

Java после Loom

альтернативные модели конкурентности

МИР Plat.Form

Прогревочная

t.me/pro_grevy

Обо мне

linkedin.com/in/alantsov

Мир Plat.Form

mir-platform.ru



ЗАЧЕМ ЭТОТ ДОКЛАД?

❖ JEP 444: Virtual Threads

«It is not a goal to change the basic concurrency model of Java»

❖ Java Renaissance by Georges Saab

«No need to change paradigms, just make threads better»

	Kotlin: coroutines	Go: goroutines
Стек	stackless	stackful
Планирование	dispatcher	GMP-модель
Переключение	явные точки приостановки	в любом месте
Интуиция	функция, превращенная компилятором в конечный автомат	как поток, но только сильно дешевле

	Java: virtual threads	Go: goroutines
Стек	stackful	stackful
Планирование	MN-модель	GMP-модель
Переключение	в любом месте*	в любом месте
Интуиция	как поток, но только сильно дешевле	как поток, но только сильно дешевле



для написания
конкурентных
программ на Go
не обязательно
знать модель
памяти

The Go Memory Model

Version of June 6, 2022

Introduction

The Go memory model specifies the conditions under which reads of a variable in one goroutine can be guaranteed to observe values produced by writes to the same variable in a different goroutine.

Advice

Programs that modify data being simultaneously accessed by multiple goroutines must serialize such access.

To serialize access, protect the data with channel operations or other synchronization primitives such as those in the [sync](#) and [sync/atomic](#) packages.

If you must read the rest of this document to understand the behavior of your program, you are being too clever.

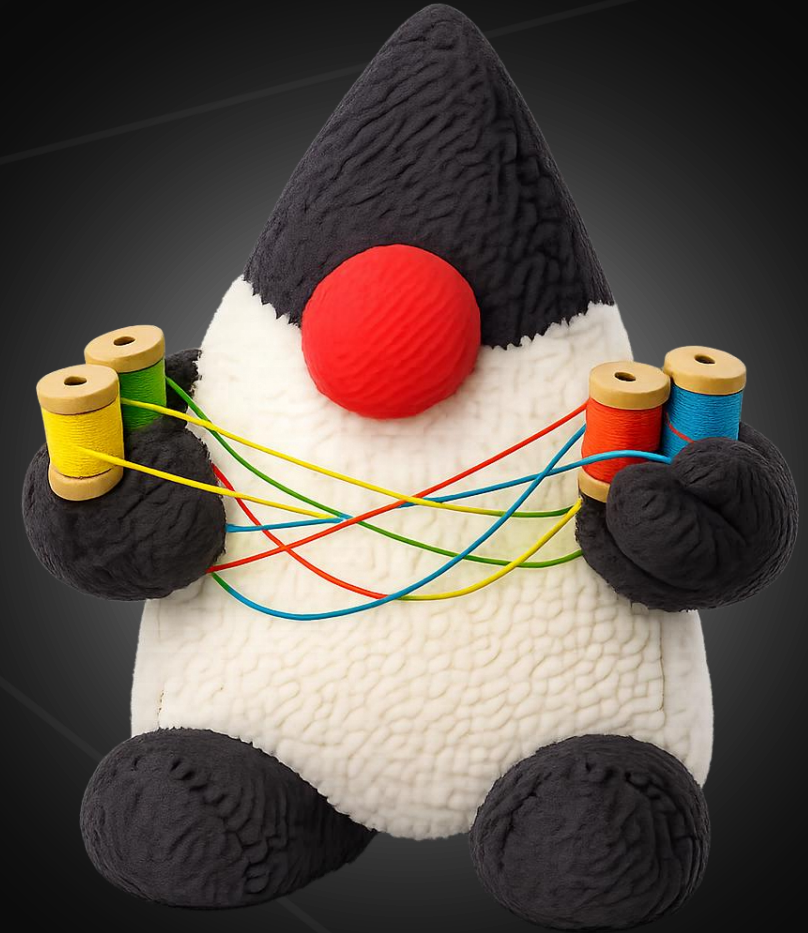
Don't be clever.

Project Loom: Fibers and Continuations for the Java Virtual Machine

«Fibers can provide a low-level primitive upon which interesting programming paradigms can be implemented, like channels, actors and dataflow...»

План

- ❏ ~~Зачем этот доклад~~
- ❏ Пререквизиты
- ❏ Акторы
- ❏ Каналы и CSP
- ❏ Паттерн Pipeline
- ❏ Итоги



Прerequisites

```
int x, y, z  
volatile boolean ready = false
```

Thread 1

```
x = 20  
y = 13  
z = 42  
ready = true
```

Thread 2

```
while(!ready)  
    Thread.onSpinWait()  
  
assert x == 20  
assert y == 13  
assert z == 42
```

hb

hb

hb

```
class Item { int x, y, z }  
final BlockingQueue<Item> queue = new LinkedBlockingQueue<>()
```

Thread 1

hb

```
var item = new Item()  
item.x = 20  
item.y = 13  
item.z = 42  
queue.add(item)
```

hb

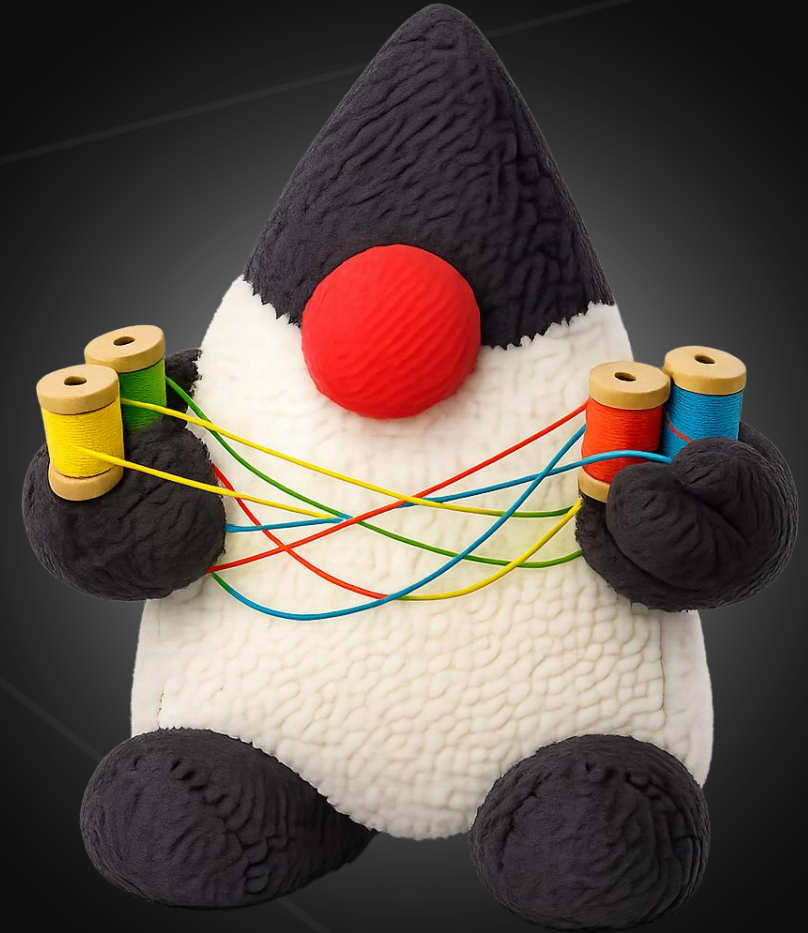
The diagram illustrates a concurrent execution scenario. Thread 1 (left) creates a new Item object with x=20, y=13, and z=42, and adds it to a shared queue. Thread 2 (right) takes an item from the queue and asserts that its values are x=20, y=13, and z=42. Red arrows and 'hb' labels indicate memory barriers. A red arrow points from the 'queue.add(item)' call in Thread 1 to the 'var item = queue.take()' call in Thread 2. Another red arrow points from the 'queue.add(item)' call in Thread 1 to the 'assert item.x == 20' line in Thread 2. A third red arrow points from the 'assert item.z == 42' line in Thread 2 to the right.

