IT's TINKOFF

Project Loom – серебряная пуля?

Или все же нет?

Иван Лягаев

17.08.2023

Скачать презентацию

Иван Лягаев

Ведущий Scala разработчик,

Тинькофф.Бизнес

i.lyagaev@tinkoff.ru

Telegram: @FireFoxIL

Github: @FireFoxIL

«Project Loom – убийца реактивного стека и полноценная замена корутинам Kotlin»

Распространенное мнение

~~«Project Loom – убийца реактивного стека и полноценная замена корутинам Kotlin»~~

«Пора ли менять профессию?»

Распространенное мнение

Содержание

Зачем нужен Project Loom и его аналоги из других языков?

Что из себя представляет Project Loom? Что он дает?

Как схожие проблемы решает экосистема Scala?

Итоги

Проблематика

- Рассмотрим, как выглядит типичное backend приложение
- Моделируем обработку запроса, как функцию обработчик с последовательностью действий

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

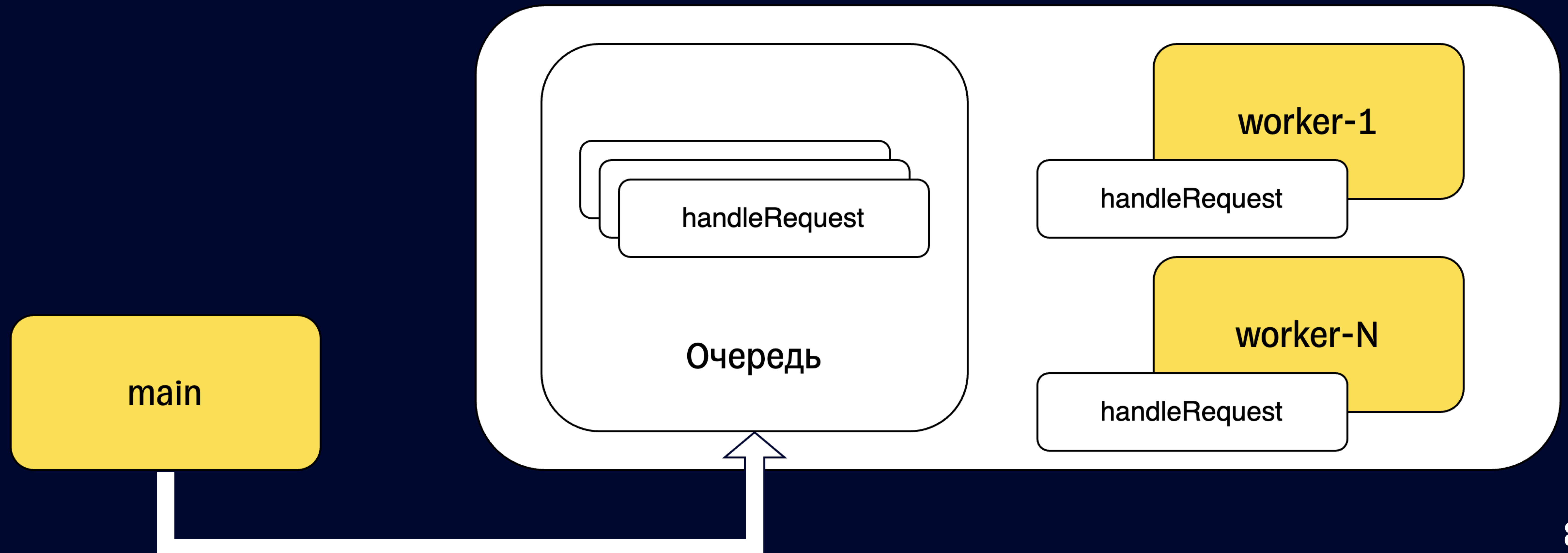
Проблематика

- На уровне конкретного фреймворка происходит следующее
- Обработка запроса в отдельном Runnable, все запросы попадают в очередь пула потоков

```
final val ParallelLimit = 10
val executor = Executors.newFixedThreadPool(ParallelLimit)
while (true) {
  val req = acceptRequest()
  executor.submit { () =>
    val response = handleRequest(req)
    serveResponse(response)
  }
}
```

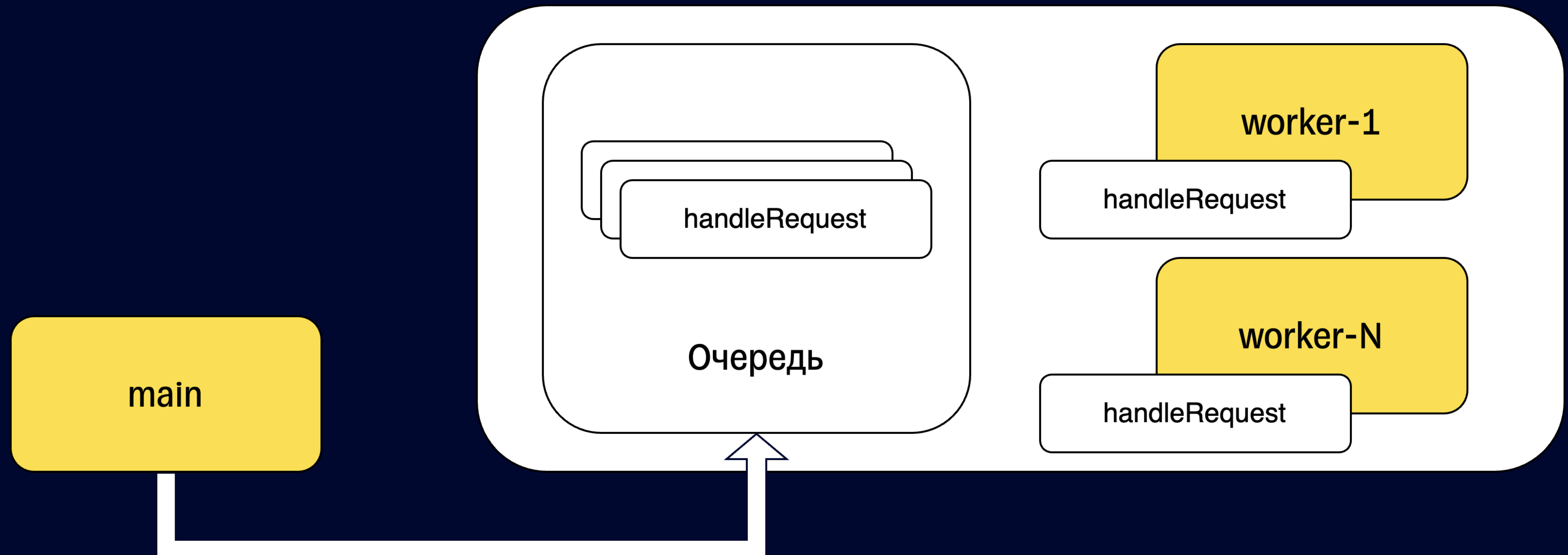
Проблематика

- На уровне конкретного фреймворка происходит следующее
- Обработка запроса в отдельном Runnable, все запросы попадают в очередь пула потоков



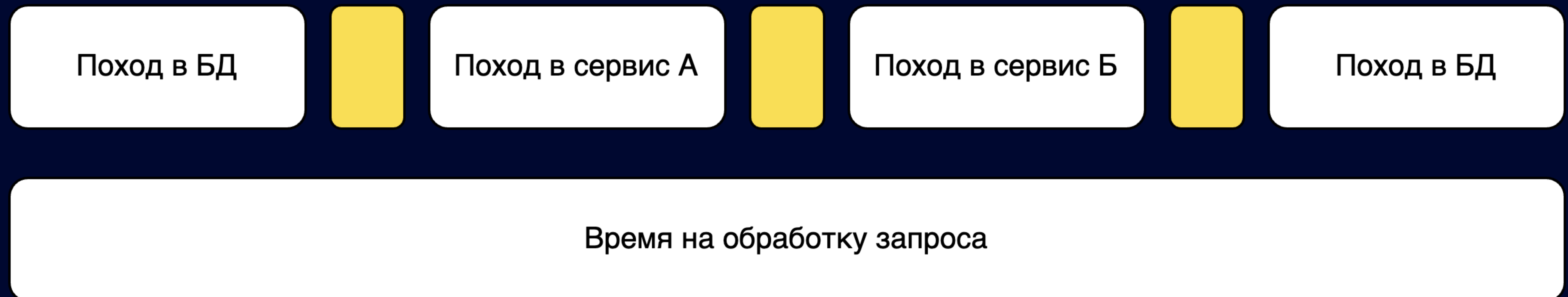
Проблема 10К соединений

- Одновременная обработка запросов ограничена N
- Чем больше N, тем больше переключений контекста
- Нельзя переступить порог одновременной обработки 10к соединений



Ограниченность I/O

- Типичное backend приложение ходит в другие внешние системы по сети (I/O)
- Время обработки запроса ~= потраченному времени на I/O

