Using Aspects To Transform Your Code With ZIO Environment

Agenda

- **Aspects In ZIO Test And Caliban**
- The Promise Of Aspect Oriented Programming
- Limitations Of Traditional Approaches
- Using ZIO's Environment Type
- The Future Of Aspect Oriented Programming

ZIO Test

ZIO Test is a next generation testing library for functional Scala:

- Tests as first class values
- Interruptibility
- Resource safety
- Environment type
- Property based testing

Aspects In ZIO Test

ZIO Test has a concept of aspects, called test aspects, that allow modifying how a test is executed:

```
test @@ timeout(1.seconds)
```

Common Test Aspects

There are a wide variety of test aspects:

- **`diagnose`** - do a localized fiber dump if a test times out
- **`jvmOnly`** - only run a test on the JVM
- **`nonFlaky`** - run a test repeatedly to make sure it is stable
- **`tag`** - tag a test for reporting
- **`timed`** - time a test to identify slow tests
- **`timeout`** - time out a test after specified duration

Test Aspects Compose

Test aspects can be composed to modify tests in more complex ways:

```
test @@ jvm(nonFlaky) @@ timeout(60.seconds)
```

Caliban

Caliban is a next generation library GraphQL library for Scala:

- Automatic derivation of schemas from data types
- Query parsing and validation
- Effects handled by ZIO

Aspects In Caliban

Caliban supports a concept of aspects, called wrappers, that allow modifying query parsing, validation, and execution:

```
val api =
  graphQL(???) @@
    maxDepth(50) @@
    timeout(3 seconds) @@
    printSlowQueries(500 millis) @@
    apolloTracing @@
    apolloCaching
```

Agenda

- Aspects In ZIO Test And Caliban
- **The Promise Of Aspect Oriented Programming**
- Limitations Of Traditional Approaches
- Using ZIO's Environment Type
- The Future Of Aspect Oriented Programming

Aspect Oriented Programming

- In any domain there are **cross cutting concerns** that are shared among different parts of our main program logic
- Often these concerns are **tangled** with each part of our main program logic and **scattered** across different parts
- We want to increase the modularity of our programs by **separating** these concerns from our main program logic

Cross Cutting Concerns

Cross cutting concerns are typically related to **how** we do something rather than **what** we are doing:

- What level of **authorization** should this transfer require?
- How should this transfer be **logged**?
- How should this transfer be recorded to our **database**?

Example: Testing

Main program logic is tests, but there are a variety of concerns of **how** we run tests that are distinct from the tests themselves:

- How many times should we run a test?
- What environments should we run the test on?
- What sample size should we use for property based tests?
- What degree of parallelism?
- What timeout to use?

Example: GraphQL

Main program logic is queries, but there are a variety of concerns of **how** we run queries that are distinct from the queries themselves:

- What is the maximum depth of nested queries we should support?
- What is the maximum number of fields we should support?
- What timeout should we use?
- How should we handle slow queries?
- What kind of tracing and caching should we use?

Tangled And Scattered Code

```
testM("foreachPar preserves ordering") {
  val zio = ZIO.foreach(1 to 100) { _ =>
    ZIO.foreachPar(1 to 100)(ZIO.succeed(_)).map(_ == (1 to 100))
  }.map(_.forall(identity))
  assertM(zio)(isTrue)
}
```

- It easy to tangle questions of **how** with our main program logic of **what**
- How many times will we **scatter** code like this across our main program logic?

Separating Concerns

```
testM("foreachPar preserves ordering") {
  assertM(ZIO.foreachPar(1 to 100)(ZIO.succeed(_)))(equalTo(1 to 100))
} @@ nonFlaky
```

- By separating questions of **what** versus questions of **how** we can clean up code like this significantly
- Actual logic of what we are testing much clearer now that it is not tangled with cross cutting concern of repetition
- All logic related to repetition of tests now consolidated in one place

Agenda

- Aspects In ZIO Test And Caliban
- The Promise Of Aspect Oriented Programming
- **Limitations Of Traditional Approaches**
- Using ZIO's Environment Type
- The Future Of Aspect Oriented Programming