Thread 2

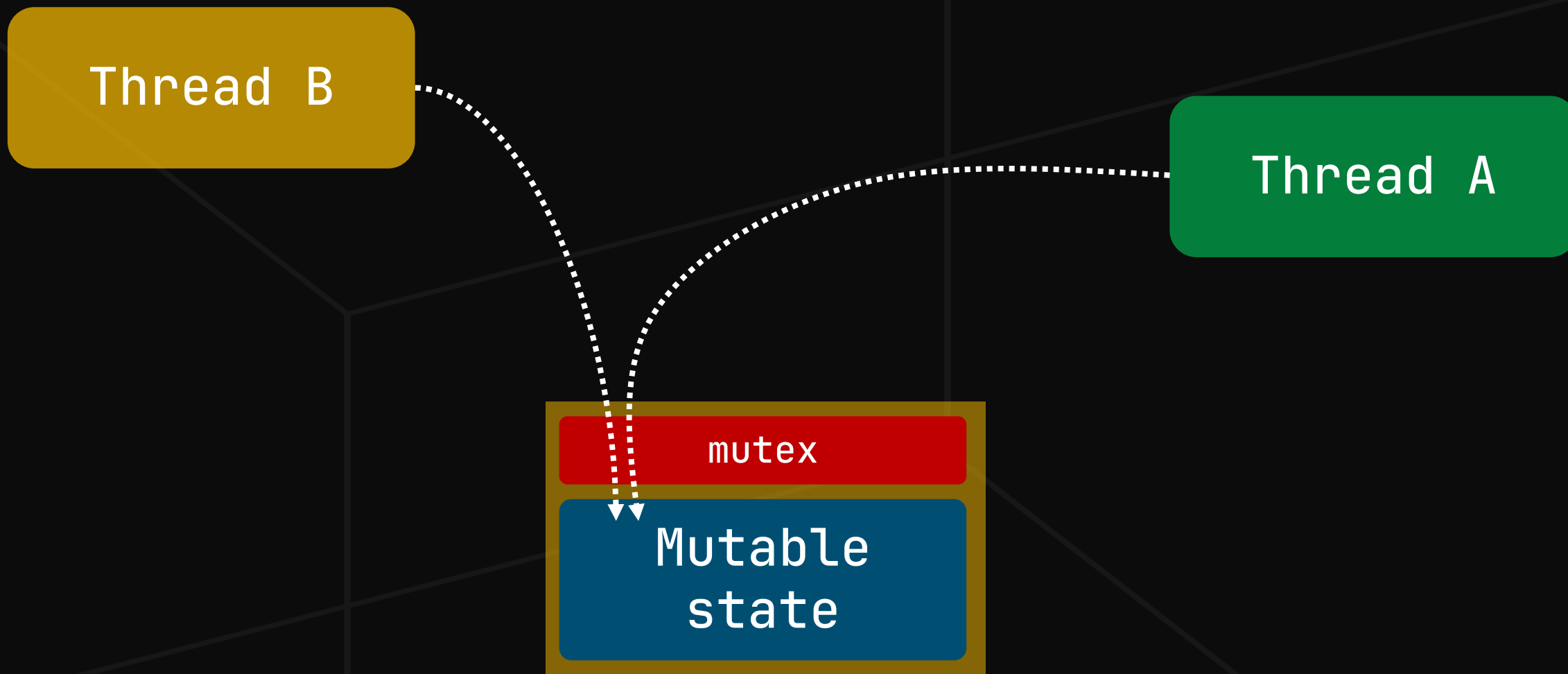
```
var item = queue.take()  
  
assert item.x == 20  
assert item.y == 13  
assert item.z == 42
```

