АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 9

ПЛАН

- Введение в корутины

МНОГОПОТОЧНОСТЬ В JVM

- Thread - понятие, узаконенное в языке и vm
- Есть формализованная модель памяти
- Есть механизм синхронизации и volatile-переменные
- Из коробки - много надстроек
- Пулы нитей, локи, семафоры, блокирующиеся очереди и т.п.

ПРИМЕРЫ ПРИЛОЖЕНИЙ

- Торрент-клиент
- Сервис-агретатор, делегирующий запросы партнерам
- Web-crawler
- Клиентское приложение мессенджера
- Прием данных из брокера сообщений

BASELINE-ПОДХОДЫ И ИХ ПРОБЛЕМЫ

- Чистые нити - недостаточно абстрактны
- Слишком дороги, чтобы бесконтрольно выделяться
- Коммуникация через wait-notity недостаточно абстрактна
- Реальный baseline: пулы нитей, атомики, коллекции из java.utiil.concurrent

BASELINE-ПОДХОДЫ И ИХ ПРОБЛЕМЫ

- Пулы добавляют сценарии для дедлоков
- Усложняется обработка ошибок
- Нет коробочного решения для базовых шаблонов типа "асинхронная цепочка"
- Нарушается принцип single-responsibility
  - Логика распараллеливания смешивается с бизнес-логикой

ПОДХОДЫ К РЕШЕНИЮ

- Future в Scala - решает задачу асинхронной цепочки
- Асинхронный контейнер, вписанный в структуру коллекций
- Можно делать отложенные map-ы, порождать производную Future по завершении текущей
- Можно унифицированно обрабатывать ошибки или делегировать обработку
- Легкий объект, но под капотом - конкурентная многопоточность

ПОДХОДЫ К РЕШЕНИЮ

- Функциональные фреймворки (Monix, Cat Effects) - обеспечивают кооперативную многопоточность
- Требуют освоения неочевидных сходу понятий
- RxJava/RxKotlin - реализация парадигмы реактивного программирования
- Фокусируются скорее на структурах
- Неплохо отделяют структуры от бизнес-логики

КОРУТИНЫ

- Корутины - механизм кооперативной многозадачности
- Классическая корутина - это кусок логики, предполагающий ожидание чего-либо
- Таймаута, асинхронного ввода, ответа на запрос, завершения другой корутины
- В момент ожидания она дает возможность поработать другим

ПРИМЕР

```
 1 import kotlinx.coroutines.*
 2
 3 fun main() = runBlocking {
 4     println("start")
 5     launch {
 6         delay(1000L)
 7         println("World!")
 8     }
 9     println("Hello")
10 }
```

РАЗБЕРЕМ

- runBlocking - портал в мир корутин
- Создает контекст, запускает первую корутину, ожидает завершения
- launch запускает вторую корутину в контексте
- Вторая корутина засыпает "правильным" способом

РАЗБЕРЕМ

- Во время сна она не занимает нить
- Корутинный фреймворк освободившуюся нить может отдать кому-то
- В нашем случае - первой корутине
- По истечении таймаута - корутина снова займет какую-то нить

МНОГО КОРУТИН

```
 1 import kotlinx.coroutines.*
 2
 3 fun main() = runBlocking {
 4     repeat(50_000) {
 5         launch {
 6             delay(5000L)
 7             print(".")
 8         }
 9     }
10 }
```

НИТИ

```
 1 import kotlinx.coroutines.*
 2
 3 fun main() = runBlocking {
 4     repeat(100) {
 5         launch {
 6             delay(5000L)
 7             println(Thread.currentThread().id)
 8         }
 9     }
10 }
```

ДЕКОМПОЗИЦИЯ

- Засыпающие функции помечаются ключевым словом suspend
- Функция, вызываюшая suspend-функции, обязана быть suspend-функцией
- Вызов suspend-функции вне корутинного контекста запрещен
- launch возвращает объект класса Job

JOB

- Над Job можно вызвать join - и дождаться завершения
- Можно вызвать cancel - в надежде прервать работу Job-а
- И это случится, как только Job войдет в suspend- режим
- Интересен эффект от брошенного исключения в рамках порожденной корутины

ПРИМЕР

```
1  import kotlinx.coroutines.*
2
3  fun main() = runBlocking {
4      val job = launch {
5          println("job-1")
6          delay(1000L)
7          //Thread.sleep(1000L)
8          println("job-2")
9      }
10 // ..............
```

