Concurrency in Cats Effect 3

Cats Effect 3 is just around the corner! The library has seen several major changes between 2.0 and 3.0, so in an effort to highlight those changes, we will be releasing a series of blog posts covering a range of topics. If you would like to see a blog post about a particular subject, don't hesitate to reach out! You can try out early Cats Effect 3 releases here.

Introduction

In this post, we offer a broad overview of the concurrency model that serves as the foundation for Cats Effect and its abstractions. We discuss what fibers are and how to interact with them. We also talk about how to leverage the concurrency model to build powerful concurrent state machines. Example programs are written in terms of `cats.effect.IO`.

Before going any further, let's briefly review what concurrency is, how it's useful, and why it's tedious to work with.

Concurrency

Concurrent programming is about designing programs in which multiple logical threads of control are executing at the same time. There is a bit to unpack in this definition: what are logical threads and what does it mean for them to execute "at the same time?"

A logical thread is merely a description of a sequence of discrete actions. An action is one of the most primitive operations that can be expressed in the host language.

In programs that are written in traditional high-level languages like Java or C++, these logical threads are typically represented by native threads that are managed by the operating system. The actions that comprise these threads are native processor instructions or VM bytecode instructions. In programs that are written with Cats Effect, these logical threads are represented by fibers, and the actions that comprise them are `IO` values.

What does it mean for logical threads to execute at the same time? Because the actions of a logical thread are discrete, the actions of many logical threads can be interleaved into one or more streams of actions. This interleaving is largely influenced by external factors such as scheduler preemption and I/O operations, so it is usually nondeterministic. More specifically, in the absence of synchronization, there is no guarantee that the actions of distinct logical threads occur in some particular order.

Another common perspective is that concurrency generates a partial order among all the actions of all the logical threads in a program. Actions *within* a logical thread are ordered consistently with program order. Actions *between* multiple logical threads are not ordered in the absence of synchronization.

Concurrency is useful

Concurrency is a tool for designing high-performance applications that are responsive and resilient. These applications typically involve multiple interactions or tasks that must happen at the same time. For example, a computer game needs to listen for keyboard input, play sound effects, and run game loop logic, all while rendering graphics to the screen. A multiplayer game must also communicate with a network server to exchange game state information.

Traditional sequential programming models quickly become inadequate when applied to building these kinds of responsive programs. The computer game program must perform all of the tasks described above at the same time; it wouldn’t be a very good game if we couldn’t respond to player input while playing a sound! Concurrency enables us to structure our program so that each interaction is confined to its own logical thread(s), all of which execute at the same time.

Concurrency complements modular program design. Interactions like graphics rendering and network communication are largely unrelated; it would be tedious and messy to design a program that constantly switches between graphics and audio I/O operations. Instead of having one complex thread that performs every interaction, we can confine each interaction to its own logical thread. This allows us to think about and code each interaction independently of each other while having the assurance that they occur at the same time. The modularity that concurrency affords makes for programs that are much easier to understand, maintain, and evolve.

Concurrency is hard

Concurrency is notoriously cumbersome to work with. This is no surprise to any developer that has worked with threads and locks in any mainstream programming language. Once we start dealing with concurrency, we have to deal with a slew concerns: deadlocks, starvation, race conditions, thread leaks, thread blocking and so on. A bug in any one of these concerns can have devastating consequences in terms of correctness and performance.

The most popular method for achieving concurrency in Scala is with `Future`, which enables the evaluation of asynchronous operations. However, it is plainly insufficient for those of us who practice strict functional programming. Furthermore, `Future` doesn't expose first-class features like cancellation, finalization, or synchronization that are crucial for building safe concurrent applications.

Akka actors are another common way to achieve concurrency in Scala. For similar reasons as `Future`, they are also insufficient in strict functional programming, however, one could certainly build a pure actor library on top of the Cats Effect's concurrency model.

Cats Effect Concurrency

Cats Effect takes the perspective that concurrency is a necessary technique for building useful applications, but existing tools for achieving concurrency in Scala are tedious to work with. A goal of Cats Effect is then to provide library authors and application developers a concurrency model that is safe and simple to work with.

Most users should never directly interact with the concurrency model of Cats Effect because it is a low-level API. This is intentional: low-level concurrency constructs are inherently unsafe and effectful, so forcing users to touch it would only be a burden for them. Cats Effect and other libraries provide higher-level and safer abstractions that users can exploit to achieve concurrency without having to understand what's happening under the hood. For example, http4s achieves concurrency on requests by spawning a fiber for every request it receives; users need only specify the request handling code that is eventually run inside the fiber.

