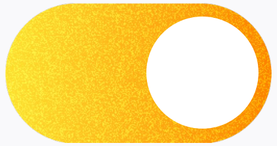


Online

Иван Лягаев | Главный разработчик

Культура отмены

Как правильно отменять вычисления?

Иван Лягаев

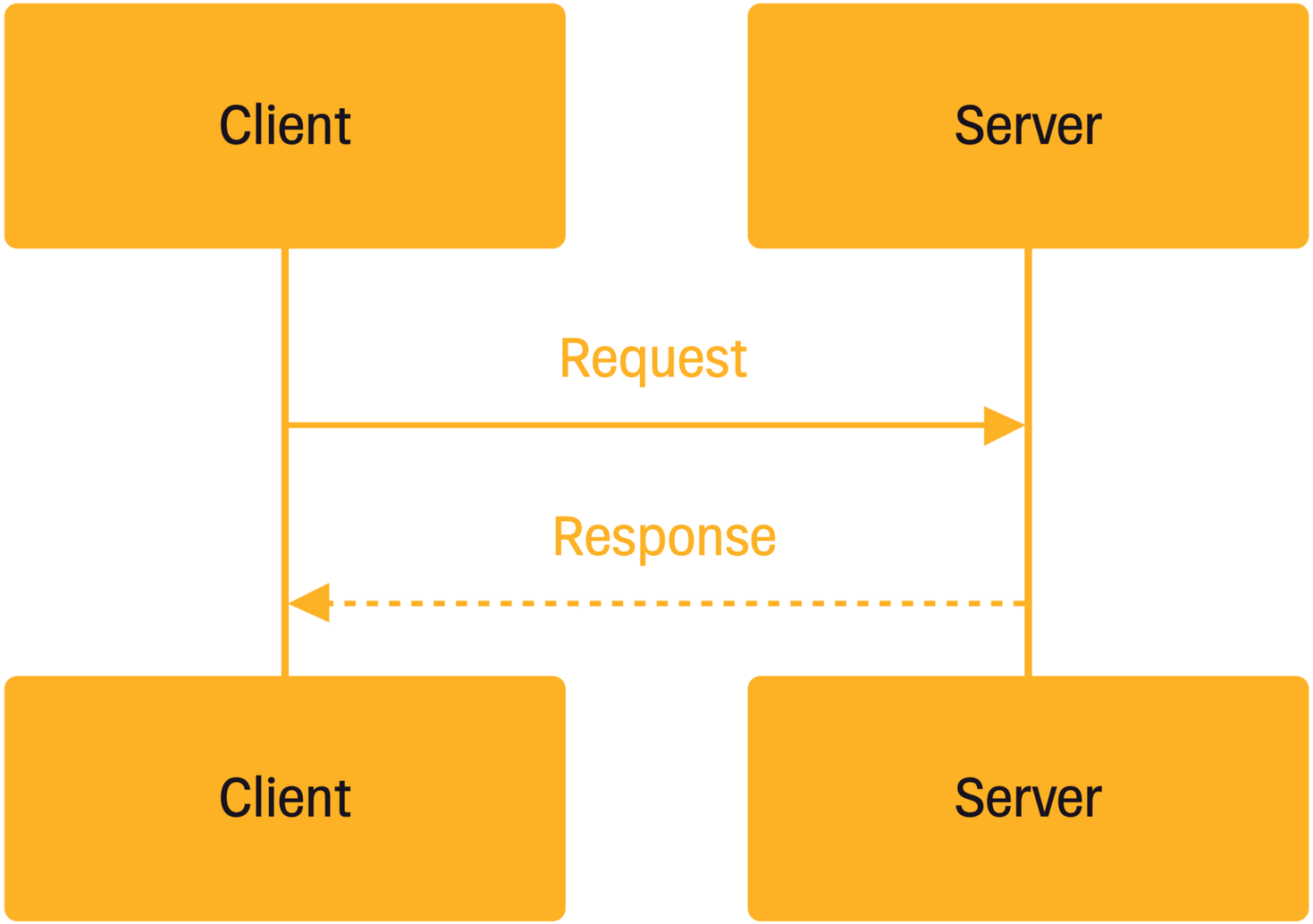
B2B Tech, Т-Бизнес

- Занимаюсь разработкой библиотек и инструментов для Scala разработчиков
- Развиваю профессию Scala в Т-Банке
- В Scala 7+ лет

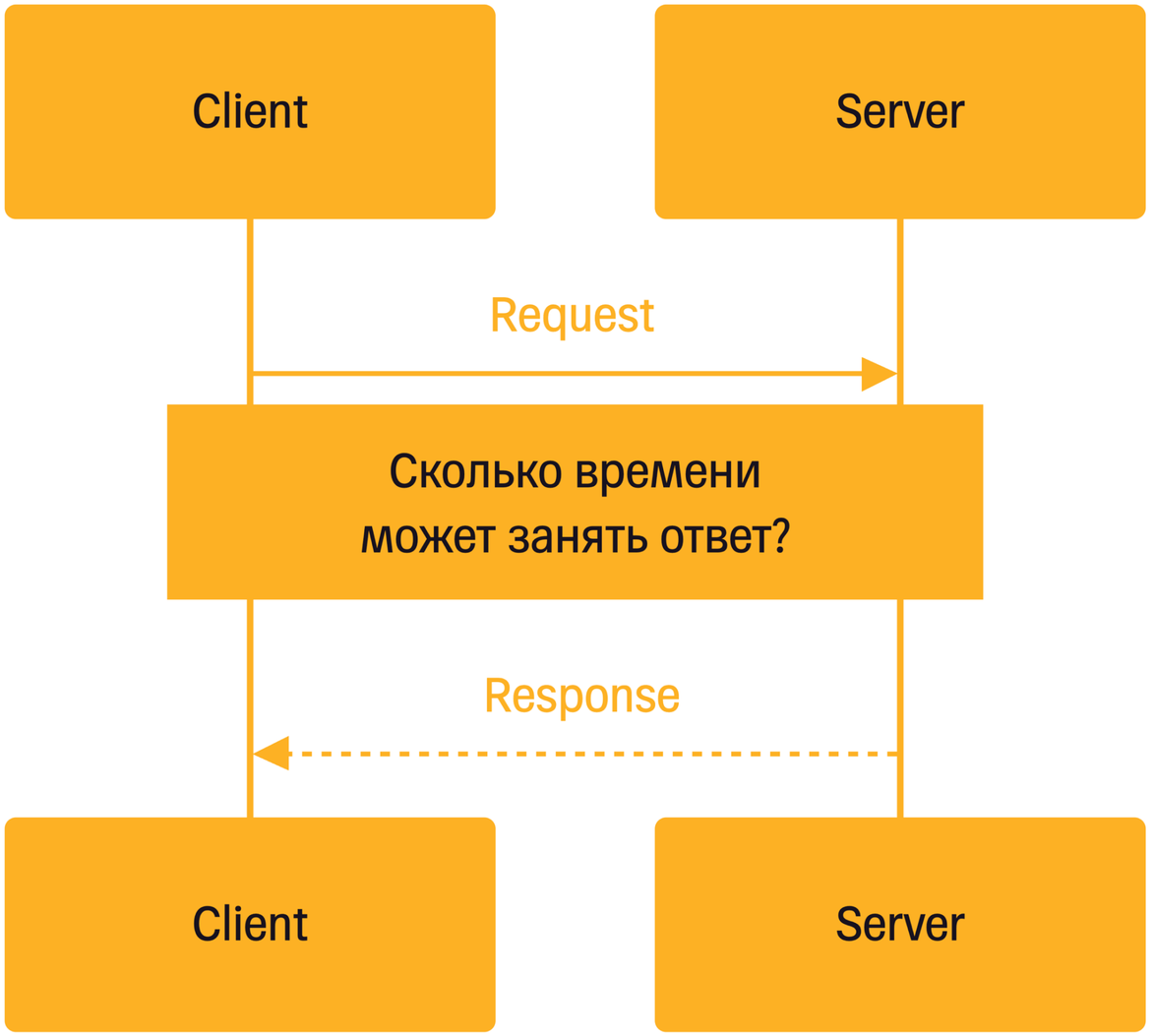
О чем сегодня поговорим?

- Почему стоит отменять вычисления? Почему это сложно?
- Как реализуется отмена стандартными средствами Java?
- Как реализуется отмена в системах эффектов Scala?
- Сравнение подходов и выводы

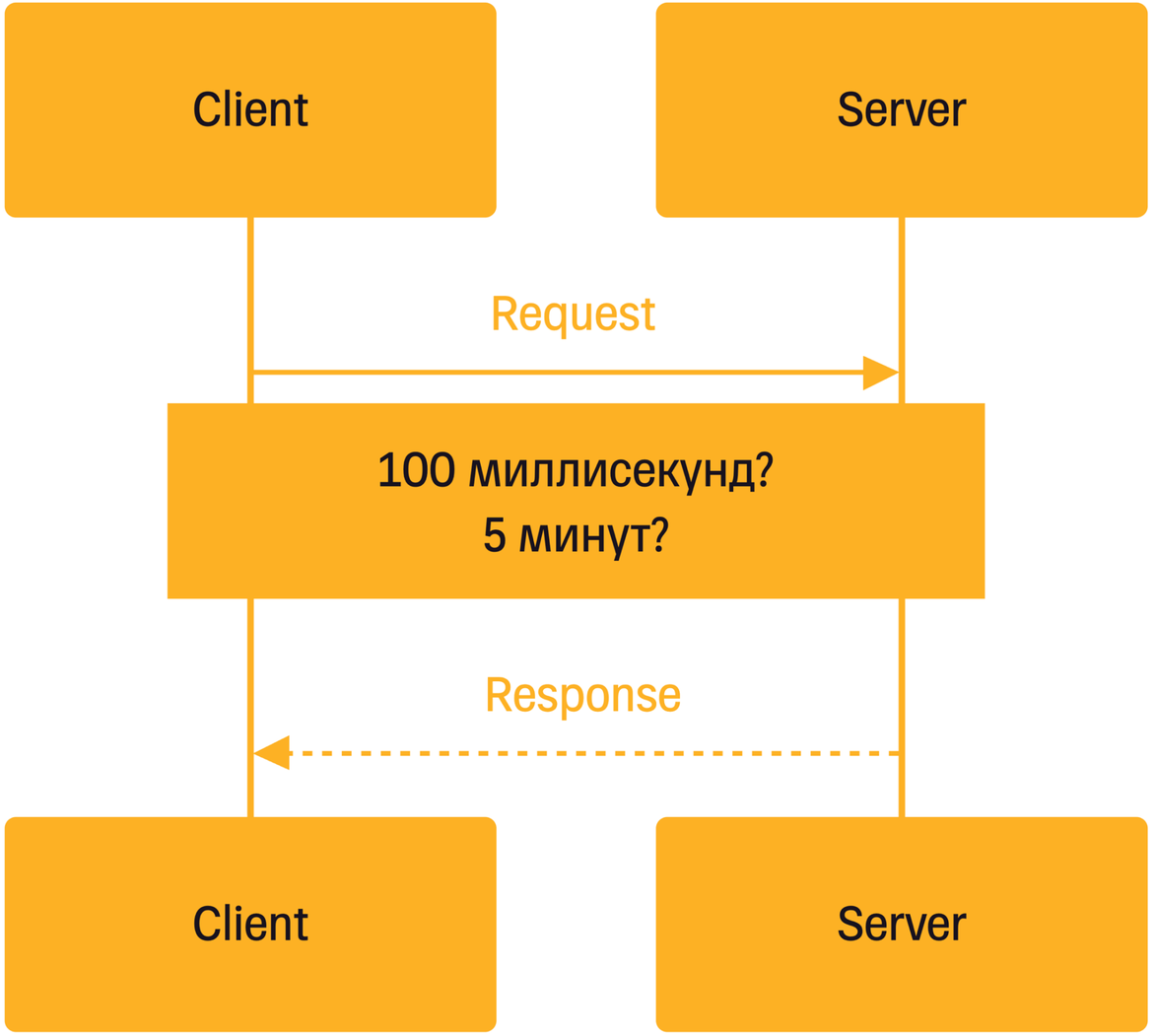
Почему стоит отменять?



