

به نظر می‌رسد که حیوانی که روی جلد این کتاب دیده می‌شود، یعنی «زباد خرما» (common palm civet)، به موضوع کتاب مرتبط است. من تا زمانی که این تصویر را ندیده بودم، چیزی درباره‌ی این حیوان نمی‌دانستم، بنابراین آن را بررسی کردم. زبادهای خرما به عنوان آفت شناخته می‌شوند، زیرا در سقف‌خانه‌ها مدفوع می‌کنند و در بدترین زمان‌ها با یکدیگر سر و صدا و دعوا راه می‌اندازند. غدد بویایی مقعدی آنها ترشحات بدبو ایجاد می‌کند. این حیوان در دسته‌بندی گونه‌های در معرض خطر قرار ندارد و وضعیتش به عنوان «کمترین نگرانی» (Least Concern) تعیین شده است که ظاهراً به شکل مودبانه‌ای به این معنی است: «هر چقدر می‌خواهید از این حیوان‌ها بکشید، کسی دلتنگشان نمی‌شود.» زبادهای خرما علاقه‌ی زیادی به خوردن میوه‌های قهوه دارند و دانه‌های قهوه را از طریق دستگاه گوارش خود عبور می‌دهند. یکی از گران‌ترین قهوه‌های دنیا، یعنی «کپی لواک» (Kopi Luwak)، از دانه‌های قهوه‌ای که از مدفوع این حیوان استخراج می‌شود، تهیه می‌گردد. طبق گفته‌ی انجمن قهوه تخصصی آمریکا، «این قهوه طعم بدی دارد.»

این موضوع، زباد خرما را به نمادی مناسب برای توسعه‌ی هم‌زمانی (concurrency) و برنامه‌نویسی چندنخی (multithreading) تبدیل می‌کند. برای افراد ناآشنا، هم‌زمانی و چندنخی بودن اموری نامطلوب هستند. آن‌ها باعث می‌شوند که کدی که به خوبی عمل می‌کند، به بدترین شکل ممکن رفتار کند. شرایط رقابتی (race conditions) و مشکلات دیگر معمولاً باعث کرش‌های پرصدایی می‌شوند (و این همیشه به نظر می‌رسد که یا در تولید یا در زمان نمایش اتفاق می‌افتد). برخی حتی تا جایی پیش رفته‌اند که اعلام کرده‌اند «نخ‌ها بد هستند» و به طور کامل از هم‌زمانی اجتناب کرده‌اند. تعدادی توسعه‌دهنده هستند که علاقه‌مند به هم‌زمانی شده‌اند و بدون ترس از آن استفاده می‌کنند؛ اما بیشتر توسعه‌دهندگان در گذشته از هم‌زمانی آسیب دیده‌اند و آن تجربه طعم بدی را برایشان به جا گذاشته است.

با این حال، برای برنامه‌های مدرن، هم‌زمانی به سرعت به یک نیاز تبدیل شده است. امروزه کاربران انتظار رابط‌های کاملاً واکنش‌گرا دارند و برنامه‌های سرور باید به سطح بی‌سابقه‌ای مقیاس‌پذیر شوند. هم‌زمانی به هر دوی این گرایش‌ها پاسخ می‌دهد.

خوشبختانه، کتابخانه‌های مدرن زیادی وجود دارند که هم‌زمانی را بسیار آسان‌تر کرده‌اند! پردازش موازی و برنامه‌نویسی غیرهم‌زمان دیگر حوزه‌هایی انحصاری برای جادوگران نیستند. با افزایش سطح انتزاع، این کتابخانه‌ها توسعه برنامه‌های واکنش‌گرا و مقیاس‌پذیر را به هدفی واقع‌بینانه برای هر توسعه‌دهنده تبدیل کرده‌اند. اگر در گذشته

از هم‌زمانی آسیب دیده‌اید، زمانی که این کار بسیار دشوار بود، پیشنهاد می‌کنم با ابزارهای مدرن دوباره آن را امتحان کنید. احتمالاً نمی‌توانیم هرگز بگوییم که هم‌زمانی آسان است، اما قطعاً دیگر به سختی گذشته نیست!

چه کسانی باید این کتاب را بخوانند؟

این کتاب برای توسعه‌دهندگانی نوشته شده که می‌خواهند روش‌های مدرن هم‌زمانی (Concurrency) را بیاموزند. من فرض می‌کنم که شما تجربه قابل توجهی در استفاده از .NET دارید و با مفاهیمی مانند مجموعه‌های جنریک، enumerables و LINQ آشنایی دارید. اما انتظار ندارم که دانشی در زمینه برنامه‌نویسی چندنخی (Multithreading) یا برنامه‌نویسی غیرهم‌زمان (Asynchronous Programming) داشته باشید. اگر هم تجربه‌ای در این زمینه‌ها دارید، این کتاب همچنان برای شما مفید خواهد بود، زیرا به معرفی کتابخانه‌های جدیدتری می‌پردازد که هم‌ایمن‌تر و هم‌آسان‌تر هستند.

هم‌زمانی برای هر نوع برنامه‌ای مفید است. فرقی نمی‌کند که شما روی برنامه‌های دسکتاپ، موبایل یا سرور کار کنید؛ این روزها هم‌زمانی تقریباً یک نیاز اساسی در همه‌جا محسوب می‌شود. شما می‌توانید از راهکارهای این کتاب برای ایجاد رابط‌های کاربری واکنش‌گرا و سرورهای مقیاس‌پذیر استفاده کنید. هم‌زمانی حالا به یک امر فراگیر تبدیل شده و درک این تکنیک‌ها و کاربردهای آن‌ها از دانش ضروری برای هر توسعه‌دهنده حرفه‌ای است.

چرا این کتاب را نوشتم؟

اوایل دوران حرفه‌ای من، چندنخی را به سختی یاد گرفتم. بعد از چند سال، برنامه‌نویسی غیرهم‌زمان را هم به سختی یاد گرفتم. اگرچه این تجربیات ارزشمندی بودند، اما ای کاش آن زمان ابزارها و منابعی که امروز در دسترس است را در اختیار داشتم. به ویژه پشتیبانی از `await` و `async` در زبان‌های مدرن .NET یک موهبت بزرگ است.

با این حال، اگر به کتاب‌ها و منابعی که امروزه برای یادگیری هم‌زمانی موجود است نگاه کنید، تقریباً همه آنها با معرفی مفاهیم سطح پایین شروع می‌کنند. پوشش خوبی از نخ‌ها و ابزارهای همگام‌سازی (serialization primitives) وجود دارد و تکنیک‌های سطح بالاتر معمولاً در مراحل بعدی معرفی می‌شوند، اگر اصلاً پوشش داده شوند. به نظر من، این رویکرد به دو دلیل اشتباه است. اولاً، بسیاری از توسعه‌دهندگان هم‌زمانی، از جمله خود من، ابتدا مفاهیم سطح پایین را یاد گرفتند و با تکنیک‌های قدیمی و دشوار دست‌وپنجه نرم کردند. ثانیاً، بسیاری از کتاب‌ها قدیمی هستند و تکنیک‌های منسوخ را پوشش می‌دهند؛ حتی وقتی این کتاب‌ها به‌روزرسانی می‌شوند تا روش‌های جدیدتر را هم شامل کنند، متأسفانه آنها را در انتهای کتاب قرار می‌دهند.

