

Project Loom – серебряная пуля?

Или все же нет?

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



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

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

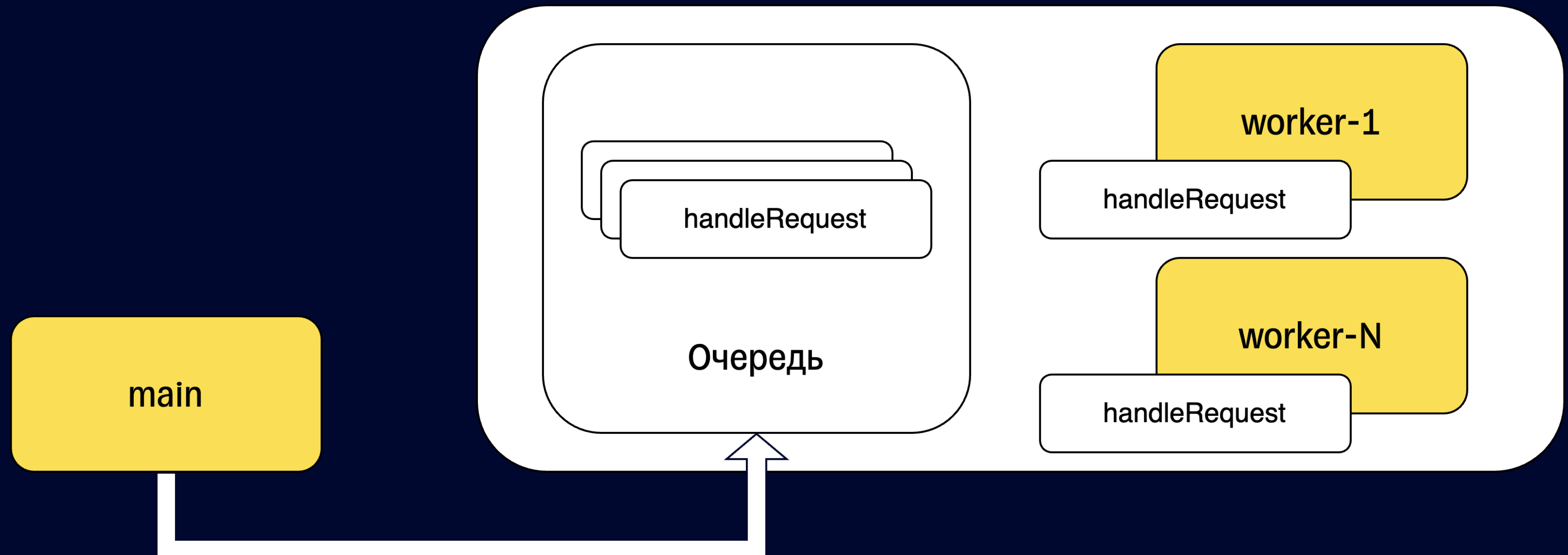
Проблематика

- 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

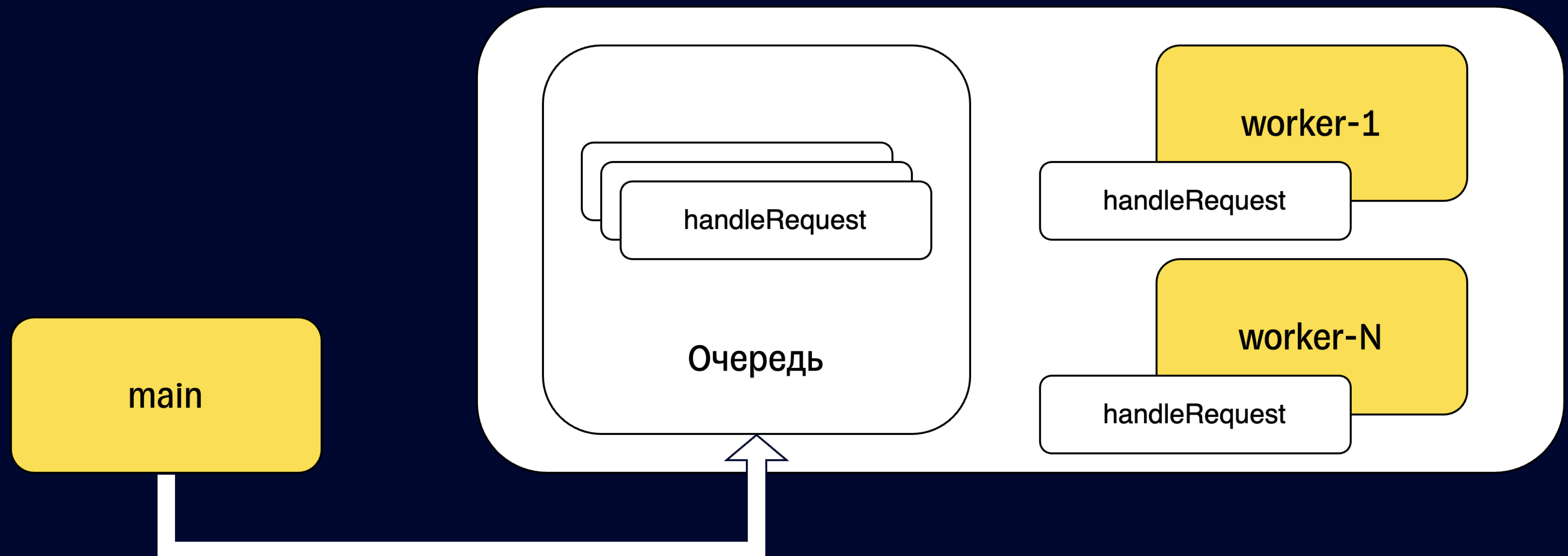


Проблема 10К соединений

Одновременная обработка запросов ограничена N

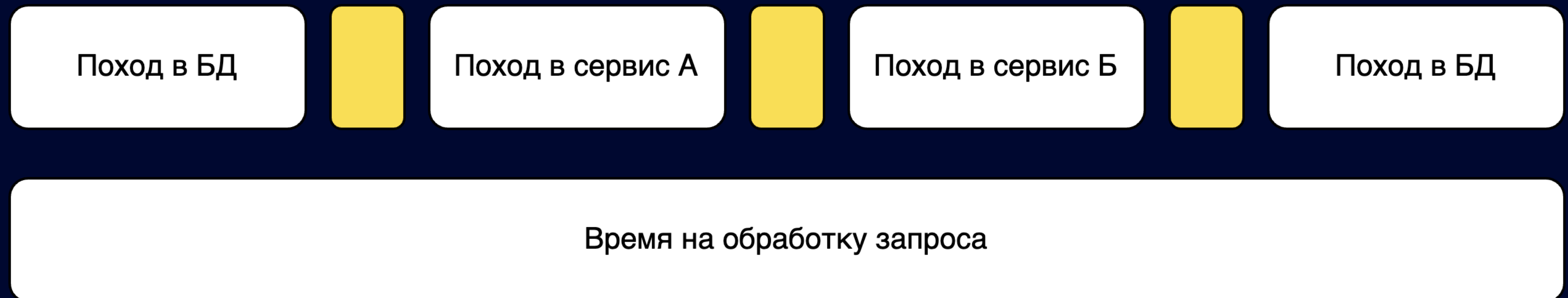
Чем больше N, тем больше переключений контекста