An Implementation Problem

- The pain points identified by aspect oriented programming are real
- Aspects can dramatically improve the modularity of our code
- The problem is how aspects have traditionally been implemented

Not Enough Information

```scala
def transfer(from: Account, to: Account, amount: Int): ZIO[Any, TransferError, Unit] =
  ???
```

- As a challenge, try to implement an operator that adds logging to a step in the account transfer process
- We can't do it because the **`transfer`** method is completely opaque
- We can't "reach inside" its implementation to add code for logging each step

Metaprogramming

To get around this, traditional approaches to aspect oriented programming turned to metaprogramming:

- Tools such as AspectJ allow programmers to insert additional code called "advice" into existing source code
- Code can be inserted at "join points"
- Can use "point cuts" to designate which join points advice should be inserted at

Example: AspectJ

```
pointcut set(Account account, int amount):
  call(void Account.set(int))
  && target(account)
  && args(amount);

after(Account account, int amount) returning: set(account, amount) {
  System.out.println("set balance of account " + account + "to" + amount + ".")
}
```

- Point cut finds any invocations of **`set`** method on **`Account`** and captures the account and amount as new variables
- Advice code runs after every invocation matching the point cut and has access to the variables exposed by the point cut

Limitations

While this solves the immediately problem of how to modify the implementation of existing code, it raises several new issues:

- Accessibility
- Understandability
- Robustness

Accessibility

Programming in this style is essentially programming in a new metaprogramming language rather than the base language:

- Need to learn new domain specific language
- Different paradigm as units of analysis are code fragments themselves
- Knowledge gained is not applicable in other areas

Understandability

Can't understand program logic from original source code because additional logic may be inserted:

- Arbitrary logic can be inserted at any point in ordinary control flow
- Like a "GOTO" statement but a "COME FROM" statement
- Whether a point cut matches may be dynamically determined

Robustness

Makes code harder to safely refactor:

- Relies on implementation details such as class and method names that may change
- Requires whole program knowledge of source code and aspects
- No longer able to statically type check if code is dynamically generated

Agenda

- Aspects In ZIO Test And Caliban
- The Promise Of Aspect Oriented Programming
- Limitations Of Traditional Approaches
- **Using ZIO's Environment Type**
- The Future Of Aspect Oriented Programming

ZIO

```scala
trait ZIO[-R, +E, +A]
```

- A blueprint for a concurrent program that requires a set of services **`R`** and will either fail with an **`E`** or succeed with an **`A`**
- Compose programs using operators such as **`flatMap`** for sequential composition or **`zipWithPar`** for parallel composition to build more complex programs to solve business problems

The ZIO Environment Type

- The environment type **`R`** represents the services that a program needs to run
- For example, a program might require access to a logging service or a database service
- To run a **`ZIO`** effect we need to provide it with all the services it needs

Accessing The Environment

In ZIO, we access the environment using the **`environment`** operator, which allows us to access a service so we can do something with it:

```scala
type Database = Has[Database.Service]

object Database {
  trait Service {
    def transfer(from: Account, to: Account, amount: Int): ZIO[Any, TransferError, Unit]
  }
}

val needsDatabase: ZIO[Database, TransferError, Unit] =
  ZIO.environment[Database].flatMap(_.get.transfer(alice, bob, 100))
```

Providing The Environment

To run a ZIO program we need to provide it with all the services it needs, typically using the **`provideLayer`** operator:

```
val liveDatabase: ZLayer[Any, Nothing, Database] =
  ???

val readyToRun: ZIO[Any, TransferError, Unit] =
  needsDatabase.provideLayer(liveDatabase)
```

Deferring Dependencies

The ZIO environment type allows us to work with a service in the environment while deferring providing a concrete implementation:

```scala
// We are describing doing something with a database but don't have a database yet
val needsDatabase: ZIO[Database, Throwable, Unit] =
  ZIO.environment[Database].flatMap(_.get.transfer(alice, bob, 100))

// Now we are providing an actual database implementation
val readyToRun: ZIO[Any, Throwable, Unit] =
  needsDatabase.provideLayer(liveDatabase)
```

Providing More Information

```
def transfer(from: Account, to: Account, amount: Int): ZIO[Database, TransferError, Unit] =
  ???
```

- We now know that the **`transfer`** method depends on a **`Database`** service
- We can adding logging to **`transfer`** by adding it to **`Database`**
- And we know how to transform **`Database`** because it is in the environment

Transforming the Environment

```scala
def log[R <: Database with Logging, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A] =
  ???
```

- We can transform the environment of an effect using the `updateService` operator to **decorate** any service with additional functionality
- This operator logs every successful database transaction
- These are starting to look a lot like aspects!