فکر می‌کنم این روش اشتباه است. این کتاب تنها رویکردهای مدرن به هم‌زمانی را پوشش می‌دهد. این به این معنا نیست که مفاهیم سطح پایین بی‌ارزش هستند. وقتی من در دانشگاه برنامه‌نویسی می‌خواندم، یک کلاس داشتم که باید در آن یک واحد پردازش مرکزی مجازی (CPU) را از چندین گیت منطقی می‌ساختیم و یک کلاس دیگر هم بود که برنامه‌نویسی اسمبلی را آموزش می‌داد. در طول دوران حرفه‌ای‌ام، هرگز یک CPU طراحی نکرده‌ام و فقط چند ده خط اسمبلی نوشته‌ام، اما درک من از اصول پایه همچنان هر روز به من کمک می‌کند. با این حال، بهتر است ابتدا با انتزاعات سطح بالاتر شروع کنیم؛ کلاس اول برنامه‌نویسی من در زبان اسمبلی نبود!

این کتاب یک جایگاه خاص را پر می‌کند: مقدمه‌ای است بر (و مرجعی برای) هم‌زمانی با استفاده از رویکردهای مدرن. این کتاب انواع مختلف هم‌زمانی را پوشش می‌دهد، از جمله برنامه‌نویسی موازی (Parallel)، غیرهم‌زمان (Asynchronous) و واکنشی (Reactive). با این حال، هیچ‌یک از تکنیک‌های قدیمی را پوشش نمی‌دهد، چرا که آنها در کتاب‌ها و منابع آنلاین دیگر به اندازه کافی توضیح داده شده‌اند.

راهنمای استفاده از این کتاب

ساختار کتاب به شرح زیر است:

- **فصل 1** مقدمه‌ای است بر انواع مختلف هم‌زمانی که در این کتاب پوشش داده شده‌اند: پردازش موازی (parallel)، غیرهم‌زمان (asynchronous)، واکنشی (reactive) و جریان داده (dataflow).

- **فصل‌های 2 تا 6** مقدمه‌ای کامل‌تر بر این انواع هم‌زمانی ارائه می‌دهند.

- **فصل‌های باقی‌مانده** هر کدام به جنبه‌ای خاص از هم‌زمانی می‌پردازند و به عنوان مرجعی برای حل مشکلات رایج در این زمینه عمل می‌کنند.

توصیه می‌کنم که حتی اگر با برخی از انواع هم‌زمانی آشنایی دارید، فصل اول را بخوانید یا حداقل مرور کنید.

< در زمان انتشار این کتاب، NET Core 3.0 هنوز در نسخه بتا قرار دارد، بنابراین برخی جزئیات مربوط به جریان‌های غیرهم‌زمان (asynchronous streams) ممکن است تغییر کنند.

منابع آنلاین

این کتاب یک مقدمه گسترده بر چند نوع هم‌زمانی ارائه می‌دهد. من سعی کرده‌ام تکنیک‌هایی را که به نظر خودم و دیگران مفید بوده‌اند، در آن بگنجانم، اما این کتاب جامع نیست. منابع زیر، بهترین منابعی هستند که برای بررسی عمیق‌تر این تکنولوژی‌ها پیدا کرده‌ام:

- برای **برنامه‌نویسی موازی**، بهترین منبع کتاب *Parallel Programming with Microsoft .NET* از انتشارات مایکروسافت است که متن آن به صورت آنلاین در دسترس است. متأسفانه، این کتاب کمی قدیمی است. بخش مربوط به "futures" باید به جای خود از کد غیرهم‌زمان استفاده کند و بخش مربوط به "pipelines" باید از Channels یا TPL Dataflow استفاده کند.

- برای **برنامه‌نویسی غیرهم‌زمان**، مستندات MSDN بسیار خوب است، به خصوص بخش "Asynchronous Programming".

- مایکروسافت همچنین مستنداتی برای **TPL Dataflow** در دسترس قرار داده است.

- **System.Reactive (Rx)** کتابخانه‌ای است که به سرعت در حال گسترش است و به تکامل خود ادامه می‌دهد. به نظر من، بهترین منبع برای Rx در حال حاضر، کتاب الکترونیکی **Introduction to Rx** نوشته‌ی لی کمپبل است.

نحوه استفاده از علائم نگارشی در این کتاب

در این کتاب از علائم نگارشی زیر استفاده شده است:

- **ایتالیک**

برای نمایش اصطلاحات جدید، آدرس‌های اینترنتی (URLs)، آدرس‌های ایمیل، نام فایل‌ها و پسوند فایل‌ها استفاده می‌شود.

- **عرض ثابت**

برای فهرست‌های برنامه‌نویسی و همچنین در داخل پاراگراف‌ها برای اشاره به عناصر برنامه مانند نام متغیرها یا توابع، پایگاه‌های داده، انواع داده‌ها، متغیرهای محیطی، عبارات و کلمات کلیدی استفاده می‌شود.

- **عرض ثابت ضخیم**

برای نمایش دستورات یا متنی که باید دقیقاً توسط کاربر تایپ شوند، استفاده می‌شود.

- **عرض ثابت ایتالیک**

برای نمایش متنی که باید توسط کاربر مقداردهی شود یا مقادیری که بر اساس بافت تغییر می‌کنند، استفاده می‌شود.

استفاده از مثال‌های کد

مطالب تکمیلی (مثال‌های کد، تمرین‌ها و غیره) برای دانلود در آدرس زیر موجود است:

[\[https://oreil.ly/concur-c-ckbk2\]](https://oreil.ly/concur-c-ckbk2)(<https://oreil.ly/concur-c-ckbk2>)

این کتاب به شما کمک می‌کند کار خود را انجام دهید. به طور کلی، اگر مثال‌های کدی همراه با این کتاب ارائه شده باشد، می‌توانید از آن‌ها در برنامه‌ها و مستندات خود استفاده کنید. لازم نیست برای استفاده از این کدها با ما تماس بگیرید، مگر اینکه بخواهید بخش بزرگی از کد را بازتولید کنید. به عنوان مثال، نوشتن یک برنامه که از چند بخش کد این کتاب استفاده می‌کند نیازی به اجازه ندارد. فروش یا توزیع CD-ROM حاوی مثال‌های کتاب‌های O'Reilly نیاز به اجازه دارد. پاسخ به سوالی با استناد به این کتاب و نقل قول از کدهای مثال، نیازی به اجازه ندارد. ادغام مقدار زیادی از کدهای مثال این کتاب در مستندات محصول شما نیاز به اجازه دارد.

اگر احساس می‌کنید استفاده شما از مثال‌های کد خارج از محدوده استفاده منصفانه یا مجوز داده شده است، می‌توانید با ما از طریق ایمیل permissions@oreilly.com تماس بگیرید.

آموزش آنلاین O'Reilly

برای تقریباً ۴۰ سال، O'Reilly Media آموزش‌های فنی و کسب‌وکار، دانش و بینش را ارائه کرده تا به شرکت‌ها در موفقیت کمک کند.

شبکه منحصربه‌فرد ما از کارشناسان و نوآوران، دانش و تجربه خود را از طریق کتاب‌ها، مقالات، کنفرانس‌ها و پلتفرم آموزش آنلاین ما به اشتراک می‌گذارند. پلتفرم آموزش آنلاین O'Reilly به شما دسترسی بر حسب تقاضا به دوره‌های آموزش زنده، مسیرهای یادگیری عمیق، محیط‌های کدنویسی تعاملی و مجموعه عظیمی از متون و ویدیوها از O'Reilly

و بیش از ۲۰۰ ناشر دیگر می‌دهد. برای اطلاعات بیشتر به <http://oreilly.com> [\[http://oreilly.com\]](http://oreilly.com) مراجعه کنید.

