DOI: https://doi.org/10.33216/2222-3428-2024-27-5
УДК 004.4'236

Щербаков Є.В., Щербакова М.Є.

ПОРІВНЯЛЬНИЙ АНАЛІЗ ФУНКЦІОНАЛЬНОСТІ КОРУТИН В СУЧАСНИХ МОВАХ ПРОГРАМУВАННЯ

Корутини - це спрощені потоки-функції, які не створюють нових реальних потоків для кожної асинхронної операції, а виконуються в контексті уже раніше створених реальних потоків. Вони використовують меншу кількість потоків та ефективно застосовують їх повторно, що дозволяє знизити накладні витрати на створення та знищення додаткових потоків, а також на дороговартісне перемикання контекстів між цими потоками операційною системою. Корутини дозволяють писати асинхронний код, який виглядає як послідовний і легко читається, забезпечуючи ефективне використання ресурсів і виконання задач додатка. В останній час корутини стали широко використовуватись при розробці настільних, Android- і веб-додатків завдяки їх лаконічності та простоті використання, оскільки розробники можуть оголошувати функції, які призупиняють виконання без блокування потоку і потім продовжують роботу після завершення асинхронної операції. Зараз корутини є функціональною частиною найбільш використовуваних продуктових мов програмування, таких як Kotlin та C++, що означає, що вони інтегровані в мову синтаксично, семантично та підтримуються відповідними стандартними бібліотеками. Це робить їх застосування природним та зручним для розробників, які вже використовують відповідну мову програмування. Слід зауважити, що в клієнтській мові веб-програмування JavaScript функціональність корутин практично повністю перекривається функціями зворотного виклику, об'єктами Promise, async-функціями, оператором await та функціями-генераторами, які є базою асинхронного програмування в веб-браузерах.

Ключові слова: *асинхронне виконання, корутина, потік, подія, функція зворотного виклику, проміс, async-функція, оператор await.*

Вступ. Корутина — це функція, яка підтримує свій стан між викликами і може виконуватися асинхронно, не блокуючи потік, з якого вона була викликана [1]. Корутини можна виконувати, не турбуючись про створення складних багатозадачних конфігурацій або безпосереднє управління багатьма потоками. Через те, як вони реалізуються, корутини набагато ефективніші та менш ресурсомісткі, ніж традиційні багатопоточні опції. За допомогою корутин також формується код, який набагато легше писати, розуміти та підтримувати, оскільки вони дозволяють писати код послідовно без необхідності створення функцій зворотного виклику для обробки подій і результатів, пов'язаних з потоками.

Хоча це відносно нещодавнє доповнення до сучасних мов програмування, у корутинах немає нічого нового чи інноваційного. Функціональність корутин в тій чи іншій формі існувала у мовах програмування з 1960-х років і базувалась на моделі, відомій як CSP (Communicating Sequential Processes) [2]. Різниця між послідовностями виконання коду при викликах звичайних функцій і корутин схематично показана на рис. 1.

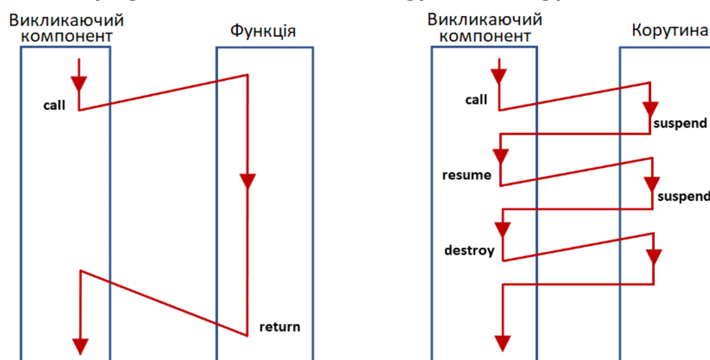

Рисунок 1 - Переходи виконання коду в додатках при викликах функцій та корутин

Якщо функцію можна лише викликати на виконання та повернутися назад, то корутину можна викликати, призупиняти та відновлювати виконання, а також завершити призупинену корутину. Корутини дають можливість писати асинхронний код у послідовному стилі, без необхідності використання колбеків або ланцюжків викликів. Це спрощує читання та підтримку коду, роблячи його більш логічним та структурованим. Крім того, корутини пропонують зручні оператори для комбінування та управління послідовностями асинхронних операцій, такі як async, await, withContext та інші.

