مقدمه

به نظر می‌رسد که حیوانی که روی جلد این کتاب دیده می‌شود، یعنی «زباد خرما» (common palm civet)، به موضوع کتاب مرتبط اســت. من تا زمانی که این تصــویر را ندیده بودم، چیزی درباره‌ی این حیوان نمی‌دانســتم، بنابراین آن را بررسی کردم. زبادهای خرما به عنوان آفت شناخته می‌شوند، زیرا در سقف‌خانه‌ها مدفوع می‌کنند و در بدترین زمان‌ها با یکدیگر سـر و صـدا و دعوا راه می‌اندازند. غدد بویایی مقعدی آنها ترشـحی بدبو ایجاد می‌کند. این حیوان در دسـته‌بندی گونه‌های در معرض خطر قرار ندارد و وضـعیتش به عنوان «کمترین نگرانی» (Least Concern) تعیین شـده اسـت که ظاهراً به شـکل مودبانه‌ای به این معنی اسـت: «هر چقدر می‌خواهید از این حیوان‌ها بکشـید، کسـی دلتنگشـان نمی‌شـود.» زبادهای خرما علاقه‌ی زیادی به خوردن میوه‌های قهوه دارند و دانه‌های قهوه را از طریق دســتگاه گوارش خود عبور می‌دهنـد. یکی از گران‌ترین قهوه‌هـای دنیـا، یعنی «کپی لواک» (Kopi Luwak)، از دانـه‌هـای قهوه‌ای کـه از مـدفوع این حیوان استخراج می‌شود، تهیه می‌گردد. طبق گفته‌ی انجمن قهوه تخصصی آمریکا، «این قهوه طعم بدی دارد.»

این موضــوع، زباد خرما را به نمادی مناســب برای توســعه‌ی هم‌زمانی (concurrency) و برنامه‌نویســی چندنخی (multithreading) تبدیل می‌کند. برای افراد ناآشــنا، هم‌زمانی و چندنخی بودن اموری نامطلوب هســتند. آن‌ها باعث می‌شــوند که کدی که به خوبی عمل می‌کند، به بدترین شــکل ممکن رفتار کند. شــرایط 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 هســتند» و به طور کامل از هم‌زمانی اجتناب کرده‌اند. تعدادی توسـعه‌دهنده هسـتند که علاقه‌مند به هم‌زمانی شـده‌اند و بدون ترس از آن اسـتفاده می‌کنند؛ اما بیشـتر 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 هم ایمن‌تر و هم آسان‌تر 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 async و await 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 اســتفاده 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.

استفاده از مثال‌های کد

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

.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] 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: 707–829–0515 (بین‌المللی یا محلی)

فکس: 707–829–0104

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] 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 **مراجعه کنید.**

ما را در فیسبوک پیدا کنید: http://facebook.com/oreilly

ما را در توییتر دنبال کنید: http://twitter.com/oreillymedia

ما را در یوتیوب تماشا کنید: 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. همسرم، 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. دوستتان دارم!

البته، این کتاب به این خوبی نمی‌بود اگر ویرایشـــگران و بازبینی‌کنندگان فنی ما نبودند: اســـتیون توب، پتر آندرکا ("svick")، نیک پالدینو ("casper-One")، لی کمپبل و پدرو فلیکس. بنابراین اگر اشتباهاتی وجود دارد، کاملاً تقصیر آن‌هاســت. شــوخی می‌کنم! ورودی آن‌ها در شــکل‌دهی (و اصــلاح) محتوای کتاب بی‌نهایت ارزشــمند بوده و هر اشــتباه باقی‌مانده به طور قطع مربوط به من اســت. تشــکر ویژه‌ای از اســتیون توب دارم که به من هک Boolean Argument **(فرمول 14.5) و همچنین موضـــوعات دیگر مربوط به** async **را آموزش داد؛ و لی کمپبل 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 تشــکر کنم: اســتیون توب، لوســیان ویســچیک، توماس لوســـک، لی کمپبل، اعضـــای Stack Overflow **و انجمن‌های** MSDN **و شـــرکت‌کنندگان در کنفرانس‌های نرم‌افزاری در ایالت میشـیگان. از اینکه بخشـی از جامعه توسـعه نرم‌افزار هسـتم، 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 زیباسـت. برای دهه‌ها، همزمانی امکان‌پذیر بود اما دسـتیابی به آن دشـوار بود. نرم‌افزارهای همزمان نوشـتن، اشـکال‌زدایی و نگهداری دشـواری داشـتند. در نتیجه، بسـیاری از توسـعه‌دهندگان مسـیر آســان‌تری را انتخاب کردند و از همزمانی اجتناب کردند. با کتابخانه‌ها و ویژگی‌های زبانی که برای برنامه‌های مدرن .NET در دسـترس اسـت، همزمانی اکنون بسـیار آسـان‌تر شـده اسـت. مایکروسـافت پیشـگام در کاهش قابل توجه بار همزمانی بوده اسـت. قبلاً، برنامه‌نویسـی همزمان حوزه‌ی کارشـناسـان بود؛ اما این روزها هر توسـعه‌دهنده‌ای 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. بیایید با همزمانی شروع کنیم.

همزمانی

انجام چندین کار به صورت همزمان.