ПРИМЕР

```
1 // ..............
2     println("out-1")
3     delay(10)
4     println("out-2")
5     job.join()
6     println("out-3")
7 }
```

ПРИМЕР

```
 1 fun main() = runBlocking {
 2     val job = launch {
 3         println("job-1")
 4         delay(1000L)
 5         //Thread.sleep(1000L)
 6         println("job-2")
 7     }
 8     println("out-1")
 9     delay(10)
10     println("out-2")
11     job.cancel()
12     println("out-3: " + job.isCancelled)
13     delay(1)
14 }
```

ПРИМЕР

```
1 fun main() = runBlocking {
2     val job = launch {
3         println("job-1")
4         delay(1000L)
5         throw RuntimeException()
6         println("job-2")
7     }
8 // ........
```

ПРИМЕР

```
1  // ..........
2      try {
3          println("out-1")
4          delay(10)
5          println("out-2")
6          Thread.sleep(500)
7          println("out-3: " + job.isCancelled)
8          delay(1000)
9      } catch (t: Throwable) {
10         println("GOT: " + t)
11         t.printStackTrace(System.out)
12     }
13 // .......
```

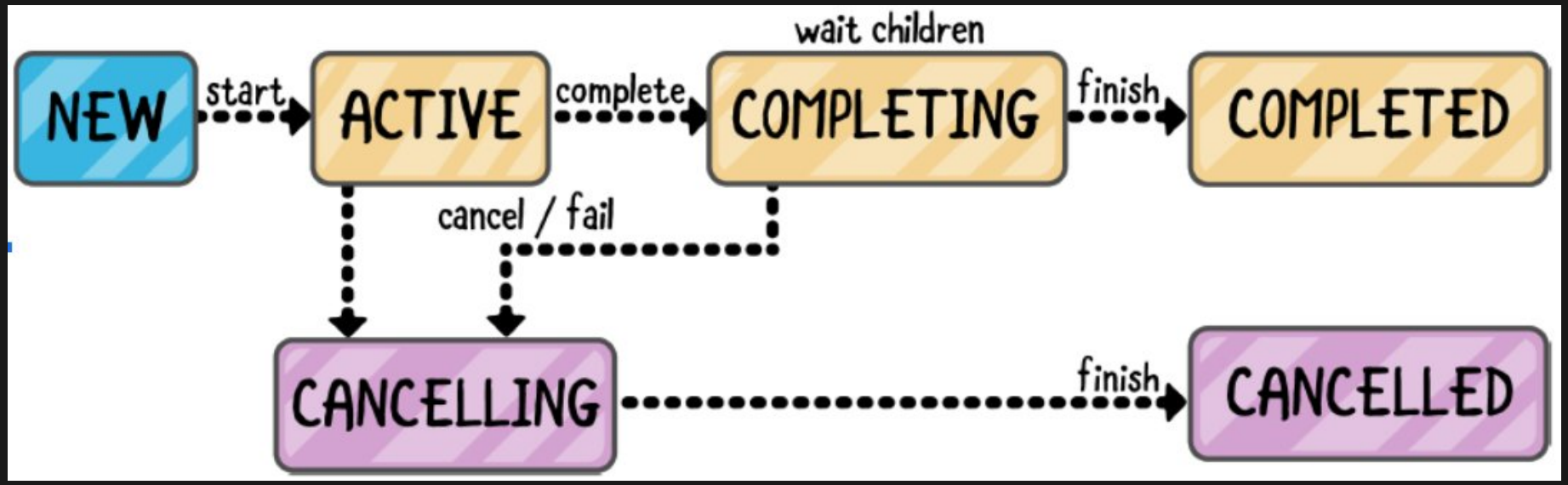
ПРИМЕР

```
1 // ..........
2     println("out-4")
3     delay(1000)
4     println("out-5")
5 }
```

ПОЧЕМУ ТАК

- Концептуально корутина и порожденные ею корутины считаются частью единого решения
- И по умолчанию - если случается проблема в одном компоненте, это проблема всего решения
- Этого можно избежать, подробности попозже
- Если не избежать, корутина может еще доработать до первого suspend-а