Потоки та корутини. Коли додаток запускається вперше, операційна система створює єдиний головний потік, в якому за замовчуванням будуть працювати всі компоненти інтерфейсу користувача додатка [3]. Будь-які інші компоненти, які запускаються в додатку, за замовчуванням також виконуються в головному потоці. Але будь-який код, який виконує тривалу по часу задачу в головному потоці, призведе до того, що весь додаток буде заблокований, доки ця задача не буде виконана. Це зазвичай призводить до того, що операційна система відображає користувачеві попередження типу «Додаток не відповідає». Це далеко не бажана поведінка для будь-якого додатка. Тому до появи корутин, для виконання таких довготривалих задач, щоб уникнути блокування головного потоку, створювалися додаткові потоки.

Проблема з потоками полягає в тому, що вони є обмеженим ресурсом і дорогим з точки зору можливостей центрального процесора та системних накладних витрат. У фоновому режимі виконується багато роботи зі створення, диспетчеризації та знищення потоків. Хоча сучасні CPU можуть запускати велику кількість потоків, фактична кількість потоків, які можуть бути запущені паралельно в будь-який момент часу, обмежена кількістю ядер CPU (хоча нові CPU мають 8 або більше ядер, більшість сучасних комп'ютерів, в тому числі пристрої Android, містять CPU з 4 ядрами). Коли потрібно більше потоків, ніж ядер CPU, операційна система виконує диспетчеризацію потоків, щоб вирішити, як виконання цих потоків розподілити між доступними ядрами. Щоб уникнути цих накладних витрат, замість того, щоб запускати новий потік для кожної корутини, а потім знищувати його, коли корутина завершує роботу, середовище виконання додатка підтримує пул активних потоків і управляє тим, як корутини призначаються цим потокам. Коли активна корутина призупиняється, її стан зберігається середовищем виконання, а інша корутина поновлює своє виконання замість неї. Коли корутина поновлюється, вона просто записує свій стан до фрейму наявного незайнятого потоку в пулі, щоб продовжити виконання, доки вона не завершиться або не буде призупинена. Використовуючи цей підхід, обмежена кількість потоків ефективно виконує асинхронні задачі з потенціалом для виконання великої кількості одночасних задач без внутрішнього зниження продуктивності, яке могло б виникнути при стандартному використанні багатопоточності. Чесно кажучи, корутини не обов'язково мають відношення до багатопоточності; натомість вони просто забезпечують спосіб призупинення та відновлення виконання функцій. Звичайно, якщо необхідно, корутини можна використовувати в багатопоточному середовищі так само добре.

Робота з корутинами в додатках на Kotlin. Найбільш широкі можливості по реалізації конкурентного програмування в додатках забезпечують корутини мови Kotlin [4] .

При використанні мови Kotlin кожна з корутин повинна виконуватись в певній області видимості (coroutine scope), що дозволяє управляти ними як групами, а не як індивідуальними сутностями. Це особливо важливо під час скасування та очищення корутин та забезпечення того, щоб корутини не "протікали" (іншими словами, продовжували працювати у фоновому режимі, коли вони більше не потрібні додатку). Призначивши корутини до певної області видимості, вони, наприклад, можуть бути завершені масивом, коли більше не потрібні.

Kotlin і Android надають декілька вбудованих областей видимості, а також можливість створювати спеціальні області видимості за допомогою класу CoroutineScope. Вбудовані області видимості можна узагальнити наступним чином:

- GlobalScope – ця область використовується для запуску корутин верхнього рівня, які пов'язані з усім життєвим циклом додатка. Оскільки це може призвести до того, що корутини в цій області видимості можуть продовжувати працювати, коли вони не потрібні (наприклад, коли завершується активність), використання цієї області видимості не рекомендується в додатках Android. Вважається, що корутини, що працюють у GlobalScope, використовують неструктуровану конкурентність (unstructured concurrency).
- ViewModelScope – надається спеціально для використання в об'єктах ViewModel при використанні компонента ViewModel з архітектури Jetpack. Корутини, запущені в цій області видимості з об'єкта ViewModel, автоматично завершуються системою виконання Kotlin, коли відповідний об'єкт ViewModel знищується.
- LifecycleScope - кожен володар життєвого циклу пов'язує з ним LifecycleScope. Ця області видимості скасовується, коли відповідний володар життєвого циклу знищується, що робить її особливо корисною для запуску корутин із композиційних елементів і активностей.

