Засоби асинхронного програмування - потужний механізм, добавлений у версію 5.0 мови програмування C#. Це відбулося саме в той момент, коли продуктивність і розпаралелювання викликають все більш пильний інтерес у розробників програмного забезпечення. При правильному використанні нові засоби дозволяють створювати програми з такими характеристиками продуктивності і паралелізму, яких раніше можна було досягти тільки за рахунок написання об'ємного і громіздкого коду. Однак тема ця непроста, і у неї є маса нюансів, які є неочевидні з першого погляду. Якщо не вважати Visual Basic .NET, в який засоби асинхронного програмування були додані одночасно з C #, то ні в одному з інших широко поширених мов програмування нічого еквівалентного немає. Рекомендації щодо їх застосування в реальних програмах знайти важко, досвід ще не напрацьований

Код називається асинхронним, якщо він запускає якусь довготривалу операцію, але не чекає її завершення. Протилежністю є блокуючий код, який нічого не робить, поки операція не завершиться.

До числа таких тривалих операцій можна віднести:

* мережеві запити;
* доступ до диска;
* тривалі затримки.

Основна відмінність полягає в тому, в якому потоці виконується код. У всіх популярних мовах програмування код працює в контексті якогось потоку операційної системи. Якщо цей потік продовжує робити щось ще, поки виконується тривала операція, то код асинхронний. Якщо потік в цей час нічого не робить,значить, він заблокований , отже було написано блокуючий код.

Для асинхронного коду характерна типова трудність: як дізнатися,коли операція завершилася? Адже тільки після цього можна приступити до обробки її результатів. У блокую чому коді все тривіально - наступний рядок поміщається відразу після виклику тривалої операції. Але в асинхронному світі так зробити не можна, тому що розміщена в цьому місці рядок майже напевно буде виконана раніше, ніж асинхронна операція завершиться.

Для вирішення цієї проблеми придуманий цілий ряд прийомів, які дозволяють виконати код по завершенні фонової операції:

* включити потрібний код до складу самої операції, після коду.
* підписатися на подію, що генерується по завершенні;
* передати делегат або лямбда-функцію, яка повинна бути

виконана по завершенні (зворотний виклик).

Якщо код, який іде за асинхронною операцією, необхідно виконати в конкретному потоці (наприклад, в потоці користувацького інтерфейсу в програмі на базі WinForms або WPF), то доводиться ставити операцію в чергу цього потоку.

Асинхронний код звільняє потік, з якого був запущений. І це дуже добре з багатьох причин. Насамперед, потоки не потребують ресурсів комп'ютера, а чим менше витрачається ресурсів, тим краще. Часто існує лише один потік, здатний виконати певне завдання (наприклад, потік користувацького інтерфейсу) і, якщо не звільнити його швидко, то додаток перестане

реагувати на дії користувача. Але найважливішим є той факт, що асинхронне виконання відкриває можливість для паралельних обчислень. Можна структурувати програму по-новому, реалізувавши конструкторський паралелізм, але не жертвуючи простотою і зручністю супроводу.

У версії C # 5.0 Microsoft додав механізм, котрий постає у вигляді двох нових ключових слів: async і await. Цей механізм спирається на ряд нововведень в .NET Framework4.5, без яких був би марний Механізм async вбудований в компілятор і без підтримки з його боку не міг би бути реалізований в бібліотеці. Компілятор перетворює вихідний код, тобто діє приблизно по тому же принципу, що і лямбда-вирази і літератори в попередніх версіях C #. Ця можливість істотно спрощує асинхронне програмування, позбавляючи від необхідності використовувати складні прийоми, як то було в попередніх версіях мови. З її допомогою можна написати всю програму цілком в асинхронному стилі. На C # завжди можна було писати асинхронні програми, але це вимагало значних зусиль з боку програміста.

Механізм async дає легкий спосіб висловити, що повинна робити

програма по завершенні тривалої асинхронної операції. метод, позначений ключовим словом async, компілятор перетворює так, що асинхронний код виглядає дуже схоже на блокуючий еквівалент. Найбільший інтерес представляє ключове слово await. Бачачи його, компілятор переписує метод таким чином, що весь код після await переноситься в окремий метод.

Механізм async навмисно спроектований так, щоб максимально нагадувати блокуючий код. Можна розглядати довготривалі або віддалені операції, як ніби вони виконуються локально і швидко, збільшивши продуктивність за рахунок асинхронності. Однак він не дає вам зовсім забути про те, що насправді операція виконується у фоновому режимі і відбувається зворотний виклик.