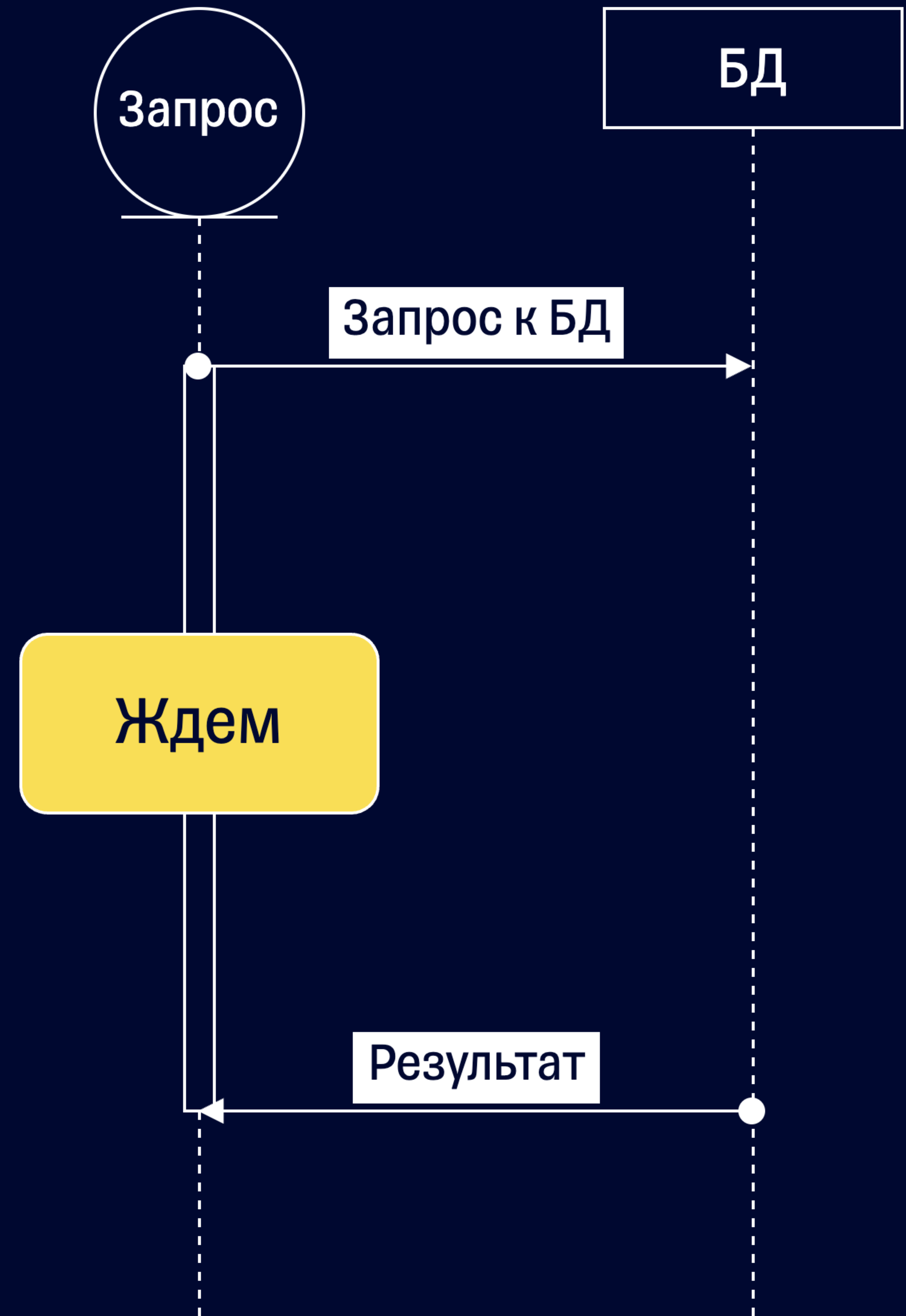


Синхронное API

- При взаимодействии с внешней системой (например, БД) принято использовать синхронное API

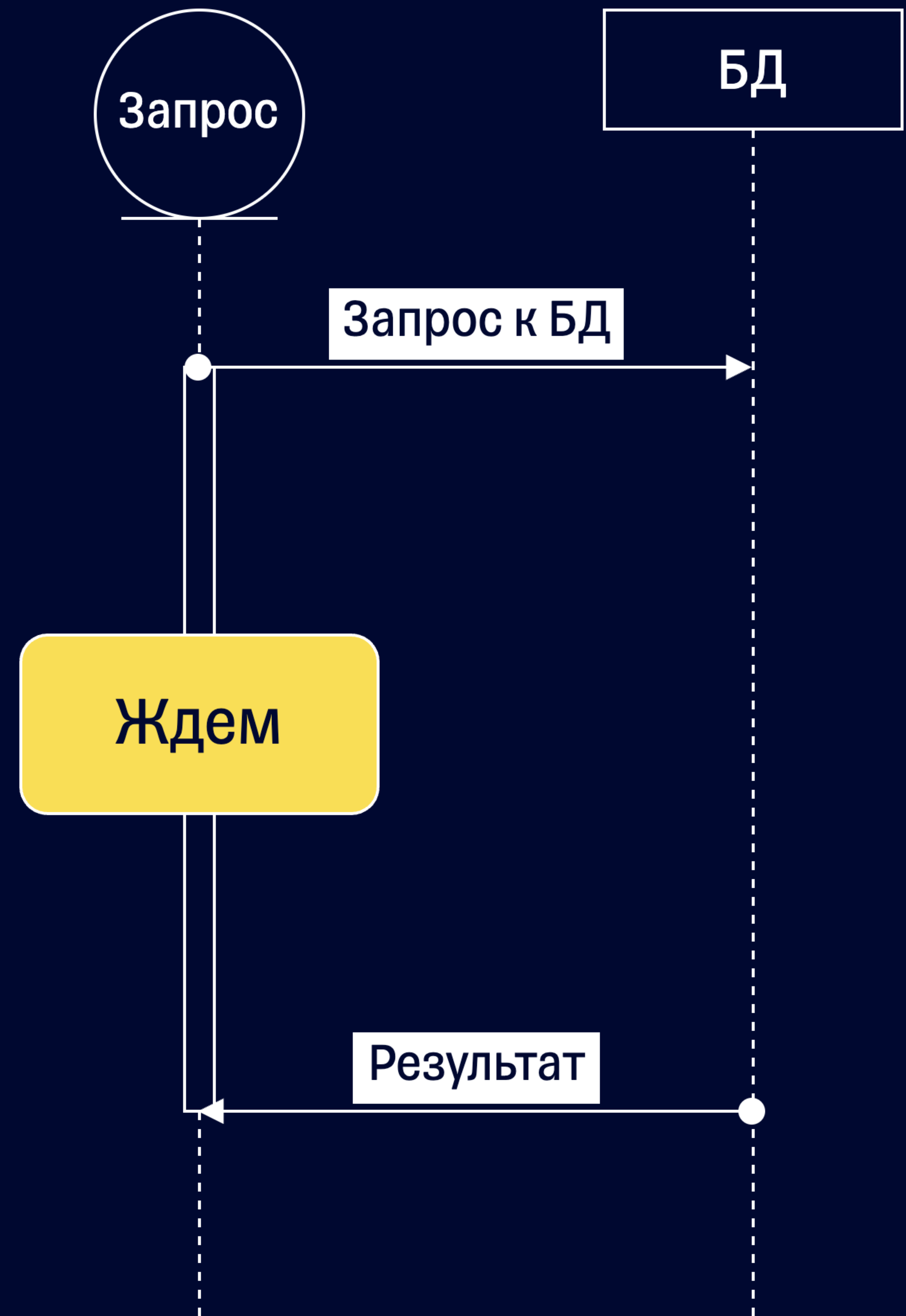
Синхронное API

- При взаимодействии с внешней системой (например, БД) принято использовать синхронное API
- При вызове внешней системы поток для обработки запроса просто ждет результата (блокируется)

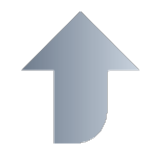


Синхронное API

- При взаимодействии с внешней системой (например, БД) принято использовать синхронное API
- При вызове внешней системы поток для обработки запроса просто ждет результата (блокируется)
- Поток не утилизирует выданные ресурсы
- Поток мог бы дать другим потокам время на исполнение



Что предлагает Scala?