Для більшості задач найкращий спосіб отримати доступ до області видимості корутини з композиційного елемента — це зробити виклик функції rememberCoroutineScope() наступним чином:

```
val coroutineScope = rememberCoroutineScope()
```

Змінна coroutineScope тепер містить диспетчер, який може використовуватися для запуску корутин, і на нього слід посилатися кожного разу, коли запускається корутина, якщо вона має бути включена в область видимості. Всі запущені корутини в області видимості можуть бути скасовані за допомогою виклику методу cancel() об'єкта області видимості:

```
coroutineScope.cancel()
```

Диспетчер (dispatcher) визначає контекст виконання корутини, зокрема, на якому потоці чи потоках буде виконуватись код корутини. Вибір правильного диспетчера важливий для ефективного використання ресурсів і запобігання блокування інтерфейсу користувача. Існує кілька вбудованих диспетчерів, доступних у Kotlin:

- Dispatchers.Main – запускає корутину в основному потоці та підходить для корутин, яким потрібне внесення змін до UI, і як опція загального призначення для виконання легких задач.

- Dispatchers.IO – рекомендується для корутин, які виконують операції з мережею, диском або базою даних.
- Dispatchers.Default – призначений для завдань із інтенсивним використанням CPU, таких як сортування даних або виконання складних обчислень.

Диспетчер відповідає за призначення корутин відповідним потокам, а також за призупинення та відновлення корутини протягом її життєвого циклу. Наприклад, наступний код запускає корутину за допомогою диспетчера введення-виведення:

```
coroutineScope.launch(Dispatchers.IO) {
    performSlowTask()
}
```

Окрім попередньо визначених диспетчерів, також можна створювати диспетчери для кастомних пулів потоків. В Kotlin коди корутин містяться в особливому типу функцій – suspend-функціях. Вони оголошуються за допомогою ключового слова suspend, яке вказує Kotlin, що функція може переводитись в паузу, та її виконання відновиться пізніше, дозволяючи довготривалим обчисленням виконуватися без блокування основного потоку додатка. Нижче наведено приклад оголошення функції призупинення:

```
suspend fun performSlowTask() {
    println("performSlowTask: перед паузою")
    delay(5000) // моделює довготривалу задачу
    println("performSlowTask: після паузи")
}
```

Зводять разом усі розглянуті компоненти та запускають корутини, щоб вони почали виконуватися, шість побудовників корутин (coroutine builders):

- launch – запускає корутину без блокування поточного потоку і не повертає результат викликаючому компоненту. Цей побудовник треба використовувати під час виклику функції призупинення з традиційної функції та коли не потрібно обробляти результати корутини (іноді їх називають корутинами «запустив і забув»).
- async – запускає корутину і дозволяє викликаючому компоненту чекати результат за допомогою функції await() без блокування поточного потоку. Треба використовувати async, якщо є кілька корутин, які потрібно виконувати паралельно. Побудовник async можна використовувати лише в межах іншої функції призупинення.
- withContext – дозволяє запускати корутину в контексті, відмінному від того, що використовується батьківською корутиною. Корутина, яка виконується в контексті Main, може, наприклад, запустити дочірню корутину в контексті Default за допомогою цього побудовника. Побудовник withContext також надає корисну альтернативу async при поверненні результатів із корутини.
- coroutineScope – ідеально підходить для ситуацій, коли функція призупинення запускає декілька корутин, які виконуватимуться паралельно, і де певну дію потрібно виконати лише тоді, коли всі корутини досягнуть завершення. Якщо ці корутини запускаються за допомогою побудовника coroutineScope, викликаюча функція не виконає return, доки не будуть завершені всі дочірні корутини. При використанні coroutineScope збій (failure) у будь-якій із корутин призведе до скасування всіх інших корутин.
- supervisorScope – подібно до coroutineScope, описаного вище, за винятком того, що збій в одній дочірній корутині не призводить до скасування інших корутин.
- runBlocking - запускає корутину та блокує потік, доки корутина не досягне завершення. Це, як правило, протилежне тому, що потрібно від корутин, але корисно для тестування коду та інтеграції застарілого коду і бібліотек. В іншому разі цього слід уникати.