امیدوارم واضح باشد که همزمانی چگونه می‌تواند مفید باشد. برنامه‌های کاربر نهایی از همزمانی برای پاسخ به ورودی کاربر در حین نوشتن به یک پایگاه داده استفاده می‌کنند. برنامه‌های سروری از همزمانی برای پاسخ به یک درخواست دوم در حالی که درخواست اول را تمام می‌کنند، استفاده می‌کنند. 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 نرم‌افزاری در جهان می‌تواند از همزمانی بهره‌مند شود.

بسیاری از توسعه‌دهندگان با شنیدن اصطلاح "همزمانی" به طور خودکار به "چندریسمانی" فکر می‌کنند. می‌خواهم تمایزی بین این دو قائل شوم.

چندریسمانی

شکلی از همزمانی که از چندین رشته اجرایی استفاده می‌کند.

چندریسمانی به معنای واقعی استفاده از چندین رشته است. همان‌طور که در بسیاری از دستورالعمل‌های این کتاب نشان داده شده است، چندریسمانی یک شکل از همزمانی است، اما قطعاً تنها شکل آن نیست. در واقع، استفاده مستقیم از انواع رشته‌های 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 تکنیک‌های قدیمی را به حداقل می‌رسانم. هیچ یک از

دستورالعمل‌های چندریسمانی در این کتاب از نوع‌های `Thread` یا `BackgroundWorker` استفاده نمی‌کنند؛ آن‌ها با جایگزین‌های برتر تعویض شده‌اند.

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

اما این‌گونه تصور نکنید که چندریسمانی مرده است! چندریسمانی در استخر رشته(**Thread Pool**) 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 `async` و `await` 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 ماشـــین حالت عظیم در نظر بگیرید، رفتار برنامه می‌تواند به عنوان واکنش به یک ســـری رویدادها توصیف شود که در هر رویداد وضعیت خود را به‌روزرسانی می‌کند. این تا این حد انتزاعی یا نظری نیست که به نظر می‌رسـد؛ فریم‌ورک‌های مدرن این رویکرد را در برنامه‌های واقعی بسـیار مفید می‌کنند. برنامه‌نویسـی واکنشـی 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 اسـتخر رشـته‌ها) و برنامه‌نویسـی ناهمزمان اسـتفاده می‌کنند. از ترکیب و تطابق همه اشـکال مختلف همزمانی دریغ نکنید، استفاده کنید.

مقدمه‌ای بر برنامه‌نویسی ناهمزمان

برنامه‌نویســـی ناهمزمان دو مزیت اصـــلی 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 می‌شـود؛ یک برنامه ناهمزمان می‌تواند در حین کار به ورودی کاربر پاسـخگو باشـد. دومین مزیت برای برنامه‌های سمت سرور است: برنامه‌نویسی ناهمزمان امکان مقیاس‌پذیری را فراهم می‌کند. یک برنامه سروری می‌تواند به‌طور نسـبی با اسـتفاده از اسـتخر رشـته مقیاس‌پذیر باشـد، اما یک برنامه سـرور ناهمزمان معمولاً می‌تواند به طور چشمگیری بهتر از آن مقیاس‌پذیر باشد.

هر دو مزیت برنامه‌نویسـی ناهمزمان از یک جنبه زیرسـاختی ناشـی می‌شـود: برنامه‌نویسـی ناهمزمان یک رشـته را آزاد می‌کند. برای برنامه‌های GUI ، برنامه‌نویســـی ناهمزمان رشـــته UI را آزاد می‌کند؛ این اجازه می‌دهد تا برنامه GUI به ورودی کاربر پاسـخگو بماند. برای برنامه‌های سـرور، برنامه‌نویسـی ناهمزمان رشـته‌های درخواسـت را آزاد می‌کند؛ این اجازه می‌دهد تا سرور از رشته‌های خود برای خدمت به درخواست‌های بیشتری استفاده کند.

