文调用时会死锁。 此行为可能会令人困惑，尤其是通过调试程序单步执行时，这意味着没完没了的等待。 在调用 Task.Wait 时，导致死锁的实际原因在调用堆栈中上移。

**图 3 在异步代码上阻塞时的常见死锁问题**

C#

[复制](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_96ff6cac-1067-4d5f-888b-ef3883c614c9');)

public static class DeadlockDemo

{

  private static async Task DelayAsync()

  {

    await Task.Delay(1000);

  }

  // This method causes a deadlock when called in a GUI or ASP.NET context.

public static void Test()

  {

    // Start the delay.

var delayTask = DelayAsync();

    // Wait for the delay to complete.

delayTask.Wait();

  }

}

这种死锁的根本原因是 await 处理上下文的方式。 默认情况下，当等待未完成的 Task 时，会捕获当前“上下文”，在 Task 完成时使用该上下文恢复方法的执行。 此“上下文”是当前 SynchronizationContext（除非它是 null，这种情况下则为当前 TaskScheduler）。 GUI 和 ASP.NET 应用程序具有 SynchronizationContext，它每次仅允许一个代码区块运行。 当 await 完成时，它会尝试在捕获的上下文中执行 async 方法的剩余部分。 但是该上下文已含有一个线程，该线程在（同步）等待 async 方法完成。 它们相互等待对方，从而导致死锁。

请注意，控制台应用程序不会形成这种死锁。 它们具有线程池 SynchronizationContext 而不是每次执行一个区块的 SynchronizationContext，因此当 await 完成时，它会在线程池线程上安排 async 方法的剩余部分。 该方法能够完成，并完成其返回任务，因此不存在死锁。 当程序员编写测试控制台程序，观察到部分异步代码按预期方式工作，然后将相同代码移动到 GUI 或 ASP.NET 应用程序中会发生死锁，此行为差异可能会令人困惑。

此问题的最佳解决方案是允许异步代码通过基本代码自然扩展。 如果采用此解决方案，则会看到异步代码扩展到其入口点（通常是事件处理程序或控制器操作）。 控制台应用程序不能完全采用此解决方案，因为 Main 方法不能是 async。 如果 Main 方法是 async，则可能会在完成之前返回，从而导致程序结束。 **图 4** 演示了指导原则的这一例外情况： 控制台应用程序的 Main 方法是代码可以在异步方法上阻塞为数不多的几种情况之一。

**图 4 Main 方法可以调用 Task.Wait 或 Task.Result**

C#

[复制](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_d1b46e44-7575-4a86-89da-b0477adf548f');)

class Program

{

  static void Main()

  {

    MainAsync().Wait();

  }

  static async Task MainAsync()

  {

    try

    {

      // Asynchronous implementation.

await Task.Delay(1000);

    }

    catch (Exception ex)

    {

      // Handle exceptions.

}

  }

}

允许异步代码通过基本代码扩展是最佳解决方案，但是这意味着需进行许多初始工作，该应用程序才能体现出异步代码的实际好处。 可通过几种方法逐渐将大量基本代码转换为异步代码，但是这超出了本文的范围。 在某些情况下，使用 Task.Wait 或 Task.Result 可能有助于进行部分转换，但是需要了解死锁问题以及错误处理问题。 我现在说明错误处理问题，并在本文后面演示如何避免死锁问题。

每个 Task 都会存储一个异常列表。 等待 Task 时，会重新引发第一个异常，因此可以捕获特定异常类型（如 InvalidOperationException）。 但是，在 Task 上使用 Task.Wait 或 Task.Result 同步阻塞时，所有异常都会用 AggregateException 包装后引发。 请再次参阅**图 4**。 MainAsync 中的 try/catch 会捕获特定异常类型，但是如果将 try/catch 置于 Main 中，则它会始终捕获 AggregateException。 当没有 AggregateException 时，错误处理要容易处理得多，因此我将“全局”try/catch 置于 MainAsync 中。