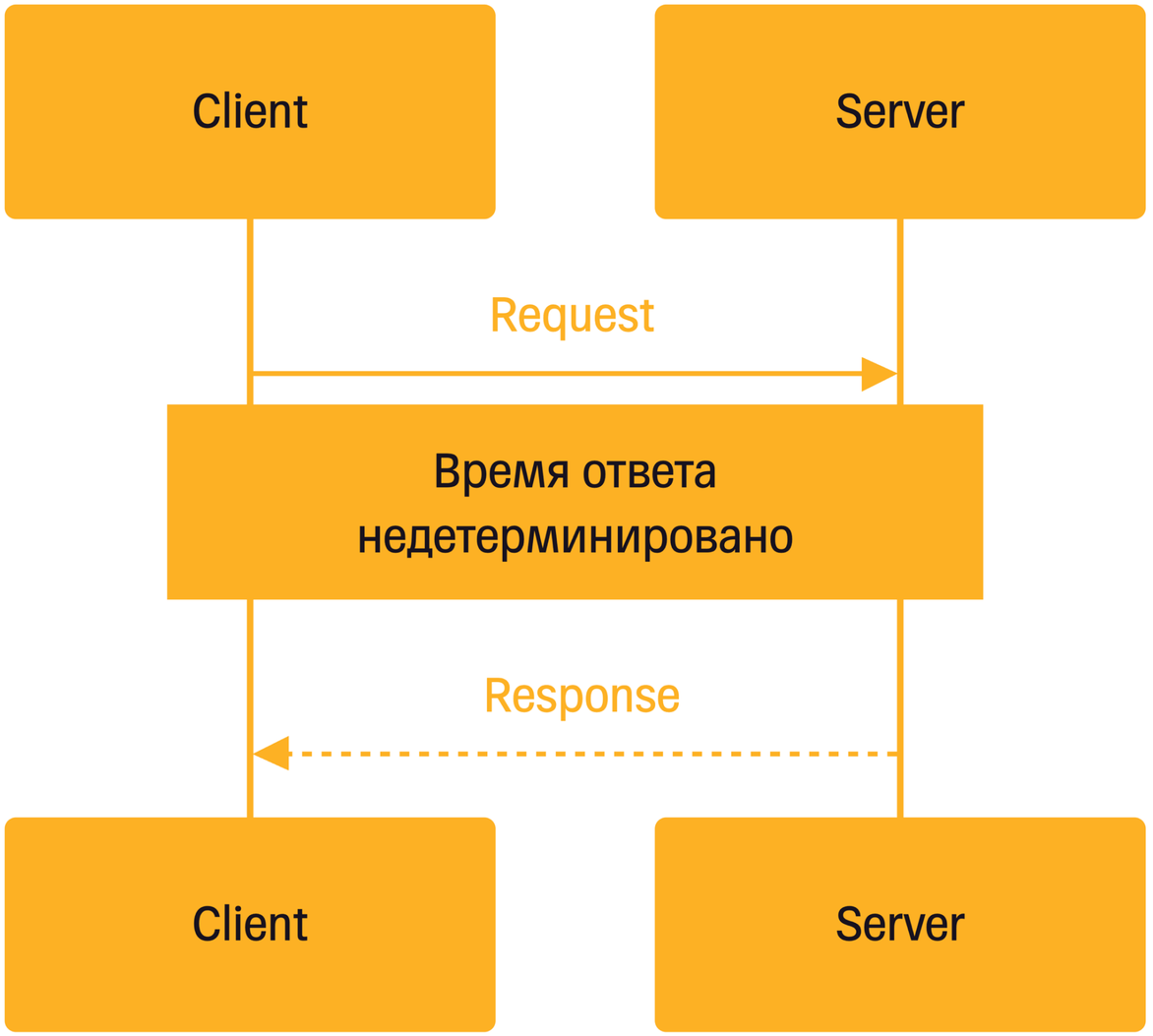
Почему стоит отменять?



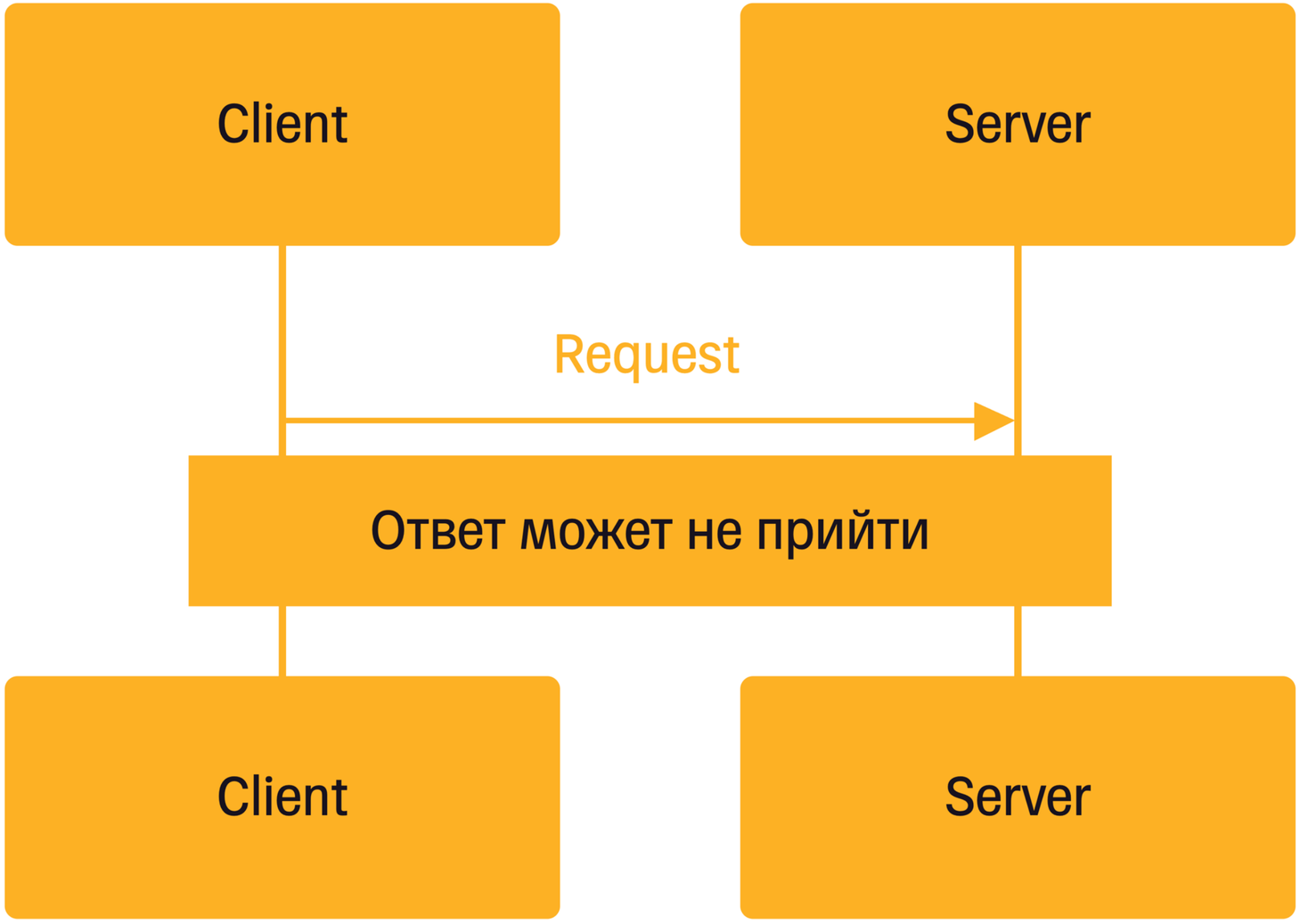
Почему стоит отменять?



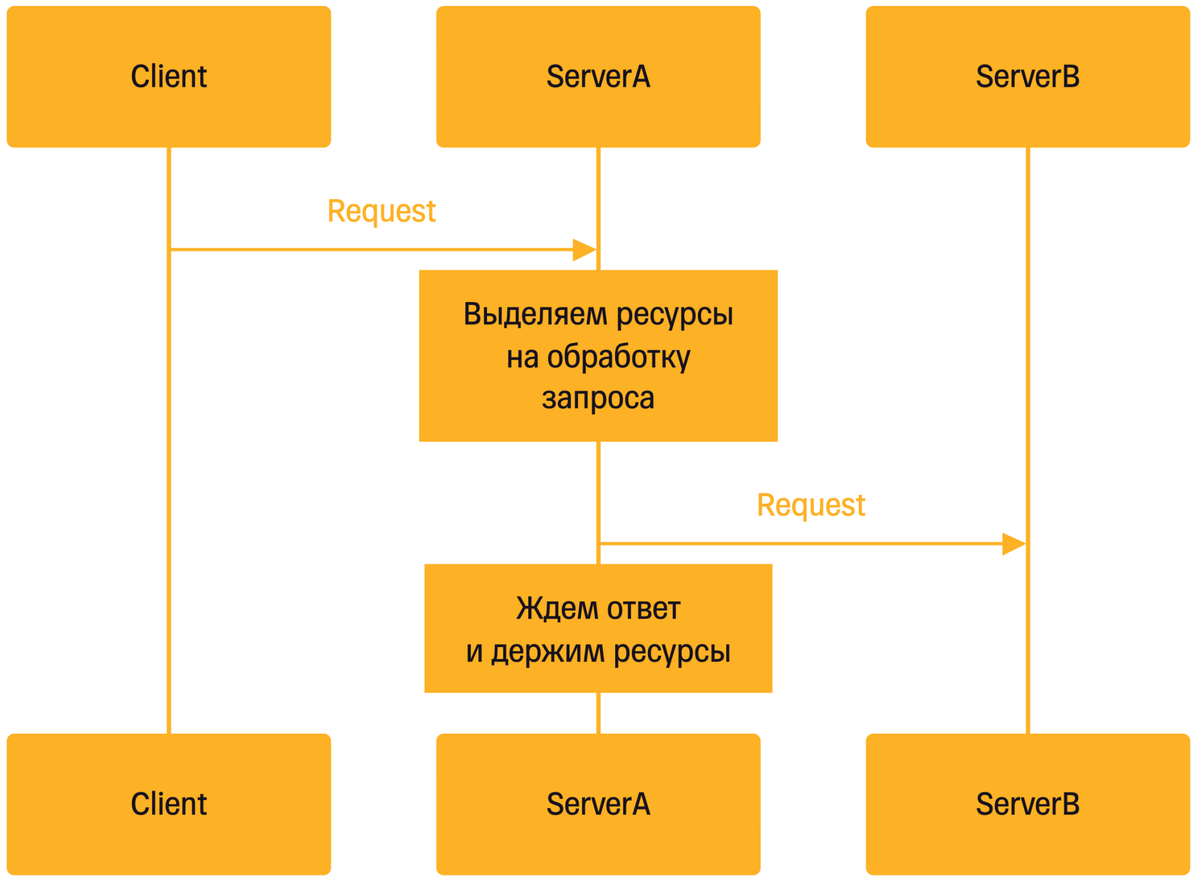
Почему стоит отменять?



Почему стоит отменять?