نحوه تماس با ما

لطفاً نظرات و سوالات خود را درباره این کتاب به ناشر زیر ارسال کنید:

****O'Reilly Media, Inc****

Gravenstein Highway North 1005

Sebastopol, CA 95472

تلفن: 800-998-9938 (در ایالات متحده یا کانادا)

تلفن: 0515-829-707 (بین‌المللی یا محلی)

فکس: 0104-829-707

ما یک صفحه وب برای این کتاب داریم که در آن اشتباهات، مثال‌ها و هر اطلاعات اضافی را لیست می‌کنیم. شما می‌توانید به این صفحه در <https://oreil.ly/concur-c-ckbk2> [\[https://oreil.ly/concur-c-ckbk2\]](https://oreil.ly/concur-c-ckbk2) دسترسی پیدا کنید.

برای ارسال نظرات یا پرسش‌های فنی درباره این کتاب، ایمیل خود را به ****bookquestions@oreilly.com**** ارسال کنید.

برای کسب اطلاعات بیشتر در مورد کتاب‌ها، دوره‌ها، کنفرانس‌ها و اخبار ما، به وب‌سایت ما به آدرس <http://www.oreilly.com> [\[http://www.oreilly.com\]](http://www.oreilly.com) مراجعه کنید.

ما را در فیسبوک پیدا کنید: (<http://facebook.com/oreilly>)(<http://facebook.com/oreilly>)

ما را در توییتر دنبال کنید: (<http://twitter.com/oreillymedia>)(<http://twitter.com/oreillymedia>)

ما را در یوتیوب تماشا کنید:

(<http://www.youtube.com/oreillymedia>)(<http://www.youtube.com/oreillymedia>)

قدردانی‌ها

این کتاب بدون کمک بسیاری از افراد وجود نمی‌داشت!

اول از همه، می‌خواهم از پروردگار و نجات‌دهنده‌ام، عیسی مسیح، قدردانی کنم. تبدیل شدن به یک مسیحی مهم‌ترین تصمیم زندگی من بود! اگر به اطلاعات بیشتری در این زمینه نیاز دارید، می‌توانید از طریق صفحه وب شخصی‌ام با من تماس بگیرید.

دوم، از خانواده‌ام تشکر می‌کنم که به من اجازه دادند زمان زیادی را با آن‌ها کنار نگذارم. وقتی شروع به نوشتن کردم، برخی از دوستان نویسنده‌ام به من گفتند: "برای یک سال خداحافظی با خانواده‌ات را بگو!" و من فکر کردم که آن‌ها شوخی می‌کنند. همسر، مندی، و فرزندانمان، SD و اما، در زمانی که روزهای طولانی را در محل کار می‌گذراندم و به نوشتن در شب‌ها و آخر هفته‌ها می‌پرداختم، بسیار درک‌کننده بودند. بسیار سپاسگزارم. دوستان دارم!

البته، این کتاب به این خوبی نمی‌بود اگر ویرایشگران و بازیینی‌کنندگان فنی ما نبودند: استیون توب، پتر آندرکا ("svick")، نیک پالدینو ("casper-One")، لی کمپیل و پدرو فلیکس. بنابراین اگر اشتباهاتی وجود دارد، کاملاً تقصیر آن‌هاست. شوخی می‌کنم! ورودی آن‌ها در شکل‌دهی (و اصلاح) محتوای کتاب بی‌نهایت ارزشمند بوده و هر اشتباه باقی‌مانده به طور قطع مربوط به من است. تشکر ویژه‌ای از استیون توب دارم که به من `Boolean Argument` (فرمول 14.5) و همچنین موضوعات دیگر مربوط به `async` را آموزش داد؛ و لی کمپیل که به من کمک کرد تا `System.Reactive` را یاد بگیرم و کد مشاهداتی‌ام را بیشتر به زبان ایدئوماتیک تبدیل کنم.

در نهایت، می‌خواهم از افرادی که این تکنیک‌ها را از آن‌ها یاد گرفته‌ام تشکر کنم: استیون توب، لوسیان ویسچیک، توماس لوسک، لی کمپیل، اعضای Stack Overflow و انجمن‌های MSDN و شرکت‌کنندگان در کنفرانس‌های نرم‌افزاری در ایالت میشیگان. از اینکه بخشی از جامعه توسعه نرم‌افزار هستم، قدردانی می‌کنم و اگر این کتاب کمکی می‌کند، به خاطر افرادی است که قبلاً راه را نشان داده‌اند. از همه شما متشکرم!

همزمانی: یک مرور کلی

همزمانی یکی از جنبه‌های کلیدی نرم‌افزار زیباست. برای دهه‌ها، همزمانی امکان‌پذیر بود اما دستیابی به آن دشوار بود. نرم‌افزارهای همزمان نوشتن، اشکال‌زدایی و نگهداری دشواری داشتند. در نتیجه، بسیاری از توسعه‌دهندگان مسیر آسان‌تری را انتخاب کردند و از همزمانی اجتناب کردند. با کتابخانه‌ها و ویژگی‌های زبانی که برای برنامه‌های مدرن NET در دسترس است، همزمانی اکنون بسیار آسان‌تر شده است. مایکروسافت پیشگام در کاهش قابل توجه بار همزمانی بوده است. قبلاً، برنامه‌نویسی همزمان حوزه‌ی کارشناسان بود؛ اما این روزها هر توسعه‌دهنده‌ای می‌تواند (و باید) همزمانی را بپذیرد.

مقدمه‌ای بر همزمانی

قبل از ادامه، می‌خواهم برخی از اصطلاحات را که در طول این کتاب استفاده می‌کنم، روشن کنم. این‌ها تعاریف خود من هستند که برای تفکیک تکنیک‌های مختلف برنامه‌نویسی به طور مداوم استفاده می‌کنم. بیا با همزمانی شروع کنیم.

همزمانی

****انجام چندین کار به صورت همزمان.****

امیدوارم واضح باشد که همزمانی چگونه می تواند مفید باشد. برنامه های کاربر نهایی از همزمانی برای پاسخ به ورودی کاربر در حین نوشتن به یک پایگاه داده استفاده می کنند. برنامه های سروری از همزمانی برای پاسخ به یک درخواست دوم در حالی که درخواست اول را تمام می کنند، استفاده می کنند. شما به همزمانی نیاز دارید هر زمان که بخواهید یک برنامه یک کار را انجام دهد در حالی که در حال انجام کار دیگری است. تقریباً هر برنامه نرم افزاری در جهان می تواند از همزمانی بهره مند شود.

بسیاری از توسعه دهندگان با شنیدن اصطلاح "همزمانی" به طور خودکار به "چندریسمانی" فکر می کنند. می خواهم تمایزی بین این دو قائل شوم.

چندریسمانی

****شکلی از همزمانی که از چندین رشته اجرایی استفاده می کند.****

چندریسمانی به معنای واقعی استفاده از چندین رشته است. همان طور که در بسیاری از دستورالعمل های این کتاب نشان داده شده است، چندریسمانی یک شکل از همزمانی است، اما قطعاً تنها شکل آن نیست. در واقع، استفاده مستقیم از انواع رشته های سطح پایین تقریباً هیچ هدفی در یک برنامه مدرن ندارد؛ انتزاعات سطح بالاتر از چندریسمانی قدیمی قدرتمندتر و کارآمدتر هستند. به همین دلیل، پوشش من از تکنیک های قدیمی را به حداقل می رسانم. هیچ یک از دستورالعمل های چندریسمانی در این کتاب از نوع های `Thread` یا `BackgroundWorker` استفاده نمی کنند؛ آن ها با جایگزین های برتر تعویض شده اند.

به محض اینکه `new Thread` را تایپ کنید، تمام است؛ پروژه شما قبلاً کد قدیمی دارد.

اما این گونه تصور نکنید که چندریسمانی مرده است! چندریسمانی در ****استخر رشته‌ها**** ادامه دارد، که مکان مفیدی برای صف کردن کار است و به طور خودکار بر اساس تقاضا خود را تنظیم می‌کند. به نوبه خود، استخر رشته‌ها شکل مهم دیگری از همزمانی را ممکن می‌سازد: ****پردازش موازی****.