استفاده از کلمات کلیدی async وawait

برنامه‌های مدرن ناهمزمان در .NET. از دو کلمه کلیدی استفاده می‌کنند async و await کلمه کلیدی async به یک اعلان متد اضافه می‌شود و دو هدف را انجام می‌دهد: این کلمه کلیدی امکان استفاده از کلمه کلیدی awaitرا 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، ITaskاگر مقداری را برنگرداند، 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;
  // به‌صورت ناهمزمان 1 ثانیه صبر کنید.
  await Task.Delay(TimeSpan.FromSeconds(1));
  value *= 2;
  // به‌صورت ناهمزمان 1 ثانیه صبر کنید.
  await Task.Delay(TimeSpan.FromSeconds(1));
  Trace.WriteLine(value);
{
```

یک متد async مانند هر متد دیگری به‌صورت 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 این‌گونه باشد، ادامه به اجرا (به‌صورت همزمان) می‌دهد. در غیر این‌صورت، متد asyncرا متوقف کرده و یک تسک ناتمام را برمی‌گرداند. وقتی آن عملیات بعداً به پایان می‌رسد، متد asyncدوباره به اجرا ادامه خواهد داد.

می‌توانید یک متد asyncرا به‌عنوان داشتن چندین بخش همزمان تصور کنید که توسط بیانیه‌های awaitجدا 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 . Coreبه‌جای 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 falseبرای 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. به‌عنوان مثال، کتابخانه کلاس پایه شامل نوع ValueTask<T> است که تخصیص‌های حافظه را کاهش می‌دهد اگر نتیجه به‌طور معمول همزمان باشد؛ برای مثال، اگر نتیجه را بتوان از یک کش درون‌حافظه‌ای خواند . ValueTask<T> به‌طور مستقیم به Task<T> قابل تبدیل نیست، اما این الگو را دنبال می‌کند، بنابراین می‌توانید به‌طور مستقیم بر روی آن انتظار بکشید. مثال‌های دیگری نیز وجود دارد و می‌توانید خودتان انواع قابل انتظار بسازید، اما بیشتر اوقات awaitیک Taskیا Task<TResult> را می‌گیرد.

دو روش اصلی برای ایجاد نمونهTask

دو روش اساسی برای ایجاد یک نمونه Taskوجود دارد. برخی از تسک‌ها نمایانگر کد واقعی هستند که باید توسط CPUاجرا 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

تســک‌های مبتنی بر رویداد با اســتفاده 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 اســتثنا را به‌طور طبیعی بگیرد. اســتثنای گرفته شـده دارای ردیابی پشـته صـحیح اسـت و به‌صـورت مصـنوعی در TargetInvocationException یا AggregateException بسته نمی‌شود:

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

وقتی یک متد async یک اسـتثنا پرتاب می‌کند (یا آن را انتشـار می‌دهد)، اسـتثنا در تسـک برگردانده می‌شـود و تسـک کامل می‌شود. وقتی آن تسک انتظار می‌کشد، عملگر await آن استثنا را بازیابی کرده و به‌گونه‌ای دوباره پرتاب می‌کند که ردیـابی پشـــته اصـــلی آن حفظ شـــود. بنـابراین، کـدی مـانند مثال زیر به‌طور مورد انتظـار کـار خواهـد کرد اگر PossibleExceptionAsync یک متد async باشد:

```
async Task TrySomethingAsync()
{
  // استثنا در تسک خواهد بود، نه به‌طور مستقیم پرتاب شده.
  Task task = PossibleExceptionAsync();
  try
  {
    // ایجاد خواهد شد await استثنای 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 asyncرا فراخوانی می‌کنید، باید (در نهایت) تسکی را که برمی‌گرداند، انتظار بکشید. از وسوسه فراخوانی Task.Wait، Task<TResult>.Resultیا

