Reksoft

Kotlin Coroutines. Устройство и возможности использования.

Соловьев Андрей,
разработчик Reksoft

9.10.2024

Кто я такой?

Reksoft

- Java разработчик
- Работаю в финансовой сфере
- Активно пишу статьи на Хабре
- Пишу на Kotlin

Откуда родилась тема доклада?

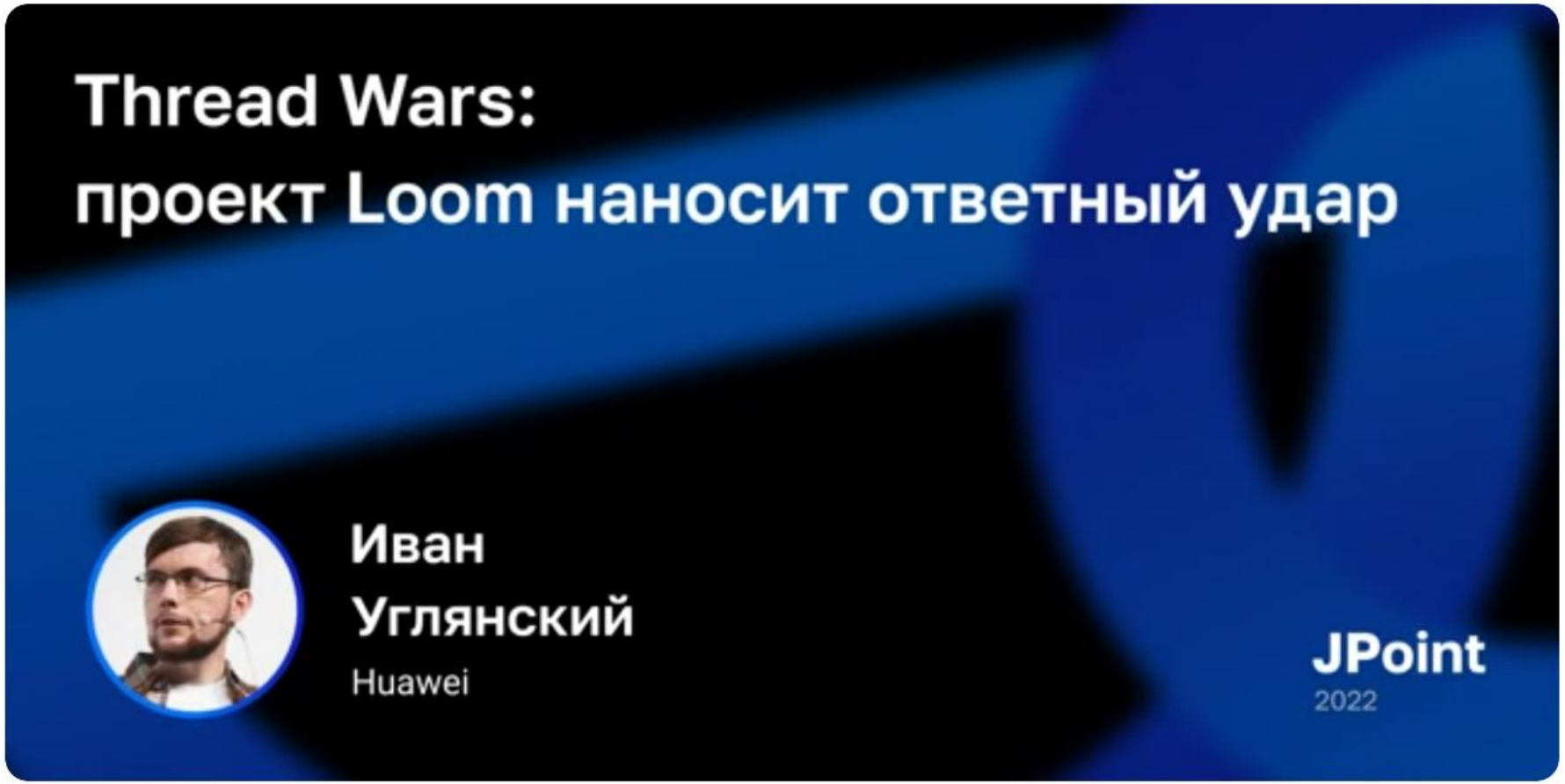
Reksoft

- Начало жизни на Kotlin проекте с асинхронными операциям
- Вопрос от опытных разработчиков: *Всё ещё непонятно, в чём принципиальная разница между Coroutine и другими технологиями, которая позволяет писать асинхронный код так, будто он синхронный.*

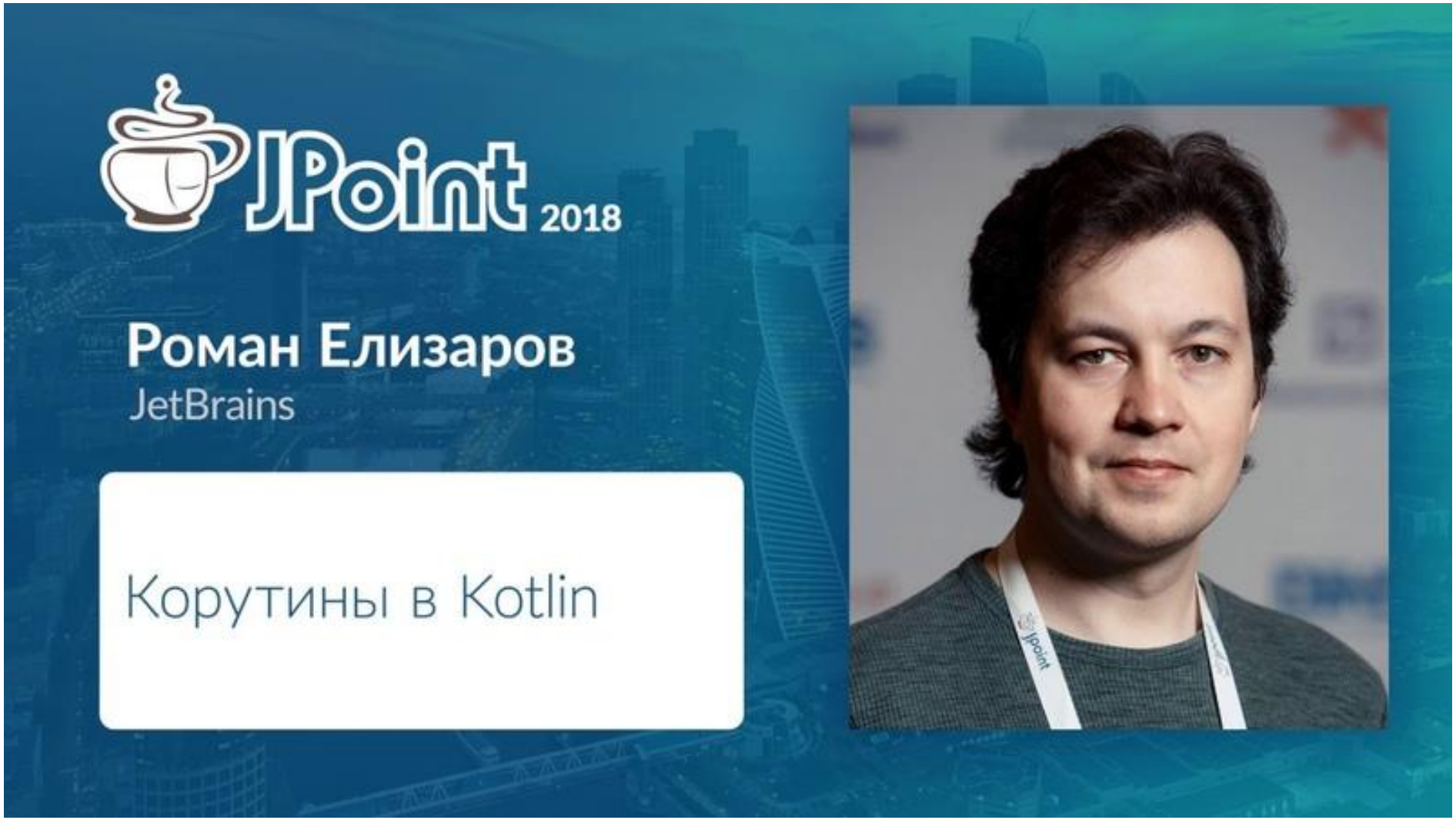
Советую посмотреть!