至此，我演示了两个与异步代码上阻塞有关的问题： 可能的死锁和更复杂的错误处理。 对于在 async 方法中使用阻塞代码，也有一个问题。 请考虑此简单示例：

C#

[复制](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_f5e126e5-7cb2-47e2-9dc4-c6306a3a77ef');)

public static class NotFullyAsynchronousDemo

{

  // This method synchronously blocks a thread.

public static async Task TestNotFullyAsync()

  {

    await Task.Yield();

    Thread.Sleep(5000);

  }

}

此方法不是完全异步的。 它会立即放弃，返回未完成的任务，但是当它恢复执行时，会同步阻塞线程正在运行的任何内容。 如果此方法是从 GUI 上下文调用，则它会阻塞 GUI 线程；如果是从 ASP.NET 请求上下文调用，则会阻塞当前 ASP.NET 请求线程。 如果异步代码不同步阻塞，则其工作效果最佳。 **图 5** 是将同步操作替换为异步替换的速查表。

**图 5 执行操作的“异步方式”**

|  |  |  |
| --- | --- | --- |
| 执行以下操作… | 替换以下方式… | 使用以下方式 |
| 检索后台任务的结果 | Task.Wait 或 Task.Result | await |
| 等待任何任务完成 | Task.WaitAny | await Task.WhenAny |
| 检索多个任务的结果 | Task.WaitAll | await Task.WhenAll |
| 等待一段时间 | Thread.Sleep | await Task.Delay |

总结这第二个指导原则便是，应避免混合使用异步代码和阻塞代码。 混合异步代码和阻塞代码可能会导致死锁、更复杂的错误处理及上下文线程的意外阻塞。 此指导原则的例外情况是控制台应用程序的 Main 方法，或是（如果是高级用户）管理部分异步的基本代码。

**配置上下文**

在本文前面，我简要说明了当等待未完成 Task 时默认情况下如何捕获“上下文”，以及此捕获的上下文用于恢复 async 方法的执行。 **图 3** 中的示例演示在上下文上的恢复执行如何与同步阻塞发生冲突从而导致死锁。 此上下文行为还可能会导致另一个问题 — 性能问题。 随着异步 GUI 应用程序在不断增长，可能会发现 async 方法的许多小部件都在使用 GUI 线程作为其上下文。 这可能会形成迟滞，因为会由于“成千上万的剪纸”而降低响应性。

若要缓解此问题，请尽可能等待 ConfigureAwait 的结果。 下面的代码段说明了默认上下文行为和 ConfigureAwait 的用法：

C#

[复制](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_9f2a3c96-e9bb-4e2c-9389-79ca60c7e634');)

async Task MyMethodAsync()

{

  // Code here runs in the original context.

await Task.Delay(1000);

  // Code here runs in the original context.

await Task.Delay(1000).ConfigureAwait(

    continueOnCapturedContext: false);

  // Code here runs without the original

  // context (in this case, on the thread pool).

}

通过使用 ConfigureAwait，可以实现少量并行性： 某些异步代码可以与 GUI 线程并行运行，而不是不断塞入零碎的工作。

除了性能之外，ConfigureAwait 还具有另一个重要方面： 它可以避免死锁。 再次考虑**图 3**；如果向 DelayAsync 中的代码行添加“ConfigureAwait(false)”，则可避免死锁。 此时，当等待完成时，它会尝试在线程池上下文中执行 async 方法的剩余部分。 该方法能够完成，并完成其返回任务，因此不存在死锁。 如果需要逐渐将应用程序从同步转换为异步，则此方法会特别有用。

如果可以在方法中的某处使用 ConfigureAwait，则建议对该方法中此后的每个 await 都使用它。 前面曾提到，如果等待未完成的 Task，则会捕获上下文；如果 Task 已完成，则不会捕获上下文。 在不同硬件和网络情况下，某些任务的完成速度可能比预期速度更快，需要谨慎处理在等待之前完成的返回任务。 **图 6** 显示了一个修改后的示例。

**图 6 处理在等待之前完成的返回任务**

C#

[复制](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_2ef258fa-6842-4788-bfff-d0c133ee5294');)

async Task MyMethodAsync()

{

  // Code here runs in the original context.

await Task.FromResult(1);

  // Code here runs in the original context.

await Task.FromResult(1).ConfigureAwait(continueOnCapturedContext: false);

  // Code here runs in the original context.

var random = new Random();

  int delay = random.Next(2); // Delay is either 0 or 1

  await Task.Delay(delay).ConfigureAwait(continueOnCapturedContext: false);

  // Code here might or might not run in the original context.

