Next Generation Operations With ZIO

Agenda

- **Support for Operational Concerns in ZIO 2.0**
- Why in ZIO?
- Logging
- Metrics

Definition

Operations are how we understand what is happening in our application without changing our source code:

- What happened with this request?
- How quickly are we responding?
- Where did this failure occur?

Practically

Operations are how we diagnose problems in large applications on a timely basis so we can fix them:

- Tools like debug statements, code reviews, and unit tests not sufficient for large applications
- Significant lead time for deploying applications and observing failures
- Need to be able to understand what is happening in running application in real time

Operational Tools

Variety of operational tools:

- Logging - qualitative information about what is happening in our application
- Metrics - quantitative information about what is happening in our application
- Tracing - more detailed qualitative information when an error occurs

Logging

```scala
ZIO.logSpan("parsing") {
  ZIO.acquireReleaseWith(openFile("data.csv")) {
    file => closeFile(file).catchAll(ZIO.logError(_))
  } {
    file => parseFile(file)
  }
}
```

- Define composable log spans
- Log anywhere in your ZIO application
- Plug in any logging backend

Metrics

```scala
val trackHit: ZIOMetric.Counter[Any] =
  ZIOMetric.count("cache-hits")

cache.get(key).flatMap {
  case Some(value) => ZIO.succeed(value) @@ trackHit
  case None        => ???
```

- Use predefined and custom counters, gauges, histograms, summaries, and set counts
- Apply metrics as aspects or call methods directly
- Plug in any metrics backend

Tracing

— Not new but improved

— More than twice as fast

— Improved rendering

Agenda

- Support for Operational Concerns in ZIO 2.0
- **Why in ZIO?**
- Logging
- Metrics

Why Include Operational Support in ZIO?

Three Reasons

1. Easy Onboarding
2. Share Best Practices
3. Communicate More Information

Easy Onboarding

We want to make it as easy as possible to get started using ZIO and support for operations is "table stakes" for real world applications:

- Reduce questions when getting started
- Provide an out of the box solution
- Let you focus on your business problems

Share Best Practices

Create a community in which use of these tools is standard:

- Part of being a ZIO developer is knowing how to use these tools
- We leverage them in a wide variety of open source projects
- We have shared expectations about how they should be used

Communicate More Information

Give ZIO ecosystem libraries a common language for operational concerns:

- Currently support for operations requires additional dependencies which many libraries want to avoid
- Many libraries leave it to users to handle
- Direct support will allow for much higher quality operational information from these libraries

Functional Effects Are Binary

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

trait Runtime[+R] {
  def unsafeRunSync[E, A](zio: ZIO[R, E, A]): Exit[E, A]
}

trait Exit[+E, +A]

final case class Failure[+E](e: E) extends Exit[E, Nothing]
final case class Success[+A](a: A) extends Exit[Nothing, A]
```

The result of running a `ZIO` workflow is always either a success or failure but never both

The World Is Not Binary

```
for {
  _ <- ZIO.fail("uh oh").forkDaemon // forked fiber is never joined
} yield ()
```

- The fiber is forked but never joined so its result is not part of our result
- But the fact that the fiber failed seems potentially important
- Best we could do before was print it to the console or let user specify how to handle it

Compression Destroys Information

Compressing a non-binary world to a binary result destroys information:

- Console rendering may not not even be visible
- Forces arbitrary decisions about what to display
- Either too little or too much for some users
- One off user configuration not scalable

Need Another Channel

Need another channel to communicate this information:

- Able to convey arbitrary metadata
- Exactly what operational tools provide
- Logging is structured qualitative metadata
- Metrics are structured quantitative metadata

Use In ZIO Applications