GetAwaiter().GetResult(); خودداری کنید؛ زیرا این کار ممکن است منجر به قفل (deadlock) شود. به متد زیر توجه کنید:

```
async Task WaitAsync()
{
  // این await زمینه فعلی را ثبت می‌کند ...
  await Task.Delay(TimeSpan.FromSeconds(1));
  // ... و سعی می‌کند متد را در این زمینه دوباره ادامه دهد.
}
void Deadlock()
{
  // تأخیر را شروع کنید.
  Task task = WaitAsync();
  // به‌طور همزمان مسدود کنید و منتظر بمانید تا متد async کامل شود.
  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 می‌دهند یک رشته در هر بار وارد شوند Deadlock WaitAsync را فراخوانی خواهد کرد که تأخیر را شروع می‌کند. سپس Deadlock(به‌طور همزمان) منتظر می‌ماند تا آن متد کامل شود و رشته زمینه را مسدود می‌کند. هنگامی که تأخیر کامل می‌شود، awaitسعی می‌کند WaitAsyncرا در زمینه ثبت شده ادامه دهد، اما نمی‌تواند زیرا در حال حاضر یک رشته در زمینه مسدود شده است و زمینه تنها اجازه می‌دهد یک رشته در هر بار وارد شود. قفل می‌تواند با دو روش جلوگیری شود: می‌توانید در WaitAsyncاز ConfigureAwait(false)استفاده کنید (که باعث می‌شود awaitزمینه

خود را نادیده بگیرد، یا می‌توانید فراخوانی WaitAsync را انتظار بکشید که Deadlock را به یک متد async تبدیل می‌کند.

استفاده از async به‌طور کامل

اگر از async استفاده می‌کنید، بهتر است به‌طور کامل از async استفاده کنید.

برای یک معرفی کامل‌تر از async ، مستندات آنلاین که مایکروسافت برای async ارائه داده، فوق‌العاده است؛ من توصیه می‌کنم حداقل بخش کلیات برنامه‌نویسی ناهمزمان و الگوی ناهمزمان مبتنی بر تسک (TAP) را مطالعه کنید. اگر می‌خواهید عمیق‌تر 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 می‌دهند. جریانات ناهمزمان حول مفهوم Enumerable های ناهمزمان ساخته شده‌اند، که مانند Enumerable های معمولی هستند، به جز اینکه اجازه می‌دهند کارهای ناهمزمان هنگام بازیابی آیتم بعدی در توالی انجام شود. این یک مفهوم بسیار قدرتمند است که در فصل ۳ به‌طور مفصل‌تری پوشش داده می‌شود. جریانات ناهمزمان به‌ویژه در مواقعی که یک توالی از داده‌ها به‌صورت تک به تک یا در تکه‌ها وارد می‌شود، بسیار مفید هستند. برای مثال، اگر برنامه شما پاسخ یکAPI را که از پارامترهای محدودیت و آفست استفاده می‌کند، پردازش می‌کند، آنگاه جریانات ناهمزمان یک انتزاع ایده‌آل هستند. در زمان نوشتن این متن، جریانات ناهمزمان تنها در جدیدترین پلتفرم‌های NET. در دسترس هستند.

مقدمه‌ای بر برنامه‌نویسی موازی

برنامه‌نویسی موازی باید هر زمان که مقدار قابل توجهی کار محاسباتی دارید که می‌تواند به تکه‌های مستقل تقسیم شود، استفاده شود. برنامه‌نویسی موازی به‌طور موقت استفاده از CPU را افزایش می‌دهد تا از طریق آن، کارایی را بهبود بخشد؛ این موضوع در سیستم‌های کلاینت که CPU ها معمولاً بیکار هستند، مطلوب است، اما معمولاً برای سیستم‌های سرور مناسب نیست. اکثر سرورها دارای برخی موازی‌سازی‌های داخلی هستند؛ به‌عنوان مثال، ASP.NET چندین درخواست را به‌طور موازی پردازش خواهد کرد. نوشتن کد موازی در سرور ممکن است در برخی موارد هنوز مفید باشد (اگر می‌دانید که تعداد کاربران هم‌زمان همیشه کم خواهد بود)، اما به‌طور کلی، برنامه‌نویسی موازی در سرور با موازی‌سازی داخلی آن کار می‌کند و بنابراین هیچ مزیت واقعی ارائه نمی‌دهد.

انواع موازی‌سازی

دو نوع موازی‌سازی وجود دارد: موازی‌سازی داده‌ای و موازی‌سازی تسکی. موازی‌سازی داده‌ای زمانی است که مجموعه‌ای از داده‌ها برای پردازش دارید و پردازش هر قسمت از داده عمدتاً مستقل از دیگر قسمت‌ها 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 کوئری‌هایLINQ 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 تمامCPU ها پخش کند. نقطه ضعف Parallel این است که صریح‌تر است؛ در حالی که PLINQدر بسیاری از موارد کدهای زیباتر و elegent 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

صـــرف نظر از روشـــی 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 شـما مسـتقل از همه تکه‌های دیگر باشـد، موازی‌سـازی خود را به حداکثر می‌رسـانید. 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 شـــما می‌تواند به روش‌های مختلفی مدیریت شـــود. می‌توانید نتایج را در نوعی مجموعه همزمان قرار دهید، یا می‌توانید نتایج را به یک خلاصـــه تجمیع کنید. تجمیع در پردازش موازی رایج اســـت؛ 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)
{
  // پردازش فشرده CPU...
}
```

نوع Task و استفاده از آن

نوع Taskدر 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 Waitبرای 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 دستور العمل 4.4 توضیح داده شده است.

موازی‌سازی تسک‌ها نیز مانند موازی‌سازی داده‌ها باید مستقل باشند. هرچه تسک‌ها مستقل‌تر باشند، برنامه کارآمدتر خواهد بود. همچنین، اگر تسک‌ها مستقل نباشند، نیاز به همگام‌سازی دارند و نوشتن کدی که به همگام‌سازی نیاز دارد سخت‌تر است. در موازی‌سازی تسک‌ها باید به ویژه مراقب متغیرهای محلی (Closures) باشید که در درون بلاک‌ها (Closures) نگهداری می‌شوند. توجه داشته باشید که این متغیرها ارجاع‌ها (References) را نگهداری می‌کنند، نه مقادیر را، بنابراین ممکن است با اشتراک‌گذاری غیرمنتظره مواجه شوید.