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 \approx потраченному времени на I/O

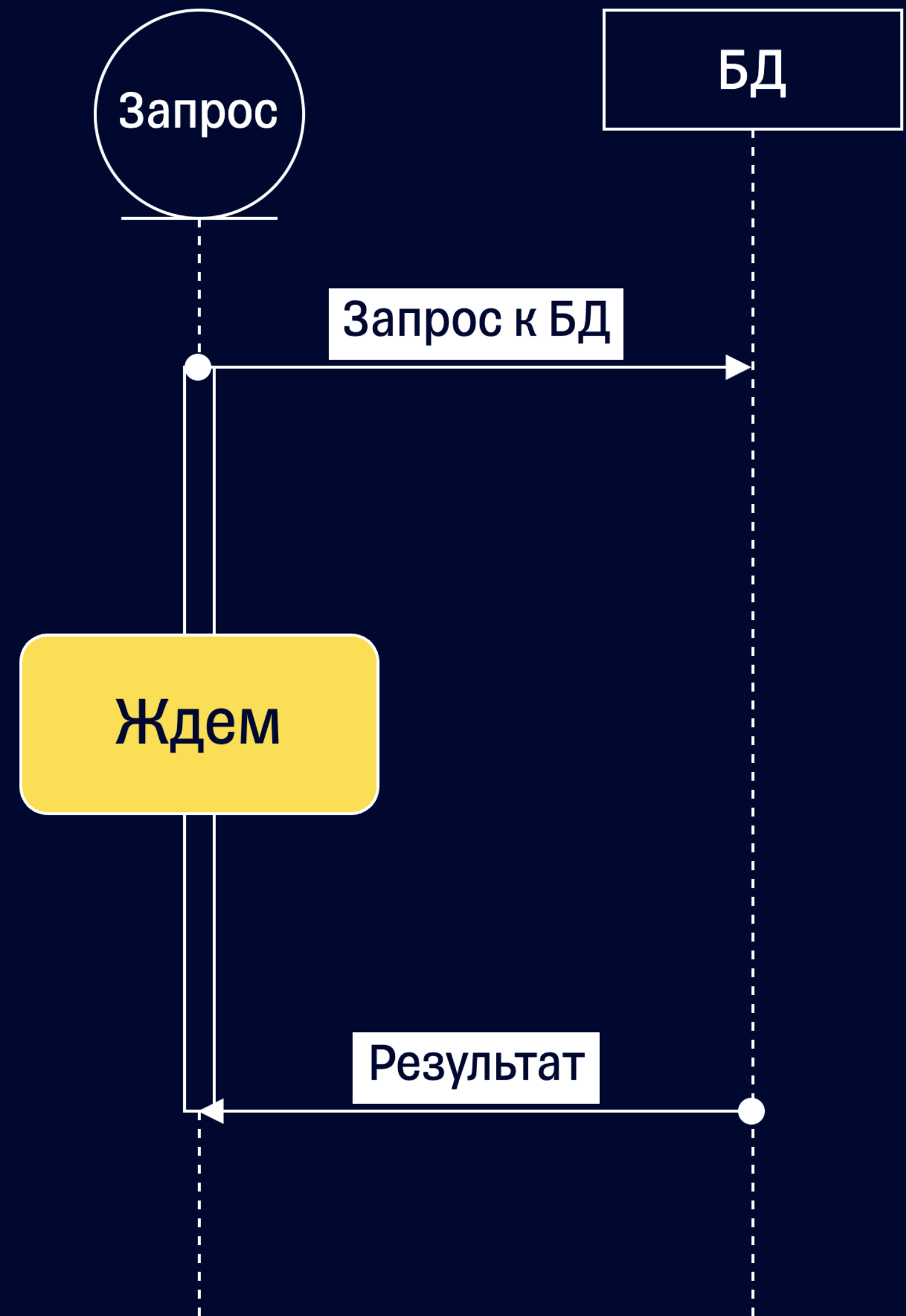


Синхронное API

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

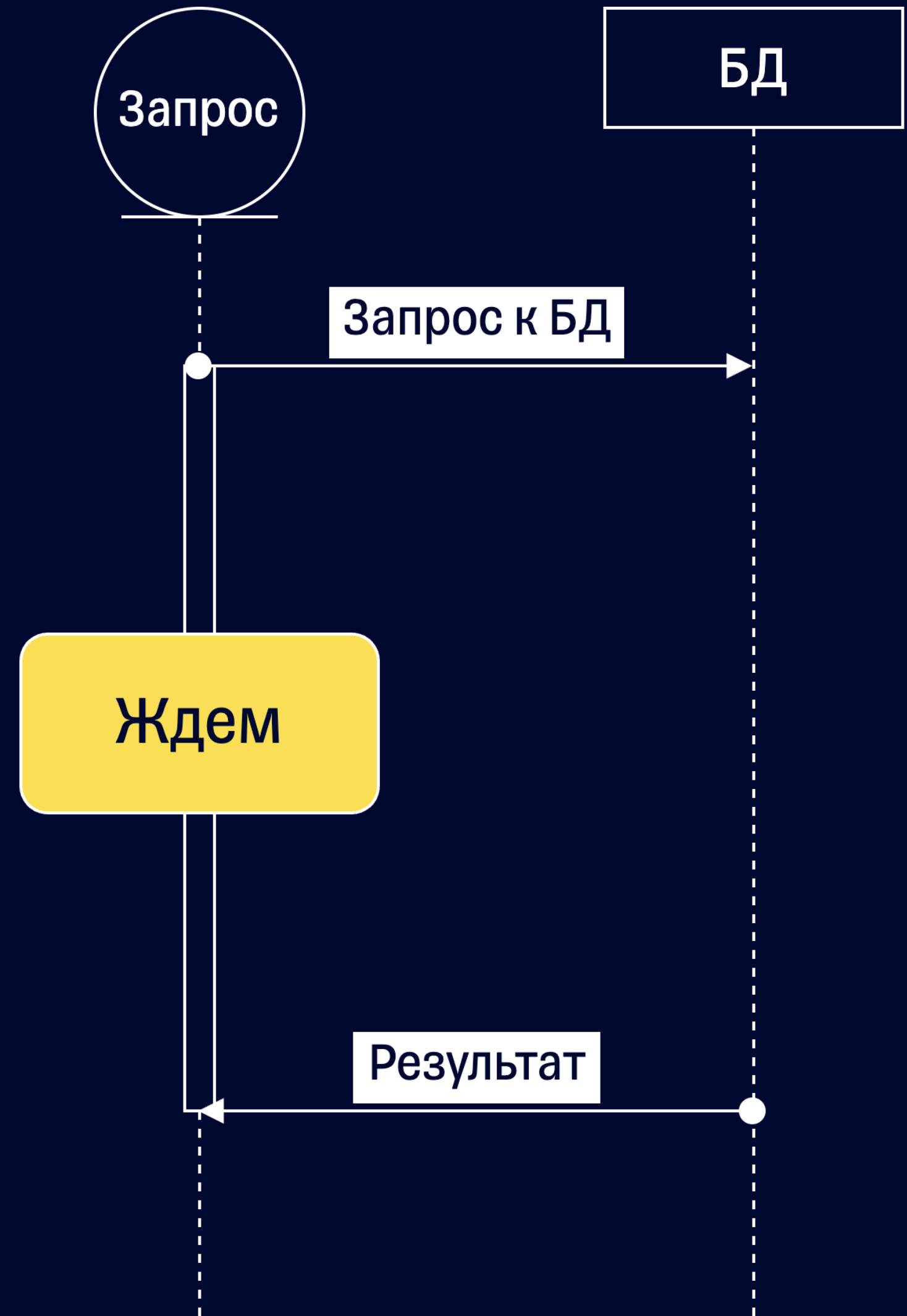
Синхронное API

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

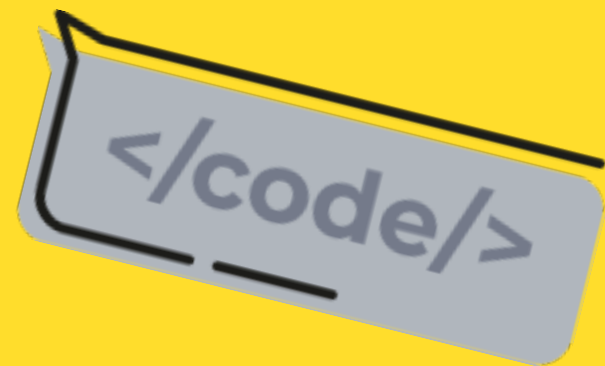


Синхронное API

- 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



Как другие языки и подходы решают схожую задачу?



- Как они позволяют лучше утилизировать потоки?
- Как они обходят блокировку потоков?

Что предлагает Scala и Kotlin?



Новые абстракции

Они вводят новые абстракции: корутины и фиберы



Асинхронное API

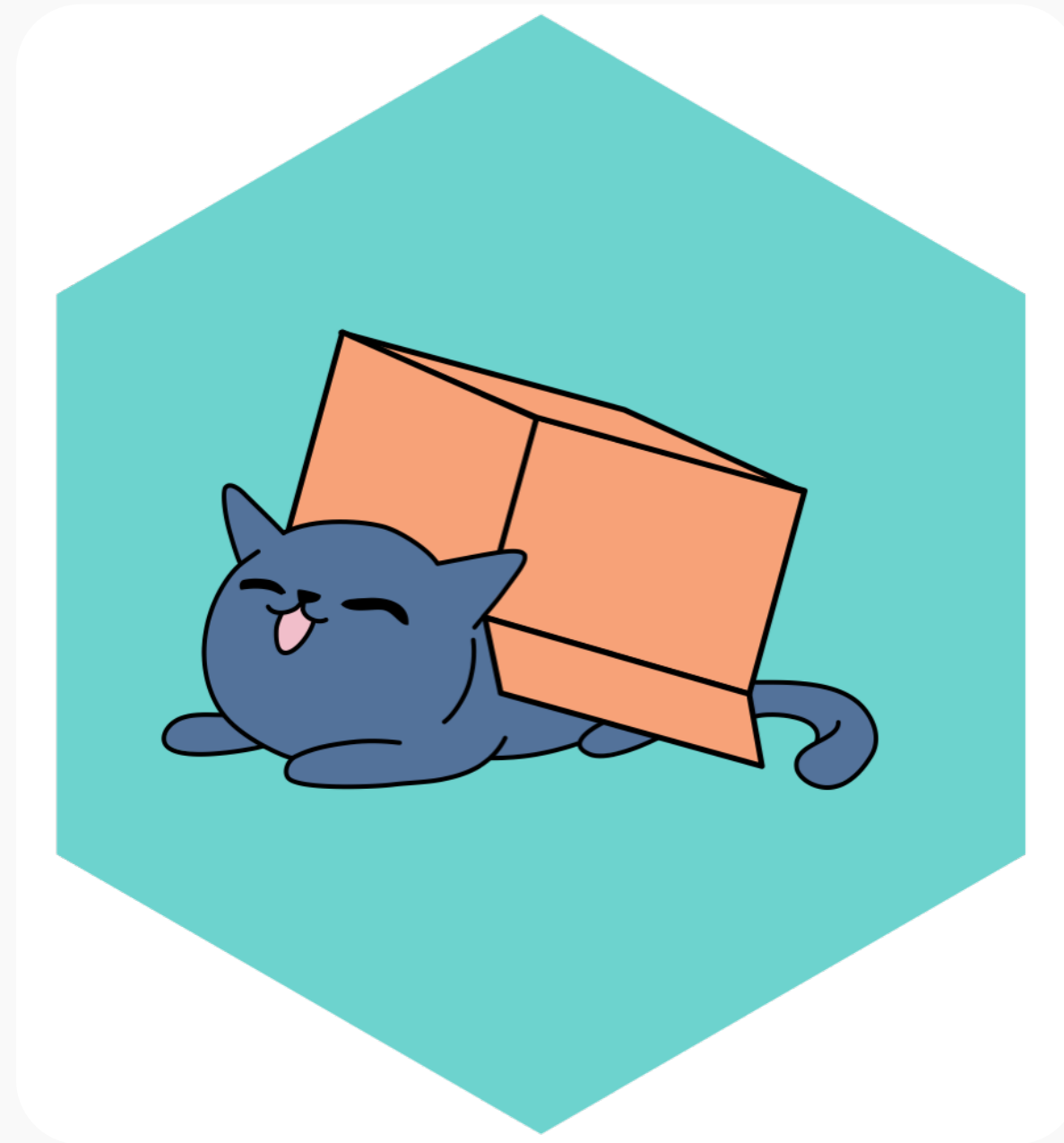
Используют с абстракциями неблокирующее и асинхронное API для I/O операционной системы