Кожен виклик побудовника корутин, наприклад launch або async, повертає об'єкт задачі (Job), який, у свою чергу, можна використовувати для відстеження та управління життєвим циклом відповідної корутини. Подальші виклики побудовника всередині корутини створюють нові об'єкти Job, які стають дочірніми елементами безпосереднього батьківського об'єкта Job, утворюючи дерево взаємозв'язків батьківських і дочірніх об'єктів Job, де скасування батьківського Job рекурсивно скасує всі його дочірні об'єкти Job. Статус корутини можна визначити за допомогою доступу до властивостей isActive, isCompleted і isCancelled асоційованого об'єкта Job. Ця ієрархічна структура Job разом із областями видимості корутин формує основу структурованої конкурентності, метою якої є забезпечення того, щоб корутини не виконувались довше, ніж потрібно, без необхідності вручну зберігати посилання на кожну корутину.

Функціональні можливості корутин в С++. В мові **С++** будь-яка функція з одним із таких ключових слів у своєму тілі є корутиною [5]:

- co_await - призупиняє корутину в очікуванні завершення обчислення. Її виконання відновлюється після завершення обчислення.
- co_return - повертає управління з корутини (в корутині заборонено використовувати звичайну інструкцію return). Після цього виконання корутини відновити неможливо.
- co_yield - повертає значення із корутини назад викликаючому компоненту та призупиняє корутину. Згодом повторний виклик корутини продовжить її виконання з точки, в якій її було призупинено.

В термінології С++ існує два типи корутин: стекові та безстекові. Стекову корутину можна призупиняти на любій глибині вкладених викликів. З іншого боку, безстекову корутину можна призупиняти лише на

верхньому стековому фреймі потоку. Коли безстекова корутина призупиняється, зберігаються лише локальні та тимчасові змінні функції; стек викликів не зберігається. Таким чином, використання пам'яті для безстекових корутин є мінімальним, що дозволяє тисячам або навіть мільйонам корутин працювати одночасно. C++ підтримує лише безстековий варіант корутин.

Корутини можна використовувати для реалізації асинхронних операцій, застосовуючи синхронний стиль програмування. Варіанти використання корутин включають в себе наступне:

- функції – генератори;
- асинхронні операції введення – виведення;
- ліниві обчислення;
- додатки, керовані подіями;
- машини станів (кінцеві автомати).

На жаль, хоча всі низькорівневі засоби мови C++ доступні для написання користувальницьких корутин, можливостей вбудованих в мову високорівневих корутин небагато. Перелічені вище можливості по роботі з корутинами (стандарт C++ 20) радше визначають тільки каркас для роботи з корутинами. Тому, для зручності розробників, були створені бібліотеки класів для поширених видів корутин: cppcoro, concurrencpp, libcoro, QCoro та інші. У відповідності зі стандартом C++ 23 в стандартну бібліотеку мови була введена одна корутина високого рівня, функція-генератор std::generator. Генератор забезпечує механізм для перемикання одного потоку між генеруванням результатів і обробкою цих результатів без залучення кількох потоків.

Можливості асинхронного програмування в JavaScript. Хоча в JavaScript термін «coroutine» не використовується, в клієнтській мові веб-програмування JavaScript функціональність корутин практично повністю перекривається такими сутностями, як функції-генератори, об'єкти Promise, async-функції та оператори await, які є базою асинхронного програмування в веб-браузерах.

В основних браузерах JavaScript реалізується потужним движком Chrome V8 від Google [6]. Цей движок, написаний на C++, відповідає за компіляцію та виконання коду JavaScript, управління ресурсами та розподіл пам'яті, забезпечуючи високу продуктивність і гнучкість. Оскільки V8 є віртуальною машиною, це означає, що він абстрагує архітектуру і операційну систему комп'ютера від фактичного коду JavaScript, імітуючи універсальне середовище виконання. Таким чином, V8, а значить і програми на JavaScript, можна використовувати на Windows, Linux, Mac і Android майже безперешкодно, що дозволяє розробникам розгортати свої додатки в будь-якому середовищі виконання, не надто турбуючись про базові комп'ютерні засоби.