ДИАГРАММА СОСТОЯНИЙ



ПРИМЕР

```
 1 fun main() {
 2     runBlocking(Dispatchers.Default) {
 3         val job = Job()
 4         println(job)
 5         println(job.isActive)
 6         job.complete()
 7         println(job)
 8         println(job.isActive)
 9         println(job.isCompleted)
10
11 // ..............
```

ПРИМЕР

```
// ..............

        val job2 = launch {
            delay(2000)
        }
        println(job2)
        println(job2.isActive)
        job2.join()
        println(job2)
        println(job2.isActive)
        println(job2.isCompleted)
    }
}
```

ПРИМЕР

```
1 fun main() {
2     runBlocking(Dispatchers.Default) {
3         val job = launch(start=CoroutineStart.LAZY) {
4             delay(2000)
5         }
6         println(job)
7         println(job.isActive)
8         job.start()
9 // ......................
```

ПРИМЕР

```
1 // ......................
2         println(job)
3         println(job.isActive)
4         job.join()
5         println(job)
6         println(job.isActive)
7         println(job.isCompleted)
8     }
9 }
```

ПРИМЕР С ДЕТЬМИ

```
fun main() {
    runBlocking(Dispatchers.Default) {
        val job = launch {
            launch {
                delay(10000)
            }
            launch {
                delay(20000)
            }
// ..............
```

ПРИМЕР С ДЕТЬМИ

```
1  // ..............
2              delay(1000)
3              println(
4                  coroutineContext.job.children.toList()
5              )
6              delay(2000)
7          }
8
9          println(
10              "1: "
11              + coroutineContext.job.children.toList()
12         )
13         delay(500)
14 // ..............
```

ПРИМЕР С ДЕТЬМИ

```
1 // ..............
2         println(
3             "2: "
4             + coroutineContext.job.children.toList()
5         )
6         delay(5000)
7 // ..............
```

ПРИМЕР С ДЕТЬМИ

```
1 // ................
2         println("3: " + coroutineContext.job.isActive)
3         println(
4             "3: "
5             + coroutineContext.job.children.toList()
6         )
7         delay(8000)
8 // ................
```

ПРИМЕР С ДЕТЬМИ

```
// ..............
        println(
            "4: "
            + coroutineContext.job.isActive
        )
        println(
            "4: "
            + coroutineContext.job.children.toList()
        )
        delay(8000)
// ..............
```

ПРИМЕР С ДЕТЬМИ

```
// ................
        println(
            "5: "
            + coroutineContext.job.isActive
        )
        println(
            "5: "
            + coroutineContext.job.children.toList()
        )
    }
}
```

ПРИМЕР: CANCEL ИЗНУТРИ

```
 1 fun main() {
 2     runBlocking(Dispatchers.Default) {
 3         launch {
 4             launch {
 5                 launch {
 6                     println("one")
 7                     delay(1000)
 8                     println("two")
 9                     cancel()
10                     delay(1000)
11                     println("three")
12                 }
13 // ..................
```

ПРИМЕР: CANCEL ИЗНУТРИ

```
1  // ................
2
3                  println("wait 10s")
4                  delay(10000)
5                  println("wait 10s done")
6              }
7              launch {
8                  println("wait 20s")
9                  delay(20000)
10                 println("wait 20s done")
11             }
12 // ................
```

ПРИМЕР: CANCEL ИЗНУТРИ

```
1 // ..................
2
3             delay(1000)
4             println(coroutineContext.job.children.toList())
5             delay(2000)
6         }
7     }
8 }
```

ПРИМЕР: CANCEL ИЗНУТРИ С ДЕТЬМИ

```
 1 fun main() {
 2     runBlocking(Dispatchers.Default) {
 3         launch {
 4             launch {
 5                 launch {
 6                     println("one")
 7                     delay(1000)
 8                     println("two")
 9                     delay(1000)
10                     println("three")
11                 }
12 // ..............
```

ПРИМЕР: CANCEL ИЗНУТРИ С ДЕТЬМИ

```
1  // ..............
2                  println("wait 10s")
3                  cancel()
4                  delay(10000)
5                  println("wait 10s done")
6              }
7              launch {
8                  println("wait 20s")
9                  delay(20000)
10                 println("wait 20s done")
11             }
12 // ..............
```

ПРИМЕР: CANCEL ИЗНУТРИ С ДЕТЬМИ

```
1 // ..............
2             delay(1000)
3             println(
4                 coroutineContext.job.children.toList()
5             )
6             delay(2000)
7         }
8     }
9 }
```