Рантайм

Используют 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

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



```
def handleRequest(req: Request): IO[Response] =  
  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

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

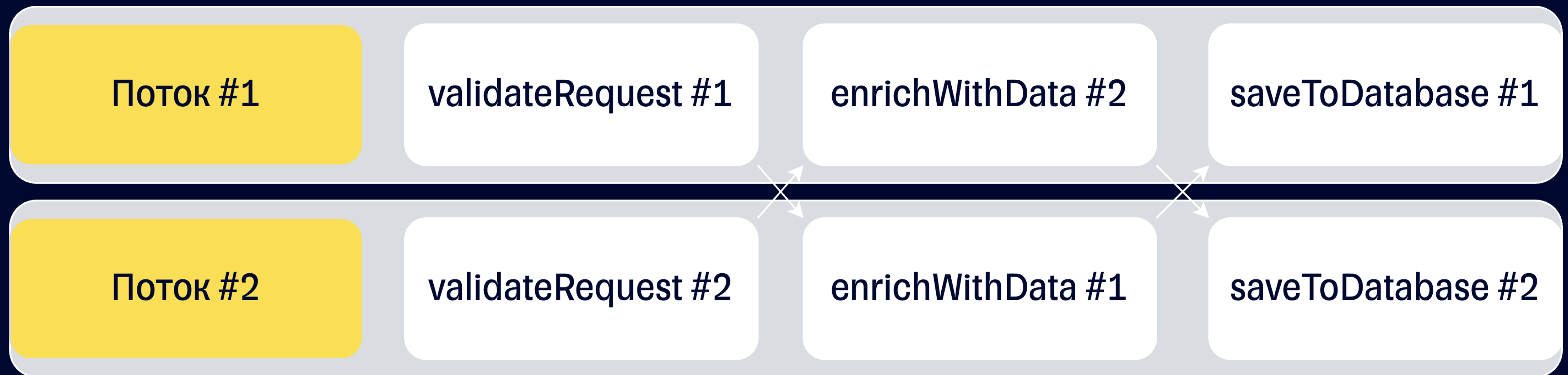


