IT's TINKOFF

Project Loom – серебряная пуля?

Или все же нет?

Иван Лягаев

31.10.2023

Скачать презентацию

Иван Лягаев

Главный Scala разработчик,

Тинькофф.Бизнес

i.lyagaev@tinkoff.ru

Telegram: @FireFoxIL

Github: @FireFoxIL

Содержание

Проблематика

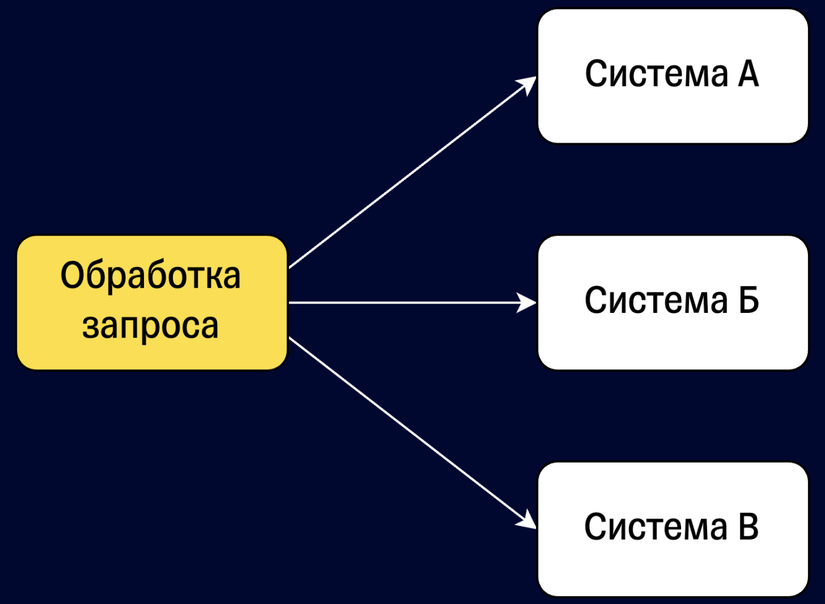
- Мы живем в эпоху микросервисов
- Работа нашего сервиса зачастую зависит от работы других сервисов

Проблематика

- Мы живем в эпоху микросервисов
- Работа нашего сервиса зачастую зависит от работы других сервисов
- Хотим, чтобы наш сервис предоставлял ответ как можно быстрее
- Это вынуждает нас уметь писать конкурентный код

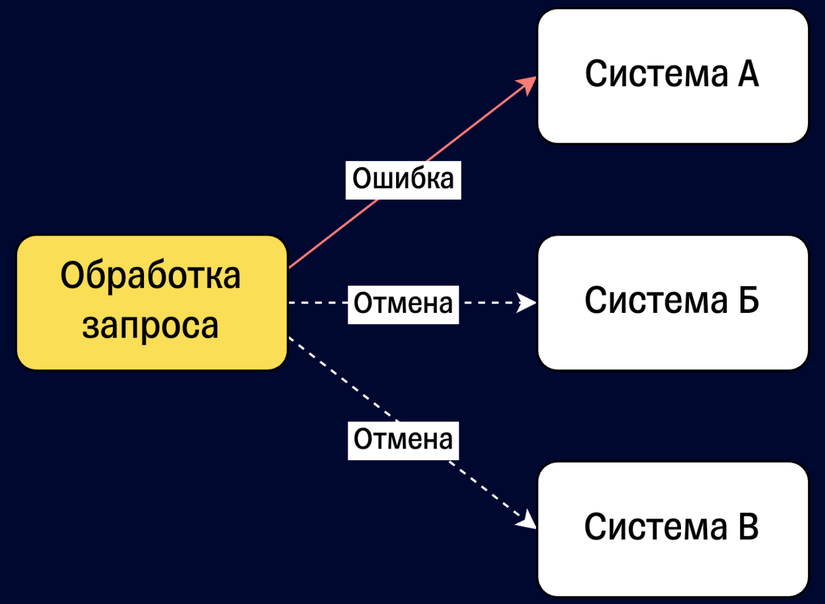
Проблематика

- Например, с помощью сервисов А, Б и В мы можем получить различную информацию о пользователе
- Запросы друг от друга не зависят, а значит мы можем запрашивать данные одновременно
- Оптимизируем время исполнения запроса



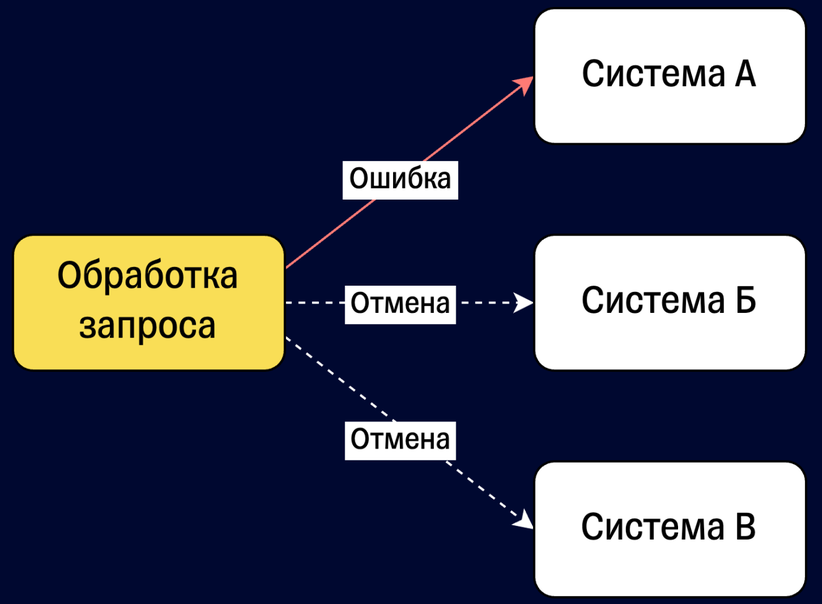
Проблематика

- Сложность таких задач заключается в правильной остановке вычислений
- Нужно уметь правильно задавать стратегию завершения
- Например: При возникновении ошибки при вызове системы А, останавливать вызовы к системе Б и В