```
test("zipWithLatest") {
  val s1 = ZStream.iterate(0)(_ + 1).fixed(100.milliseconds)
  val s2 = ZStream.iterate(0)(_ + 1).fixed(70.milliseconds)
  val s3 = s1.zipWithLatest(s2)((_, _))

  for {
    q      <- Queue.unbounded[(Int, Int)]
    _      <- s3.foreach(q.offer).fork
    fiber  <- ZIO.collectAll(ZIO.replicate(4)(q.take)).fork
    _      <- TestClock.adjust(1.second)
    result <- fiber.join
  } yield assert(result)(equalTo(List(0 -> 0, 0 -> 1, 1 -> 1, 1 -> 2)))
}
```

- ZIO Test provides `TestClock` for code involving time
- Works great, but you have to adjust the clock
- Now can log a warning if you don't

Agenda

- Support for Operational Concerns in ZIO 2.0
- Why in ZIO?
- **Logging**
- Metrics

Design Goals

- Simple
- Extensible
- Zero dependency

Strategy

- Front End - describe what you want to log
- Back End - execute your description
- ZIO Runtime - handle everything in between

Simple Logging

```
def log(message: => String): UIO[Unit] =
  new Logged(() => message) // new primitive
```

But what about everything else?

- Log levels
- Log spans
- Structured logging

FiberRef To The Rescue

```
trait FiberRef[A] {
  def locally[R, E, B](value: A)(use: ZIO[R, E, B]): ZIO[R, E, B]
}
```

- Sets the `FiberRef` to the specified value for this computation and then sets it back
- Value is local to this fiber so change not visible to concurrent processes
- Great for propagating information about how we should do something

Log Levels

```
final case class LogLevel(ordinal: Int, label, String, syslog: Int)

val Error: LogLevel = ???
val Info: LogLevel = ???
val Warn: LogLevel = ???

val currentLogLevel: FiberRef[LogLevel] =
  ??? // created by ZIO runtime
```

- Log levels contain a priority, a label, and a severity
- All basic log levels supported
- Can also define your own

Logging with Log Levels

```
def logError(message: => String): UIO[Unit] =
  currentLogLevel.locally(LogLevel.Error)(log(message))
```

- Now when we are executing a `log` we can look at the value of the `FiberRef` and get the current log level
- Can describe the log level separately from the log message

Log Spans

```
final case class LogSpan(label: String, startTime: Long)

val currentLogSpan: FiberRef[List[LogSpan]] =
  ???
```

- We can use the same approach for log spans
- This time the `FiberRef` maintains a "stack" of log spans
- Create a new span by pushing it onto the stack

Logging with Log Spans

```
def logSpan(label: String): ZIOLogSpan =
  new ZIOLogSpan(label)

trait ZIOLogSpan(label: String) {
  def apply[R, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A] =
    currentLogSpan.get.flatMap { stack =>
      currentLogSpan.locally(LogSpan(message) :: stack)(zio)
    }
}
```

— Set a log span for an entire region

— Nest inner log spans inside outer ones

Structured Logging

```
final case class CorrelationId(value: Long)

val currentCorrelationId: FiberRef[CorrelationId] =
  ???
```

- Use the same technique to support structured logging
- Write data either when we log or at higher level
- All information is available to logger implementation

ZLogger

```scala
trait ZLogger[+A] {
  def apply(
    trace: ZTraceElement,
    fiberId: FiberId,
    logLevel: LogLevel,
    message: () => String,
    context: Map[FiberRef[_], AnyRef],
    spans: List[LogSpan]
  ): A
}
```

— Logger knows what was logged and the context

— Can produce value or do something (e.g. write to file)

Loggers Compose

```scala
trait ZLogger[+A] {
  def ++[A1 >: A](that: ZLogger[A1]): ZLogger[A1]
  def filterLogLevel(logLevel: LogLevel): Logger[Option[A]]
  def map[B](f: A => B): Logger[B]
```

- Log to two or more different loggers
- Filter the logs
- Transform the output

ZIO Logging

```scala
object MyApp extends ZIOAppDefault {

  val runtimeConfig: RuntimeConfig =
    RuntimeConfig.default.copy(logger = myLogger)

  def run =
    myApp.withRuntimeConfig(runtimeConfig)
}
```

- Will provide implementations of `ZLogger` for common logging backends
- Higher level logging functionality built on top of this
- Plug in a logger and you are ready to go

Putting It All Together