```
def handleRequest(req: Request): IO[Response] =  
  for {  
    _ <- validateRequest(req)  
    data <- enrichWithData(req)  
    response <- saveToDatabase(data)  
  } yield response
```

- Все вычисления теперь обернуты в структуру данных IO
- С IO можно использовать синтаксис с for, можно писать такой же императивный код

Как оно работает?

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

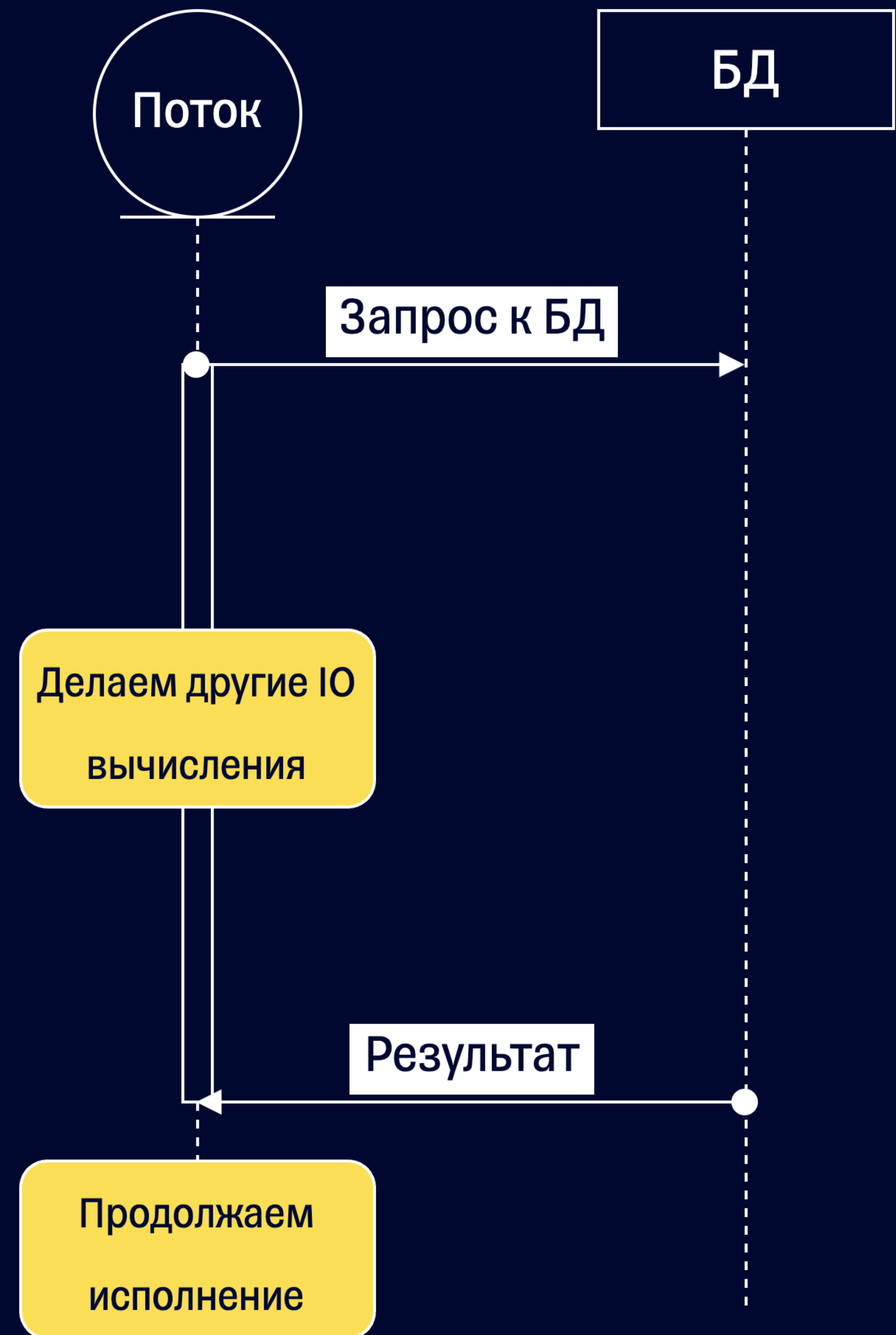


Как оно работает?

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

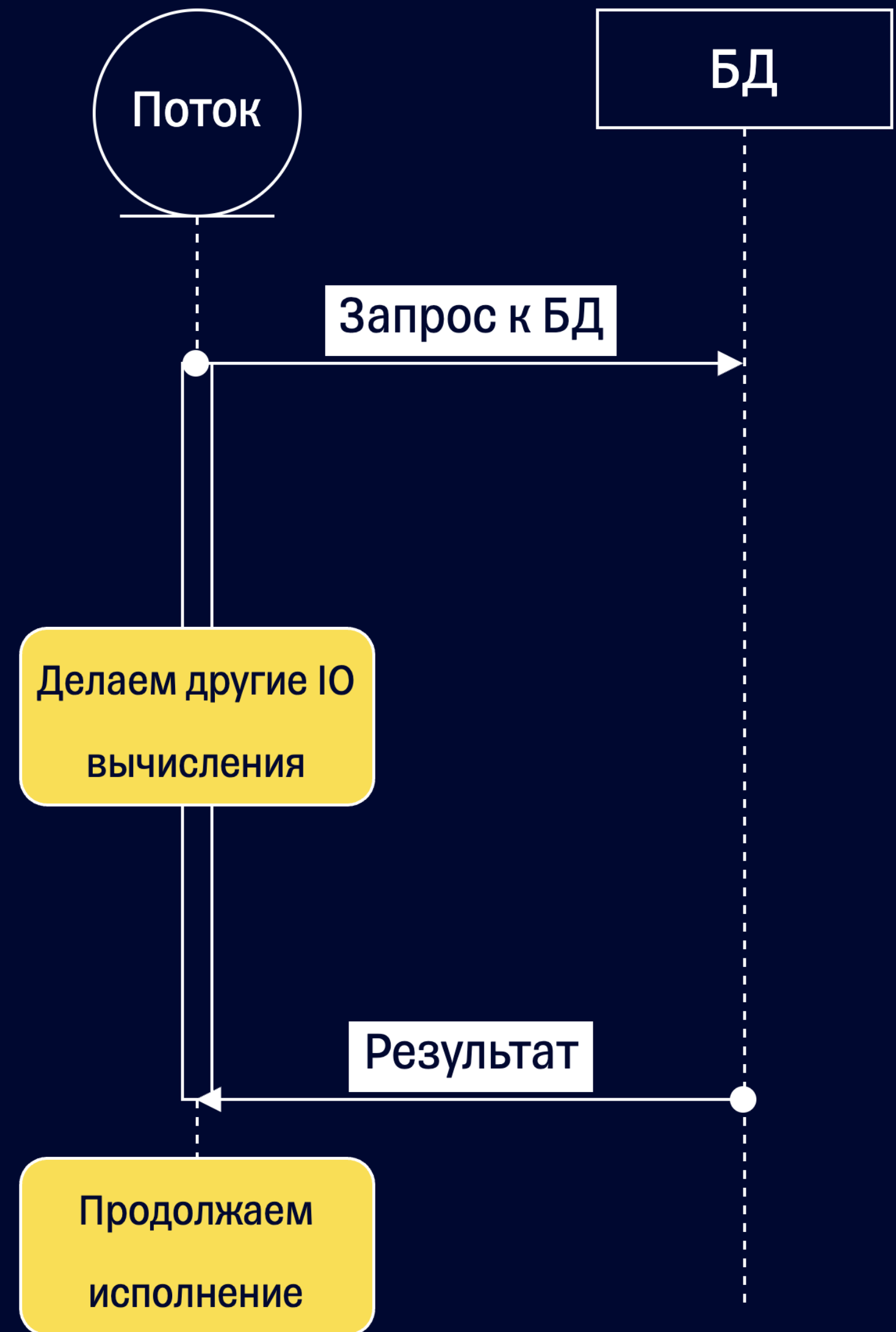
Как оно работает?

- 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



Системы эффектов - Плюсы

Преодолеваем порог
обработки 10к соединений

Системы эффектов - Плюсы

Преодолеваем порог
обработки 10к соединений



Нет простоям

Делаем работу, пока ждем
ответ от I/O

Системы эффектов - Плюсы

Преодолеваем порог
обработки 10к соединений

✓ Нет простоям

Делаем работу, пока ждем
ответ от I/O



Меньше потоков

Количество потоков сравнимо с
количеством ядер процессора



Меньше переключений

Меньше потоков ОС = меньше
переключений контекста

