Direct-style Scala using Ox

Adam Warski, June 2024
@adamwarski / @softwaremill.social / softwaremill.com

SOFTWAREMILL

Ox を用いた Direct-Style Scala

Safe direct style concurrency and resiliency for Scala on the JVM

JVM上のScalaのための安全な、Direct-styleな並行性と耐障害性

Safe direct style concurrency and resiliency for Scala on the JVM

- Using static typing
- Providing APIs for safe concurrency & resources
- Proposing an approach to error handling

静的型付けの使用・安全な同時実行とリソースのためのAPIの提供・エラー処理へのアプローチの提案

Safe direct style concurrency and resiliency for Scala on the JVM

- An approach to programming which leverages the built-in control flow constructs of the language as the basic building blocks of effectful code.
- Enabling teams to deliver working software quickly and with confidence
- Other projects: sttp client, tapir, gears, many Scala & Java libraries

言語に組み込まれた制御フローを活用することでチームでの効率的な開発を可能にする

Safe direct style concurrency and resiliency for Scala on the JVM

- High-level operators: avoiding concurrency
- Safe low-level building blocks
- Channels for communication (as in Go)
- Retries, rate limiters, ...

高レベル演算子による明示的な並行プログラムの回避・安全な低レベル部品・Goのような通信チャネル、再試行やレートリミッタを提供

Safe direct style concurrency and resiliency for Scala on the JVM

- Using Scala 3 features: context functions, inlines, macros, opaque types, enums, extension methods
- JVM: leveraging virtual threads introduced in Java 21+ (Project Loom)
  - built-in asynchronous runtime
  - structured concurrency

Scala 3機能の活用・Java21+で導入された仮想スレッドの活用(Project Loom)・組み込み非同期ランタイム・構造化並行性

High-level concurrency

```scala
import ox.{par, sleep}
import scala.concurrent.duration.*

def computation1: Int =
  sleep(2.seconds)
  1

def computation2: String =
  sleep(1.second)
  "2"

val result: (Int, String) = par(computation1, computation2)
// (1, "2")
```

oxが実際にどのように動作するのか、具体的なコード例

High-level concurrency

```scala
import ox.{race, sleep}
import scala.concurrent.duration.*

def computation1: Int =
  sleep(2.seconds)
  1

def computation2: Int =
  sleep(1.second)
  2

val result: Int = race(computation1, computation2)
// 2
```

2つの計算を競わせるrace演算子。先に終わった計算結果が返されもう一方は中断。race
が完了するのは両方のスレッドが完了してから

Structured concurrency:
an approach where the lifetime of a thread is determined by the syntactic structure of the code.

構造化並行処理：スレッドの寿命がコードの構文構造によって決定されるアプローチ

Structured concurrency

```
import ox.{fork, sleep, supervised}
import scala.concurrent.duration.*

supervised {                    // ① starts a scope
  val f1 = fork {               // ② can only be called within a scope
    sleep(2.seconds)
    1
  }

  val f2 = fork {
    sleep(1.second)
    2
  }

  (f1.join(), f2.join()) // ③ block until fork done
}
// (1, 2)
```

例えば、構造化並行処理を使用して2つの計算を並行して実行するparメソッドの再実装を紹介

Let it crash

```
import ox.{fork, sleep, supervised}
import scala.concurrent.duration.*

supervised {
  forkUser {              // ① block scope completion
    sleep(1.second)
    println("Hello!")
  }

  fork {
    sleep(500.millis)
    throw new RuntimeException("boom!")
  }
}                         // ② only complete scope once all forks complete
// throws "boom!"
```

いずれかが失敗した場合、スコープ全体も終了しまだ実行中の他のスレッドも中断される。すべてのスレッドが完了した時点でスコープが終了し例外が投げられる

Channels

```
supervised {
  val c = Channel.buffered[String](5)    // ① the 6th send blocks

  fork {                                 // ② daemon fork
    repeatWhile {                        // ③ loop
      val s = readLine()
      if s == "done"
      then { c.done();  false }          // ④ channel & fork completion
      else { c.send(s); true }           // ⑤ send to channel
    }
  }

  repeatWhile {
    c.receiveOrClosed() match            // ⑥ block until data available
      case Error(r) => false
      case Done     => false             // ⑦ complete scope on done/error
      case v        => process(v); true  // ⑧ expensive processing
  }
}
```

スレッド間の通信にはチャンネル。容量5個のバッファ付きチャンネルの例

Channels

```
supervised {
  Source
    .iterate(0)(_ + 1)                      // ① starts a fork, returns a channel
    .transform(
       _.filter(_ % 2 == 0)
        .map(_ + 1)
        .take(10))                          // ② async transform
    .foreach(n => println(n.toString)) // ③ blocking operation
}
```

さっきのコードは低レベルのインターフェース、高レベルAPIも提供

What else?

- Error handling using Eithers, with boundary-break:

```
either { myEither.ok() + anotherEither.ok() }
```

- Retries: `retry(RetryPolicy.backoff(…))(computationR)`
- Select exactly one clause to complete: `select(ch1.receiveClause(), ch2.receiveClause())`
- Local actor pattern
- Direct style utilities: `.discard`, `.tap`, `.pipe`, `uninterruptible`, …
- Resources: `supervised { …; useInScope(createConnection); … }`
- IO: `IO.unsafe { …; inputStream.read(); … }`

他の機能: Eitherによるboundary-break付きエラー処理・再試行・ローカルアクターなど
など

Comparing to functional effects: basics

	Syntax overhead / coloring	Error handling	Stack traces	Control flow methods	Referential transparency
cats-effect / ZIO	💔	🍀	💔	⚠️	🍀
direct / Ox	🍀	⚠️ typed Eithers + untyped exceptions	🍀	🍀	💔

構文的オーバーヘッドでは関数型に勝っている

Comparing to functional effects: concurrency

	High-level "fearless" concurrency	Low-level concurrency	Supervision	Interruptions	Thread locals
cats-effect / ZIO	🍀	🍀	⚠️ one-way - parent->child	🍀	⚠️
direct / Ox	🍀	🍀	🍀 structured concurrency	⚠️ injecting exceptions	🍀

どっちの方法でも高レベルな並行性は書ける

Comparing to functional effects: other

	Lazy / eager	Testing	Maturity/ ecosystem	Resource safety	Developer experience
cats-effect / ZIO	🍀 always lazy	🍀	🍀	🍀	⚠️
direct / Ox	⚠️ manual eager/lazy	💔	⚠️	⚠️ no dedicated type	🍀

遅延評価やリソース管理ではエフェクトシステムに軍配が上がる

To Ox, or not to Ox?

- We gain simpler syntax, lower learning curve, better readability and debugability
- We retain concurrency & supervision
- We partially loose principled errors, interruptions & resources
- We loose referential transparency

シンプルな構文、可読性とデバッグ性の向上。エラーハンドリング、割り込み処理、リソース管理の厳密な実装や参照透過性は一部失われる。並行性と監視機能は維持

- Give Ox a try!
  - https://ox.softwaremill.com
  - 0.2.0 available, Apache2 licensed
  - ⭐ if the project is interesting!

- Give us feedback!
  - https://softwaremill.community/

初版がリリースされたOxを試し、APIが分かりづらかったり安全性に問題があれば
フィードバックをお願いしたい

Solving the hard problems that our clients face using software

SoftwareMillは顧客が直面する困難な問題をソフトウェアで解決するコンサルティング会社

```
supervised {
  fork {
    println("Thank you!")
  }
}
```

@adamwarski / @softwaremill.social / softwaremill.com

SOFTWAREMILL