That being said, it can be tremendously helpful to learn about Cats Effect's internals and concurrency model to understand how our applications behave and how to tune them, so let's jump into it.

Fibers

Cats Effect chooses fibers as the foundation for its concurrency model. The main benefit of fibers is that they are conceptually similar to native threads: both are types of logical threads that describe a sequence of computations. Unlike native threads, fibers are not associated with any system resources. They coexist and are scheduled completely within the userspace process, independently of the operating system. This mode of concurrency reaps major benefits for users and performance:

1. Fibers are incredibly cheap in terms of memory overhead so we can create hundreds of thousands of them without thrashing the process. This means that we also don't need to pool

fibers; we just create a new one whenever we need it.

2. Context switching between fibers is orders of magnitude faster than context switching between threads.
3. Blocking a fiber doesn't necessarily block a native thread; this is called semantic blocking. This is particularly true for nonblocking I/O and other asynchronous operations.

Concretely, a fiber is a logical thread that encapsulates the execution of an `IO[A]` program, which is a sequence of `IO` effects (actions) that are bound together via `flatMap`. Fibers are logical threads, so they can run concurrently.

The active execution of some fiber of an effect `IO[A]` is represented by the type `FiberIO[A]`. The execution of a `FiberIO[A]` terminates with one of three possible outcomes, which are encoded by the datatype `OutcomeIO[A]`:

1. `Succeeded`: indicates success with a value of type `A`
2. `Errored`: indicates failure with a value of type `Throwable`
3. `Canceled`: indicates abnormal termination via cancellation

Additionally, a fiber may never produce an outcome, in which case it is said to be nonterminating.

Cats Effect exposes several functions for interacting with fibers directly. We'll explore parts of this API in the following sections. Again, fibers are considered to be an unsafe and low-level feature and must be given more caution than we offer in the examples. Users are encouraged to use the higher-level concurrency constructs that Cats Effect provides.

Starting and joining fibers

The most basic action of concurrency in Cats Effect is to start or spawn a new fiber. This requests to the scheduler to begin the concurrent execution of a program `IO[A]` inside a new logical thread. The actions of the current fiber and the spawned fiber are interleaved in a nondeterministic fashion.

After a fiber is `start`ed, it can be `join`ed which semantically blocks the joiner until the joinee has terminated, after which `join` will return the outcome of the joinee. Let's take a look at an example.

Let's take a look at an example that demonstrates spawning and joining of fibers, as well as the nondeterministic interleaving of their executions.

```
import cats.effect.{IO, IOApp}
import cats.syntax.all._

object ExampleOne extends IOApp.Simple {
  def repeat(letter: String): IO[Unit] =
    IO.print(letter).replicateA(100).void
  override def run: IO[Unit] =
    for {
      fa <- (repeat("A") *> repeat("B")).as("foo!").start
      fb <- (repeat("C") *> repeat("D")).as("bar!").start
      // joinWithNever is a variant of join that asserts
      // the fiber has an outcome of Succeeded and returns the
      // associated value.
      ra <- fa.joinWithNever
      rb <- fb.joinWithNever
      _ <- IO.println(s"\ndone: a says: $ra, b says: $rb")
    } yield ()
}
```

In this program, the main fiber spawns two fibers, one which prints `A` 100 times and `B` 100 times, and another which prints `C` 100 times and `D` 100 times. It then joins on both fibers, awaiting their termination. Here is one possible output from an execution of the program:

```
AAAAAAAAAAAAAAAAAAAAAAACACACACACACACACACACACACACACACACACACACACACACAACACACACACACAC
done: a says: foo!, b says: bar!
```

We can observe that there is no consistent ordering between the effects of separate fibers; it is completely nondeterministic! However, the effects *within* a given fiber are always sequentially consistent, as dictated by program order; `A` is never printed after `B`, and `C` is never printed after `D`. This is how `join` imposes an ordering on the execution of the fibers: the main fiber will only ever print the `done:` message after the two spawned fibers have completed.

It is recommended to use the safer `IO.background` instead of `IO.start` for spawning fibers.

