Modern JVM Multithreading

Paweł Jurczenko, 2020

モダン JVM マルチスレッディング

About me

- Senior Software Engineer
  at Allegro (~1000 microservices)
- 5 years of Scala development
- Distributed systems
- Concurrent computing
- Functional programming

Allegro 社勤務、Scala 5年目

Agenda

1. Overview
2. Threading models
3. Concurrency primitives
4. Non-blocking I/O
5. Thread pools
6. Best practices
7. Async stacktraces
8. Application architecture

スレッドから始めて、アプリ・アーキテクチャまでカバーする

Overview

Layered architecture

Custom threads

Green threads

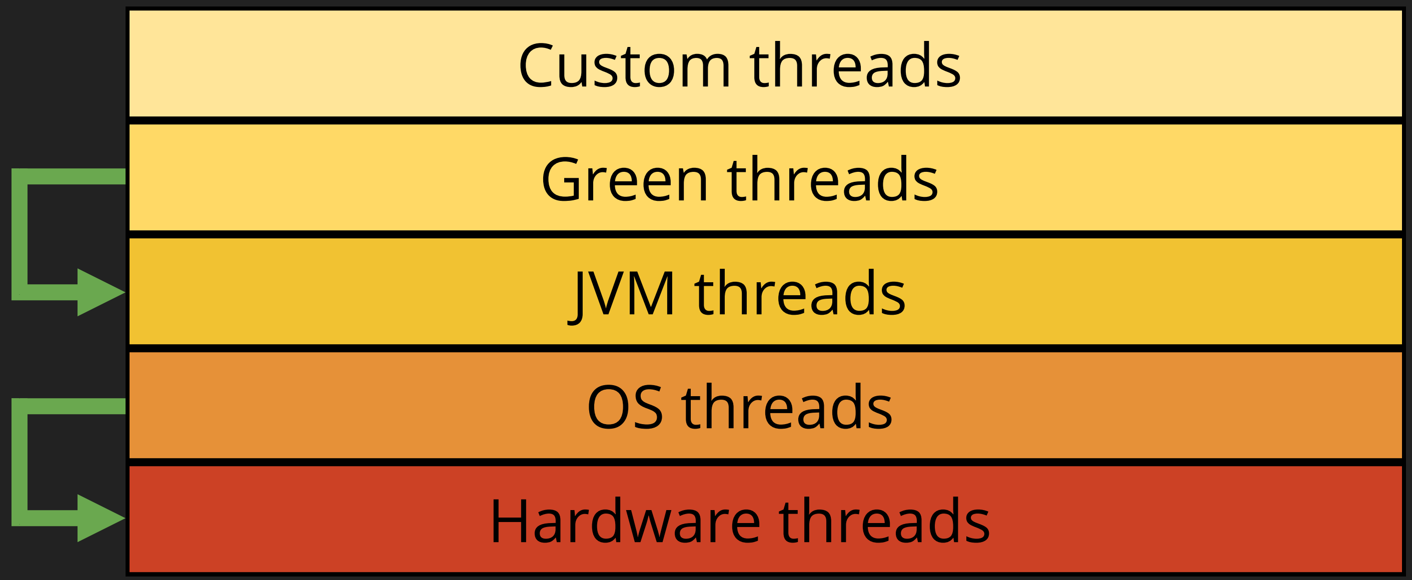