Необхідно мати на увазі, що багато засобів мови в асинхронному режимі ведуть себе по-іншому, зокрема:

* виключення і блоки try-catch-finally;
* Значення які метод повертає;
* потоки і контекст;
* продуктивність.

Асинхронне програмування - річ важлива і корисна, але чому саме важлива, залежить від виду програми. До додатків для настільних комп'ютерів пред'являється одна важлива вимога - вони повинні реагувати на дії користувача. Дослідження в галузі людино-машинного інтерфейсу показують, що користувач не звертає уваги на повільну роботу програми, якщо вона відгукується на його дії і бажано має індикатор виконання. Але якщо програма зависає, то користувач незадоволений. Зазвичай зависання пов'язані з тим, що програма перестає реагувати на дії користувача під час виконання тривалої операції, будь то повільне обчислення або операція вводу / виводу, наприклад - звернення до мережі.

Всі каркаси для організації користувацького інтерфейсу в C # працюють в одному потоці. Це відноситься і до WinForms, і до WPF, і до Silverlight. Тільки цей потік може керувати вмістом вікна, розпізнавати дії користувача і реагувати на них. Якщо він зайнятий або блокований довше декількох десятків мілісекунд, то користувач вважатиме, що додаток завис. Асинхронний код, навіть написаний вручну, дозволяє потоку користувацького інтерфейсу повернутися до свого основного обов’язку – опитування черги повідомлень і реагуванню на події які з’являються. Він також може анімувати хід виконання завдання. Це і служить для користувача наочним підтвердженням того, що програма працює.

**Серверний код веб-додатка**

У ASP.NET-додатків на веб-сервері немає обмеження на єдиний потік, як у випадку програм з користувацьким інтерфейсом. І тим не менше асинхронне виконання може виявитися досить корисним, так як для таких додатків характерні тривалі операції, особливо запити до бази даних.

Залежно від версії IIS може бути обмеженим або загальне число потоків, які обслуговують веб-запити, або загальне число одночасних оброблюваних запитів. Якщо велика частина часу обробки запиту йде на звернення до бази даних, то збільшення числа одночасно оброблюваних запитів може знизити пропускну здатність сервера.

Коли потік блокований в очікуванні якоїсь події, він не використовує процесорний час. Але не слід думати, що він взагалі не споживає ресурси сервера. Насправді, будь-який потік, навіть блокуючий, споживає два цінних ресурси:

* Пам'ять;

У Windows для кожного керованого потоку резервується приблизно один мегабайт віртуальної пам'яті. Якщо кількість потоків визначається десятками, то це не страшно, але коли їх сотні, то це може перетворитися на проблему. Якщо операційна система змушена вивантажувати пам'ять на диск, то відновлення потоку різко сповільнюється.

* Ресурси планувальника;

Планувальник операційної системи відповідає за виділення потокам процесорів. Планувальник повинен розглядати навіть заблоковані потоки, інакше він не буде знати, коли вони розблокуються. Це уповільнює контекстне переключення, а, значить, і роботу системи в цілому.

У сукупності ці накладні витрати означають додаткове навантаження на сервер, а, отже, збільшення затримки і зниження пропускної здатності. Основна відмінна особливість асинхронного коду полягає в тому, що потік, який почав тривалу операцію, звільняється для інших справ. У разі ASP.NET цей потік береться з пулу потоків, тому після запуску тривалої операції він відразу ж повертається в пул і може потім використовуватися для обробки інших запитів. Таким чином, для обробки однієї і тієї ж кількості запитів потрібно менше потоків.

**Паралельний код**

Сучасні комп'ютери оснащуються декількома процесорними ядрами, що працюють незалежно один від одного. Бажано,щоб програма могла задіяти наявні ядра, але два ядра не можуть писати в одну і ту ж комірку пам'яті, так як це загрожує пошкодженням її вмісту. Можливо, було б краще застосовувати чисте (інакше кажучи,функціональне) програмування, при якому не існує побічних ефектів, тобто стан пам'яті не змінюється. Це допомогло б повніше скористатися перевагами паралелізму, але для деяких програм це неприйнятно. Для користувацьких інтерфейсів поняття стану необхідно. Бази даних самі по собі є станом.