// The same is true when you await any Task

  // that might complete very quickly.

}

如果方法中在 await 之后具有需要上下文的代码，则不应使用 ConfigureAwait。 对于 GUI 应用程序，包括任何操作 GUI 元素、编写数据绑定属性或取决于特定于 GUI 的类型（如 Dispatcher/CoreDispatcher）的代码。 对于 ASP.NET 应用程序，这包括任何使用 HttpContext.Current 或构建 ASP.NET 响应的代码（包括控制器操作中的返回语句）。 **图 7** 演示 GUI 应用程序中的一个常见模式：让 async 事件处理程序在方法开始时禁用其控制，执行某些 await，然后在处理程序结束时重新启用其控制；因为这一点，事件处理程序不能放弃其上下文。

**图 7 让 async 事件处理程序禁用并重新启用其控制**

C#

[复制](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_11bd7d6a-786b-4235-abd8-222057e8ad3d');)

private async void button1\_Click(object sender, EventArgs e)

{

  button1.Enabled = false;

  try

  {

    // Can't use ConfigureAwait here ...

await Task.Delay(1000);

  }

  finally

  {

    // Because we need the context here.

button1.Enabled = true;

  }

}

每个 async 方法都具有自己的上下文，因此如果一个 async 方法调用另一个 async 方法，则其上下文是独立的。 **图 8** 演示的代码对**图 7** 进行了少量改动。

**图 8 每个 async 方法都具有自己的上下文**

C#

[复制](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_5639276e-52e8-4e40-8eb0-d19a1b2c3bcb');)

private async Task HandleClickAsync()

{

  // Can use ConfigureAwait here.

await Task.Delay(1000).ConfigureAwait(continueOnCapturedContext: false);

}

private async void button1\_Click(object sender, EventArgs e)

{

  button1.Enabled = false;

  try

  {

    // Can't use ConfigureAwait here.

await HandleClickAsync();

  }

  finally

  {

    // We are back on the original context for this method.

button1.Enabled = true;

  }

}

无上下文的代码可重用性更高。尝试在代码中隔离上下文相关代码与无上下文的代码，并尽可能减少上下文相关代码。在**图 8** 中，建议将事件处理程序的所有核心逻辑都置于一个可测试且无上下文的 async Task 方法中，仅在上下文相关事件处理程序中保留最少量的代码。即使是编写 ASP.NET 应用程序，如果存在一个可能与桌面应用程序共享的核心库，请考虑在库代码中使用 ConfigureAwait。

总结这第三个指导原则便是，应尽可能使用 Configure­Await。无上下文的代码对于 GUI 应用程序具有最佳性能，是一种可在使用部分 async 基本代码时避免死锁的方法。此指导原则的例外情况是需要上下文的方法。

**了解您的工具**

关于 async 和 await 有许多需要了解的内容，这自然会有点迷失方向。**图 9** 是常见问题的解决方案的快速参考。

**图 9 常见异步问题的解决方案**

|  |  |
| --- | --- |
| 问题 | 解决方案 |
| 创建任务以执行代码 | Task.Run 或 TaskFactory.StartNew（*不是* Task 构造函数或 Task.Start） |
| 为操作或事件创建任务包装 | TaskFactory.FromAsync 或 TaskCompletionSource<T> |
| 支持取消 | CancellationTokenSource 和 CancellationToken |
| 报告进度 | IProgress<T> 和 Progress<T> |
| 处理数据流 | TPL 数据流或被动扩展 |
| 同步对共享资源的访问 | SemaphoreSlim |
| 异步初始化资源 | AsyncLazy<T> |
| 异步就绪生产者/使用者结构 | TPL 数据流或 AsyncCollection<T> |