Reksoft

Thread Wars: проект Loom наносит ответный удар



Reksoft



Что такое «асинхронное программирование»?

Reksoft

Wikipedia:

Асинхронное программирование — концепция программирования, которая заключается в том, что результат выполнения функции доступен не сразу, а через некоторое время в виде некоторого асинхронного (нарушающего обычный порядок выполнения) вызова.

Абстрактная задача

Reksoft

У нас есть приложение, в котором одна из функций — это поиск и агрегация данных из нескольких источников.
К примеру, мы ищем билеты на самолет и опрашиваем всех поставщиков.
Пусть таких источников 5.

Решаем на CompletableFuture

```
private fun aggregateTickets(
    suppliers: List<TicketSupplier>
): CompletableFuture<List<Ticket>> {
    val futures = suppliers.map { fetchTickets(it.name) }
```

List<CompletableFuture<List<Ticket>>>

Решаем на CompletableFuture

Reksoft

```
private fun aggregateTickets(
    suppliers: List<TicketSupplier>
): CompletableFuture<List<Ticket>> {
    val futures = suppliers.map { fetchTickets(it.name) }
```

List<CompletableFuture<List<Ticket>>>

Решаем на CompletableFuture

Reksoft

```
return CompletableFuture.allOf(*futures.toTypedArray())
    .thenApply { it: Void!
        futures.flatMap { completableFuture ->
            completableFuture.join()
        }
    }.exceptionally { ex ->
        handleError(ex)
        emptyList() ^exceptionally
    }
```

Решаем на CompletableFuture

Reksoft

```
return CompletableFuture.allOf(*futures.toTypedArray())
    .thenApply { it: Void!
        futures.flatMap { completableFuture ->
            completableFuture.join()
        }
    }.exceptionally { ex ->
        handleError(ex)
        emptyList() ^exceptionally
    }
```

Решаем на CompletableFuture

Reksoft

```
return CompletableFuture.allOf(*futures.toTypedArray())
    .thenApply { it: Void!
        futures.flatMap { completableFuture ->
            completableFuture.join()
        }
    }.exceptionally { ex ->
        handleError(ex)
        emptyList() ^exceptionally
    }
```

Отлично! Добавляем условие

Reksoft

После получения данных от поставщиков, мы можем фильтровать результаты по цене, затем сортировать их и, в конце концов, сохранять в базе данных.

Оставляем начало

Reksoft

```
private fun aggregateTickets(
    suppliers: List<TicketSupplier>
): CompletableFuture<List<Ticket>> {
    val futures = suppliers.map { fetchTickets(it.name) }
    return CompletableFuture.allOf(*futures.toTypedArray()) CompletableFuture<Void!>!
        .thenApply { it: Void!
            futures.flatMap { completableFuture ->
                completableFuture.join()
            }
        } CompletableFuture<List<Ticket>!>!
```

Что добавили?

Reksoft

```
.thenCompose { tickets ->
    filterTickets(tickets)
}.thenCompose { filteredTickets ->
    sortTickets(filteredTickets)
}.thenApply { sortedTickets ->
    saveTickets(sortedTickets)
    sortedTickets ^thenApply // Возвращаем отсортированные билеты
}.exceptionally { ex ->
    handleError(ex)
    emptyList() ^exceptionally // Возвращаем пустой список в случае ошибки
}
```

Что добавили?

Reksoft

```
.thenCompose { tickets ->
    filterTickets(tickets)
}.thenCompose { filteredTickets ->
    sortTickets(filteredTickets)
}.thenApply { sortedTickets ->
    saveTickets(sortedTickets)
    sortedTickets ^thenApply // Возвращаем отсортированные билеты
}.exceptionally { ex ->
    handleError(ex)
    emptyList() ^exceptionally // Возвращаем пустой список в случае ошибки
}
```

Что добавили?

Reksoft

```
.thenCompose { tickets ->
    filterTickets(tickets)
}.thenCompose { filteredTickets ->
    sortTickets(filteredTickets)
}.thenApply { sortedTickets ->
    saveTickets(sortedTickets)
    sortedTickets ^thenApply // Возвращаем отсортированные билеты
}.exceptionally { ex ->
    handleError(ex)
    emptyList() ^exceptionally // Возвращаем пустой список в случае ошибки
}
```

CompletableFuture, характеристика

Reksoft

- *Callback hell все еще с нами.*
- *Необходимость изучить совершенно новый API.*
- *Возвращаемый тип отличается от фактических данных, которые нам нужны.*
- *Обработка ошибок может быть сложной.*

Основные реализации, какие они?

Reksoft

Completable Future

Callback

Coroutine

Thread

Reksoft

Coroutine

Ключевые особенности Coroutine

Reksoft

- ***Легковесные потоки!***
- ***Освобождение ресурсов при приостановке.***
- ***Императивный подход!***
- ***Простое лучше чем сложное!***

Ключевая особенность, suspend функция!

Reksoft

- Помечается модификатором словом: ***suspend***
- Явно говорит компилятору, что потенциально функция может быть ***приостановлена***
- Приостанавливающая функция реализована ***как конечные автоматы***

Coroutine state machine

Reksoft

- У каждой Coroutine имеется свое состояние

- Состояние сохраняется в объект ***Continuation***: локальные переменные и целочисленное поле для текущего состояния в конечном автомате

```
suspend fun complying(param: Int) {
    delay( timeMillis: 100)
    requestData( url: "Url")
}
```

Decompile #1

Reksoft

```
@Nullable
public static final Object complying(final int param, @NotNull Continuation $completion) {
   return CoroutineScopeKt.coroutineScope((Function2)(new Function2((Continuation)null) {
      no usages
      int label;
```

Decompile #1

Reksoft

```
@Nullable
public static final Object complying(final int param, @NotNull Continuation $completion) {
   return CoroutineScopeKt.coroutineScope((Function2)(new Function2((Continuation)null) {
      no usages
      int label;
```

Decompile #2

Reksoft

```
public final Object invokeSuspend(@NotNull Object $result) {
   Object var3 = IntrinsicsKt.getCOROUTINE_SUSPENDED();
   Object var10000;
   switch (this.label) {
      case 0:
         ResultKt.throwOnFailure($result);
         int var2 = param;
         this.label = 1;
         if (DelayKt.delay(100L, this) == var3) {
            return var3;
         }
         break;
      case 1:
         ResultKt.throwOnFailure($result);
         break;
      case 2:
         ResultKt.throwOnFailure($result);
         var10000 = $result;
         return var10000;
      default:
         throw new IllegalStateException("call to 'resume' before 'invoke' with coroutine");
   }
```

Suspend-и осторожно!

Coroutine это что?

Reksoft

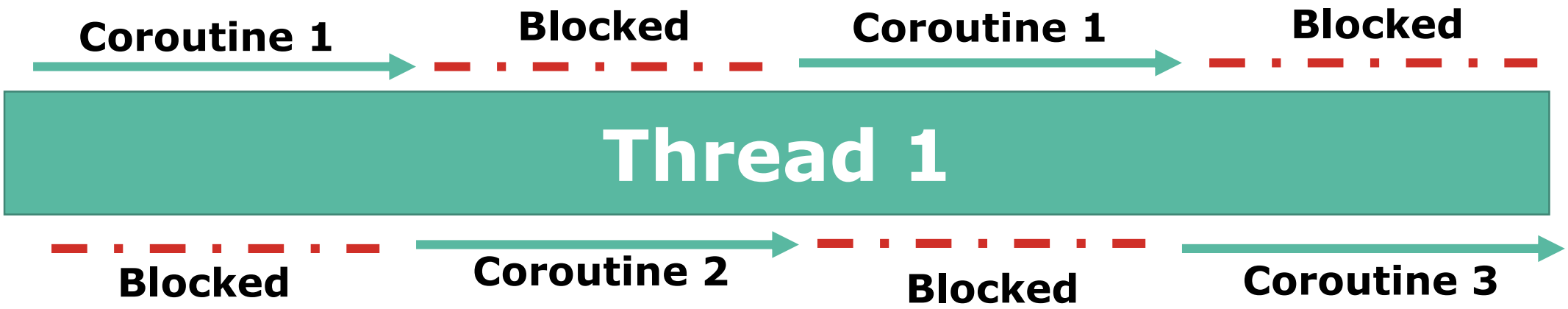
- ***A coroutine is an instance of a suspendable computation.***
- ***It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code.***
- ***However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.***

Киллер фича?