Стандартне рішення припускає використання взаємно виключних блокувань (м'ютексів) у випадках, коли кілька ядер потенційно можуть звертатися до однієї і тієї ж комірки пам'яті. Але тут є свої проблеми. Часто буває так, що програма захоплює блокування, а потім викликає метод або генерується подія, в якій захоплюється інше блокування. Іноді утримувати відразу два блокування необов'язково, але так код стає простішим. У результаті іншим потокам доводиться чекати звільнення блокування, хоча вони могли б у цей час займатися корисною роботою. Гірше того, іноді виникає ситуація, коли потрібний потік чекає звільнення блокування, яка є зайнята іншим, а це призводить до взаємоблокування (deadlock). Такі помилки важко передбачити, відтворювати і виправляти. Одне з найбільш багатообіцяючих рішень - модель обчислень,заснована на акторах. При такому підході кожна ділянка пам'яті належить рівно одному актору. Єдиний спосіб записати дані в цю пам'ять - відправляти актору-власнику повідомлення, які він обробляє по черзі і, можливо, посилає повідомлення у відповідь. Але це якраз і є асинхронне програмування. Запит дії у актора - типова асинхронна операція,

оскільки ми можемо займатися іншими речами, поки не прийде

відповідь - повідомлення. А значить, для програмування такої моделі

можна використовувати механізм async.

Сприймати механізм async в C # 5.0 і, зокрема, ключове слово await можна двома способами:

* як мовний засіб з чітко певною поведінкою, яке можна вивчити і застосовувати;
* як перетворення програми на етапі компіляції, тобто синтаксичну глазур, що приховує більш складний код на C #, в якому ключове слово async не використовується.

Коли потік виконання програми доходить до оператора await, повинні відбутися дві речі.

Поточний потік повинен бути звільнений, щоб поведінка програми була асинхронною. З звичною, синхронною, точкою зору це означає, що метод повинен повернути управління. Коли завдання Task, в операторі await, завершиться, ваш метод повинен продовжити виконання з того місця, де перед цим повернув управління, як ніби цього повернення ніколи не було.

Щоб домогтися такої поведінки, метод повинен призупинити виконання, дійшовши до await, і відновити його згодом. Цю послідовність подій можна розглядати як аналог режиму гібернації комп'ютера (режим S4), тільки в більш дрібному масштабі. Поточний стан методу зберігається, і метод повертає управління. При переході в режим гібернації динамічний стан всіх програм записується на диск і комп'ютер вимикається. Відключення живлення не нанесе комп'ютеру, що знаходиться в режимі гібернації, ніякої шкоди. Точно так же, метод що очікується не споживає ніяких ресурсів, окрім хіба що крихітного об'єму пам’яті, оскільки потік що його звільнив звільнений. В ідеалі програміст не повинен помічати, що мала місце гібернація. Незважаючи на те, що припинення методу в середині і відновлення - досить складна операція, C # гарантує, що програма продовжить виконання, ніби нічого не трапилося.

**Стан методу**

Щоб стало ясніше, скільки роботи повинен виконати компілятор C #, зустрівши в програмі оператор await, перерахуємо, які аспекти стану методу необхідно зберегти. По-перше, запам'ятовуються всі локальні змінні методу, в

тому числі:

* параметри методу;
* всі змінні, визначені в поточній області видимості;
* всі інші змінні, наприклад лічильники циклів;
* змінну this, якщо метод не статичний.

В результаті після відновлення методу виявляться доступні всі змінні-члени класу. Все це зберігається у вигляді об'єкта в купі .NET, що обслуговується

складальником сміття. Таким чином, зустрівши await, компілятор виділяє пам'ять для об'єкта, тобто витрачає ресурси, але в більшості випадків це не призводить до втрати продуктивності. C # також запам'ятовує місце, де зустрівся оператор await. Цю інформацію можна представити у вигляді числа, рівного порядковому номеру досягнутого оператора await серед всіх інших які зустрічається в методі. Далі, коли програма доходить до першого await в методі, цей метод повертає управління. Якщо метод не оголошений як async void, то в цей момент повертається об'єкт Task, щоб викликаюча програма могла дочекатися завершення. C # повинен також запам'ятати цей об'єкт, щоб по завершенні методу його можна було заповнити і продовжити виконання ланцюжка асинхронних методів.

**Контекст**

Намагаючись зробити процес очікування максимально прозорим, C # запам'ятовує різні аспекти контексту в точці, де зустрівся await, а потім, при поновленні методу, відновлює їх. Найбільш важливим з усіх є контекст синхронізації, який серед іншого дозволяє відновити виконання методу в конкретному потоці. Це особливо важливо для додатків з графічним інтерфейсом, яким можна маніпулювати тільки з одного, цілком певного потоку. Є й інші види контексту викликаючого потоку, які так само необхідно запам'ятати. Всі вони збираються у відповідних класах, найбільш важливі з яких перераховані нижче:

* ExecutionContext.