Новые абстракции

Новая абстракция – файбер

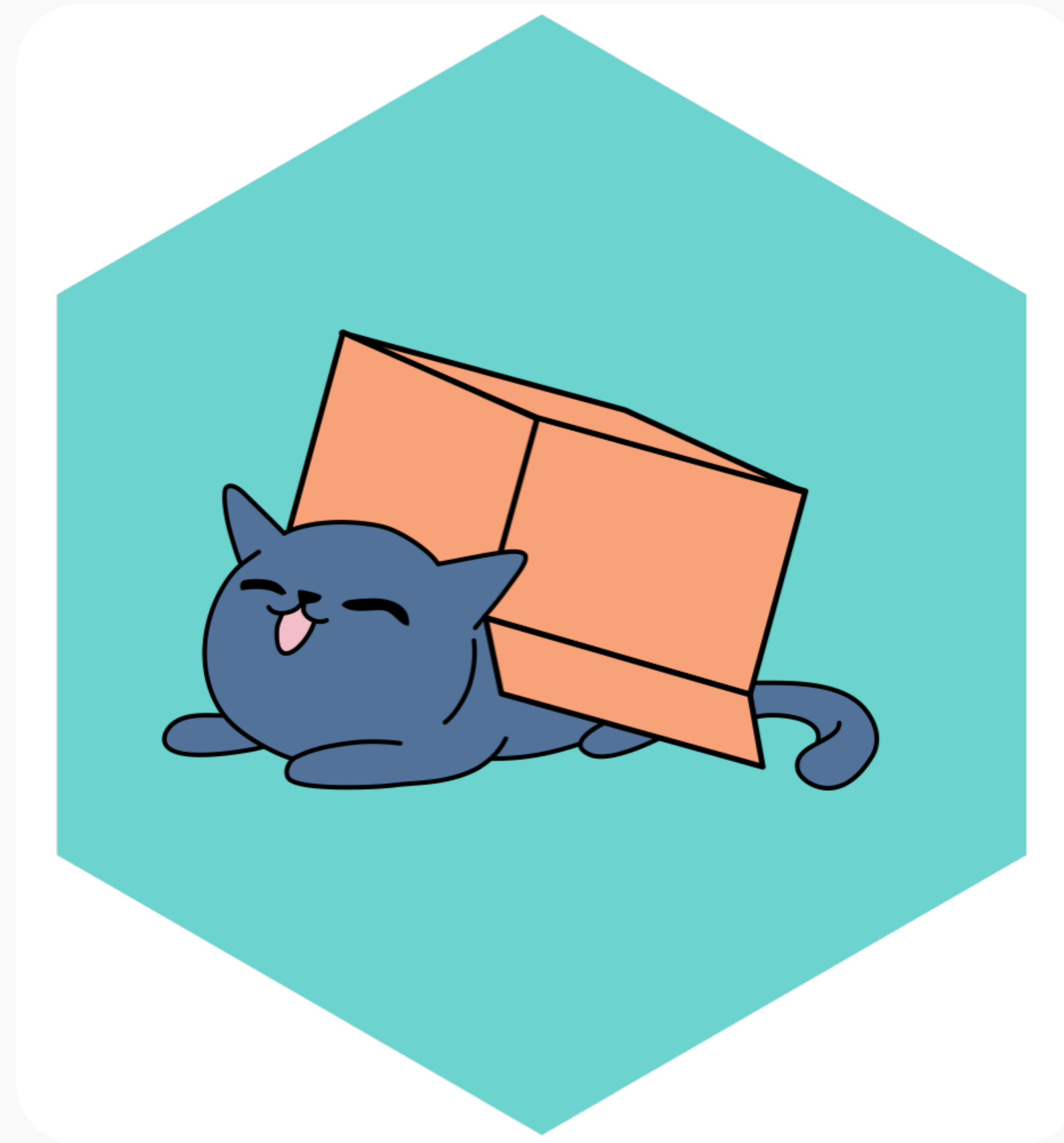
Асинхронное API

Использование с файберами неблокирующего и асинхронного API

Рантайм

Свой рантайм для лучшей утилизации потоков

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

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

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

Как писать код?