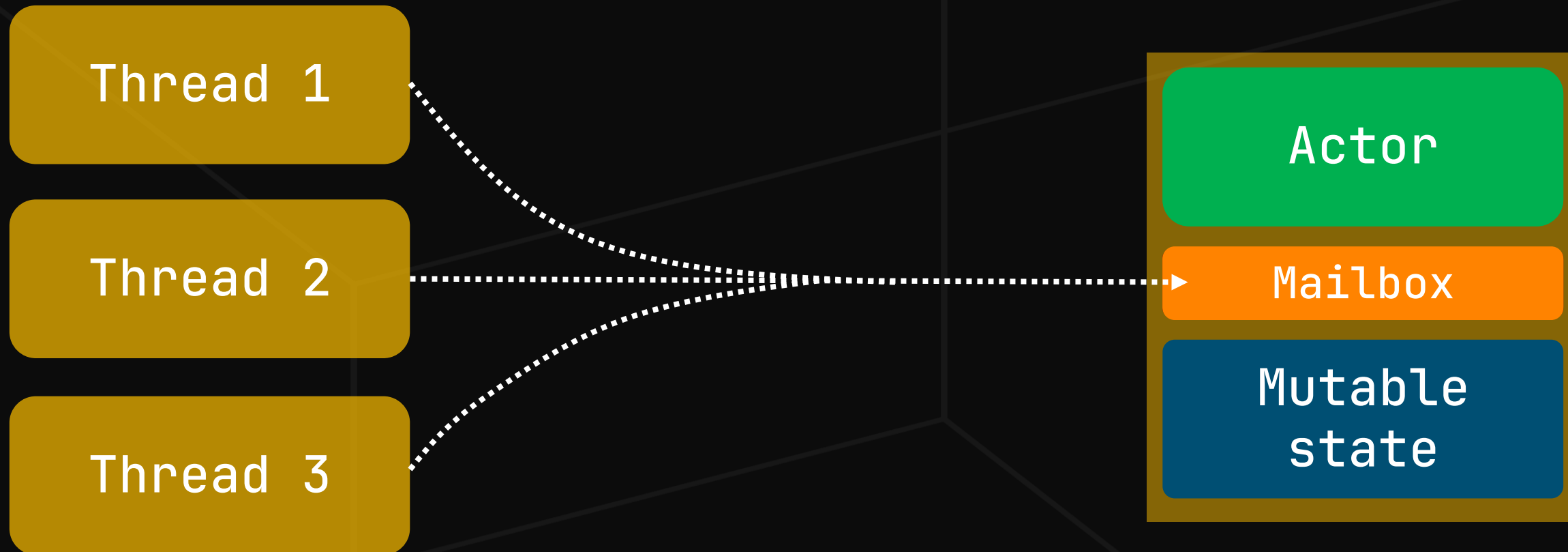
План

- ❏ ~~Зачем этот доклад~~
- ❏ Пререквизиты
- ❏ Акторы
- ❏ Каналы и CSP
- ❏ Паттерн Pipeline
- ❏ Итоги



Акторы





```
abstract class Actor<M> implements AutoCloseable {  
    private final BlockingQueue<M> mailbox = new LinkedBlockingQueue<>()  
    private final Thread worker = Thread.ofVirtual().unstarted(this::runLoop)
```

```
    public void start() { worker.start() }  
    public void close() { worker.interrupt() }
```

```
    protected abstract void onMessage(M message)  
    private void runLoop() {
```

```
        try {  
            while (!Thread.currentThread().isInterrupted()) {  
                M message = mailbox.take()  
                onMessage(message)  
            }  
        } catch (InterruptedException e) {  
            Thread.currentThread().interrupt()  
        }  
    }
```

```
}
```

```
}
```

логика обработки



*пример:
многопользовательский
чат с отдельными
комнатами*

Акторы: когда использовать?

хорошо

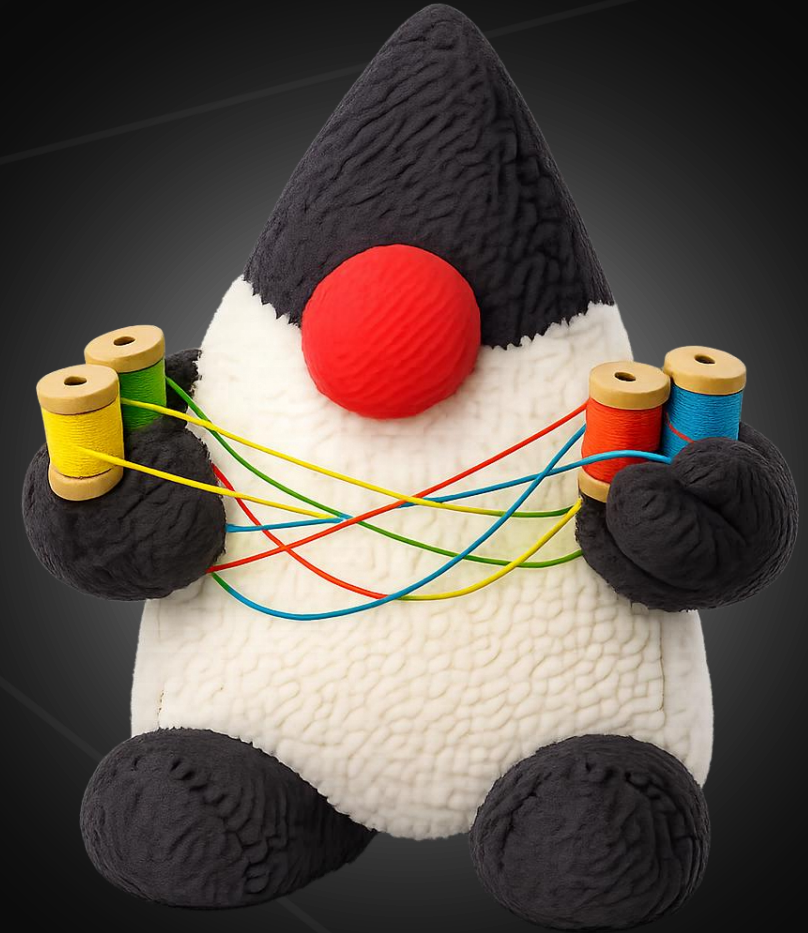
- ❏ Много независимых stateful-сущностей
- ❏ В рамках одной сущности обработка последовательная
- ❏ Взаимодействие выражается через сообщения

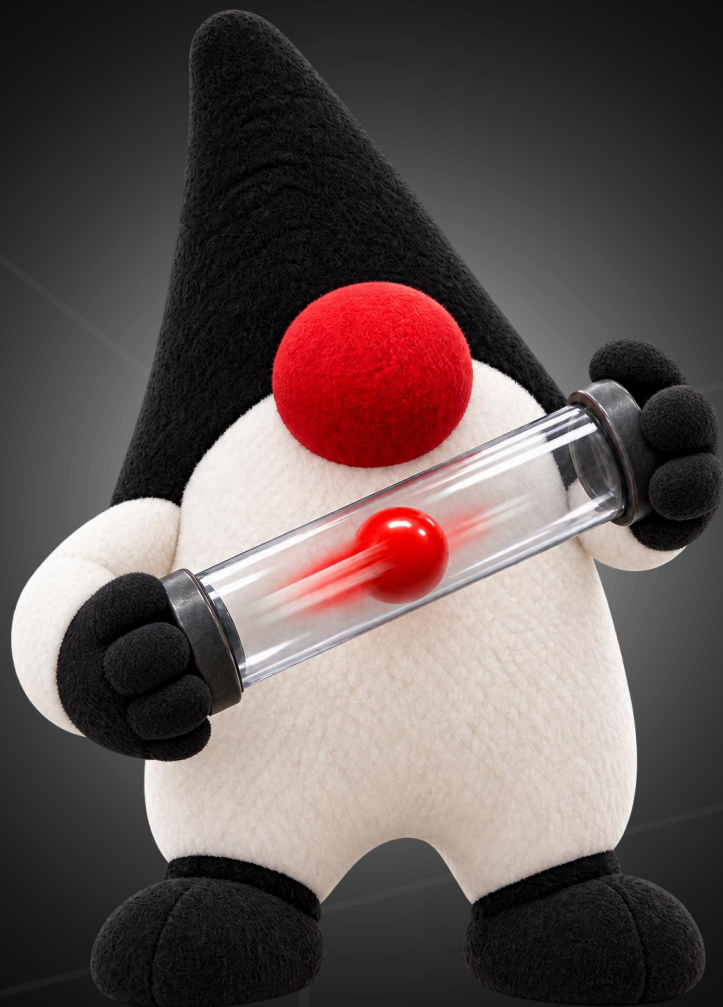
не очень

- ❏ CPU-bound параллельные задачи
- ❏ Общее глобальное состояние
- ❏ Слишком много сообщений в один актор