Системы эффектов - Минусы

Не все так идеально

Системы эффектов - Минусы

Не все так идеально



Переписывание кода

Все нужно переписать на Scala

Системы эффектов - Минусы

Не все так идеально



Новые подходы

Накладные расходы на обучение разработчиков



Переписывание кода

Все нужно переписать на Scala

Системы эффектов - Минусы

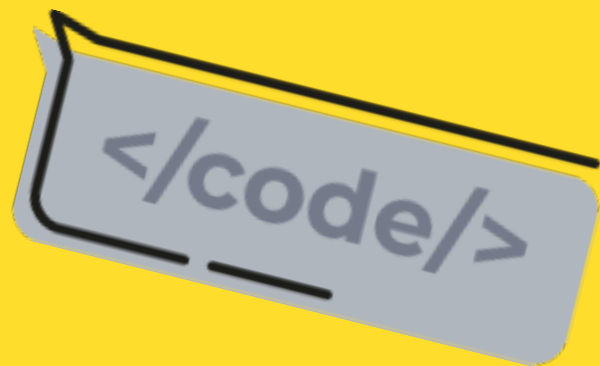
Не все так идеально

- ✓ **Новые подходы**
Накладные расходы на обучение разработчиков

- ✓ **Переписывание кода**
Все нужно переписать на Scala

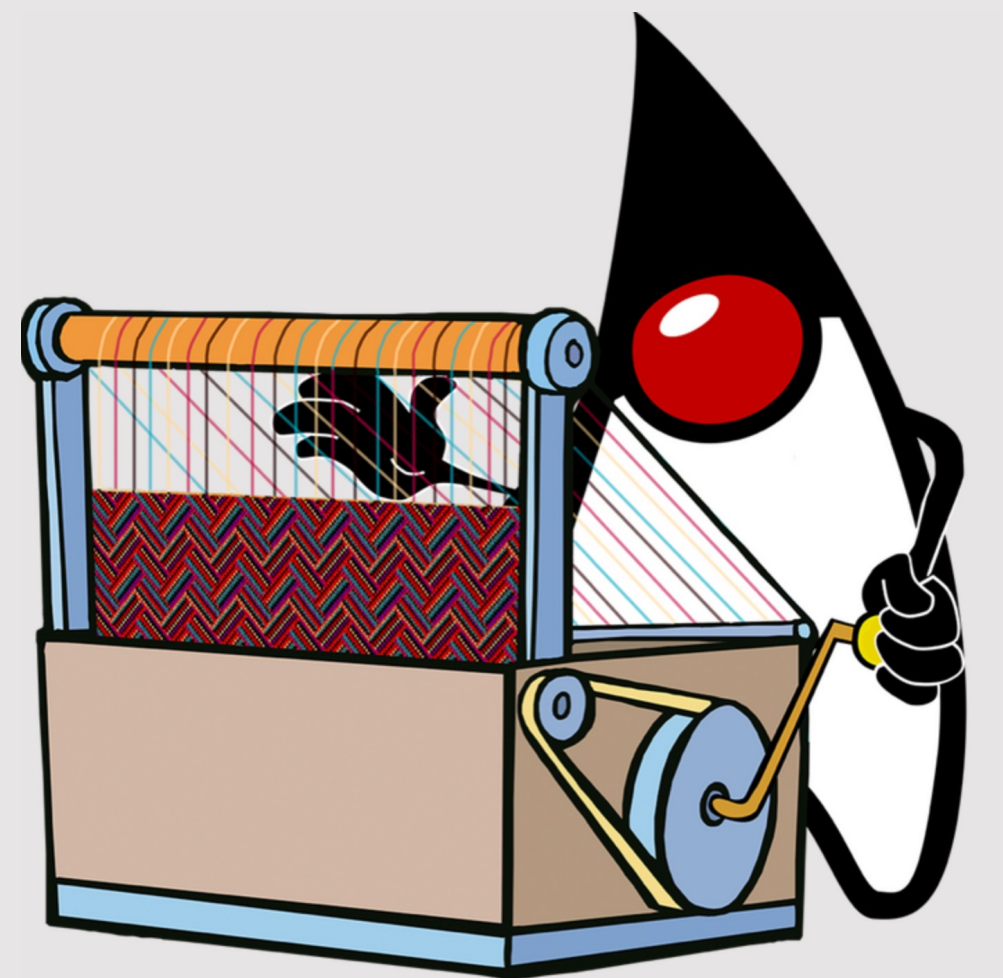
- ✓ **Тулинг**
Не такой богатый тулинг для мониторинга

А что предлагает Loom?



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

Project Loom



Project Loom

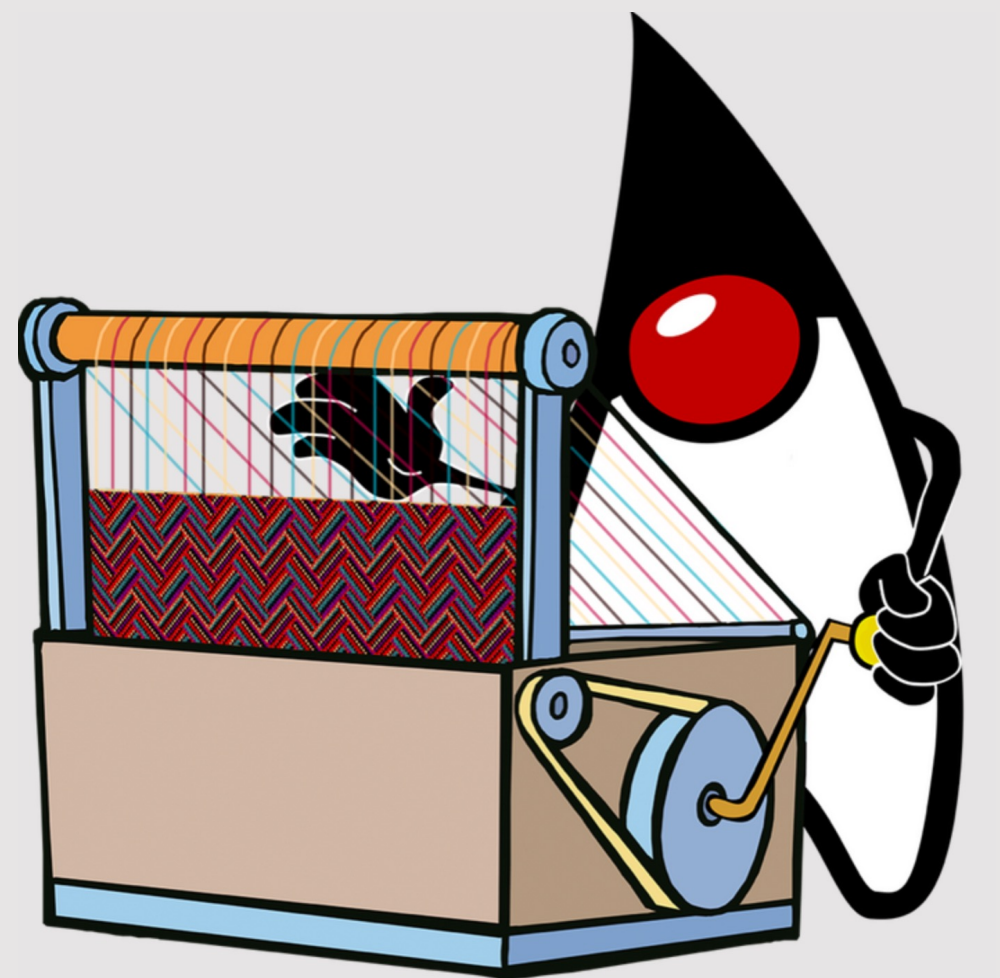
История

2017

Старт разработки Project Loom

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

Project Loom



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 Kotlin и ввести новую абстракцию ([java.lang.Routine](#)), но вместо этого избрал кардинально другой путь

Project Loom