Проблематика

- В рамках запросов мы можем выделять ресурсы, которые требуют обязательного закрытия
- Во время отмены должны гарантировать закрытие
- Иначе – риск утечки

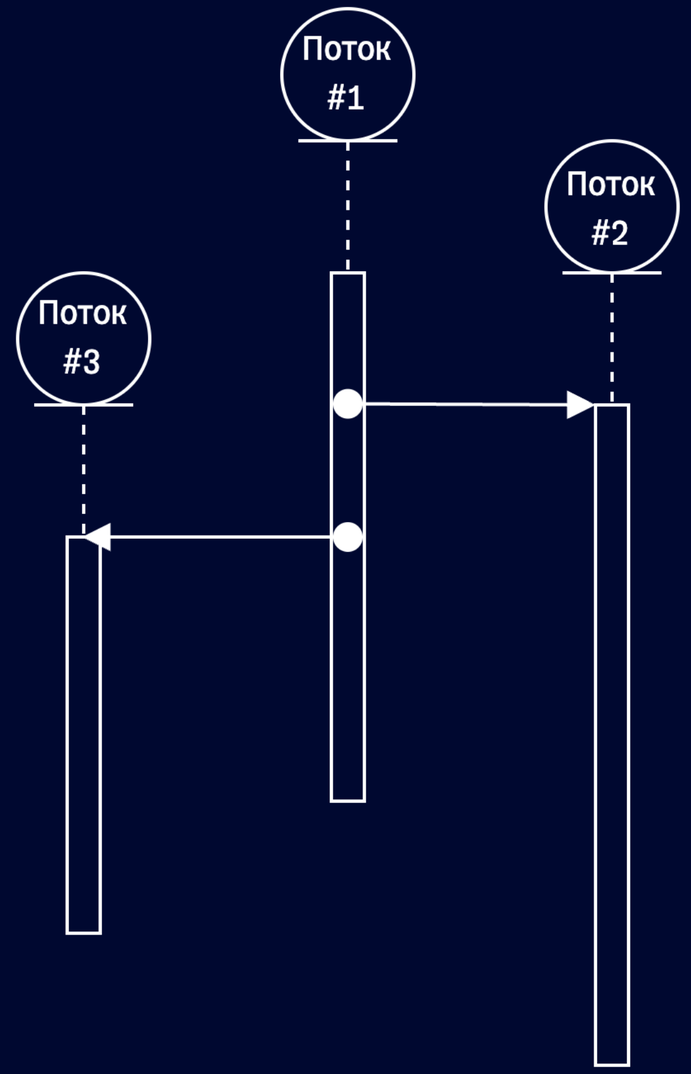


Зачем структура?

- В JDK 21 в превью доступно новое API для написания простого конкурентного кода – StructuredTaskScope
- Использует виртуальные потоки
- Призвано упростить работу с одновременными запросами
- Реализация идеи структурированной конкурентности (structured concurrency)