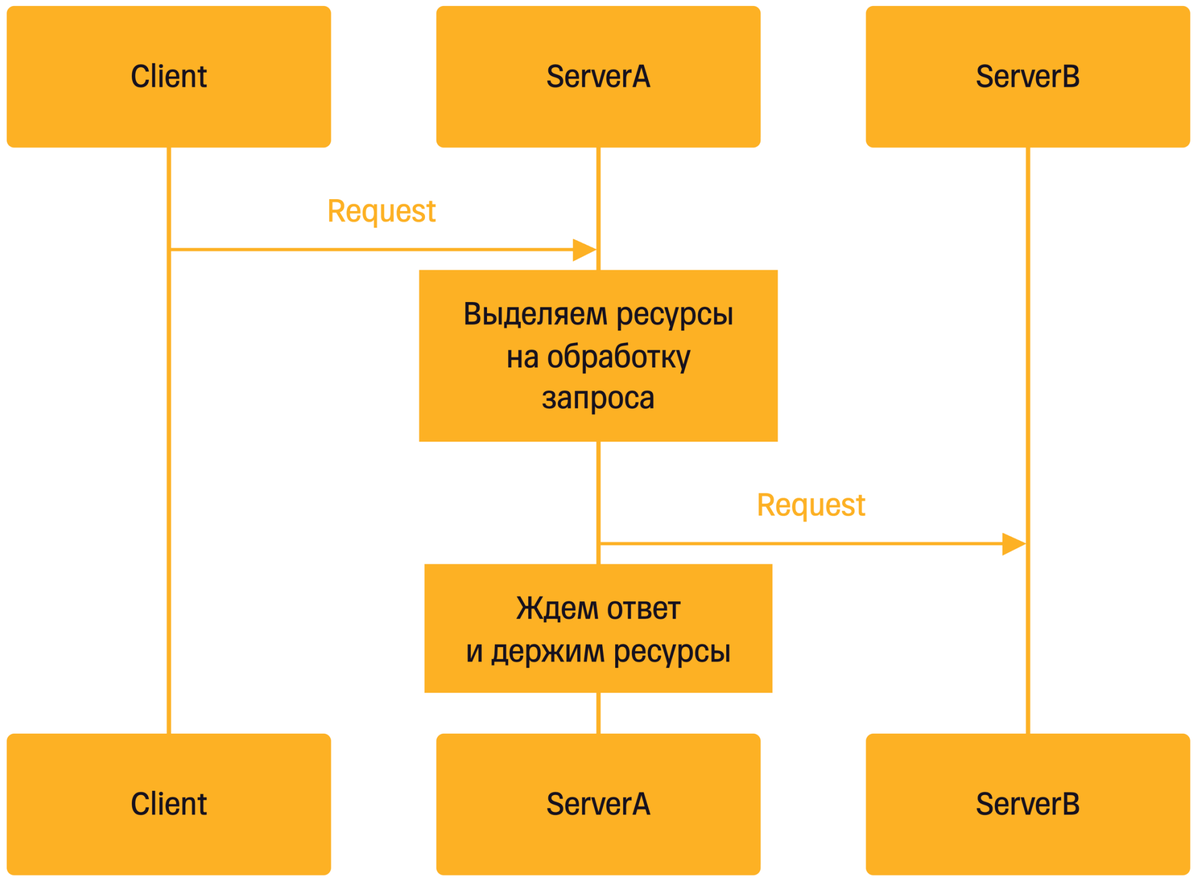


Почему стоит отменять?



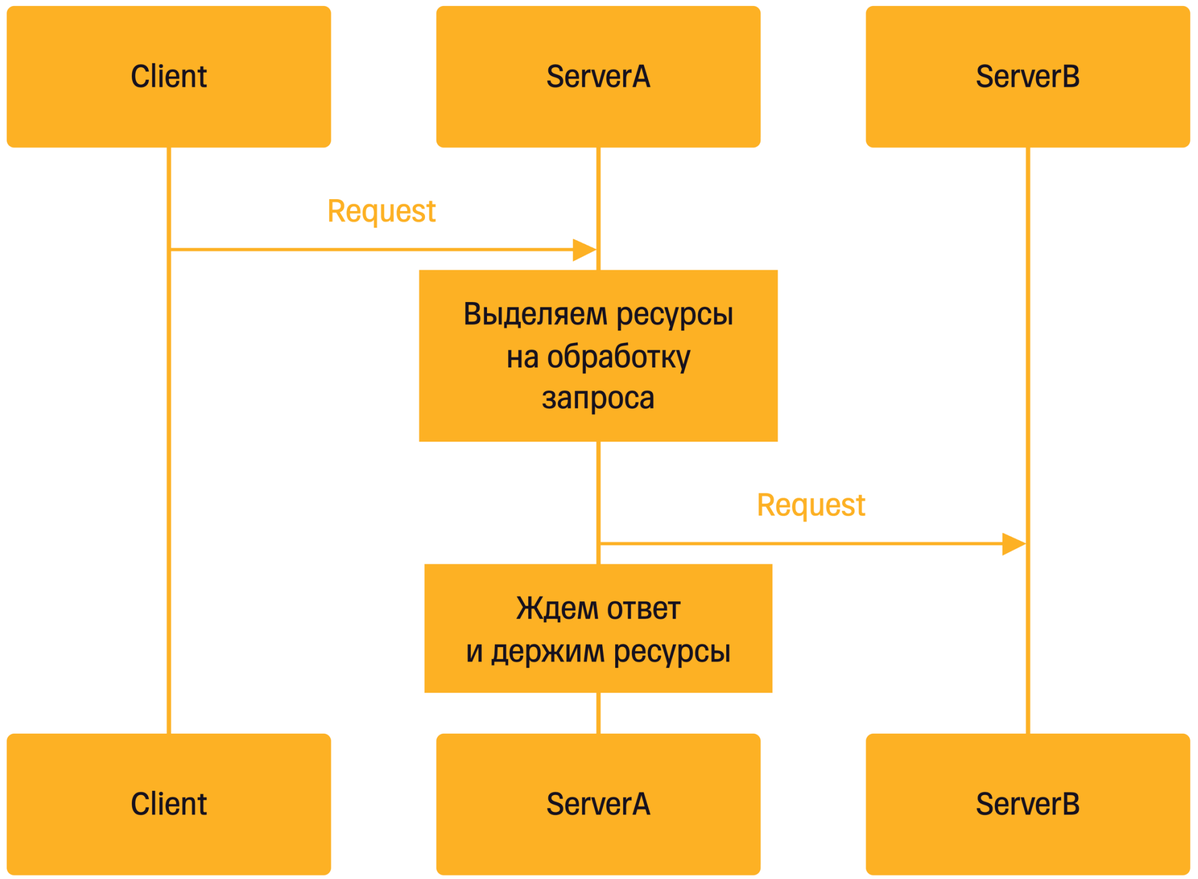
Почему стоит отменять?

- Каждый запрос тратит ресурсы: CPU, RAM, соединения и т.д.



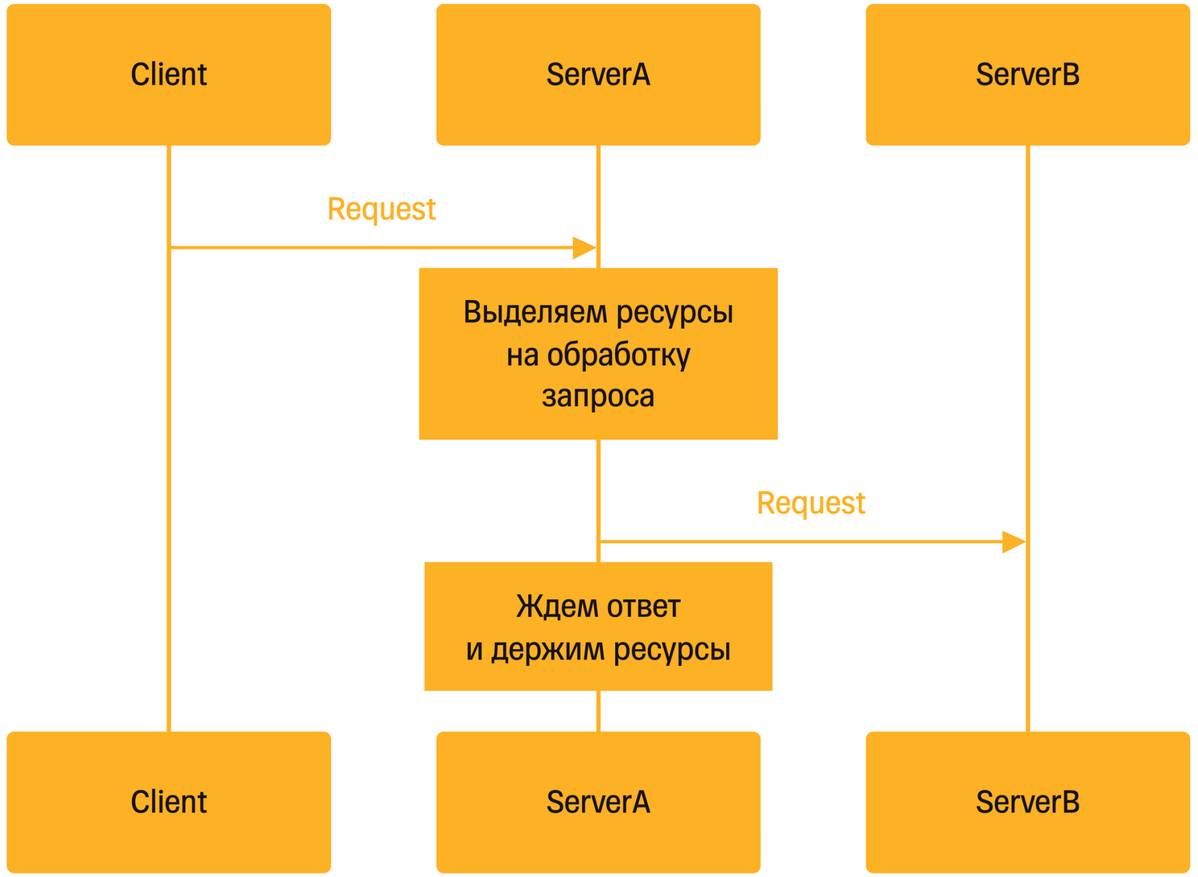
Почему стоит отменять?

- Каждый запрос тратит ресурсы: CPU, RAM, соединения и т.д.
- Ресурсы всегда конечны, пропускная способность системы ограничена