JVM threads

OS threads

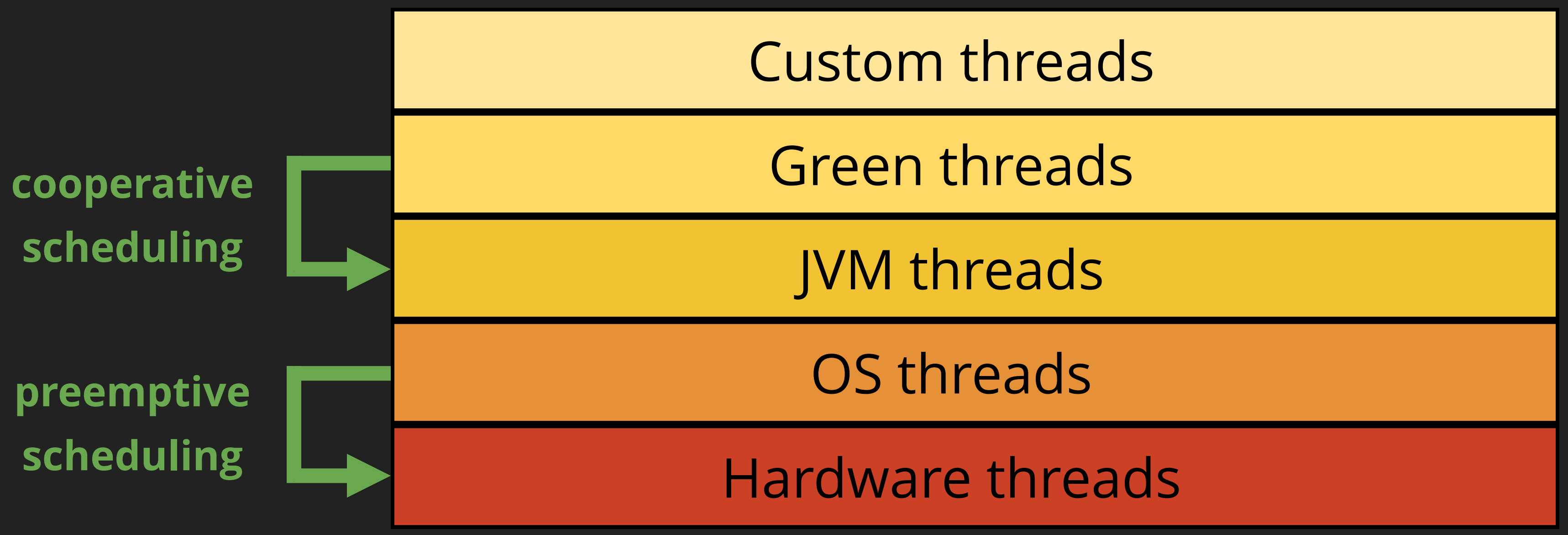
Hardware threads

レイヤード・アーキテクチャ

Layered architecture

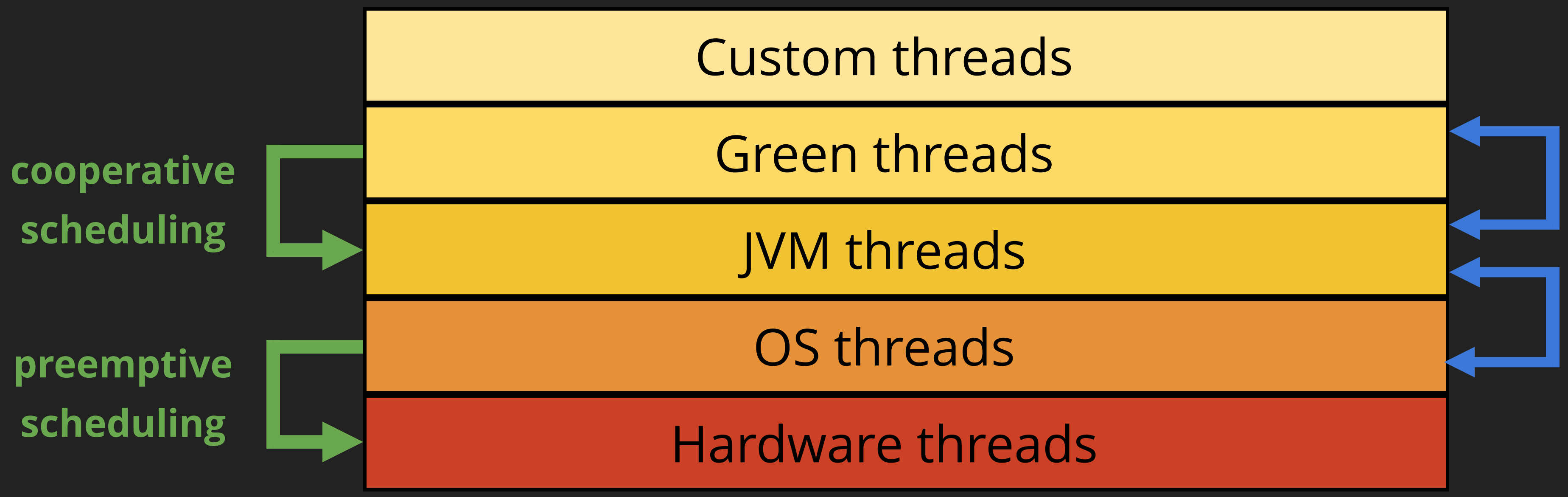


Layered architecture

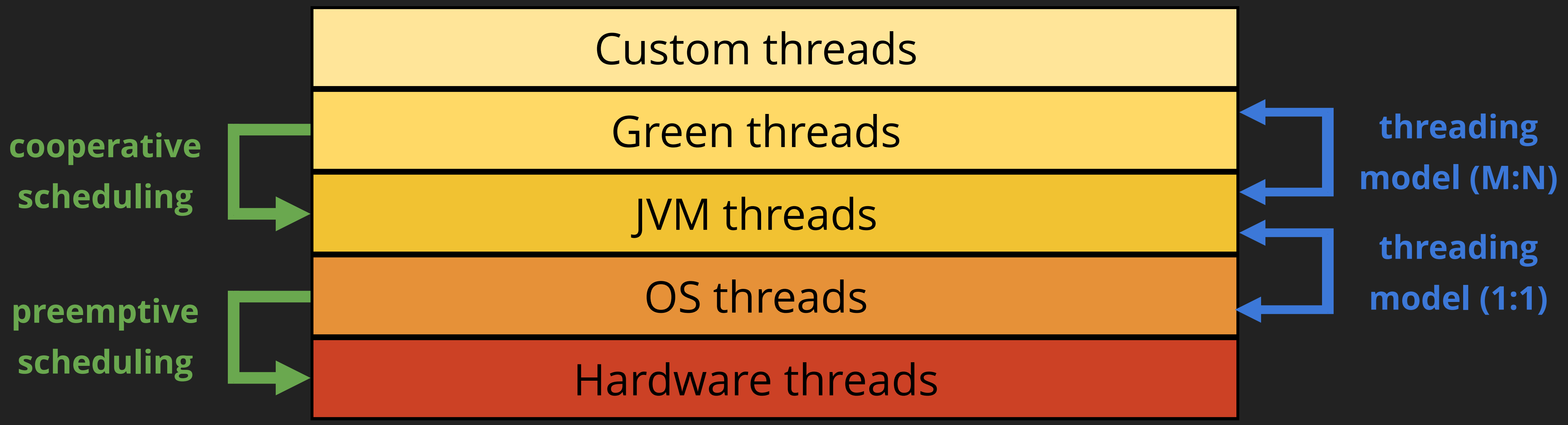


協調スケジューリング、プリエンプティブ・スケジューリング

Layered architecture



Layered architecture



多対多、1対1 のスレッド・モデリング

Green threads

グリーン・スレッド

Green threads

Overview

- Threads managed in user space instead of kernel space
- Scheduled by runtime library or virtual machine
- Usually scheduled cooperatively
- Examples: coroutines, goroutines, fibers