پردازش موازی

****انجام کارهای زیاد با تقسیم آن‌ها بین چندین رشته که به صورت همزمان اجرا می‌شوند.****

پردازش موازی (یا برنامه‌نویسی موازی) از چندریسمانی برای حداکثر استفاده از چندین هسته پردازنده استفاده می‌کند. پردازنده‌های مدرن دارای چندین هسته هستند، و اگر کار زیادی برای انجام وجود داشته باشد، منطقی نیست که یک هسته همه کار را انجام دهد در حالی که هسته‌های دیگر بیکار نشسته‌اند. پردازش موازی کار را بین چندین رشته تقسیم می‌کند که می‌توانند به طور مستقل بر روی هسته‌های مختلف اجرا شوند.

پردازش موازی یک نوع از چندریسمانی است و چندریسمانی یک نوع از همزمانی است. نوع دیگری از همزمانی که در برنامه‌های مدرن مهم است اما برای بسیاری از توسعه‌دهندگان آشنا نیست، ****برنامه‌نویسی ناهمزمان**** است.

برنامه‌نویسی ناهمزمان

****شکلی از همزمانی که از آینده‌ها یا بازگشت‌های تماس برای اجتناب از رشته‌های غیرضروری استفاده می‌کند.****

یک ****آینده**** (یا ****وعده****) نوعی است که نمایانگر یک عمل است که در آینده کامل خواهد شد. برخی از انواع آینده‌های مدرن در NET شامل `Task`` و `Task<TResult>` هستند. API‌های ناهمزمان قدیمی‌تر به جای آینده‌ها از بازگشت‌های تماس یا رویدادها استفاده می‌کنند. برنامه‌نویسی ناهمزمان بر اساس ایده‌ی یک عملیات ناهمزمان متمرکز است: یک عملیاتی که آغاز می‌شود و بعداً کامل خواهد شد. در حالی که این عمل در حال پیشرفت است، رشته‌ی اصلی مسدود نمی‌شود؛ رشته‌ای که عمل را شروع می‌کند، آزاد است تا کارهای دیگری انجام دهد. وقتی عمل کامل شد، به آینده خود اطلاع می‌دهد یا بازگشت تماس یا رویداد خود را فعال می‌کند تا به برنامه بگوید که عملیات تمام شده است.

برنامه‌نویسی ناهمزمان یک شکل قدرتمند از همزمانی است، اما تا به حال به کد بسیار پیچیده‌ای نیاز داشت. پشتیبانی از `await`` و `async`` در زبان‌های مدرن برنامه‌نویسی، برنامه‌نویسی ناهمزمان را تقریباً به آسانی برنامه‌نویسی همزمان (غیر همزمان) کرده است.

شکل دیگری از همزمانی ****برنامه‌نویسی واکنشی**** است. برنامه‌نویسی ناهمزمان به این معنی است که برنامه عملیاتی را شروع می‌کند که در یک زمان بعدی کامل می‌شود. برنامه‌نویسی واکنشی به شدت با برنامه‌نویسی ناهمزمان مرتبط است، اما بر اساس رویدادهای ناهمزمان ساخته شده است به جای عملیات ناهمزمان. رویدادهای ناهمزمان ممکن است واقعاً "شروع" نداشته باشند، ممکن است هر زمان اتفاق بیفتند و ممکن است چندین بار ایجاد شوند. یک مثال ورودی کاربر است.

برنامه‌نویسی واکنشی

****یک سبک برنامه‌نویسی اعلامی که در آن برنامه به رویدادها واکنش نشان می‌دهد.****

اگر برنامه‌ای را به عنوان یک ماشین حالت عظیم در نظر بگیرید، رفتار برنامه می‌تواند به عنوان واکنش به یک سری رویدادها توصیف شود که در هر رویداد وضعیت خود را به‌روزرسانی می‌کند. این تا این حد انتزاعی یا نظری نیست که به

نظر می‌رسد؛ فریم‌ورک‌های مدرن این رویکرد را در برنامه‌های واقعی بسیار مفید می‌کنند. برنامه‌نویسی واکنشی لزوماً همزمان نیست، اما به شدت با همزمانی مرتبط است، بنابراین این کتاب مبانی آن را پوشش می‌دهد.

معمولاً از ترکیبی از تکنیک‌ها هنگام نوشتن یک برنامه همزمان استفاده می‌شود. اکثر برنامه‌ها حداقل از چندریسمانی (از طریق استخر رشته‌ها) و برنامه‌نویسی ناهمزمان استفاده می‌کنند. از ترکیب و تطابق همه اشکال مختلف همزمانی دریغ نکنید، استفاده کنید.

مقدمه‌ای بر برنامه‌نویسی ناهمزمان

برنامه‌نویسی ناهمزمان دو مزیت اصلی دارد. اولین مزیت برای برنامه‌های GUI (رابط کاربری گرافیکی) کاربر نهایی است: برنامه‌نویسی ناهمزمان امکان پاسخگویی را فراهم می‌کند. همه ما از برنامه‌ای استفاده کرده‌ایم که در حین کار، به طور موقت قفل می‌شود؛ یک برنامه ناهمزمان می‌تواند در حین کار به ورودی کاربر پاسخگو باشد. دومین مزیت برای برنامه‌های سمت سرور است: برنامه‌نویسی ناهمزمان امکان مقیاس‌پذیری را فراهم می‌کند. یک برنامه سروری می‌تواند به‌طور نسبی با استفاده از استخر رشته مقیاس‌پذیر باشد، اما یک برنامه سرور ناهمزمان معمولاً می‌تواند به طور چشمگیری بهتر از آن مقیاس‌پذیر باشد.

هر دو مزیت برنامه‌نویسی ناهمزمان از یک جنبه زیرساختی ناشی می‌شود: برنامه‌نویسی ناهمزمان یک رشته را آزاد می‌کند. برای برنامه‌های GUI، برنامه‌نویسی ناهمزمان رشته UI را آزاد می‌کند؛ این اجازه می‌دهد تا برنامه GUI به ورودی کاربر پاسخگو بماند. برای برنامه‌های سرور، برنامه‌نویسی ناهمزمان رشته‌های درخواست را آزاد می‌کند؛ این اجازه می‌دهد تا سرور از رشته‌های خود برای خدمت به درخواست‌های بیشتری استفاده کند.

استفاده از کلمات کلیدی `await` و `async`

برنامه‌های مدرن ناهمزمان در .NET از دو کلمه کلیدی استفاده می‌کنند `await` و `async` کلمه کلیدی `async` به یک اعلان متد اضافه می‌شود و دو هدف را انجام می‌دهد: این کلمه کلیدی امکان استفاده از کلمه کلیدی `await` را در آن متد فراهم می‌کند و به کامپایلر علامت می‌دهد که یک ماشین حالت برای آن متد ایجاد کند،

مشابه نحوه عملکرد `yield return`. یک متد `async` ممکن است `Task<TResult>` را برگرداند اگر مقداری را برگرداند، `Task` اگر مقداری را نبرگرداند، یا هر نوع "شبیه به تسک" دیگری مانند `ValueTask` علاوه بر این، یک متد `async` ممکن است `IAsyncEnumerable<T>` یا `IAsyncEnumerator<T>` را برگرداند اگر چندین مقدار را

در یک شمارش برگرداند. انواع شبیه به تسک نمایانگر آینده‌ها هستند؛ آن‌ها می‌توانند به کد فراخوانی اطلاع دهند که متد `async` تکمیل شده است.