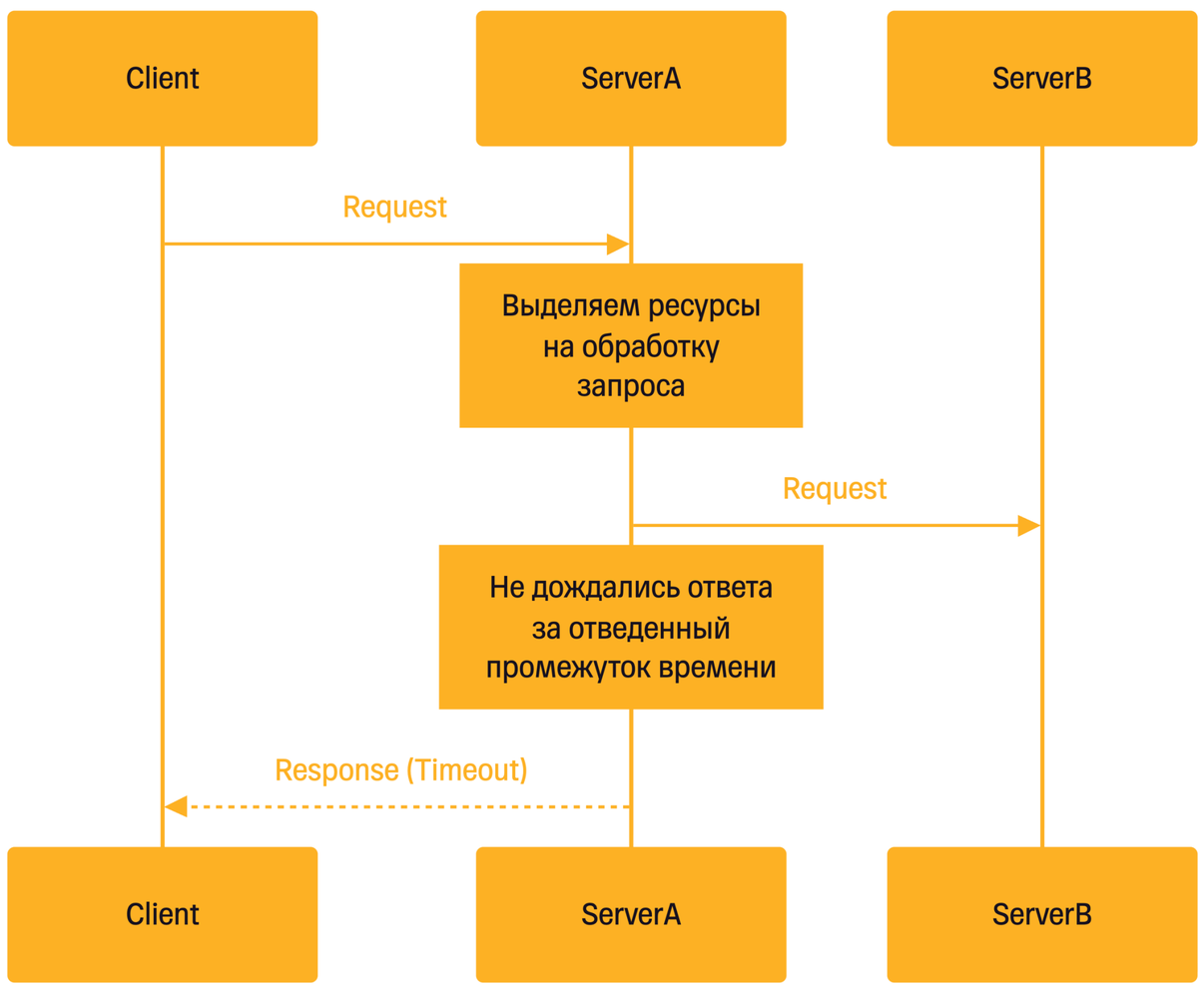
Почему стоит отменять?

- Каждый запрос тратит ресурсы: CPU, RAM, соединения и т.д.
- Ресурсы всегда конечны, пропускная способность системы ограничена
- Если не ограничивать время исполнения, система будет деградировать



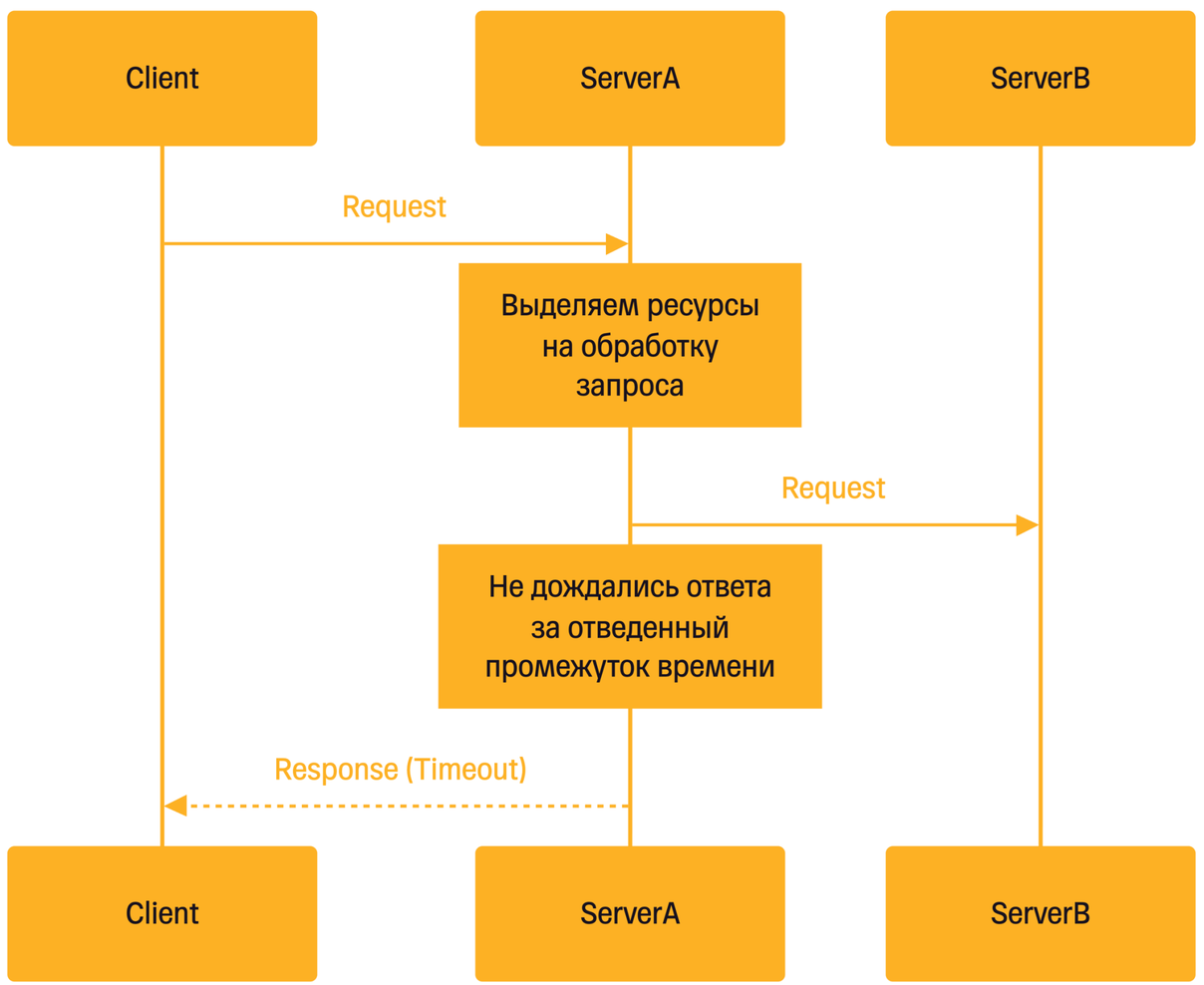
Почему стоит отменять?

- Нужен таймаут на время исполнение запроса
- По таймауту останавливаем обработку запроса и освобождаем ресурсы



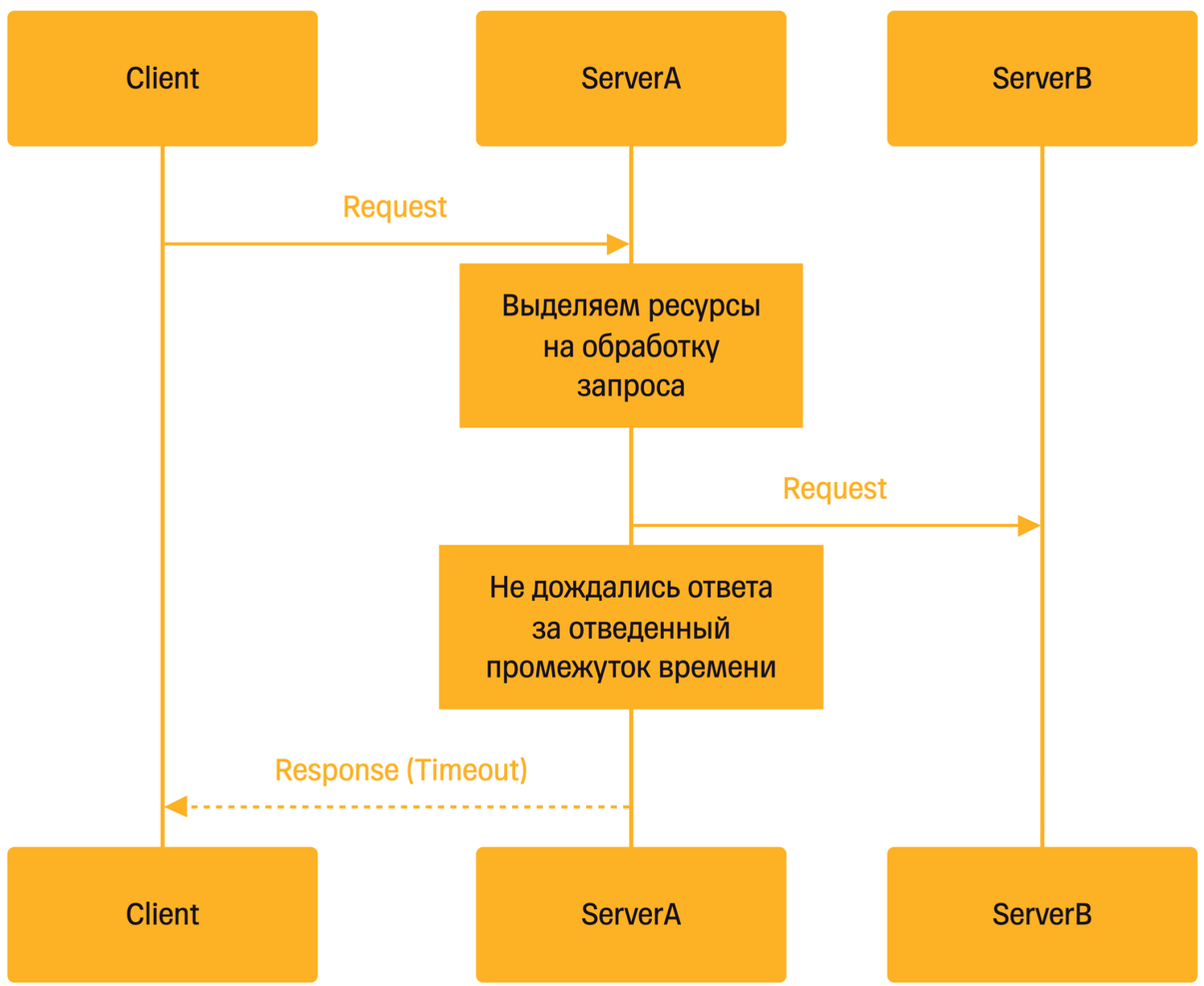
Почему стоит отменять?

- Остановка вычисления может занимать время, отсюда вопрос – когда возвращать ответа? Сразу? Или после завершения?