コルーチンなどユーザースペースで管理されるもの

Green threads

Goals

- Lower management costs
- More efficient resources usage
- Ability to block without blocking kernel threads
- Ability to be spawned in thousands
- Multithreading without native thread support

管理コストの低下、リソース利用の効率化がゴール
数1000個単位を作ることができる

Green threads

Implementations

- **Continuations / Coroutines** - *Project Loom, Kotlin Coroutines*
- **Fibers** - *Quasar, Monix, Cats Effect, ZIO*
- **Virtual threads** - *Project Loom*
- **Goroutines** - *Go runtime*
- **Haskell threads** - *GHC runtime*
- **Erlang processes** - *ERTS (Erlang RunTime System)*

グリーンスレッドの実装例は Project Loom、
ファイバーだと Monix、Cats Effect、ZIO など

Green threads

Project Loom: Introduction

- **Continuation**: a sequence of instructions that might be suspended and resumed
- **Virtual thread**: continuation + scheduler (e.g. ForkJoinPool)

Loom の紹介。継続と仮想スレッドから構成される。
継続： 一時停止したり、再開できる命令の列

Green threads

Project Loom

- Native JVM support for continuations
- Virtual threads built on top of continuations
- Virtual threads will be used in the code just as regular threads, but they'll have different runtime characteristics
- Thread blocking will be transparently replaced by virtual thread blocking
- Main challenge: managing call stacks independently of the kernel threads

Loom は JVM による継続をサポート
継続に基づいた仮想スレッド

Green threads

Project Loom - example

```java
// Thread.java
public static void sleep(long millis) throws InterruptedException {
    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }
    if (currentThread().isVirtual()) {
        long nanos = NANOSECONDS.convert(millis, MILLISECONDS);
        ((VirtualThread) currentThread()).sleepNanos(nanos);
    } else {
        sleep0(millis);
    }
}

private static native void sleep0(long millis) throws InterruptedException;
```

Concurrency primitives

並行プリミティブ：並行処理を構成する下位レベル機構

Concurrency primitives

Reborn

- **Locks**
- **Semaphores**
- **Channels**
- **Queues**
- **MVars**
- **STM**
- **Actors**

ロック、セマフォ、チャンネル、アクターなど

Concurrency primitives

Example

```
// java.util.concurrent.ArrayBlockingQueue
val queue = new ArrayBlockingQueue[String](capacity = 10)
queue.add("Allegro")   // throws "IllegalStateException" when queue is full
queue.offer("Allegro") // returns "false" when queue is full
queue.put("Allegro")   // blocks a thread when queue is full
```

Concurrency primitives

Example

```
// java.util.concurrent.ArrayBlockingQueue
val queue = new ArrayBlockingQueue[String](capacity = 10)
queue.add("Allegro")   // throws "IllegalStateException" when queue is full
queue.offer("Allegro") // returns "false" when queue is full
queue.put("Allegro")   // blocks a thread when queue is full

// monix.catnap.ConcurrentQueue
ConcurrentQueue[Task]
  .bounded[String](capacity = 10)
  .flatMap(queue => queue.offer("Allegro")) // returns "Task[Unit]"
```

Non-blocking I/O

ノンブロッキングI/O

Async vs Non-Blocking

Comparison

Asynchronous execution

Processing happens outside of the current control flow.

Non-blocking execution

Processing happens without blocking the current thread.

非同期： 現行制御フロー外での処理
ノンブロッキング： 現行スレッドをブロックしない処理

Blocking I/O

Risks of having too many threads

- Memory consumption
- Context switch overhead
- Decreased throughput
- Increased cache misses
- Increased number of GC roots
- Increased risk of deadlocks

大量のスレッドを作る弊害
メモリ量、コンテキスト切り替えオーバーヘッドなど

Non-blocking I/O

Common misconception

It's not about better I/O performance.

It's about more efficient resources usage.

I/O 性能の向上のためというのは誤解
そうではなく、リソース利用の効率化

Non-blocking I/O

System-level capabilities

- Linux: epoll, AIO, io_uring
- Windows: IOCP
- Mac OS: kqueue
- FreeBSD / NetBSD: kqueue
- Solaris: event ports

epoll などが OS レベルで提供される
ノンブロッキング機構の例

Non-blocking I/O

Network I/O

- Well supported by the operating systems
- Well supported by the JVM
- Examples: async-http-client, Spring WebClient, Java 11 HTTP Client

ネットワークI/O は OS や JVM でもサポートしている

Non-blocking I/O

File I/O

- Well supported only by some operating systems
- Not fully supported on Linux*
- JVM on Linux: non-blocking file I/O doesn't exist
- JVM on Linux: AsynchronousFileChannel is blocking
- Affects not only JVM: libuv has the exact same problems

* might change with **io_uring**

ノンブロッキングなファイルI/O は一部のOS のみ

Non-blocking I/O

Non-relational databases

- Well supported by non-relational databases
- MongoDB: Async Driver
- Cassandra: DataStax Java Driver
- Redis: Lettuce
- ...and many more!

NoSQL ではノンブロッキングI/O が充実している

Non-blocking I/O

Relational databases - problem

JDBC (Java Database Connectivity):

- Really old - February 19, 1997
- Completely blocking API
- Low-level, leaky abstractions
- Nulls, exceptions, side-effects

JDBC は古いし完全にブロッキング

Non-blocking I/O

Relational databases - solutions

Low-level:

- PostgreSQL: postgresql-async, jasync-sql
- MySQL: mysql-async, jasync-sql
- Generic: Loom-based JDBC 🚧

High-level:

- PostgreSQL: Quill, Skunk 🚧
- MySQL: Quill
- Generic: R2DBC

RDBMS 用に様々なソリューションが登場している

Non-blocking I/O

Relational databases - example

Goal:

We want to execute some application-level code each time a new offer is inserted into the database.