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

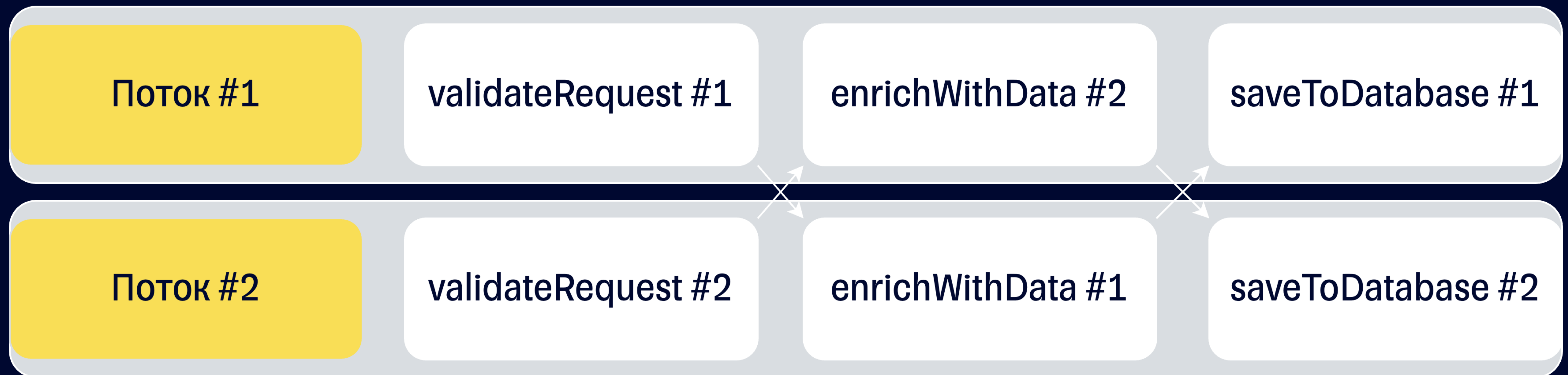
```scala
def handleRequest(req: Request): Response = {
  validateRequest(req)
  val data = enrichWithData(req)
  val response = saveToDatabase(data)
  response
}
```

```scala
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

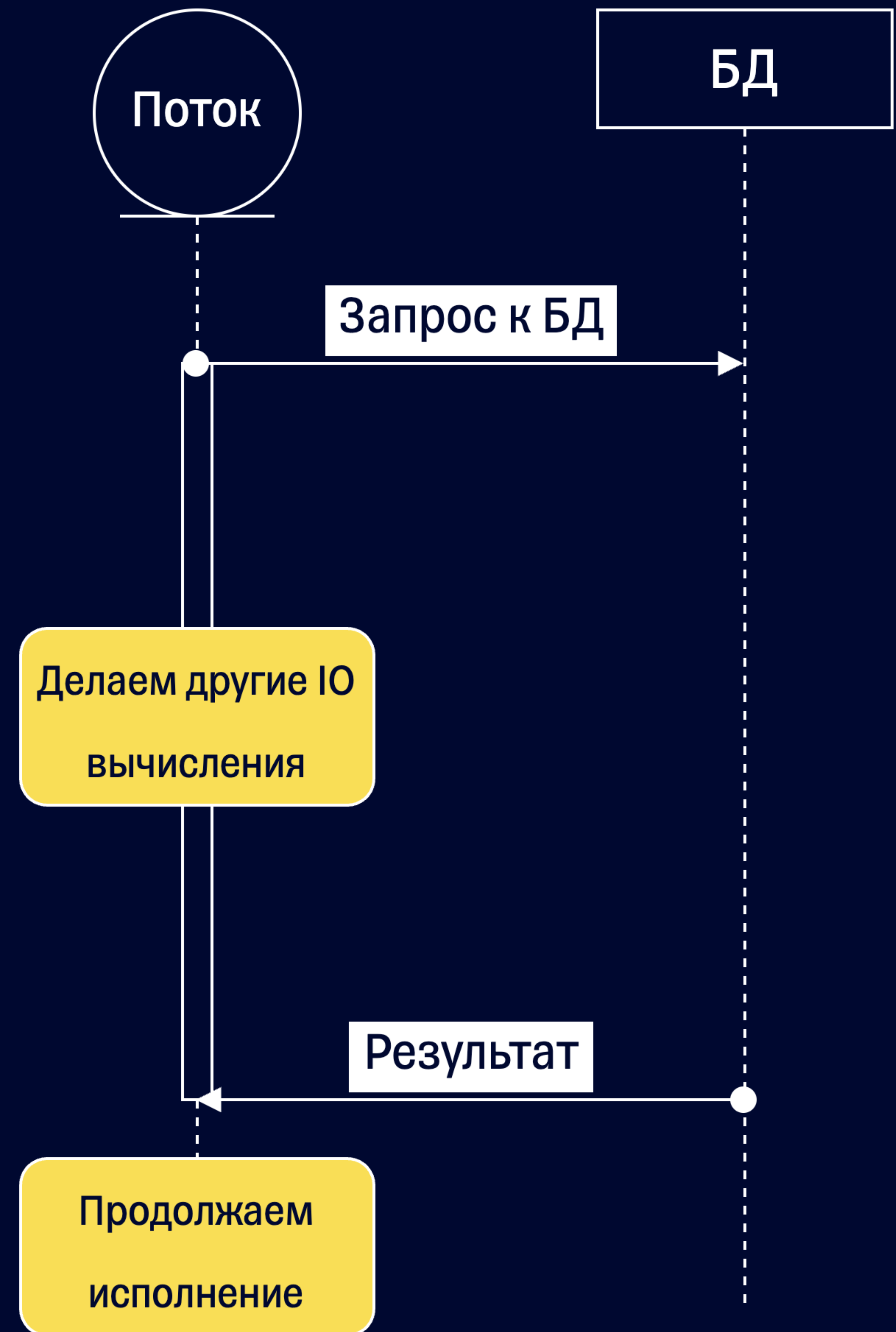


Как оно работает?

- Такой подход позволяет обходить блокировку потоков и встраивать работу с асинхронным I/O

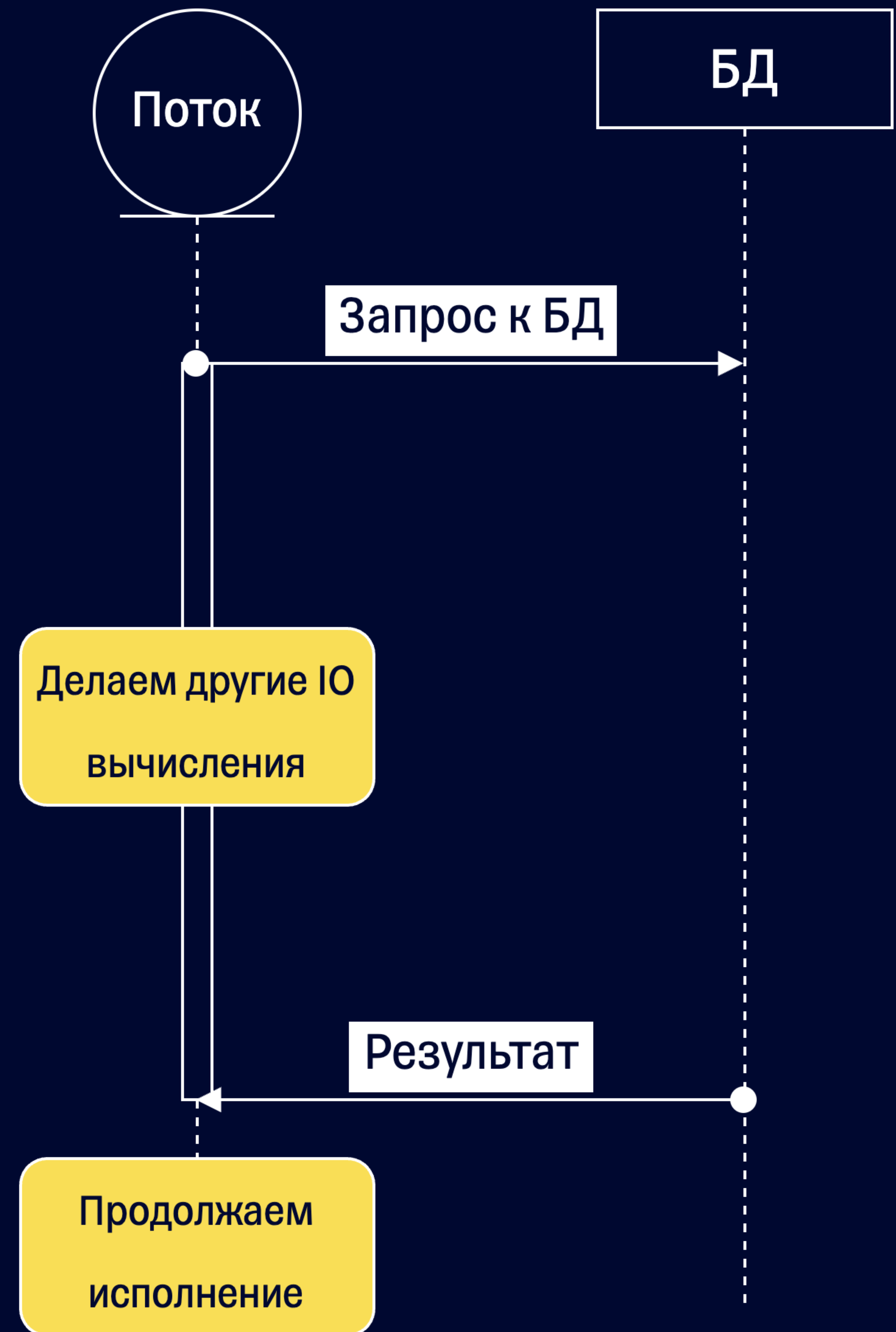
Как оно работает?

- Такой подход позволяет обходить блокировку потоков и встраивать работу с асинхронным I/O

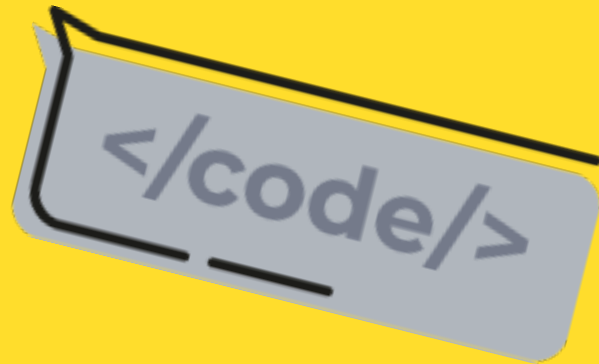


Как оно работает?

- Такой подход позволяет обходить блокировку потоков и встраивать работу с асинхронным I/O
- Можем исполнять другие IO вычисления, пока ждем ответа от БД
- При получении ответа можем продолжить обработку нашего запроса

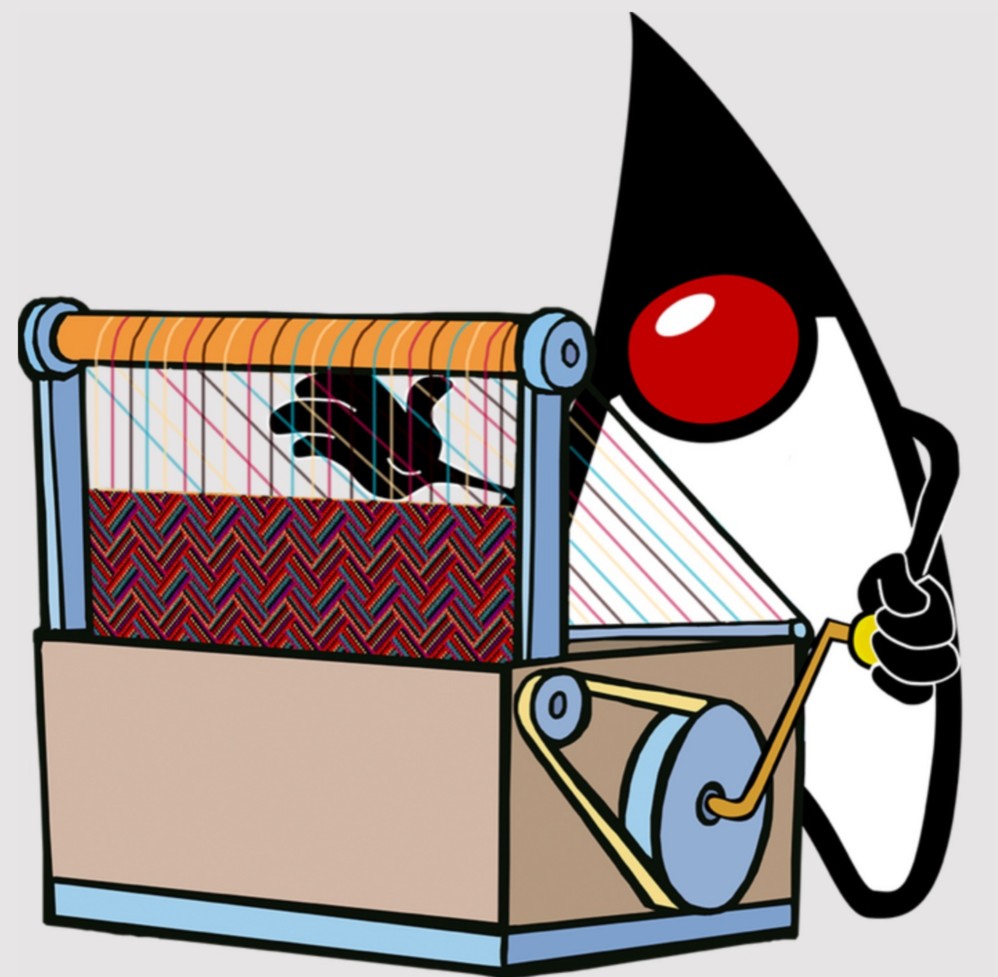


А что предлагает Loom?



- Какой путь избрал Loom?
- Какие задачи он решает?
- Помогает ли Loom только Java приложениям?

Project Loom



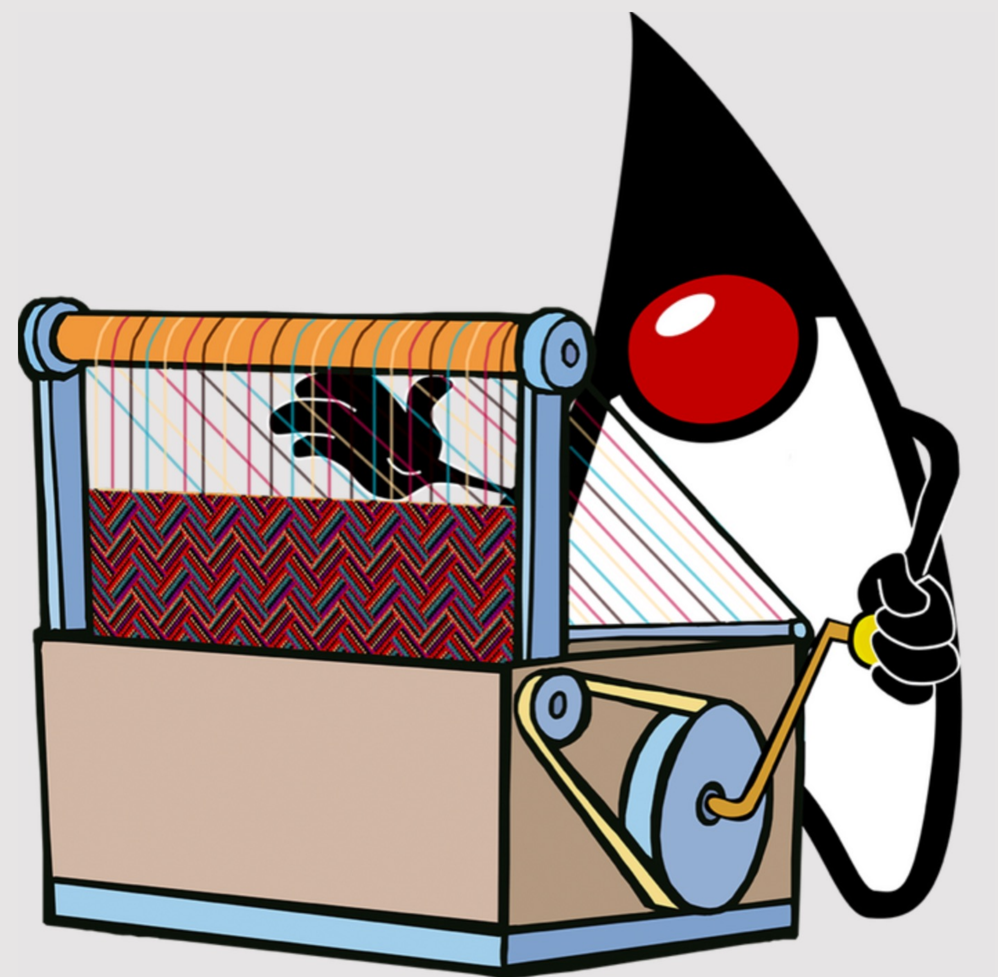