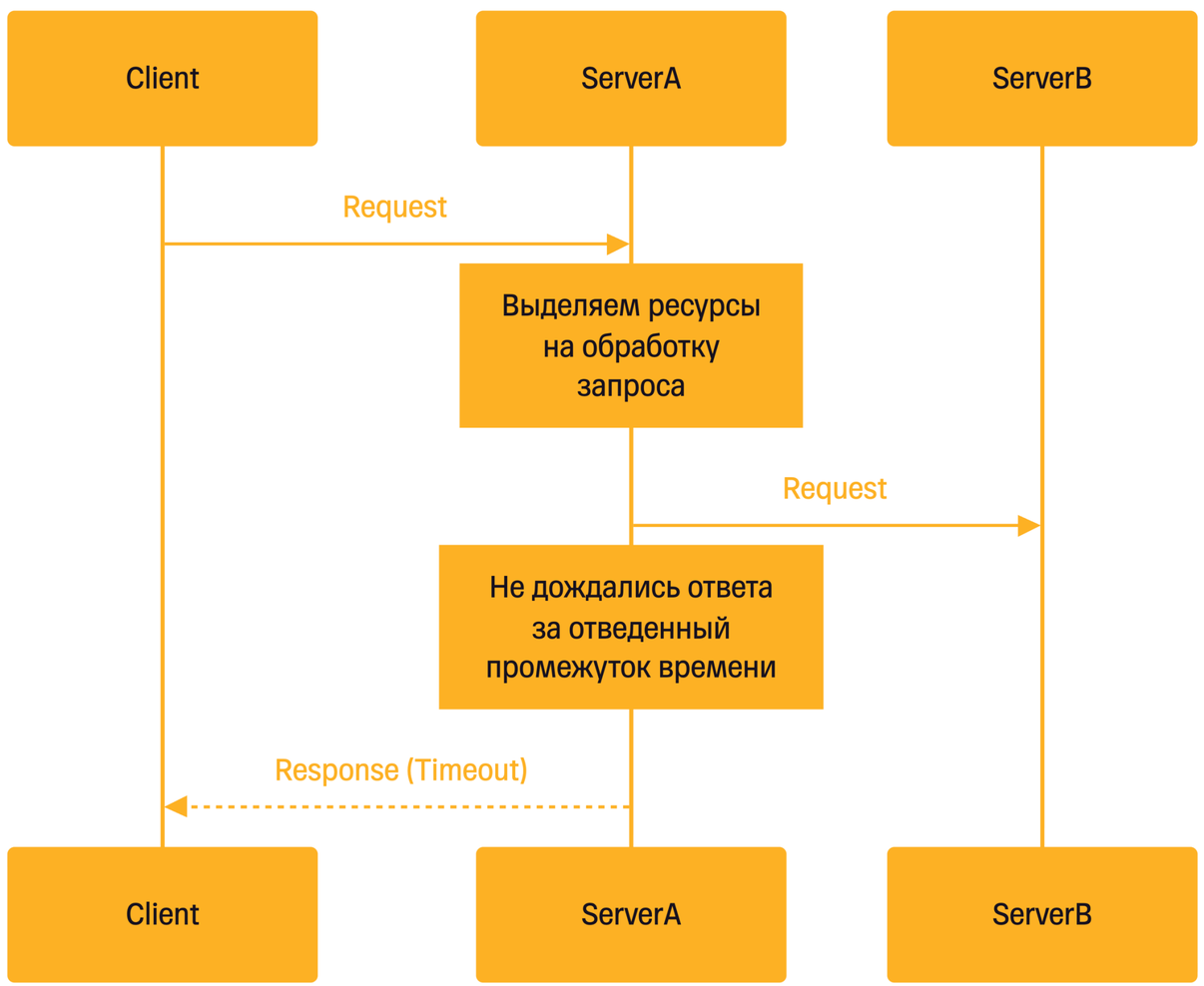
Почему стоит отменять?

- Мое личное предпочтение – лучше дожидаться завершения
- Нет риска утечки ресурсов, всегда все высвобождается
- Проще мониторить, время остановки включается в метрики исполнения запросов



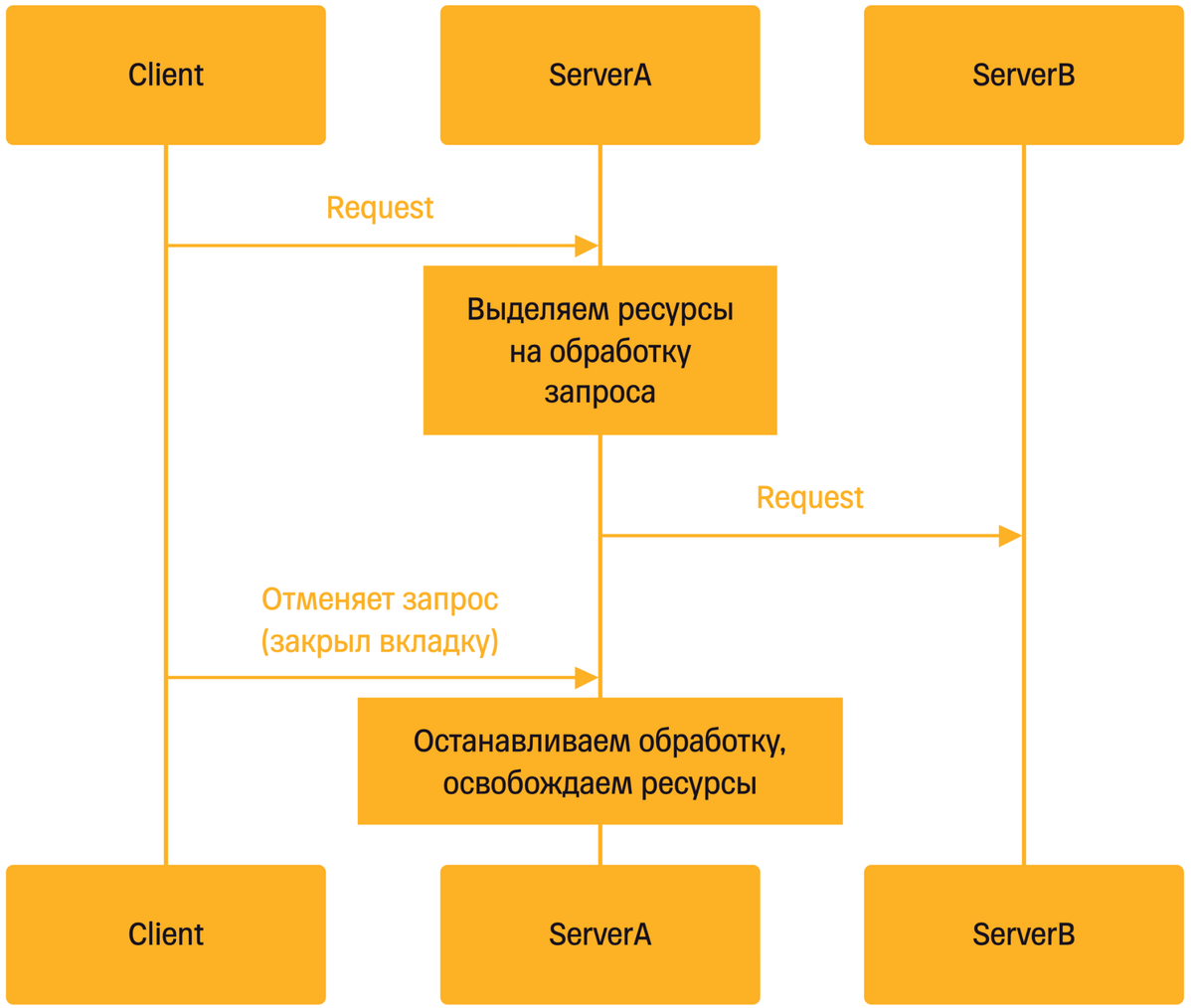
Почему стоит отменять?

- Но, в таком случае нужна гарантия своевременной остановки
- Добиться этого очень сложно



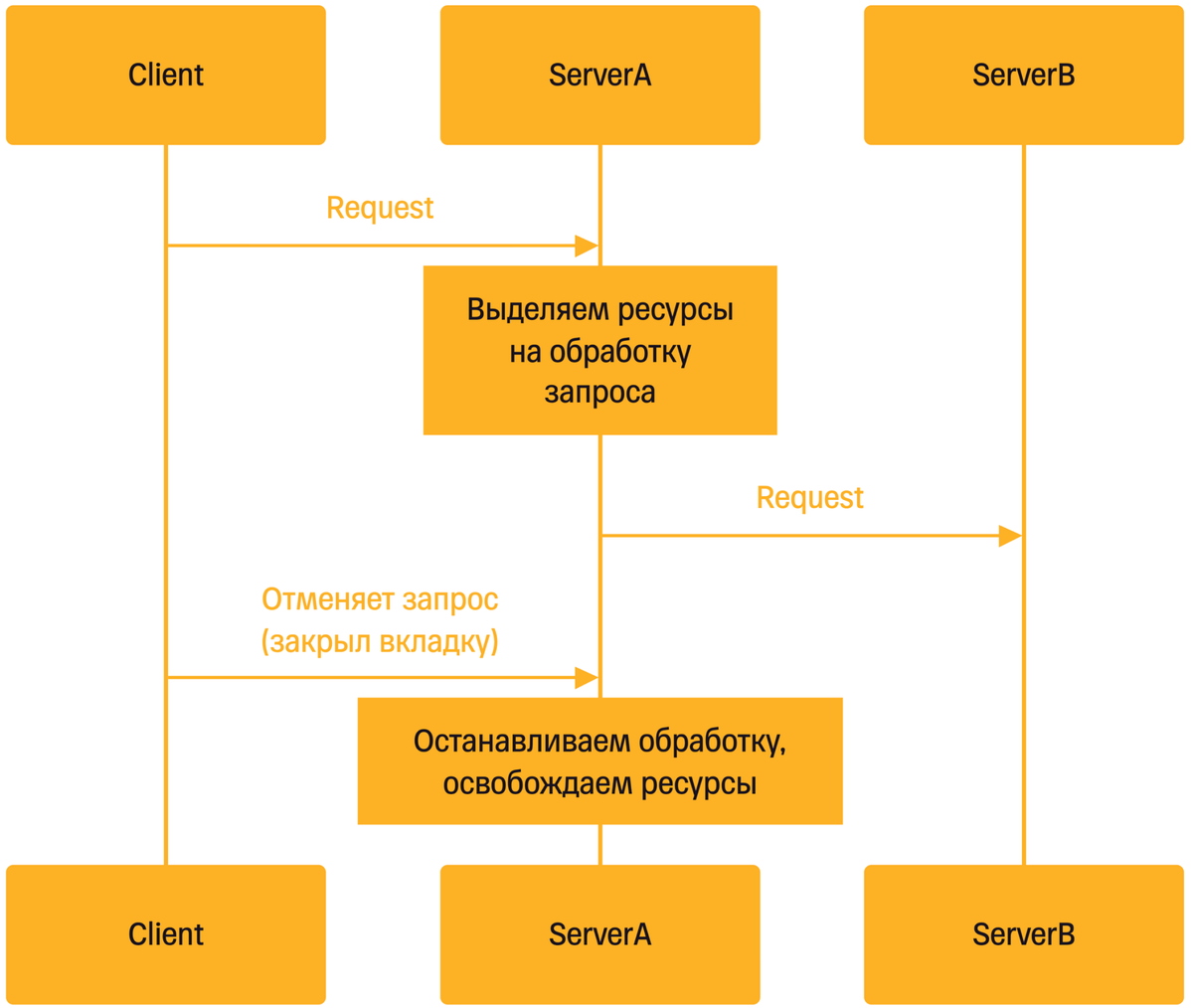
Почему стоит отменять?

- Отдельно стоит рассмотреть сценарий отмены со стороны клиента



Почему стоит отменять?

- Отдельно стоит рассмотреть сценарий отмены со стороны клиента
- В моменты сбоя, клиентам свойственно перезагружать вкладки
- Можно снизить эффект от таких запросов, если уметь останавливать вычисления



Отмена - ключ к стабильной системе

Как должна работать отмена?

- Нужно уметь останавливать обработку запроса по внешнему сигналу (от таймера или при закрытии соединения)
- Обработка запроса не должна игнорировать сигнал об остановке
- Обработка запроса должна своевременно останавливаться

Как остановить вычисление?