JavaScript є синхронною за замовчуванням, що означає, що код виконується послідовно в одному потоці. Тому для обробки подій раніше і тепер використовуються функції зворотного виклику — прості функції, які передаються в якості аргументів іншим функціям та виконуються лише тоді, коли відбувається відповідна подія, наприклад:

```
setTimeout(() => { /* Цей код буде запущений на виконання через 2 секунди*/ }, 2000);
```

Функції зворотного виклику добре працюють при обробці асинхронних викликів, але з задіянням кожної нової функції зворотного виклику з попередньої додається рівень їх вкладеності, що може призводити до так званого пекла зворотних викликів (callback hell), коли в якійсь частині додатка реалізується багато зворотних викликів, які потім дуже складно відслідкувати і супроводжувати.

Полегшують вирішення проблем, що виникають при застосуванні подій і функцій зворотного виклику, об'єкти Promise, введені в JavaScript специфікацією ES6 (ECMAScript 2015) [7]. Об'єкт Promise представляє можливий успіх або невдачу асинхронної операції та її кінцевий результат, можливо, відкладений. Іншими словами, проміс — це об'єкт, до якого приєднуються функції зворотного виклику замість того, щоб передавати їх безпосередньо в якості аргументів другим функціям. Проміси створюються за допомогою конструктора, синтаксис виклику якого при використанні стрілочної функції наступний:

```
let promise = new Promise((resolve, reject) =>{
// Асинхронні обчислення: в разі успіху викликається resolve(результат);
// в разі помилки - reject(помилка)
}):
```

Конструктору Promise() в якості єдиного аргументу передається функція-виконавець, яка автоматично викликається для ініціювання асинхронного обчислення або відкладеної дії та сигналізації про завершення. Функція-виконавець в якості аргументів приймає дві функції зворотного виклику: resolve та reject. Код функції-виконавця в кінці роботи може викликати функцію resolve, щоб вказати, що робота пов'язаного об'єкта Promise виконана. Аргумент, що передається функції resolve, повинен бути результатом асинхронної операції або відкладеної дії і може бути або фактичним значенням, отриманим в результаті виконання асинхронної операції, або іншим об'єктом Promise, який згенерує фактичне значення, якщо буде виконаний. Код виконавця в кінцевому підсумку також може викликати функцію reject, щоб вказати, що відповідний об'єкт Promise відхиляється і ніколи не буде виконаний. Аргумент, який передається функції reject, використовується як причина відхилення об'єкта Promise (зазвичай, це об'єкт Error).

Будь-який об'єкт Promise може знаходитись в одному з трьох взаємовиключних станах: fulfilled (виконано) - якщо викликана функція resolve з аргументом, який не є об'єктом Promise, або без аргументів, асинхронна операція вважається виконаною; rejected (відхилено) - якщо викликана функція reject або виявлено виключення всередині виконавця, асинхронна операція вважається відхиленою; pending (в очікуванні) - якщо жодна з функцій - resolve і reject - ще не викликана, асинхронна операція знаходиться в очікуванні.

По завершенні обчислень асинхронний код переводить об'єкт Promise в стан fulfilled або rejected. При цьому автоматично викликаються відповідні обробники подій в основній програмі. Метод then() забезпечує підключення обробників подій до уже існуючого об'єкта Promise, які потім виконуються асинхронно:

```
promise.then(onFulfilled, onRejected);
```

Кожен виклик then() насправді створює і повертає ще один об'єкт Promise, який обробляється лише тоді, коли попередній був виконаний або відхилений. Таким чином, для управління вкладеними асинхронними діями будуть створюватись ланцюжки об'єктів Promise.

Ланцюжки промісів можуть бути надто довгими, тому в специфікації ES8 (ECMAScript 2017) в JavaScript були введені асинхронні (async) функції та оператор await, які дозволяють в більшості випадків уникнути довгих ланцюжків .then(). Щоб зробити функцію або метод асинхронним, треба додати ключове слово async перед визначенням цієї функції чи методу, наприклад:

```
async function doSomething(…) {…}
```

Асинхронні функції відрізняються від синхронних наступним:

- значення, яке повертається функцією, завжди є промісом;
- збуджені виключення - це відхилення проміса;
- можна використовувати оператор await;
- можна використовувати цикл for-wait-of.