Це батьківський контекст, що включає всі інші контексти. Він не має власної поведінки, а служить тільки для запам'ятовування та передачі контексту і використовується такими компонентами .NET, як клас Task.

* SecurityContext.

Тут зберігається інформація про безпеку, зазвичай дійсна тільки в межах поточного потоку. Якщо код повинен виконуватися від імені конкретного користувача, то, скоріш за все, програма уособлює цього користувача, або це робить ASP.NET. У такому випадку відомості про уособлення зберігаються в об'єкті SecurityContext.

* CallContext.

Дозволяє програмісту ним самим зберегти певні дані, які повинні бути доступні протягом усього часу життя логічного потоку. У більшості випадків використовування цієї можливості не заохочується, але таким чином можна уникнути передачі великого числа параметрів методам в межах однієї програми, помістивши їх замість цього в контекст виклику. Клас LogicalCallContext призначений для цієї ж мети, але працює через границі доменів додатків.

Слід зазначити, що поточно-локальна пам'ять, призначена для тієї ж мети, що CallContext, для асинхронних програм не годиться, бо на час виконання тривалої операції потік звільняється і може бути використаний для інших цілей. Ваш метод може бути відновлений зовсім в іншому потоці.

Перед поновленням методу C # відновлює всі ці контексти. З відновленням пов'язані певні накладні витрати. Наприклад, асинхронна програма, в якій використовується уособлення, може працювати значно повільніше. Рекомендовано не користуватися засобами .NET, що створюють контексти, якщо це не є критичною необхідністю.

**Коли не можна використовувати await**

Оператор await можна використовувати майже в будь-якому місці методу, поміченого ключовим словом async. Однак є кілька випадків, коли використання await заборонено:

* Блоки catch і finally.

Оператор await може зустрічатися всередині блоку try, але не всередині блоків catch або finally. У блоці catch часто, а в блоці finally завжди, виключення ще перебуває у фазі розкрутки стека і пізніше може

бути порушено в блоці повторно. Якщо використовувати в цій точці

await, то стек виявиться іншим, і визначити в цій ситуації поведінку

повторного виклику виключення було б дуже складно.

* Блоки lock.

Ключове слово lock дозволяє заборонити іншим потокам доступ до об'єктів, з якими в даний момент працює поточний потік. Оскільки асинхронний метод зазвичай звільняє потік, в якому почав асинхронну операцію, і через невизначено довгий час може бути відновлений в іншому потоці, то утримувати блокування під час виконання await не має сенсу.

* Вирази LINQ-запитів.

У C # є синтаксис для спрощення запису декларативних запитів на фільтрацію, трансформацію, упорядкування та групування даних. Згодом запит може бути застосований до колекції .NET Або перетворений у форму, придатну для застосування до бази даних або іншого джерела даних. У більшості місць всередині виразу запиту вживання await неприпустимо. Пояснюється це тим, що ці місця компілятор перетворює в лямбда-вирази, і, значить, такий лямбда-вираз слід було б помітити ключовим словом async. Однак синтаксису, що дозволяє помітити ці неявні лямбда-вирази як async,просто не існує, і спроба ввести його тільки призвела б до непорозумінь.

* Небезпечний код.

Код, що позначений ключовим словом unsafe, не може містити await. Необхідність в небезпечному коді виникає дуже рідко, і зазвичай він містить автономні методи, які не потребують асинхронності. Та й все рівно перетворення, що виконуються компілятором при обробці await, в більшості випадків зробили б небезпечний код непрацездатним.

**Запам'ятовування винятків**

По ідеї, виключення в асинхронних методах повинні працювати практично так само, як в синхронних. Але через додаткові складності механізму async існують деякі тонкі відмінності. По завершенні операції в об'єкті Task зберігається інформація про те, чи завершилася вона успішно або з помилкою. Отримати до неї доступ найпростіше за допомогою властивості IsFaulted, яке дорівнює true, якщо під час виконання операції відбулося виняток. Оператор await знає про це і повторно збуджує виняток,яке зберігається в Task. Async-методи також знають про винятки. Будь-яке виключення, порушене, але не перехоплене в async-методі, поміщається в об'єкт Task, що повертається викликаючій програмі. Якщо в цей момент викликаюча програма вже чекає об'єкта Task, то виключення буде порушено в точці очікування. Таким чином, виключення передається викликаючій програмі разом зі сформованим віртуальним трасуванням стека - точно так само, як в синхронному коді.