План

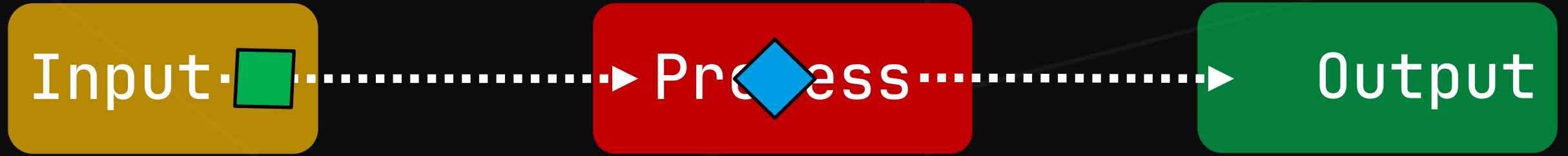
- ❏ ~~Зачем этот доклад~~
- ❏ Пререквизиты
- ❏ Акторы
- ❏ Каналы и CSP
- ❏ Паттерн Pipeline
- ❏ Итоги





Каналы и communicating sequential processes

CSP: process



- ✧ Получаем входное сообщение
- ✧ Обрабатываем
- ✧ Подаем результат на выход
- ✧ `Function<In, Out>` - это тоже процесс ?

CSP: sequential



- ❖ Процесс – виртуальный поток-обработчик
- ❖ Вычитывает входную очередь
- ❖ Обработывает и пишет в выходную очередь
- ❖ Обработка по FIFO

CSP: communicating



- ❖ Все взаимодействие между процессами – только через отправку сообщений

Упрощенная модель CSP

- ❏ Каждый процесс-обработчик взаимодействует только со своими очередями: входной и выходной
- ❏ Отправка-и-получение через очередь дает *happens-before*: процессам не надо думать об потокобезопасности
- ❏ Контракт: владение объектом после его передачи через очередь прекращается

```
abstract class Processor<In,Out> implements AutoCloseable {
```

```
    private final BlockingQueue<In> input
```

```
    private final BlockingQueue<Out> output
```

```
    private final Thread worker = Thread.ofVirtual().unstarted(this::runLoop)
```

```
    Processor(BlockingQueue<In> input, BlockingQueue<Out> output) { this.input = input; this.output = output }
```

```
    public void start() { worker.start() }
```

```
    public void close() { worker.interrupt() }
```

```
    protected abstract Out doProcessing(In message)
```

```
    private void runLoop() {
```

```
        try {
```

```
            while (!Thread.currentThread().isInterrupted()) {
```

```
                In in = input.take()
```

```
                Out out = doProcessing(in)
```

```
                output.put(out)
```

```
            }
```

```
        } catch (InterruptedException e) {
```

```
            Thread.currentThread().interrupt()
```

```
        }
```

```
    }
```

```
}
```

Мир Plat.Form

ЛОГИКА ОБРАБОТКИ

- ❏ Мы можем заблокироваться на очереди: `take()`
- ❏ Как понять, что входных данных больше нет?
- ❏ Как получать данные из нескольких очередей?
- ❏ Как сделать корректное завершение всех процессов?



**Для CSP
нужно что-то
лучше очереди
Но что?**

Канал (go-style)

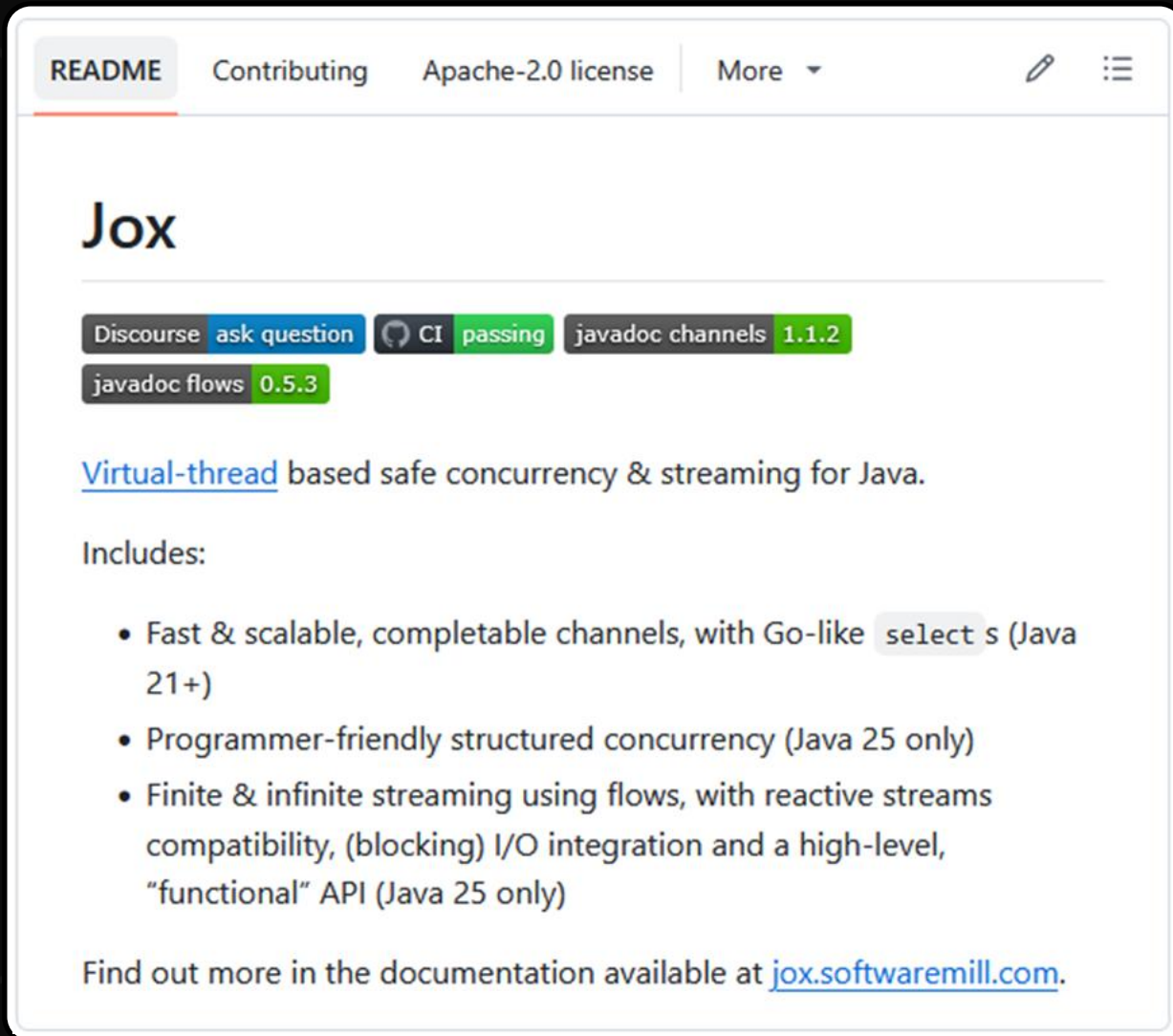
- ❖ Получение и отправка сообщения:
`send()` и `receive()` – блокирующие
- ❖ Заккрытие канала:
`close()` – только один раз
- ❖ Буферизированный: блокирование зависит от оставшейся емкости
- ❖ Небуферизированный: *rendezvous points*



```
select {  
    case msg := <-channelA:  
        fmt.Println("Received from channel A:", msg)  
    case msg := <-channelB:  
        fmt.Println("Received from channel B:", msg)  
}
```