Project Loom мог пойти по пути Scala и Kotlin и ввести новую абстракцию ([java.lang.Routine](#)), 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

- Виртуальные потоки не требуют кардинального переписывания кода (как в случае с cats-effect)

Виртуальные потоки

- Виртуальные потоки не требуют кардинального переписывания кода (как в случае с cats-effect)
- 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

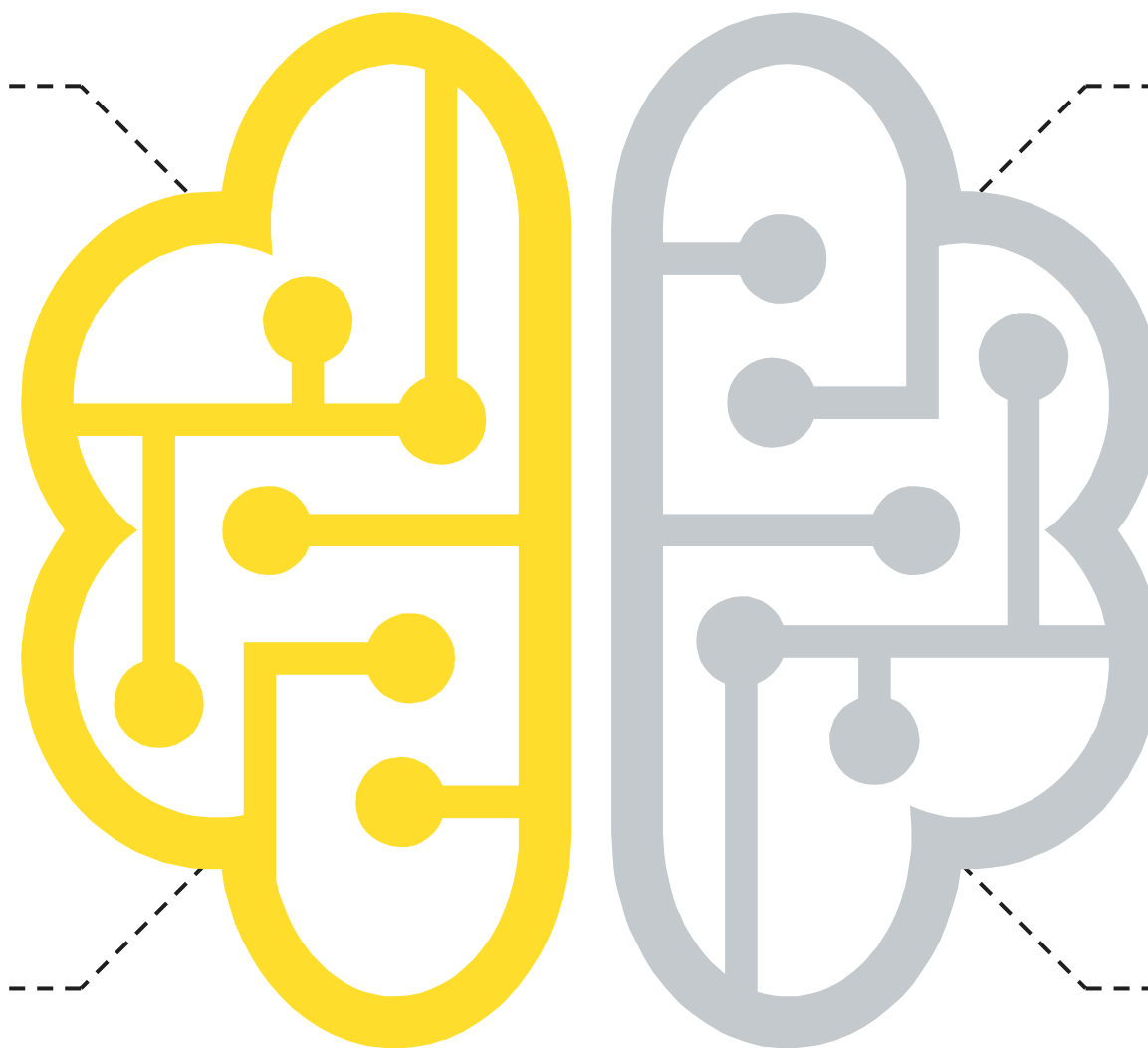
Минимальные изменения кода

Примитивы синхронизации

`java.util.concurrent` работает с виртуальными потоками

”Бесплатная” блокировка

Блокировка виртуальных потоков условно бесплатна



Мониторинг

Виртуальные потоки поддерживают мониторинг JDK

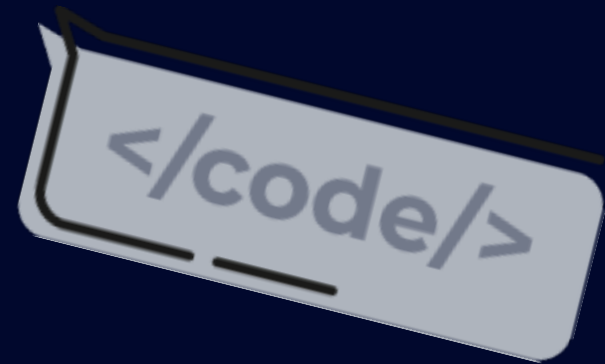
Ничего нового

Используется всем известное API, не нужно учить новые концепты

Улучшенный I/O

Большинство старых API для I/O становится неблокирующим

Project Loom – серебряная пуля?



- Не может же быть все так хорошо?
- Или может?

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

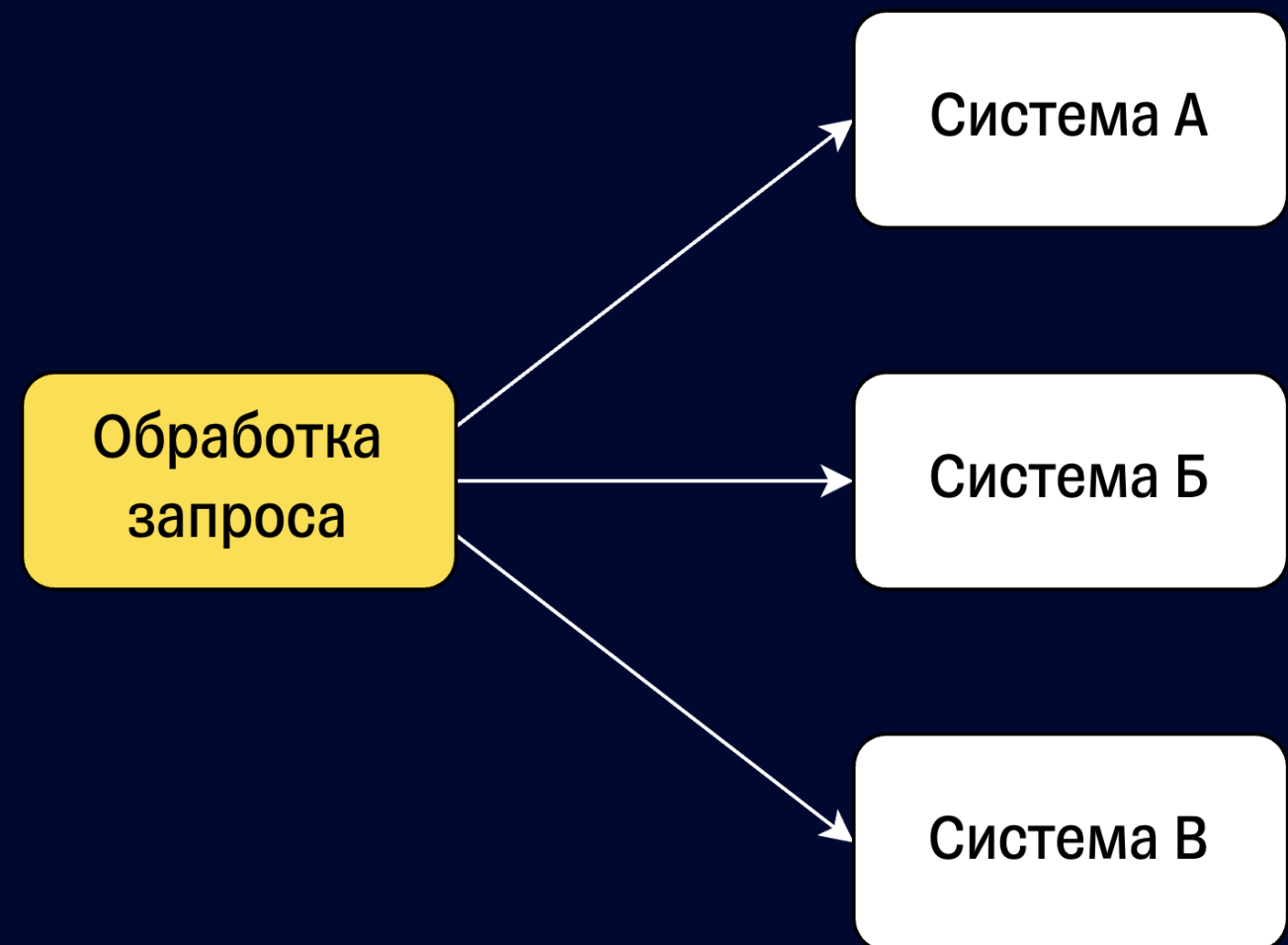
```
@native def callJni(param: String): Unit = ???  
  