Асинхронні методи до пори виконуються синхронно.Вище зазначено, що async-метод стає асинхронним, тільки зустрівши виклик асинхронного методу всередині оператора await. До цього моменту він працює в тому потоці, в якому викликаний, як звичайний синхронний метод. Іноді це призводить до цілком відчутних наслідків, особливо якщо є шанс, що весь ланцюжок async-методів виповнюється в синхронному режимі.

Аsync-метод призупиняється, лише дійшовши до першого await. Але навіть у цьому випадку буває, що припинення не потрібне, тому що іноді завдання Task, передана оператору await, вже завершене. Так може статися в таких ситуаціях:

* Воно була завершено вже в момент створення методом Task.
* Воно було повернуто async-методом, який так жодного разу і не дійшов до await.
* Дійсно була виконана асинхронна операція, але вона вже завершилася (можливо тому, що до моменту виклику await поточний потік робив щось ще).
* Воно була повернуто async-методом, який дійшов до await, але завдання Task, завершення якої той чекав, вже завершилось.

Саме остання можливість - коли глибоко всередині ланцюжка async-методів відбулося очікування вже завершеного завдання - дає цікавий ефект. Виконання всього ланцюжка цілком може виявитися синхронним. Пояснюється це тим, що в ланцюжку async- методів першим завжди викликається await з найглибшим рівнем вкладеності. До решти справа доходить тільки після того, як у самого глибоко вкладеного методу був шанс завершитися синхронно. Якби можна було гарантувати, що метод завжди виконується синхронно, то так, було б ефективніше відразу написати синхронний код, ніж async-метод без await. Але справа в тому, що бувають ситуації, коли метод повертає управління синхронно лише в деяких випадках. Наприклад, якщо метод кешує результати в пам'яті, то він може завершитися синхронно, коли результат вже знаходиться в кеші, а в іншому випадку буде асинхронно звертатися до мережі. Крім того, іноді має сенс повертати з методу Task або Task <T> з прицілом на майбутнє, якщо відомо, що згодом він буде переписаний і зроблений асинхронним.

**Паттерн TAP**

Патерн Task-based Asynchronous Pattern (TAP) - це пропонований Microsoft набір рекомендацій з написання асинхронних API в .NET за допомогою класу Task. У документі, написаному Стівеном Тоубом (Stephen Toub) з групи паралельного програмування в Microsoft наводяться змістовні приклади.

Дотримання цього паттерну дозволяє будувати API, що допускають використання всередині await, і, хоча додавання ключового слова async породжує методи, узгоджені з TAP, іноді корисно працювати безпосередньо з класом Task. Основна ідея TAP полягає в тому, що асинхронний метод повинен повертати об'єкт типу Task, що інкапсулює обіцянку тривалої операції завершитися в майбутньому. Без цього більш старішим асинхронним паттернам доводилося вводити додаткові параметри методу або включати в інтерфейс додаткові методи або події, щоб підтримати механізм зворотних викликів.

Об'єкт Task може містити всю інфраструктуру, необхідну для зворотних викликів, не засмічуючи код технічними деталями. У такого підходу є і ще одна перевага: оскільки механізм асинхронних викликів тепер знаходиться в Task, немає потреби дублювати його при кожному асинхронному виклику.

Це в свою чергу означає, що механізм можна зробити більш складним і потужним, зокрема відновлювати різні контексти, наприклад контекст

синхронізації, перед зворотним викликом. Нарешті, TAP пропонує єдиний API для роботи з асинхронними операціями, що дозволяє реалізувати такі засоби, як async, на рівні компілятора; при використанні колишніх патернів це було неможливо.

**Потік в якому відбувається асинхронний код**

У будь-якому async-методі якийсь код передує першому входженню ключового слова await. І, зрозуміло, сам очікуваний вираз теж містить певний код. Весь цей код виконується в викликаючому потоці. До першого await не відбувається нічого цікавого. Цю частину механізму async найчастіше розуміють неправильно. Async не планує виконання методу у фоновому потоці.

Єдиний спосіб зробити це - скористатися методом Task.Run, який спеціально призначений для цієї мети, або чимось подібним. У додатку з інтерфейсом це означає, що код до першого await працює в потоці користувальницького інтерфейсу. А в веб-додатку на базі ASP.NET - в робочому потоці ASP.