Обычный код не остановить

- Возьмем простой цикл, который делает какую-то работу
- Он будет исполняться вечно

Work.scala

```
while (true) {
  println("Do work")
}
```

Сигнал от наблюдателя

- Код исполняется в потоках
- Чтобы остановить, нужен сигнал извне
- Нужно добавить обработку этого сигнала в код
- Сигналом может быть переменная

Work.scala

```
// Вычисление в потоке
var isStopped = false

while (!isStopped) {
  println("Do work")
}

// Передача сигнала из наблюдателя
isStopped = true
```

Java Thread API

- java.lang.Thread так и устроен
- Уже есть специальный флаг – переменная interrupted
- isInterrupted – чтение флага
- interrupt – выставление флага в true

Work.scala

```scala
// Вычисление в потоке
val thread = Thread.ofVirtual().start { _ =>
  while (!Thread.currentThread().isInterrupted) {
    println("Do work")
  }
}
// Передача сигнала из наблюдателя
thread.interrupt()
```

Важное уточнение

- Interruption переводится, как ”прерывание”. То есть выставляя флаг, мы просим поток прервать свое исполнение
- Единой трактовки семантики прерывания - нет
- На практике чаще всего: прерывание = остановка вычисления
- Но это необязательно :)

Специальные методы

- Ряд стандартных блокирующих методов умеет реагировать на флаг прерывания
- Например, Thread.sleep

Work.scala

```scala
// Вычисление в потоке
val thread = Thread.ofVirtual().start { _ =>
  while (true) {
    // Если видит сигнал прерывания,
    // то выбрасывает InterruptedException
    Thread.sleep(1_000)
    println("Do work")
  }
}
// Передача сигнала из наблюдателя
thread.interrupt()
```

Специальные методы

- Ряд стандартных блокирующих методов умеет реагировать на флаг прерывания
- Например, Thread.sleep
- Если флаг выставлен, то метод его видит, очищает флаг и выбрасывает исключение – InterruptedException

- В примере: поток завершит свое исполнение с исключением

Work.scala

```scala
// Вычисление в потоке
val thread = Thread.ofVirtual().start { _ =>
  while (true) {
    // Если видит сигнал прерывания,
    // то выбрасывает InterruptedException
    Thread.sleep(1_000)
    println("Do work")
  }
}
// Передача сигнала из наблюдателя
thread.interrupt()
```

Специальные методы

- Если InterruptedException ловится, то сигнал на прерывание игнорируется и работа продолжается
- Так делать крайне не рекомендуется!

Work.scala

```scala
import scala.util.Try

// Вычисление в потоке
val thread = Thread.ofVirtual().start { _ =>
  while (true) {
    // Ловим любые ошибки и игнорируем их
    // Эквивалентно try-catch
    Try(Thread.sleep(1_000))
    println("Do work")
  }
}
// Передача сигнала из наблюдателя
thread.interrupt()
```

Специальные методы

Work.scala

```scala
import scala.util.{Try, Success, Failure}

def unsafeSleep(millis: Long): Unit = {
  Thread.sleep(millis)
  throw new RuntimeException("Boom!")
}
```

Специальные методы

Work.scala

```scala
import scala.util.{Try, Success, Failure}

def unsafeSleep(millis: Long): Unit = {
  Thread.sleep(millis)
  throw new RuntimeException("Boom!")
}

while (true) {
  Try(unsafeSleep(1_000)) match {
    case Success(_) =>
      // игнорируем успех
    case Failure(_) => ???
  }
  println("Do work")
}
```

Специальные методы

- Если нам необходимо обработать другие исключения, то есть 2 варианта
- Оба варианта продиктованы неявной конвенцией по работе с прерываниями

Work.scala

```scala
import scala.util.{Try, Success, Failure}

def unsafeSleep(millis: Long): Unit = {
  Thread.sleep(millis)
  throw new RuntimeException("Boom!")
}

while (true) {
  Try(unsafeSleep(1_000)) match {
    case Success(_) =>
      // игнорируем успех
    case Failure(_) => ???
  }
  println("Do work")
}
```

Специальные методы

- Вариант №1 – Пробросить исключение дальше
- В примере: останавливаем исполнение с ошибкой

Work.scala

```scala
import scala.util.{Try, Success, Failure}

def unsafeSleep(millis: Long): Unit = {
  Thread.sleep(millis)
  throw new RuntimeException("Boom!")
}

while (true) {
  Try(unsafeSleep(1_000)) match {
    case Success(_) =>
      // игнорируем успех
    case Failure(ex: InterruptedException) =>
      throw ex
    case Failure(ex) =>
      println(s"ignore: $ex")
  }
  println("Do work")
}
```

Специальные методы

- Вариант №2 – Восстановить флаг
- Поднимаем сигнал наверх и даем вызываемой стороне пронаблюдать прерывание

- В примере: поймаем исключение, выставим флаг, сделаем вывод в консоль и успешно завершимся

Work.scala

```scala
import scala.util.{Try, Success, Failure}

def unsafeSleep(millis: Long): Unit = {
  Thread.sleep(millis)
  throw new RuntimeException("Boom!")
}

while (!Thread.currentThread().isInterrupted) {
  Try(unsafeSleep(1_000)) match {
    case Success(_) =>
      // игнорируем успех
    case Failure(ex: InterruptedException) =>
      // Восстанвливаем флаг прерывания
      Thread.currentThread().interrupt()
    case Failure(ex) =>
      println(s"ignore: $ex")
  }
  println("Do work")
}
```

Почему сигнал о прерывании - ошибка?

- Нужен механизм, который бы мог “вырваться” из блокирующего кода
- Ошибка – единственный возможный вариант остановить вычисление
- Если есть работа с ресурсами, то ошибка позволит их корректно закрыть
- Самый худший сценарий в такой схеме – игнорирование прерывания

Сложности

Work.scala

```scala
case class Order(price: BigDecimal)

// Считаем цену заказа через удаленный сервис
def remoteCalculatePrice(): BigDecimal
// Локальный расчет цены заказа
def fallbackCalculatePrice(): BigDecimal
// Сохранение в БД
def saveOrder(order: Order): Unit
```

Сложности

Work.scala

```scala
case class Order(price: BigDecimal)

// Считаем цену заказа через удаленный сервис
def remoteCalculatePrice(): BigDecimal
// Локальный расчет цены заказа
def fallbackCalculatePrice(): BigDecimal
// Сохранение в БД
def saveOrder(order: Order): Unit

def processOrder(): Unit = {
  // Если вызов удаленного сервиса не удался,
  // то используем локальный расчет
  val price =
    Try(remoteCalculatePrice())
      .getOrElse(fallbackCalculatePrice())

  saveOrder(Order(price))
}
```

Сложности

- Сложность №1: Нужно очень аккуратно обрабатывать все ошибки в коде
- В примере: если прерывание произошло во время похода в удаленный сервис, то сигнал игнорируется

Work.scala

```scala
case class Order(price: BigDecimal)

// Считаем цену заказа через удаленный сервис
def remoteCalculatePrice(): BigDecimal
// Локальный расчет цены заказа
def fallbackCalculatePrice(): BigDecimal
// Сохранение в БД
def saveOrder(order: Order): Unit

def processOrder(): Unit = {
  // Если вызов удаленного сервиса не удался,
  // то используем локальный расчет
  val price =
    Try(remoteCalculatePrice())
      .getOrElse(fallbackCalculatePrice())

  saveOrder(Order(price))
}
```

Сложности

- Сложность №1: Нужно очень аккуратно обрабатывать все ошибки в коде
- Чтобы исправить, надо пробросить исключение InterruptedException
- Неявно создаем отдельную траекторию по работе с прерываниями

Work.scala

```
def processOrder(): Unit = {
  // Если вызов удаленного сервиса не удался,
  // то используем локальный расчет
  val price =
    Try(remoteCalculatePrice()) match {
      case Success(price) =>
        price
      case Failure(ex: InterruptedException) =>
        throw ex
      case Failure(_) =>
        fallbackCalculatePrice()
    }

  saveOrder(Order(price))
}
```

Сложности

- Сложность №2: Явно добавлять обработку сигналов о прерывании, чтобы сделать код более отзывчивым
- Если есть длинные вычисления/циклы, стоит добавить обработку сигналов прерывания

Work.scala

```
while (!Thread.currentThread().isInterrupted) {
  // Очень долгие вычисления
}
```

Сложности

- Сложность №3: Использовать правильные методы библиотек
- Всегда, где есть блокировка, нужно смотреть, как вызываемый код реагирует на прерывания
- Стоит вызывать методы, которые могут бросить InterruptedException

Work.scala

```scala
import java.util.concurrent.locks.ReentrantLock

val lock = new ReentrantLock()

// Плохо
lock.lock()

// Хорошо
lock.lockInterruptibly()
```

А что в Scala?

Системы эффектов

- Описываем вычисления через системы эффектов
- Примеры используют библиотеку – cats-effect
- Вычисления композируются через flatMap

CE3.scala

```scala
import cats.effect.IO

val program: IO[Unit] =
  IO.println("Step #1")
    .flatMap(_ => IO.println("Step #2"))
    .flatMap(_ => IO.println("Step #3"))
```

Системы эффектов

- Описываем вычисления через системы эффектов
- Примеры используют библиотеку – cats-effect
- Вычисления композируются через flatMap
- Вычисления ленивые, нужно явно запускать

CE3.scala

```scala
import cats.effect.IO

val program: IO[Unit] =
  IO.println("Step #1")
    .flatMap(_ => IO.println("Step #2"))
    .flatMap(_ => IO.println("Step #3"))

// Ничего не происходит
// Нужно явно запустить
program.unsafeRunSync()
// std: Step #1
// std: Step #2
// std: Step #3
```

Императивный код

Ver1.scala × Ver2.scala

```scala
val program: IO[Unit] =
  IO.println("Step #1")
    .flatMap(_ => IO.println("Step #2"))
    .flatMap(_ => IO.println("Step #3"))
```


Ver1.scala Ver2.scala ×

```scala
val program: IO[Unit] =
  for {
    _ <- IO.println("Step #1")
    _ <- IO.println("Step #2")
    _ <- IO.println("Step #3")
  } yield ()

// Читается как

val classicProgram = {
  println("Step #1")
  println("Step #2")
  println("Step #3")
}
```

Файберы

- Вычисления можно запускать в легковесных потоках – файберах
- Можно его отменить
- Можно дождаться результата вычисления в файбере

- Очень похоже на Thread API

CE3.scala

```scala
val computation: IO[Unit]

val program: IO[Unit] =
  for {
    fiber <- computation.start
    // Вычисление computation исполняется
    _     <- fiber.cancel
    // Вычисление computation точно
    // остановлено
    _     <- fiber.join
  } yield ()
```

Файберы

- Вычисления можно запускать в легковесных потоках – файберах
- Можно его отменить
- Можно дождаться результата вычисления в файбере

- Очень похоже на Thread API
- Главное отличие – исполнение cancel гарантирует остановку

CE3.scala

```scala
val computation: IO[Unit]

val program: IO[Unit] =
  for {
    fiber <- computation.start
    // Вычисление computation исполняется
    _     <- fiber.cancel
    // Вычисление computation точно
    // остановлено
    _     <- fiber.join
  } yield ()
```

Файберы

- Исполнение файбера может завершится 3-емя исходами: успех, ошибка, отмена

CE3.scala

```scala
enum Outcome[+A] {
  case Success[A](value: A)
    extends Outcome[A]
  case Failure(exception: Throwable)
    extends Outcome[Nothing]
  case Cancelled
    extends Outcome[Nothing]
}

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

Файберы

- Исполнение файбера может завершится 3-емя исходами: успех, ошибка, отмена
- Канал отмены – отдельный
- В случае вызова cancel, последующий join вернет Cancelled

CE3.scala

```scala
enum Outcome[+A] {
  case Success[A](value: A)
    extends Outcome[A]
  case Failure(exception: Throwable)
    extends Outcome[Nothing]
  case Cancelled
    extends Outcome[Nothing]
}

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

Файберы

- Исполнение файбера может завершится 3-емя исходами: успех, ошибка, отмена
- Канал отмены – отдельный
- В случае вызова cancel, последующий join вернет Cancelled
- Нельзя отменить отмену, как в случае с ошибкой

CE3.scala

```scala
enum Outcome[+A] {
  case Success[A](value: A)
    extends Outcome[A]
  case Failure(exception: Throwable)
    extends Outcome[Nothing]
  case Cancelled
    extends Outcome[Nothing]
}

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

Пример отмены

- Сделаем бесконечное вычисление, которое будет в цикле выводить что-то в консоль
- Запустим его в отдельном файбере и спустя время отменим его

CE3.scala