История

2017

Старт разработки Project Loom

- Цель: уменьшить усилия разработчиков при написании высоко-нагруженных систем

Project Loom



История

2017

Старт разработки Project Loom

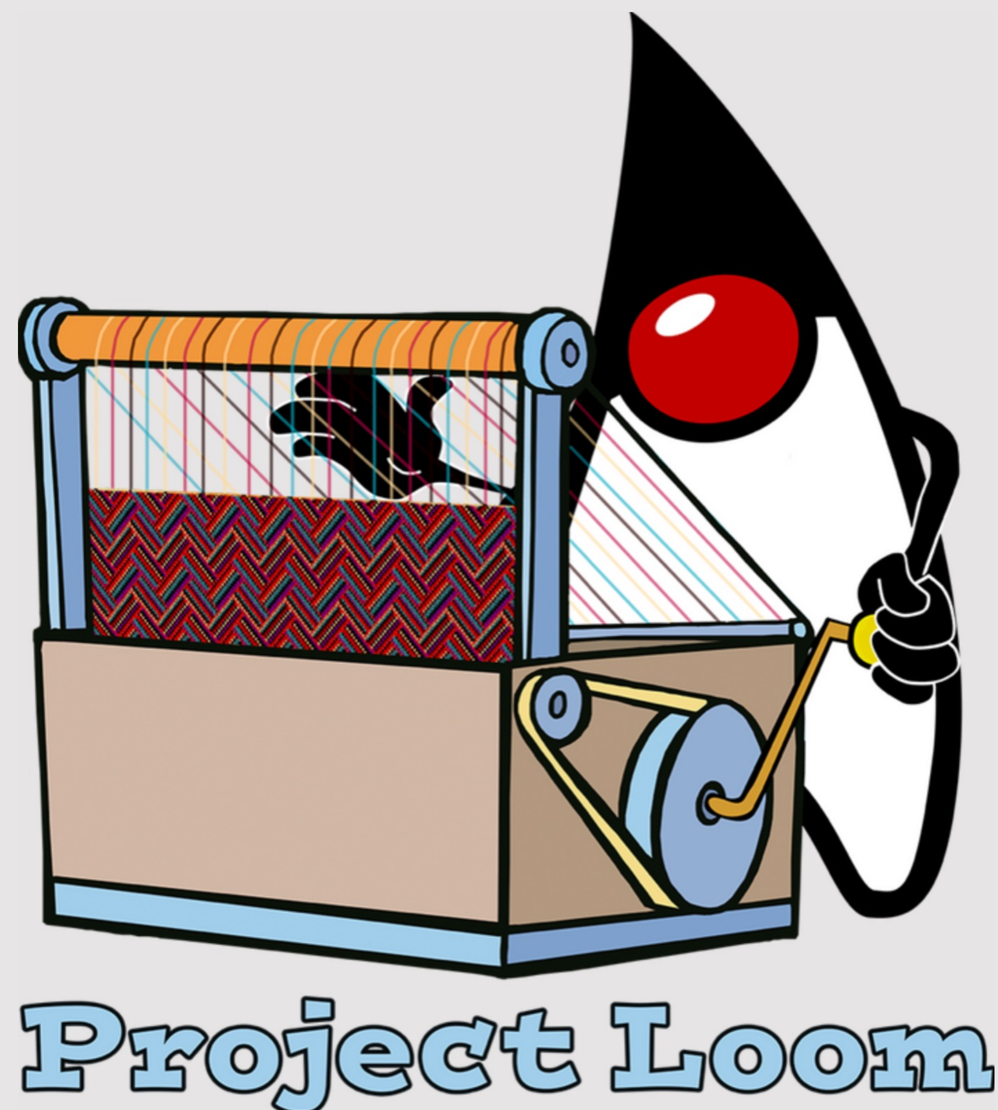
- Цель: уменьшить усилия разработчиков при написании высоко-нагруженных систем

2022

Первое появление в JDK 19

- JEP 425 – Виртуальные потоки (превью)
- JEP 428 – Структурированная конкурентность (инкубатор)

Project Loom



История

2017

Старт разработки Project Loom

- Цель: уменьшить усилия разработчиков при написании высоко-нагруженных систем

2022

Первое появление в JDK 19

- JEP 425 – Виртуальные потоки (превью)
- JEP 428 – Структурированная конкурентность (инкубатор)

2023

Первый релиз в JDK 21:

- JEP 444 – Виртуальные потоки (релиз)
- Превью остальных доработок

Project Loom

Project Loom мог пойти по пути Scala и ввести новую абстракцию (java.lang.Fiber), но вместо этого избрал кардинально другой путь

Project Loom

Project Loom мог пойти по пути Scala и ввести новую абстракцию (java.lang.Fiber), но вместо этого избрал кардинально другой путь