NET. Часто буває, що вираз в рядку, що містить перший await, містить ще один async-метод. Оскільки цей вираз передує першому await, воно також виконується в викликаючому потоці. Таким чином, викликаючий потік продовжує «заглублюватися »в код програми, поки не зустріне метод, який дійсно повертає об'єкт Task. Це може бути метод, який є частиною каркаса, або метод, який створює задачу-маріонетку за допомогою TaskCompletionSource. Саме цей метод і є джерелом асинхронності - всі інші async-методи просто поширюють асинхронність вгору по стеку викликів. Шлях до першої реальної точки асинхронності може виявитися досить довгим, і весь код який лежить на цьому шляху виконується в потоці користувацького інтерфейсу, а, отже, інтерфейс не реагує на дії користувача. На щастя, в більшості випадків він виконується недовго, але важливо пам'ятати, що одна лише присутність ключового слова async не гарантує відклику інтерфейсу.

**SynchronizationContext**

Клас SynchronizationContext призначений для виконання коду в потоці конкретного виду. В .NET є різні контексти синхронізації, але найбільш важливі є контексти потоку користувацького інтерфейсу, використовувані в WinForms і в WPF. Сам по собі клас SynchronizationContext не робить нічого

корисного, інтерес представляють лише його підкласи. В класі є статичні члени, що дозволяють отримати поточний SynchronizationContext і керувати ним. Поточний контекст SynchronizationContext - це властивість поточного потоку. Ідея в тому, що всякий раз, як код виконується в якомусь спеціальному потоці, ми можемо отримати поточний контекст синхронізації і зберегти його. Згодом цей контекст можна використовувати для того, щоб продовжити виконання коду в тому потоці, в якому воно було розпочато. Тому нам не потрібно точно знати, в якому потоці почалося виконання, достатньо мати відповідний об'єкт SynchronizationContext. У класі SynchronizationContext є важливий метод Post,

який гарантує, що переданий делегат буде виконуватися в правильному контексті. Деякі підкласи SynchronizationContext інкапсулюють

єдиний потік, наприклад потік користувацького інтерфейсу. Інші інкапсулюють потоки певного виду, наприклад взяті з пулу потоків, але можуть вибирати будь-який з них. Є й такі, які взагалі не змінюють потік, в якому виконується код, а використовуються тільки для моніторингу, наприклад контекст синхронізації в ASP.NET. Відомо, що код, який передує першому await, виконується в викликаючому потоці, але що відбувається, коли виконання методу поновлюється після await? Насправді, в більшості випадків він також виконуватиметься в викликаючому потоці, незважаючи на те, що в проміжку викликаючий потік міг робити щось ще. Це істотно спрощує написання

коду. Для досягнення такого ефекту використовується клас SynchronizationContext.В момент припинення методу при зустрічі оператора await поточний контекст SynchronizationContext зберігається. Далі, коли метод

поновлюється, компілятор вставляє виклик Post, щоб виконання

відновилося в запам’ятованому контексті. Метод може поновитися в потоці, відмінному від того, де був початий, при виконанні наступних

умов:

* якщо запамятований контекст SynchronizationContext інкапсулює кілька потоків, наприклад пул потоків;
* якщо контекст не має на увазі перемикання потоків;
* якщо в точці, де зустрівся оператор await, взагалі не було

поточного контексту синхронізації, як, наприклад, в консоль-

ном додатку;

* якщо об'єкт Task налаштований так, що при відновленні SynchronizationContext не використовується.

На щастя, до додатків з графічним інтерфейсом, де відновлення в тому ж потоці найбільш істотно, жодне з цих умов не застосовується, тому після await можна без побоювання маніпулювати користувацьким інтерфейсом.

**Механізм async в додатках ASP.NET**

Більшість розробників на платформі .NET пишуть веб-додатки. Механізм async відкриває нові можливості для підвищення продуктивності серверного коду.

При обробці запиту веб-сервером чутливість не грає такої ролі, як в програмах з інтерфейсом. Продуктивність веб-сервера вимірюється в термінах пропускноїздатності, затримки і сталості цих характеристик.

Асинхронний код на навантаженому веб-сервері вимагає менше потоків, ніж синхронний, обслуговуючий це ж навантаження. Кожен по- тік споживає пам'ять, а обсяг пам'яті часто є слабким місцем веб-сервера. Якщо пам'яті не вистачає, то збирач сміття запускається частіше і змушений виконувати більше роботи. Якщо недостатньо фізичної пам’яті, то починається вивантаження сторінок на диск, а у випадку, коли вивантажені сторінки скоро знову стануть необхідні, робота системи істотно сповільнюється. Можливість писати асинхронний веб-серверний код з'явилася в ASP.NET починаючи з версії 2.0, але без підтримки з боку мови це було нелегко. Більшість розробників вважало, що простіше і рентабельніше встановити додаткові сервери і балансувати

навантаження між ними. Але з виходом C # 5.0 і .NET 4.5 писати асинхронні

код стало настільки просто, що кожному розробнику є сенс скористуватися властивою йому ефективністю.