お題: `offer` がデータベースに挿入されるたびに
何らかのアプリレベルのコードを実行したい

Non-blocking I/O

Relational databases - example

```
CREATE FUNCTION offer_created() RETURNS trigger as $$
  BEGIN
    PERFORM pg_notify('offers', NEW.offer_id);
    RETURN NEW;
  END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER offer_created_trigger
AFTER INSERT ON offers
FOR EACH ROW EXECUTE PROCEDURE offer_created();
```

Non-blocking I/O

Relational databases - example

```scala
// Skunk
val session: Resource[IO, Session[IO]] =
  Session.single(
    host = "localhost",
    port = 5432,
    user = "postgres",
    database = "world"
  )

session.use { s =>
  val notifications = s
    .channel(id"offers")
    .listen(maxQueued = 10) // fs2.Stream[IO, Notification[String]]
    .toUnicastPublisher     // org.reactivestreams.Publisher[Notification[String]]
    .toObservable           // monix.reactive.Observable[Notification[String]]

  ...
}
```

Skunk を使った例

Non-blocking I/O

Relational databases - generic solutions

ADBA (Asynchronous Database Access API)

- Oracle initiative
- Announced in 2016
- Also known as "Asynchronous JDBC"
- Responses wrapped in ***CompletableFuture***
- No streaming/backpressure capabilities
- Not developed anymore: Loom-based JDBC will be used instead

R2DBC (Reactive Relational Database Connectivity)

- Spring (Pivotal) initiative
- Announced in 2018
- Responses wrapped in ***Mono/Flux***
- Compliant with Reactive Streams
- First released in November of 2019

汎用ソリューションとして 2つある
ADBA は現在開発中止中

Non-blocking I/O

R2DBC sneak peak

```
// Implicit
interface OfferRepository extends ReactiveCrudRepository<Offer, OfferId> {

  @Query("SELECT id, userId, categoryId
          FROM offers
          WHERE offers.userId = :userId")
  Flux<Offer> findByUserId(@Param("userId") Long userId);

}
```

R2DBC の先取り

Non-blocking I/O

R2DBC sneak peak

```
// Explicit
val client: DatabaseClient = DatabaseClient
  .create(postgresConnectionFactory);

val offers: Flux[Offer] = client
  .execute("""
      SELECT id, userId, categoryId
      FROM offers
      WHERE offers.userId = :userId"""
  )
  .bind("userId", userId)
  .as(classOf[Offer])
  .fetch()
  .all()
```

Thread pools

スレッドプール

Thread pools

Overview

Separate CPU-bound tasks from blocking I/O tasks.

Many applications will work fine with Scala's global thread pool.
However, when we have rigorous performance requirements,
it's good to have at least three different thread pools:

- one for CPU-bound tasks,
- one for blocking I/O tasks,
- one for non-blocking I/O tasks.

高速化のためには 3つのスレッドプールを作る
CPUバウンドとI/Oバウンドなタスクは分ける

Thread pools

Single thread pool

When the application uses a single thread pool:

You can use *ForkJoinPool*, which is a very good general-purpose thread pool.
It works well when you're mixing CPU-bound and IO-bound tasks.

ForkJoinPool は汎用性が高い
1つのスレッドプールを使うならこれ

Thread pools

Multiple thread pools

CPU-bound tasks

- many small tasks: beware thread contention (use e.g. *ForkJoinPool)*
- long-running tasks: use bounded pool (e.g. *newFixedThreadPool*)
- when in doubt: benchmark

Blocking I/O tasks

- use unbounded pool (e.g. *newCachedThreadPool*)
- provide limits at the higher, semantic level