```scala
def loop(num: Int): IO[Unit] =
  IO.println(s"Итерация #$num")
    .flatMap(_ => loop(num + 1))

val program: IO[Unit] =
  for {
    loopFiber <- loop(1).start
    _         <- IO.sleep(30.seconds)
    _         <- loopFiber.cancel
  } yield ()
```

Пример отмены

- Сделаем бесконечное вычисление, которое будет в цикле выводить что-то в консоль
- Запустим его в отдельном файбере и спустя время отменим его
- Если проводить аналогию с while(true), то мы бы ожидали, что вычисление никогда не завершится

CE3.scala

```scala
def loop(num: Int): IO[Unit] =
  IO.println(s"Итерация #$num")
    .flatMap(_ => loop(num + 1))

val program: IO[Unit] =
  for {
    loopFiber <- loop(1).start
    _         <- IO.sleep(30.seconds)
    _         <- loopFiber.cancel
  } yield ()
```

Пример отмены

- Сделаем бесконечное вычисление, которое будет в цикле выводить что-то в консоль
- Запустим его в отдельном файбере и спустя время отменим его
- Если проводить аналогию с while(true), то мы бы ожидали, что вычисление никогда не завершится
- Но это не так, в примере вычисление успешно отменится

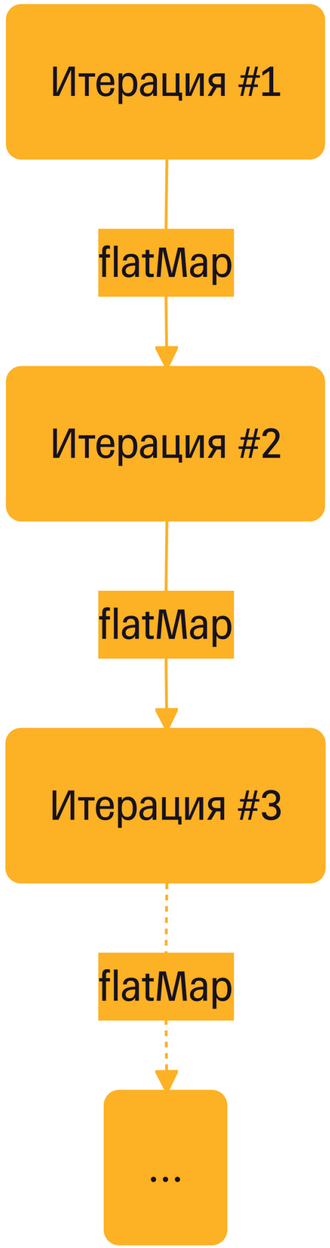
CE3.scala

```scala
def loop(num: Int): IO[Unit] =
  IO.println(s"Итерация #$num")
    .flatMap(_ => loop(num + 1))

val program: IO[Unit] =
  for {
    loopFiber <- loop(1).start
    _         <- IO.sleep(30.seconds)
    _         <- loopFiber.cancel
  } yield ()
```

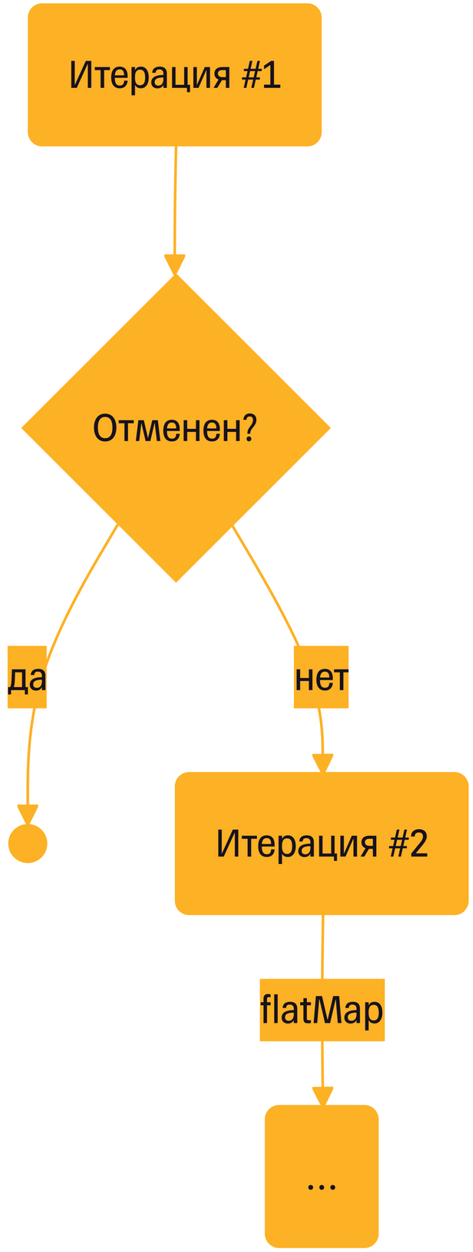
Почему так?

- Каждое вычисление можно представить, как граф
- Каждая вершина – отдельный шаг вычисления
- Ребра формируются из вызовов flatMap



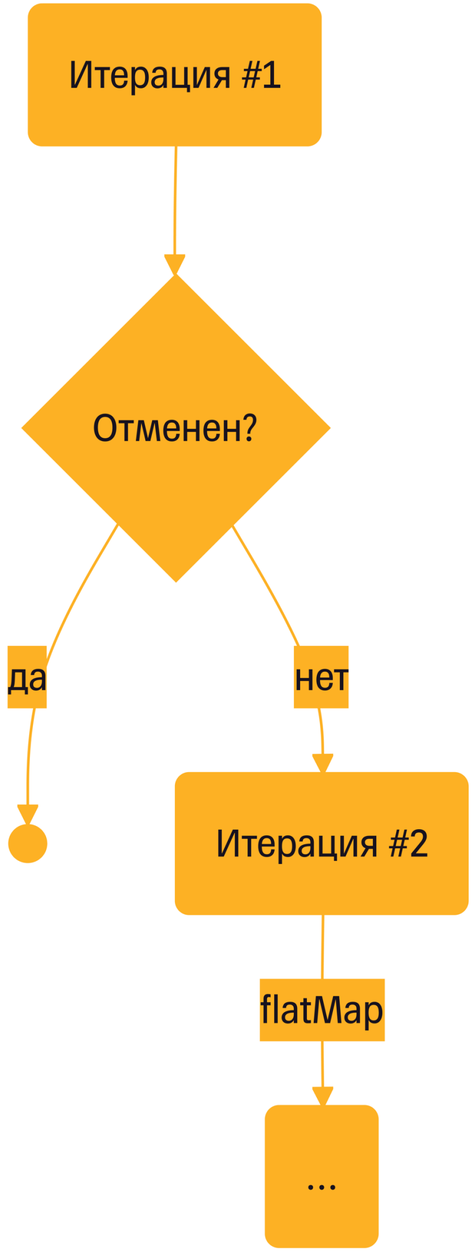
Почему так?

- В каждом flatMap неявно есть проверка на отмену
- Если отмена произошла, то вычисление останавливается
- Иначе говоря, все вычисления отменяемы по умолчанию



Почему так?

- Логичный вопрос дальше: как быть с ресурсами?
- Нельзя же просто взять и остановить вычисление?
- Их же нужно высвободить в случае отмены?



Управление ресурсами

CE3.scala

```scala
def loop(num: Int, r: String): IO[Unit] =
  IO.println(s"Итерация #$num (Ресурс: $r)")
    .flatMap(_ => loop(num + 1, r))
```

Управление ресурсами

- С помощью специальных операторов bracket/guarantee можно объявить взятие и освобождение ресурса
- Очень похоже на try-with-resources

CE3.scala

```scala
def loop(num: Int, r: String): IO[Unit] =
  IO.println(s"Итерация #$num (Ресурс: $r)")
    .flatMap(_ => loop(num + 1, r))

val loopWithResource: IO[Unit] =
  IO.println("Выделяем ресурс")
    .as("resource")
    .bracket { r => loop(1, r) } { r =>
      IO.println(s"Освобождаем ресурс: $r")
    }
```

Управление ресурсами

- С помощью специальных операторов bracket/guarantee можно объявить взятие и освобождение ресурса
- Очень похоже на try-with-resources

CE3.scala

```scala
def loop(num: Int, r: String): IO[Unit] =
  IO.println(s"Итерация #$num (Ресурс: $r)")
    .flatMap(_ => loop(num + 1, r))

val loopWithResource: IO[Unit] =
  IO.println("Выделяем ресурс")
    .as("resource")
    .bracket { r => loop(1, r) } { r =>
      IO.println(s"Освобождаем ресурс: $r")
    }
```

Управление ресурсами

- С помощью специальных операторов bracket/guarantee можно объявить взятие и освобождение ресурса
- Очень похоже на try-with-resources

CE3.scala

```scala
def loop(num: Int, r: String): IO[Unit] =
  IO.println(s"Итерация #$num (Ресурс: $r)")
    .flatMap(_ => loop(num + 1, r))

val loopWithResource: IO[Unit] =
  IO.println("Выделяем ресурс")
    .as("resource")
    .bracket { r => loop(1, r) } { r =>
      IO.println(s"Освобождаем ресурс: $r")
    }
```

Управление ресурсами

- С помощью специальных операторов bracket/guarantee можно объявить взятие и освобождение ресурса
- Очень похоже на try-with-resources

CE3.scala

```scala
def loop(num: Int, r: String): IO[Unit] =
  IO.println(s"Итерация #$num (Ресурс: $r)")
    .flatMap(_ => loop(num + 1, r))

val loopWithResource: IO[Unit] =
  IO.println("Выделяем ресурс")
    .as("resource")
    .bracket { r => loop(1, r) } { r =>
      IO.println(s"Освобождаем ресурс: $r")
    }
```

Управление ресурсами

- С помощью специальных операторов bracket/guarantee можно объявить взятие и освобождение ресурса
- Очень похоже на try-with-resources
- В случае любого завершения файбера, ресурс будет освобожден
- Ресурсов или гарантированных действий может быть сколько угодно

CE3.scala

```scala
def loop(num: Int, r: String): IO[Unit] =
  IO.println(s"Итерация #$num (Ресурс: $r)")
    .flatMap(_ => loop(num + 1, r))

val loopWithResource: IO[Unit] =
  IO.println("Выделяем ресурс")
    .as("resource")
    .bracket { r => loop(1, r) } { r =>
      IO.println(s"Освобождаем ресурс: $r")
    }

val program: IO[Unit] =
  for {
    loopFiber <- loopWithResource.start
    _         <- IO.sleep(30.seconds)
    _         <- loopFiber.cancel
    _         <- IO.println("Файбер остановлен")
  } yield ()
```

Управление ресурсами

- Исполнение cancel гарантирует, что отменяемый файбер остановлен, включая все освобождения ресурсов

CE3.scala