Decorating Services

```scala
def log[R <: Database with Logging, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A] =
  ZIO.environment[Logging].flatMap { logging =>
    zio.updateService[Database.Service] { database =>
      new Database.Service {
        def transfer(from: Account, to: Account, amount: Int): IO[TransferError, Unit] =
          database
            .transfer(from, to, amount)
            .tap(_ => logging.get.logLine(s"transferred $amount from $from to $to"))
      }
    }
  }
```

Aspects

```scala
trait Aspect[-R, +E] {
  def apply[R1 <: R, E1 >: E, A](zio: ZIO[R1, E1, A]): ZIO[R1, E1, A]
}
```

- Aspects are polymorphic functions from an effect type to the same effect type, potentially constraining the environment or widening the error type
- This captures the idea that aspects transform the **how** but not the **what**

Syntactic Sugar

We can recover the nice syntax we saw for working with aspects as well:

```
implicit final class AspectSyntax[-R, +E, +A](private val zio: ZIO[R, E, A]) {
  def @@[R1 <: R, E1 >: E](aspect: Aspect[R1, E1]): ZIO[R1, E1, A] =
    aspect(zio)
}
```

Implementing Aspects

We can just implement aspects in terms of the existing functions we have written for transforming the environment as long as they are sufficiently polymorphic:

```
val logging: Aspect[Database with Logging, Nothing] =
  new Aspect {
    def apply[R <: Database with Logging, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A] =
      log(zio)
  }
```

Putting It All Together

```
transfer(alice, bob, 100) @@ logging
```

- We have achieved the goals of aspect oriented programming by separating the concern of transferring funds from the concern of logging
- We have done it with plain Scala and the power of functional programming
- Everything is type safe and composable

Agenda

- Aspects In ZIO Test And Caliban
- The Promise Of Aspect Oriented Programming
- Limitations Of Traditional Approaches
- Using ZIO's Environment Type
- **The Future Of Aspect Oriented Programming**

Specialized Aspects

```
trait StreamAspect[-R, +E] {
  def apply[R1 <: R, E1 >: E, O](stream: ZStream[R1, E1, O]): ZStream[R1, E1, O]
}
```

- Aspects can be defined for any type that is "effect like"
- This stream aspect allows modifying a stream
- What data types are you working with where it would be helpful to modify how they are executed in a powerful and composable way?

Polymorphic Aspects

```scala
type StreamAspectPoly = StreamAspect[Any, Nothing]

def chunk(n: Int): StreamAspectPoly =
  new StreamAspectPoly {
    def apply[R, E, O](stream: ZStream[R, E, O]): ZStream[R, E, O] =
      stream.chunkN(n)
  }
```

- Some aspects can be completely polymorphic
- More possibilities to define these for specialized data types that have more "structure"
- But often aspects that are completely polymorphic will not have enough information to do the most interesting things

Environment Type

```scala
trait Authorization

val authorizedOnly: Aspect[Authorization with Database, AuthorizationError] =
  ???

transfer(alice, bob, 100) @@ logging @@ authorizedOnly
```

- An aspect that only executes database transactions when the caller has permission
- Implement operators in your domain in terms of services
- Define aspects to transform those services in ways that are relevant to your domain

Conclusion

- Aspects are tools you can use in your code today
- Separate what you want to do from how you want to do it
- Use services to describe functionality in a modular way
- Use aspects to describe ways of modifying those services
- Excited to see what people build with this!

Thank You

- commercetools for generously sponsoring this event
- John de Goes for his mentorship and feedback on this presentation
- ZIO contributors for their collaboration
- ZIO users for putting your faith in us and your honest feedback
- You for attending this talk