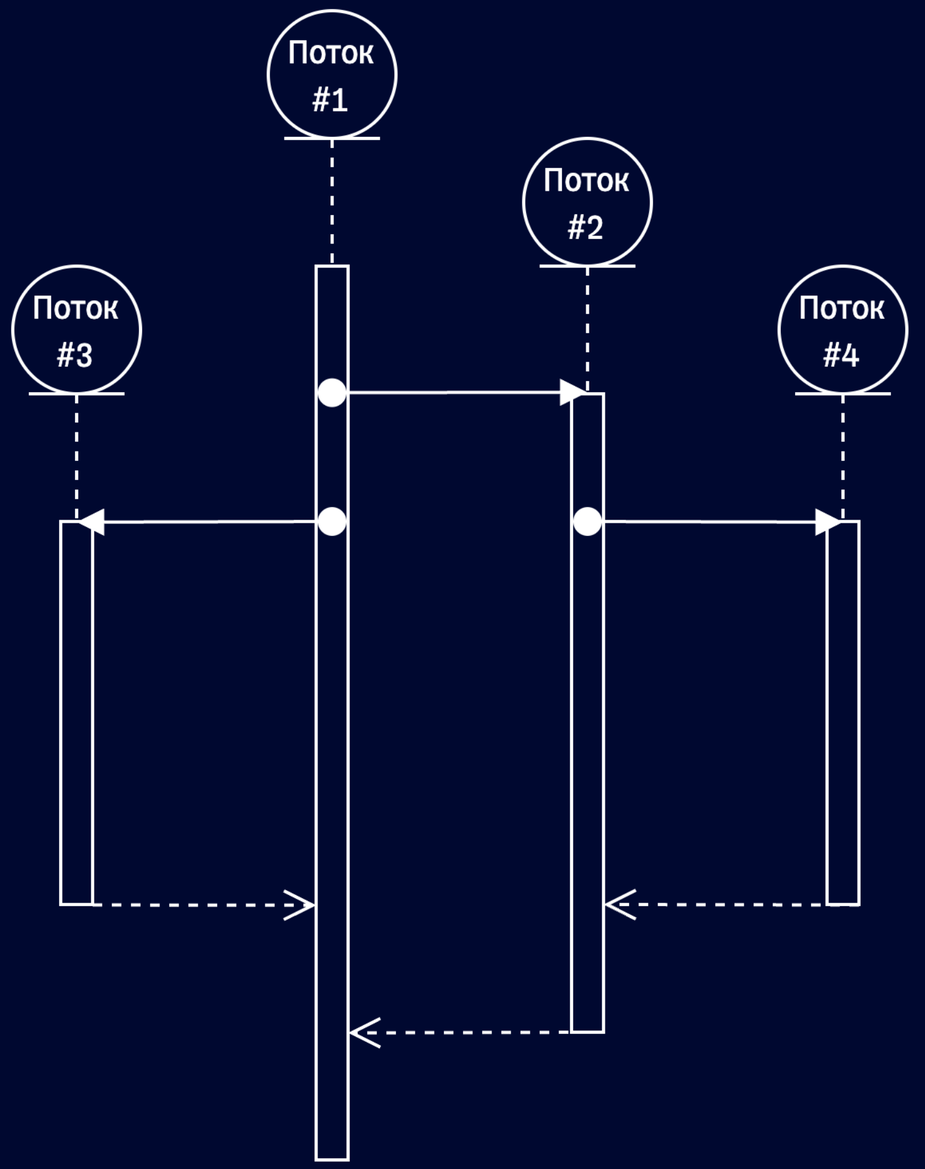
Зачем структура?

- Можно представить запуск потока, как некий аналог **goto** инструкции - запускаем любой блок кода, только конкурентно
- Из многолетнего опыта мы помним, что не стоит использовать **goto**. Нужно держать весь контекст программы в голове, получается спаггетти код
- Вместо этого стоит использовать специальные структуры контроля: **if**, **while**, **for**, вызов функции и т.д.



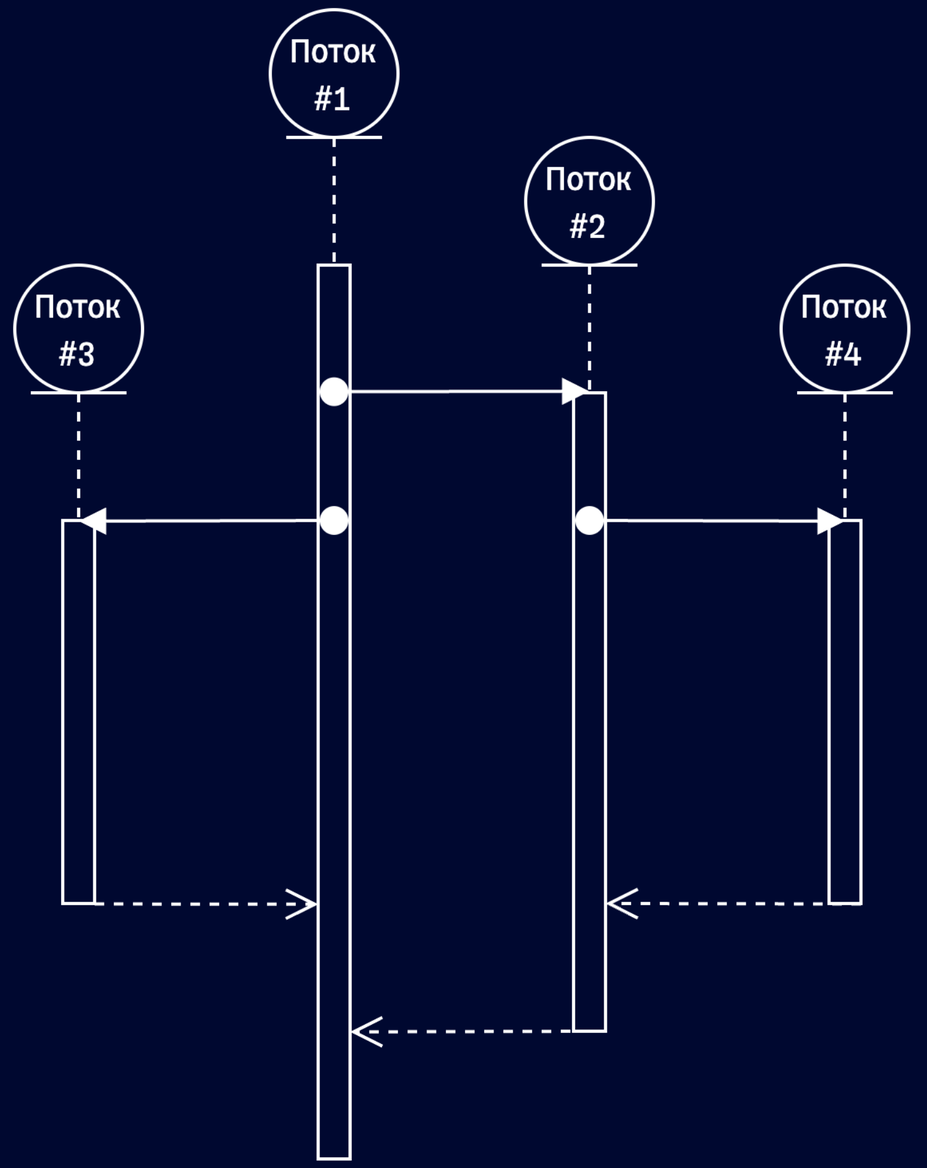
Зачем структура?