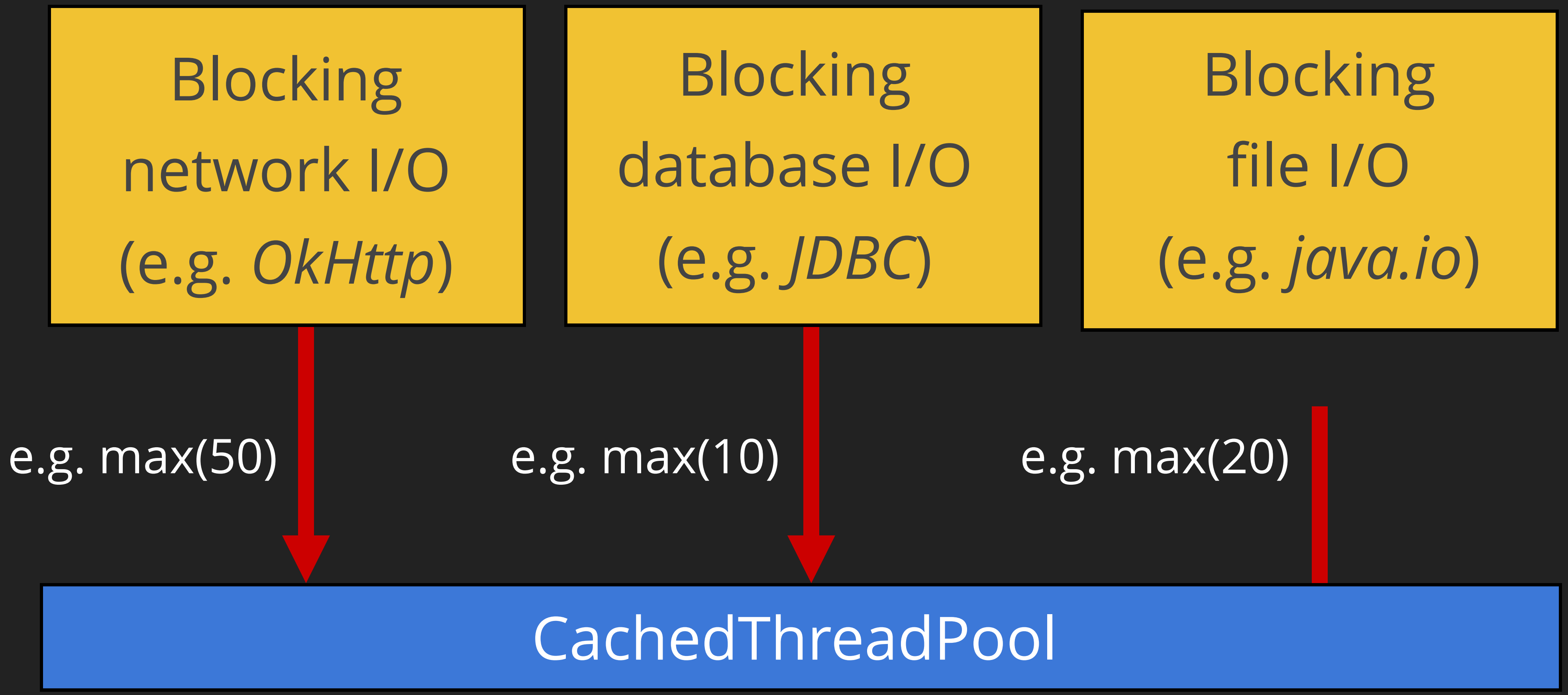
Non-blocking I/O tasks

- use bounded pool (e.g. *newFixedThreadPool*)
- one or two threads should be enough
- should work only as a dispatcher

複数のスレッドプールを作る場合のガイドライン
スレッドプールに上限を付けるか否かが重要

Thread pools

Unbounded pool for blocking I/O



ブロッキングIO 用の上限無しスレッドプール

Best practices

ベストプラクティス

Best practices

#1

Avoid concurrency for as long as possible.

可能な限り並行処理を避ける

Best practices

#2

Prefer high-level concurrency over low-level concurrency.

Times when you had to use *wait()* and *notify()* are long gone.

ロックなどの下位レベルのプリミティブよりも
上位レベルの並行機構を使えるか検討する

Best practices

#3

Choose concurrency primitives carefully.

There is a reason for the existence of so many of them:

each addresses a different problem.

並行プリミティブの選択には気をつける
やたらと数が多いのは用途が違うからだ

Best practices

#4

Know your thread pools.

Control your own thread pools and identify thread pools from external libraries.
Otherwise this might hit you at the worst time.

スレッドプールと仲良くなる
ライブラリが導入するスレッドプールにも注意

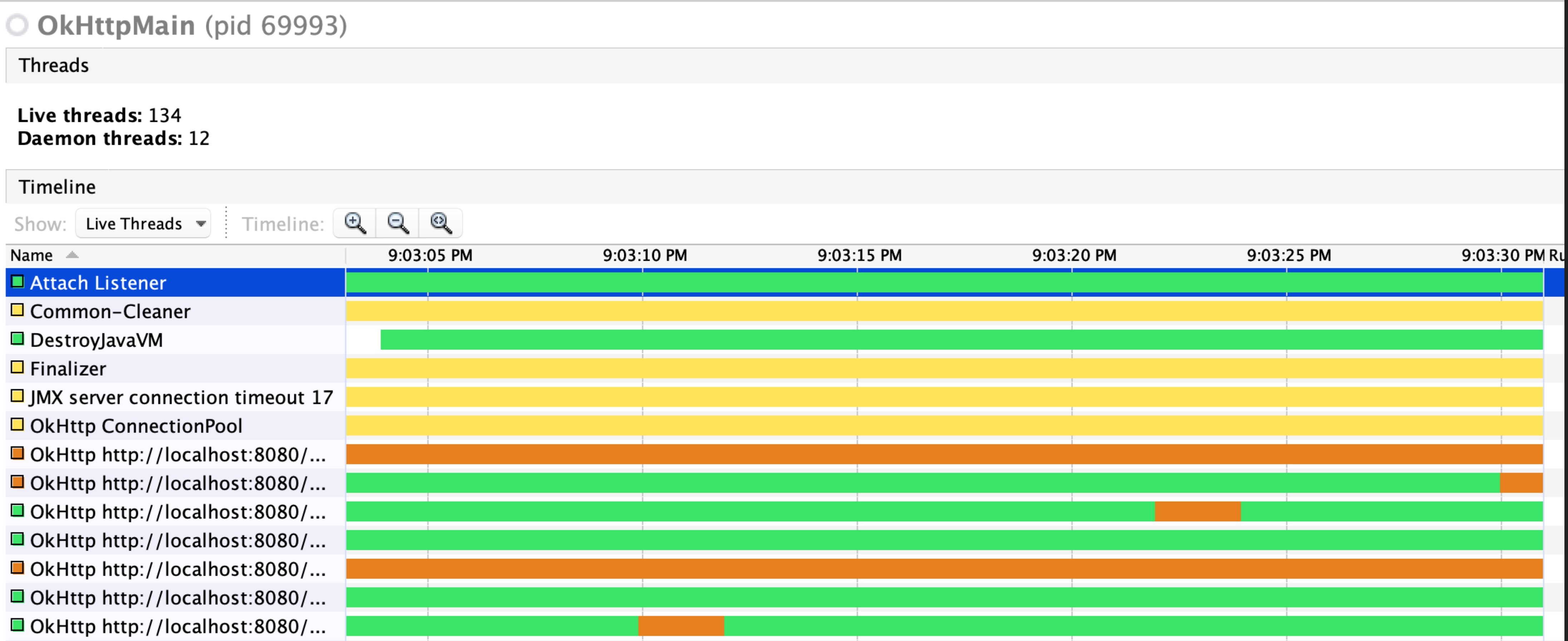
Best practices

#4 - example

```
// OkHttp
val dispatcher = new Dispatcher()
dispatcher.setMaxRequestsPerHost(100)
dispatcher.setMaxRequests(100)
val client = new OkHttpClient()
  .newBuilder()
  .dispatcher(dispatcher)
  .build()
```

Best practices

#4 - example



Best practices

#5

Prefer libraries with pluggable thread pools.