Reksoft

Более наглядно!

Reksoft



Билдеры в корутинах

Reksoft

launch

async

run Blocking

launch builder

Reksoft

Описание:

- Запускает корутину, которая выполняется параллельно с основной программой и не возвращает результат.

Особенности:

- Может не завершить свою работу

Пишем код!

Reksoft

```
fun main() {
    GlobalScope.launch { this: CoroutineScope
        delay( timeMillis: 2000L)
        println("Launch is on Joker!")
    }
    println("Mama,")
}
```

Вывод

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Mama,

Process finished with exit code 0
```

Не взлетело, а почему?

```
fun main() {
    GlobalScope.launch { this: CoroutineScope
        delay( timeMillis: 2000L)
        println("Launch is on Joker!")
    }
    println("Mama,")
}
```

Программа завершила работу, а лаунч не успел ☹

runBlocking builder

Reksoft

Описание:

- Запускает корутину и блокирует текущий поток до её завершения. Обычно используется в тестах или в main функции.

Особенности:

- Блокирующая функция
- Создание корутинного контекста
- Можно вызывать не из suspend функции

Не взлетело, а почему?

Reksoft

```
fun main() = runBlocking { this: CoroutineScope
    launch { this: CoroutineScope
        delay( timeMillis: 2000L)
        println("Launch is on Joker!")
    }
    println("Mama,")
}
```

Вывод

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Mama,
Launch am on Joker!
```

async builder

Reksoft

Описание:

- Запускает корутину и возвращает объект Deferred, который можно использовать для получения результата выполнения.

Особенности:

- Возвращает результат
- Блокирует Thread при вызове await()

Пишем код!

```
fun main() = runBlocking { this: CoroutineScope
    val result1 =  async { this: CoroutineScope
        delay( timeMillis: 2000L)
        "Mama!" ^async
    }
    val result2 = async { this: CoroutineScope
        delay( timeMillis: 4000)
        "Async is on Joker!" ^async
    }
    println(result2.await())
    println(result1.await())
}
```

Вывод

```
Async is on Joker!
Mama!
```

coroutineScope

Reksoft

Описание:

- Создаёт новый корутинный контекст, который управляет жизненным циклом корутин, запущенных внутри блока.

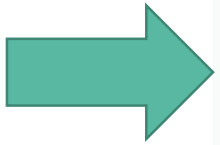
Особенности:

- Не блокирует основной поток
- Гарантия завершения
- Вызывается только из suspend функции

Пишем код!

Reksoft

```
suspend fun main() {
    coroutineScope { this: CoroutineScope
        launch { this: CoroutineScope
            println("I am a launch builder")
        }
        delay( timeMillis: 3000L)
        println("coroutineScope is on Joker")
    }
    println("Mama!")
}
```



Вывод

```
I am a launch builder
coroutineScope is on Joker
Mama!
```

Окей, runBlocking vs coroutineScope!

Reksoft

runBlocking is a regular function
and coroutineScope is a suspending function.

Пишем на корутинах

Reksoft

```
private suspend fun aggregateTickets(
    suppliers: List<TicketSupplier>
): List<Ticket> {
    return coroutineScope { this: CoroutineScope
        val tickets = suppliers.map { it: TicketSupplier
            async { fetchTickets(it.name) }
        }
        return@coroutineScope tickets.awaitAll().flatten()
    }
}
```

Пока все классно. Но давайте посмотрим глубже

Job: почему так важна и какую роль играет?

Reksoft

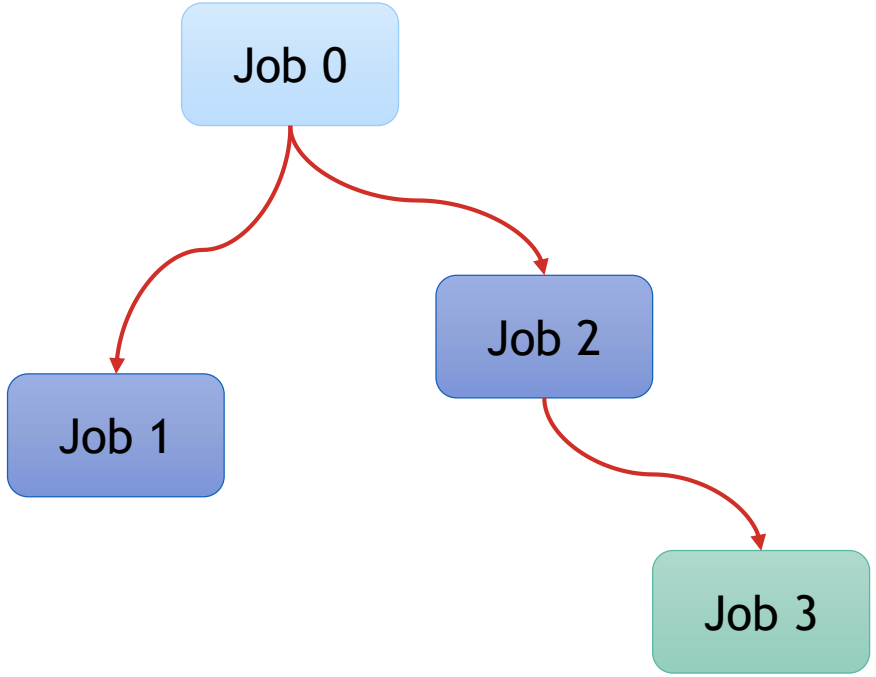
```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
    val newContext = newCoroutineContext(context)
    val coroutine = if (start.isLazy)
        LazyStandaloneCoroutine(newContext, block) else
        StandaloneCoroutine(newContext, active = true)
    coroutine.start(start, coroutine, block)
    return coroutine
}
```

Необыкновенная и прекрасная Job

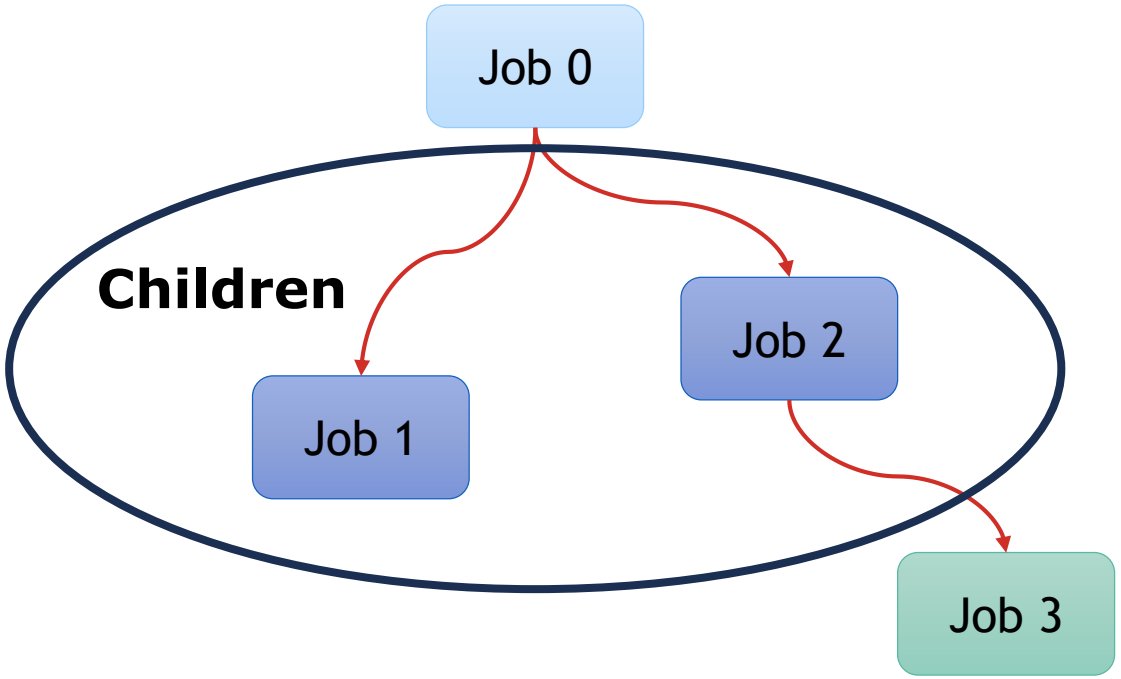
Reksoft

```
public interface Job : CoroutineContext.Element {

    @ExperimentalCoroutinesApi
    public val parent: Job?
    public val isActive: Boolean
    public val isCompleted: Boolean
    public val isCancelled: Boolean
    public fun start(): Boolean
    public fun cancel(cause: CancellationException? = null)
    public val children: Sequence<Job>
    public suspend fun join()
```

Parent? Зачем же нам...