第一个问题是任务创建。显然，async 方法可以创建任务，这是最简单的选项。如果需要在线程池上运行代码，请使用 Task.Run。如果要为现有异步操作或事件创建任务包装，请使用 TaskCompletionSource<T>。下一个常见问题是如何处理取消和进度报告。基类库 (BCL) 包括专门用于解决这些问题的类型： CancellationTokenSource/CancellationToken 和 IProgress<T>/Progress<T>。异步代码应使用基于任务的异步模式（或称为 TAP，[msdn.microsoft.com/library/hh873175](http://msdn.microsoft.com/library/hh873175)），该模式详细说明了任务创建、取消和进度报告。

出现的另一个问题是如何处理异步数据流。任务很棒，但是只能返回一个对象并且只能完成一次。对于异步流，可以使用 TPL 数据流或被动扩展 (Rx)。TPL 数据流会创建类似于主角的“网格”。Rx 更加强大和高效，不过也更加难以学习。TPL 数据流和 Rx 都具有异步就绪方法，十分适用于异步代码。

仅仅因为代码是异步的，并不意味着就安全。共享资源仍需要受到保护，由于无法在锁中等待，因此这比较复杂。下面是一个异步代码示例，该代码如果执行两次，则可能会破坏共享状态，即使始终在同一个线程上运行也是如此：

int value;

Task<int> GetNextValueAsync(int current);

async Task UpdateValueAsync()

{

  value = await GetNextValueAsync(value);

}

C#

[复制](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_4984e8bc-f31c-4b4a-9919-3cd1bdb7ea33');)

问题在于，方法读取值并在等待时挂起自己，当方法恢复执行时，它假设值未更改。为了解决此问题，使用异步就绪 WaitAsync 重载扩展了 SemaphoreSlim 类。**图 10** 演示 SemaphoreSlim.WaitAsync。

**图 10 SemaphoreSlim 允许异步同步**

SemaphoreSlim mutex = new SemaphoreSlim(1);

int value;

Task<int> GetNextValueAsync(int current);

async Task UpdateValueAsync()

{

  await mutex.WaitAsync().ConfigureAwait(false);

  try

  {

    value = await GetNextValueAsync(value);

  }

  finally

  {

    mutex.Release();

  }

}

C#

[复制](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_a645efb0-5754-4909-8959-cefa804e2b2b');)

异步代码通常用于初始化随后会缓存并共享的资源。没有用于此用途的内置类型，但是 Stephen Toub 开发了 AsyncLazy<T>，其行为相当于 Task<T> 和 Lazy<T> 合二为一。该原始类型在其博客 ([bit.ly/dEN178](http://bit.ly/dEN178)) 上进行了介绍，并且在我的 AsyncEx 库 ([nitoasyncex.codeplex.com](http://nitoasyncex.codeplex.com/)) 中提供了更新版本。

最后，有时需要某些异步就绪数据结构。TPL 数据流提供了 BufferBlock<T>，其行为如同异步就绪生产者/使用者队列。而 AsyncEx 提供了 AsyncCollection<T>，这是异步版本的 BlockingCollection<T>。

我希望本文中的指导原则和指示能有所帮助。异步真的是非常棒的语言功能，现在正是开始使用它的好时机！

**Stephen Cleary** *生活在密歇根州北部，他是一位丈夫、父亲和程序员。 他已从事了 16 年的多线程和异步编程工作，自第一个 CTP 以来便在使用 Microsoft .NET Framework 中的异步支持。 他的主页（包括博客）位于* [*stephencleary.com*](http://stephencleary.com/)*。*

衷心感谢以下技术专家对本文的审阅： Stephen Toub  
Stephen Toub 在 Microsoft 的 Visual Studio 团队中工作。他专门从事与并行性和异步相关的领域。