Многопоточность – это специфика выполнения программы, основанного на параллельных вычислениях

Важный принцип многопоточности – два и более потока

Поток – это последовательно выполнение набора команд (инструкций). Поток имеет собственную память (стек), которая содержит очередь вызовов функций, примитивные значения, а также делит общую память с другими потоками.

Потоки объединены в один процесс

Процесс – это контейнер, который содержит независимую от других процессов память, может иметь один и несколько потоков. Процесс является верхней точкой запуска программы (т.е. при запуске чего-либо обязательно должен создаться процесс).

Для работы процесса и выполнения кода используется **центральный процессор**. Обычно используется **вытесняющая многозадачность** (когда процессы сменяются друг с другом через определенное время). Если центральный процессор имеет два и более ядер, то вычисления могут выполняться и параллельно друг другу.

Параллельность может быть двух видов: **реальная параллельность** и **псевдо-параллельность** (зависит от кол-ва ядер в процессоре).

Помимо параллельности задач мы можем использовать **асинхронность**

Асинхронность – выполнение инструкций каждой задачи поочередно (очередь может определяться ЯП, определенной реализацией и выполняемым алгоритмом).

Реализация многопоточности и асинхронности на языке Kotlin

Язык Kotlin (если речь не об Native) в основе своей работает через JVM, поэтому многие правила использования многопоточности применимы и в Java.

Так, для работы с отдельным потоком мы используем функцию thread, которая создает отдельный поток с Runnable – интерфейсом с одной функцией run, куда мы записываем участок кода, который мы хотим обработать.

Если же у нас большое приложение и мы не хотим создавать под каждую задачу свой поток (да и не сможем, так как это в конечном счете приведет к

OutOfMemory) мы можем использовать ThreadPoolExecutor – контейнер, который предоставляет переиспользуемые потоки в месте вызова.

Это основные механизмы, которые достались языку Kotlin от Java, но они имеют большое кол-во минусов

1. Трудозатратность – создание потока это сложная операция, где нам нужно выделять под него отдельные ресурсы (как минимум, отдельный стек памяти). Хотя, конечно, у нас есть решение этой проблемы в виде ThreadPoolExecutor (где потоков определенное кол-во и памяти должно хватить), но это все равно не решает проблему с передачей ресурсов (вместо выполнения многих задач, поток вынужден решать одну)
2. Сложная обработка цепочек задач – для обработки цепочек задач один Thread не подойдет, мы должны уметь переключаться между потоками во время исполнения программы и при получении результата, обрабатывать ошибки и т.д. Для решения подобных задач было придумано много фреймворков и подходов (например, RxJava и соответствующее ему реактивное программирование).
3. Отслеживание жизненного цикла цепочек – RxJava, как и потоки не умеют отслеживать жизненный цикл элемента по стандарту, для работы с ним мы должны сами дописывать необходимые под наши задачи механизмы что, конечно, усложняет жизнь

Основное решение языка Kotlin стали **coroutines (сопрограммы)**

Coroutines (сопрограммы) – блок кода, исполняемый асинхронно с другими корутинами в отдельной области выполнения (CoroutineScope)

Корутины входят в библиотеку **kotlinx.coroutines** и поддерживается языком Kotlin на уровне компиляции. Основная идея корутин – это покрытие участка кода специальными метками, для отслеживания прогресса прохождения области кода (так называемая state-machine). Специальный интерфейс контейнер, который позволяет сохранять предыдущее состояние корутины называется Continuation. Он сохраняет под собой индекс прошедшей операции (и другие промежуточные данные), позволяя запускать код там, где мы остановились.

Корутины бывают двух видов, одни объявляются через специальные строительные функции (suspendableCoroutine, suspendableCancellationCoroutine и др.), а другие через ключевое слово suspend при объявлении сигнатуры функции.

Различие состоит в том что внутри строительных блоков мы знаем об исполняемом контексте suspend-блока, а suspend-функции же могут переиспользоваться в разных ситуациях и они знают об контексте уже по факту вызова (во время компиляции).

Для вызова любого suspend-блока (что в первой, что во второй ситуации) нам необходимо сначала создать среду исполнения корутин.

За среду исполнения отвечает класс CoroutineScope – это контейнер, обладающий несколькими свойствами:

1. Есть CoroutineContext – это не какой-то отдельный класс, а скорее набор свойств, используемых для исполнения программы (содержит имя, обработчик ошибок, Dispatcher с пулом потоков и другое)
2. Structured Concurrency – замечательное свойство, позволяющее настраивать цепочки исполнения корутин и контролировать жизненный цикл так как нам нужно
3. Жизненный цикл – после создания Scope мы можем легко контролировать жизненный цикл, отменяя все запущенные работы в месте окончания программы (через метод cancel) или наоборот создавать отдельные точки жизненного цикла, создавая вложенные scopes (через методы launch, async)

CoroutineScope является обязательным условием для запуска любой корутины, но также нужно учесть что она лишь контролирует жизненный цикл, но сама не проводит вычисления. За нее этим занимается Dispatcher – пул потоков.

Dispatcher может быть нескольких видов:

1. Dispatcher.IO – контролирует операции ввода-вывода. Кол-во потоков больше чем число реальных потоков, поэтому должно использоваться не при вычислениях, а при операциях ожидания (чем IO операции и являются)
2. Dispatcher.Default – содержит реальное кол-во потоков в системе, благодаря чему может использоваться в сложных/долгих вычислениях
3. Dispatcher.Unconfined – говорит системе что в качестве потока мы выбираем тот, который сейчас используется/использовался при вызове корутины
4. Dispatcher.Main – главный поток в системе, по-умолчанию такого потока нет (за исключением, например, Android), так что мы должны назначить его сами

5. Dispatcher.Main.immediate – аналогично Dispatcher.Main, только тут мы говорим системе что наша операция должны идти впереди всей очереди других операций на этом потоке

При исполнении корутин мы можем использовать любые из этих диспатчеров для любых ситуаций (или создавать свои), но важно учесть, не каждый из них подойдет для любой работы (например, создавая много задач на главном потоке мы можем перегрузить систему).