- Структурированная конкурентность перекладывает эту идею на потоки
- Любое создание потока подразумевает выстраивание отношения родитель-ребенок
- Ребенок не может пережить родителя
- Родитель занимается наблюдением за детьми



Зачем структура?

- Такой подход позволяет формировать иерархию потоков
- Это уменьшает сложность и повышает понимание кода
- А также помогает легче работать с ресурсами и ошибками
- Можно собрать все ошибки дочерних вычислений, или гарантировать закрытие ресурса



StructuredTaskScope

- StructuredTaskScope позволяет задавать область, в которой можно создавать дочерние задачи

- scope.join() – позволяет дождаться исполнения всех задач

```
def handleRequest(req: Request): Response =
  Using(new StructuredTaskScope[Any]()) { scope =>
    val userNameTask = scope.fork(() => getUserName(req.userId))
    val userAddressTask = scope.fork(() => getUserAddress(req.userId))
    scope.join()
    Response(userNameTask.get(), userAddressTask.get())
  }.get
```

StructuredTaskScope

- Дефолтный StructuredTaskScope дожидается исполнения всех задач
- Если произошла ошибка в одном из запросов, отмены другого запроса не происходит

```
def handleRequest(req: Request): Response =
  Using(new StructuredTaskScope[Any]()) { scope =>
    val userNameTask = scope.fork(() => getUserName(req.userId))
    val userAddressTask = scope.fork(() => getUserAddress(req.userId))
    scope.join()
    Response(userNameTask.get(), userAddressTask.get())
  }.get
```

StructuredTaskScope

- Существует политика - ShutdownOnFailure

- Отмена всех выполняющихся вычислений, если хотя бы в одном произошла ошибка

```
def handleRequest(req: Request): Response =
  Using(new StructuredTaskScope.ShutdownOnFailure()) { scope =>
    val userNameTask = scope.fork(() => getUserName(req.userId))
    val userAddressTask = scope.fork(() => getUserAddress(req.userId))
    scope.join()
    scope.throwIfFailed()
    Response(userNameTask.get(), userAddressTask.get())
  }.get
```

StructuredTaskScope

- Существует вторая политика - ShutdownOnSuccess
- Отмена всех выполняющихся вычислений при получении успеха на одном из вычислений

```
def handleRequest(req: Request): Response =
  Using(new StructuredTaskScope.ShutdownOnSuccess[Response]()) { scope =>
    scope.fork(() => getUserNameFromA(req.userId))
    scope.fork(() => getUserNameFromB(req.userId))
    scope.join().result()
  }.get
```

Другие политики?

- “Из коробки” есть только две политики: ShutdownOnFailure и ShutdownOnSuccess
- Любые другие политики через наследование StructuredTaskScope и реализацию метода handleComplete
- Например, если понадобиться получать все успешные результаты или делать отмену, только в случае определенных ошибок

```
class CollectAllResults[T]
  extends StructuredTaskScope[T] {

  var innerResults: mutable.Buffer[T] =
mutable.Buffer.empty[T]

  override def handleComplete(
      subtask: StructuredTaskScope.Subtask[_ <:
T]
  ): Unit =
    if (subtask.state() == State.SUCCESS) {
      innerResults.addOne(subtask.get())
    }

  def results(): List[T] =
    innerResults.toList
}
```

Другие политики?

- Имхо, не сильно удобно – много лишнего кода
- Возможно решится появлением новых инструментов-библиотек

```
class CollectAllResults[T]
  extends StructuredTaskScope[T] {

  var innerResults: mutable.Buffer[T] =
mutable.Buffer.empty[T]

  override def handleComplete(
      subtask: StructuredTaskScope.Subtask[_ <:
T]
  ): Unit =
    if (subtask.state() == State.SUCCESS) {
      innerResults.addOne(subtask.get())
    }

  def results(): List[T] =
    innerResults.toList
}
```

Другие политики?

- Имхо, не сильно удобно – много лишнего кода
- Возможно решится появлением новых инструментов-библиотек
- Еще нужно помнить про конкурентный доступ до состояния. Могут возникать гонки между вычислениями

```
class SafeCollectAllResults[T]
  extends StructuredTaskScope[T] {

  val queue: ConcurrentLinkedQueue[T] =
    new ConcurrentLinkedQueue[T]()

  override def handleComplete(
      subtask: StructuredTaskScope.Subtask[_ <:
T]
  ): Unit = {
    if (subtask.state() == State.SUCCESS)
      queue.add(subtask.get())
  }

  def results: Stream[T] = queue.stream()
}
```

Thread.interrupt()

- Project Loom предлагает использовать для остановки вычислений Thread.interrupt
- Давно известно, что это не самое удобное API

```
class Thread {
  def start(): Unit = ???
  def join(): Unit = ???
  def interrupt(): Unit = ???
}
```

Алгоритм

- Все вычисления по умолчанию - неотменяемые
- У каждого потока есть флаг, который говорит о том, остановлен ли поток
- Thread.interrupt просто устанавливает флаг для вызываемого потока