مدیریت استثنا در انواع مختلف موازی‌سازی مشابه است. از آنجایی که عملیات‌ها به صورت موازی انجام می‌شوند، ممکن است استثناهای متعددی به وجود آید که همه آن‌ها در قالب یک AggregateException **به کد شما منتقل می‌شوند. این رفتار در متدهای** Parallel.ForEach، Parallel.Invoke، Task.Wait **و موارد مشابه ثابت است. نوع** AggregateException **دارای متدهای مفیدی مانند** Flatten **و** Handle **است که برای ساده‌سازی مدیریت استثنا به کار می‌روند:**

```
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. موازی‌سازی داده‌ها و تسک‌ها از بخش‌کننده‌های پویا** (Dynamically Adjusting Partitioner) **برای تقسیم کار بین نخ‌ها استفاده می‌کنند. استخر نخ‌ها تعداد نخ‌های خود را به صورت پویا افزایش می‌دهد.**

استخر نخ‌ها دارای یک صف کار (Work Queue) **است و هر نخ در استخر نیز دارای صف کار خود می‌باشد. وقتی یک نخ کاری اضافی را صف‌بندی می‌کند، ابتدا آن را به صف خود ارسال می‌کند، زیرا معمولاً آن کار با آیتم کاری فعلی مرتبط است؛ این رفتار باعث می‌شود 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 می‌دزدد. مایکروسافت برای بهینه‌سازی**

استخر نخ‌ها تلاش زیادی کرده و تنظیمات بسیاری برای به حداکثر رساندن عملکرد وجود دارد. با این حال، تا زمانی که تسک‌های شما خیلی کوتاه نیستند، با تنظیمات پیش‌فرض به خوبی کار می‌کنند.

تسک‌ها نباید بیش از حد کوتاه یا خیلی طولانی باشند .اگر تسک‌های شما خیلی کوتاه باشند، سربار (Overhead) تقسیم داده به تسک‌ها و صف‌بندی آن‌ها در استخر نخ‌ها قابل توجه می‌شود. اگر تسک‌ها خیلی طولانی باشند، استخر نخ‌ها نمی‌تواند به‌طور پویا تعادل کاری را به خوبی تنظیم کند. تعیین این که چه زمانی تسک‌ها خیلی کوتاه یا خیلی طولانی هستند، به مشکل و سخت‌افزار بستگی دارد. به عنوان یک قاعده کلی، سعی می‌کنم تسک‌هایم را تا حد ممکن کوتاه نگه دارم بدون اینکه با مشکلات عملکردی مواجه شوم (زمانی که تسک‌ها خیلی کوتاه باشند، ناگهان عملکرد کاهش پیدا می‌کند). حتی بهتر است به جای استفاده مستقیم از تسک‌ها، از نوع‌های بالاتر مانند Parallel یا PLINQ استفاده کنید. این نوع‌های بالاتر دارای بخش‌بندی خودکار هستند و در زمان اجرا تنظیمات لازم را به صورت خودکار انجام می‌دهند.

اگر می‌خواهید در موازی‌سازی عمیق‌تر شوید،

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

مقدمه‌ای بر برنامه‌نویسی واکنشی(Rx)

برنامه‌نویسی واکنشی نسبت به سایر اشکال همروندی (concurrency) پیچیدگی بیشتری دارد و یادگیری آن زمان‌بر است. همچنین، اگر مهارت‌های خود را در این زمینه به‌روزرسانی نکنید، نگهداری کد واکنشی می‌تواند دشوار باشد. با این حال، اگر تمایل به یادگیری آن داشته باشید، برنامه‌نویسی 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. به‌عنوان 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 IObserver<in T>
{
 void OnNext(T item);
 void OnCompleted();
 void OnError(Exception error);
}

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

با این حال، شـــما نباید این اینترفیس‌ها را پیاده‌ســـازی کنید. کتابخانه‌ی System.Reactive مایکروســـافت 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 Debuggerنوشته می‌شود.(Subscribe)

نگران نباشید اگر عملگرهایی مانند Intervalبرایتان 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. به‌عنوان 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 OnNext، OnError 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) استانداردی 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.