Оператор await використовується для очікування виконання або відхилення проміса. Він може використовуватися лише всередині асинхронної функції і змушує її код чекати, доки проміс не поверне результат. Оператор await повідомляє движку JavaScript, що треба призупинити виконання поточної функції, доки указаний в ньому проміс не буде вирішений, після чого відновлюється виконання функції з поверненням значення, яке містилося в промісі.

Таким чином, хоча в JavaScript при викликах функцій фрейми з їх локальними змінними розміщуються в динамічній пам'яті, а не в стековій пам'яті активного потоку, асинхронні функції JavaScript реалізують ту ж функціональність, що і корутини мови Kotlin або стекові корутини мови C++.

Здатність функцій призупинятися, а потім знову відновлювати виконання, не є винятковою для асинхронних функцій. JavaScript також має функції, які називається функціями-генераторами, які схожі на асинхронні функції, але не використовують проміси. Генератор — це функція, що повертає ітератор. Функції–генератори позначаються символом * після ключового слова function та використовують інструкцію yield в тілі функції, наприклад:

```
function* powers(n) {
   for (let current = n;; current *= n) {
      yield current;
   }
}
```

Функція–генератор призупиняє виконання після кожної інструкції yield і не виконує нічого іншого, доки не буде викликаний метод next() ітератора. Інструкції yield можуть з'являтися лише безпосередньо у самій функції-генераторі, а не у внутрішній функції, яка в ній визначається. Тому стан, який генератор зберігає при призупиненні виконання інструкцією yield, це лише його локальні змінні та вказівник на наступну інструкцію.

Таким чином, якщо в функціях-генераторах інструкції yield в якості виразів будуть використовуватись проміси, то такі генератори мови JavaScript будуть забезпечувати таку ж функціональність, як і безстекові корутини мови C++.

Висновки. Корутини і подібні їм механізми корпоративної багатозадачності дають можливість реалізувати асинхронне виконання програм додатків, уникаючи при цьому використання багатопоточних схем з їх внутрішньою складністю і великими накладними витратами на переключення процесорів між потоками виконання. Вважається, що в однопоточних додатках з використанням корутин споживаний обсяг пам'яті істотно нижче, загальна пропускна здатність вище, реактивність системи під навантаженням краще, а модель програмування простіше. В даний час синтаксичні і семантичні елементи роботи з корутинами включаються як у відносно нові мови програмування, такі як Kotlin, яка зараз є основною мовою розробки Android-додатків, так і добавляються в широко використовувані раніше мови програмування, прикладом яких може бути мова системного програмування C++. Відразу декілька варіантів кооперативної багатозадачності, в якійсь мірі функціонально рівнозначних корутинам, використовуються в клієнтській мові веб-програмування JavaScript, причому проміси, async-функції та await-вирази були введені в мову тільки декілька років тому.

Найбільш широкі можливості по реалізації конкурентного програмування з використанням корутин в додатках забезпечує мова Kotlin. При використанні цієї мови кожна з корутин повинна виконуватись в певній області видимості (GlobalScope, ViewModelScope, LifecycleScope), що забезпечує, окрім індивідуального, групове управління корутинами. Можна використовувати різні диспетчери корутин, такі як Dispatchers.IO, Dispatchers.Main, Dispatchers.Default, що дає можливість управляти, на якому потоці буде виконуватись корутина. В Kotlin коди корутин містяться в suspend-функціях, які дозволяють призупиняти виконання до завершення асинхронної операції без блокування відповідного потоку додатка. Зводять все разом та запускають корутини на виконання побудовники корутин launch, async, withContext та інші.

Тільки на базовому рівні забезпечується підтримка функціональності корутин в мові C++. Відповідно до стандартів C++ 20 та C++ 23 ця мова разом зі стандартною бібліотекою прямо забезпечує кодування і роботу

лише безстекових корутин та функцій-генераторів. Безстекову корутину можна призупиняти лише на верхньому фреймі стека потоку.