❏ Но в Java нет ни каналов,
ни оператора `select`

Библиотека Jox



The screenshot shows the README page for the Jox library. At the top, there are navigation tabs: 'README' (selected), 'Contributing', 'Apache-2.0 license', and 'More'. Below the tabs is the 'Jox' logo. A row of status badges follows: 'Discourse ask question', 'CI passing', 'javadoc channels 1.1.2', and 'javadoc flows 0.5.3'. The main text describes Jox as 'Virtual-thread based safe concurrency & streaming for Java.' It lists features under the heading 'Includes:':

- Fast & scalable, completable channels, with Go-like `select s` (Java 21+)
- Programmer-friendly structured concurrency (Java 25 only)
- Finite & infinite streaming using flows, with reactive streams compatibility, (blocking) I/O integration and a high-level, "functional" API (Java 25 only)

At the bottom, it says 'Find out more in the documentation available at jox.softwaremill.com.'

- ❏ Без внешних зависимостей
- ❏ Java 21+
- ❏ Каналы – эффективная реализация из Kotlin
- ❏ Поддержка `select` в стиле Go

```
var ch = Channel.<Integer>newBufferedChannel(10)
```

```
ch.send(1)
```

```
System.out.println("Sent 1")
```

```
ch.send(2)
```

```
System.out.println("Sent 2")
```

```
ch.send(3)
```

```
System.out.println("Sent 3")
```

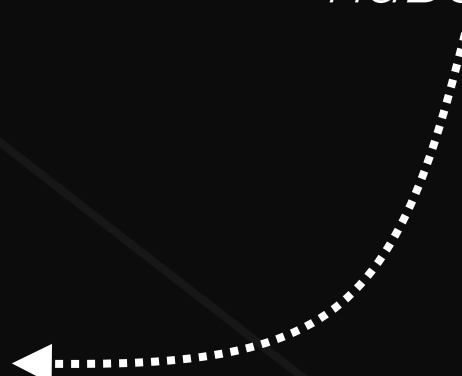
```
System.out.println("Received: " + ch.receive())
```

```
System.out.println("Received: " + ch.receive())
```

```
System.out.println("Received: " + ch.receive())
```

```
System.out.println("Received: " + ch.receive())
```

*заблокируется
навсегда*



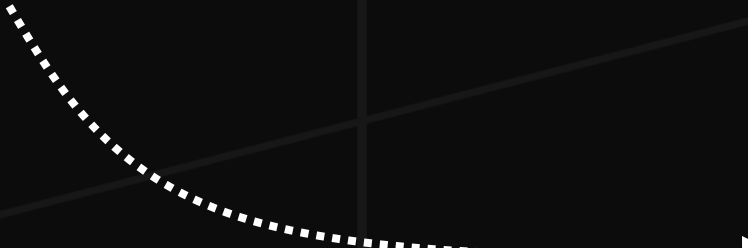
```
var ch1 = Channel.<Integer>newBufferedChannel(1)
var ch2 = Channel.<Integer>newBufferedChannel(1)
var ch3 = Channel.<Integer>newBufferedChannel(1)
```

```
ch2.send(29)
ch3.send(32)
```

```
for (int i = 0; i < 3; i++) {
    var received = select(
        ch1.receiveClause(),
        ch2.receiveClause(),
        ch3.receiveClause()
    )
```

```
    System.out.println(i + "; Received: " + received)
}
```

0; Received: 29
1; Received: 32
заблокируется



CSP: когда использовать?

хорошо

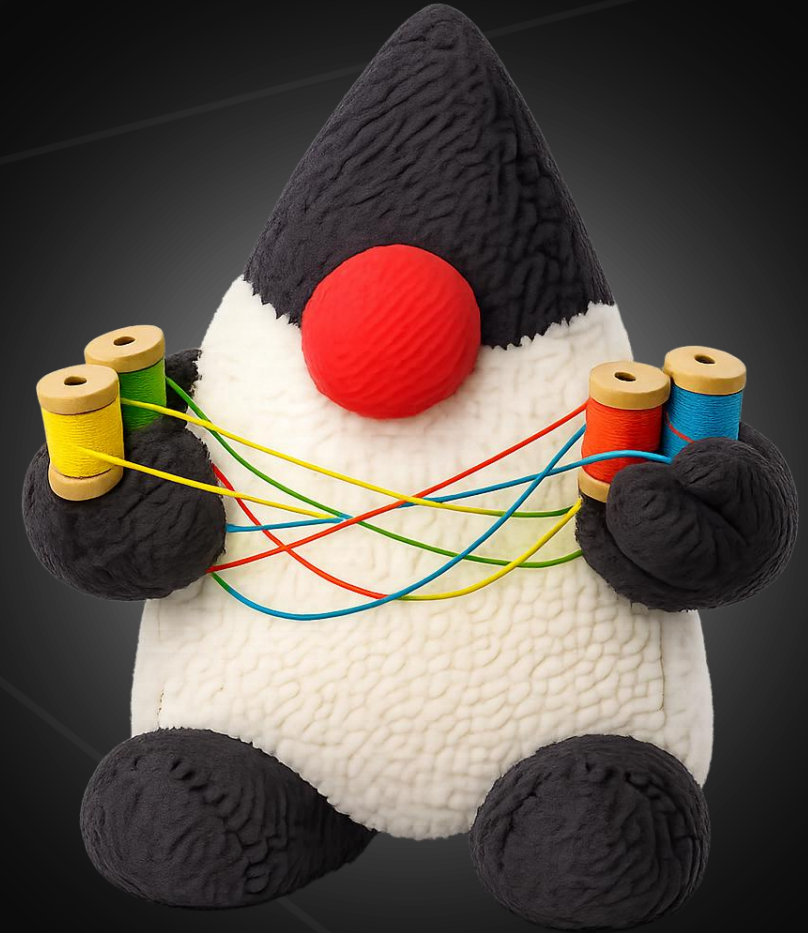
- ❖ Цепочки взаимодействующих процессов
- ❖ Компоненты слабо связаны
- ❖ Много процессов большую часть времени ждут