مقدمه‌ای بر Dataflows

TPL Dataflow ترکیبی جالب از فناوری‌های غیرهمزمان و موازی است. این ابزار زمانی مفید است که بخواهید یک دنباله از فرآیندها را بر روی داده‌هایتان اعمال کنید. به‌عنوان مثال، ممکن است نیاز داشته باشید داده‌ها را از یک URL دانلود کنید، آن‌ها را تجزیه کنید و سپس به‌طور موازی با داده‌های دیگر پردازش کنید TPL Dataflow .معمولاً به‌عنوان یک خط لوله ساده استفاده می‌شود که در آن داده‌ها از یک سمت وارد می‌شوند و تا زمانی که از سمت دیگر خارج شوند، حرکت می‌کنند. با این حال، TPL Dataflow بسیار قدرتمندتر از این است و قادر به مدیریت هر نوع شبکه‌ای (mesh) می‌باشد.

شما می‌توانید انشعاب‌ها، اتصالات و حلقه‌ها را در یک شبکه تعریف کنید و TPL Dataflow به‌خوبی آن‌ها را مدیریت خواهد کرد. با این حال، بیشتر اوقات، شبکه‌های TPL Dataflow به‌عنوان یک خط لوله استفاده می‌شوند. واحد اصلی ساختاری یک شبکه Dataflow یک بلوک داده است. یک بلوک می‌تواند یک بلوک هدف (دریافت‌کننده داده)، یک بلوک منبع (تولیدکننده داده) یا هر دو باشد. بلوک‌های منبع می‌توانند به بلوک‌های هدف متصل شوند تا شبکه‌ای را ایجاد کنند؛ این موضوع در دستورالعمل 5.1 بررسی شده است.

بلوک‌ها به‌طور نیمه‌مستقل عمل می‌کنند و تلاش می‌کنند داده‌ها را به محض رسیدن پردازش کرده و نتایج را به سمت پایین‌دست ارسال کنند. روش معمول استفاده از TPL Dataflow این است که ابتدا تمام بلوک‌ها را ایجاد کنید، آن‌ها را به هم متصل کنید و سپس داده‌ها را از یک سمت وارد کنید. داده‌ها سپس به‌طور خودکار از سمت دیگر خارج می‌شوند. دوباره تأکید می‌شود که Dataflow بسیار قدرتمندتر از این است؛ این امکان وجود دارد که اتصالات را قطع کرده،

بلوک‌های جدید ایجاد کرده و آن‌ها را در حالی که داده‌ها در حال عبور از شبکه هستند اضافه کنید، اما این یک سناریوی بسیار پیشرفته است.

بلوک‌های هدف دارای بافرهایی برای داده‌هایی هستند که دریافت می‌کنند. وجود این بافرها به آن‌ها این امکان را می‌دهد که داده‌های جدید را بپذیرند حتی اگر هنوز آماده پردازش آن‌ها نباشند؛ این کار باعث می‌شود که جریان داده‌ها در شبکه ادامه یابد. این بافرینگ می‌تواند در سناریوهای انشعابی مشکلاتی ایجاد کند، جایی که یک بلوک منبع به دو بلوک هدف متصل است. هنگامی که بلوک منبع داده‌ای برای ارسال به پایین‌دست دارد، شروع به ارائه آن به بلوک‌های متصل به‌صورت یک‌به‌یک می‌کند. به‌طور پیش‌فرض، بلوک هدف اول فقط داده را می‌گیرد و آن را بافر می‌کند، و بلوک هدف دوم هرگز داده‌ای دریافت نمی‌کند. راه‌حل این مشکل محدود کردن بافرهای بلوک‌های هدف با غیرحریص کردن آن‌ها است؛ این موضوع در دستورالعمل 5.4 پوشش داده شده است.

یک بلوک زمانی دچار خطا می‌شود که چیزی اشتباه پیش برود، به‌عنوان‌مثال، زمانی که پردازش‌کننده داده استثنایی را در حین پردازش یک داده پرتاب کند. وقتی یک بلوک دچار خطا می‌شود، دریافت داده‌ها را متوقف می‌کند. به‌طور پیش‌فرض، این وضعیت کل شبکه را از کار نمی‌اندازد؛ این امکان را به شما می‌دهد که آن قسمت از شبکه را بازسازی کنید یا داده‌ها را به سمت دیگری هدایت کنید. بااین‌حال، این یک سناریوی پیشرفته است؛ بیشتر مواقع، شما می‌خواهید که خطاها در طول لینک‌ها به بلوک‌های هدف منتقل شوند Dataflow .از این گزینه نیز پشتیبانی می‌کند؛ تنها بخش چالش‌برانگیز این است که وقتی یک استثنا در طول یک لینک منتقل می‌شود، در یک AggregateException پیچیده می‌شود. بنابراین، اگر یک خط لوله طولانی داشته باشید، ممکن است با یک استثنای عمیقاً تو در تو روبرو شوید؛ متد AggregateException.Flatten می‌تواند برای دور زدن این مشکل مورد استفاده قرار گیرد:

```
try
{
   var multiplyBlock = new TransformBlock<int, int>(item =>
   {
      if (item == 1)
         throw new InvalidOperationException("Blech.");
      return item * 2;
   });
   var subtractBlock = new TransformBlock<int, int>(item => item - 2);
   multiplyBlock.LinkTo(subtractBlock, new DataflowLinkOptions { PropagateCompletion = true });
   multiplyBlock.Post(1);
   subtractBlock.Completion.Wait();
}
catch (AggregateException exception)
{
   AggregateException ex = exception.Flatten();
   Trace.WriteLine(ex.InnerException);
}
```

دستورالعمل 5.2 به بررسی جزئیات بیشتر مدیریت خطا در Dataflow می‌پردازد. در نگاه اول، شبکه‌های Dataflow بسیار شبیه به جریان‌های قابل مشاهده (observable streams) به نظر می‌رسند و در بسیاری از جنبه‌ها با هم مشترک هستند. هر دو شبکه و جریان دارای مفهوم عبور داده‌ها از طریق آن‌ها هستند. همچنین، هر دو دارای ایده اتمام عادی (اطلاع‌رسانی که دیگر داده‌ای نمی‌آید) و اتمام خطا (اطلاع‌رسانی که در حین پردازش داده، خطایی رخ داده است) می‌باشند. اما System.Reactive (Rx) و TPL Dataflow قابلیت‌های یکسانی ندارند. Observable های Rx به‌طور کلی در انجام هر گونه کار مرتبط با زمان بهتر از بلوک‌های Dataflow هستند. بلوک‌های Dataflow به‌طور کلی در انجام پردازش‌های موازی بهتر از Observable های Rx عمل می‌کنند. از نظر

مفهومی، Rxبیشتر شبیه راه‌اندازی callbacks است: هر مرحله در Observable به‌طور مستقیم مرحله بعدی را فراخوانی می‌کند. در مقابل، هر بلوک در یک شبکه Dataflow از سایر بلوک‌ها مستقل است. هر دو Rx و TPL Dataflowکاربردهای خاص خود را دارند و برخی همپوشانی نیز دارند. آن‌ها همچنین به‌خوبی با هم کار می‌کنند؛ دستورالعمل 8.8 به بررسی قابلیت همکاری بین Rx و TPL Dataflow می‌پردازد.

اگر با فریم‌ورک‌های بازیگر (actor frameworks) آشنا باشید، TPL Dataflowبه نظر می‌رسد که شباهت‌هایی با آن‌ها دارد. هر بلوک Dataflow مستقل است و در این معنا که وظایف لازم را به‌صورت خودکار اجرا می‌کند، مانند اجرای یک delegate تبدیل یا ارسال خروجی به بلوک بعدی. همچنین می‌توانید هر بلوک را طوری تنظیم کنید که به‌صورت موازی اجرا شود، بنابراین می‌تواند وظایف متعددی را برای پردازش ورودی اضافی راه‌اندازی کند. به دلیل این رفتار، هر بلوک شباهت‌هایی با یک بازیگر در یک فریم‌ورک بازیگری دارد. با این حال، TPL Dataflowیک فریم‌ورک کامل بازیگری نیست؛ به‌خصوص اینکه از بازیابی خطاها یا تلاش‌های پاکسازی‌شده به‌طور داخلی پشتیبانی نمی‌کند TPL . Dataflowیک کتابخانه با احساسی شبیه به بازیگران است، اما یک فریم‌ورک بازیگری کامل نیست.

پرکاربردترین انواع بلوک‌های TPL Dataflow شامل) TransformBlock<TInput, TOutput> مشابه Select در (LINQ) و TransformManyBlock<TInput, TOutput> (LINQ) مشابه SelectMany در (LINQ) و ActionBlock<TResult>است که یک delegate را برای هر داده اجرا می‌کند. برای اطلاعات بیشتر در مورد TPL Dataflow، مستندات MSDN و "راهنمای پیاده‌سازی بلوک‌های سفارشی TPL Dataflow" را توصیه می‌کنم.

مقدمه‌ای بر برنامه‌نویسی چندنخی

یک رشته (thread) یک اجرایی مستقل است. هر فرایند دارای چندین رشته است و هر یک از این رشته‌ها می‌تواند به‌طور همزمان کارهای مختلفی را انجام دهد. هر رشته دارای یک پشته مستقل است، اما از همان حافظه به‌صورت مشترک با تمام رشته‌های دیگر در یک فرایند استفاده می‌کند. در برخی از برنامه‌ها، یک رشته خاص وجود دارد. به‌عنوان مثال، برنامه‌های رابط کاربری (UI) دارای یک رشته خاص UI هستند و برنامه‌های کنسولی یک رشته خاص اصلی دارند.

هر برنامه NET. یک استخر رشته (thread pool) دارد. استخر رشته تعدادی از رشته‌های کاری را نگه‌داری می‌کند که منتظر اجرای هر کاری که شما برای آن‌ها دارید، هستند. استخر رشته مسئول تعیین تعداد رشته‌ها در هر لحظه در

استخر است. ده‌ها تنظیمات پیکربندی وجود دارد که می‌توانید با آن‌ها کار کنید تا این رفتار را تغییر دهید، اما پیشنهاد می‌کنم آن را به حال خود رها کنید؛ استخر رشته به‌دقت تنظیم شده است تا بیشتر سناریوهای واقعی را پوشش دهد.

تقریباً هیچ نیازی نیست که خودتان یک رشته جدید ایجاد کنید. تنها زمانی که باید یک نمونه Thread ایجاد کنید، زمانی است که به یک رشته STA برای ارتباط COM نیاز دارید.

یک رشته یک انتزاع در سطح پایین است. استخر رشته یک انتزاع در سطح کمی بالاتر است؛ وقتی کدی کارها را به استخر رشته صف می‌کند، خود استخر رشته در صورت لزوم، ایجاد یک رشته را بر عهده می‌گیرد. انتزاعات پوشش داده‌شده در این کتاب بالاتر از این‌ها هستند: صف‌های پردازش موازی و داده‌جریان (dataflow) به‌طور ضروری با استخر رشته کار می‌کنند. کدی که از این انتزاعات بالاتر استفاده می‌کند، به‌مراتب آسان‌تر از کدی است که از انتزاعات سطح پایین استفاده می‌کند.

به همین دلیل، انواع Thread و BackgroundWorker در این کتاب به‌طور کامل پوشش داده نشده‌اند. آن‌ها زمانی داشته‌اند و آن زمان به پایان رسیده است.