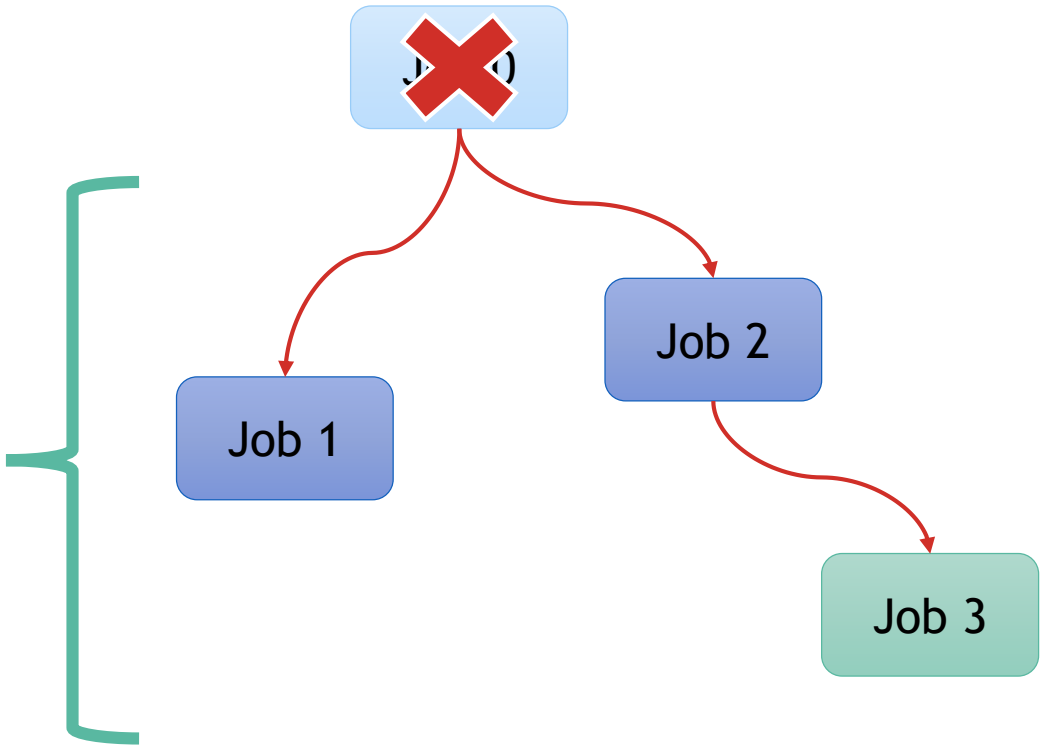
Рассмотрим детальней



Parent? Зачем же нам...

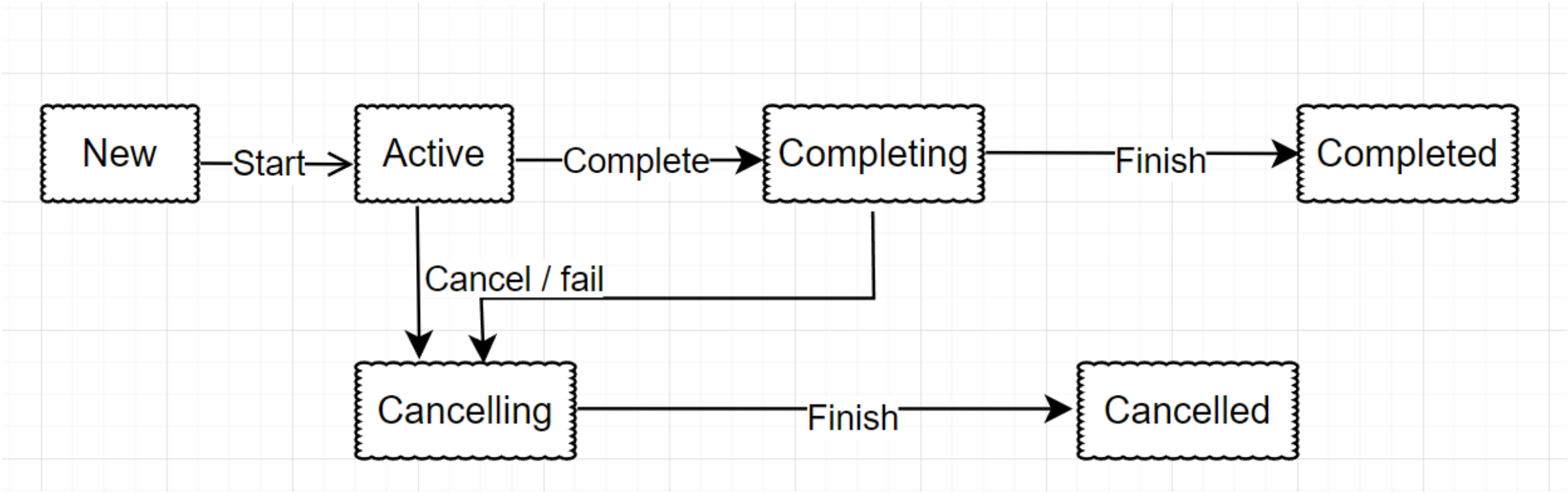
Reksoft

Отменяет все дочерние



Жизненный цикл!

Reksoft



Состояние корутин

Reksoft

State	isActive	isCompleted	isCancelled
New (optional initial state)	`false`	`false`	`false`
Active (default initial state)	`true`	`false`	`false`
Completing (transient state)	`true`	`false`	`false`
Cancelling (transient state)	`false`	`false`	`true`
Cancelled (final state)	`false`	`true`	`true`
Completed (final state)	`false`	`true`	`false`

Пишем код для примера!

Reksoft

```
fun main() = runBlocking { this: CoroutineScope
    val job = launch { this: CoroutineScope
        val async2 = async { this: CoroutineScope
            delay( timeMillis: 5000)
            println("async 2")
        }
        async2.await()
    }
    println(job) /*      StandaloneCoroutine{Active} */
    delay( timeMillis: 1000)
    job.cancel()
    println(job) /*      Standalone Coroutine{Cancelling} */
    delay( timeMillis: 100)
    println(job) /*      Standalone Coroutine{Cancelled} */
}
```

Промежуточные выводы

Reksoft

- ***Имеется поддержка «отношений» родитель-ребенок.***
- ***Сопрограмма заканчивает свою работу либо со статусом Completed, либо Cancelled.***
- ***Дочерняя сопрограмма при уничтожении уничтожает и родительскую сопрограмму, или наоборот***

Вернемся к примеру, а что еще не знакомо?

Reksoft

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
```

Продолжим с малого -> CoroutineStart

Reksoft

- **DEFAULT**
- **LAZY**
- **ATOMIC**
- **UNDISPATCHED**

CoroutineStart.Default

Reksoft

Корутина начинает выполнение немедленно, но может быть отложена, если родительская корутина или контекст не готовы.

```
fun main() {
    GlobalScope.launch(start = CoroutineStart.DEFAULT) {
        // some work
    }
}
```

CoroutineScope.Lazy

Reksoft

Запускает корутину только тогда, когда ее результат требуется для await или если вызывается функция start у Job.

```
fun main() {
    val job = GlobalScope.launch(start = CoroutineStart.LAZY) {
        // some work
    }
    job.start()
}
```

CoroutineStart. Atomic

Reksoft

Тоже самое, что и DEFAULT, но не может быть отменена до момента ее запуска.

```
fun main() {
    GlobalScope.launch(start = CoroutineStart.ATOMIC) {
        // some work
    }
}
```

CoroutineStart.UNDISPATCHED

Reksoft

Запускает сопрограмму до ее первой приостановки.