```scala
def loop(num: Int, r: String): IO[Unit] =
  IO.println(s"Итерация #$num (Ресурс: $r)")
    .flatMap(_ => loop(num + 1, r))

val loopWithResource: IO[Unit] =
  IO.println("Выделяем ресурс")
    .as("resource")
    .bracket { r => loop(1, r) } { r =>
      IO.println(s"Освобождаем ресурс: $r")
    }

val program: IO[Unit] =
  for {
    loopFiber <- loopWithResource.start
    _         <- IO.sleep(30.seconds)
    _         <- loopFiber.cancel
    _         <- IO.println("Файбер остановлен")
  } yield ()
```

Управление ресурсами

- Исполнение cancel гарантирует, что отменяемый файбер остановлен, включая все освобождения ресурсов
- Иначе говоря, есть гарантия на то, что “Файбер остановлен” произойдет строго после “Освобождаем ресурс”

CE3.scala

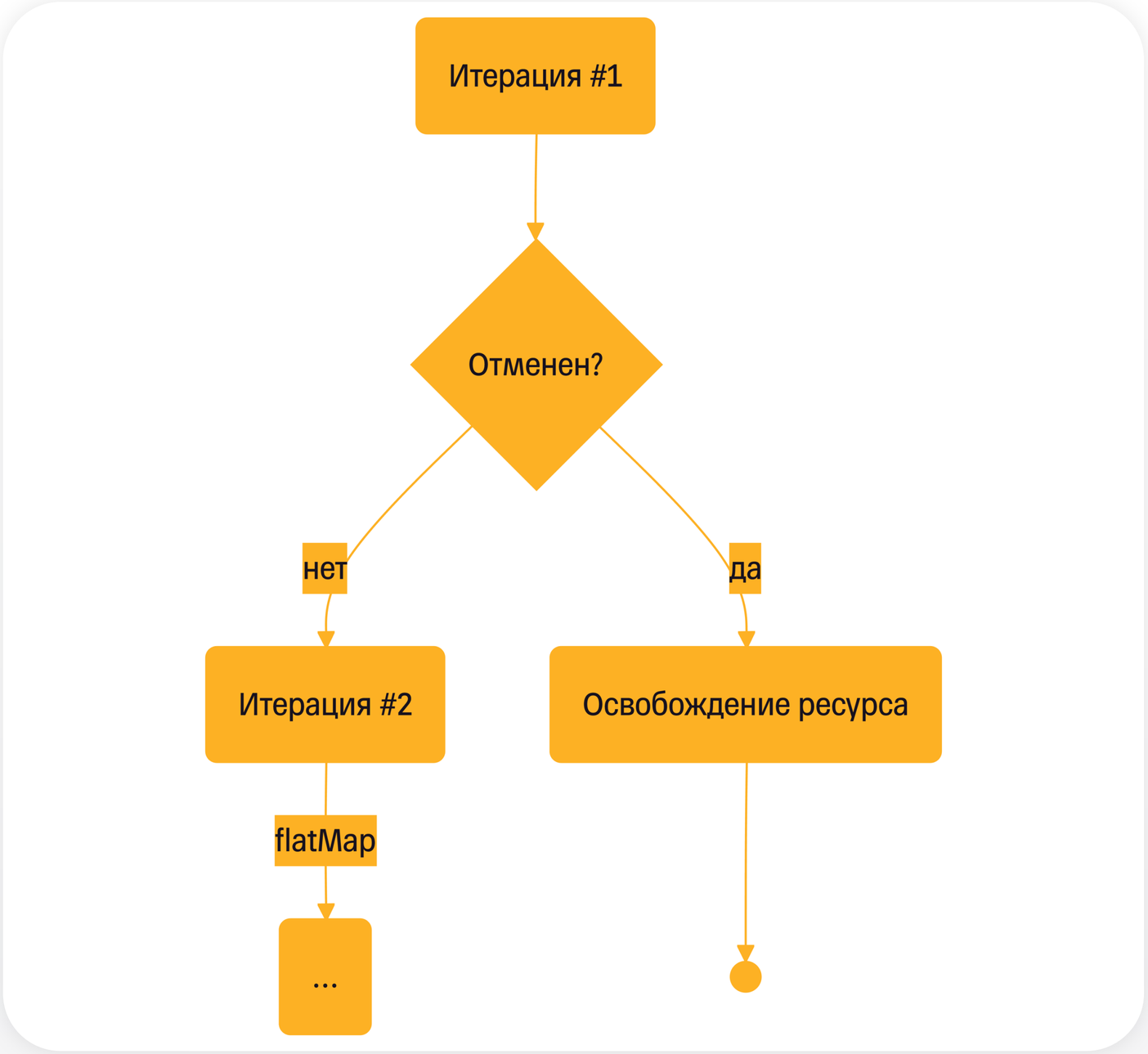
```scala
def loop(num: Int, r: String): IO[Unit] =
  IO.println(s"Итерация #$num (Ресурс: $r)")
    .flatMap(_ => loop(num + 1, r))

val loopWithResource: IO[Unit] =
  IO.println("Выделяем ресурс")
    .as("resource")
    .bracket { r => loop(1, r) } { r =>
      IO.println(s"Освобождаем ресурс: $r")
    }

val program: IO[Unit] =
  for {
    loopFiber <- loopWithResource.start
    _         <- IO.sleep(30.seconds)
    _         <- loopFiber.cancel
    _         <- IO.println("Файбер остановлен")
  } yield ()
```

Управление ресурсами

- В случае отмены, помимо остановки вычисления, будут выполнены все “гарантированные” действия



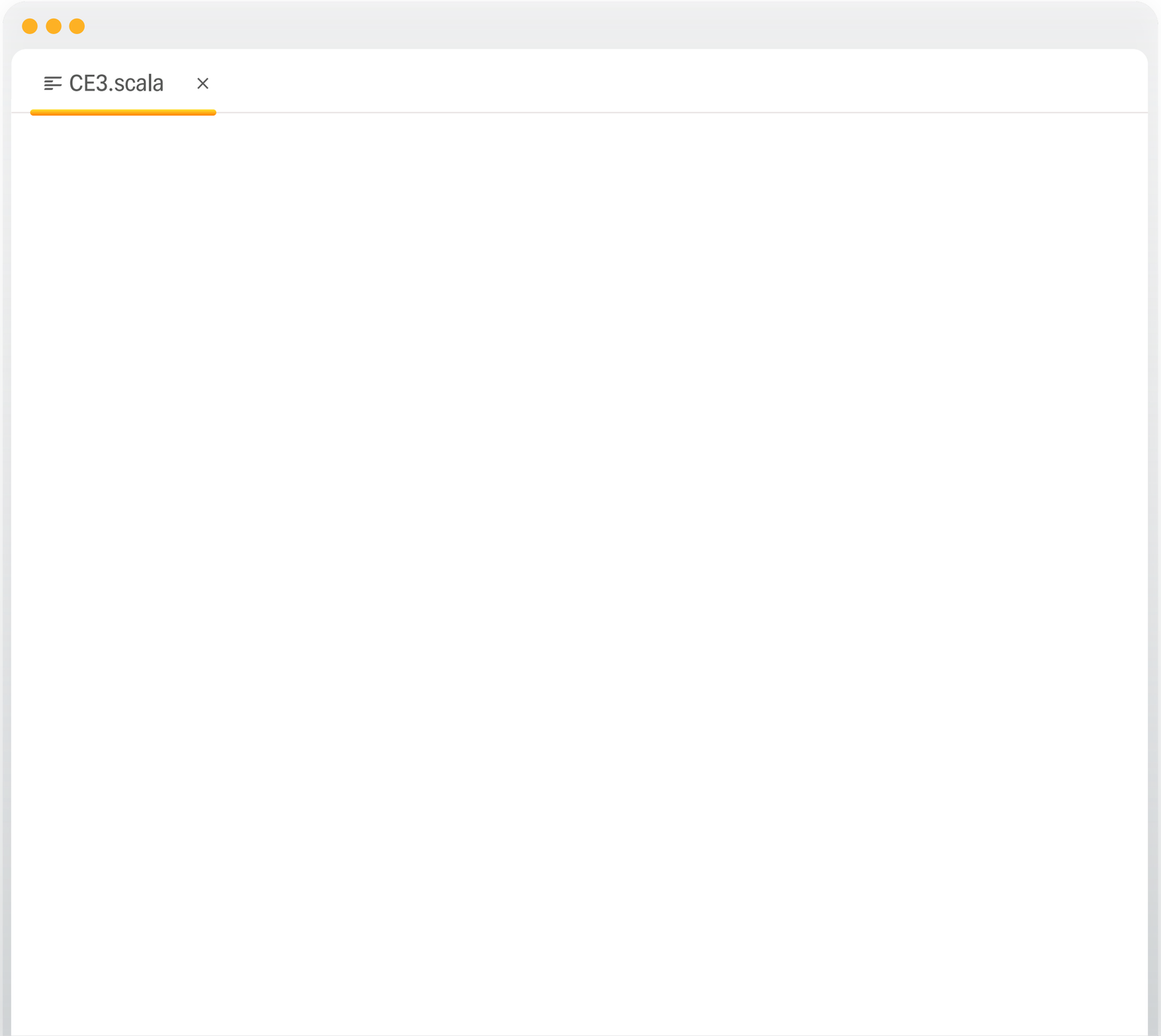
Что еще можно делать?

Кратко пробежимся по тому, что еще можно делать

- Можно определять ”гарантированные” действия в зависимости от результата вычисления (успех, ошибка, отмена)
- Можно размечать ”неотменяемые” регионы, чтобы гарантировать транзакционность изменений
- Можно исполнять блокирующие Java операции на отдельном пуле потоков
- В случае отмены блокирующей операции, можно провоцировать вызов interrupt

Сложности

- Единственная сложность: время отмены в общем случае недетерминировано



Сложности

- Единственная сложность: время отмены в общем случае недетерминировано
- Само вычисление может быть блокирующим

CE3.scala

```scala
val infinite = IO.blocking {
  while (true) {
    println("Do Work")
  }
}

for {
  f <- infinite.start
  _ <- f.cancel
  _ <- IO.println("Никогда не будет вызван")
} yield ()
```

Сложности

- Единственная сложность: время отмены в общем случае недетерминировано
- Само вычисление может быть блокирующим
- Время исполнения “гарантированных” действий может быть долгим

CE3.scala

```scala
val longClose =
  IO.println("Do Work")
    .guarantee(IO.sleep(10.seconds))

for {
  f <- longClose.start
  _ <- f.cancel
  _ <- IO.println("Будет вызван через 10 секунд")
} yield ()
```

Сложности

- Единственная сложность: время отмены в общем случае недетерминировано
- Само вычисление может быть блокирующим
- Время исполнения “гарантированных” действий может быть долгим
- На практике редко можно с таким столкнуться

CE3.scala

```scala
val longClose =
  IO.println("Do Work")
    .guarantee(IO.sleep(10.seconds))

for {
  f <- longClose.start
  _ <- f.cancel
  _ <- IO.println("Будет вызван через 10 секунд")
} yield ()
```

Сравнение

Java Thread API

Отмена явная

По умолчанию нужно явно добавлять реакцию на отмену

VS

Системы эффектов

Отмена неявная

По умолчанию весь код умеет реагировать на отмену

Java Thread API

Отмена асинхронная

Вызов interrupt выставляет флаг и не дожидается отмены

VS

Системы эффектов

Отмена синхронная

Вызов cancel отменяет вычисление и дожидается полной остановки

Java Thread API

VS

Системы эффектов

Сигнал - Ошибка

Отмена происходит через выбрасывание исключений, нужно аккуратно обрабатывать ошибки

Сигнал – Отдельный канал

Отмена происходит через отдельный канал, который невозможно спутать с ошибкой

Отмены в системах эффектов проще и удобнее

Главная проблема остается

Отмена кооперативная и не моментальная

Главная проблема остается

Отмена кооперативная и не моментальная

- Весь исполняемый код должен реагировать на сигнал отмены
- Все действия по закрытию ресурсов должны быть быстрыми

Главная проблема остается

Отмена кооперативная и не моментальная

- Весь исполняемый код должен реагировать на сигнал отмены
- Все действия по закрытию ресурсов должны быть быстрыми
- Нужно глубоко понимать все, что происходит в вызываемых библиотеках (HttpClient, DB и т.д.)
- И как эти библиотеки реагируют на сигнал отмены. Если вообще реагируют

«Одна ошибка и отмена не работает»

Джейсон Стейтем

Рецепт успеха

- Использовать отмены просто необходимо при разработке стабильных систем. Но нужно учитывать все подводные камни
- Пишите интеграционные тесты на таймауты и разрывы соединений, чтобы избежать ошибок в проде
- При обращении в сторонние системы в тестах симулируйте “отсутствие” ответа
- Кратно повысите отказоустойчивость своего сервиса

Спасибо за внимание!

Telegram

@FireFoxIL