Canceling fibers

A fiber can be canceled after its execution begins with the `FiberIO#cancel` function. This semantically blocks the current fiber until the target fiber has finalized and terminated, and then returns. Let's take a look at an example.

```
import cats.effect.{IO, IOApp}
import cats.syntax.all._
import scala.concurrent.duration._

object ExampleTwo extends IOApp.Simple {
  override def run: IO[Unit] =
    for {
      fiber <- IO.println("hello!").foreverM.start
      _ <- IO.sleep(5.seconds)
```

```
        _ <- fiber.cancel
      } yield ()
  }
```

In this program, the main fiber spawns a second fiber that continuously prints `hello!` . After 5 seconds, the main fiber cancels the second fiber and then the program exits.

Cats Effect's concurrency model and cancellation model interact very closely with each other, however, the latter is out of scope for this post. It will be discussed in detail in a future post, but in the meantime, visit the Scaladoc pages for `MonadCancel` and `GenSpawn` .

Racing fibers

Cats Effect exposes several utility functions for racing `IO` actions (or their fibers) against eachother. Let's take a look at some of them:

```
object IO {
  def racePair[A, B](left: IO[A], right: IO[B]): IO[Either[(OutcomeIO[A], FiberIO
  // higher-level functions
  def race[A, B](left: IO[A], right: IO[B]): IO[Either[A, B]]
  def both[A, B](left: IO[A], right: IO[B]): IO[(A, B)]
}
```

`racePair` races two fiber and returns the outcome of the winner along with a `FiberIO` handle to the loser. `race` races two fibers and returns the successful outcome of the winner after canceling the loser. `both` races two fibers and returns the successful outcome of both (in other words, it runs both fibers concurrently and waits for both of them to complete).

`racePair` seems a bit hairy to work with, so let's try an example out with `race` :

```
import cats.effect.{IO, IOApp}
import cats.syntax.all._

object ExampleThree extends IOApp.Simple {
  def factorial(n: Long): Long =
    if (n == 0) 1 else n * factorial(n - 1)

  override def run: IO[Unit] =
    for {
      res <- IO.race(IO(factorial(20)), IO(factorial(20)))
      _ <- res.fold(
        a => IO.println(s"Left hand side won: $a"),
        b => IO.println(s"Right hand side won: $b")
      )
    } yield ()
}
```

In this program, we're racing two fibers, both of which calculate the 20th factorial. Running this program with many iterations demonstrates that either fiber can win.

Communication

We have seen how fibers can directly communicate with each other via `start`, `join`, and `cancel`. These mechanisms enable bidirectional communication, but only at the beginning and end of a fiber's lifetime. It's natural to ask if there are other ways in which fibers can communicate, particularly during their lifetime.

Shared memory is an alternative means by which fibers can indirectly communicate and synchronize with each other. Cats Effect exposes two primitive concurrent data structures that leverage shared memory: `Ref` and `Deferred`.

`Ref`

`Ref` is a concurrent data structure that represents a mutable variable. It is used to hold state that can be safely accessed and modified by many contending fibers. Let's take a look at its basic API:

```
trait Ref[A] {
  def get: IO[A]
  def set(a: A): IO[Unit]
  def update(f: A => A): IO[Unit]
}
```

`get` reads and returns the current value of the `Ref`. `set` sets the current value of the `Ref`. `update` atomically reads and sets the current value of the `Ref`. Let's take a look at an example.

```
import cats.effect.{IO, IOApp}
import cats.syntax.all._

object ExampleFour extends IOApp.Simple {
  override def run: IO[Unit] =
    for {
      state <- IO.ref(0)
      fibers <- state.update(_ + 1).start.replicateA(100)
      _ <- fibers.traverse(_.join).void
      value <- state.get
      _ <- IO.println(s"The final value is: $value")
    } yield ()
  }
```

In this program, the main fiber starts 100 fibers, each of which attempts to concurrently update the state by atomically incrementing its value. Next, the main fiber joins on each spawned fiber one after the other, waiting for their collective completion. Finally, after the spawned fibers are complete, the main fiber retrieves the final value of the state. The program should produce the following output:

```
The final value is: 100
```

Deferred

`Deferred` is a concurrent data structure that represents a condition variable. It is used to semantically block fibers until some arbitrary condition has been fulfilled. Let's take a look at its basic API:

```
trait Deferred[A] {
  def complete(a: A): IO[Unit]
  def get: IO[A]
}
```