Версія ASP.NET MVC 4 і більш пізні при запуску на платформі

.NET 4.5 і вище, повною мірою підтримують патерн TAP, тому

можна використовувати async-методи. У додатку MVC найважливішим

місцем для задіяння асинхронності є контролер. Методи контролера можна помітити ключовим словом async і повертати в них значення типу Task<ActionResult>.Це рішення спирається на той факт, що для запитів, які займають тривалий час, бажано мати асинхронний API. Багато системи об'єктно-реляційного відображення (ORM) поки не підтримують асинхронних викликів, але в API, заснований на класі .NET SqlConnection, така підтримка вбудована.

**Продуктивність асинхронного коду**

Розмірковуючи про продуктивність асинхронного коду, слід порівняти його з наявними в конкретній ситуації альтернативами. На реалізацію механізму async неминуче витрачається додатковий в порівнянні з синхронним кодом час, а перемикання Async і блокуюча тривала операція між потоками збільшують затримку. Неможливо точно виміряти накладні витрати на реалізацію асинхронності. Продуктивність додатка залежить від того, чим займаються потоки, від поведінки кеша і від інших непередбачуваних факторів. Крім того,

є відмінність між використанням процесора і додаткової затримки, оскільки час операції в асинхронної системі може зростати і без споживання ЦП - через те, що запит очікує своєї черги. Зазвичай до механізму async вдаються, коли є тривала операція, яку можна виконати асинхронно, звільнивши тим самим ресурси. У програмах з інтерфейсом асинхронність дозволяє забезпечити чуйність інтерфейсу (якщо,звичайно, операція не виконується миттєво). У серверному коді компроміс не настільки очевидний, так як ми повинні вибирати між пам'яттю, зайнятої заблокованими потоками, і додатковим

процесорним часом, що витрачається на виконання асинхронних методів.

Накладні витрати на дійсно асинхронне виконання async-методу цілком залежать від того, чи необхідно перемикання потоків за допомогою методу SynchronizationContext.Post. Якщо це так, то переважну частку накладних витрат становить саме перемикання потоків в момент поновлення методу. Це означає, що поточний контекст синхронізації відіграє дуже важливу роль.

Чи доведеться платити за перемикання потоків, залежить як від кон-

тексту синхронізації викликаючого потоку, так і від контексту синхронізації потоку, в якому завершилося завдання.

* Якщо ці потоки збігаються, то викликати метод Post початкового контексту SynchronizationContext не потрібно, і метод можна відновити в потоці, де завершилася завдання, синхронно - як частина процедури завершення.
* Якщо в викликаючому потоці був контекст синхронізації, але не той, що в потоці, де сталося завершення, то потрібно викликати метод Post, що призведе до високих накладних витрат. Те ж саме відбувається, коли в потоці завершення немає контексту синхронізації.
* Якщо в викликаючому потоці не було контексту сінхронізації, як, наприклад, в консольному додатку, то ситуація визначається контекстом синхронізації потоку завершення. Якщо він існує, то .NET припускає, що цей потік важливий і планує поновлення методу в потоці з пулу. Якщо ж контексту синхронізації в потоці завершення немає або це потік, узятий з пулу, то метод поновлюється в тому ж потоці, синхронно.

Ці правила означають, що ланцюжок async-методів призведе до одного дорогого переключенню потоків - при поновленні методу з найбільшим рівнем вкладеності. Після цього контекст синхронізації вже не змінюється, і відновлення інших методів обходиться дешево. Перемикання потоків в контексті користувацького інтерфейсу виявляється однією з найдорожчих операцій. Однак в додатку з інтерфейсом синхронне виконання тривалих операцій виглядає настільки потворно, що вибору все одно немає. Якщо мережевий запит займає 500 мс, то має сенс пожертвувати ще одну мілісекунди на забезпечення чуйності інтерфейсу.

У серверному коді, наприклад в додатках ASP.NET, вибір компромісса вимагає більш ретельного аналізу. Чи має сенс переходити на асинхронний код, залежить насамперед від того, чи вистачає серверу оперативної пам'яті, тому що саме пам'яттю доводиться розплачуватися за використання великої кількості потоків. Є цілий ряд факторів, через які синхронний додаток може

споживати пам'ять швидше, ніж процесорний час.