СВОИ SUSPEND-ФУНКЦИИ

- Пока мы вызывали delay и служебные методы
- Хочется создавать свои suspend-функции
- Чтобы они ожидали каких-то наших асинхронных событий

ИДЕЯ

- Есть интерфейс Continuation
- Каждой корутине соответствует объект, реализующий его
- Через него можно "разбудить" корутину
- Можно получить этот объект

ПРИМЕР

```
1  class CoroutineSocket() {
2      val socket = AsynchronousSocketChannel.open()
3      var isConnected: Boolean = false
4          private set
5      val isOpened: Boolean
6          get() = socket.isOpen
7      suspend fun connect(isa: InetSocketAddress) {
8          suspendCoroutine {
9              socket.connect(
10                 isa, it,ContinuationHandler(this)
11             )
12         }
13     }
14 // ..................
```

ПРИМЕР

```
// ...................
    companion object {
        class ContinuationHandler<T>(
            private val socket: CoroutineSocket
        ): CompletionHandler<T, Continuation<T>> {
            override fun completed(
                result: T, attachment: Continuation<T>
            ) {
                socket.isConnected = true
                attachment.resume(result)
            }
// ...................
```

ПРИМЕР

```
// ...................
            override fun failed(
                exc: Throwable,
                attachment: Continuation<T>
            ) {
                attachment.resumeWithException(exc)
            }
        }
    }
}
// ...................
```

ПРИМЕР

```
1 // ...................
2 suspend fun main() {
3     val socket1 = CoroutineSocket()
4     println(socket1.isConnected)
5     socket1.connect(InetSocketAddress("www.ya.ru", 443))
6     println(socket1.isConnected)
7 // ...................
```

ПРИМЕР

```
// ...................
    try {
        val socket2 = CoroutineSocket()
        println(socket2.isConnected)
        socket2.connect(
            InetSocketAddress("www.yaya.ru", 443)
        )
        println(socket2.isConnected)
    } catch (t: Throwable) {
        println("done: " + t)
    }
}
```

ПРИМЕР

```
object NetUtils {
    suspend fun isAlive(
        host: String, port: Int
    ): Boolean {
        return suspendCoroutine {
            val socket = AsynchronousSocketChannel.open()
            socket.connect(
                InetSocketAddress(host, port),
                it,
                ContinuationHandler(socket)
            )
        }
    }
// ................
```

ПРИМЕР

```
// ................
    private class ContinuationHandler(
        private val socket: AsynchronousSocketChannel
        ): CompletionHandler<Void, Continuation<Boolean>> {
            override fun completed(
                result: Void?,
                attachment: Continuation<Boolean>
            ) {
                socket.close()
                attachment.resume(true)
            }
// ................
```

ПРИМЕР

```
1  // ................
2              override fun failed(
3                  exc: Throwable,
4                  attachment: Continuation<Boolean>
5              ) {
6                  attachment.resume(false)
7              }
8          }
9  }
10
11 suspend fun main() {
12     println(NetUtils.isAlive("www.ya.ru", 443))
13     println(NetUtils.isAlive("www.yaya.ru", 443))
14 }
```

СЕРВЕРНАЯ СТОРОНА

- Тут отдельная проблема: в Java API нет callback-интерфейса на accept
- Решается через разные контексты
- Каждый контекст имеет свой пул нитей с подходящей политикой создания
- Есть отдельный диспетчер - Dispatchers.IO
- Подходит для блокирующего ввода-вывода или для интенсивных вычислений

СЕРВЕР НА МИНИМАЛКАХ

```
1  class ContinuationHandler<T>:
2          CompletionHandler<T, Continuation<T>> {
3      override fun completed(
4          result: T, attachment: Continuation<T>
5      ) {
6          attachment.resume(result)
7      }
8      override fun failed(
9          exc: Throwable, attachment: Continuation<T>
10     ) {
11         attachment.resumeWithException(exc)
12     }
13 }
14 // ..............
```

СЕРВЕР НА МИНИМАЛКАХ

```
1 // ..............
2 class CoroutineSocket(
3     val channel:AsynchronousSocketChannel =
4         AsynchronousSocketChannel.open()
5 ) {
6     var isConnected: Boolean = false
7         private set
8 // ..............
```