synchronized(this) {  
    doStuff()  
}
```

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
- Можно делать запросы к ним одновременно
- Оптимизируем время исполнения запроса



Главная проблема

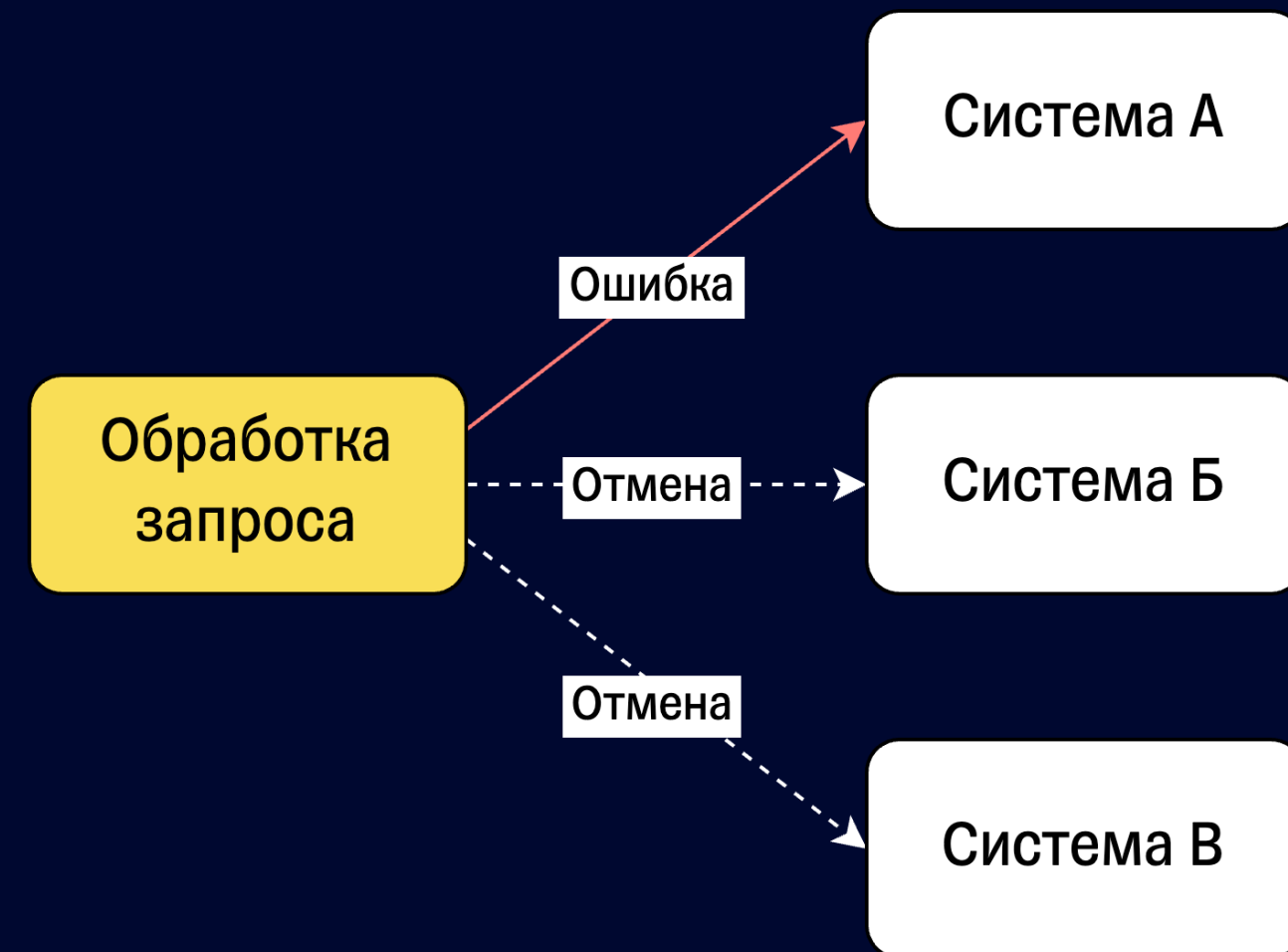
- Сложность таких задач заключается в правильной остановке вычислений

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

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