مجموعه‌ها برای برنامه‌های همزمان

دو دسته مجموعه وجود دارد که برای برنامه‌نویسی همزمان مفید هستند: مجموعه‌های همزمان (concurrent collections) و مجموعه‌های غیرقابل تغییر (immutable collections). هر دو این دسته مجموعه‌ها در فصل 9 پوشش داده شده‌اند. مجموعه‌های همزمان به چندین رشته اجازه می‌دهند تا به‌طور همزمان و به‌صورت ایمن آن‌ها را به‌روزرسانی کنند. بیشتر مجموعه‌های همزمان از snapshot ها استفاده می‌کنند تا یک رشته بتواند مقادیر را شمارش کند در حالی که رشته دیگری ممکن است مقادیر را اضافه یا حذف کند. مجموعه‌های همزمان معمولاً از محافظت یک مجموعه عادی با یک قفل، کارآمدتر هستند.

مجموعه‌های غیرقابل تغییر کمی متفاوت هستند. یک مجموعه غیرقابل تغییر نمی‌تواند واقعاً اصلاح شود؛ در عوض، برای اصلاح یک مجموعه غیرقابل تغییر، شما یک مجموعه جدید ایجاد می‌کنید که نمایانگر مجموعه اصلاح‌شده است. این ممکن است به نظر ناکارآمد بیاید، اما مجموعه‌های غیرقابل تغییر تا حد امکان حافظه را بین نمونه‌های مجموعه به اشتراک می‌گذارند، بنابراین به بدی که به نظر می‌رسد نیست. نکته مثبت درباره مجموعه‌های غیرقابل تغییر این است که تمام عملیات خالص هستند، بنابراین آن‌ها با کدهای تابعی به‌خوبی کار می‌کنند.

طراحی مدرن

بیشتر فناوری‌های همزمان یک جنبه مشابه دارند: آن‌ها به‌صورت تابعی (functional) هستند. منظور من از تابعی این نیست که "کار را انجام می‌دهند"، بلکه به‌عنوان یک سبک برنامه‌نویسی است که بر اساس ترکیب توابع است. اگر یک ذهنیت تابعی را اتخاذ کنید، طراحی‌های همزمان شما کمتر پیچیده خواهد بود.

یکی از اصول برنامه‌نویسی تابعی، خلوص (purity) است (یعنی اجتناب از اثرات جانبی). هر بخش از راه‌حل برخی ارزش‌ها را به‌عنوان ورودی می‌گیرد و برخی ارزش‌ها را به‌عنوان خروجی تولید می‌کند. تا جایی که ممکن است، باید از وابسته بودن این بخش‌ها به متغیرهای جهانی (یا مشترک) یا به‌روزرسانی ساختارهای داده جهانی (یا مشترک) اجتناب کنید. این موضوع در مورد هر بخشی صدق می‌کند، خواه این بخش یک متدasync ، یک کار موازی، یک عملیات System.Reactiveیا یک بلوک داده باشد. البته، دیر یا زود محاسبات شما باید تأثیری داشته باشند، اما خواهید دید که کد شما تمیزتر خواهد بود اگر بتوانید پردازش را با بخش‌های خالص انجام دهید و سپس با نتایج به‌روزرسانی‌ها را انجام دهید.

اصول دیگر برنامه‌نویسی تابعی غیرقابل تغییر بودن (immutability) است. غیرقابل تغییر بودن به این معناست که یک قطعه داده نمی‌تواند تغییر کند. یکی از دلایل اینکه داده‌های غیرقابل تغییر برای برنامه‌های همزمان مفید هستند، این است که شما هرگز به هماهنگی (synchronization) برای داده‌های غیرقابل تغییر نیاز ندارید؛ این واقعیت که نمی‌توانند تغییر کنند، هماهنگی را غیرضروری می‌سازد. داده‌های غیرقابل تغییر همچنین به شما کمک می‌کنند تا از اثرات جانبی جلوگیری کنید. توسعه‌دهندگان به‌تدریج از انواع غیرقابل تغییر بیشتری استفاده می‌کنند و این کتاب دارای چندین دستورالعمل است که به بررسی ساختارهای داده غیرقابل تغییر می‌پردازد.

خلاصه‌ای از تکنولوژی‌های کلیدی

چارچوب NET. از ابتدای تأسیس خود برخی از پشتیبانی‌ها را برای برنامه‌نویسی ناهمزمان asynchronous) programming)ارائه داده است. با این حال، برنامه‌نویسی ناهمزمان تا سال 2012 دشوار بود، زمانی که) 4.5 NET. به همراه 5.0 #C و (2012 VB کلیدواژه‌های asyncو awaitرا معرفی کرد. این کتاب از رویکرد مدرن async/await برای تمامی دستورالعمل‌های ناهمزمان استفاده خواهد کرد و تعدادی دستورالعمل نشان می‌دهد که چگونه بین asyncو الگوهای قدیمی‌تر برنامه‌نویسی ناهمزمان تعامل برقرار کنید. اگر به پشتیبانی از پلتفرم‌های قدیمی‌تر نیاز دارید، به پیوست A مراجعه کنید.

کتابخانه Task Parallel Library (TPL) **در** .NET 4.0. **معرفی شد و از هر دو نوع موازی‌سازی داده و وظیفه** (task parallelism) **پشتیبانی کامل می‌کند. امروزه، این کتابخانه حتی بر روی پلتفرم‌های با منابع کمتر، مانند تلفن‌های همراه نیز در دسترس است** TPL .**در** .NET. **گنجانده شده است.**

تیم System.Reactive **به‌طور جدی تلاش کرده است تا از حداکثر پلتفرم‌ها پشتیبانی کند**System.Reactive .، **مانند** asyncو await، **مزایایی برای انواع مختلف برنامه‌ها، هم برای کلاینت و هم برای سرور فراهم می‌کند .** System.Reactive**در بسته** NuGet **به نام** System.Reactive **در دسترس است.**

کتابخانه TPL Dataflow **به‌طور رسمی در بسته** NuGet **برای** System.Threading.Tasks.Dataflow**توزیع می‌شود.**

بیشتر مجموعه‌های همزمان (concurrent collections) **به‌صورت داخلی در** .NET. **ساخته شده‌اند؛ برخی مجموعه‌های همزمان اضافی نیز در بسته** NuGet **به نام** System.Threading.Channels**در دسترس است. مجموعه‌های غیرقابل تغییر** (immutable collections) **نیز در بسته** NuGet **به نام** System.Collections.Immutable**موجود هستند.**