Алгоритм

- Все вычисления по умолчанию - неотменяемые
- У каждого потока есть флаг, который говорит о том, остановлен ли поток
- Thread.interrupt просто устанавливает флаг для вызываемого потока
- После вызова Thread.interrupt нельзя быть уверенными, что поток остановлен

Алгоритм

- Внутри потока нужно самостоятельно проверять флаг через Thread.interrupted или Thread.isInterrupted
- Важно очищать ресурсы, которые были выделены для потока

```
def doStuff(): Unit =
  while (true) {
    if (Thread.interrupted()) {
      // Тут освобождаем ресурсы,
аллоцированные для потока
      throw new InterruptedException();
    }
    // Делаем какие-то вычисления в цикле
  }
```

Алгоритм

- Внутри потока нужно самостоятельно проверять флаг через Thread.interrupted или Thread.isInterrupted
- Важно очищать ресурсы, которые были выделены для потока
- Остановка вычисления моделируется через InterruptedException

```
def doStuff(): Unit =
  while (true) {
    if (Thread.interrupted()) {
      // Тут освобождаем ресурсы,
аллоцированные для потока
      throw new InterruptedException();
    }
    // Делаем какие-то вычисления в цикле
  }
```

Сложности

- Разберем на примере
- Задача – нужно *периодически* выгружать несколько файлов на удаленный сервер

Сложности

- Разберем на примере
- Задача – нужно *периодически* выгружать несколько файлов на удаленный сервер
- Ограничения
  - Сеть может сбоить. Нужны ретраи
  - При остановке процесса, нужно заканчивать текущую выгрузку файлов
  - Ошибки не приводят к падению процесса

Сложности

- Разберем на примере
- Задача – нужно *периодически* выгружать несколько файлов на удаленный сервер
- Ограничения
  - Сеть может сбоить. Нужны ретраи
  - При остановке процесса, нужно заканчивать текущую выгрузку файлов
  - Ошибки не приводят к падению процесса

```
def uploadDataWithRetry(file: File): Unit = {
  var retryFlag = false
  var attempt   = 0
  do {
    try {
      Thread.sleep(attempt * 1_000L)
      uploadFile(file)
      retryFlag = false
    } catch {
      case _: TimeoutException if attempt < 3 =>
        retryFlag = true
        attempt += 1
    }
  } while (retryFlag)
}
```

Сложности

- Разберем на примере
- Задача – нужно *периодически* выгружать несколько файлов на удаленный сервер
- Ограничения
  - Сеть может сбоить. Нужны ретраи
  - При остановке процесса, нужно заканчивать текущую выгрузку файлов
  - Ошибки не приводят к падению процесса

```
def uploadDataWithRetry(file: File): Unit = {
  var retryFlag = false
  var attempt   = 0
  do {
    try {
      Thread.sleep(attempt * 1_000L)
      uploadFile(file)
      retryFlag = false
    } catch {
      case _: TimeoutException if attempt < 3 =>
        retryFlag = true
        attempt += 1
      case _: Throwable =>
        retryFlag = false
    }
  } while (retryFlag)
}
```

Сложности

- Разберем на примере
- Задача – нужно *периодически* выгружать несколько файлов на удаленный сервер
- Ограничения
  - Сеть может сбоить. Нужны ретраи
  - При остановке процесса, нужно заканчивать текущую выгрузку файлов
  - Ошибки не приводят к падению процесса

```
def uploadAll(period: Long): Unit = {
  while (true) {
    Thread.sleep(period)

    uploadDataWithRetry("one-file")
    uploadDataWithRetry("two-file")
  }
}
```

Сложности

- С первого взгляда такой пример – корректный
- Мы ожидаем, что на Thread.sleep будет работать прерывание
- И оно, действительно, будет работать, если в момент прерывания будет исполняться Thread.sleep

```
def uploadAll(period: Long): Unit = {
  while (true) {
    Thread.sleep(period)

    uploadDataWithRetry("one-file")
    uploadDataWithRetry("two-file")
  }
}
```

Сложности

- Но, если исполнение будет в uploadDataWithRetry, то флаг прерывания будет очищен внутренним Thread.sleep
- И прерывание будет проигнорировано

```
def uploadAll(period: Long): Unit = {
  while (true) {
    Thread.sleep(period)

    uploadDataWithRetry("one-file")
    uploadDataWithRetry("two-file")
  }
}
```

Сложности

- Но, если исполнение будет в uploadDataWithRetry, то флаг прерывания будет очищен внутренним Thread.sleep
- И прерывание будет проигнорировано
- Даже добавление Thread.interrupted не поможет. Флаг уже будет очищен

```
def uploadAll(period: Long): Unit = {
  while (true) {
    if (Thread.interrupted())
      return

    Thread.sleep(period)

    uploadDataWithRetry("one-file")
    uploadDataWithRetry("two-file")
  }
}
```

Сложности

- Но, если исполнение будет в uploadDataWithRetry, то флаг прерывания будет очищен внутренним Thread.sleep
- И прерывание будет проигнорировано
- Даже добавление Thread.interrupted не поможет. Флаг уже будет очищен
- Решение – прокидывать флаг прерывания из функции uploadDataWithRetry

```
def uploadAll(period: Long): Unit = {
  var isInterrupted = false

  while (!isInterrupted) {
    Thread.sleep(period)

    isInterrupted =
      uploadDataWithRetry("one-file")
    isInterrupted =
      uploadDataWithRetry("two-file")
  }
}
```