اجتناب از استفاده از `async void`

اجتناب از `async void`! ممکن است یک متد `async` مقدار `void` برگرداند، اما تنها باید این کار را زمانی انجام دهید که یک هندلر رویداد `async` می‌نویسید. یک متد `async` عادی بدون مقدار بازگشتی باید `Task` را برگرداند، نه `void`. با این پس‌زمینه، بیایید نگاهی سریع به یک مثال بیندازیم:

```
async Task DoSomethingAsync()
{
    int value = 13;

    // به صورت ناهمزمان 1 ثانیه صبر کنید.
    await Task.Delay(TimeSpan.FromSeconds(1));

    value *= 2;

    // به صورت ناهمزمان 1 ثانیه صبر کنید.
    await Task.Delay(TimeSpan.FromSeconds(1));

    Trace.WriteLine(value);
}
```

یک متد `async` مانند هر متد دیگری به صورت همزمان شروع به اجرا می‌کند. در یک متد `async`، کلمه کلیدی `await` یک انتظار ناهمزمان بر روی آرگومان خود انجام می‌دهد. ابتدا، بررسی می‌کند که آیا عملیات قبلاً کامل شده است؛ اگر این گونه باشد، ادامه به اجرا (به صورت همزمان) می‌دهد. در غیر این صورت، متد `async` را متوقف کرده و یک تسک ناتمام را برمی‌گرداند. وقتی آن عملیات بعداً به پایان می‌رسد، متد `async` دوباره به اجرا ادامه خواهد داد. می‌توانید یک متد `async` را به عنوان داشتن چندین بخش همزمان تصور کنید که توسط بیانیه‌های `await` جدا شده‌اند. اولین بخش همزمان بر روی هر رشته‌ای که متد را فراخوانی می‌کند اجرا می‌شود، اما بخش‌های همزمان دیگر کجا اجرا می‌شوند؟ پاسخ کمی پیچیده است.

زمینه‌ها و رفتار await

وقتی که شما بر روی یک تسک انتظار می‌کشید (متداول‌ترین سناریو)، یک زمینه در زمانیکه await تصمیم می‌گیرد متد را متوقف کند، گرفته می‌شود. این زمینه معمولاً SynchronizationContext فعلی است مگر اینکه خالی باشد، در این صورت زمینه، TaskScheduler فعلی است. متد دوباره درون آن زمینه گرفته شده ادامه می‌یابد. معمولاً، این زمینه، زمینه UI است) اگر در رشته UI باشید (یا زمینه استخر رشته (در بیشتر موقعیت‌ها). اگر شما یک برنامه ASP.NET کلاسیک) پیش از (Core داشته باشید، آن زمینه همچنین می‌تواند یک زمینه درخواست ASP.NET باشد ASP.NET . Core به‌جای یک زمینه درخواست خاص، از زمینه استخر رشته استفاده می‌کند.

بنابراین، در کد قبلی، تمام بخش‌های همزمان سعی خواهند کرد که در زمینه اصلی دوباره ادامه یابند. اگر شما DoSomethingAsync را از یک رشته UI فراخوانی کنید، هر یک از بخش‌های همزمان آن در آن رشته UI اجرا خواهند شد؛ اما اگر از یک رشته استخر رشته فراخوانی کنید، هر یک از بخش‌های همزمان آن در هر رشته استخر رشته اجرا خواهند شد.

شما می‌توانید با انتظار کشیدن بر روی نتیجه متد توسعه‌پذیر ConfigureAwait و عبور false برای پارامتر continueOnCapturedContext این رفتار پیش‌فرض را اجتناب کنید. کد زیر در رشته فراخوانی شروع خواهد شد و پس از اینکه توسط await متوقف شد، در یک رشته استخر ادامه خواهد یافت:

```
async Task DoSomethingAsync()
{
    int value = 13;

    // به‌صورت ناهمزمان 1 ثانیه صبر کنید.

    await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);

    value *= 2;

    // به‌صورت ناهمزمان 1 ثانیه صبر کنید.

    await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);

    Trace.WriteLine(value);
}
```

عملکرد خوب این است که همیشه در متدهای "کتابخانه" اصلی خود ConfigureAwait را فراخوانی کنید و تنها در زمانی که به آن نیاز دارید، زمینه را از نو شروع کنید - در متدهای "رابط کاربری" خارجی خود.

انواع قابل انتظار

کلمه کلیدی await تنها به کار با تسک‌ها محدود نیست؛ بلکه می‌تواند با هر نوع قابل انتظاری که یک الگوی خاص را دنبال می‌کند، کار کند. به عنوان مثال، کتابخانه کلاس پایه شامل نوع `ValueTask<T>` است که تخصیص‌های حافظه را کاهش می‌دهد اگر نتیجه به طور معمول همزمان باشد؛ برای مثال، اگر نتیجه را بتوان از یک کش درون حافظه‌ای خواند. `ValueTask<T>` به طور مستقیم به `Task<T>` قابل تبدیل نیست، اما این الگو را دنبال می‌کند، بنابراین می‌توانید به طور مستقیم بر روی آن انتظار بکشید. مثال‌های دیگری نیز وجود دارد و می‌توانید خودتان انواع قابل انتظار بسازید، اما بیشتر اوقات await یک `Task` یا `Task<TResult>` را می‌گیرد.

دو روش اصلی برای ایجاد نمونه Task

دو روش اساسی برای ایجاد یک نمونه `Task` وجود دارد. برخی از تسک‌ها نمایانگر کد واقعی هستند که باید توسط CPU اجرا شوند؛ این تسک‌های محاسباتی باید با فراخوانی `Task.Run` یا `TaskFactory.StartNew` گر نیاز دارید که آن‌ها بر روی یک زمان‌بندی خاص اجرا شوند) ایجاد شوند. تسک‌های دیگر نمایانگر یک اعلان هستند؛ این نوع تسک‌های مبتنی بر رویداد با استفاده از `TaskCompletionSource<TResult>` (یا یکی از میانبرهای آن) ایجاد می‌شوند. اکثر تسک‌های I/O از `TaskCompletionSource<TResult>` استفاده می‌کنند.

مدیریت خطا در برنامه‌نویسی ناهمزمان

مدیریت خطا با استفاده از `async` و `await` طبیعی است. در کد زیر، `PossibleExceptionAsync` ممکن است یک `NotSupportedException` پرتاب کند، اما `TrySomethingAsync` می‌تواند این استثنا را به طور طبیعی بگیرد. استثنای گرفته شده دارای ردیابی پشته صحیح است و به صورت مصنوعی در `TargetInvocationException` یا `AggregateException` بسته نمی‌شود:


```
async Task TrySomethingAsync()
{
    try
    {
        await PossibleExceptionAsync();
    }
    catch (NotSupportedException ex)
    {
        LogException(ex);
        throw;
    }
}
```

وقتی یک متد `async` یک استثنا پرتاب می کند (یا آن را انتشار می دهد)، استثنا در تسک برگردانده می شود و تسک کامل می شود. وقتی آن تسک انتظار می کشد، عملگر `await` آن استثنا را بازیابی کرده و به گونه ای دوباره پرتاب می کند که ردیابی پشته اصلی آن حفظ شود. بنابراین، کدی مانند مثال زیر به طور مورد انتظار کار خواهد کرد اگر `PossibleExceptionAsync` یک متد `async` باشد:

```

async Task TrySomethingAsync()
{
    // استثنا در تسک خواهد بود، نه به طور مستقیم پرتاب شده
    Task task = PossibleExceptionAsync();

    try
    {
        // ایجاد خواهد شد await استثنا تسک در اینجا، در
        await task;
    }
    catch (NotSupportedException ex)
    {
        LogException(ex);
        throw;
    }
}