```
ZIO.logSpan("parsing") {
  ZIO.acquireReleaseWith(openFile("data.csv")) {
    file => closeFile(file).catchAll(ZIO.logError(_))
  } {
    file => parseFile(file)
  }
}
```

Agenda

- Support for Operational Concerns in ZIO 2.0
- Why in ZIO?
- Logging
- **Metrics**

Design Goals

- Simple
- Extensible
- Zero Dependency
- Performance

Strategy

- Front End - describe tracking a metric
- Back End - execute your description
- ZIO Runtime - handle everything in between

ZIOMetric

```
trait ZIOMetric[-A] extends ZIOAspect[Nothing, Any, Nothing, Any, Nothing, A]
```

- Apply a metric to any `ZIO` value to add metrics tracking
- More specific subtypes add additional functionality
- Metric key uniquely defines metric

Counter

```
class Counter[-A](name: String, tags: Chunk[MetricLabel]) extends ZIOMetric[A] {
  def increment(value: Double): UIO[Any]
}
```

— Tracks a cumulative value over time

— For example, number of a type of request received

Gauge

```scala
class Gauge[-A](name: String, tags: Chunk[MetricLabel]) extends ZIOMetric[A] {
  def adjust(value: Double): UIO[Any]
  def set(value: Double): UIO[Any]
}
```

- Tracks a value as of a point in time
- For example, current memory usage

Histogram

```
class Histograms[-A](
  name: String,
  boundaries: Chunk[Double],
  tags: Chunk[MetricLabel]
) extends ZIOMetric[A] {
  def observe(value: Double): UIO[Any]
}
```

- Tracks the relative frequency of numerical values
- For example, distribution of time to serve requests

Summary

```
class Summary[-A](
  name: String,
  maxAge: Duration,
  maxSize: Int,
  error: Double,
  quantiles: Chunk[Double],
  tags: Chunk[MetricLabel]
) extends ZIOMetric[A]
```

— Tracks percentile of a value over a time window

— For example, 99th percentile of response time for SLA

Set Count

```
class SetCount[-A](name: String, setTag: String, tags: Chunk[MetricLabel])
  extends ZIOMetric[A]
```

— Tracks number of occurrences of each value

— For example, frequency of different error types

Predefined Metrics

- Wide variety of metrics for common use cases
- JVM metrics just added
- Thanks to Daniel Vigovsky (@vigoo)

Custom Metrics

```
def countErrors(label: String, tags: Chunk[MetricLabel]): ZIOMetric.Counter[Any] =
  new ZIOMetric.Counter(label, tags) {
    def apply[R, E, A](zio: ZIO[R, E, A]): ZIO[R, E, A] =
      zio.tapError(_ => increment)
  }
```

- Easily define new metrics of your own
- Extend an existing metric type
- Specify what it means to apply your metric to a ZIO value

Metric Clients

```
object MetricClient {
  def unsafeInstallListener(listener: MetricListener): Unit
  def unsafeSnapshot: Map[MetricKey, MetricState]
}
```

- Need to support "push based" and "pull based" clients
- Maintain metric state internally
- Pull based clients can ask for a snapshot on demand
- Push based clients can subscribe to real time updates

~~ZIO ZMX~~ZIO Metrics

Focused on providing implementations of metrics clients:

- Prometheus
- StatsD
- New Relic (forthcoming)

Developer Mode

- Custom Laminar based metrics server
- Visualize metrics without any backend
- Thanks to Andreas Gies (@atooni)

Conclusion

- ZIO Users - will have an even better out of the box experience with ZIO
- ZIO Contributors - need to take advantage of these new tools
- ZIO Users and Contributors - opportunity to develop best practices around operations

Thank You

- Capital One - for being a pleasure to work with
- John de Goes - for his leadership
- Sandra Wolf - for organizing this fantastic event
- ZIO Contributors and Users - for making this possible
- You - for taking your time to attend today