What Java Developers Can Learn From ZIO

Agenda

- Introduction To ZIO
- Key Design Considerations
- Lessons For Java Developers

ZIO

- ZIO is a library for asynchronous and concurrent programming in Scala
- Gives developers superpowers for writing asynchronous code
- Concurrent programming for mere mortals

Problem

- Open a collection of files in parallel, performing some analysis with the contents of each file
- Never have more than four files open at a time
- If an error is encountered while reading from any file immediately terminate reading from all other files, ensuring that any file handles are properly closed

Issues

- Parallelism
- Resource handling
- Interruption

Solution

```
ZIO
  .foreachParN(4)(names) { name =>
    ZIO.bracket(openFile(name))(closeFile(_).orDie)(parseWeatherData)
  }
  .flatMap(analyzeWeatherData)
```

Parallelism

- The `foreachParN` operator performs the specified workflow in parallel for each element of a collection
- Parallelism will be limited to the specified maximum parallelism
- If any workflow fails all other workflows will be automatically interrupted

Resource Handling

- The `bracket` operator guarantees that if the workflow to acquire a resource succeeds the workflow to close the resource will always be run
- Like `try` / `finally` for asynchronous code
- Guarantee is honored even in the presence of interruption

Interruption

- Any workflow can be interrupted between steps in that workflow
- If any workflow is interrupted finalizers associated with that workflow are guaranteed to be run
- Critical sections of a workflow can be designated as uninterruptible

Problem

- Transfer an actor from being supervised by one supervisor to another
- Concurrent updates must not occur to the set of actors being supervised by either one while the transfer is in process
- Do not block any threads while doing this

Issues

- Concurrent state
- Composing atomicity
- Blocking

Solution

```
def transfer(
    actor: Actor,
    from: Supervisor,
    to: Supervisor
): ZIO[Any, Nothing, Boolean] = {
  val acquire = from.lock.acquireWrite.zip(to.lock.acquireWrite).commit
  val release = from.lock.releaseWrite.zip(to.lock.releaseWrite).commit
  ZIO.bracket(acquire)(_ => release) { _ =>
    ZIO.effectTotal {
      val removed = from.supervised.remove(actor)
      if (removed) to.supervised.add(actor)
      removed
    }
  }
}
```

Software Transactional Memory

- Changes can be composed to create a single transaction that will be performed atomically
- Automatically retry if a concurrent change is made to any of the transactional variables
- Never blocks and only retries when a transactional variable changes so no "busy loops"

Agenda

- Introduction To ZIO
- **Key Design Considerations**
- Lessons For Java Developers

Describe Don't Do

```scala
val sayHello: ZIO[Any, Nothing, Unit] =
  ZIO.effectTotal(println("Hello, World!"))

// Nothing happens
```

- A ZIO effect is a description of a workflow
- Because it is just a a description we can interpret it however we want
- We can build our own runtime on top of the platform

Describing

```
trait ZIO[-R, +E, +A] {
  def effectTotal[A](effect: => A): ZIO[Any, Nothing, A] =
    new ZIO.EffectTotal(() => effect)
}

object ZIO {
  final case class EffectTotal(effect: () => A) extends ZIO[Any, Nothing, A]
}
```

- Description forms a miniature language
- Less than twenty primitive elements
- Can embed arbitrary code from host language

Doing

```
def unsafeRun[R, E, A](zio: ZIO[R, E, A]): A =
  ???
```

- Run a workflow by interpreting it
- Can provide features not supported by host language
- Allows faster release cycle

Fiber Based Concurrency

```
trait ZIO[-R, +E, +A] {
  def fork: ZIO[R, Nothing, Fiber[E, A]]
}
```

- Forking a `Fiber` doesn't actually create a new thread
- Fiber will be run on underlying thread pool until it yields or executes specified number of steps
- Allows non-blocking awaiting and interruption independent of JVM threads

Avoiding Callback Hell

```
for {
  promise <- Promise.make[Nothing, Unit]
  _       <- promise.succeed(()).delay(5.seconds).fork
  _       <- promise.await
} yield ()
```

- This code will never block
- Delaying and waiting implemented in terms of callbacks
- But exposes a very straightforward API for users

Compositional Laws

- To build larger components out of smaller ones need laws about how components will behave
- Laws need to have the right shape so if components follow laws then system will also follow laws
- Not unique to ZIO but spend a lot of time on this, especially how laws fit together in the right way

Safe Resource Usage

```
def bracket[R, E, A, B](
  acquire: ZIO[R, E, A])(
  release: A => ZIO[R, Nothing, Any])(
  use: A => ZIO[R, E, B]): ZIO[R, E, B]
```

- If `acquire` completes execution then `release` is guaranteed to be run after `use` terminates
- This holds regardless of where `bracket` is called
- Also holds regardless of what we do in these effects

Agenda

- Introduction To ZIO
- Key Design Considerations
- **Lessons For Java Developers**

Not Scala Specific

- ZIO is written in Scala
- Takes advantage of Scala specific features to provide best possible user experience
- But ideas behind ZIO and features it provides can be implemented in other languages

Need Higher Level Operators

- Many of the tools we work with like `java.util.concurrent` are very low level
- Fantastic implementations and ZIO is implemented in terms of many of them
- But force us to do too much ourselves on day to day basis

Separate Describing And Doing

- Separating description of what we want to do from how we want to do it is a very powerful technique
- Gives us the ability to see and optimize our description before running it
- Run it in a way that supports the features we want, potentially in multiple different ways

Define Laws That Compose

- We always want to be able to build more complex systems out of simpler ones
- Properties for more complex systems must emerge from properties of simpler ones
- If you need to understand the implementation to know that property holds something has gone wrong

Conclusion

- ZIO gives you superpowers for writing asynchronous code
- Written in Scala but not limited to Scala
- Can apply the same techniques to implement higher level abstractions in Java or other languages

Thank You

- John de Goes for his mentorship and leadership of ZIO community
- ZIO contributors
- ZIO users
- Oli and team for organizing this conference
- All of you for attending today