```
Thread.ofPlatform().start { () =>
  println("Hello from platform!")
}
```

```
Thread.ofVirtual().start { () =>
  println("Hello from virtual!")
}
```

Loom вводит новую классификацию java.lang.Thread. Теперь есть **платформенные** потоки и **виртуальные**

Виртуальные потоки

- Создание виртуальных потоков практически бесплатно
- Можно иметь миллионы виртуальных потоков

Виртуальные потоки

- Создание виртуальных потоков практически бесплатно
- Можно иметь миллионы виртуальных потоков
- Блокировка виртуального потока не приводит к блокировке платформенного
- При блокировке виртуальный поток освобождает место для исполнения других виртуальных потоков

Виртуальные потоки

- Достаточно поменять Executor, чтобы получить новый уровень производительности
- Можно все также писать простой последовательный код

```
val executor = Executors
  .newVirtualThreadPerTaskExecutor()


while (true) {
  val req = acceptRequest()
  executor.submit { () =>
    val response = handleRequest(req)
    serveResponse(response)
  }
}
```

Project Loom - Что еще?

Обратная совместимость

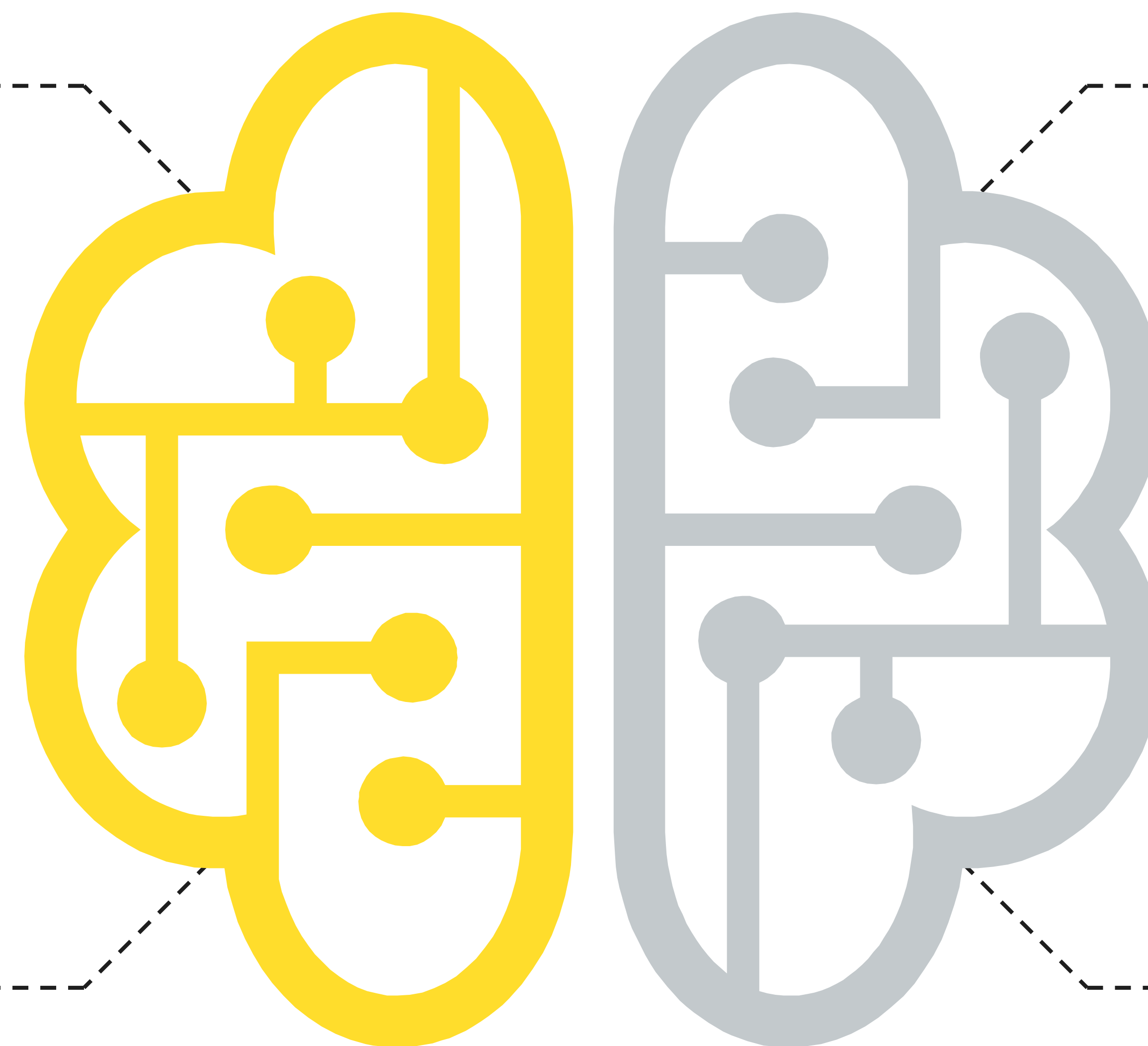
Минимальные изменения кода

Примитивы синхронизации

java.util.concurrent работает с виртуальными потоками

”Бесплатная” блокировка

Блокировка виртуальных потоков условно бесплатна



Мониторинг

Виртуальные потоки поддерживают мониторинг JDK

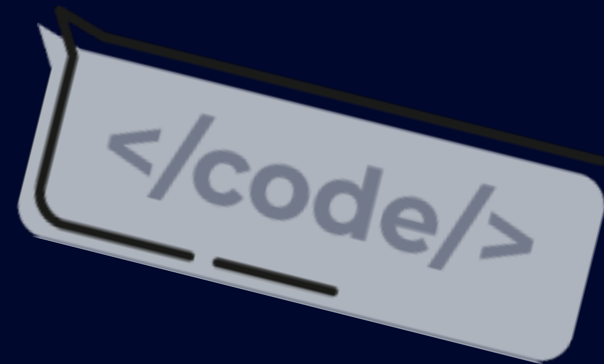
Ничего нового

Не нужно ничего знать про монады

Улучшенный I/O

Большинство старых API для I/O становится неблокирующим

Project Loom – серебряная пуля?



- Столько преимуществ. Не может же быть все так хорошо? Или может?
- Scala больше не нужна?

Ограничения

- Project Loom все-таки имеет ряд ограничений

Ограничения

- Project Loom все-таки имеет ряд ограничений
- Нативные вызовы (JNI) будут приводить к блокировке платформенного потока под виртуальным потоком
- synchronized блоки также будут приводить к блокировке платформенного потока

```
@native def callJni(param: String): Unit = ???

synchronized(this) {
  doStuff()
}
```

Ограничения

- Project Loom все-таки имеет ряд ограничений
- Нативные вызовы (JNI) будут приводить к блокировке платформенного потока под виртуальным потоком
- synchronized блоки также будут приводить к блокировке платформенного потока
- Маленькая цена за столько преимуществ

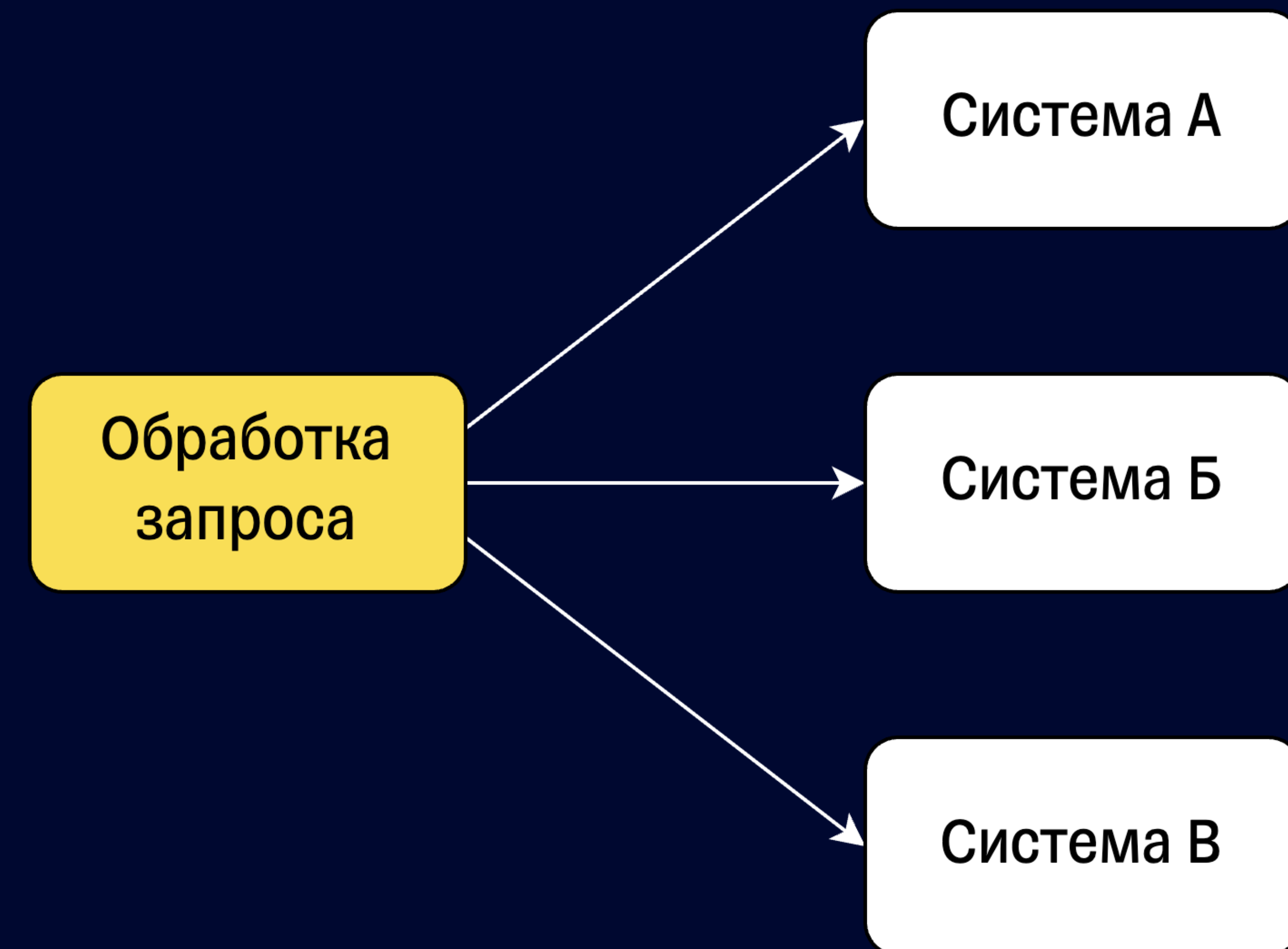
```
@native def callJni(param: String): Unit = ???