`get` blocks all calling fibers until the `Deferred` has been completed with a value, after which it will return that value. `complete` completes the `Deferred`, unblocking all waiters. A `Deferred` can not be completed more than once. Let's take a look at an example.

```
import cats.effect.{IO, IOApp}
import cats.effect.kernel.Deferred
import cats.syntax.all._
import scala.concurrent.duration._

object ExampleFive extends IOApp.Simple {
  def countdown(n: Int, pause: Int, waiter: Deferred[IO, Unit]): IO[Unit] =
    IO.println(n) *> IO.defer {
      if (n == 0) IO.unit
      else if (n == pause) IO.println("paused...") *> waiter.get *> countdown(n -
      else countdown(n - 1, pause, waiter)
    }

  override def run: IO[Unit] =
    for {
      waiter <- IO.deferred[Unit]
      f <- countdown(10, 5, waiter).start
```

```
        _ <- IO.sleep(5.seconds)
        _ <- waiter.complete(())
        _ <- f.join
        _ <- IO.println("blast off!")
      } yield ()
  }
```

In this program, the main fiber spawns a fiber that initiates a countdown. When the countdown reaches 5, it waits on a `Deferred` which is completed 5 seconds later by the main fiber. The main fiber then waits for the countdown to complete before exiting. The program should produce the following output:

```
10
9
8
7
6
5
paused...
4
3
2
1
0
blast off!
```

Building a concurrent state machine

`Ref` and `Deferred` are often composed together to build more powerful and more complex concurrent data structures. Most of the concurrent data types in the `std` module in Cats Effect are implemented in terms of `Ref` and/or `Deferred`. Example include `Semaphore`, `Queue`, and `Hotswap`.

In our next example, we create simple concurrent data structure called `Latch` that is blocks a waiter until a certain number of internal latches have been released. Here is the interface for `Latch`:

```
trait Latch {
  def release: IO[Unit]
  def await: IO[Unit]
}
```

Next, we need to define the state machine for `Latch`. We can be in two possible states. The first state reflects that the `Latch` is still active and is waiting for more releases. We need to track how many latches are still remaining, as well as a `Deferred` that is used to block new waiters. The second state reflects that the `Latch` has been completely released and will no longer block waiters.

```
sealed trait State
final case class Awaiting(latches: Int, waiter: Deferred[IO, Unit]) extends State
case object Done extends State
```

We can implement the `Latch` interface now. We create a `Ref` that holds our state machine, and then a `Deferred` that block waiters. The initial state of a `Latch` is the `Awaiting` state with a user-specified number of latches remaining.

The `release` method atomically modifies the state based on its current value: if the current state is `Awaiting` and there is more than one latch remaining, then subtract by one, but if there is only one latch left, then transition to the `Done` state and unblock all the waiters. If the current state is `Done`, then do nothing.

The `await` method inspects the current state; if it is `Done`, then allow the current fiber to pass through, otherwise, block the current fiber with the `waiter`.

```
object Latch {
  def apply(latches: Int): IO[Latch] =
    for {
      waiter <- IO.deferred[Unit]
      state <- IO.ref[State](Awaiting(latches, waiter))
    } yield new Latch {
      override def release: IO[Unit] =
        state.modify {
          case Awaiting(n, waiter) =>
            if (n > 1)
              (Awaiting(n - 1, waiter), IO.unit)
            else
              (Done, waiter.complete(()))
          case Done => (Done, IO.unit)
        }.flatten.void
      override def await: IO[Unit] =
        state.get.flatMap {
          case Done => IO.unit
          case Awaiting(_, waiter) => waiter.get
        }
    }
}
```

Finally, we can use our new concurrent data type in a runnable example:

```
object ExampleSix extends IOApp.Simple {
  override def run: IO[Unit] =
    for {
      latch <- Latch(10)
      _ <- (1 to 10).toList.traverse { idx =>
        (IO.println(s"$idx counting down") *> latch.release).start
      }
      _ <- latch.await
      _ <- IO.println("Got past the latch")
    } yield ()
  }
```

This program creates a `Latch` with 10 internal latches and spawns 10 fibers, each of which releases one internal latch. The main fiber awaits against the `Latch`. Once all 10 fibers have released a latch, the main fiber is unblocked and can proceed. The output of the program should resemble the following:

```
1 counting down
3 counting down
6 counting down
2 counting down
8 counting down
9 counting down
10 counting down
4 counting down
5 counting down
7 counting down
Got past the latch
```

Notice how the latch serves as a form of synchronization that influences the ordering of effects among the fibers; the main fiber will never proceed until after the `Latch` is completely released.