Сложности

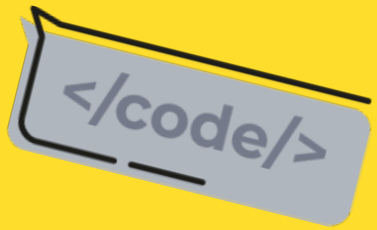
- Итого:
  - Решение довольно большое
  - Важно держать состояние флага прерывания на каждом этапе
  - Важно понимать, когда и как возвращается InterruptedException
  - Большой контекст в голове, легко совершить ошибку

```
def uploadAll(period: Long): Unit = {
  var isInterrupted = false

  while (!isInterrupted) {
    Thread.sleep(period)

    isInterrupted =
      uploadDataWithRetry("one-file")
    isInterrupted =
      uploadDataWithRetry("two-file")
  }
}
```

«While this mechanism does address a real need, it is error-prone, and we’d like to revisit it. We’ve experimented with some prototypes, but, for the moment, don’t have any concrete proposals to present.»

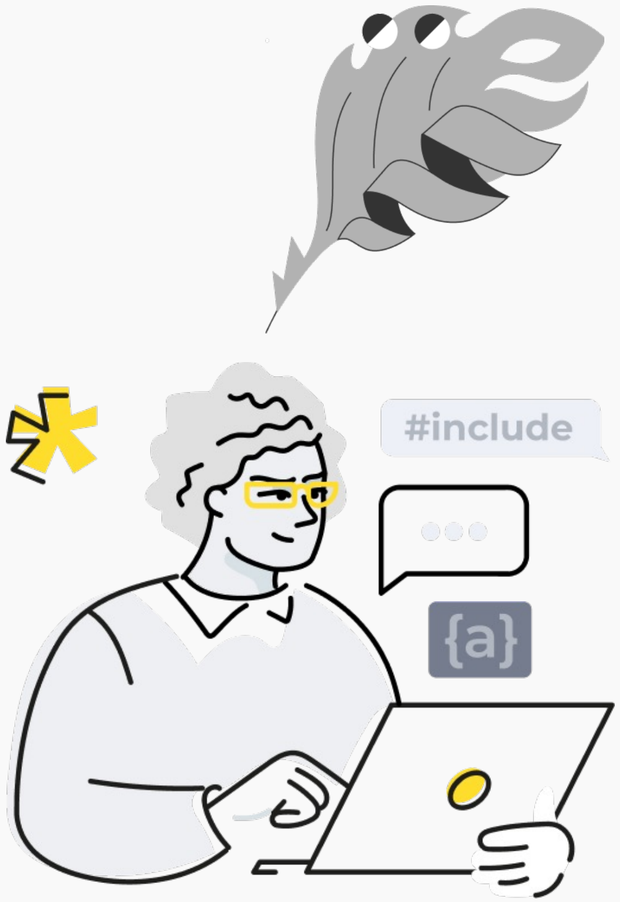
State of Loom: Part 2. Авторы Project Loom

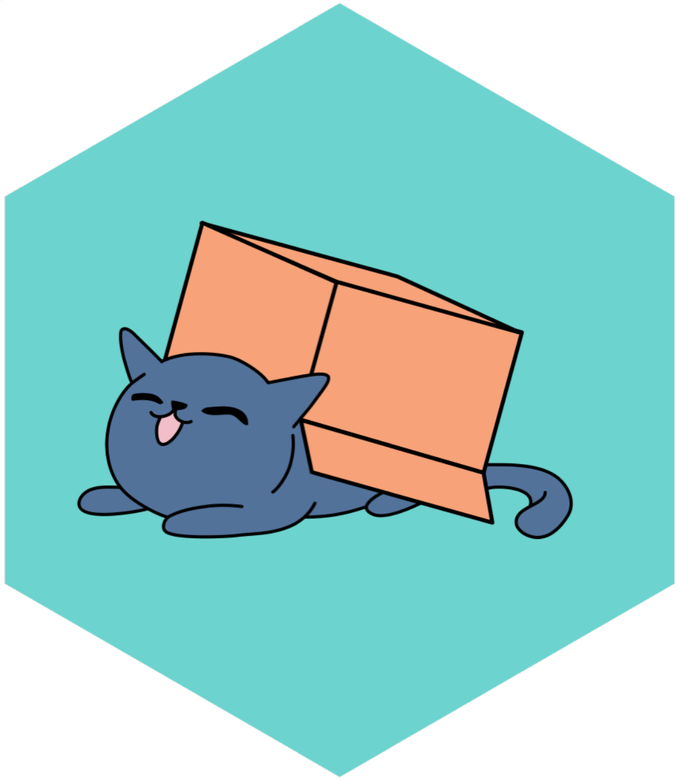
Как схожая задача решается в Scala?



- Как выглядит структурированная конкурентность в Scala?
- Как работает отмена?

Системы эффектов





cats-effect

ZIO

Как писать код?

Системы эффектов по типу cats-effect позволяют все также писать последовательный код, но в новой манере

Как писать код?

Системы эффектов по типу cats-effect позволяют все также писать последовательный код, но в новой манере

```
def handleRequest(req: Request): Response = {
  validateRequest(req)
  val data = enrichWithData(req)
  val response = saveToDatabase(data)
  response
}
```

Как писать код?