```
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

```
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

СЛОЖНОСТИ

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

```
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


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

А что у cats-effect?

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

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

А что 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

```
trait Fiber[A] {  
  def join: IO[Outcome[A]]  
  def cancel: IO[Unit]  
}  
  
def loop: IO[Unit] =  
  for {  
    _ <- IO.println("Hello!")  
    _ <- loop  
  } yield ()
```

А что 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

```

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 ()

```

А что 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

```
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

```
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


```
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

```
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)

Обработка результата

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

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

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

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

Обработка результата

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

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

Обработка результата

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

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

Конкурентность

- Все это позволяет реализовать простое API для сложных конкурентных вычислений

Конкурентность

- Все это позволяет реализовать простое API для сложных конкурентных 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 конкурентных 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
```


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

- Мы сегодня подробно рассмотрели только JEP 425/436/ 444 про Виртуальные Потоки. Только они попадут в релиз JDK 21. Но Project Loom им не ограничен.

Итоги - Парадокс

- Мы сегодня подробно рассмотрели только JEP 425/436/ 444 про Виртуальные Потоки. Только они попадут в релиз JDK 21. Но Project Loom им не ограничен.
- Специально были опущены JEP 428/437/453 про Структурированную Конкурентность

Итоги - Парадокс

- Мы сегодня подробно рассмотрели только JEP 425/436/ 444 про Виртуальные Потоки. Только они попадут в релиз JDK 21. Но Project Loom им не ограничен.
- Специально были опущены JEP 428/437/453 про Структурированную Конкурентность
- Предлагаемое им API еще сыровато и не настолько удобно, как корутины Kotlin и системы эффектов в Scala (можно сделать отдельный доклад на эту тему)

Итоги - Парадокс

- Мы сегодня подробно рассмотрели только JEP 425/436/ 444 про Виртуальные Потоки. Только они попадут в релиз JDK 21. Но Project Loom им не ограничен.
- Специально были опущены JEP 428/437/453 про Структурированную Конкурентность
- Предлагаемое им API еще сыровато и не настолько удобно, как корутины Kotlin и системы эффектов в Scala (можно сделать отдельный доклад на эту тему)
- Остается ждать либо пока доработают его, либо пока появятся кардинально новые подходы с использованием Project Loom

Итоги – Преимущества для Scala

- Системы эффектов в Scala дают небывалую гибкость для написания корректного конкурентного кода

Итоги – Преимущества для Scala

- Системы эффектов в Scala дают небывалую гибкость для написания корректного конкурентного кода
- Сами системы эффектов станут более эффективными при работе с Java библиотекам. (Например, в случае с JDBC драйверами)

Итоги – Преимущества для Scala

- Системы эффектов в Scala дают небывалую гибкость для написания корректного конкурентного кода
- Сами системы эффектов станут более эффективными при работе с Java библиотекам. (Например, в случае с JDBC драйверами)
- В будущем в Scala на основе Project Loom могут возникнуть новые более простые подходы для написания конкурентного кода. Например, Project Caprese от Мартина Одерски может открыть новые горизонты написания конкурентного кода



Вопросы?