Scheduling

We've talked about fibers as a conceptual model with which Cats Effect implements concurrency. One aspect of this that we haven't addressed is: how do fibers actually execute? Our Scala applications ultimately run on the JVM (or a JavaScript runtime if we're using Scala.js), so fibers must ultimately be mapped to native threads in order for them to actually run. The scheduler is responsible for determining how this mapping takes place.

On the JVM, Cats Effect uses an M:N scheduling model to map fibers to threads. In practice, this means that a large number of fibers is multiplexed onto much smaller pools of native threads. A typical application will reserve a fixed-size pool for CPU-bound tasks, an unbounded pool for blocking tasks, and several event handler pools for asynchronous tasks. Throughout its lifetime, a fiber will migrate between these pools to accomplish its various tasks. In JavaScript runtimes, a M:1 scheduling model is employed, where all active fibers are scheduled onto a single native thread for execution.

Another aspect of scheduling is how context switching of fibers takes place, which has many implications around the fairness and throughput properties of a concurrent application. Like many other lightweight thread runtimes, Cats Effect's default scheduler exhibits cooperative multitasking in which fibers can explicitly "yield" back to the scheduler, allowing a waiting fiber to run. The yielded fiber will be scheduled to resume execution at some later time. This is achieved with the `IO.cede` function.

Cats Effect's default scheduler also supports autoyielding which is a form of preemptive multitasking. As we mentioned earlier, logical threads are composed of a possibly infinite sequence of actions. Autoyielding "preempts" or forcibly yields a fiber after it runs a certain number of actions, enabling waiting fibers to proceed. This is particularly important on runtimes that run with 1 or 2 native threads; a fiber that runs forever but never `cede` s will starve other fibers of compute time.

Parallelism

One aspect of multithreaded programming that we have neglected to mention so far is parallelism. In theory, parallelism is completely independent of concurrency; parallelism is about simultaneous execution whereas concurrency is about interleaved execution.

Parallelism is typically achieved by exploiting multiple CPU cores or even multiple, independent machines to run a set of tasks much faster than they would run on a single CPU or machine. Parallelism is also not necessarily nondeterministic; we can run a set of deterministic tasks in parallel multiple times and expect to get back the same result every time. For our purposes, parallelism can be paired with concurrency to speed up the execution of many logical threads. This is exactly what JVMs already do: multiple native threads run simultaneously, resulting in higher throughput of tasks.

An obscure but crucial point here is that concurrency can be achieved without parallelism; this is called single-threaded concurrency. We've already seen an example of this: JavaScript runtimes run on a single compute thread, so the execution of all fibers must take place on that thread as well!

Exercises

1. Why is the low-level fiber API designated as unsafe? Hint: consider how the fiber API interacts with cancellation.
2. Implement `timeout` in terms of `IO.race`. `timeout` runs some action for up to a specified duration, after which it throws an errors.

```
def timeout[A](io: IO[A], duration: FiniteDuration): IO[A]
```

3. Implement `parTraverse` in terms of `IO.both`. `parTraverse` is the same as `traverse` except all `IO[B]` are run in parallel.

```scala
def parTraverse[A](as: List[A])(f: A => IO[B]): IO[List[B]]
```

4. Implement `Semaphore` in terms of `Ref` and `Deferred`.

```scala
trait Semaphore {
  def acquire: IO[Unit]
  def release: IO[Unit]
}
object Semaphore {
  def apply(permits: Int): IO[Semaphore]
}
```

5. Implement `Queue` in terms of `Ref` and `Deferred`.

```scala
trait Queue[A] {
  def put(a: A): IO[Unit]
  def tryPut(a: A): IO[Boolean]
  def take: IO[A]
  def tryTake: IO[Option[A]]
  def peek: IO[Option[A]]
}
object Queue {
  def apply[A](length: Int): IO[Queue[A]]
}
```

6. `Stateful` is a typeclass in Cats MTL that characterizes a monad's ability to access and manipulate state. This is typically used in monad transformer stacks in conjunction with the `StateT` transformer. Is it possible to create a `Stateful` instance given a `Ref`?

Licensing

by Raas Ahsan on Oct 30, 2020

Follow @RaasAhsan Follow @RaasAhsan

Back to blog

Projects Events Blog About