не очень

- ❖ CPU-bound параллельные задачи*
- ❖ Простой request-response
- ❖ Передача большого изменяемого состояния

План

- ❏ ~~Зачем этот доклад~~
- ❏ Пререквизиты
- ❏ Акторы
- ❏ ~~Каналы и CSP~~
- ❏ Паттерн Pipeline
- ❏ Итоги



Паттерн Pipeline



Обогащение

Валидация

Регистрация во
внешней системе

Сохранение в БД

Расчетная система платежной системы «Мир»

- ❖ Нужно обрабатывать много транзакций конкурентно и параллельно, максимизируем пропускную способность
- ❖ Было: каждая транзакция это отдельная цепочка из взаимодействующих akka-актеров
- ❖ Миграция с akka по нескольким причинам: лицензия, сложность поддержки - хотим решение проще

Вариант 1: VT на транзакцию

Обогащение

Валидация

Регистрация во
внешней системе

Сохранение в БД

- ❖ Для каждой транзакции создаем виртуальный поток-обработчик

Вариант 1: VT на транзакцию

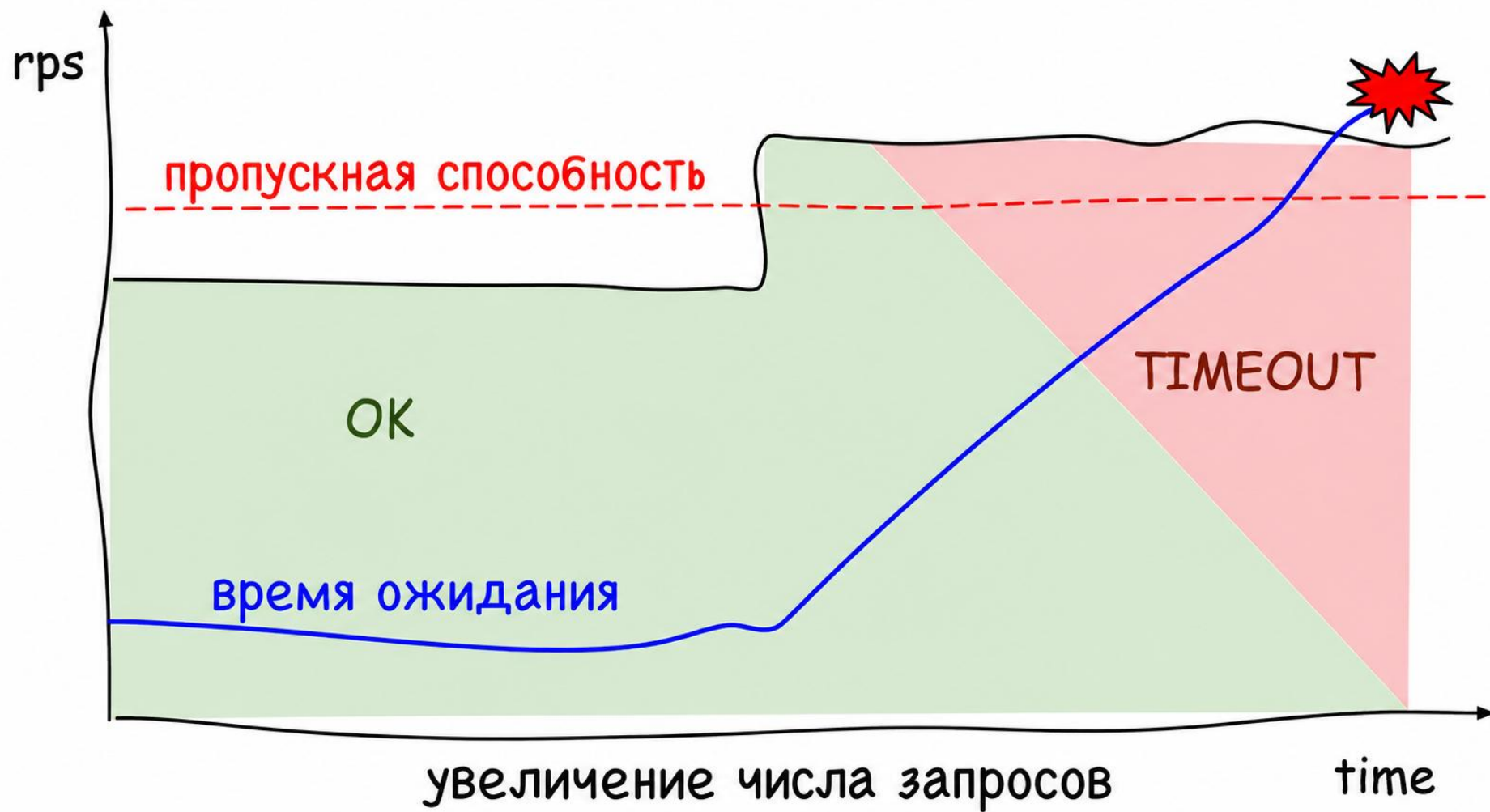


Для каждой транзакции создаем виртуальный поток-обработчик

Вирт. потоки дешевые – будет хорошо работать (?)

VT на транзакцию - особенности

- ❏ Все равно нужен какой-то ограничитель количества виртуальных потоков
- ❏ CPU-bound задачи: ограничены числом ядер в процессоре + отсутствие *preemption*
- ❏ IO-bound задачи: ограничены пропускной способностью внешней системы



<https://netflixtechblog.medium.com/performance-under-load-3e6fa9a60581>

VT на транзакцию - особенности

- ❏ Все равно нужен какой-то ограничитель количества виртуальных потоков
глобальный `java.util.concurrent.Semaphore`
- ❏ CPU-bound задачи: ограничены числом ядер в процессоре + отсутствие *preemption*
`java.util.concurrent.Semaphore` по количеству ядер
- ❏ IO-bound задачи: ограничены пропускной способностью внешней системы
`java.util.concurrent.Semaphore`, эмпирический подбор