```
fun main() {
    GlobalScope.launch(start = CoroutineStart.UNDISPATCHED) {
        println("Execute") /*      Execute */
        delay( timeMillis: 100)
        println("Won't be executed")
    }
}
```

Снова смотрим на Launch

Reksoft

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
```

А что там по CoroutineScope?

Reksoft

```
public interface CoroutineScope {
    /**
     * The context of this scope. Context is encapsulated by the scope and used for implementation of
     * coroutine builders that are extensions on the scope. Accessing this property in general code is not
     * recommended for any purposes except accessing the Job instance for advanced usages.
     *
     * By convention, should contain an instance of a job to enforce structured concurrency.
     */
    public val coroutineContext: CoroutineContext
}
```

А что там по CoroutineScope?

Reksoft

```
public interface CoroutineScope {
    /**
     * The context of this scope. Context is encapsulated by the scope and used for implementation of
     * coroutine builders that are extensions on the scope. Accessing this property in general code is not
     * recommended for any purposes except accessing the Job instance for advanced usages.
     *
     * By convention, should contain an instance of a job to enforce structured concurrency.
     */
    public val coroutineContext: CoroutineContext
}
```

Кратко об особенностях CoroutineScope

Reksoft

- ***Управляет жизненным циклом!***
- ***Определяет область применения для новых сопрограмм***
- ***Расширяется в билдерах сопрограмм***
- ***Устанавливает общее поведение***

Способы создания CoroutineScope #1(GlobalScope)

Reksoft

```
fun main() {
    GlobalScope.launch { this: CoroutineScope
        // some work
    }
}
```

Способы создания CoroutineScope #2(Custom)

Reksoft

```
class CustomScope : CoroutineScope {
    private val job = Job()

    private val exceptionHandler =
        CoroutineExceptionHandler { _, exception ->
            println("Ошибка: ${exception.localizedMessage}")
        }
```

Способы создания CoroutineScope #2(Custom)

Reksoft

```
override val coroutineContext =
    Dispatchers.Default +
        job +
        exceptionHandler

fun cancel() {
    job.cancel()
}
```

Использование CustomScope

Reksoft

```
private suspend fun aggregateTickets(
    customScope: CustomScope,
    suppliers: List<TicketSupplier>
): List<Ticket> {
    val tickets = suppliers.map { it: TicketSupplier
        customScope.async { fetchTickets(it.name) }
    }
    return tickets.awaitAll().flatten()
```

Способы создания CoroutineScope #3(Constructor)

Reksoft

```
val supervisorScope = CoroutineScope(
        context: Dispatchers.Main +
            SupervisorJob()
)
supervisorScope.launch { this: CoroutineScope
    // hmmm
}
```

CoroutineContext?

Reksoft

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
```

А что такое CoroutineContext?

Reksoft

Persistent context for the coroutine.
It is an indexed set of Element instances.
An indexed set is a mix between a set and a map.Every element in this set has a unique Key.

CoroutineContext.Element

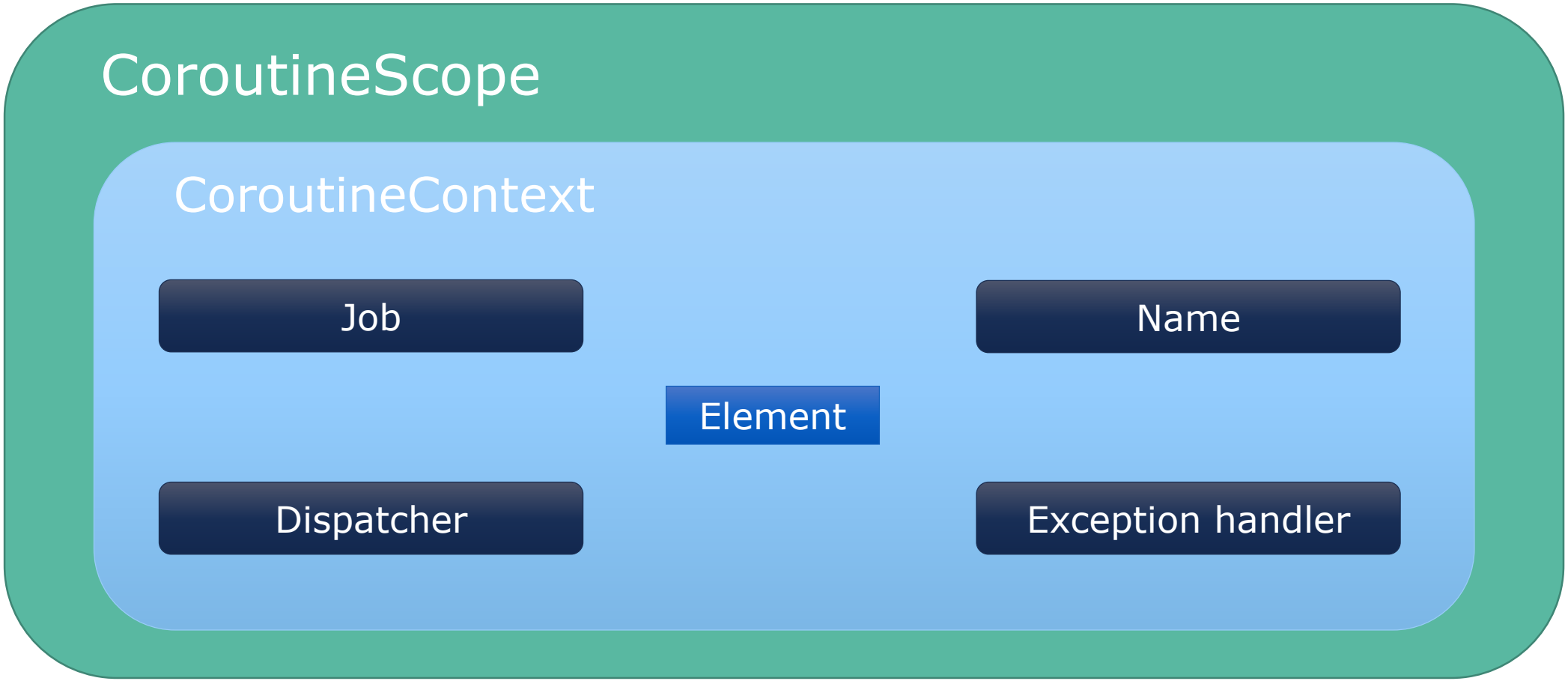
Reksoft

> An element of the `CoroutineContext`. An element of the coroutine context is a singleton context by itself.

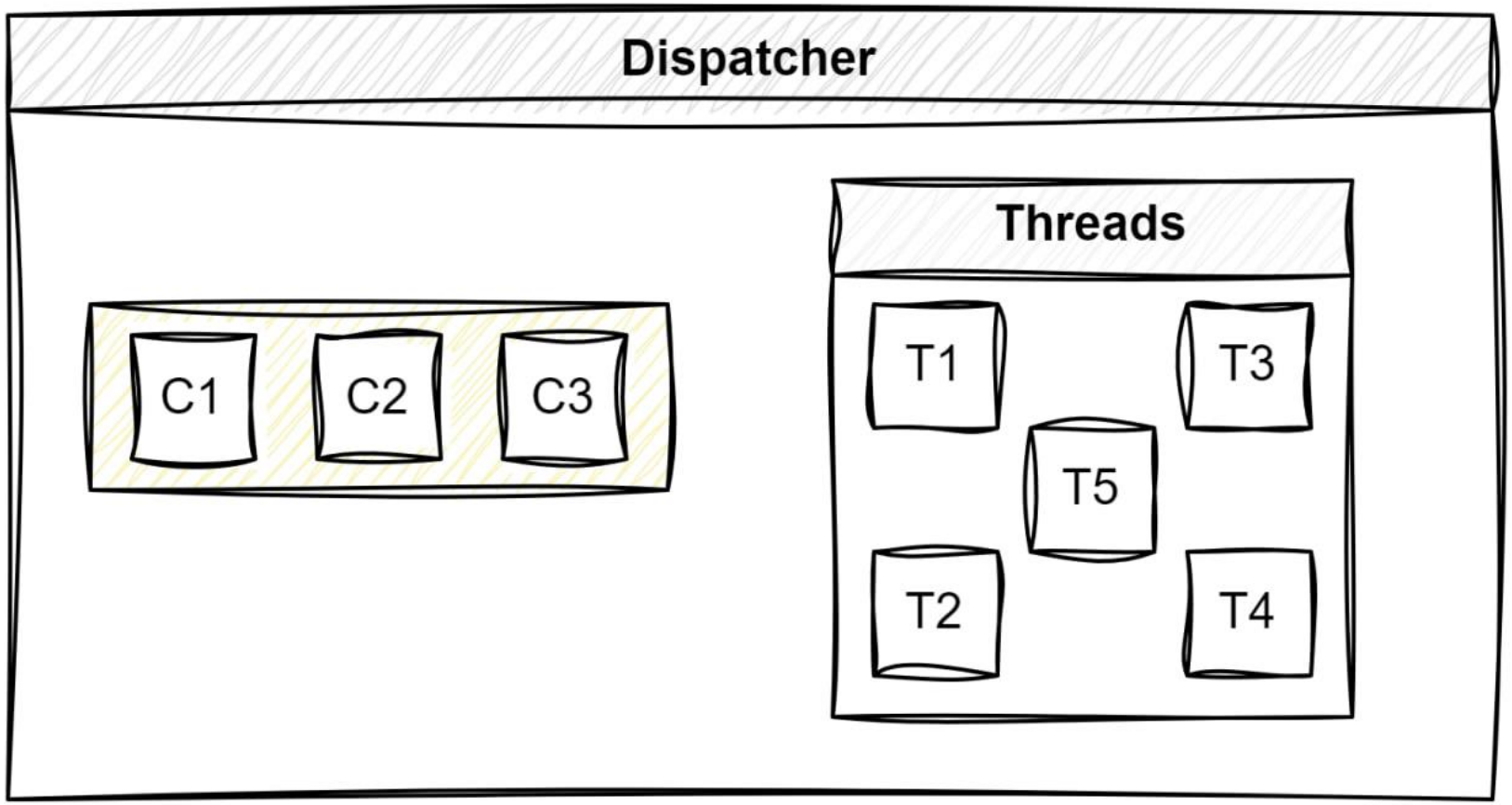
```
public interface Element : CoroutineContext {
    /**
     * A key of this coroutine context element.
     */
    public val key: Key<*>