СЕРВЕР НА МИНИМАЛКАХ

```
1 // ..............
2     suspend fun connect(isa: InetSocketAddress) {
3         suspendCoroutine {
4             channel.connect(isa, it, ContinuationHandler())
5             isConnected = true
6         }
7     }
8 // ..............
```

СЕРВЕР НА МИНИМАЛКАХ

```
1 // ..............
2     suspend fun read(
3         buffer: ByteBuffer
4     ): Int = suspendCoroutine {
5         channel.read(
6             buffer, it, ContinuationHandler<Int>()
7         )
8     }
9 // ..............
```

СЕРВЕР НА МИНИМАЛКАХ

```
1 // ..............
2     suspend fun write(
3         buffer: ByteBuffer
4     ): Int  = suspendCoroutine {
5         channel.write(buffer, it, ContinuationHandler())
6     }
7 }
8 // ..............
```

СЕРВЕР НА МИНИМАЛКАХ

```
1 // ..............
2 class CoroutineServerSocket {
3     private val channel =
4         AsynchronousServerSocketChannel.open()
5 // ..............
```

СЕРВЕР НА МИНИМАЛКАХ

```
// ..............
    suspend fun bindAndAccept(
        port: Int
    ): CoroutineSocket {
        val result = suspendCoroutine {
            channel.bind(InetSocketAddress(port))
            val handler =
    ContinuationHandler<AsynchronousSocketChannel>()
            channel.accept(it, handler)
        }
        return CoroutineSocket(result)
    }
}
```

ИСПОЛЬЗОВАНИЕ

```
 1 fun main() {
 2     runBlocking {
 3         val server = CoroutineServerSocket()
 4         launch {
 5             val clientPeer = withContext(
 6                 Dispatchers.IO
 7             ) {
 8                 server.bindAndAccept(1234)
 9             }
10
11             val readBuffer = ByteBuffer.allocate(1024)
12             val nRead = clientPeer.read(readBuffer)
13 // ..................
```

ИСПОЛЬЗОВАНИЕ

```
 1 // ..................
 2             readBuffer
 3                 .rewind()
 4                 .limit(nRead)
 5
 6             println(
 7                 "GOT: " +
 8                 Charset.defaultCharset()
 9                         .decode(readBuffer)
10             )
11 // ..................
```

ИСПОЛЬЗОВАНИЕ

```
1  // ..................
2              val lines = listOf(
3                  "HTTP/1.1 200 OK",
4                  "Content-Type: text/plain",
5                  "Content-Length: 5",
6                  "",
7                  "Hello",
8                  ""
9              )
10             val answer = lines.joinToString("\r\n")
11             val writeBuffer =
12                 Charset.defaultCharset()
13                         .encode(answer)
14 // ..................
```

ИСПОЛЬЗОВАНИЕ

```
1 // ..................
2             val nWrite = clientPeer.write(writeBuffer)
3             println(nWrite)
4         }
5     }
6 }
```

ДИСПЕТЧЕРИЗАЦИЯ

- Корутины кооперативны
- Suspention point - место, где корутина может уступить место другим
- Может возобновиться на другой нити
- На какой именно - решает контекст

КОНТЕКСТ

- Каждая корутина запускается в некотором контексте
- Контекст представлен интерфейсом CoroutineContext
- CoroutineContext - это своего рода словарь
- Со своеобразным интерфейсом

КОНТЕКСТ

```
1 public interface CoroutineContext {
2     /**
3      * Returns the element with the given
4      * [key] from this context or `null`.
5      */
6     public operator fun <E : Element>
7     get(key: Key<E>): E?
8 // ..............
```

КОНТЕКСТ

```
1  // ..............
2      /**
3       * Accumulates entries of this context starting with
4       * [initial] value and applying [operation]
5       * from left to right to current accumulator value
6       * and each element of this context.
7       */
8      public fun <R> fold(
9          initial: R, operation: (R, Element) -> R
10     ): R
11 // ..............
```

КОНТЕКСТ

```
1 // ..............
2     /**
3      * Returns a context containing elements from
4      * this context and elements from  other [context].
5      * The elements from this context with the same key
6      * as in the other one are dropped.
7      */
8     public operator fun plus(
9         context: CoroutineContext
10    ): CoroutineContext =
11          /* .......... */
12 // ..............
```