Вариант 2: pipeline

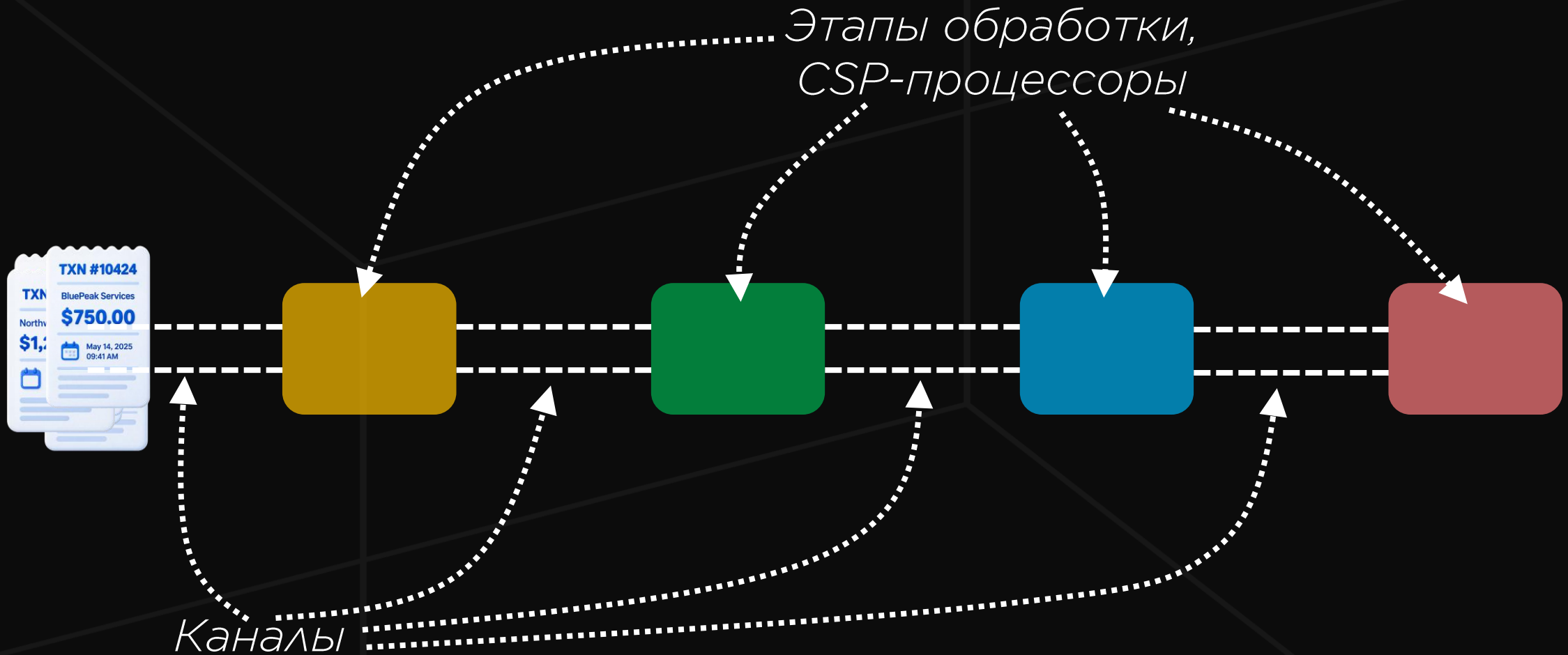
Обогащение

Валидация

Регистрация во
внешней системе

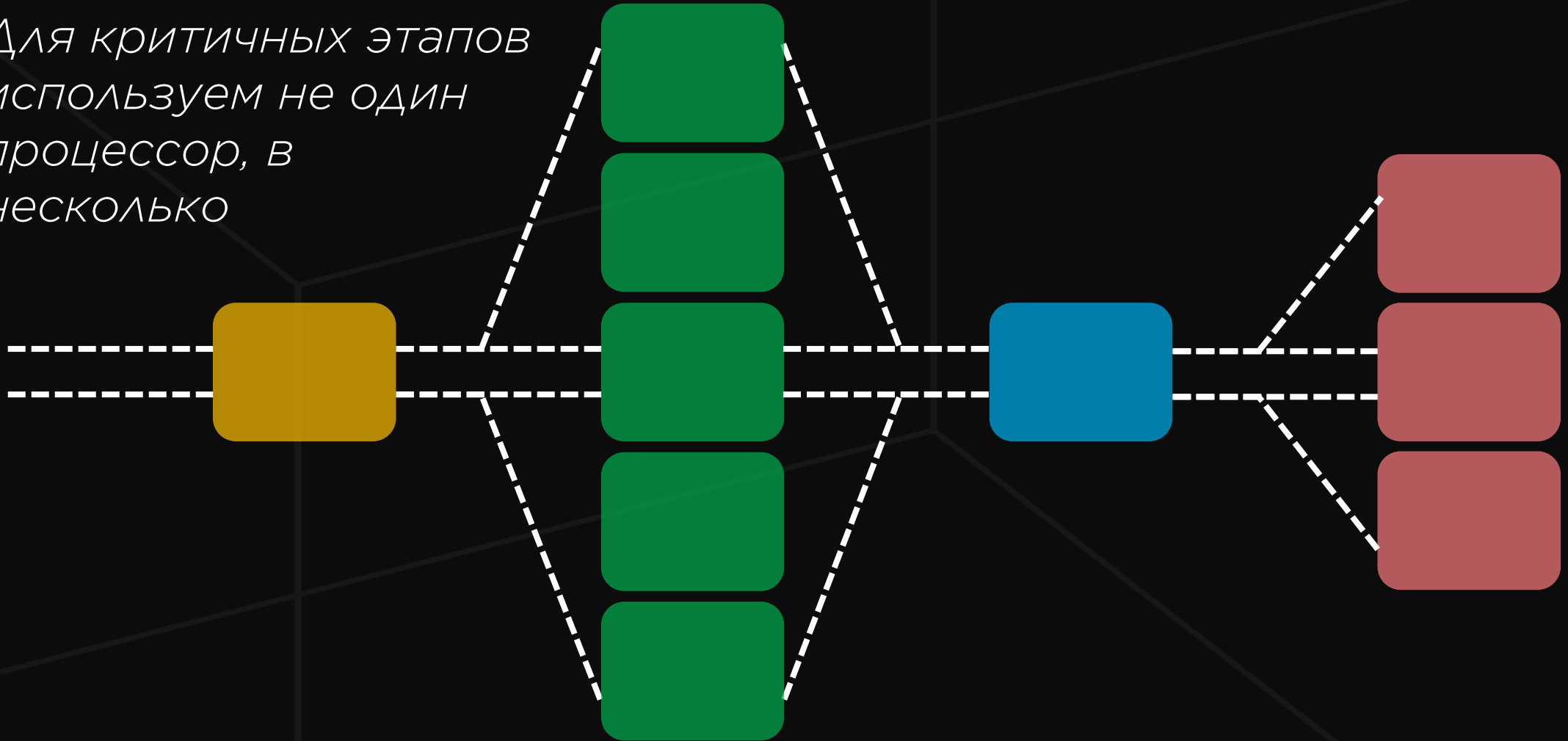
Сохранение в БД

Вариант 2: pipeline



Вариант 2: pipeline

*Для критичных этапов
используем не один
процессор, в
несколько*

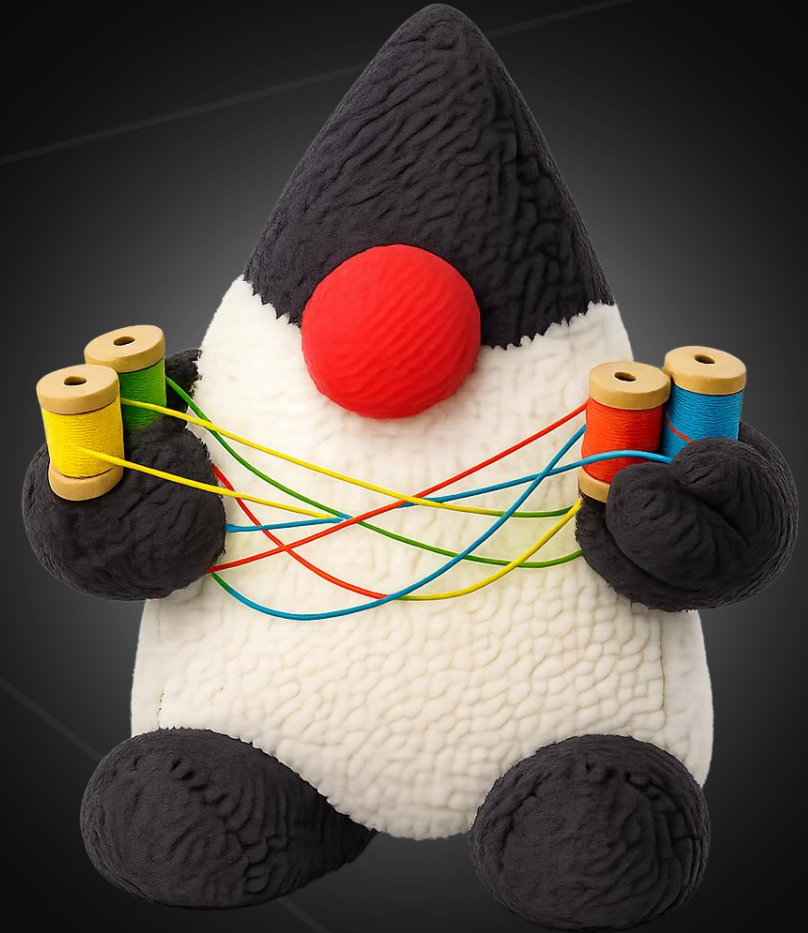


Pipeline - особенности

- ❖ Цепочка из связанных производителей-потребителей (этапов), каждый из которых работает конкурентно и параллельно друг с другом
- ❖ Используем CSP – не думаем про `memory model`
- ❖ Для критичных этапов используем пул обработчиков
- ❖ Естественный `backpressure` – если используем буферизированные каналы

План

- ❏ ~~Зачем этот доклад~~
- ❏ Пререквизиты
- ❏ Акторы
- ❏ ~~Каналы и CSP~~
- ❏ Паттерн Pipeline
- ❏ Итоги



Итоги

Классическая модель конкурентности не нужна?

Understanding Real-World Concurrency Bugs in Go

Tengfei Tu*

BUPT, Pennsylvania State University
tutengfei.kevin@bupt.edu.cn

Linhai Song

Pennsylvania State University
songlh@ist.psu.edu

Xiaoyu Liu

Purdue University
liu1962@purdue.edu

Yiying Zhang

Purdue University
yiying@purdue.edu

Abstract

Go is a statically-typed programming language that aims to provide a simple, efficient, and safe way to build multi-threaded software. Since its creation in 2009, Go has matured and gained significant adoption in production and open-source software. Go advocates for the usage of message passing as the means of inter-thread communication and provides several new concurrency mechanisms and libraries to ease multi-threading programming. It is important to understand the implication of these new proposals and the comparison of message passing and shared memory synchronization in terms of program errors, or bugs. Unfortunately, as far as we know, there has been no study on Go's concurrency bugs.

In this paper, we perform the first systematic study on concurrency bugs in real Go programs. We studied six popular Go software including Docker, Kubernetes, and gRPC. We analyzed 171 concurrency bugs in total, with more than half of them caused by non-traditional, Go-specific problems. Apart from root causes of these bugs, we also studied their fixes, performed experiments to reproduce them, and evaluated them with two publicly-available Go bug detectors. Overall, our study provides a better understanding on Go's concurrency models and can guide future researchers and practitioners in writing better, more reliable Go software and in developing debugging and diagnosis tools for Go.