    public override operator fun <E : Element> get(key: Key<E>): E? =
        @Suppress( ...names: "UNCHECKED_CAST")
        if (this.key == key) this as E else null

    public override fun <R> fold(initial: R, operation: (R, Element) -> R): R =
        operation(initial, this)

    public override fun minusKey(key: Key<*>): CoroutineContext =
        if (this.key == key) EmptyCoroutineContext else this
}
```

CoroutineContext.Element

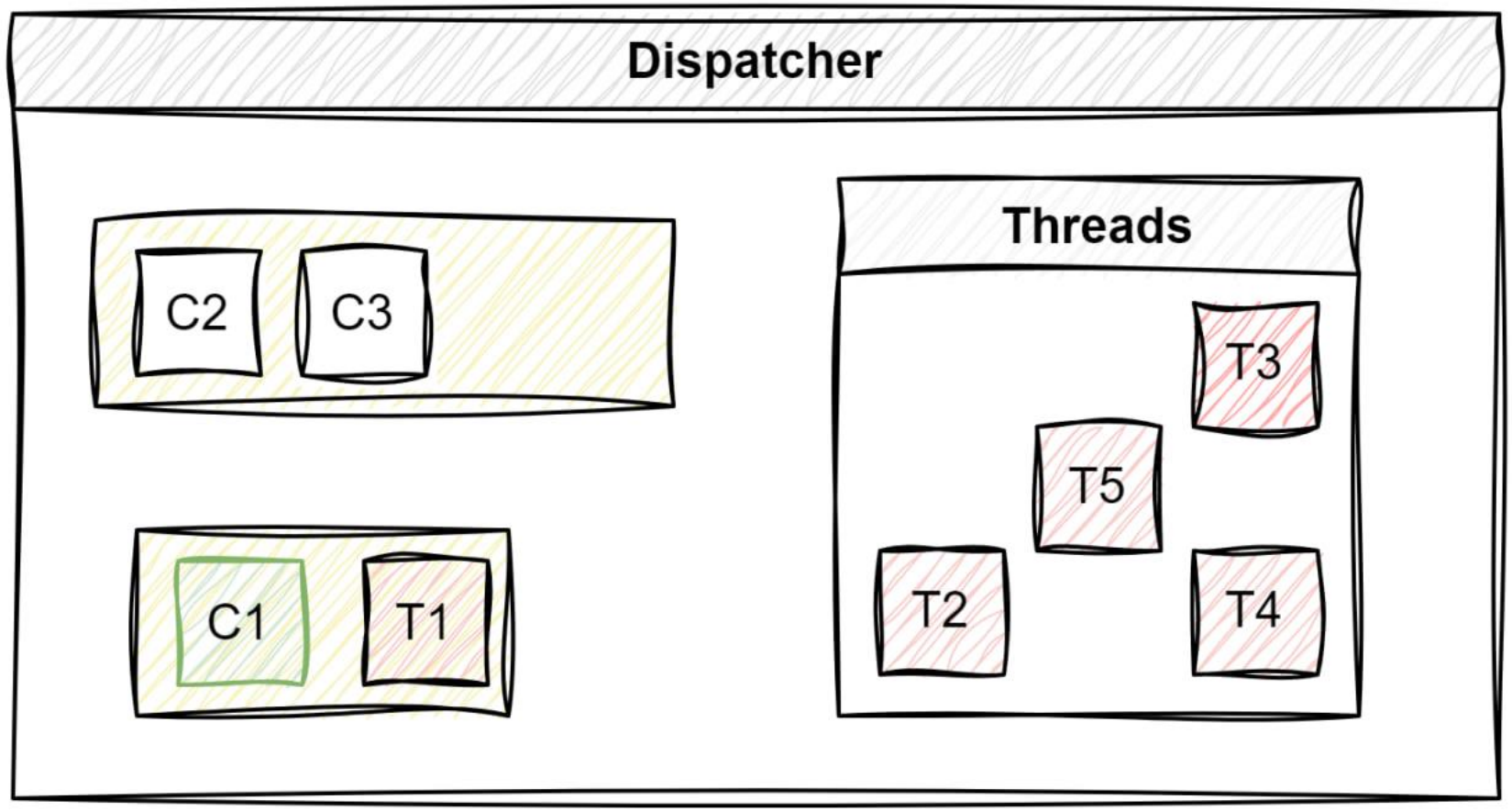
Reksoft



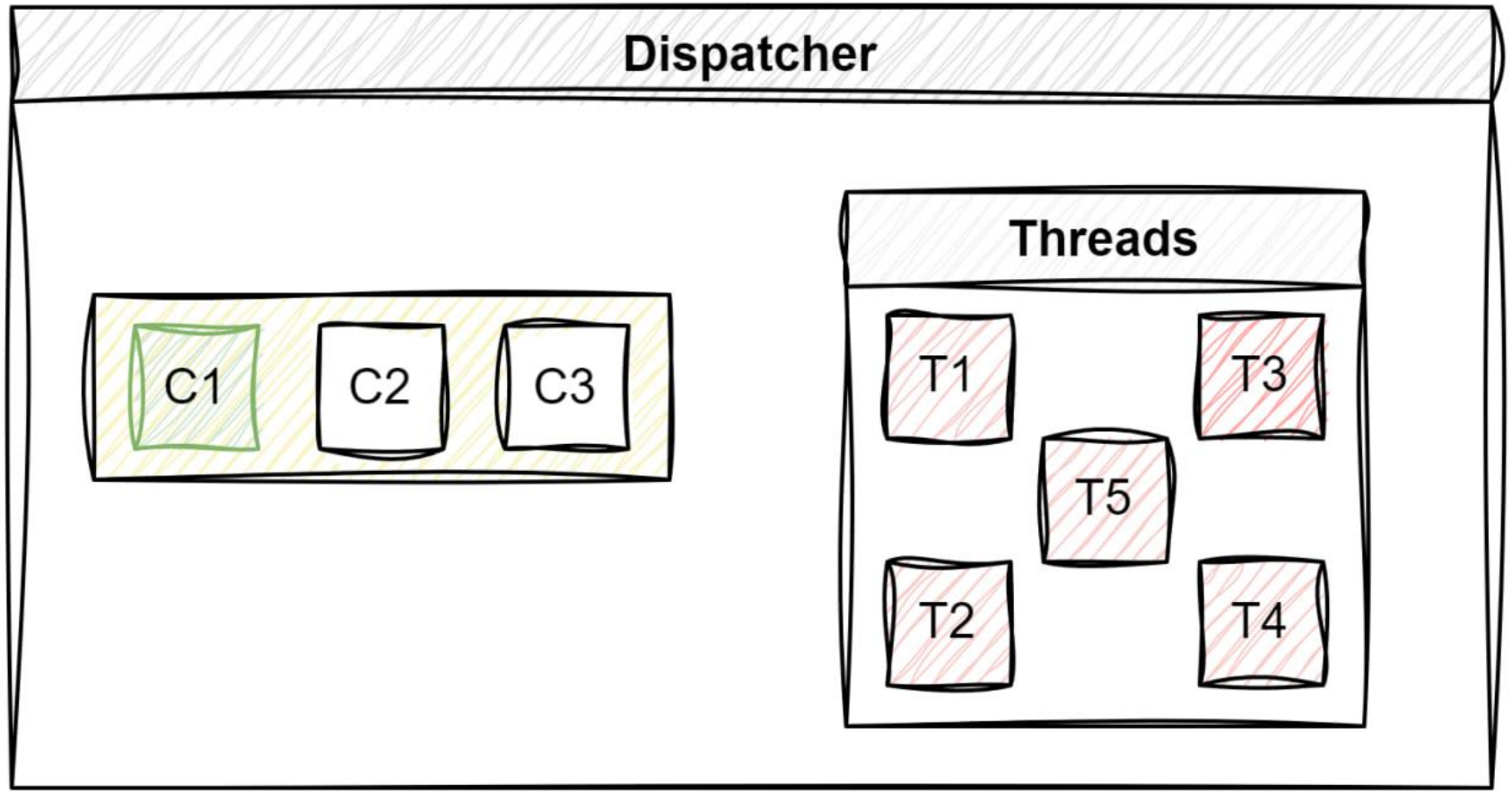
Dispatcher, а зачем он и причем тут Thread?



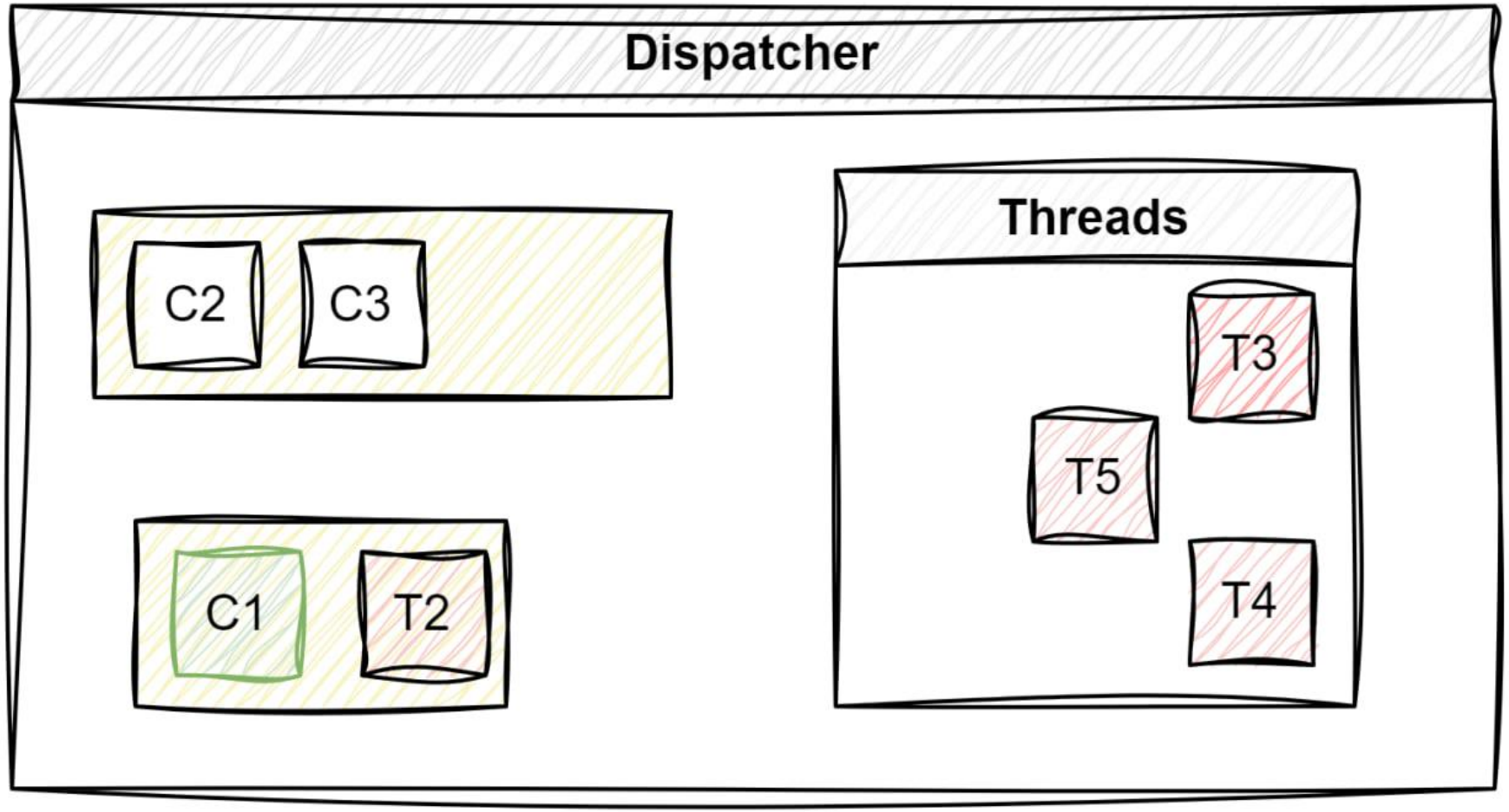
Dispatcher, а зачем он и причем тут Thread?



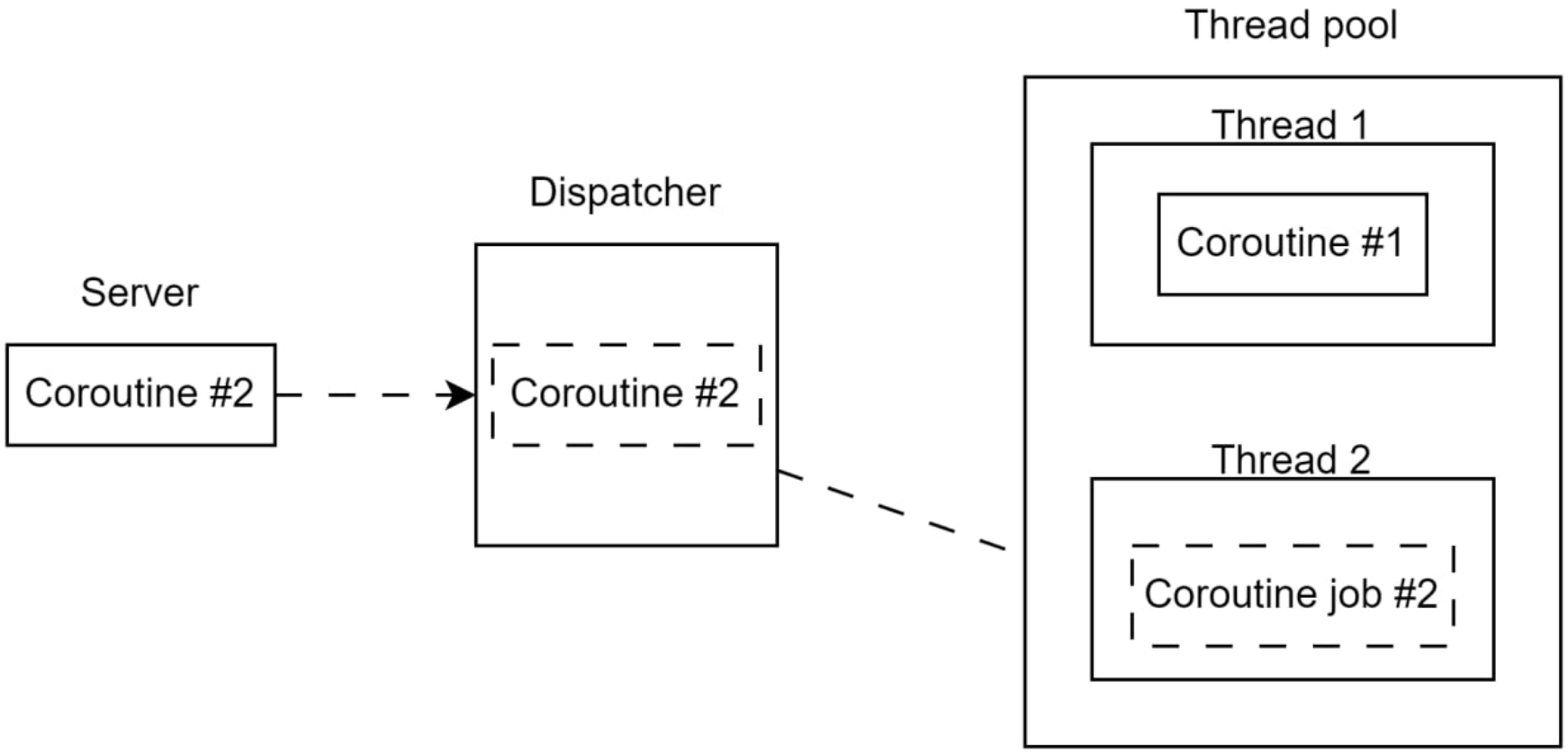
Dispatcher, а зачем он и причем тут Thread?



Dispatcher, а зачем он и причем тут Thread?



Dispatcher, а зачем он и причем тут Thread?



Dispatcher зачем?

Reksoft

- **Dispatchers.Default**: Этот диспетчер предназначен для выполнения операций с интенсивным использованием процессора и имеет пул потоков размером, равным количеству ядер на компьютере, на котором выполняется ваш код (но не менее двух).

- **Dispatchers.IO:** Этот диспетчер предназначен для выполнения операций с интенсивным вводом-выводом(сетевые запросы, работа с файлами и т.д.) и имеет больший пул потоков. Эффективно обрабатывать большое количество задач, которые часто заблокированы в ожидании завершения ввода-вывода.

- **Dispatchers.Main:** Этот диспетчер предназначен для запуска сопрограмм в главном потоке приложения Android.

- **newSingleThreadContext, newFixedThreadPoolContext:** Когда требуются пулы потоков, полностью отделенные от Dispatchers. ***Выделенный поток — очень дорогой ресурс.***

Модификация?

Reksoft