synchronized(this) {
  doStuff()
}
```

Главная проблема

- Требования к современным системам заставляют писать конкурентный код

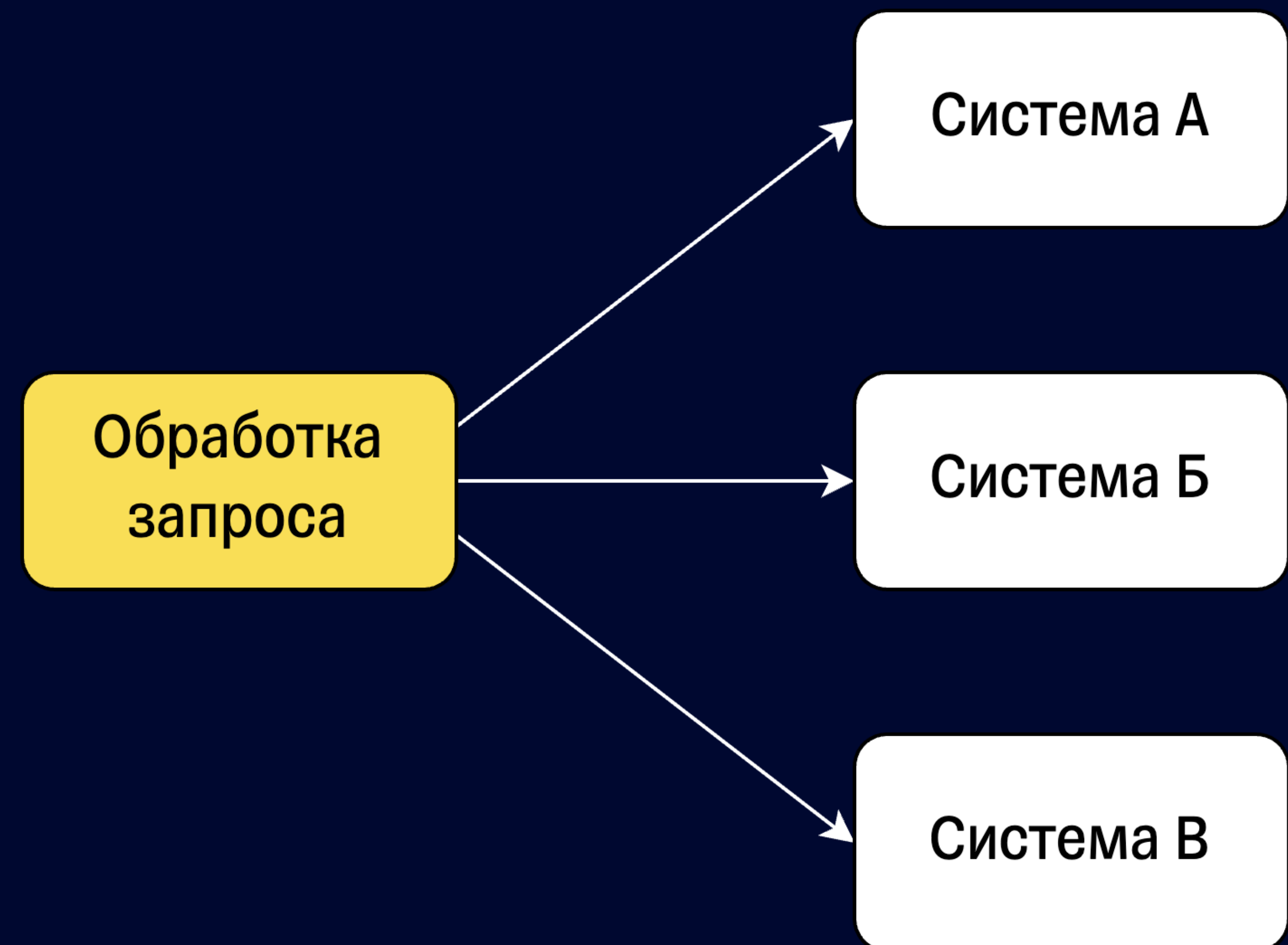
Главная проблема

- Требования к современным системам заставляют писать конкурентный код
- Некоторые запросы к системам не зависят друг от друга



Главная проблема

- Требования к современным системам заставляют писать конкурентный код
- Некоторые запросы к системам не зависят друг от друга
- Можно делать запросы к ним одновременно
- Оптимизируем время исполнения запроса



Главная проблема

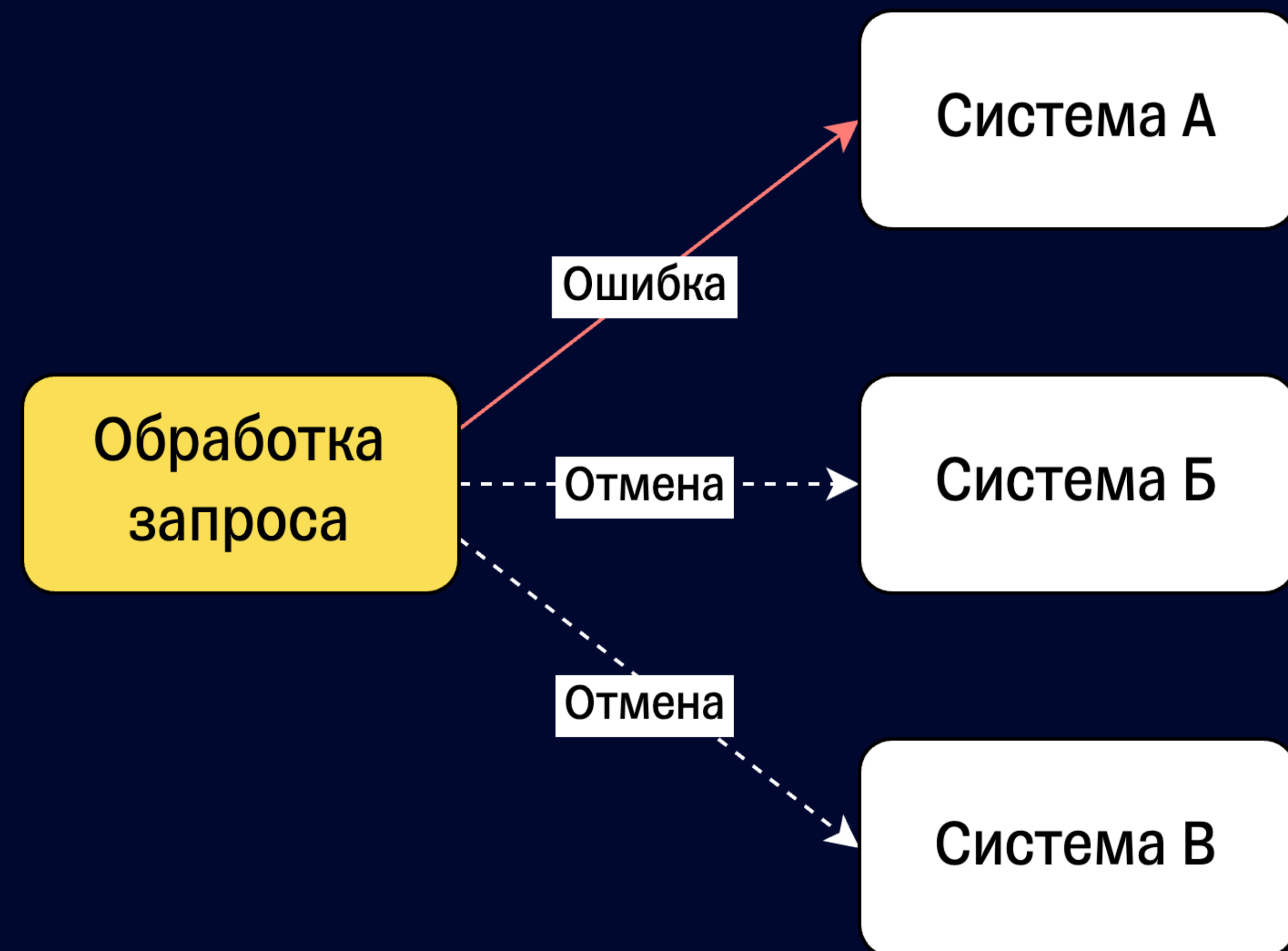
- Сложность таких задач заключается в правильной остановке вычислений

Главная проблема

- Сложность таких задач заключается в правильной остановке вычислений
- Правильно производить зачистку ресурсов

Главная проблема

- Сложность таких задач заключается в правильной остановке вычислений
- Правильно производить зачистку ресурсов
- Правильно задавать стратегию завершения
- Например: При возникновении ошибки при вызове системы А, останавливать вызов к системе Б



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

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

- Моделирование остановки через ошибку может приводить к неприятным ошибкам

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

Сложности

- Моделирование остановки через ошибку может приводить к неприятным ошибкам
- Вычисление в примере может быть не остановлено из-за обработки всех ошибок