Системы эффектов по типу cats-effect позволяют все также писать последовательный код, но в новой манере

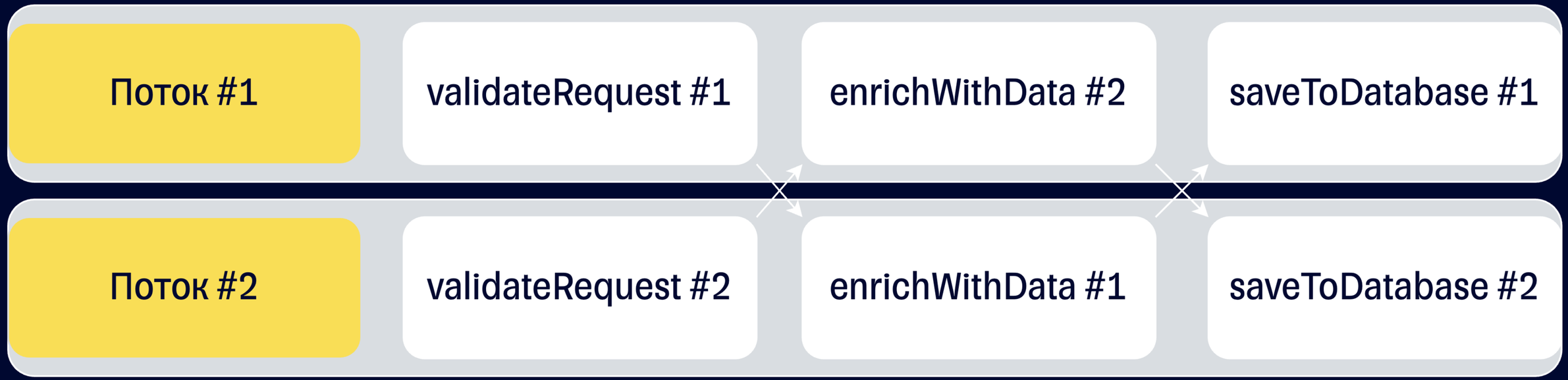
```
def handleRequest(req: Request): Response = {
  validateRequest(req)
  val data = enrichWithData(req)
  val response = saveToDatabase(data)
  response
}
```

```
def handleRequest(req: Request): IO[Reponse] =
  for {
    _        <- validateRequest(req)
    data     <- enrichWithData(req)
    response <- saveToDatabase(data)
  } yield response
```

- Все вычисления теперь обернуты в структуру данных IO
- IO является монадой, поэтому можно использовать синтаксис с for

Как оно работает?

- Наша программа теперь состоит из множества маленьких IO вычислений
- Каждое IO вычисление может исполняться на отдельном потоке



Отмена у cats-effect

- Все IO вычисления по умолчанию – отменяемые

```scala
trait Fiber[A] {
  def join: IO[Outcome[A]]
  def cancel: IO[Unit]
}
```

Отмена у cats-effect

- Все IO вычисления по умолчанию – отменяемые
- Схожий с Thread интерфейс у Fiber

```scala
trait Fiber[A] {
  def join: IO[Outcome[A]]
  def cancel: IO[Unit]
}
```

Отмена у cats-effect

- Все IO вычисления по умолчанию – отменяемые
- Схожий с Thread интерфейс у Fiber
- Между каждыми IO вычислениями есть проверка на отмену
- Цепочку IO вычислений всегда можно остановить на границе двух IO вычислений

```scala
trait Fiber[A] {
  def join: IO[Outcome[A]]
  def cancel: IO[Unit]
}

def loop: IO[Unit] =
  for {
    _ <- IO.println("Hello!")
    _ <- loop
  } yield ()
```

Отмена у cats-effect

- Все IO вычисления по умолчанию – отменяемые
- Схожий с Thread интерфейс у Fiber
- Между каждыми IO вычислениями есть проверка на отмену
- Цепочку IO вычислений всегда можно остановить на границе двух IO вычислений

```scala
trait Fiber[A] {
  def join: IO[Outcome[A]]
  def cancel: IO[Unit]
}

def loop: IO[Unit] =
  for {
    _ <- IO.println("Hello!")
    _ <- loop
  } yield ()

def program: IO[Unit] =
  for {
    fib: Fiber[Unit] <- loop.start
    _ <- fib.cancel
    _ <- IO.println("Finished")
  } yield ()
```

Отмена у cats-effect

- Все IO вычисления по умолчанию – отменяемые
- Схожий с Thread интерфейс у Fiber
- Между каждыми IO вычислениями есть проверка на отмену
- Цепочку IO вычислений всегда можно остановить на границе двух IO вычислений

Гарантия отмены

- Fiber.cancel работает по-другому

Гарантия отмены

- Fiber.cancel работает по-другому
- Во время своего исполнения дожидается все финализирующие действия

```scala
def loop: IO[Unit] =
  IO
    .println("Hello!")
    .foreverM
    .guarantee(IO.println("Finished
loop"))
```

Гарантия отмены

- Fiber.cancel работает по-другому
- Во время своего исполнения дожидается все финализирующие действия

```scala
def loop: IO[Unit] =
  IO
    .println("Hello!")
    .foreverM
    .guarantee(IO.println("Finished
loop"))
```