```
class CustomScope : CoroutineScope {
    private val job = Job()

    private val exceptionHandler =
        CoroutineExceptionHandler { _, exception ->
            println("Ошибка: ${exception.localizedMessage}")
        }

    override val coroutineContext =
        Dispatchers.IO +
            job +
            exceptionHandler
```

Callback пример!

Reksoft

```
fun aggregateTickets(
    suppliers: List<TicketSupplier>,
    callback: (List<Ticket>) -> Unit
) {
    val results = mutableListOf<Ticket>()
    val pendingRequests = suppliers.size
```

Callback пример!

Reksoft

```
suppliers.forEach { supplier ->
    fetchTickets(supplier.name) { tickets ->
        results.addAll(tickets)
        if (results.size == pendingRequests) {
            callback(results)
        }
    }
}
```

Отлично! Добавляем условие

Reksoft

После получения данных от поставщиков, мы можем фильтровать результаты по цене, затем сортировать их и, в конце концов, сохранять в базе данных.

А что там будет с Callback?

Reksoft

```
fun aggregateTickets(
    suppliers: List<TicketSupplier>,
    callback: (List<Ticket>) -> Unit
) {
    val allTickets = mutableListOf<Ticket>()
    var completedRequests = 0

    suppliers.forEach { supplier ->
        fetchTickets(supplier.name) { tickets ->
            allTickets.addAll(tickets)
            completedRequests++
```

А что там будет с Callback?

Reksoft

```
if (completedRequests == suppliers.size) {
    filterTickets(allTickets) { filtered ->
        sortTickets(filtered) { sorted ->
            saveTickets(sorted) { saved ->
                callback(saved)
            }
        }
    }
}
```

```
var floppy = require('floppy');

floppy.load('disk1', function (data1) {
    floppy.prompt('Please insert disk 2', function() {
        floppy.load('disk2', function (data2) {
            floppy.prompt('Please insert disk 3', function() {
                floppy.load('disk3', function (data3) {
                    floppy.prompt('Please insert disk 4', function() {
                        floppy.load('disk4', function (data4) {
                            loppy.prompt('Please insert disk 5', function() {
                                floppy.load('disk5', function (data5) {
                                    floppy.prompt('Please insert disk 6', funct
                                        floppy.load('disk6', function (data6) {
                                            //if node.js would have existed in
                                        });
                                });
                            });
                        });
                    );

                });
```

Как было на CompletableFuture!

Reksoft

```
20  private fun aggregateTickets(suppliers: List<TicketSupplier>): CompletableFuture<List<Ticket>> {
21      val futures = suppliers.map { fetchTickets(it.name) }
22      return CompletableFuture.allOf(*futures.toTypedArray()) CompletableFuture<Void!>!
23          .thenApply { it: Void!
24              futures.flatMap { it.join() }
25          }.thenCompose { tickets ->
26              filterTickets(tickets)
27          } CompletableFuture<List<Ticket>!>!
28          .thenCompose { filteredTickets ->
29              sortTickets(filteredTickets)
30          }.thenApply { sortedTickets ->
31              saveTickets(sortedTickets)
32              sortedTickets ^thenApply // Возвращаем отсортированные билеты
33          }.exceptionally { ex ->
34              handleError(ex)
35              emptyList() ^exceptionally // Возвращаем пустой список в случае ошибки
36          }
37  }
```

Coroutine?

Reksoft

```
suspend fun aggregateTicketsWithFilterAsync(
    customScope: CustomScope,
    suppliers: List<TicketSupplier>
): List<Ticket> {
    val tickets = suppliers.flatMap { it: TicketSupplier
        customScope.async { fetchTickets(it.name) }.await()
    }
```

Coroutine?

Reksoft

```
suspend fun aggregateTicketsWithFilterAsync(
    customScope: CustomScope,
    suppliers: List<TicketSupplier>
): List<Ticket> {
    val tickets = suppliers.flatMap { it: TicketSupplier
        customScope.async { fetchTickets(it.name) }.await()
    }
```

Coroutine?

Reksoft

```
val sorted = sortTickets(tickets)
return filterTickets(sorted)
    .also { it: List<Ticket>
        saveTickets(it)
    }
```

Еще вариант реализации?

Reksoft

```
fun aggregateTicketsWithFilter(
    customScope: CustomScope,
    suppliers: List<TicketSupplier>
): List<Ticket> {
    val tickets = mutableListOf<Ticket>()
    suppliers.forEach { it: TicketSupplier
        customScope.launch { this: CoroutineScope
            val ticket = fetchTickets(it.name)
            tickets.addAll(ticket)
        }
    }
```

Еще вариант реализации?

Reksoft

```
fun aggregateTicketsWithFilter(
    customScope: CustomScope,
    suppliers: List<TicketSupplier>
): List<Ticket> {
    val tickets = mutableListOf<Ticket>()
    suppliers.forEach { it: TicketSupplier
        customScope.launch { this: CoroutineScope
            val ticket = fetchTickets(it.name)
            tickets.addAll(ticket)
        }
    }
```

Ложка дегтя?

Reksoft

1. Нетривиальный дебаг
2. Потенциальные проблемы с производительностью
3. Можно потерять контекст выполнения
4. Необходимость управления контекстом

Почему удобно их использовать?

Reksoft

→ Можно использовать точечно, не переписывая весь код

→ Можно добавить в легаси проект

→ Простое лучше, чем сложное!

Выводы

Reksoft

1. Более четкий и естественный код
2. Эффективный механизм приостановки
3. Гибкость и контроль
4. Широкая поддержка
5. Абстракция над Thread-ами

Reksoft

Kotlin Coroutines. Устройство и возможности использования.

Соловьев Андрей

Andrey_solovyev@list.ru

9.10.2024