```scala
def doComplexStuff(): Unit =
  while (true) {
    try {
      Thread.sleep(1_000L)
      doStuff()
    } catch {
      case ex: Throwable =>
        println(s"Exception happend
cleaning up: $ex")
    }
    if (Thread.interrupted()) {
      println("Cleaning up all
resources")
      throw new InterruptedException()
    }
  }
```

Сложности

- Моделирование остановки через ошибку может приводить к неприятным ошибкам
- Вычисление в примере может быть не остановлено из-за обработки всех ошибок
- То есть нужно всегда держать в голове, что InterruptedException может вылететь и должен корректно останавливать вычисление

```
def doComplexStuff(): Unit =
  while (true) {
    try {
      Thread.sleep(1_000L)
      doStuff()
    } catch {
      case ex: Throwable =>
        println(s"Exception happend
cleaning up: $ex")
    }
    if (Thread.interrupted()) {
      println("Cleaning up all
resources")
      throw new InterruptedException()
    }
  }
```

Сложности

- Моделирование остановки через ошибку может приводить к неприятным ошибкам
- Вычисление в примере может быть не остановлено из-за обработки всех ошибок
- То есть нужно всегда держать в голове, что InterruptedException может вылететь и должен корректно останавливать вычисление

```
def doComplexStuff(): Unit = {
  while (true) {
    try {
      Thread.sleep(1_000L)
      doStuff()
    } catch {
      case ex: InterruptedException =>
        println(s"Interrupt happened,
cleaning up: $ex")
        Thread.currentThread().interrupt()
      case ex: Throwable =>
        println(s"Exception happened,
cleaning up: $ex")
    }
    if (Thread.interrupted()) {
      println("Cleaning up all
resources")
      throw new InterruptedException()
    }
  }
}
```

Сложности

- Важно также держать в голове состояние флага отмены
- Важно его правильно обрабатывать и передать выше по коду исполнения

```
def doComplexStuff(): Unit = {
  while (true) {
    try {
      Thread.sleep(1_000L)
      doStuff()
    } catch {
      case ex: InterruptedException =>
        println(s"Interrupt happened, cleaning up: $ex")
       Thread.currentThread().interrupt()
      case ex: Throwable =>
        println(s"Exception happened, cleaning up: $ex")
    }
    if (Thread.interrupted()) {
      println("Cleaning up all resources")
      throw new InterruptedException()
    }
  }
}
```

Сложности

- Важно также держать в голове состояние флага отмены
- Важно его правильно обрабатывать и передать выше по коду исполнения
- Все вместе - большая сложность для разработчика

```
def doComplexStuff(): Unit = {
  while (true) {
    try {
      Thread.sleep(1_000L)
      doStuff()
    } catch {
      case ex: InterruptedException =>
        println(s"Interrupt happened, cleaning up: $ex")
        Thread.currentThread().interrupt()
      case ex: Throwable =>
        println(s"Exception happened, cleaning up: $ex")
    }
    if (Thread.interrupted()) {
      println("Cleaning up all resources")
      throw new InterruptedException()
    }
  }
}
```

«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

А что у cats-effect?

- Все IO вычисления по умолчанию – отменяемые

А что у cats-effect?

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

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

А что у cats-effect?

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

- 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

А что у cats-effect?

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

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

- 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

А что у cats-effect?

```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 ()
```

- 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

Гарантия отмены

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

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

Гарантия отмены

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

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

Гарантия отмены

```
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 ()
```

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

Гарантия отмены

```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 ()
```

- 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)

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 – Loom (превью JDK 21)

- StructuredTaskScope – новое API для структурированной конкурентности
- Можно завершить на первой ошибке через ShutdownOnFailure

```
def handleRequestOne(req: Request): Response =
  Using(new StructuredTaskScope.ShutdownOnFailure()) { scope =>
    val f1 = scope.fork(callServiceA(req))
    val f2 = scope.fork(callServiceB(req))

    scope.join()
    scope.throwIfFailed()

    val (resA, resB) = (f1.get(), f2.get())
    ...
  }
```

Конкурентность – Loom (превью JDK 21)

- StructuredTaskScope – новое API для структурированной конкурентности
- Можно завершить на первом результате через ShutdownOnSuccess

```
def handleRequestTwo(req: Request): Response =
  Using(new StructuredTaskScope.ShutdownOnSuccess[Result]()()) { scope =>
    scope.fork(callNodeA(req))
    scope.fork(callNodeB(req))

    val res = scope.join().result()
    ...
  }
```

Конкурентность – Loom (превью JDK 21)

- StructuredTaskScope по умолчанию либо обрабатывает ошибки, либо успешный результат
- Написание более сложной логики будет требовать своей реализации StructuredTaskScope. А это боль :(
- В Scala уже есть прототип более удобного API – библиотека Ox от Адама Варски (softwaremill)

Конкурентность - Ox

- Библиотека Ox вводит очень простые комбинаторы и конструкции для работы с конкурентными вычислениями
- Можно реализовать сложную логику без привязки к системам эффектов

```scala
import ox._

def handleRequestOne(req: Request): Response = {
  val (resA, resB) = par(callServiceA(req), callServiceB(req))
  ...
}

def handleRequestTwo(req: Request): Response = {
  val res = raceResult(callNodeA(req), callNodeB(req))
    ...
}
```

Давайте подведем итоги

Итоги

- Project Loom – это огромное инженерное чудо для Java и других JVM языков
- Он закрывает кучу болей для написания типовых backend приложений
- Нацеленность на обратную совместимость принесет огромные преимущества для уже написанных систем
- Можно будет держать больше соединений, лучше утилизировать имеющееся железо

Итоги - Парадокс

- Project Loom предлагает оставаться в старой парадигме написания синхронного кода

Итоги - Парадокс

- Project Loom предлагает оставаться в старой парадигме написания синхронного кода
- Современные системы требуют написания конкурентного кода, чтобы эффективнее обрабатывать запросы

Итоги - Парадокс

- Project Loom предлагает оставаться в старой парадигме написания синхронного кода
- Современные системы требуют написания конкурентного кода, чтобы эффективнее обрабатывать запросы
- Писать корректный конкурентный код, используя старое Java API, по-прежнему очень сложно

Итоги - Парадокс

- Project Loom предлагает оставаться в старой парадигме написания синхронного кода
- Современные системы требуют написания конкурентного кода, чтобы эффективнее обрабатывать запросы
- Писать корректный конкурентный код, используя старое Java API, по-прежнему очень сложно
- Project Loom не дает отказаться или от высоко-уровневых фреймворков в Java, или от корутин Kotlin, или от систем эффектов в Scala

Итоги - Парадокс

- Поэтому, на текущий момент написание конкурентного кода – это нерешенная задача в рамках Project Loom

Итоги - Парадокс

- Поэтому, на текущий момент написание конкурентного кода – это нерешенная задача в рамках Project Loom
- Есть JEP 428/437/453 про Структурированную Конкурентность, но и он далек от идеала

Итоги - Парадокс

- Поэтому, на текущий момент написание конкурентного кода – это нерешенная задача в рамках Project Loom
- Есть JEP 428/437/453 про Структурированную Конкурентность, но и он далек от идеала
- Чтобы упростить написание конкурентного кода необходимы новые конструкции в языке / стандартной библиотеке
- Project Loom может стать для этого отличной платформой, остается только их дождаться

Итоги – Преимущества для Scala

- Системы эффектов в Scala дают небывалую гибкость для написания корректного конкурентного кода
- Они же имеют высокий порог входа. Нужно изучать другую парадигму написания кода
- Сами системы эффектов станут более эффективными при работе с Java библиотекам. (Например, в случае с JDBC драйверами)

Итоги – Преимущества для Scala

- В будущем, на основе Project Loom могут возникнуть новые более простые подходы и в Scala для написания конкурентного кода.
- Библиотека Ox – это только пример API, которого можно достичь с помощью Project Loom
- Project Caprese от Мартина Одерски может открыть новые горизонты написания конкуретного кода

На этом все!
Вопросы?

Scala Digest на Хабр

Слайды