```

راهنمایی‌های مهم در مورد متدهای **async**

یک راهنمایی مهم دیگر در مورد متدهای **async** هنگامی که شروع به استفاده از **async** می‌کنید، بهتر است اجازه دهید که آن در تمام کد شما گسترش یابد. اگر یک متد **async** را فراخوانی می‌کنید، باید (در نهایت) تسکی را که برمی‌گرداند، انتظار بکشید. از وسوسه فراخوانی **Task.Wait**، **Task<TResult>.Result** یا **GetAwaiter().GetResult()** خودداری کنید؛ زیرا این کار ممکن است منجر به قفل (deadlock) شود. به متد زیر توجه کنید:

```

async Task WaitAsync()
{
    // زمینه فعلی را ثبت می کند await این ...
    await Task.Delay(TimeSpan.FromSeconds(1));

    // سعی می کند متد را در این زمینه دوباره ادامه دهد ...
}

void Deadlock()
{
    // تأخیر را شروع کنید.
    Task task = WaitAsync();

    // کامل شود async به طور همزمان مسدود کنید و منتظر بمانید تا متد ...

    task.Wait();
}

```

کد در این مثال اگر از یک زمینه UI یا ASP.NET Classic فراخوانی شود، قفل خواهد شد زیرا هر دو زمینه تنها اجازه می دهند یک رشته در هر بار وارد شوند Deadlock WaitAsync. را فراخوانی خواهد کرد که تأخیر را شروع می کند. سپس Deadlock (به طور همزمان) منتظر می ماند تا آن متد کامل شود و رشته زمینه را مسدود می کند. هنگامی که تأخیر کامل می شود، await سعی می کند WaitAsync را در زمینه ثبت شده ادامه دهد، اما نمی تواند زیرا در حال حاضر یک رشته در زمینه مسدود شده است و زمینه تنها اجازه می دهد یک رشته در هر بار وارد شود. قفل می تواند با دو روش جلوگیری شود: می توانید در WaitAsync از ConfigureAwait(false) استفاده کنید (که باعث می شود await زمینه خود را نادیده بگیرد)، یا می توانید فراخوانی WaitAsync را انتظار بکشید (که Deadlock را به یک متد async تبدیل می کند).

استفاده از async به طور کامل

اگر از async استفاده می کنید، بهتر است به طور کامل از async استفاده کنید.

برای یک معرفی کامل تر از async، مستندات آنلاین که مایکروسافت برای async ارائه داده، فوق العاده است؛ من توصیه می کنم حداقل بخش کلیات برنامه نویسی ناهمزمان و الگوی ناهمزمان مبتنی بر تسک (TAP) را مطالعه کنید. اگر می خواهید عمیق تر بروید، همچنین مستندات Async in Depth وجود دارد.

جریانات ناهمزمان

جریانات ناهمزمان زیرساخت async و await را گرفته و آن را برای مدیریت مقادیر متعدد گسترش می دهند. جریانات ناهمزمان حول مفهوم Enumerable های ناهمزمان ساخته شده اند، که مانند Enumerable های معمولی هستند، به جز اینکه اجازه می دهند کارهای ناهمزمان هنگام بازیابی آیتم بعدی در توالی انجام شود. این یک مفهوم بسیار قدرتمند است که در فصل ۳ به طور مفصل تری پوشش داده می شود. جریانات ناهمزمان به ویژه در مواقعی که یک توالی از داده ها به صورت تک به تک یا در تکه ها وارد می شود، بسیار مفید هستند. برای مثال، اگر برنامه شما پاسخ یک API را که از پارامترهای محدودیت و آفست استفاده می کند، پردازش می کند، آنگاه جریانات ناهمزمان یک انتزاع ایده آل هستند. در زمان نوشتن این متن، جریانات ناهمزمان تنها در جدیدترین پلتفرم های NET. در دسترس هستند.

مقدمه ای بر برنامه نویسی موازی

برنامه نویسی موازی باید هر زمان که مقدار قابل توجهی کار محاسباتی دارید که می تواند به تکه های مستقل تقسیم شود، استفاده شود. برنامه نویسی موازی به طور موقت استفاده از CPU را افزایش می دهد تا از طریق آن، کارایی را بهبود بخشد؛ این موضوع در سیستم های کلاینت که CPU ها معمولاً بیکار هستند، مطلوب است، اما معمولاً برای سیستم های سرور مناسب نیست. اکثر سرورها دارای برخی موازی سازی های داخلی هستند؛ به عنوان مثال، ASP.NET چندین درخواست را به طور موازی پردازش خواهد کرد. نوشتن کد موازی در سرور ممکن است در برخی موارد هنوز مفید باشد (اگر می دانید که تعداد کاربران همزمان همیشه کم خواهد بود)، اما به طور کلی، برنامه نویسی موازی در سرور با موازی سازی داخلی آن کار می کند و بنابراین هیچ مزیت واقعی ارائه نمی دهد.

انواع موازی سازی

دو نوع موازی سازی وجود دارد: موازی سازی داده ای و موازی سازی تسکی. موازی سازی داده ای زمانی است که مجموعه ای از داده ها برای پردازش دارید و پردازش هر قسمت از داده عمدتاً مستقل از دیگر قسمت ها است.

موازی‌سازی تسکی زمانی است که مجموعه‌ای از کار برای انجام دارید و هر قسمت از کار عمدتاً مستقل از دیگر قسمت‌ها است. موازی‌سازی تسکی ممکن است پویا باشد؛ اگر یک قطعه کار منجر به چندین قطعه کار اضافی شود، می‌توان آن‌ها را به مجموعه کارها اضافه کرد.

روش‌های انجام موازی‌سازی داده‌ای

روش‌های مختلفی برای انجام موازی‌سازی داده‌ای وجود دارد. `Parallel.ForEach` مشابه یک حلقه `foreach` است و باید در صورت امکان استفاده شود. `Parallel.ForEach` در دستور پخت ۴.۱ پوشش داده شده است. کلاس `Parallel` همچنین از `Parallel.For` پشتیبانی می‌کند که مشابه یک حلقه `for` است و می‌تواند در صورت وابستگی پردازش داده‌ها به اندیس، استفاده شود. کدی که از `Parallel.ForEach` استفاده می‌کند به شکل زیر است:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees)
```

```
{
```

```
    Parallel.ForEach(matrices, matrix => matrix.Rotate(degrees));
```

```
}
```

یک گزینه دیگر `PLINQ` (Parallel LINQ) است که یک متد توسعه‌یافته `AsParallel` برای کوئری‌های LINQ ارائه می‌دهد. `Parallel` از نظر منابع دوستانه‌تر از `PLINQ` است؛ `Parallel` با سایر فرآیندهای موجود در سیستم بهتر کار می‌کند، در حالی که `PLINQ` به‌طور پیش‌فرض سعی می‌کند که خود را در سراسر تمام CPU ها پخش کند. نقطه ضعف `Parallel` این است که صریح‌تر است؛ در حالی که `PLINQ` در بسیاری از موارد کدهای زیباتر و elegant تری دارد. `PLINQ` در دستور پخت ۴.۵ پوشش داده شده و به شکل زیر است:

```
IEnumerable<bool> PrimalityTest(IEnumerable<int> values)
```

```
{
```

```
    return values.AsParallel().Select(value => IsPrime(value));
```

```
}
```

راهنمایی برای پردازش موازی

صرف نظر از روشی که انتخاب می‌کنید، یک راهنمایی کلیدی در پردازش موازی وجود دارد: تکه‌های کار باید تا حد امکان مستقل از یکدیگر باشند.

تا زمانی که تکه کار شما مستقل از همه تکه‌های دیگر باشد، موازی‌سازی خود را به حداکثر می‌رسانید. به محض اینکه شروع به اشتراک‌گذاری وضعیت بین چندین رشته کنید، باید دسترسی به آن وضعیت مشترک را همگام‌سازی کنید و برنامه شما کمتر موازی خواهد بود. فصل ۱۲ به‌طور مفصل‌تر به همگام‌سازی می‌پردازد.

خروجی پردازش موازی

خروجی پردازش موازی شما می‌تواند به روش‌های مختلفی مدیریت شود. می‌توانید نتایج را در نوعی مجموعه همزمان قرار دهید، یا می‌توانید نتایج را به یک خلاصه تجمیع کنید. تجمیع در پردازش موازی رایج است؛ این نوع عملکرد `map/reduce` نیز توسط بارگذاری‌های متد کلاس `Parallel` پشتیبانی می‌شود. دستور پخت ۴.۲ به‌طور مفصل‌تر به تجمیع می‌پردازد.

موازی‌سازی تسکی

اکنون به موازی‌سازی تسکی بپردازیم. موازی‌سازی داده‌ای بر پردازش داده متمرکز است؛ در حالی که موازی‌سازی تسکی فقط در مورد انجام کار است. در سطح بالا، موازی‌سازی داده‌ای و موازی‌سازی تسکی مشابه هستند؛ "پردازش داده" نوعی "کار" است. بسیاری از مسائل موازی‌سازی می‌توانند به‌دورشی حل شوند؛ استفاده از API که برای مسئله موجود طبیعی‌تر است، مناسب است.

`Parallel.Invoke` یکی از انواع متدهای موازی است که نوعی موازی‌سازی تسکی `fork/join` را انجام می‌دهد. این متد در دستور پخت ۴.۳ پوشش داده شده است؛ شما فقط دلخواهی را که می‌خواهید به‌طور موازی اجرا کنید، ارسال می‌کنید:

```

void ProcessArray(double[] array)
{
    Parallel.Invoke(
        () => ProcessPartialArray(array, 0, array.Length / 2),
        () => ProcessPartialArray(array, array.Length / 2, array.Length)
    );
}

```

```

void ProcessPartialArray(double[] array, int begin, int end)
{
    // CPU... پردازش فشرده
}

```

نوع Task و استفاده از آن

نوع Task در اصل برای موازی‌سازی تسک‌ها (Task Parallelism) معرفی شد، اما امروزه در برنامه‌نویسی غیرهمزمان (Asynchronous Programming) نیز مورد استفاده قرار می‌گیرد. یک نمونه از Task که در موازی‌سازی تسک‌ها به کار می‌رود، نمایانگر یک واحد کاری است. می‌توانید از متد Wait برای انتظار تا اتمام تسک استفاده کنید و از ویژگی‌های Result و Exception برای دریافت نتایج یا استثنائات (Exception) استفاده نمایید. کدهایی که به طور مستقیم از Task استفاده می‌کنند پیچیده‌تر از کدهایی هستند که از کلاس Parallel بهره می‌برند، اما زمانی که ساختار موازی‌سازی تا زمان اجرای برنامه مشخص نیست، این نوع استفاده مفید است. در این نوع موازی‌سازی پویا (Dynamic Parallelism)، تعداد واحدهای کاری مورد نیاز در ابتدای پردازش مشخص نیست و در طول اجرا مشخص می‌شود. معمولاً یک واحد کاری پویا باید تسک‌های فرعی مورد نیاز خود را آغاز کند و سپس منتظر اتمام آن‌ها بماند. نوع Task دارای یک علامت خاص به نام TaskCreationOptions.AttachedToParent است که می‌توانید از آن در این نوع موازی‌سازی استفاده کنید. موازی‌سازی پویا در دستورالعمل 4.4 توضیح داده شده است.

موازی سازی تسک ها نیز مانند موازی سازی داده ها باید مستقل باشند. هرچه تسک ها مستقل تر باشند، برنامه کارآمدتر خواهد بود. همچنین، اگر تسک ها مستقل نباشند، نیاز به همگام سازی دارند و نوشتن کدی که به همگام سازی نیاز دارد سخت تر است. در موازی سازی تسک ها باید به ویژه مراقب متغیرهای محلی (Closures) باشید که در درون بلاک ها (Closures) نگهداری می شوند. توجه داشته باشید که این متغیرها ارجاع ها (References) را نگهداری می کنند، نه مقادیر را، بنابراین ممکن است با اشتراک گذاری غیرمنتظره مواجه شوید.

مدیریت استثنا در انواع مختلف موازی سازی مشابه است. از آنجایی که عملیات ها به صورت موازی انجام می شوند، ممکن است استثنای متعددی به وجود آید که همه آن ها در قالب یک AggregateException به کد شما منتقل می شوند. این رفتار در متدهای Task.Wait، Parallel.Invoke، Parallel.ForEach و موارد مشابه ثابت است. نوع AggregateException دارای متدهای مفیدی مانند Flatten و Handle است که برای ساده سازی مدیریت استثنا به کار می روند:

```
try
{
    Parallel.Invoke(() => { throw new Exception(); },
        () => { throw new Exception(); });
}
catch (AggregateException ex)
{
    ex.Handle(exception =>
    {
        Trace.WriteLine(exception);
        return true; // "handled"
    });
}
```


معمولاً نیازی نیست نگران نحوه مدیریت کارها توسط استخر نخها (Thread Pool) باشید. موازی سازی داده ها و تسک ها از بخش کننده های پویا (Dynamically Adjusting Partitioner) برای تقسیم کار بین نخها استفاده می کنند. استخر نخها تعداد نخ های خود را به صورت پویا افزایش می دهد.

استخر نخها دارای یک صف کار (Work Queue) است و هر نخ در استخر نیز دارای صف کار خود می باشد. وقتی یک نخ کاری اضافی را صف بندی می کند، ابتدا آن را به صف خود ارسال می کند، زیرا معمولاً آن کار با آیتم کاری فعلی مرتبط است؛ این رفتار باعث می شود نخها بیشتر روی کار خود تمرکز کنند و دسترسی به کش (Cache) به حداکثر برسد. اگر نخ دیگری کاری برای انجام نداشته باشد، از صف کار نخ دیگر کاری را می دزدد. مایکروسافت برای بهینه سازی استخر نخها تلاش زیادی کرده و تنظیمات بسیاری برای به حداکثر رساندن عملکرد وجود دارد. با این حال، تا زمانی که تسک های شما خیلی کوتاه نیستند، با تنظیمات پیش فرض به خوبی کار می کنند.

تسک ها نباید بیش از حد کوتاه یا خیلی طولانی باشند. اگر تسک های شما خیلی کوتاه باشند، سربار (Overhead) تقسیم داده به تسک ها و صف بندی آن ها در استخر نخها قابل توجه می شود. اگر تسک ها خیلی طولانی باشند، استخر نخها نمی تواند به طور پویا تعادل کاری را به خوبی تنظیم کند. تعیین این که چه زمانی تسک ها خیلی کوتاه یا خیلی طولانی هستند، به مشکل و سخت افزار بستگی دارد. به عنوان یک قاعده کلی، سعی می کنم تسک هایم را تا حد ممکن کوتاه نگه دارم بدون اینکه با مشکلات عملکردی مواجه شوم (زمانی که تسک ها خیلی کوتاه باشند، ناگهان عملکرد کاهش پیدا می کند). حتی بهتر است به جای استفاده مستقیم از تسک ها، از نوع های بالاتر مانند Parallel یا PLINQ استفاده کنید. این نوع های بالاتر دارای بخش بندی خودکار هستند و در زمان اجرا تنظیمات لازم را به صورت خودکار انجام می دهند.

اگر می خواهید در موازی سازی عمیق تر شوید،

بهترین کتاب در این زمینه "Parallel Programming with Microsoft .NET" نوشته Colin Campbell و انتشارات (Microsoft Press) است.

مقدمه ای بر برنامه نویسی واکنشی (Rx)

برنامه نویسی واکنشی نسبت به سایر اشکال همروندی (concurrency) پیچیدگی بیشتری دارد و یادگیری آن زمان بر است. همچنین، اگر مهارت های خود را در این زمینه به روزرسانی نکنید، نگهداری کد واکنشی می تواند دشوار باشد. با این حال، اگر تمایل به یادگیری آن داشته باشید، برنامه نویسی واکنشی بسیار قدرتمند است.

برنامه‌نویسی واکنشی به شما این امکان را می‌دهد که با یک جریان رویدادها (stream of events) مانند یک جریان داده رفتار کنید. به عنوان یک قانون کلی، اگر در کد خود از پارامترهای رویدادها (event arguments) استفاده می‌کنید، به جای استفاده از یک هندلر رویداد معمولی، کد شما می‌تواند با استفاده از System.Reactive بهبود یابد. System.Reactive که قبلاً با نام Reactive Extensions شناخته می‌شد و اغلب به اختصار "Rx" نامیده می‌شد، یک تکنولوژی واحد است که از آن صحبت می‌شود.

برنامه‌نویسی واکنشی بر اساس مفهوم جریان‌های قابل مشاهده (Observable Streams) بنا شده است. هنگامی که به یک جریان قابل مشاهده (observable) اشتراک می‌کنید، ممکن است هر تعداد آیتم داده رویداد (OnNext) را دریافت کنید و سپس جریان ممکن است با یک خطا رویداد (OnError) یا یک اعلان پایان جریان رویداد (OnCompleted) خاتمه یابد. برخی از جریان‌های قابل مشاهده هیچ‌گاه به پایان نمی‌رسند. اینترفیس‌های مربوط به این جریان‌ها به صورت زیر هستند:

```
interface IObservable<in T>
{
    void OnNext(T item);
    void OnCompleted();
    void OnError(Exception error);
}
```

```
interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<TResult> observer);
}
```

با این حال، شما نباید این اینترفیس‌ها را پیاده‌سازی کنید. کتابخانه‌ی System.Reactive مایکروسافت تمام پیاده‌سازی‌های مورد نیاز شما را ارائه می‌دهد. کد واکنشی شباهت زیادی به LINQ دارد؛

می‌توانید آن را به عنوان "LINQ برای رویدادها" در نظر بگیرید System.Reactive. تمام امکانات LINQ را داراست و علاوه بر آن تعداد زیادی عملگرهای مخصوص زمان را اضافه می‌کند.

کد زیر از عملگرهایی مانند Interval و Timestamp استفاده می‌کند و در پایان از Subscribe استفاده شده است. در میان آن‌ها نیز از عملگرهای آشنا مانند Where و Select استفاده می‌شود:

```
Observable.Interval(TimeSpan.FromSeconds(1))
```

```
.Timestamp()
```

```
.Where(x => x.Value % 2 == 0)
```

```
.Select(x => x.Timestamp)
```

```
.Subscribe(x => Trace.WriteLine(x));
```

در این کد، یک شمارنده که از تایمر دوره‌ای (Interval) استفاده می‌کند، شروع به کار می‌کند و به هر رویداد یک زمان‌سنج (Timestamp) اضافه می‌کند. سپس رویدادها برای مقادیر زوج فیلتر می‌شوند (Where)، و زمان‌سنج انتخاب می‌شود (Timestamp) و هر مقدار نتیجه نهایی در Debugger نوشته می‌شود. (Subscribe)

نگران نباشید اگر عملگرهایی مانند Interval برایتان ناآشنا هستند؛ این‌ها در قسمت‌های بعدی کتاب توضیح داده خواهند شد. فعلاً فقط به یاد داشته باشید که این یک پرس‌وجوی LINQ است که بسیار شبیه به پرس‌وجوهای LINQ ای است که با آن‌ها آشنایی دارید. تفاوت اصلی این است که LINQ to Objects و LINQ to Entities از مدل "کشش (pull)" استفاده می‌کنند، در حالی که LINQ برای رویدادها (System.Reactive) از مدل "فشار (push)" استفاده می‌کند، یعنی رویدادها به‌طور خودکار از میان پرس‌وجوها عبور می‌کنند.

تعریف یک جریان قابل مشاهده از اشتراک‌های آن مستقل است. مثال قبلی را می‌توان به صورت زیر بازنویسی کرد:

```
IObservable<DateTimeOffset> timestamps =  
Observable.Interval(TimeSpan.FromSeconds(1))  
.Timestamp()  
.Where(x => x.Value % 2 == 0)  
.Select(x => x.Timestamp);  
  
timestamps.Subscribe(x => Trace.WriteLine(x));
```

این معمول است که یک نوع، جریان‌های قابل مشاهده را تعریف کرده و آن‌ها را به عنوان منبعی از نوع `IObservable<TResult>` در دسترس قرار دهد. سایر انواع می‌توانند به این جریان‌ها اشتراک کنند یا آن‌ها را با عملگرهای دیگر ترکیب کرده و یک جریان قابل مشاهده دیگر ایجاد کنند.

یک اشتراک `Rx` نیز یک منبع است. عملگرهای `Subscribe` یک `IDisposable` بازمی‌گردانند که نمایانگر اشتراک است. زمانی که کد شما کار خود را با یک جریان قابل مشاهده به پایان رساند، باید اشتراک خود را حذف کند.

اشتراک‌ها در جریان‌های گرم (`hot`) و سرد (`cold`) متفاوت عمل می‌کنند. یک جریان گرم، یک جریان رویداد است که همیشه در حال وقوع است و اگر هنگام وقوع رویدادها مشترکی نباشد، آن رویدادها از دست می‌روند. به عنوان مثال، حرکت ماوس یک جریان گرم است. یک جریان سرد، جریانی است که همیشه رویدادهای ورودی ندارد. یک جریان سرد در پاسخ به اشتراک، دنباله‌ای از رویدادها را آغاز می‌کند. برای مثال، دانلود `HTTP` یک جریان سرد است؛ اشتراک باعث ارسال درخواست `HTTP` می‌شود.

عملگر Subscribe همیشه باید یک پارامتر برای مدیریت خطا نیز داشته باشد. مثال‌های قبلی این نکته را رعایت نکرده‌اند؛ مثال زیر نمونه بهتری است که به درستی در صورت وقوع خطا در جریان قابل مشاهده (observable)، واکنش نشان می‌دهد:

```
Observable.Interval(TimeSpan.FromSeconds(1))
    .Timestamp()
    .Where(x => x.Value % 2 == 0)
    .Select(x => x.Timestamp)
    .Subscribe(
        x => Trace.WriteLine(x),
        ex => Trace.WriteLine(ex)
    );
```

نوع `Subject<TResult>` یکی از انواع مفیدی است که هنگام کار با `System.Reactive` به کار می‌آید. این "Subject" مانند پیاده‌سازی دستی یک جریان قابل مشاهده (observable) عمل می‌کند. کد شما می‌تواند متدهای `OnError`، `OnNext` و `OnCompleted` را فراخوانی کند و Subject آن‌ها را به مشترکان (subscribers) خود ارسال می‌کند `Subject<TResult>`. برای آزمایش و بررسی عملکرد بسیار عالی است، اما در کد تولیدی باید تلاش کنید از عملگرهای (operators) استاندارد استفاده کنید که در فصل ۶ پوشش داده شده‌اند، استفاده کنید.

تعداد زیادی از عملگرهای مفید در `System.Reactive` وجود دارند و من فقط تعدادی از آن‌ها را در این کتاب بررسی کرده‌ام. برای اطلاعات بیشتر درباره `System.Reactive`، کتاب آنلاین فوق‌العاده `Introduction to Rx` را پیشنهاد می‌کنم.