- you should be in the full control of your application's runtime, not the developers of external libraries you're using
- if thread pools aren't pluggable, make sure they are at least configurable

スレッドプールを自分で選べるライブラリを使う

Best practices

#5 - example

```
// OkHttp
val threadPool = Executors.newFixedThreadPool(100)
val dispatcher = new Dispatcher(threadPool)
dispatcher.setMaxRequestsPerHost(100)
dispatcher.setMaxRequests(100)
val client = new OkHttpClient()
  .newBuilder()
  .dispatcher(dispatcher)
  .build()
```

threadPool 変数に注目

Best practices

#5 - example

Best practices

#6

Decide whether you optimize for fairness, or throughput.

The reason is that it determines the type of thread pools,
and the number of their threads. Examples:

- *ForkJoinPool* is optimized for fairness
- *CPU-bound tasks*: exactly one thread per each CPU core is optimal

公平さとスループットのどちらを優先させるのかを決める

Best practices

#7

Be careful with *Runtime.getRuntime().availableProcessors()*

There are two reasons for that:

- it's not 100% reliable when it comes to virtualized environments (e.g. it might return **1** if you have **4** cores)
- even if **1** is the correct answer, it might lead to some trivial deadlocks

availableProcessor に気を付ける

Best practices

#7 - solution

```
// The exact minimum depends on your environment
val numCores = math.max(4, Runtime.getRuntime().availableProcessors())
```

Async stacktraces

非同期スタックトレース

Async stacktraces

Problem

```scala
object Main {
  def main(args: Array[String]): Unit =
    Await.result(Foo.foo, Duration.Inf)
}

object Foo {
  def foo: Future[Nothing] =
    Future(123).flatMap(_ => Bar.bar)
}

object Bar {
  def bar: Future[Nothing] =
    Future(throw new IllegalArgumentException("Test exception"))
}
```

Async stacktraces

Problem

```
java.lang.IllegalArgumentException: Test exception
  at Bar$.$anonfun$bar$1(Main.scala:17)
  at scala.concurrent.Future$.$anonfun$apply$1(Future.scala:671)
  at scala.concurrent.impl.Promise$Transformation.run(Promise.scala:430)
  at scala.concurrent.BatchingExecutor$AbstractBatch.runN(BatchingExecutor.scala:134)
  at scala.concurrent.BatchingExecutor$AsyncBatch.apply(BatchingExecutor.scala:163)
  at scala.concurrent.BatchingExecutor$AsyncBatch.apply(BatchingExecutor.scala:146)
  at scala.concurrent.BlockContext$.usingBlockContext(BlockContext.scala:107)
  at scala.concurrent.BatchingExecutor$AsyncBatch.run(BatchingExecutor.scala:154)
  at java.base/java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec(ForkJoinTask.java:1426)
  at java.base/java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:290)
  at java.base/java.util.concurrent.ForkJoinPool$WorkQueue.topLevelExec(ForkJoinPool.java:1020)
  at java.base/java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:1656)
  at java.base/java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1594)
  at java.base/java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:177)
```

どの `Future` が何を呼んだのかが分からない

Async stacktraces

Solution attempts:

- Reactor's onOperatorDebug()
- Reactor Debug Agent
- Kotlin Coroutines (e.g. Debug Mode)
- IntelliJ Async Stack Traces
- ZIO
- Cats Effect
- Monix 🚧

様々な試みがある

Async stacktraces

Example

```scala
// Cats Effect
object Main {
  def main(args: Array[String]): Unit =
    Foo.foo.unsafeRunSync()
}

object Foo {
  def foo: IO[Nothing] =
    IO(123).flatMap(_ => Bar.bar)
}

object Bar {
  def bar: IO[Nothing] =
    IO(throw new IllegalArgumentException("Test exception"))
}
```

Async stacktraces

Example

```
// Cats Effect 2.1.x
java.lang.IllegalArgumentException: Test exception
  at Bar$.$anonfun$bar$1(Main.scala:15)
  at cats.effect.internals.IORunLoop$.step(IORunLoop.scala:235)
  at cats.effect.IO.unsafeRunTimed(IO.scala:338)
  at cats.effect.IO.unsafeRunSync(IO.scala:256)
  at Main$.main(Main.scala:12)
  at Main.main(Main.scala)
  at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(...)
  at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(...)
  at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(...)
  at java.base/java.lang.reflect.Method.invoke(Method.java:566)
```

Async stacktraces

Example

```
// Cats Effect 2.1.x
java.lang.IllegalArgumentException: Test exception
  at Bar$.$anonfun$bar$1(Main.scala:15)
  at cats.effect.internals.IORunLoop$.step(IORunLoop.scala:235)
  at cats.effect.IO.unsafeRunTimed(IO.scala:338)
  at cats.effect.IO.unsafeRunSync(IO.scala:256)
  at Main$.main(Main.scala:12)
  at Main.main(Main.scala)
  at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(...)
  at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(...)
  at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(...)
  at java.base/java.lang.reflect.Method.invoke(Method.java:566)

// Cats Effect 2.2.x
java.lang.IllegalArgumentException: Test exception
  at Bar$.$anonfun$bar$1(Main.scala:15)
  at flatMap @ Foo$.foo(Main.scala:10)
```