Якщо при розробці веб-додатка або сайту на JavaScript виявляється, що є тільки однорівнева обробка асинхронних операцій, можна успішно використовувати функції зворотного виклику, оскільки код в цьому випадку буде залишатись цілком керованим і зрозумілим. Коли ж в додатку існує кілька ланцюжкових (або залежних) асинхронних операцій, тоді, щоб уникнути пекла зворотних викликів, краще використовувати проміси, які є чудовим інструментом для підтримки організованості та передбачуваності асинхронних операцій. Механізм async/await також є чудовим інструментом, за допомогою якого можна для асинхронних операцій писати майже «синхронний» код і легше управляти ланцюжками об'єктів Promise.

Література

1. Stroustrup B. A Tour of C++. Third Edition / B. Stroustrup. - Hoboken, NJ: Pearson Education, 2023. - 312 p.
2. Tanenbaum A. S. Modern Operating Systems, 5th Edition / A. S. Tanenbaum, H. Bos. - Hoboken, NJ: Pearson Education, 2024. - 1185 p.
3. Grimm R. Concurrency with Modern C++ [Електронний ресурс] / Leanpub. - Режим доступу: https://leanpub.com/concurrencywithmodernc
4. Kotlin coroutines on Android [Електронний ресурс] / Android developers. - Режим доступу: https://developer.android.com/kotlin/coroutines
5. Gregoire M. Professional C++. Sixth Edition / M. Gregoire. - Hoboken, NJ: John Wiley & Sons, Inc., 2024. - 1379 p.
6. Frisbie M. Professional JavaScript for Web Developers. Fifth Edition/ M. Frisbie. - Hoboken, NJ: John Wiley & Sons, Inc., 2024. - 1105 p.
7. Zakas N. C. Understanding JavaScript Promises [Електронний ресурс] / Leanpub. - Режим доступу: http://leanpub.com/understanding-javascript-promises

References

1. Stroustrup B. A Tour of C++. Third Edition / B. Stroustrup. - Hoboken, NJ: Pearson Education, 2023. - 312 p.
2. Tanenbaum A. S. Modern Operating Systems, 5th Edition / A. S. Tanenbaum, H. Bos. - Hoboken, NJ: Pearson Education, 2024. - 1185 p.
3. Grimm R. Concurrency with Modern C++ [Electronic resource] / Leanpub. - Access mode: https://leanpub.com/concurrencywithmodernc
4. Kotlin coroutines on Android [Electronic resource] / Android developers. - Access mode: https://developer.android.com/kotlin/coroutines
5. Gregoire M. Professional C++. Sixth Edition / M. Gregoire. - Hoboken, NJ: John Wiley & Sons, Inc., 2024. - 1379 p.
6. Frisbie M. Professional JavaScript for Web Developers. Fifth Edition/ M. Frisbie. - Hoboken, NJ: John Wiley & Sons, Inc., 2024. - 1105 p.
7. Zakas N. C. Understanding JavaScript Promises [Electronic resource] / Leanpub. - Access mode: http://leanpub.com/understanding-javascript-promises

Coroutines are simplified threads-functions that do not create new real threads for each asynchronous operation, but are executed in the context of previously created real threads. They use fewer threads and efficiently reuse them, that reduces the overhead of creating and destroying additional threads, as well as the expensive context switching between these threads by the operating system. Coroutines allow you to write asynchronous code that looks consistent and is easy to read, ensuring efficient use of resources and execution of application tasks. Recently, coroutines have become widely used in desktop, Android, and web application development due to their brevity and ease of use, as developers can declare functions that suspend execution without blocking the thread and then continue after the asynchronous operation completes. Coroutines are now a functional part of the most widely used production programming languages, such as Kotlin and C++, which means that they are integrated into the language syntactically, semantically, and supported by the corresponding standard libraries. This makes their application natural and convenient for developers who already use the corresponding programming language. It should be noted that in the JavaScript client web programming language, the functionality of coroutines is almost completely covered by callback functions, Promise objects, async functions, await expressions and generator functions, which are the basis of asynchronous programming in web browsers.

***Keywords**: asynchronous execution, coroutine, thread, event, callback function, promise, async function, await operator.*

Щербаков Є.В. – к.т.н., доцент, доцент кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету ім. В. Даля, e-mail: gkvarc@gmail.com

Щербакова М.Є. – к.т.н., доцент, доцент кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету ім. В. Даля, e-mail: m.shcherbakova432@gmail.com