Гарантия отмены

- Fiber.cancel работает по-другому
- Во время своего исполнения дожидается все финализирующие действия
- Все, что должно быть тщательно завершено – будет завершено

```scala
def loop: IO[Unit] =
  IO
    .println("Hello!")
    .foreverM
    .guarantee(IO.println("Finished
loop"))

def program: IO[Unit] =
  for {
    fib: Fiber[Unit] <- loop.start
    _ <- fib.cancel
    // "Finished loop" уже будет
остановлен
    _ <- IO.println("Finished")
  } yield ()
```

Гарантия отмены

- Fiber.cancel работает по-другому
- Во время своего исполнения дожидается все финализирующие действия
- Все, что должно быть тщательно завершено – будет завершено

```scala
def loop: IO[Unit] =
  IO
    .println("Hello!")
    .foreverM
    .guarantee(IO.println("Finished
loop"))

def program: IO[Unit] =
  for {
    fib: Fiber[Unit] <- loop.start
    _ <- fib.cancel
    // "Finished loop" уже будет
остановлен
    _ <- IO.println("Finished")
  } yield ()
```

Гарантия отмены

- Fiber.cancel работает по-другому
- Во время своего исполнения дожидается все финализирующие действия
- Все, что должно быть тщательно завершено – будет завершено
- Такой подход позволяет удобнее работать с ресурсами (сам ресурс с его логикой завершения можно объявить отдельно от использования)

```scala
sealed trait Outcome[A]
object Outcome {
  case class Succeded[A](a: A)
    extends Outcome[A]

  case class Errored[A](
    e: Throwable
  ) extends Outcome[A]

  case class Cancelled[A]()
    extends Outcome[A]
}
```

Обработка результата

- Fiber.join возвращает явный результат работы файбера
- Возможны 3 исхода:
  - Успешный результат
  - Завершение с ошибкой
  - Отмена вычисления

```scala
def program: IO[Unit] =
  for {
    fib: Fiber[Unit] <- loop.start
    _ <- fib.cancel
    outcome <- fib.join
    _ <- outcome match {
      case Outcome.Succeded(_) =>
        IO.println("Success")
      case Outcome.Errored(_) =>
        IO.println("Error")
      case Outcome.Cancelled() =>
        IO.println("Cancel")
    }
  } yield ()
```

Обработка результата

- Fiber.join возвращает явный результат работы файбера
- Возможны 3 исхода:
  - Успешный результат
  - Завершение с ошибкой
  - Отмена вычисления

```scala
def program: IO[Unit] =
  for {
    fib: Fiber[Unit] <- loop.start
    _ <- fib.cancel
    outcome <- fib.join
    _ <- outcome match {
      case Outcome.Succeded(_) =>
        IO.println("Success")
      case Outcome.Errored(_) =>
        IO.println("Error")
      case Outcome.Cancelled() =>
        IO.println("Cancel")
    }
  } yield ()
```

Обработка результата

- Fiber.join возвращает явный результат работы файбера
- Возможны 3 исхода:
  - Успешный результат
  - Завершение с ошибкой
  - Отмена вычисления
- Всегда требуется явная обработка случая отмены

```scala
def uploadDataWithRetry(
    file: File,
    attempts: Int = 3
): IO[Unit] =
  uploadFile(file)
    .handleErrorWith {
      case _: TimeoutException if attempts > 0 =>
        uploadDataWithRetry(
          file,
          attempts = attempts - 1
        )
          .delayBy((4 - attempts).seconds)
    }
    .uncancelable
```

Пример с загрузкой

- uncancellable оборачивает выделенную цепочку IO выражений и не позволяет ее разорвать при отмене
- При отмене цепочка разорвется за границей uncancellable

```scala
def uploadAll(
    period: FiniteDuration
): IO[Unit] =
  IO
    .sleep(period)
    .flatMap(_ =>
      {
        for {
          _ <- uploadDataWithRetry("one-file")
          _ <- uploadDataWithRetry("two-file")
        } yield ()
      }.uncancelable
    )
    .foreverM
```

Пример с загрузкой

- uncancellable оборачивает выделенную цепочку IO выражений и не позволяет ее разорвать при отмене
- При отмене цепочка разорвется за границей uncancellable

Конкурентность

- Все это позволяет реализовать простое API для сложных вычислений
  - Например, для одновременных запросов через both

```
def handleRequestOne(req: Request): IO[Response] =
  for {
    (resA, resB) <- callServiceA(req).both(callServiceB(req))
      ...
  } yield response
```

Конкурентность

- Все это позволяет реализовать простое API для сложных вычислений
- Или, например, для гонок запросов через race

```
def handleRequestTwo(req: Request): IO[Response] =
  for {
    res <- callNodeA(req).race(callNodeB(req)).map(_.merge)
      ...
  } yield response
```

Конкурентность

- Поверх можно естественно реализовывать любую стратегию
- Например, получение всех успешных результатов

```
def collectAll[A](ios: List[IO[A]]): IO[List[A]] =
  ios
    .map(_.attempt)
    .parSequence
    .map(
      _.collect {
        case Right(value) => value
      }
    )
```

Давайте подведем итоги

Итоги

- Project Loom делает верные шаги с введением StructuredTaskScope
- Подходит для простых случаев. Для более сложных – тяжело. Появление новых библиотек/инструментов может поменять ситуацию
- Использовать Thread.interrupt для отмены вычислений тяжело и может приводить к ошибкам
- cats-effect предлагает альтернативную модель для отмены вычислений, более прозрачную и простую в освоении


Вопросы?