Application architecture

アプリケーション・アーキテクチャ

Application structure

Common approaches when it comes to structuring asynchronous Scala applications:

- Futures
- Akka Actors
- IO Monads (Task, IO, ZIO)
- Free monads
- Tagless-final encoding
- Other

非同期アプリを組むときの一般的な方法

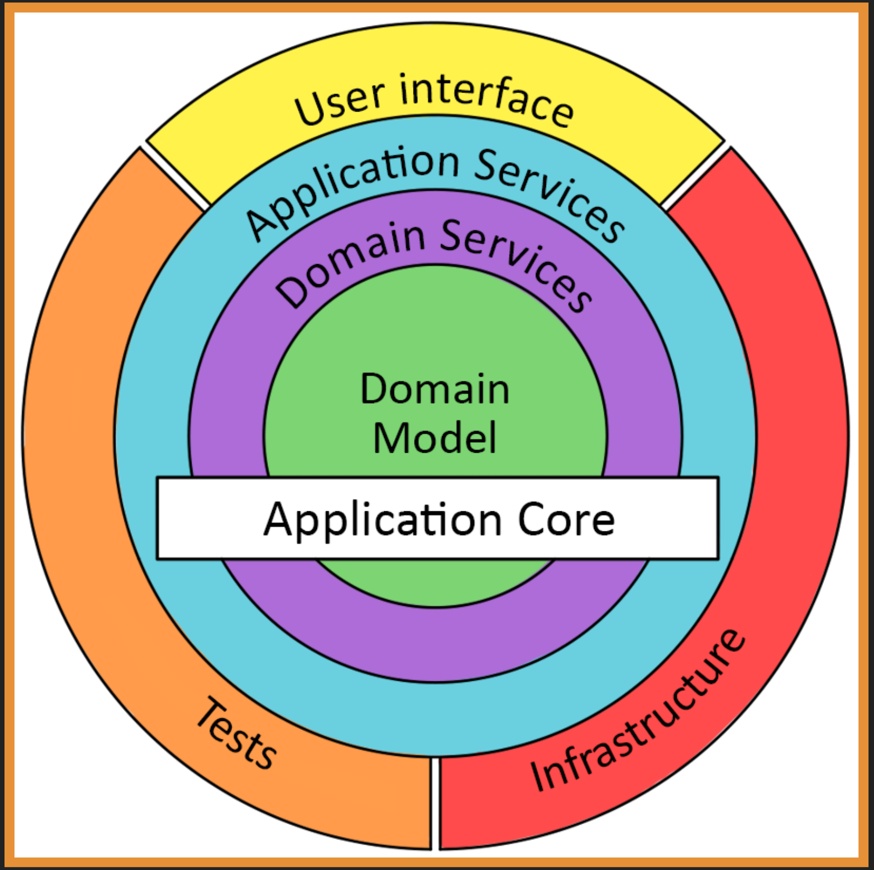
Application architecture

General approaches

- *Hexagonal architecture*
- *Ports and adapters*
- *Clean architecture*
- *Onion architecture*

アーキテクチャは高レベルでの方法論

Application architecture



Source: https://dzone.com/articles/onion-architecture-is-interesting

Application architecture

Common denominator:

Dependency Inversion Principle *(SOLID)*

依存性逆転の原則が共通項

Application architecture

Dependency Inversion Principle

1. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

2. Abstractions should not depend upon details. Details should depend upon abstractions.

~ Robert C. Martin, *C++ Report*, May 1996

上位レベルモジュールを下位レベルモジュールに依存させない

Application architecture

Approach #1 - concrete data types

```scala
// Domain
trait OfferRepository {
  def insert(offer: Offer): Task[Unit]
}

trait OfferEventPublisher {
  def publish(offerCreated: OfferCreated): Task[Unit]
}

class OfferService(
  repository: OfferRepository,
  publisher: OfferEventPublisher
) {
  def insertAndNotify(offer: Offer): Task[Unit] =
    for {
      _ <- repository.insert(offer)
      _ <- publisher.publish(OfferCreated(offer))
    } yield ()
}
```

具象データ型

Application architecture

Approach #1 - concrete data types

```scala
// Domain
trait OfferRepository {
  def insert(offer: Offer): Task[Unit]
}

trait OfferEventPublisher {
  def publish(offerCreated: OfferCreated): Task[Unit]
}

class OfferService(
  repository: OfferRepository,
  publisher: OfferEventPublisher
) {
  def insertAndNotify(offer: Offer): Task[Unit] =
    for {
      _ <- repository.insert(offer)
      _ <- publisher.publish(OfferCreated(offer))
    } yield ()
}
```

具象データ型

Application architecture

Approach #1 - concrete data types

```scala
// Domain
trait OfferRepository {
  def insert(offer: Offer): Task[Unit]
}

// Infrastructure
class PostgresOfferRepository(postgresConfig: PostgresConfig)
    extends OfferRepository {
  override def insert(offer: Offer): Task[Unit] = ???
}

// Tests
class InMemoryOfferRepository extends OfferRepository {
  private val repository = TrieMap.empty[OfferId, Offer]

  override def insert(offer: Offer): Task[Unit] =
    Task(repository.putIfAbsent(offer.id, offer))
}
```