CCS Concepts • Computing methodologies → Concurrent programming languages; • Software and its engineering → Software testing and debugging.

*The work was done when Tengfei Tu was a visiting student at Pennsylvania State University.

Keywords Go; Concurrency Bug; Bug Study

ACM Reference Format:

Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3297858.3304069>

1 Introduction

Go [20] is a statically typed language originally developed by Google in 2009. Over the past few years, it has quickly gained attraction and is now adopted by many types of software in real production. These Go applications range from libraries [19] and high-level software [26] to cloud infrastructure software like container systems [13, 36] and key-value databases [10, 15].

A major design goal of Go is to improve traditional multi-threaded programming languages and make concurrent programming easier and less error-prone. For this purpose, Go centers its multi-threading design around two principles: 1) making threads (called *goroutines*) lightweight and easy to create and 2) using explicit messaging (called *channel*) to communicate across threads. With these design principles, Go proposes not only a set of new primitives and new libraries but also new implementation of existing semantics.

It is crucial to understand how Go's new concurrency primitives and mechanisms impact concurrency bugs, the type of bugs that is the most difficult to debug and the most widely studied [40, 43, 45, 57, 61] in traditional multi-threaded programming languages. Unfortunately, there has been no prior work in studying Go concurrency bugs. As a result, to date, it is still unclear if these concurrency mechanisms actually make Go easier to program and less error-prone to concurrency bugs than traditional languages.



Количество багов примерно такое же, как для классической модели



Реже: datarace



Чаще: deadlock, goroutines leak

Итоги

- ❖ Виртуальные потоки дают возможность использовать альтернативные модели *еще* проще
- ❖ Каждая модель сфокусирована на своем аспекте: координация, явная защита состояния, сериализация доступа к состоянию
- ❖ Не верьте в серебряную пулю!
- ❖ Модель памяти знать все же стоит



Спасибо за внимание!