При использовании Scope мы можем создавать разнообразное кол-во задач и подзадач, применяя свойство Structured concurrency, но что произойдет если одна из задач придет с ошибкой? Мы могли бы ожидать, что если задача придет с ошибкой, то она должна отменить свои подзадачи или нет? Или она вообще ничего не отменяет и Scope продолжит свою работу?

Одно из важнейших свойств Structured Concurrency – это пробрасывание результата наверх к другим корутинам, пока она не дойдет до верха.

Это свойство вводит в ступор и оставляет множество вопросов, как поступить если я не хочу чтобы ошибка отменяла смежные задачи? И как вообще обрабатывать ошибки, ведь если ошибки пробрасываются наверх, то стандартный механизм try/catch не должен работать, или должен?

Как сделать так, чтобы отмена смежных задач не происходила при отмене одной из них

У Kotlin есть два варианта работы (на самом деле 3, с учетом Deferred) – это Job и SupervisorJob

Если по Job представляться не надо, работа – это работа, то с SupervisorJob вызывает вопросы. Для начала, что вообще такое Supervisor?

Супервайзер – профессия, где человек организовывает, координирует и контролирует работу людей, через которых выстраивается «коммуникация» потребителя с товаром.

Переводя на наш язык, супервайзер реализует работу между корутинами, введенными в его подчинение

```
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Job
import kotlinx.coroutines.delay
import kotlinx.coroutines.isActive
import kotlinx.coroutines.launch

val scope = CoroutineScope(Job())

fun startCoroutineExample() {
```

```
    scope.launch {
        println("Start work 1")
        delay(3000)
        throw Exception("Exception 1")
    }
    scope.launch {
        println("Start work 2")
        delay(5000)
        println("Success work 2")
    }
}

fun main() {
    startCoroutineExample()
    while (scope.isActive);
}
```

Этот пример показывает что стандартная Job не контролирует работу корутин и из-за этого когда первая корутина ломается, вторая не продолжает работу

```
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Job
import kotlinx.coroutines.SupervisorJob
import kotlinx.coroutines.delay
import kotlinx.coroutines.isActive
import kotlinx.coroutines.launch

val scope = CoroutineScope(SupervisorJob())

fun startCoroutineExample() {
    scope.launch {
        println("Start work 1")
        delay(3000)
        throw Exception("Exception 1")
    }
    scope.launch {
        println("Start work 2")
        delay(5000)
        println("Success work 2")
    }
}

fun main() {
    startCoroutineExample()
    while (scope.isActive);
}
```

Здесь же благодаря SupervisorJob мы спокойно можем продолжать работу.

Главное запомнить, что SupervisorJob контролирует работу СМЕЖНЫХ корутин и подкорутин

```
val scope = CoroutineScope(SupervisorJob())

fun startCoroutineExample() = scope.launch {
    launch {
        println("Start work 1")
        delay(3000)
        throw Exception("Exception 1")
```

```
    }
    launch {
        println("Start work 2")
        delay(5000)
        println("Success work 2")
    }
}

fun main() {
    startCoroutineExample()
    while (scope.isActive);
}
```

Этот пример кода хоть и использует супервайзер, но не задействует его. Из-за этого вторая работа не выполнится.

Обработка ошибок в блоках

Так как корутин-блоки имеют свойство пробрасывания, работа с ошибками обрабатывается особыми образами.

Во-первых, необходимо уяснить, какие корутин-задачи пробрасываются, а какие распространяются наверх

И launch и async функции не пробрасывают исключения, которые возникают внутри. Вместо этого, они РАСПРОСТРАНЯЮТ их вверх по иерархии корутины. За исключением async, который находится на верхнем уровне.

Корутины запускаемые через async становятся верхнеуровневыми, и обрабатывают исключения по-другому, чем вложенные async:

async верхнего уровня скрывает обработку исключения в объекте Deffered, который возвращает билдер. Объект выбрасывает нормальное исключение только при вызове метода await().

Общие проблемы при использовании многопоточности

При использовании многопоточности стоит учесть, что существуют шаблонные проблемы, вот одни из них

1. Deadlock – мертвая блокировка, характеризуется тем, что у нас два потока ожидают снятия блокировки друг с друга, чтобы продолжить работу.
2. Livelock – похож на deadlock, но характеризуется тем что два потока продолжают выполнять работу, при этом условие на разблокировку потока никогда не наступает и потоки продолжают свою работу
3. Race conditions – гонка, это когда два и более потока получают доступ к критической секции, пытаются ее поменять так как сами хотят

(прочитать или записать данные), а по итогу данные получаются случайными (будут зависеть от итогового исполнения программы).

Решение проблем, связанных с многопоточностью

Для решения проблем с многопоточностью в Kotlin стоит учесть, что для разных видов выполнения работы (через потоки или корутины) есть свои инструменты.

Так, для корутин не подойдет метод синхронизации, как у стандартных потоков. Это связано с тем, что при исполнении блок кода в корутине может передать управление другому месту, из-за чего поток, который находится в синхронизируемом режиме не сможет использоваться в другом месте, также стоит учесть что для одного блока может использоваться несколько потоков, что тоже делает механизм синхронизации невалидным

Поэтому для корутин применяются CAS (Check-and-swap) операции, Atomic-переменные и Mutex. Также можно использовать хак с withContext(Dispatchers.Main), так как тогда здесь над операцией будет работать всего один поток