具象データ型

Application architecture

Approach #1 - concrete data types

```scala
// Domain
trait OfferRepository {
  def insert(offer: Offer): Task[Unit]
}

// Infrastructure
class PostgresOfferRepository(postgresConfig: PostgresConfig)
    extends OfferRepository {
  override def insert(offer: Offer): Task[Unit] = ???
}

// Tests
class InMemoryOfferRepository extends OfferRepository {
  private val repository = TrieMap.empty[OfferId, Offer]

  override def insert(offer: Offer): Task[Unit] =
    Task(repository.putIfAbsent(offer.id, offer))
}
```

具象データ型

Application architecture

Approach #1 - problem

```scala
// Domain
trait OfferRepository {
  def insert(offer: Offer): Task[Unit]
}

trait OfferEventPublisher {
  def publish(offerCreated: OfferCreated): Task[Unit]
}

class OfferService(
  repository: OfferRepository,
  publisher: OfferEventPublisher
) {
  def insertAndNotify(offer: Offer): Task[Unit] =
    for {
      _ <- repository.insert(offer)
      _ <- publisher.publish(OfferCreated(offer))
    } yield ()
}
```

`Task` という実装レベルの責務が漏れている

Application architecture

Approach #2 - Tagless-final encoding

```scala
// Domain
trait OfferRepository[F[_]] {
  def insert(offer: Offer): F[Unit]
}

trait OfferEventPublisher[F[_]] {
  def publish(offerCreated: OfferCreated): F[Unit]
}

class OfferService[F[_]: Monad](
  repository: OfferRepository[F],
  publisher: OfferEventPublisher[F]
) {
  def insertAndNotify(offer: Offer): F[Unit] =
    for {
      _ <- repository.insert(offer)
      _ <- publisher.publish(OfferCreated(offer))
    } yield ()
}
```

tagless-final エンコーディングを使ってみる

Application architecture

Approach #2 - Tagless-final encoding

```scala
// Domain
trait OfferRepository[F[_]] {
  def insert(offer: Offer): F[Unit]
}

trait OfferEventPublisher[F[_]] {
  def publish(offerCreated: OfferCreated): F[Unit]
}

class OfferService[F[_]: Monad](
  repository: OfferRepository[F],
  publisher: OfferEventPublisher[F]
) {
  def insertAndNotify(offer: Offer): F[Unit] =
    for {
      _ <- repository.insert(offer)
      _ <- publisher.publish(OfferCreated(offer))
    } yield ()
}
```

tagless-final エンコーディングを使ってみる

Application architecture

Approach #2 - Tagless-final encoding

```scala
// Domain
trait OfferRepository[F[_]] {
  def insert(offer: Offer): F[Unit]
}

// Infrastructure
class PostgresOfferRepository[F[_]: ???](
  postgresConfig: PostgresConfig
) extends OfferRepository[F] {
    override def insert(offer: Offer): F[Unit] = ???
}

// Tests
class InMemoryOfferRepository extends OfferRepository[Task] {
  private val repository = TrieMap.empty[OfferId, Offer]

  override def insert(offer: Offer): Task[Unit] =
    Task(repository.putIfAbsent(offer.id, offer))
}
```

抽象化された F[_] という形の型が返る

Application architecture

Approach #2 - Tagless-final encoding

```scala
// Domain
trait OfferRepository[F[_]] {
  def insert(offer: Offer): F[Unit]
}

// Infrastructure
class PostgresOfferRepository[F[_]: ???](
  postgresConfig: PostgresConfig
) extends OfferRepository[F] {
    override def insert(offer: Offer): F[Unit] = ???
}

// Tests
class InMemoryOfferRepository extends OfferRepository[Task] {
  private val repository = TrieMap.empty[OfferId, Offer]

  override def insert(offer: Offer): Task[Unit] =
    Task(repository.putIfAbsent(offer.id, offer))
}
```

抽象化された `F[_]` という形の型が返る

Application architecture

Approach #2 - problem

```
// Domain
trait OfferRepository[F[_]] {
  def insert(offer: Offer): F[Unit]
}

// Infrastructure
class PostgresOfferRepository[F[_]: ???](
  postgresConfig: PostgresConfig
) extends OfferRepository[F] {
    override def insert(offer: Offer): F[Unit] = ???
}

// Tests
class InMemoryOfferRepository extends OfferRepository[Task] {
  private val repository = TrieMap.empty[OfferId, Offer]

  override def insert(offer: Offer): Task[Unit] =
    Task(repository.putIfAbsent(offer.id, offer))
}
```

Application architecture

Approach #3 - Hybrid solution

```scala
// Domain
trait OfferRepository[F[_]] {
  def insert(offer: Offer): F[Unit]
}

trait OfferEventPublisher[F[_]] {
  def publish(offerCreated: OfferCreated): F[Unit]
}

class OfferService[F[_]: Monad](
  repository: OfferRepository[F],
  publisher: OfferEventPublisher[F]
) {
  def insertAndNotify(offer: Offer): F[Unit] =
    for {
      _ <- repository.insert(offer)
      _ <- publisher.publish(OfferCreated(offer))
    } yield ()
}
```

Application architecture

Approach #3 - Hybrid solution

```scala
// Domain
trait OfferRepository[F[_]] {
  def insert(offer: Offer): F[Unit]
}

// Infrastructure
class PostgresOfferRepository(postgresConfig: PostgresConfig)
    extends OfferRepository[Task] {
  override def insert(offer: Offer): Task[Unit] = ???
}

// Tests
class InMemoryOfferRepository extends OfferRepository[Task] {
  private val repository = TrieMap.empty[OfferId, Offer]

  override def insert(offer: Offer): Task[Unit] =
    Task(repository.putIfAbsent(offer.id, offer))
}
```

Summary

Contact

- E-mail: **pawel.jurczenko@gmail.com**
- Twitter: **@pawel_jurczenko**