КОНТЕКСТ

```
1  // ..............
2      /**
3       * Returns a context containing elements from
4       * this context, but without an element with
5       * the specified [key].
6       */
7      public fun minusKey(key: Key<*>): CoroutineContext
8
9      /**
10      * Key for the elements of [CoroutineContext].
11      * [E] is a type of element with this key.
12      */
13     public interface Key<E : Element>
14 // ..............
```

КОНТЕКСТ

```
1 // ..............
2     /**
3      * An element of the [CoroutineContext].
4      * An element of the coroutine context
5      * is a singleton context by itself.
6      */
7     public interface Element : CoroutineContext {
8         /**
9          * A key of this coroutine context element.
10         */
11        public val key: Key<*>
12 // ..............
```

КОНТЕКСТ

```
1  // ................
2          public override operator fun <E> : Element
3          get(key: Key<E>): E? =
4              @Suppress("UNCHECKED_CAST")
5              if (this.key == key) this as E else null
6
7          public override fun <R> fold(
8              initial: R, operation: (R, Element) -> R
9          ): R =
10             operation(initial, this)
11 // ................
```

КОНТЕКСТ

```
1 // ................
2         public override fun minusKey(
3             key: Key<*>
4         ): CoroutineContext =
5             if (this.key == key) EmptyCoroutineContext
6             else this
7     }
```

ВИД СНАРУЖИ

```
fun main() {
    runBlocking {
        coroutineContext.fold(Unit) { a, b ->
            println("key class" + b.key.javaClass)
            println("value class" + b[b.key]?.javaClass)
            println("key" + b.key)
            println("value" + b[b.key])
        }
// ...................
```

ВИД СНАРУЖИ

```
// ...................
        println()

        println(coroutineContext[Job.Key])
        println(
            coroutineContext[ContinuationInterceptor.Key]
        )

        println(coroutineContext[Job])
        println(
            coroutineContext[ContinuationInterceptor]
        )
    }
}
```

РАЗБЕРЕМ

- Контекст можем включать несколько элементов
- В нашем случае - два
- Один описывает детали Job-а
- Другой - способ диспетчеризации

КОНТЕКСТ ЗАПУСКАЕМОЙ КОРУТИНЫ

- В первом приближении - job-часть определяется типом корутины
- Часть дипетчеризации - тем, что было передано параметром в launch или аналог
- По умолчанию - передается CoroutineContext.EMPTY
- Что означает наследование способа диспетчеризации

ПРИМЕР

```
fun main() {
    println(Thread.currentThread())
    runBlocking {
        println(coroutineContext[ContinuationInterceptor])
        repeat(5) {
            launch {
                println(
                    "start: ${Thread.currentThread()}"
                )
// ...................
```

ПРИМЕР

```
1  // ...................
2                  Thread.sleep(3000)
3                  println(
4                      "finish: ${Thread.currentThread()}"
5                  )
6              }
7              val t = Thread.currentThread()
8              println(
9                  "launched $it from $t"
10             )
11             // delay(1)
12         }
13     }
14 }
```

ПРИМЕР

```
1 fun current() = Thread.currentThread()
2
3 fun main() {
4     println(t())
5     runBlocking(Dispatchers.Default) {
6         println(coroutineContext[ContinuationInterceptor])
7         repeat(100) { i ->
8             launch {
9 // ................
```

ПРИМЕР

```
// ...................
                println("start $i: ${current()}")
                Thread.sleep(3000)
                println("finish $i: ${current()}")
            }
            println("launched $i from ${current()}")
            // delay(1)
        }
    }
}
```

ПРИМЕР

```
1 fun current() = Thread.currentThread()
2
3 fun main() {
4     println(current())
5     runBlocking(Dispatchers.Default) {
6         println(coroutineContext[ContinuationInterceptor])
7         repeat(100) { i ->
8             launch {
9 // ..............
```

ПРИМЕР

```
1  // ..............
2                  println("start $i: ${current()}")
3                  delay(3000)
4                  println("finish $i: ${current()}")
5              }
6              println("launched $i from ${current()}")
7              // delay(1)
8          }
9      }
10 }
```

ПРИМЕР С ОГРАНИЧЕНИЕМ ПАРАЛЛЕЛИЗМА

```
 1 fun current() = Thread.currentThread()
 2
 3 fun main() {
 4     println(current())
 5     runBlocking(
 6         Dispatchers.Default.limitedParallelism(3)
 7     ) {
 8         println(
 9             coroutineContext[ContinuationInterceptor]
10         )
11 // ....................
```