* Викликаються тривалі операції, що займають відносно багато часу.
* Тривалі операції розпаралелюються за рахунок використання додаткових потоків.
* Є багато запитів, які вимагають запуску тривалих операцій і не можуть бути обслужені з кеша в пам'яті.
* Для породження відповіді не потрібно багато процесорного часу.

Єдиний спосіб з'ясувати, як йде справа - заміряти використання пам'яті сервером. Якщо це дійсно проблема і пам'ять виділяється надмірно великій кількості потоків, то перехід до асинхронного виконання може виявитися непоганим рішенням. При цьому буде споживатися трохи більше процесорного часу, але якщо сервера не вистачає пам'яті, а процесорних потужностей в надлишку, то з цим легко змиритися.Хоча async-методи завжди споживають більше процессорного часу, ніж синхронні, різниця насправді зовсім невелика і може виявитися непомітна на тлі інших завдань, які вирішує додаток.

Вельми типова ситуація, коли деякий метод іноді виконується довго, але в 99 відсотках випадків працює дуже швидко. Прикладом може служити кешований мережевий запит, коли більшість запитів обслуговуються з кеша. Чи використовувати в таких випадках асинхронний код, часто залежить від накладних витрат у типовому випадку, коли метод завершується синхронно, а не від витрат в одному відсотку випадків, коли дійсно потрібно асинхронна мережева операція. Ключове слово await не зупиняє метод без необхідності, коли завдання вже завершене. Метод, який містить await, в цьому випадку також завершується синхронно і повертає вже завершений об'єкт Task. Отже, весь ланцюжок async-методів відпрацьовується синхронно. Async-методи, навіть коли вони виконуються синхронно, виявляються повільнішими еквівалентних синхронних методів. І в даному випадку ми не отримуємо ніякого виграшу від звільнення ресурсів. Так званий async-метод не є асинхронним, а так званий блокуючий метод нічого не блокує. Тим не

менш переваги, які асинхронність дає в одному відсотку випадків, коли запит не можна обслужити з кешу, можуть бути настільки великі, що написання асинхронного коду виправдано.Все залежить від того, наскільки асинхронний код повільніше звичайного, коли той і інший синхронно повертають результат з кеша.Але точне вимірювання важко здійснити, так як залежить від занадто

багатьох факторів.

**Оптимізація асинхронного коду без тривалої операції**

Накладні витрати на виклик синхронно-завершуваного async-метода, які приблизно в 10 разів перевищують вартість виклику пустого синхронного методу, є наслідком декількох джерел.Велика їх частина неминуча - наприклад, виконання згенерваного компілятором коду, виклики каркаса і неможливість рызних оптимізацій через спосіб обробки виключень, виникаючих в async-методах. З тих витрат, яких можна уникнути, основна частина приходиться на виділення пам'яті для об'єктів з пула. Власне видіня пам'яті - дуже дешева операція. Але коли таких об'єктів багато, доводиться частіше запускати збирач сміття, а, об'єкт, все ще використовуваний під час збірки сміття, обходиться дорого. Механізм async спроектований так, щоб пам'ять з купи виділялась якомога рідше. Саме тому кінцевої автомат є структурою, так само як і типи AsyncTaskMethodBuilder. Вони перерозміщуються в купу, тільки якщо async-метод призупиняється. Але Task - не структура, тому завжди виділяється з купи. За цієї причини в .NET заздалегідь виділено кілька об'єктів Task,

які використовуються, коли async-метод завершується синхронно і

повертає одне з наступних типових значень:

* неуніверсальність, успішно завершений об'єкт Task;
* об'єкт типу Task <bool>, що містить true або false;
* об'єкт типу Task <int>, що містить невелике ціле число;
* об'єкт типу Task <T>, що містить null.

Якщо розробляється кеш, який повинен володіти дуже високою продуктивністю, і жоден з цих випадків непридатний, то уникнути виділення пам'яті з купи можна шляхом кешування завершеного об'єкта Task, а не просто значення. Втім, це рідко буває виправданим, оскільки, швидше за все, все одно виділяється пам'ять для об'єктів з купи в інших місцях програми. На закінчення відзначимо, що async-методи, що завершуються синхронно, вже працюють дуже швидко і подальша оптимізація є важкою. Витрачати сили на кешування об'єктів Task має сенс, тільки якщо додаток працює не так швидко, як хотілось би, а причиною є саме збірка сміття.