ПРИМЕР С ОГРАНИЧЕНИЕМ ПАРАЛЛЕЛИЗМА

```
// ......................
        repeat(100) { i ->
            launch {
                println("start $i: ${current()}")
                delay(3000)
                println("finish $i: ${current()}")
            }
            println("launched $i from ${current()}")
            // delay(1)
        }
    }
}
```

ОБРАБОТКА ОШИБОК В ДЕТЯХ

- Можем сделать родителя супервизором
- Ему будут приходить извещения о проблемах детей
- Супервизор что-то с этим делает
- Может игнорировать, может перезапускать

ПРИМЕР

```
fun main() {
    runBlocking(Dispatchers.Default) {
        launch {
            launch {
                val scope = CoroutineScope(
                    SupervisorJob()
                )
                scope.launch {
                    println("one")
                    delay(1000)
                    println("two")
                    Thread.sleep(1000)
// ...................
```

ПРИМЕР

```
1  // ...................
2                      println("going to throw....")
3                      throw IOException()
4                      delay(1000)
5                      println("three")
6                  }
7                  println("wait 10s")
8                  println(
9                      currentCoroutineContext().job
10                                           .children
11                                           .toList()
12                 )
13 // ...................
```

ПРИМЕР

```
// ...................
                println(
                    scope.coroutineContext
                         .job
                         .children
                         .toList()
                )
                delay(10000)
// ...................
```

ПРИМЕР

```
1 // ...................
2                 println(
3                     currentCoroutineContext().job
4                                              .children
5                                              .toList()
6                 )
7 // ...................
```

ПРИМЕР

```
// ..................
                println(
                    scope.coroutineContext.job
                                          .children
                                          .toList()
                )
                println("wait 10s done")
            }
        }
    }
}
```

ПРИМЕР С ОБРАБОТКОЙ

```
fun main() {
    val action: suspend CoroutineScope.() -> Unit = {
        println("one")
        delay(1000)
        println("two")
        Thread.sleep(1000)
        println("going to throw....")
        throw IOException()
        delay(1000)
        println("three")
    }
// ...................
```

ПРИМЕР С ОБРАБОТКОЙ

```
1 // ..................
2
3     runBlocking(Dispatchers.Default) {
4         launch {
5             launch {
6                 val scope = CoroutineScope(SupervisorJob())
7                 val job1 = scope.launch(block = action)
8                 val job2 = scope.launch(block = action)
9 // ..................
```

ПРИМЕР С ОБРАБОТКОЙ

```
 1 // ...................
 2                 job1.invokeOnCompletion {
 3                     println("complete: $it")
 4                     it?.let {
 5                         scope.launch {
 6                             println("once again")
 7                             action(scope)
 8                         }
 9                     }
10                 }
11 // ...................
```

ПРИМЕР С ОБРАБОТКОЙ

```
1 // ...................
2                 println("wait 10s")
3                 println(currentCoroutineContext().job
4                                         .children.toList())
5                 println(scope.coroutineContext.job
6                                         .children.toList())
7                 delay(10000)
8 // ...................
```

ПРИМЕР С ОБРАБОТКОЙ

```
// ...................
                println(currentCoroutineContext()
                                        .job.children.toList())
                println(scope.coroutineContext
                                        .job.children.toList())
                println("wait 10s done")
            }
        }
    }
}
```

ИДЕМ ДАЛЬШЕ

- suspend-функция сама по себе - не корутина
- launch порождает Job/корутину
- Внутри могут вызываться suspend-функции
- Их значения можем использовать при вызове других
- Но launch ничего не возвращает

ASYNC/DEFERED

- Defered<T> - развитие Job
- Корутина, возвращающая значение
- async - аналог launch для конструирования Defered
- У Defered есть suspend-метод await - ожидает результат

ПРИМЕР

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {
    val deferred: Deferred<Int> = async {
        loadData()
    }
    println("waiting...")
    println(deferred.await())
}
// ..............
```

ПРИМЕР

```
1 // ..............
2
3 suspend fun loadData(): Int {
4     println("loading...")
5     delay(1000L)
6     println("loaded!")
7     return 42
8 }
```